Efficient Web Archive Searching

Authors

Ming Cheng

Lin Zhang

Yijing Wu

Xiaolin Zhou

Jinyang Li

CS4624: Multimedia, Hypertext, and Information Access

Instructor: Dr. Edward Fox

Client: Mr. Xinyue Wang

April 6, 2020

Blacksburg, Virginia

Efficient Web Archive Searching

(ABSTRACT)

The field of efficient web archive searching is at a turning point. In the early years of web archive searching, the organizations only use the URL as a key to search through the dataset, which is inefficient but acceptable. In recent years, as the volume of data in web archives has grown larger and larger, the ordinary search methods have been gradually replaced by more efficient searching methods. This report will address the theoretical and methodological implications of choosing and running some suitable hashing algorithms locally, and eventually to improve the whole performance of web archive searching in time and space complexity. The report introduces the design and implementation of various hashing algorithms to convert URLs to a sortable and shortened format, then chooses the best one by comparing benchmark results.

TBD

Acknowledgments

Xinyue Wang

Client

Virginia Tech

Blacksburg, VA 24061

xw0078@vt.edu

Office: Torgersen Hall 2030

Dr. Edward A. Fox

Supervisor

Virginia Tech

Blacksburg, VA 24061

fox@vt.edu

(540)-231-5113

Office: Torgersen Hall 2160G

Dr. Lenwood S. Heath

Consultant

Virginia Tech

Blacksburg, VA 24061

heath@vt.edu

(540)-231-4352

Office: Torgersen Hall 2160J

Contents

List of Figures

Chapter 1

Introduction

Before we begin to examine the various methods and approaches to web archive searching, it is essential to start by asking why we archive web materials at all. Nowadays, the web is an important cultural resource as the venue for a large amount of social interaction and many cultural productions, and the forum for much of the public debate. The web plays an important role in society and people's lives. Therefore, the web has become a key source for scholars studying social and cultural phenomena. Web archives facilitate this work by documenting and preserving snapshots of the web for future research [2].

As a result, many websites have been collected by companies and organizations. Most of them are non-profit groups, aiming to create a public web library and provide a free platform for online users to conduct research and data analysis. In the meantime, one thing we are gradually facing is web-mutability. The web is an unpredictable medium. The pace of change is rapid, and it can be hard to predict what types of new online content and services will become the next hit on the web. Since we want to create a library storing all historical versions of the websites, this large collection of data becomes enormously large as more and more data is gathered along the way, and leads to searching efficiency issues when looking it up. With more social scientists, historians and other researchers accessing these online archives more frequently, we can not simply reduce the size of the dataset but have to resolve this problem by bringing some methods to improve searching efficiency.

According to the Internet Archive Organization, the largest web archives dataset we have

now is approximately 50 PB, it contains 9.8 petabytes books/music/video collections, 18.5 petabytes unique data [5]. We will give more details here about web archive service by introducing a public website called "Wayback Machine". Wayback Machine is a non-profit digital library of internet sites and digital-formed cultural artifacts, whose main purpose is to provide universal access to all knowledge.

The Internet Archive service has been active for more than 20 years since 1996 and it already collected over 330 billion web pages, 20 million books and text, 4.5 million audio records. Anyone with a free account can upload media to the Internet Archive and they work with thousands of partners globally to save copies of their work into special collections [5].

Just like mentioned above, with the ever-increasing datasets, there seem to be demands for making some improvements regarding the searching efficiency. An idea of making the URLs shorten and sortable to reduce time and space complexity and increase the searching speed was brought out by our client Xinyue Wang.

We will be implementing several hashing algorithms to run locally which can shorten the URL and makes it sortable. TinyURL is a useful website to provide services that take an original URL and produce a shortened URL. However, we can not abuse such services with our huge collection of data. Therefore, our hashing process must be performed locally.

We can explain why shortened and sortable URLs could be really helpful for making progress in web archive searching. The reason why we need to make the URL shorter is quite simple, to increase the searching efficiency by reducing the factors we need to consider and check for each URL. For example, we have a 50 characters long URL compared to a 15 characters URL, it is obvious that the latter will be easier to be checked and searched. Also, it is necessary to have the URLs sortable because we need to improve the clustering of the datasets, thereby the values in the set are widely scattered, then enhance the searching efficiency.

