Node.js based Document Store for Web Crawling

CS 298 Project Report

Presented to

Prof. Chris Pollett

Prof. Katerina Potika

Prof. Ben Reed

Department of Computer Science

San Jose State University

by

David Bui

Fall 2021

Abstract

WARC files are central to internet preservation projects. They contain the raw resources of web crawled data and can be used to create windows into the past of web pages at the time they were accessed. Yet there are few tools that manipulate WARC files outside of basic parsing. The creation of our tool WARC-KIT gives users in the Node.js JavaScript environment, a tool kit to interact with and manipulate WARC files.

Included with WARC-KIT is a WARC parsing tool known as WARCFilter that can be used standalone tool to parse, filter, and create new WARC files. WARCFilter can also, create CDX index files on the WARC files, parse existing CDX files, or even generate webgraph datasets for graph analysis algorithms. Aside from WARCFilter, WARC-KIT includes a custom on disk database system implemented with an underlying Linear Hash Table data structure. The database system is the first of its kind as a JavaScript only on disk document store. The overall main application of WARC-KIT is that it allows users to create custom indices upon collections of WARC files. After creating an index on a WARC collections, users are then query their collection using the GraphQL query language to retrieve desired WARC records.

Experiments with WARCFilter on a WARC dataset composed of 238,000 WARC records demonstrates that utilizing CDX index files speeds WARC record filtering around ten to twenty times faster than raw WARC parsing. Database timing tests with the JavaScript Linear Hash Table database system displayed twice as fast insertion and retrieval operations than a similar Rust implemented Linear Hash Table database. Experiments with the overall WARC-KIT application on the same 238,000 WARC record dataset exhibited consistent query times for different complex queries.

Contents

List of Figures

List of Tables

1. Introduction

Web crawling is the act of programmatically browsing the internet to either index websites for search engines or archive internet resources for both research and preservation purposes. The Internet Archive is one such organization dedicated to preservation of the World Wide Web [27]. Through their open source web crawler Heritix and their archive viewer the Wayback Machine, the Internet Archive stands as the organizational standard to strive towards in internet archival projects. In order to store the data obtained from their crawls, the Internet Archive created the WARC file format as a standardized way to store crawled data. WARC files contain a series a records with each record corresponding to either the original web page, an image, or a video retrieved from a crawled URL on the internet. The Internet Archive and other organizations like it usually provide an interface allowing average users to access their web crawls to view their WARC files. The original data while preserved and viewable through one of the many WARC viewer tools, can only be accessed if the user remembers the original URL. This is a inconvenient for most users as exact URLs are usually not human readable let alone rememberable. Thankfully if one wants to create one's own WARC file collections to save one's favorite web pages, there are many free open source web crawlers that can save specified websites into personal WARC file collections like the Internet Archive's Heritix. Tools to process and analyze large amounts of WARC files also exist, although they most rely on existing online WARC file resources such as Common Crawl [25]. Unfortunately, there are only a few dedicated tools for users to maintain, and search their own WARC collections. Many of those tools are also written only in either Java or Python leaving programmers in other languages fending for themselves.

One such tool that is written in another programming language that allows users to maintain and index their own WARC collections is Yioop, an open source PHP search engine [34]. Having tools like Yioop in other programming languages would allow more extensibility in the processing of WARC files. For example, JavaScript with the advent of Node.js has a large open source community of useful modules such as Express.js for server side web programming. Yet, there are few JavaScript modules to work with WARC files. To solve this gap in WARC processing and indexing tools for users, we created a JavaScript tool that has these WARC manipulation features named WARC-KIT. WARC-KIT is a tool composed of a combination of a newly created WARC processing tool along with a custom made database system in Node.js that allows users to filter, query, and index their own WARC file collections.

The rest of the report is organized as follows. We discuss the the format of WARC files, their CDX index files, and other derivatives of WARC files. Subsequently, we discuss dynamic hashing techniques and why we chose to develop the tool in Node.js. Next we discuss the creation and design decisions of our toolkit to work with WARC files and their derivatives from existing WARC collections. Then we discuss the creation of the database system, its underlying dynamic hashing implementation, and document style storage of rows. After we discuss the overall design of WARC-KIT starting with the extraction of a table format using the WARC parsing tool, insertion of the generated indices into the database system, and the creation of a GraphQL server to allow GraphQL queries of our WARC files. Thereafter, we conduct experiments on WARC-KIT, discuss its performance in various scenarios, and conduct comparisons to other similar systems. Finally we conclude this paper with a discussion on improvements and future ideas.

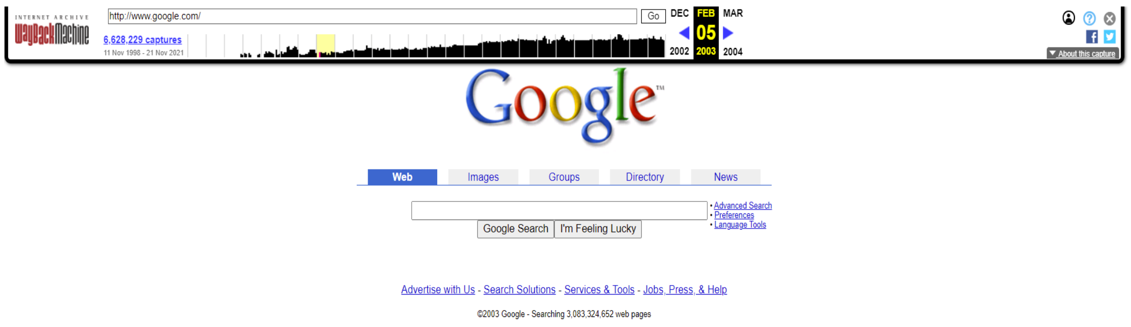


Figure 1.1: A snapshot of Google's homepage in 2003 viewed through the Internet Archive's Wayback Machine.

2. Preliminaries

Before we discuss the design and implementation of the the tools that compose WARC-KIT we will discuss the WARC file format plus it's derivatives, dynamic hashing concepts, and the JavaScript modules that are integrated into WARC-KIT.

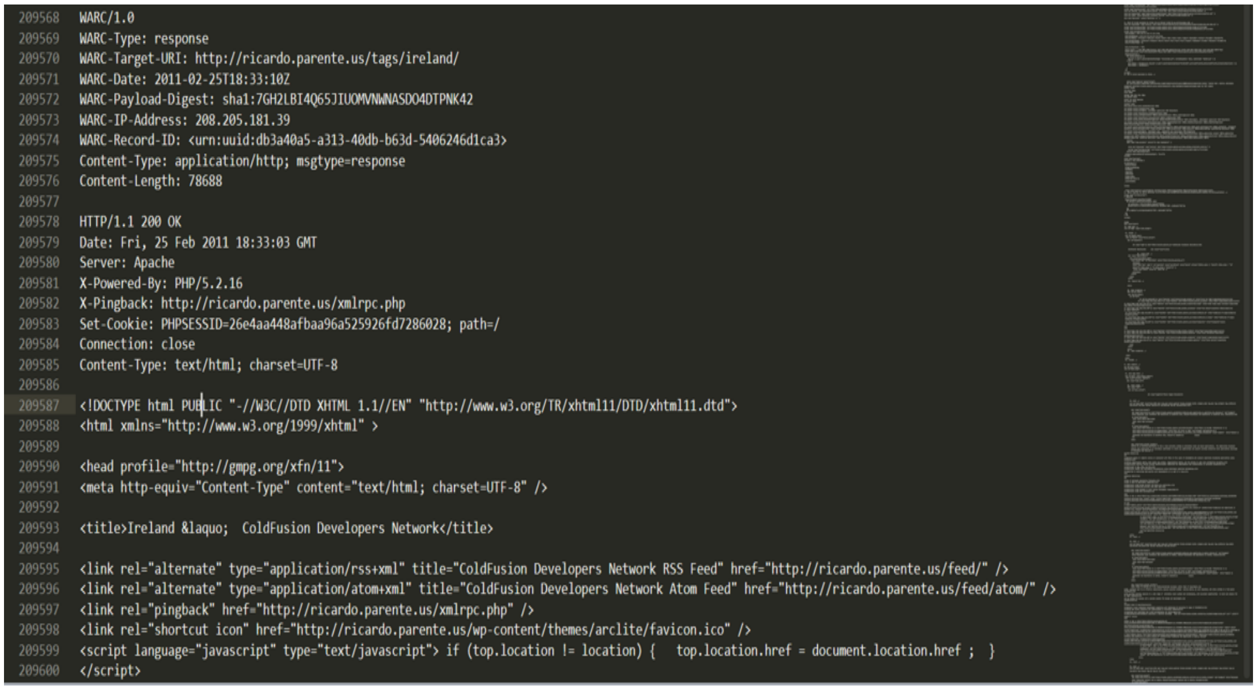
2.1 Warc File Format

Web ARChive or WARC files are an aggegragate file format created for the purpose of archiving multiple different types of digital resources from web pages [32]. First defined in 2008, the WARC file format was created by the Internet Archive as a successor to their original ARChive or ARC file format. Enhancements from ARC to WARC include better assigned metadata definitions, easier duplicate detection, date format transformations, and more. WARC files are given and identified by the *.warc* extension. They are almost always gzipped compressed as well due to their large file sizes, which produces the commonly seen *.warc.gz* extension when a user encounters a WARC file.

WARC file collections are currently only used to store web crawls, though they may become more important in the future to the average internet user. Currently for most internet users, the internet is just a series of web pages they access to do various activities like watch their favorite online videos, view funny images, or post personal status updates on social media. Unfortunately for the average user many of their favorite web pages may not be there in 10 or 20 years. Although there's a popular saying that once something's on the internet its there forever, studies have shown that saying is simply not true. In a recent study [8] by the New York Times, researches found that in just New York Times articles 25% of links going to other websites were "rotted" meaning the web pages were unreachable. Furthermore, they found that the older the link was the higher chance the link was rotted. Around 6% of links from 2018, 43% of links from 2008, and 72% of links from 1998 were found to be unreachable. This is a phenomenon known as link rot where over time web pages on the internet become unreachable because the original URL is no longer valid due to website domain name changes, removal of the original web page, URL syntax changes etc. In another study on link rot in 2012, the Chesapeake Digital Preservation Group found in a dataset of 800 web pages from 2007 to 2012 that 26% of web pages no longer worked [20]. The irony of this study is that the URL linking to the original report has also rotted. This means a report on how web pages can no longer be accessed is itself no longer accessible. Though there is no way to directly stop link rot, there is a way to preserve internet resources even when the original URL is no longer reachable and that is with WARC files. Many Internet preservation organizations have large online WARC file collections that are made available for free to users. Users who have lost their favorite web page may be able to find a copy in one of these many WARC collections.

Recent work to expand user access to cloud hosted WARC collections have found some success. The Archives Unleashed Project has been developing their website to provide a non-programmer friendly online interface to a variety of cloud hosted WARC collections like the Internet Archive and Common Crawl [22, 23]. Although the Archives Unleashed project aims to be a fully operational sometime in the near future, the project acknowledges that their ongoing struggle to bridge the knowledge gap for non-programmatical users is their most challenging task. Further studies by the Archive unleashed project have shown the feasibility of developing personal cloud based tools for accessing WARC file collections. They found that the cost of analyzing cloud hosted WARC collections costs just around 7 USD per TB [4]. Studies have also been conducted to mitigate the computational cost of processing online WARC collections such as constructing a WARC processing framework using Apache Hive and SparkSQL [29]. While optimizing access to current cloud hosted WARC collections is important, there is still a lack of programmatic variety in local tools for users who maintain their own WARC collections.

Pivoting back to the WARC file format, a WARC file consists of the concatenation one or more WARC records that are new line separated. A WARC record can be broken down in two distinct parts a WARC header and a WARC content block. As shown in Figure 2.1 a WARC record header will always start with the WARC format type, the current format is WARC 1.0, along with various header fields. The various WARC header fields can differ depending on the type of WARC record. Mandatory WARC Header fields include the WARC-Type, WARC-Date, WARC-Record-ID and Content Length fields. The WARC-Type field identifies the type of WARC record, the WARC-Date field holds the date the record was created, the WARC-Record-ID contains a uniquely generated record ID of a record and the Content-Length field holds the length of the WARC content block in number of bytes. Other WARC header record fields are optional and can be included at the discretion of the WARC file creator. Following a WARC record header will always be a new line then the WARC record content block. What is contained in the content block depends on the WARC record type but should always be the raw web content retrieved from a web crawl free of any modifications. These can be things such as the HTML response of a web page, raw image data, or raw video data.

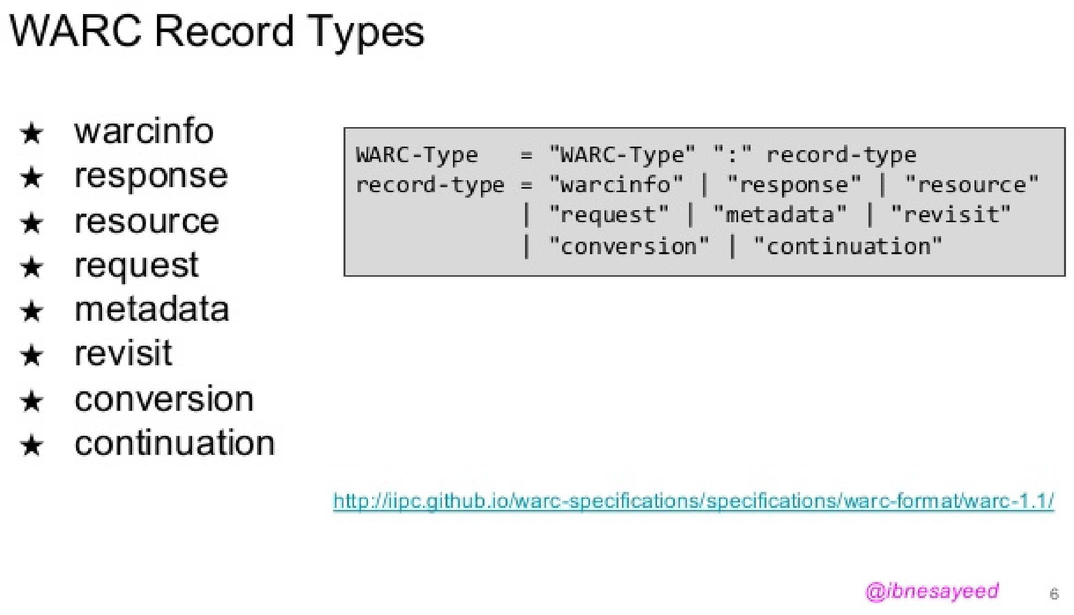


```
WARC/1.0
WARC-Type: response
WARC-Target-URI: http://ricardo.parente.us/tags/ireland/
WARC-Date: 2011-02-25T18:33:10Z
WARC-Payload-Digest: sha1:7GH2LBI4Q65JIUOMVNWNASDO4DTPNK42
WARC-IP-Address: 208.205.181.39
WARC-Record-ID: <urn:uuid:db3a40a5-a313-40db-b63d-5406246d1ca3>
Content-Type: application/http; msgtype=response
Content-Length: 78688

HTTP/1.1 200 OK
Date: Fri, 25 Feb 2011 18:33:03 GMT
Server: Apache
X-Powered-By: PHP/5.2.16
X-Pingback: http://ricardo.parente.us/xmlrpc.php
Set-Cookie: PHPSESSID=26e4aa448afbaa96a525926fd7286028; path=/
Connection: close
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<head profile="http://gmpg.org/xfn/11">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

<title>Ireland &laquo;  ColdFusion Developers Network</title>

<link rel="alternate" type="application/rss+xml" title="ColdFusion Developers Network RSS Feed" href="http://ricardo.parente.us/feed/" />
<link rel="alternate" type="application/atom+xml" title="ColdFusion Developers Network Atom Feed" href="http://ricardo.parente.us/feed/atom/" />
<link rel="pingback" href="http://ricardo.parente.us/xmlrpc.php" />
<link rel="shortcut icon" href="http://ricardo.parente.us/wp-content/themes/arclite/favicon.ico" />
<script language="javascript" type="text/javascript"> if (top.location != location) {   top.location.href = document.location.href ;  }
</script>
```

Figure 2.1: Example of a WARC record, consisting of both the header and content block.

There are a total of 8 different types of WARC records defined by the official WARC file specification as seen in Figure 2.2. Each record type will always follow the same WARC header convention with differences in optional headers fields and in the WARC content block. The warcinfo record type is the header record of a WARC file. Typically the first record in a WARC file, the warcinfo record contains the information about the web crawl that generated the current WARC file. The response record type contains the full HTTP response of a crawled URL including HTTP headers. If there are any errors due to link rot or other issues, the WARC writer should account for this and store in the WARC content block the full HTTP error message. Successful

HTTP responses typically usually includes the entire HTML of the web page being accessed and is stored in the the content block as retrieved. Resource type records differ from HTTP response record by removing protocol response data and containing just the raw data of the crawled URL. This includes HTML of a web page, images, videos, etc. Request type records are records generated from the HTTP request sent to a crawled URL and includes the full HTTP or HTTPS request scheme. Metadata type records are user specified generated records used to generate more descriptive data of other WARC record types. Revisit record types as the name implies are records specified to have already been crawled in the current web crawl. Conversion records are records specified to have transformed retrieved data in some way. The transformation is done because the resource data is in a deprecated file format that would be impossible to view on current systems like Adobe Flash Video files. Continuation records are records that are extensions of other record types and are created when a record's byte length exceeds a user defined limit. These are the 8 different types of records a person would encounter when viewing a WARC file. However, users will never usually directly view the WARC file themselves as the files are large and will crash most normal test editors. Thus external programs are usually used to interact with WARC files. Due to the difficulty of handling WARC files because of their size, the Internet Archive also designed a file format known as the CDX file to assist with the situation.



WARC Record Types

★ warcinfo
★ response
★ resource
★ request
★ metadata
★ revisit
★ conversion
★ continuation

```
WARC-Type   = "WARC-Type" ":" record-type
record-type = "warcinfo" | "response" | "resource"
            | "request" | "metadata" | "revisit"
            | "conversion" | "continuation"
```

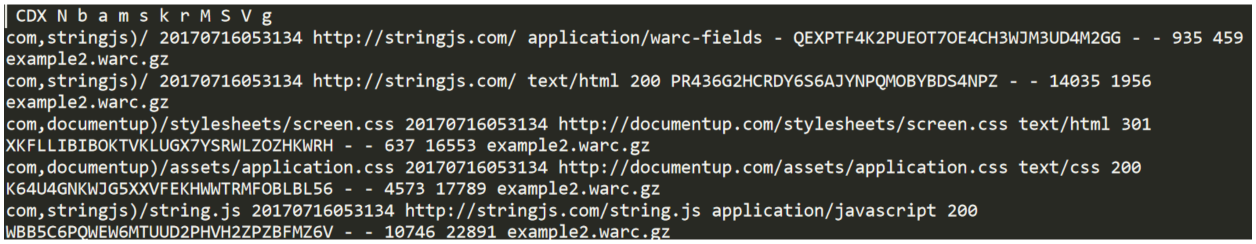
http://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.1/

@ibnesayeed 6

Figure 2.2: The possible types of WARC record types defined in [32].

2.2 CDX File Format

CDX files, first defined in 2015 by the Internet Archive, were created as a WARC record index file where each line of text in the CDX file refers to an individual WARC record in a WARC file [26]. CDX files are identified using the *.cdx* extension but are usually gzip compressed like WARC files to form the commonly seen *.cdx.gz* extension. CDX files are structured with the first line in the file being a header indicating the the format of each subsequent line. Users are able to define their own CDX header for CDX files using the Internet Archive's CDX legend [2]. However, most web crawlers use a standard known as the CDX11 format as shown shown in Figure 2.3. While the index contains useful information such as the type of corresponding WARC record, the most important fields are the compressed file offset and compressed record length. These fields allow programs to programmatically read just the chunk containing the WARC record of a WARC file corresponding to these fields without having to read the entire WARC file. The usefulness of these CDX file indices is why the Internet Archive typically bundles their WARC file dumps with a complimentary set of CDX files.



```
 CDX N b a m s k r M S V g
com,stringjs)/ 20170716053134 http://stringjs.com/ application/warc-fields - QEXPTF4K2PUEOT7OE4CH3WJM3UD4M2GG - - 935 459
example2.warc.gz
com,stringjs)/ 20170716053134 http://stringjs.com/ text/html 200 PR436G2HCRDY6S6AJYNPQMOBYBDS4NPZ - - 14035 1956
example2.warc.gz
com,documentup)/stylesheets/screen.css 20170716053134 http://documentup.com/stylesheets/screen.css text/html 301
XKFLLIBIBOKTVKLUGX7YSRWLZOZHKWRH - - 637 16553 example2.warc.gz
com,documentup)/assets/application.css 20170716053134 http://documentup.com/assets/application.css text/css 200
K64U4GNKWJG5XXVFEKHWWTRMFOBLBL56 - - 4573 17789 example2.warc.gz
com,stringjs)/string.js 20170716053134 http://stringjs.com/string.js application/javascript 200
WBB5C6PQWEW6MTUUD2PHVH2ZPZBFMZ6V - - 10746 22891 example2.warc.gz
```

Figure 2.3: An example of a CDX file, including the initial header legend and index lines.

2.3 WAT and WET Formats

Other than the Internet Archive, there are other organizations that participate in web crawling and internet archiving such as the Common Crawl Project. While the Internet Archive focuses on preservation, Common Crawl focuses on making web crawled data publicly accessible for scientific analysis and studies [3]. Common Crawl datasets date back to 2008 and have since 2013 been using the WARC file format to store their web crawls. Web crawled data at Common Crawl is posted monthly and is available to download for free at their public Amazon S3 instance.

Besides storing their data in the WARC file format they have also invented two variations of the WARC file format, the WAT and WET file formats. The WAT file format like a WARC file contains a series of WARC records. However, the WAT file format transforms regular WARC records into a JSON formatted block that only contains any data and metadata deemed useful by Common Crawl. As shown in Figure 2.4, the JSON format allows for easy programmatic parsing of a record and lowers the amount of data needed to parse a record. The WET file format resembles the original WARC file's header and content block record, however, the content block will only contain any plaintext data that is located at a URL. This is useful for storing news articles or forum posts which have pages filled with plaintext.

```
Envelope
  WARC-Header-Metadata
    WARC-Target-URI [string]
    WARC-Type [string]
    WARC-Date [datetime string]
    ...
  Payload-Metadata
    HTTP-Response-Metadata
      Headers
        Content-Language
        Content-Encoding
        ...
      HTML-Metadata
        Head
          Title [string]
          Link [list]
          Metas [list]
        Links [list]
      Headers-Length [int]
      Entity-Length [int]
      ...
    ...
  ...
Container
  Gzip-Metadata [object]
  Compressed [boolean]
  Offset [int]
```

Figure 2.4: A WAT file JSON envelope example including header block and content block.

WARC files are the archival file type of choice for web crawls. They can be indexed using created CDX files and to allow for easier extraction of records. WAT and WET files variations, created the Common Crawl Project, exist to allow for easier programmatic parsing and are made available for free publicly along with the original WARC files by the Common Crawl organization. In order to parse WARC, CDX, and WAT files, we created our own parsing tool with enhancements and features others might find useful which we will discuss in the next chapter.

2.4 Linear Hashing

In order to build a database capable of indexing on large WARC file collections we first needed to decide upon an underlying data structure. For our database's underlying data structure we chose to implement a Linear Hash Table as the backbone of the database system. Linear Hashing is one many techniques in a group of hashing schemes known as dynamic hashing [18]. Dynamic hash tables shrink and expand by utilizing hash functions to generate more key bits as the table expands while reducing the number of key bits as the table shrinks. Dynamic hashing techniques fall under two schemes one with directories and one without. Extendible Hashing is a hashing technique that falls under the scheme with directories. Extendible Hashing has a separate structure known as a directory. Each entry in the directory points to a bucket where the data is hashed. An example of the directory structure for Extendible Hashing is displayed in Figure 2.5. This hashing scheme grows exponentially as the number of buckets double as the number of directory bits are doubled. On the other hand, Linear Hashing is a directory-less scheme and expands it's hash table one bucket at a time. Linear Hashing is directory-less because keys are hashed directly to a bucket. Linear Hashing get its name from the one bucket at a time growth, hence the name Linear Hashing. One small advantage Extendible hashing has over Linear Hashing is that the due its criterion design it is possible to set an upper bound to limit on access times for get operations. However, the wasted memory from an exponentially growing directory is a downside that puts Extendible Hashing below Linear Hashing in most use cases. Thus the data structure chosen for WARC-Kits database design was an on disk Linear Hash Table implementation.

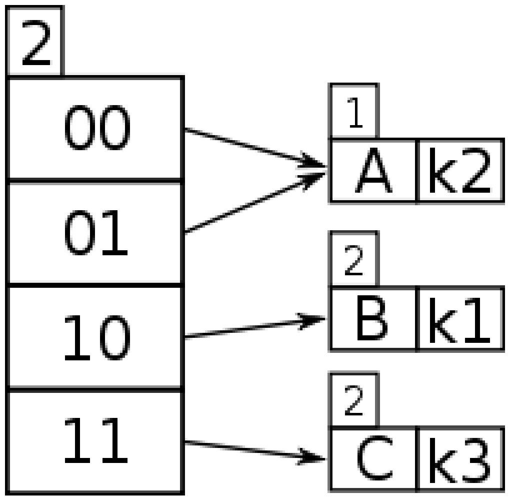


Figure 2.5: The directory structure of the Extendible Hashing scheme.

Linear Hashing has 4 factors that need to be tracked to determine when to grow or shrink the table. *N* which is the number of initial buckets and must always be a power of 2, *S* a pointer to a bucket designated to be split when a split criterion is met, *I* which is the number of bits used to address N buckets, hence why *N* must be a power of 2, and a load factor which is the number of items in the table divided by the current number of buckets times the average or maximum size of each bucket. Load factor is used as the threshold to grow or shrink the hash table one bucket at a time. All of these factors revolve around a central hash function, whose *I* bits become our bucket index.

Hash functions are designed to always output a specific number of bits no matter the input. Connecting this feature of hash functions to our Linear Hash table let's say for example, some hash function gives a 32-bit output for a key of some size. In Linear Hashing we will only use the first *I* bits of the hash function output, since a Linear hash table only indices initially to N buckets. If we start with $N = 2$ buckets, 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. Once number of insertions exceeds the load factor, we add one additional bucket to *N*. If we exceed the number of buckets addressable with I bits, we need to increment I to be able to address to the new bucket. When any bucket is added we rehash the keys at split bucket *S* and then increment *S* to indicate the next bucket as the new split bucket. Once *N* has doubled from where it was initially, we reset the *S* the split bucket index back to 0.

A Linear Hash Table gives us the flexibility of growing our database as we insert in new data while maintaining near constant average access and insertion times. Data systems that use Linear Hashing include BerkleyDB a commercial key-value embedded database system currently owned by Oracle[5]. In the next chapter we will discuss our implementation of our own Linear Hash Table in JavaScript and our improvements to make it document oriented. However, before that we discuss in the next section why we chose to develop our tool in JavaScript using the Node.js environment.

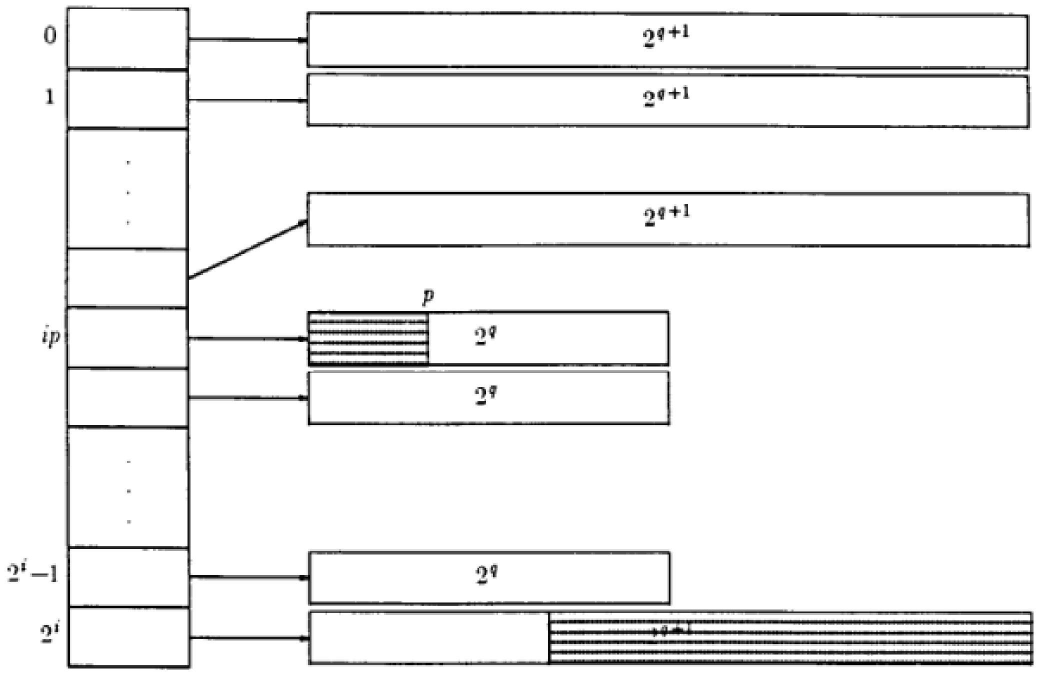


Figure 2.6: Example of how Linear Hashing is conducted, where *I* bits represents a mapping to a bucket index. From [18], Figure 21.

2.5 Node.js

We chose to create WARC-KIT tool in JavaScript due to the distinct lack of both JavaScript libraries to work with WARC files and lack of database implementations in JavaScript. Although JavaScript is the defacto programming language of the web, the vast majority of web crawling and WARC parsing tools are created in Java and Python [25]. Besides being used for web programming, JavaScript can also be run server side with Node.js, an asynchronous JavaScript runtime environment based on Google's JavaScript V8 engine [12]. Node.js is both a fast and lightweight runtime environment that has become the backend environment of choice that many companies use for great results. For instance in 2012, LinkedIn switched their mobile backend servers from Ruby on Rails to Node.js and saw 20x increase in performance during testing scenarios [17]. Additionally, the advantage of unifying frontend and backend development under one programming language allows for the development of software that interacts with both ends of the stack. Hence why WARC-KIT was created specifically in Node.js to take advantage of the software that was developed in the unified JavaScript ecosystem.

One of the greatest assets of Node.js is the Node Package Manager or as its more commonly known NPM. NPM allows any user to publish their own packages to the NPM registry. Other users can then download and use any package in the NPM registry for their own JavaScript programs. NPM is easy to install and easy to use while maintaining the dependency structure of any downloaded packages by downloading a local copy to a folder known as node modules. This is done so that if an author updates their package on the NPM registry your Node.js program won't suddenly stop working. There are caveats to working with NPM though, for instance when you download a package from NPM all of its dependencies are added to node modules. Downloading simple package for managing server routes can balloon into a 100 package dependency chain which can be hundred of megabytes in size to your node modules. There's also no good way to move or delete node modules because of its size. This has lead to the creation of recursive NPM packages just to remove node modules [19]. However, the advantage of having access to the open source Node.js community far outweighs these minor detriments. As such many useful packages were included into the design of WARC-KIT. A brief highlight the main NPM packages used in WARC-KIT can be found below in table 2.1.

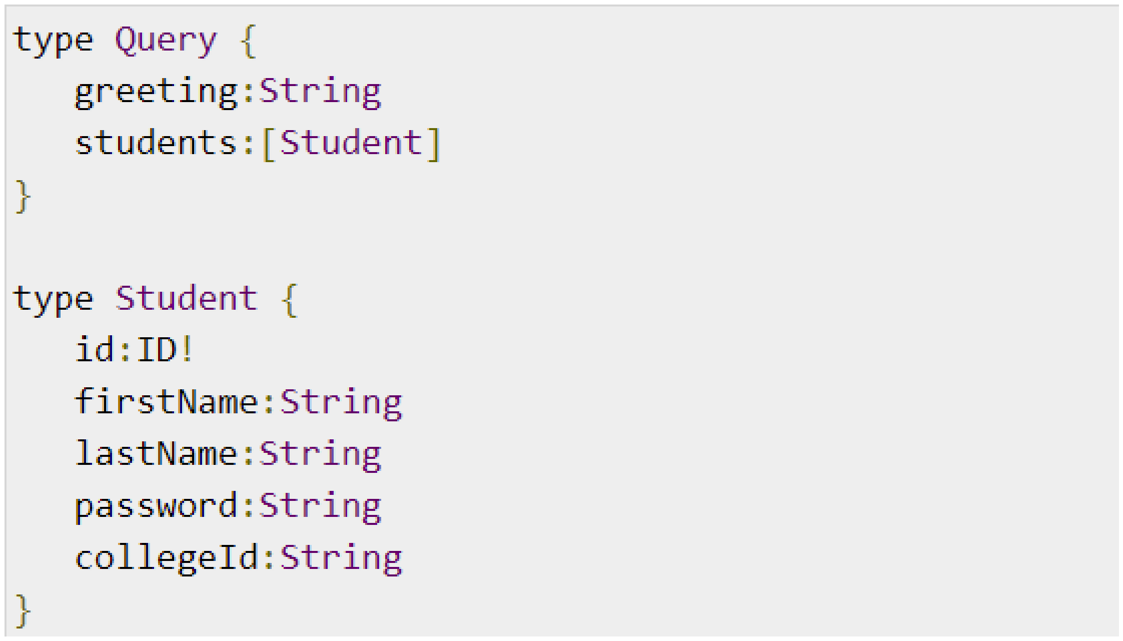
Table 2.1: NPM Packages used in WARC-KIT.

Package Name	Description
fs-extra	Enhancement of the standard Node.js fs library
express	A web application framework
node-fetch	Node.js port of the Fetch API
axios	Promise based HTTP client
locutus	Node.js port of other programming language libraries
graphql	Data query language for APIs

2.6 GraphQL

While the other Node.js packages are notable in their own right, GraphQL is distinctly notable for WARC-KIT as it was chosen as the query language for the database system. GraphQL is an open source query language created by Facebook to query web APIs [6]. GraphQL also includes a server side runtime environment for executing the queries and has language bindings for almost every currently used programming language. Specifically in Node.js, GraphQL has integration with popular server frameworks such Express.js and Apollo.js. This allows developers to build their APIs from the ground up with GraphQL queries in mind. Some notable companies that currently use GraphQL for their APIs include NBC, Starbucks, Twitter, PayPal, Github, and of course Facebook.

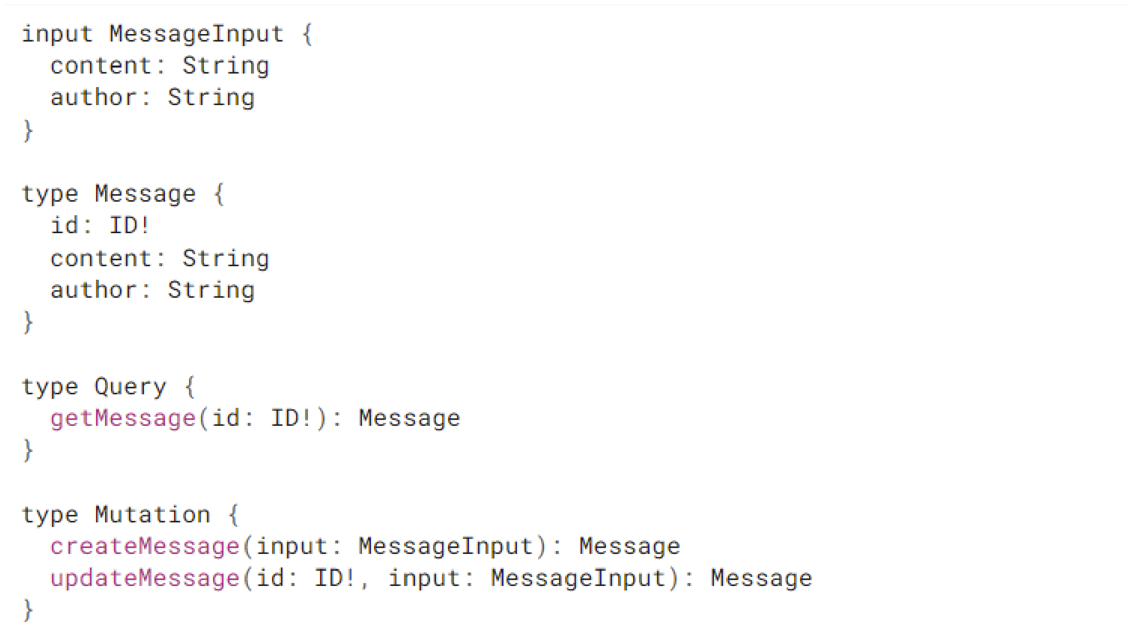
GraphQL starts with a user defined API schema. Here the user has to strictly define what custom data types GraphQL can read, what types of queries can be performed, and what mutations on the data source can be made. As show in Figure 2.7 every possible query must be defined and any classes and member variables must be defined as well. Types that do not need to be defined are primitive types String, Int, Float, Boolean, and ID.



```
type Query {
   greeting:String
   students:[Student]
}

type Student {
   id:ID!
   firstName:String
   lastName:String
   password:String
   collegeId:String
}
```

Figure 2.7: Example of a simple GraphQL Schema definition.

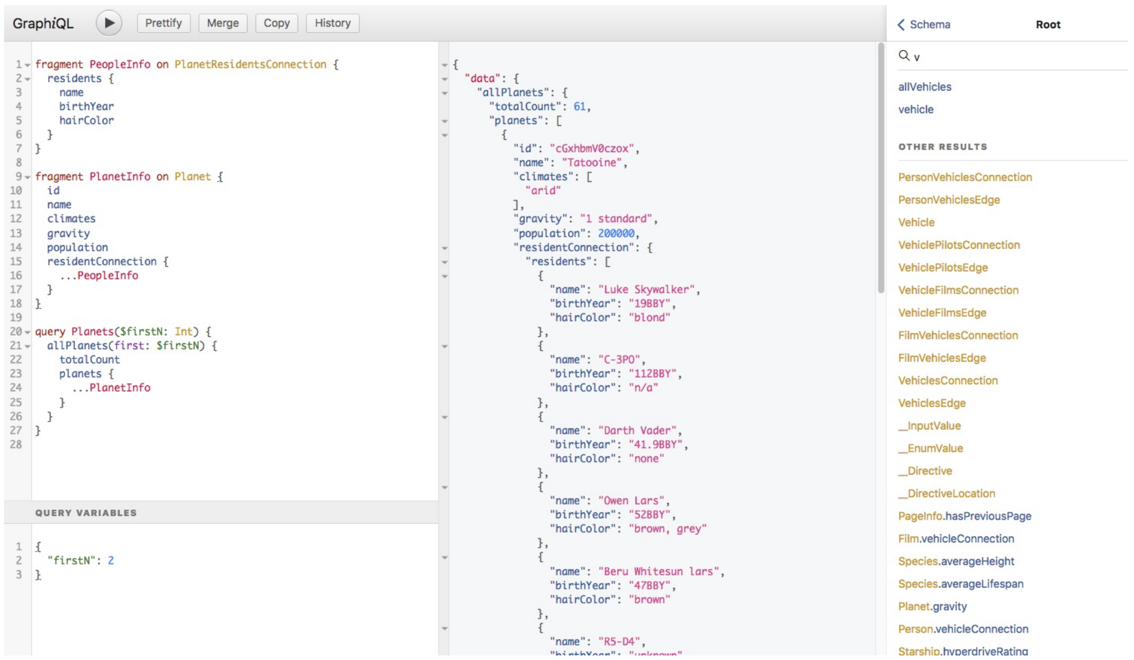
Any changes to the data source must also be defined as a mutation in GraphQL. While it is possible to modify a data source with a query in GraphQL, mutations are defined purely to differentiate GraphQL queries that alter data rather than read. Defining mutations and queries is similar to defining ordinary API routes for a backend server. Mutations and queries must also be assigned corresponding user defined resolver functions which perform the actual work of querying the data source. As shown in Figure 2.8 mutations like queries must be defined in the schema and cannot be changed once a schema is created.



```
input MessageInput {
  content: String
  author: String
}

type Message {
  id: ID!
  content: String
  author: String
}

type Query {
  getMessage(id: ID!): Message
}

type Mutation {
  createMessage(input: MessageInput): Message
  updateMessage(id: ID!, input: MessageInput): Message
}
```

Figure 2.8: Example of a GraphQL Schema definition with both queries and mutations.

The original function and motivation for GraphQL is to allow users to query API data sources in one HTTP request. Users would pass in a query function in as an HTTP request and receive back the data returned from the GraphQL resolver function. An added bonus is that GraphQL also provides a built in GUI for developers and users to test out queries. This GUI program known as GraphiQL, notice the extra i, comes built in and is activated as a argument parameter when starting a GraphQL server. Users and developers can then navigate to the GUI running on localhost to test out queries directly with the GraphQL server. An example of the GraphiQL interface can be seen in Figure 2.9.



```
fragment PeopleInfo on PlanetResidentsConnection {
  residents {
    name
    birthYear
    hairColor
  }
}

fragment PlanetInfo on Planet {
  id
  name
  climates
  gravity
  population
  residentConnection {
    ...PeopleInfo
  }
}

query Planets($firstN: Int) {
  allPlanets(first: $firstN) {
    totalCount
    planets {
      ...PlanetInfo
    }
  }
}
```

```
{
  "firstN": 2
}
```

```
{
  "data": {
    "allPlanets": {
      "totalCount": 61,
      "planets": [
        {
          "id": "cGxhbmV0czox",
          "name": "Tatooine",
          "climates": [
            "arid"
          ],
          "gravity": "1 standard",
          "population": 200000,
          "residentConnection": {
            "residents": [
              {
                "name": "Luke Skywalker",
                "birthYear": "19BBY",
                "hairColor": "blond"
              },
              {
                "name": "C-3PO",
                "birthYear": "112BBY",
                "hairColor": "n/a"
              },
              {
                "name": "Darth Vader",
                "birthYear": "41.9BBY",
                "hairColor": "none"
              },
              {
                "name": "Owen Lars",
                "birthYear": "52BBY",
                "hairColor": "brown, grey"
              },
              {
                "name": "Beru Whitesun lars",
                "birthYear": "47BBY",
                "hairColor": "brown"
              },
              {
                "name": "R5-D4",
```

Figure 2.9: The GraphiQL GUI which allows users and developers to perform GraphQL queries on the fly.

The ease of use and widespread availability of GraphQL is why we chose to integrate it as the query language for WARC-Kits database. Since GraphQL has its own syntax other programming languages can simply send GraphQL formatted HTTP requests to a running GraphQL server and receive a GraphQL formatted response. This allows WARC-KIT the ability to interfaced by other languages similar to a database driver. The implementation of this we will be discussing in the next chapter.

3. Implementation

In this chapter we discuss the implementation of the various tools that compose WARC-KIT. These include both a standalone WARCFilter tool that interacts with WARC files, a database system with an underlying Linear Hash Table implementation, and their combination with a GraphiQL server to create WARC-KIT.

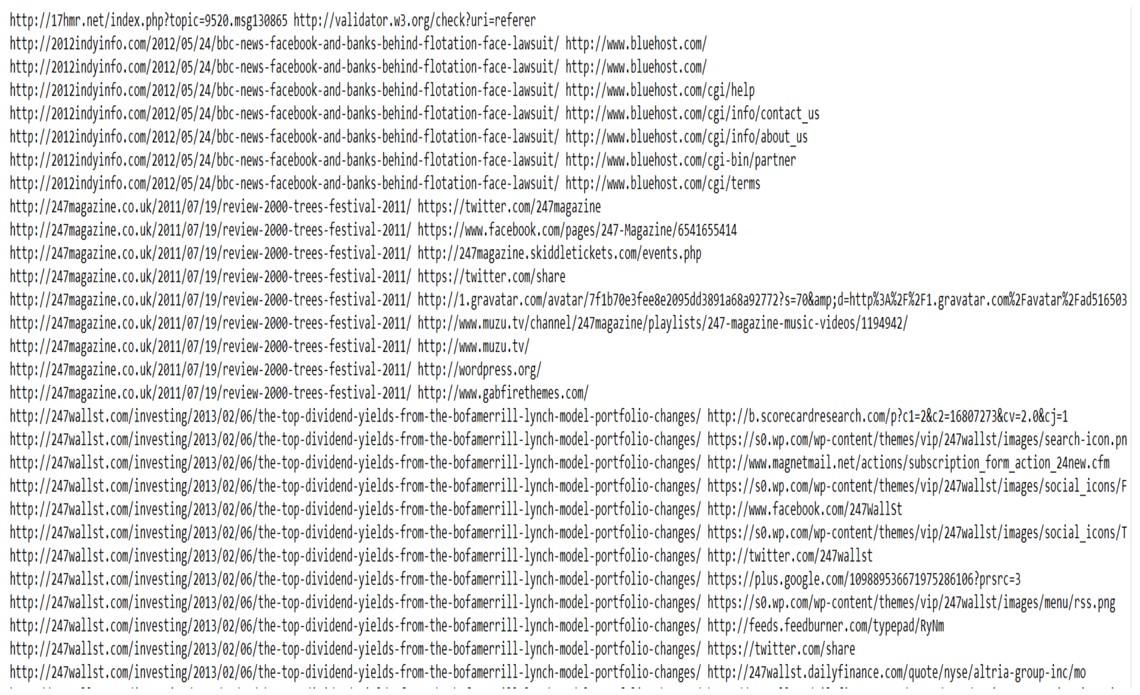
3.1 WARCFilter

In order to create our own enhanced indexing on WARC files, we created our own parsing program that is capable of parsing WARC files. WARC files are large in size and cannot be read into memory all at once. Therefore file streams, where file data is loaded into memory only a few hundred bytes at a time, are necessary to process WARC files. File streams also allow us to process the data as soon as it arrives rather than wait for the full file to be read into memory. As our tool is made in Node.js, we use the *fs-extra* library, a community patch of the standard Node.js fs library, to help us. We first simply open the WARC file in a gzip decompression stream to decompress the initial stream of data. Then feed that decompression stream to a standard read stream to allow us to begin reading the WARC file data. As seen before in Figure 2.1. Each WARC record begins with a header block with the WARC format type; which is WARC 1.0 currently. We simply take this initial format type as the start of a WARC record and read until we encounter a new line. The header block and content block are always separated by a new line so, we treat any previous line as part of the header block and any subsequent line as part of the content block. We continue reading through the content block until we encounter another WARC 1.0 style format header. We take this encounter as the start of another WARC record and treat all previous read lines as one WARC record. With that we are now able to simply run through a WARC file record by record.

With the traversal through a WARC file completed, the creation of a CDX index file is now possible as well. In order to create a CDX index file of a WARC record we keep track using a counter of the number of compressed bits going into the decompression stream and the number of bits of text coming out of the decompression stream to get the compressed record offset and length. We then convert the relevant fields of the WARC record header into the CDX style format and write the index to our created CDX file. One thing to note is that legend of a CDX header in a CDX file defined by the Internet Archive does not have an official mapping from the WARC record counterpart to the CDX. As such what is included in a CDX index is usually left to the author writing a CDX creator. That is why CDX files from different organizations can have slightly different fields even if both organizations are following the same CDX header style. For our tool, we follow the Internet Archive's CDX format. Besides the parsing of WARC files and creation of CDX indices, our custom parsing tool includes a feature to retrieve the full WARC record using the CDX's WARC compressed file offset and length. Users can then specify the creation of a new WARC file containing the retrieved records.

Another added feature is to parse and create new WARC files from records retrieved based on a argument format based on date, URL or type of record. This allows for the creation of custom WARC files containing only WARC records that matched the query argument. Meaning if the user uses this tool on a WARC file collection they can extrapolate only records that match their query criteria and have their own WARC file collection comprised of only records that they want. This filtering of WARC records is what gives the standalone tool, WARCFilter, its name.

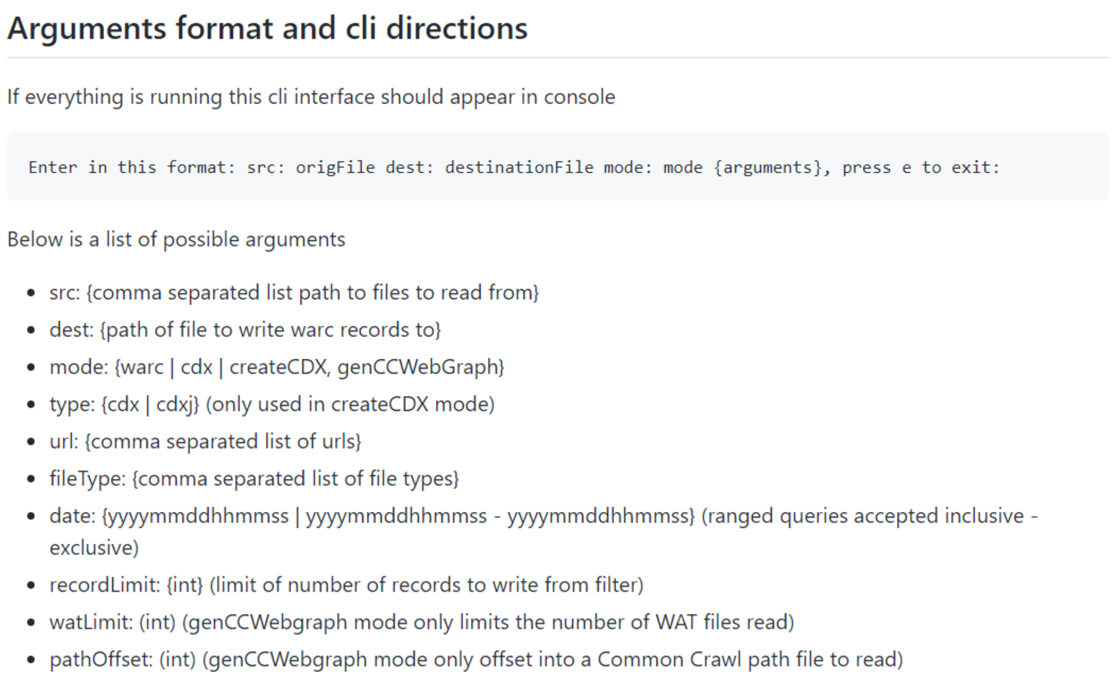
An additional feature of this tool is the generation of webgraph datasets using Common Crawl's WAT files. Webgraphs are directed graphs where vertices are URLs representing web pages on the internet. Each directed edge in the Webgraph represents the link from one web page to another. WARCFilter when given a Common Crawl *wat.paths.gz* file as input, parses the paths file, downloads the corresponding WAT file at the path URL, and begins parsing the WARC JSON records. Only JSON records that are web pages are parsed and any link information that was on the original web page is retrieved. The original URL of the web page and any embedded links are written to a text file where each line is an original web page URL and an external link URL pair. Each line represents an edge in a graph and can thus be evaluated and analyzed using graph algorithms. Figure 3.1 shows a an example of how each line in the webgraph is written.



```
http://17hmr.net/index.php?topic=9520.msg130865 http://validator.w3.org/check?uri=referer
http://2012indyinfo.com/2012/05/24/bbc-news-facebook-and-banks-behind-flotation-face-lawsuit/ http://www.bluehost.com/
http://2012indyinfo.com/2012/05/24/bbc-news-facebook-and-banks-behind-flotation-face-lawsuit/ http://www.bluehost.com/
http://2012indyinfo.com/2012/05/24/bbc-news-facebook-and-banks-behind-flotation-face-lawsuit/ http://www.bluehost.com/cgi/help
http://2012indyinfo.com/2012/05/24/bbc-news-facebook-and-banks-behind-flotation-face-lawsuit/ http://www.bluehost.com/cgi/info/contact_us
http://2012indyinfo.com/2012/05/24/bbc-news-facebook-and-banks-behind-flotation-face-lawsuit/ http://www.bluehost.com/cgi/info/about_us
http://2012indyinfo.com/2012/05/24/bbc-news-facebook-and-banks-behind-flotation-face-lawsuit/ http://www.bluehost.com/cgi-bin/partner
http://2012indyinfo.com/2012/05/24/bbc-news-facebook-and-banks-behind-flotation-face-lawsuit/ http://www.bluehost.com/cgi/terms
http://247magazine.co.uk/2011/07/19/review-2000-trees-festival-2011/ https://twitter.com/247magazine
http://247magazine.co.uk/2011/07/19/review-2000-trees-festival-2011/ https://www.facebook.com/pages/247-Magazine/6541655414
http://247magazine.co.uk/2011/07/19/review-2000-trees-festival-2011/ http://247magazine.skiddletickets.com/events.php
http://247magazine.co.uk/2011/07/19/review-2000-trees-festival-2011/ https://twitter.com/share
http://247magazine.co.uk/2011/07/19/review-2000-trees-festival-2011/ http://1.gravatar.com/avatar/7f1b70e3fee8e2095dd3891a68a92772?s=70&amp;d=http%3A%2F%2F1.gravatar.com%2Favatar%2Fad516503
http://247magazine.co.uk/2011/07/19/review-2000-trees-festival-2011/ http://www.muzu.tv/channel/247magazine/playlists/247-magazine-music-videos/1194942/
http://247magazine.co.uk/2011/07/19/review-2000-trees-festival-2011/ http://www.muzu.tv/
http://247magazine.co.uk/2011/07/19/review-2000-trees-festival-2011/ http://wordpress.org/
http://247magazine.co.uk/2011/07/19/review-2000-trees-festival-2011/ http://www.gabfirethemes.com/
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ http://b.scorecardresearch.com/p?c1=2&c2=16807273&cv=2.0&cj=1
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ https://s0.wp.com/wp-content/themes/vip/247wallst/images/search-icon.pn
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ http://www.magnetmail.net/actions/subscription_form_action_24new.cfm
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ https://s0.wp.com/wp-content/themes/vip/247wallst/images/social_icons/F
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ http://www.facebook.com/247WallSt
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ https://s0.wp.com/wp-content/themes/vip/247wallst/images/social_icons/T
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ http://twitter.com/247wallst
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ https://plus.google.com/109889536671975286106?prsrc=3
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ https://s0.wp.com/wp-content/themes/vip/247wallst/images/menu/rss.png
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ http://feeds.feedburner.com/typepad/RyNm
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ https://twitter.com/share
http://247wallst.com/investing/2013/02/06/the-top-dividend-yields-from-the-bofamerrill-lynch-model-portfolio-changes/ http://247wallst.dailyfinance.com/quote/nyse/altria-group-inc/mo
```

Figure 3.1: Example of a webgraph dataset file generated from a Common Crawl WAT file collection.

WARCFilter can be used independent of WARC-KIT, and comes with its own CLI program. From the CLI program users can specify a source and destination file/folder to create either WARC files, CDX files, or webgraph datasets. As shown in Figure 3.2 the argument format is simply one long formatted string. WARCFilter come with 4 modes that have to specified in the command string. The first mode *warc* will parse a WARC file and write WARC records that match the query criteria to the specified destination file. The second mode *cdx* will read a CDX file, extract records from the corresponding WARC file and output the WARC records that pass the filter to a specified destination file. The third mode *creatCDX* will generate a CDX index file on a specified WARC file. The fourth mode *genCCWebgraph* will create a webgraph dataset file from a specified Common Crawl *wat.paths* file.


Arguments format and cli directions

If everything is running this cli interface should appear in console

```
Enter in this format: src: origFile dest: destinationFile mode: mode {arguments}, press e to exit:
```

Below is a list of possible arguments

- src: {comma separated list path to files to read from}
- dest: {path of file to write warc records to}
- mode: {warc | cdx | createCDX, genCCWebGraph}
- type: {cdx | cdxj} (only used in createCDX mode)
- url: {comma separated list of urls}
- fileType: {comma separated list of file types}
- date: {yyyymmddhhmmss | yyyymmddhhmmss - yyyymmddhhmmss} (ranged queries accepted inclusive - exclusive)
- recordLimit: {int} (limit of number of records to write from filter)
- watLimit: (int) (genCCWebgraph mode only limits the number of WAT files read)
- pathOffset: (int) (genCCWebgraph mode only offset into a Common Crawl path file to read)

Figure 3.2: The argument format of the WARCFilter CLI program.

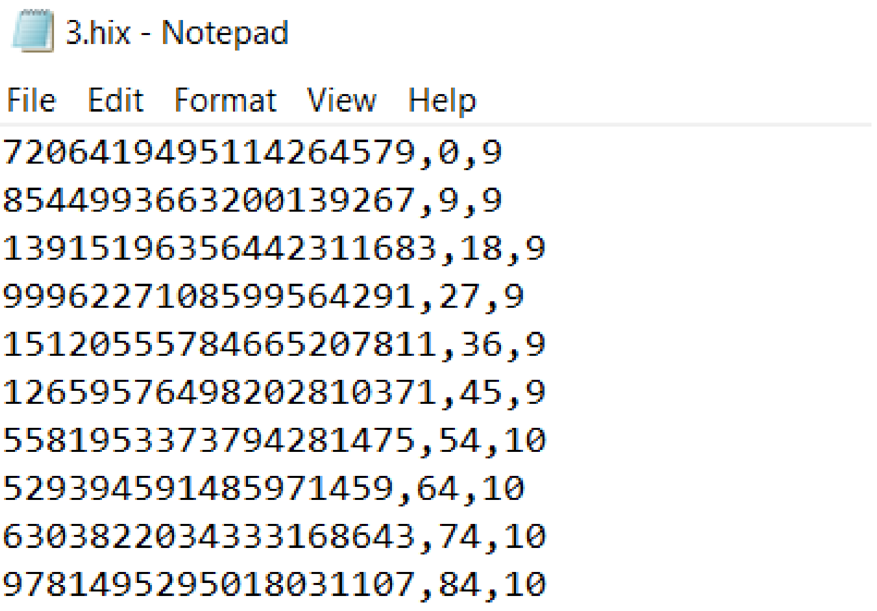
The code for the standalone WARCFilter tool can be found on GitHub under the name *warcfilter* [33]. In the next section we will describe the creation of our JavaScript database system that works in tandem with WARCFilter to allow database style queries on WARC collections.

3.2 Node.js Database

Although JavaScript is known as the language as the web, even with the advent of Node.js, JavaScript still falls behind in performance compared to languages like C/C++. Unsurprisingly, most database implementations are in C/C++ for performance. However, JavaScript implemented databases do exist, such as PouchDB an in memory database that runs within the browser [16]. Hybrid databases also exist such as HarperDB, a JavaScript implemented database that has as its underlying implementation connected to a Lightning Memory-Mapped Database (LMDB), a C implemented database [7]. One of the goals in the creation of WARC-KIT is to make the implementation wholly in JavaScript so that future development can be done solely in one language. Since JavaScript is a web based language, we have to take advantage of its strengths in order to compete with other database implementations.

Most database management systems use an underlying on disk data structure like a B+ tree for LMDB or a Log-structured merge-tree such as Facebook's RocksDB.[21]. One of the most well known examples is SQLite which condenses its data into one file organized as a B+ tree. For the implementation of WARC-KIT's on disk database we used the previously discussed Linear Hash Table data structure as our underlying implementation. The Linear Hash Table allows for the linear growth of the table to store increased items while maintaining quick access times with its key-value storage paradigm.

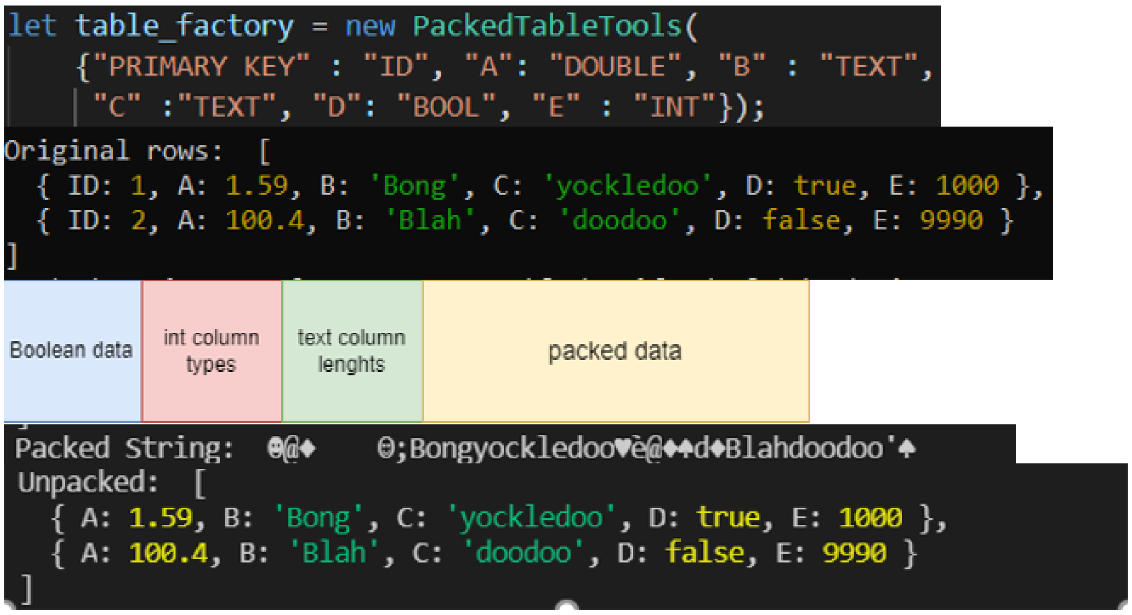
As we have discussed the various details of a Linear Hash Table in the previous chapter we now translate that design into code. The hash function chosen for the Linear Hash Table was the MD5 hash function. MD5 was chosen for its relatively small 128 bit hash output and fast computation speed. MD5 loses in computation speed only to SHA-1, which has a 160 bit hash output, and its predecessor MD4 which has more collision issues than its successor [13]. Only the first I bits of the MD5 hash will be used to index to a bucket giving us an absolute maximum of 2^{128} - 1 possible buckets to index to. Our bucket design takes inspiration from the Extendable Hashing structure while preserving the Linear Hash Table Structure. Buckets are represented as a folders containing two files a header *.hix* file and data file a *.txt* file. The header file contains index lines with each line containing the hash of the key plus the offset and length of the corresponding value in the data file. Figure 3.3 shows an example header file. This allows us to only parse the header file and then quickly read the corresponding chunk we need from the data file once we find the right key. The default design of a hash table however, is restrictive because associating a key to a single value is limiting in storing extra data. So we opted to improve our Linear Hash Table by adding a document style design for storing values.



```
7206419495114264579,0,9
8544993663200139267,9,9
13915196356442311683,18,9
9996227108599564291,27,9
15120555784665207811,36,9
12659576498202810371,45,9
5581953373794281475,54,10
529394591485971459,64,10
6303822034333168643,74,10
9781495295018031107,84,10
```

Figure 3.3: The indices in a header *.hix* file. Each line contains a hash in JavaScript BigInt representation, a offset, and a length of a value in the data file.

In PHP there is a function called pack, which is a port of the Perl function of the same name. The pack function allows us to encode primitive variables such as ints, floats, and booleans to a binary string format. Recall that we discussed earlier that the open source search engine Yioop is implemented in PHP. In Yioop, there is a class called PackedTableTools which allows users to define a SQL like table with with each column packed into binary string using PHP's pack. For the Linear Hash Table database a JavaScript port was made of PackedTableTools so that we could use this format in the table. The PHP function pack utilized in Yioop's PackedTableTools was ported with a JavaScript version of the pack function from Locutus, a Node.js library intent on porting every function from every other programming language such as PHP [9]. With the implementation of the PackedTableTools in JavaScript we were able to design a document style format for the Linear Hash Table. The primary key of the PackedTableTools format serves as the key hashed in the Linear Hash Table while the rest of the columns in the format will packed into a binary string representing the value in the hash table. Figure 3.4 displays the process of defining a PackedTableTools format, packing formatted objects to a binary string, and then unpacking. With the inclusion of PackedTableTools into our database, we are able to achieve a document style format for storing keys and values in our Linear Hash Table.



```
let table_factory = new PackedTableTools(
    {"PRIMARY KEY" : "ID", "A": "DOUBLE", "B" : "TEXT",
     "C" :"TEXT", "D": "BOOL", "E" : "INT"});
Original rows:  [
  { ID: 1, A: 1.59, B: 'Bong', C: 'yockledoo', D: true, E: 1000 },
  { ID: 2, A: 100.4, B: 'Blah', C: 'doodoo', D: false, E: 9990 }
]
```

```
Packed String:  ☻@♦    Θ;Bongyockledoo♥è@♦♠d♦Blahdoodoo'♠
Unpacked:  [
  { A: 1.59, B: 'Bong', C: 'yockledoo', D: true, E: 1000 },
  { A: 100.4, B: 'Blah', C: 'doodoo', D: false, E: 9990 }
]
```

Figure 3.4: The process of defining and packing data using PackedTableTools. PackedTableTools formatted objects are packed into formatted binary string with metadata columns boolean, int, and text proceeding the packed data.

Four main functions were implemented into the Linear Hash Table, a put, get, delete, and update function. However, unlike in memory hash tables designs, the time complexity in the worse case for the Linear Hash Table differs. For a get operation we have to search the number of I initial bit buckets we first started with, to $I + X$ number of bits we have added. For example, if our Linear Hash Table first started with $I = 5$ bits to map 2^5 buckets and eventually grew to map 2^7 buckets with $I = 7$ bits we would need to search 3 buckets. Consequently the time complexity in the worse case for a get is $I_{\text{curr}} - I_{\text{init}} + 1)$ buckets. In practice, if we set our initial addressable buckets to a high number of I bits we will only need to search the length of a single bucket. A put operation in the Linear Hash Table has the same time complexity as a get operation but as a slightly worse as we have to check for duplicates. If no duplicates are found the put is simply a file append. Since we are writing our data to text files, on disk deletion and update operations are expensive. To optimize deletions only the index in the header file is deleted when a delete operation is called on a valid key. When the number of deletions surpass a threshold of 10% of the number of values in the data file we rewrite the data file to excluding data whose headers were deleted. Updates require a full data file rewrite for operations as we need to retrieve the packed data row update the value and rewrite. With those 4 functions implemented the, creation of a on disk database in JavaScript is complete. While the creation of the database complete there is still one more thing to implement for the database and that is a database driver.

3.3 Express Servers

Database connectivity drivers offer a way for programs to establish a connection to a database. These drivers also offer an API for programming languages to programmatically establish connections to database systems. The most well known example of this is the ODBC driver or the Open Database Connectivity driver [10]. ODBC was designed to be independent of any database management system so that developers could write ODBC compliant drivers for their own databases. For WARC-KIT, the original plan was to write a ODBC compliant driver for our Linear Hash Table database. Unfortunately because there are no JavaScript databases that have used ODBC in the past, writing a ODBC compliant driver in JavaScript would have to be done from the ground up with no documentation. So, we opted for a more JavaScript like solution for an API to access the database and it all starts with Express.js.

Express.js is a back end server framework in Node.js that has become the unofficial standard server framework for Node.js. For the Linear Hash Table, to setup database driver style API we opted to setup an Express HTTP server with HTTP routes that can interact with and manipulate a Linear Hash Table. The standalone nature of HTTP means that other programming languages can make HTTP requests to our Express server to interact with our database system. Allowing for cross language compatibility. However, that is not all we did with Express servers we also opted utilize GraphQL, the API query language discussed in the previous chapter, as a query language for our database rather than make the user send our own user defined format. GraphQL has a node package that also provides direct integration with Express with the aptly named *express-graphql*. However, there is an issue with having all API functionality routed through GraphQL.

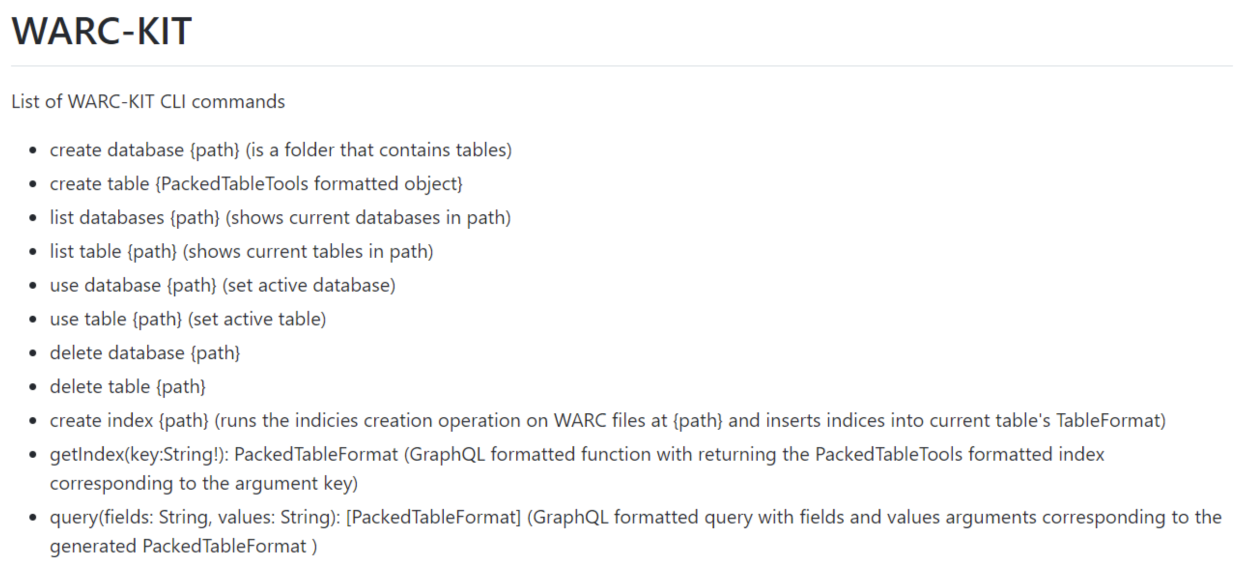
GraphQL as we discussed in the previous chapter must have a strictly defined static schema before creating a GraphQL Express server. Unfortunately for our database system, table formats can differ between table to table as the PackedTableTools format is user defined. All database systems are capable of having different formats for each created table. So, having a single fixed format for each table in our database system is neither scalable nor dynamic. The work around for this is to dynamically generate a schema before creating a GraphQL Express server. When a user specifies and creates a Linear Hash Table with a defined PackedTableTools format we generate GraphQL schema to match the format and write the schema into a file stored along with the Linear Hash Table. When we switch between tables with different formats we simply close the server with one format and open a new one with the needed file. This allows us to have a somewhat dynamic format with GraphQL even though GraphQL was originally static schemed. Unfortunately even with this workaround we could not route all database system API functionality through GraphQL.

GraphQL mutations have a stricter calling syntax than queries. Since mutations are supposed change the underlying data source and we intend the have users be able to define many different types of table schema, dynamic generation of a GraphQL with mutations is an arduous task. As such, we opted to have two API servers running simultaneously. One server is a regular Express.js server to handle regular database management system operations such as creating new tables, inserting values into tables, deleting tables etc. The other server would be a Express GraphQL server that can only query the Linear Hash Table its attached to. This solution requires the shut down and start up of a different GraphQL server each time a user is switching between different types of Linear Hash Tables. Upon the completion of our database API we have finished the final piece to needed create WARC-KIT.

3.4 WARC-KIT

With the creation of of both a WARCFilter tool to parse WARC files, a database system with underlying Linear Hash Table implementation, and a database API using Express.js in tandem with GraphQL we are ready for the final task. The final step is to combine all of our tools into a unified application we call WARC-KIT. The main function of WARC-KIT is to combine all the previous implemented components into one application that users can use to index and query their WARC file collections, In order to interact with WARC-KIT directly in Node.js a simple console program was developed to allow users to type in commands to directly interact with database functions. All commands typed to the console are translated to HTTP requests made using axios, a Node.js HTTP request client package. The requests are sent to the main Express server which handles and translates the request to a database system function. This means that users can make a HTTP request directly to the server rather than just using the console program. When first running WARC-KIT the initial Express server will start immediately and display the console program. HTTP requests to perform operations can be done immediately at that point.

The first step to getting started with WARC-KIT is to create a database. A database will be any folder specified to contain any Linear Hash Tables. Once that is finished users can then create a Linear Hash Table with a user defined PackedTableTools format. All created tables will be saved to current active database. Users can then specify the folder containing their WARC files as the target of an ingesting operation. WARC-KIT will then use WARCFilter to parse all WARC files located in the specified folder and generate a PackedTableTools formatted row from each parsed WARC record. Each row representing, a custom index format, is then inserted into the Linear Hash Table. When the insertions are finished a GraphQL Express server is generated for the defined PackedTableTools format and run on the port adjacent to the main Express server. Users can then send query requests to the GraphQL server to query their active Linear Hash Table. Users can query by key value which is the URL of the WARC record or they can query by any field in their PackedTableTools format. Users will then receive their results in GraphQL formatted HTTP response. In GraphiQL, all query results will be directly displayed to the right of the screen. Users are then able to feed these indices back into WARCFilter to generate a WARC file containing these records or are free to feed the indices to their own WARC generation programs as well. The ingestion operation only needs to be done once as when the user closes WARC-KIT, the configurations of both the table and the GraphQL server will be saved to a parameters file. This file will be loaded again on program restart. The one time cost of indexing on a large WARC collection is invaluable as every subsequent query to receive desired WARC records can performed without having to parse through the entire WARC file collection. Figure 3.5 contains a list of the possible commands for WARC-KIT.


WARC-KIT

List of WARC-KIT CLI commands

- create database {path} (is a folder that contains tables)
- create table {PackedTableTools formatted object}
- list databases {path} (shows current databases in path)
- list table {path} (shows current tables in path)
- use database {path} (set active database)
- use table {path} (set active table)
- delete database {path}
- delete table {path}
- create index {path} (runs the indicies creation operation on WARC files at {path} and inserts indices into current table's TableFormat)
- getIndex(key:String!): PackedTableFormat (GraphQL formatted function with returning the PackedTableTools formatted index corresponding to the argument key)
- query(fields: String, values: String): [PackedTableFormat] (GraphQL formatted query with fields and values arguments corresponding to the generated PackedTableFormat)

Figure 3.5: List of commands possible through the WARC-KIT CLI.

Figure 3.6 both illustrates how a user interacts with WARC-KIT and how WARC-KIT utilizes both WARCFilter and the Linear Hash Table to accomplish a WARC indexing operation. Not shown in the figure is that the user can also programmatically interact with the Express and GraphQL servers through their own HTTP requests. Additionally users can interact directly with the GraphQL server through the GraphiQL interface.

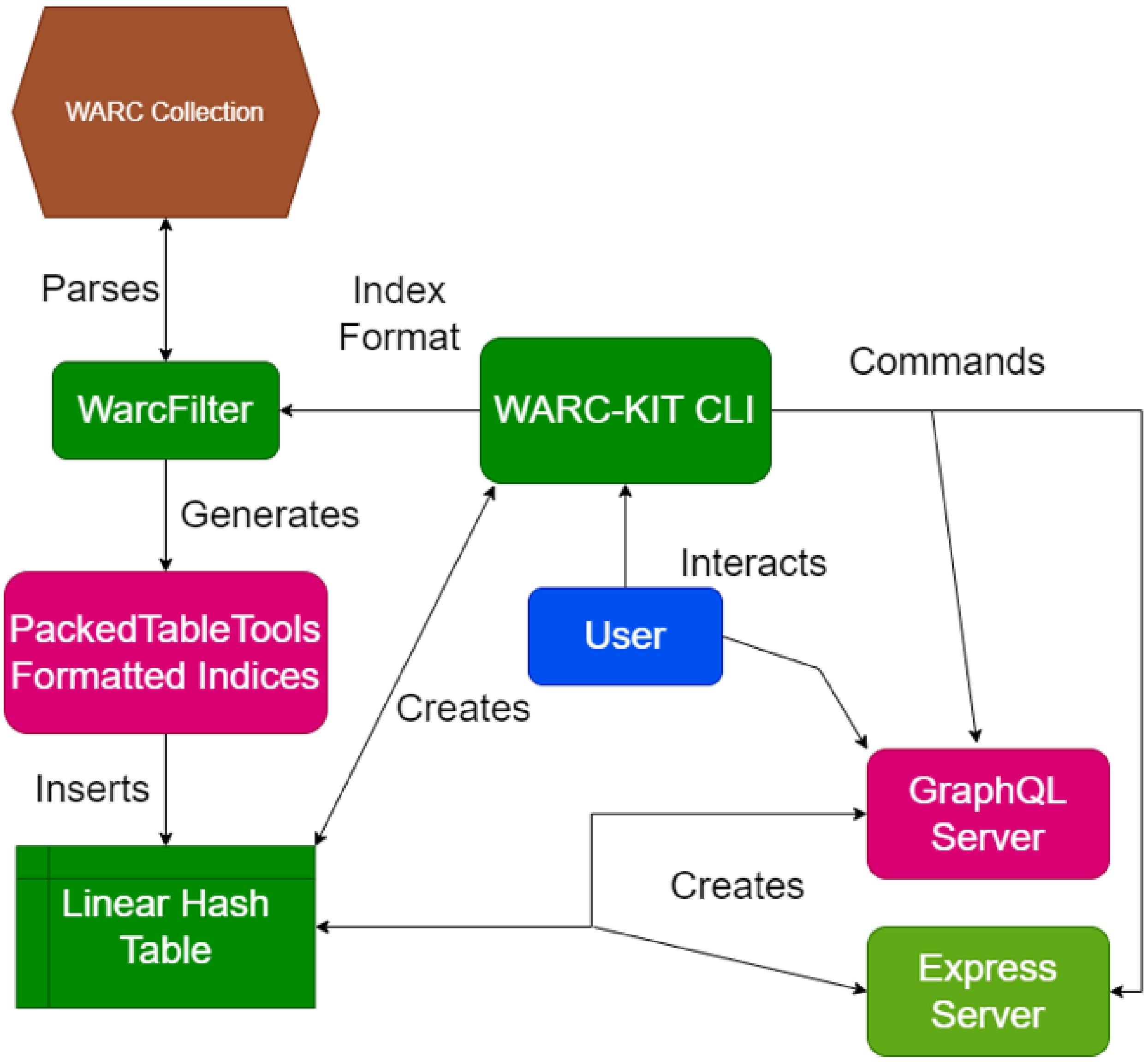
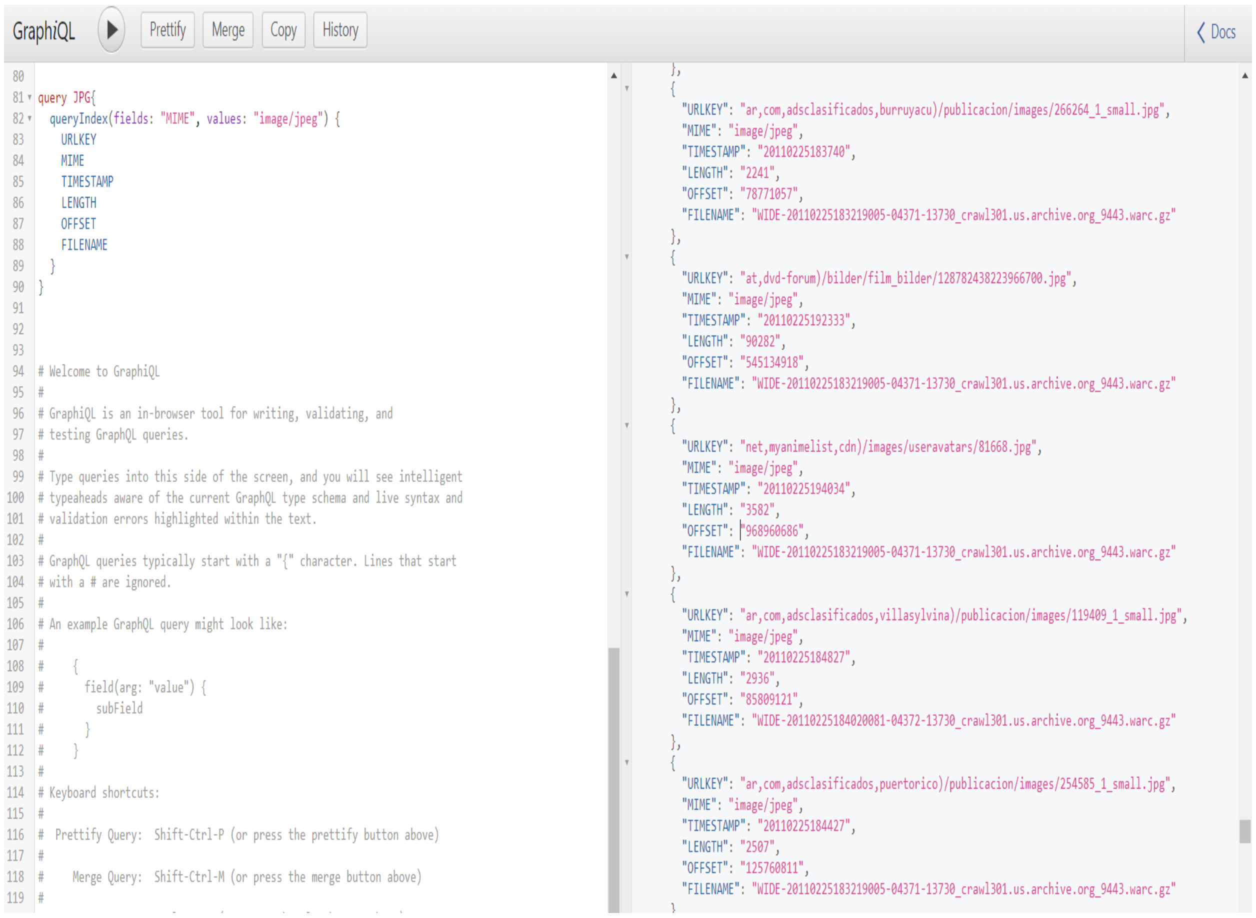


Figure 3.6: Flow chart of a WARC file collection indexing operation using WARC-KIT.

In Figure, 3.7 we display GraphiQL query page of a indexed WARC collection. Besides using GraphiQL, users are also able to programmatically use a GraphQL formatted HTTP request to receive the same output shown here. With the main function of WARC-KIT implemented we have successfully created an JavaScript application that allows users to manipulate their WARC file collections. While WARC-KIT's main function is to index and query WARC collections, it's composing applications WARCFilter and the Linear Hash Table database can also be used independently. The code for WARC-KIT can be found on the web page documenting this project [31].



```
query JPG{
  queryIndex(fields: "MIME", values: "image/jpeg") {
    URLKEY
    MIME
    TIMESTAMP
    LENGTH
    OFFSET
    FILENAME
  }
}



# Welcome to GraphiQL
#
# GraphiQL is an in-browser tool for writing, validating, and
# testing GraphQL queries.
#
# Type queries into this side of the screen, and you will see intelligent
# typeaheads aware of the current GraphQL type schema and live syntax and
# validation errors highlighted within the text.
#
# GraphQL queries typically start with a "{" character. Lines that start
# with a # are ignored.
#
# An example GraphQL query might look like:
#
#     {
#       field(arg: "value") {
#         subField
#       }
#     }
#
# Keyboard shortcuts:
#
#  Prettify Query:  Shift-Ctrl-P (or press the prettify button above)
#
#     Merge Query:  Shift-Ctrl-M (or press the merge button above)
#
```

```
      },
      {
        "URLKEY": "ar,com,adsclasificados,burruyacu)/publicacion/images/266264_1_small.jpg",
        "MIME": "image/jpeg",
        "TIMESTAMP": "20110225183740",
        "LENGTH": "2241",
        "OFFSET": "78771057",
        "FILENAME": "WIDE-20110225183219005-04371-13730_crawl301.us.archive.org_9443.warc.gz"
      },
      {
        "URLKEY": "at,dvd-forum)/bilder/film_bilder/128782438223966700.jpg",
        "MIME": "image/jpeg",
        "TIMESTAMP": "20110225192333",
        "LENGTH": "90282",
        "OFFSET": "545134918",
        "FILENAME": "WIDE-20110225183219005-04371-13730_crawl301.us.archive.org_9443.warc.gz"
      },
      {
        "URLKEY": "net,myanimelist,cdn)/images/useravatars/81668.jpg",
        "MIME": "image/jpeg",
        "TIMESTAMP": "20110225194034",
        "LENGTH": "3582",
        "OFFSET": "968960686",
        "FILENAME": "WIDE-20110225183219005-04371-13730_crawl301.us.archive.org_9443.warc.gz"
      },
      {
        "URLKEY": "ar,com,adsclasificados,villasylvina)/publicacion/images/119409_1_small.jpg",
        "MIME": "image/jpeg",
        "TIMESTAMP": "20110225184827",
        "LENGTH": "2936",
        "OFFSET": "85809121",
        "FILENAME": "WIDE-20110225184020081-04372-13730_crawl301.us.archive.org_9443.warc.gz"
      },
      {
        "URLKEY": "ar,com,adsclasificados,puertorico)/publicacion/images/254585_1_small.jpg",
        "MIME": "image/jpeg",
        "TIMESTAMP": "20110225184427",
        "LENGTH": "2507",
        "OFFSET": "125760811",
        "FILENAME": "WIDE-20110225183219005-04371-13730_crawl301.us.archive.org_9443.warc.gz"
      }
```

Figure 3.7: The GraphiQL Query page result of a JPG query on a WARC-KIT GraphQL server.

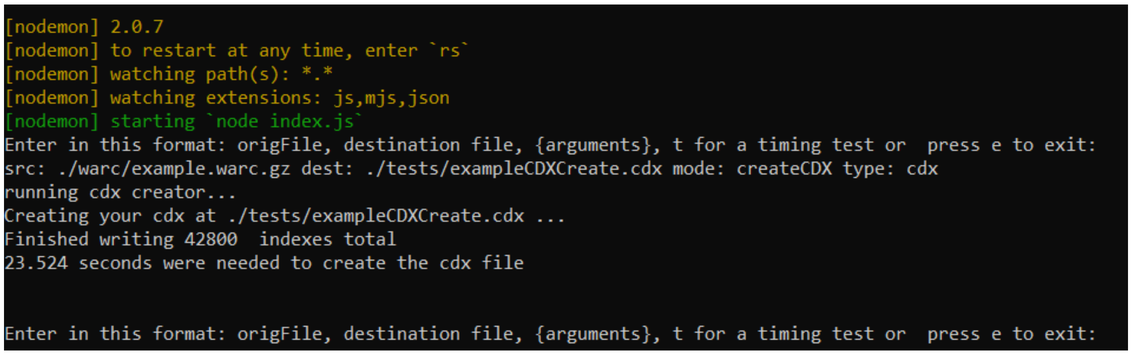
4. Experiments

In this chapter we demonstrate each individual tool and the overall function of WARC-KIT with various experiments. We also conduct experiments that compare and contrast with other tools with similar functions to WARC-KIT. All experiments in this chapter are run on a 2018 Dell G7 15 laptop.

4.1 WARCFilter Experiments

In this section we show off the stand alone WARCFilter tool to parse and filter out WARC records, generate new WARC files, create CDX file indices, and generate a webgraph dataset using the online Common Core datasets.

We first begin by showing of the CDX creation mode. In Figure, 4.1 we show an example of passing in a single WARC file and show the time it takes to create a simple CDX file. Average time to create a normal sized CDX file is around 24 seconds. WARCFilter also allows for argument chaining which lets users process multiple WARC files in one command.



```
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Enter in this format: origFile, destination file, {arguments}, t for a timing test or  press e to exit:
src: ./warc/example.warc.gz dest: ./tests/exampleCDXCreate.cdx mode: createCDX type: cdx
running cdx creator...
Creating your cdx at ./tests/exampleCDXCreate.cdx ...
Finished writing 42800  indexes total
23.524 seconds were needed to create the cdx file


Enter in this format: origFile, destination file, {arguments}, t for a timing test or  press e to exit:
```

Figure 4.1: An example of using WARCFilter's CLI program to create a CDX file from a WARC file.

The advantage of creating a CDX file is not fully exhibited until we demonstrate parsing on larger datasets. For this next example, we demonstrate the filtering of WARC records from a small dataset of WARC files from 2011 saved on the Internet Archive [30]. We use 5 WARC files from the dataset numbering 238,000 WARC records generated using the Internet Archive's Heritix web crawler. We then generate 5 CDX files on the 5 WARC files to run a filter operation. In this experiment we are simply are looking for WARC records that contain image files. We run WARCFilter in *warc* and *cdx* mode looking for records with JPG and PNG file type images. In table 4.1, we can see the results running WARCFilter with the image argument criteria on these files. As we can see from the results, CDX files vastly speed up the time it takes to retrieve desired WARC records.

WARC File Name	Total # of records	WARC Filter Time	CDX Filter Time
WIDE-20110225183219005-04371	42800	38.716 sec.	0.938 sec
WIDE-20110225184020081-04372	57557	39.655 sec.	1.876 sec.
WIDE-20110225210142891-04382	43129	37.162 sec.	2.94 sec.
WIDE-20110225215415804-04385	44646	37.456 sec.	1.812 sec.
WIDE-20110225221304846-04388	50493	39.367 sec.	3.099 sec.

Table 4.1: Comparison of the time WARCFilter takes to parse and filter WARC files directly versus parsing their CDX files.

In this next experiment, we show off a Node.js package known as *node-warc* which acts as a JavaScript library to parse WARC Records [11]. We run *node-warc* on the same WARC dataset for the experiment. The difference between *node-warc* and WARCFilter is that *node-warc* does not have any filter functions. This means *node-warc* simply reads through a WARC file without much parsing. As shown in table 4.2 *node-warc* performs well when parsing through just WARC records. However, on closer inspection how *node-warc* parses WARC records *node-warc* takes a shortcut by only parsing the WARC header of a record and then packaging subsequent WARC content block in a buffer object without parsing. This explains the fast parse time of *node-warc* as much of the parsing time of WARCFilter is spent on parsing both the WARC header and content block to search for filter criteria. This searching results in WARCFilter's much longer parse time. Thus while *node-warc* provides fast parsing WARC files, it lacks the filtering capabilities of WARCFilter.

Table 4.2: Comparison of the time *node-warc* takes to parse through WARC file Records

WARC File Name	node-warc parse time
WIDE-20110225183219005-04371	14.187 sec.
WIDE-20110225184020081-04372	16.163 sec.
WIDE-20110225210142891-04382	14.859 sec.
WIDE-20110225215415804-04385	16.187 sec.
WIDE-20110225221304846-04388	14.919 sec.

The last standalone demonstration of WARCFilter is the generation of webgraph datasets. Running WARCFilter in *genCCWebGraph* mode we pass in a Common Crawl *wat.paths* file as the source file. This mode requires an active internet connection as it will download and parse the corresponding online WAT file simultaneously. New Common Crawl datasets are generated every month. We opted to generate a dataset from the years 2015 - 2020 aiming for datasets generated in or around November. We parse 100 WAT files per Common Crawl paths file starting at offset 100 into the paths file. Results are shown in the table 4.3 below. One thing to note is the November 2015 dataset has double the number of generated vertices compared to the other datasets. This is because it was run before unique URL filtering was applied. When unique URL filtering is specified as an argument, edges whose vertices have the same base URL are not added to the dataset. Meaning a Google.com web page connected with a Google.com search result web page will not be added as they share the same base URL.

Table 4.3: Table showing the time and number of edges generated for each Common Crawl dataset paths file.

Common Crawl WAT datasets	# Edges Generated	Generation time
cc-nov-2015-wat.paths	188,721,679	4219.540 sec.
cc-dec-2016-wat.paths	120,007,747	4268.837 sec.
cc-nov-2017-wat.path	81,434,921	4085.955 sec.
cc-nov-2018-wat.path	75,781,930	4303.162 sec.
cc-nov-2019-wat.path-04385	81,100,750	4135.753 sec.
cc-nov-dec-2020-wat.path	57,240,145	3818.900 sec.

As we can see the results in the table above, generation time for a webgraph dataset takes on average when filtering for unique base URLs around one hour and ten minutes per dataset generation. Common Crawl WAT files are around 500 MB each. Each WAT File is downloaded then thrown away one by one to save hard drive space. Around 250 GB of WAT files were downloaded for the generation of these datasets. Users of this WARCFilter mode should beware if they have a internet data cap.

These graph files follow the standard webgraph dataset format. Meaning algorithms or programs that parse and analyze webgraph datasets will also work on these generated datasets. An example community detection webgraph created from the generated Common November 2015 webgraph dataset is shown in Figure 4.2.

Figure 4.2: A community detection graph produced from a Webgraph dataset generated from the WARCFilter tool [1].

4.2 Database Experiments

In this section we show off the stand alone Linear Hash Table database we created for WARC-KIT. Expectations are that since JavaScript is a web language with no direct file access even with Node.js, file operations will be slow compared to other languages.

The first experiment consists of a pure insertion test of a number of simple string key-value pairs. We test two configurations of the Linear Hash Table database one configured with 256 initial buckets and one with 1024 initial buckets. From the results in Figure 4.3, we see that initial bucket configuration of a Linear Hash Table makes a large impact on performance as the number of insertions and pairs in the table increases. This is because the bucket splitting operation to add a bucket to the table is the most expensive operation for a Linear Hash Table. Even though the keys in only one bucket are rehashed during a split operation. The smaller 256 bucket database has to perform a significantly larger number of more split operations, leading to a performance decrease. As such at least for Linear Hash Tables, it is better to set a higher number of initial number of buckets to minimize split operations.

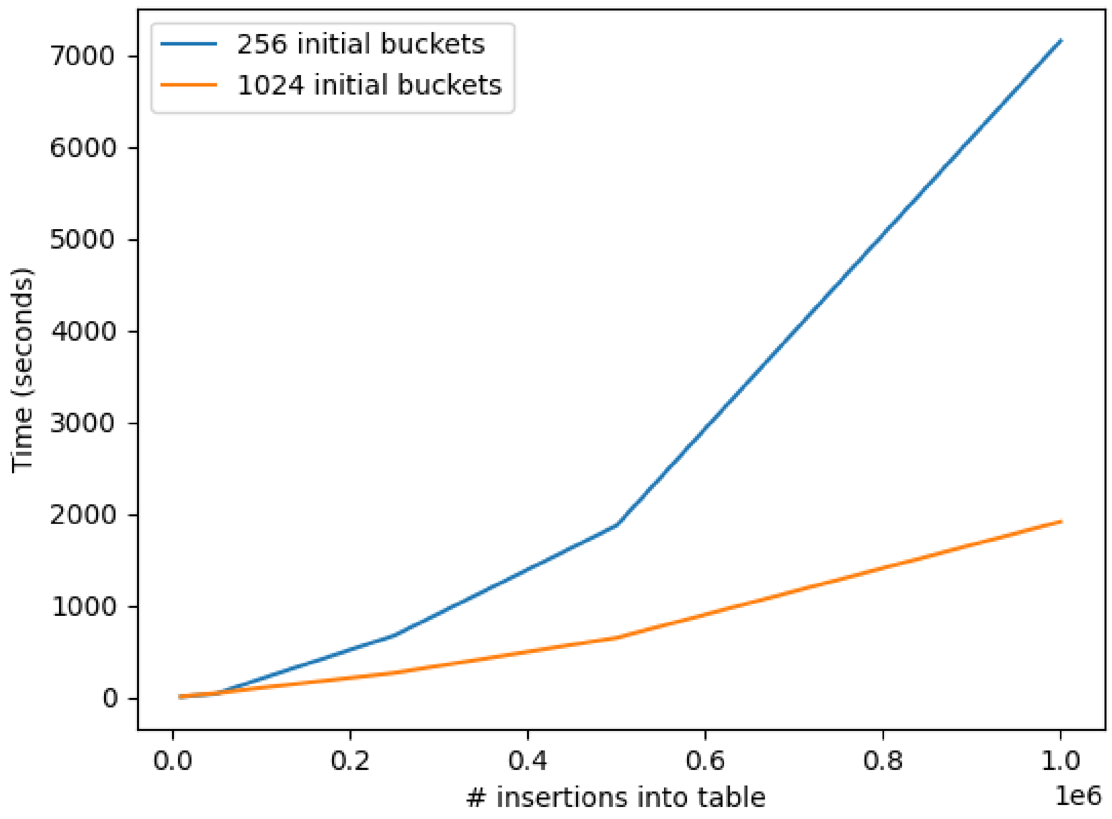


Figure 4.3: Comparison of insertion runtime between a Linear Hash Table with initial number of bucket set to 256 buckets versus 1024 buckets.

Next we perform a pure get test by retrieving all values inserted after n insertions. For this test we also wanted to compare our Linear Hash Table implementation with a similar system. Luckily there is a Linear Hash Table implementation in Rust as well [24]. Rust is a relatively new programming language invented in 2010 and is touted as having performance comparable to C++ [28]. However the Rust Linear Hash Table implementation has some drawbacks that affect performance. First the Rust table has oddly inconsistent insertion times for key-value pairs but consistent get times. Second the table is hard coded at a currently unchangeable 32 initial number of buckets. Therefore we expect the current Rust Linear Hash Table to underperform in the following experiment. As we can see in table 4.4 there is linear growth for the retrieval times in our JavaScript implemented Linear Hash Tables with no performance differences between initial bucket configurations. Also as we predicted, the Rust Linear Hash Table implementation performance is underwhelming. Future work can include improving this Rust Linear Hash Table to have the same underlying implementation as our Linear Hash Table design. We predict that when given identical Linear Hash Table implementations the improved Rust implementation we outperform our JavaScript implementation.

Table 4.4: Get performance test between our JavaScript implemented Linear Hash Table vs a Rust implemented Linear Hash Table

Number of gets	LHT256 time	LHT1024 time	Rust LHT time
10,000 key-value gets	2.023 sec.	2.634 sec.	8.582 sec.
100,000 key-value gets	42.733 sec.	40.267 sec.	102.321 sec.

4.3 WARC-KIT Experiments

In this section, we show off WARC-KIT experiments including timing experiments and a comparison in functionality to Common Crawl's index server. Experiments will be done using the same WARC file dataset from the WARCFilter section.

First we show a simple experiment for the time WARC-KIT takes to create an Index on a WARC file collections of different sizes. Linear Hash Table configurations are set to an initial bucket number of 1024 buckets. The single WARC file test contains 42000 indices, the 3 file WARC test contains about 143,000 indices, and the combined 5 file WARC test contains 238,000 indices. We expect similar index creation and insertion times to the number of regular insertions in a Linear Hash Table. As shown in the simple table 4.5 the insertion times do indeed comparatively line up similar to the corresponding number of Linear Hash Table insertions. Also, important to note is that these creation operations only need to be done once as a one time setup.

Table 4.5: Time to create and insert PackedTableTools formatted indices into Linear Hash Table and create a GraphQL query server on a WARC file collection

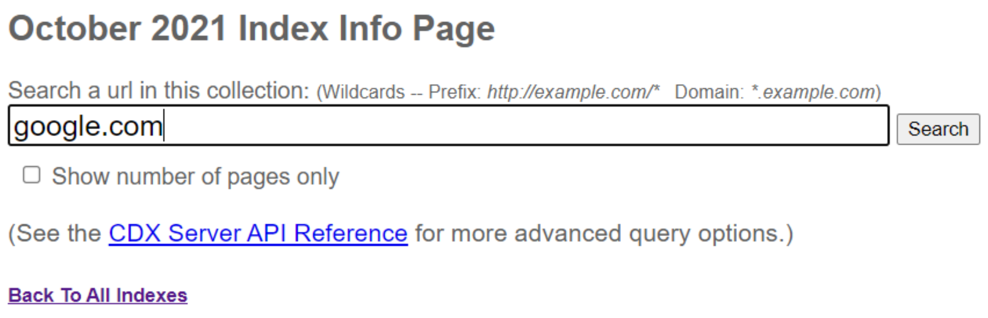
1 WARC file	3 WARC files	5 WARC files
39.741 sec.	156.639 sec.	238.414 sec.

For the next experiment, we display the timing tests of various queries on the GraphQL server through GraphiQL interface. The results are displayed in table 4.6 below. As expected the query results also sync closely to the Linear Hash Table get times. One thing interesting to note is that query times for a WARC file collections of the same size stays relatively consistent even if a query is more complex. Another interesting thing to discuss is that if the user knows the exact URL of the record they want the query is nearly instantaneous, even for large collections.

Table 4.6: Various Queries on the GraphQL query server created from a indexed WARC file collection.

Query Name	1 WARC file index	3 WARC file index	5 WARC file index
Single URL get	0.001 sec.	0.002 sec.	0.003 sec.
HTML Query	2.784 sec.	10.525 sec.	17.067 sec.
JPG Query	2.62 sec.	10.311 sec.	17.151 sec.
UK HTML Query	2.91 sec.	10.433 sec.	17.130 sec.
RU PNG Query	2.55 sec	10.312 sec.	17.007 sec

In Figure 4.4, the interface of Common Crawl's Index Server search is shown. Common Crawl's index server is a Python based program that allows users to search through their CDX collection of files. Similar to WARC-KIT, there is an advance query section to filter and output CDX indices. A contrast to WARC-KIT is that is the original WARC files are not interacted with in this application. Meaning a user must use a different application to download and retrieve the WARC records. Common Crawl's index server are also limited to the CDX file format that Common Crawl generates. While WARC-KIT offers custom PackedTableTools formatted indices that can be augmented to include user defined information.


October 2021 Index Info Page

Search a url in this collection: (Wildcards -- Prefix: *http://example.com/** Domain: **.example.com*)

google.com

(See the CDX Server API Reference for more advanced query options.)

Figure 4.4: The Common Crawl index server search page, which allows users to query their CDX files.

Thus concludes our experiments on the various features of WARC-KIT. WARCFilter itself works as a standard WARC parsing and filtering tool. As a Node.js package, *node-warc* provides fast traversal of WARC files but lacks the filtering and content block parsing that WARCFilter offers. The Linear Hash Table database system offers the first of its kind a on disk JavaScript document store. When compared to a unoptimized Rust Linear Hash Table implementation our JavaScript Linear Hash Table database outperforms the former. However, while Linear Hash Tables are dynamic in their growth, performance is significantly impacted by the number bucket splits. WARC-KIT is a combination of the previous tools that allows for custom index format creation and more informative queries on WARC file collections. A system with similar functionality is the Common Crawl Index server. However, the Common Crawl index is constrained by the CDX format and does not act upon the original WARC file collection compared to WARC-KIT.

5. Conclusion

Few tools exist to allow users to manage their own WARC files collections. Even fewer of those tools are created in JavaScript which with NPM has a large open source community of packages waiting to be explored. Thus WARC-KIT is one of the few applications capable of working with with WARC files in JavaScript. The separate parts of WARC-KIT such as WARCFilter can be used as an independent tool to manipulate regular WARC files and their CDX index files. WARCFilter can also be used to generate webgraph datasets from existing Common Crawl file collections. The Linear Hash Table database is the first JavaScript on disk document store of it's kind. The Linear Hash Table database can also be used independently as a database application for users that desire a wholly in JavaScript database solution. The overall WARC-KIT application itself allows users to create informative indices on WARC file records. Users can then query the database using a GraphQL server.

Future work can be done to improve the various functions of WARC-KIT. For example, WARCFilter can be improved to allow for the generation of customizable datasets other than webgraph datasets from Common Crawl's online WARC file collections. Another improvement would be the ability to interface with other online WARC resources such as the Internet Archive's collection. For the Linear Hash Table database, improvements could be made to lessen the performance impact of bucket splitting operations. Another improvement could also be made on the PackedTableTools format itself, by improving upon the size efficiency of packing values into a binary string. For WARC-KIT itself, the GraphQL server schema could be improved to handle even more complex queries. Another improvement is the to have better generation of PackedTableTools formatted objects from WARC records. The additional information from a WARC record content block can be metadata such a community tags in video WARC records.

WARC-KIT currently stands as one of the few JavaScript applications that lets users work with their personal WARC file collections. Whether it's users find use in its standalone components or the sum of its applications, it exists as a beneficial tool to assist in the preservation in the World Wide Web.

References

[1] Panchal Akshar. *Overlapping Community Detection in Social Networks.* San Jose State University, 2021.

[2] *CDX and DAT Legend.* URL: https://archive.org/web/researcher/cdx_legend.php (visited on 11/10/2021).

[3] *Common Crawl Data.* URL: https://commoncrawl.org/the-data/get-started/ (visited on 11/14/2021).

[4] Ryan Deschamps, Samantha Fritz, Jimmy Lin, Ian Milligan, and Nick Ruest. "The cost of a WARC: Analyzing web archives in the cloud". In: *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL).* IEEE. 2019, pp. 261–264.

[5] *Dynamic hashing technique of Berkeley DB.* URL: https://titanwolf.org/Network/Articles/Article?AID=9823fa36-325a-40bc-99bc-8f6e0173be50 (visited on 11/14/2021).

[6] *GraphQL.* URL: https://graphql.org/ (visited on 11/14/2021).

[7] *HarperDB.* URL: https://harperdb.io/ (visited on 11/14/2021).

[8] Bowers J., Stanton C., and Zittrain J. *What the ephemerality of the Web means for your hyperlinks.* URL: https://www.cjr.org/analysis/linkrot-content-drift-new-york-times.php (visited on 11/14/2021).

[9] *Locutus.* URL: https://locutus.io/ (visited on 11/14/2021).

[10] *Microsoft ODBC.* URL: https://docs.microsoft.com/en-us/sql/odbc/microsoft-open-database-connectivity-odbc?view=sql-server-ver15 (visited on 11/19/2021).

[11] *node-warc: Parse Web Archive (WARC) files or create WARC files.* URL: https://github.com/N0taN3rd/node-warc (visited on 11/14/2021).

[12] *Node.js About.* URL: https://nodejs.org/en/about/ (visited on 11/14/2021).

[13] *Node.js Hash Performance.* URL: https://github.com/hex7c0/nodejs-hash-performance (visited on 11/14/2021).

[14] *Node.js:fs-extra.* URL: https://www.npmjs.com/package/fs-extra (visited on 11/14/2021).

[15] *PHP Pack.* URL: https://www.php.net/manual/en/function.pack.php (visited on 11/13/2021).

[16] *PouchDB.* URL: https://pouchdb.com/ (visited on 11/14/2021).

[17] Paul R. *A behind-the-scenes look at LinkedIn's mobile engineering.* 2012. URL: https://arstechnica.com/information-technology/2012/10/a-behind-the-scenes-look-at-linkedins-mobile-engineering/2/ (visited on 11/15/2021).

[18] Enbody R.J. and Du H.C. “Dynamic hashing schemes”. In: *ACM Computing Surveys* 20.2 (1998), pp. 850–113.

[19] *Remove Node Modules*. URL: https://www.npmjs.com/package/remove-node-modules (visited on 11/14/2021).

[20] Adi Robertson. *Link rot in 2012: keeping track of how web addresses go dead*. May 15, 2012. URL: https://www.theverge.com/2012/5/15/3021913/chesapeake-digital-preservation-group-link-rot-report (visited on 11/10/2021).

[21] *RocksDB*. URL: http://rocksdb.org/ (visited on 11/12/2021).

[22] Nick Ruest, Samantha Fritz, Ryan Deschamps, Jimmy Lin, and Ian Milligan. “From archive to analysis: accessing web archives at scale through a cloud-based interface”. In: *International Journal of Digital Humanities* (2021), pp. 1–20.

[23] Nick Ruest, Jimmy Lin, Ian Milligan, and Samantha Fritz. “The archives unleashed project: technology, process, and community to improve scholarly access to web archives”. In: *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020*. 2020, pp. 157–166.

[24] *Rust Linear Hash Table implmentation*. URL: https://github.com/samrat/rust-linhash (visited on 11/17/2021).

[25] *Stanford Web Archiving Tutorials and Resources*. URL: https://library.stanford.edu/projects/web-archiving/research-resources/tutorials-and-examples (visited on 11/15/2021).

[26] *The CDX File Format (2015)*. URL: https://iipc.github.io/warc-specifications/specifications/cdx-format/cdx-2015/ (visited on 11/10/2021).

[27] *The Internet Archive*. URL: https://archive.org/about/ (visited on 11/12/2021).

[28] *The Rust Programming Language*. URL: https://www.rust-lang.org/ (visited on 11/17/2021).

[29] Xinyue Wang and Zhiwu Xie. “Web archive analysis using hive and SparkSQL”. In: *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE. 2019, pp. 424–425.

[30] *WARC file dataset*. URL: https://archive.org/download/testWARCfiles (visited on 11/14/2021).

[31] *WARC-KIT Code*. URL: https://www.cs.sjsu.edu/faculty/pollett/masters/Semesters/Spring21/david/WARC_KIT_Code.html.

[32] *WARC, Web ARChive file format*. URL: https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.0/#warc-record-types (visited on 11/10/2021).

[33] *WARCFilter github*. URL: https://github.com/bbdavidbb/warcfilter (visited on 11/14/2021).

[34] *Yioop: Open Source Search Engine Software*. URL: https://www.seekquarry.com/ (visited on 11/10/2021).