Chapter 2

Requirements

The main purpose of our project is to find a method to convert URLs to a sortable and shortened format locally and provide corresponding design and implementation. A benchmark method will be provided to measure the searching efficiency improvement in terms of time and space complexity. We will also deliver multiple alternative hashing functions with their benchmarking results and detailed analysis to demonstrate the pros and cons of each method. Therefore, users can decide on which method to apply, based on their use cases.

TBD

Chapter 3

Design

3.1 Goals in Detail

To support both of the goals of (a) examining how different hash functions convert URLs and the time/space complexity during web archive searching, and (b) implementation of the method which converts URLs and reduces the time and space complexity during web archive searching, the system needs to support three tasks.

1. Extract from the sample dataset.
2. Apply hashing functions, which depend on the dataset.
3. Benchmark the functions; this is dependent on the hashing result values from the functions.

As shown in Figure 3.1, the tasks are dependent on one another. As a result, we can derive a sequence of tasks required to accomplish the goal; these are explained in the following subsections.

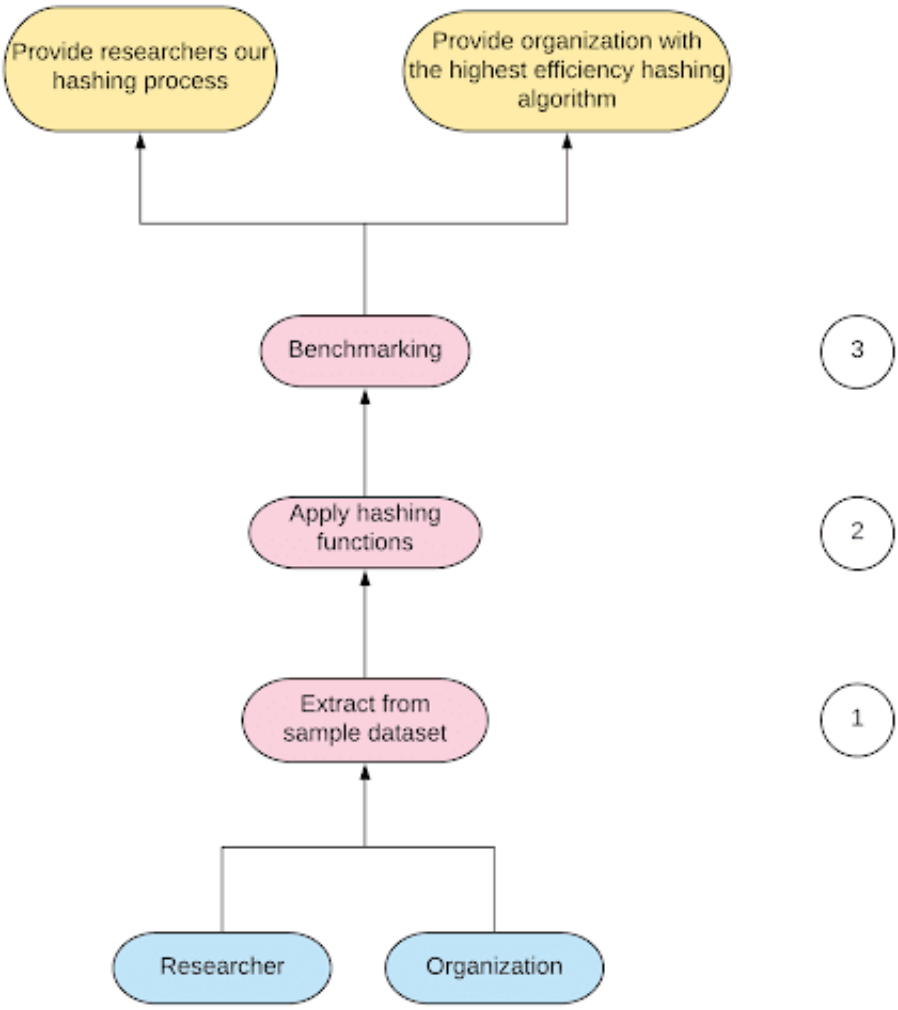

Figure 3.1: Project Goals and Tasks

3.2 Sub-tasks in Detail

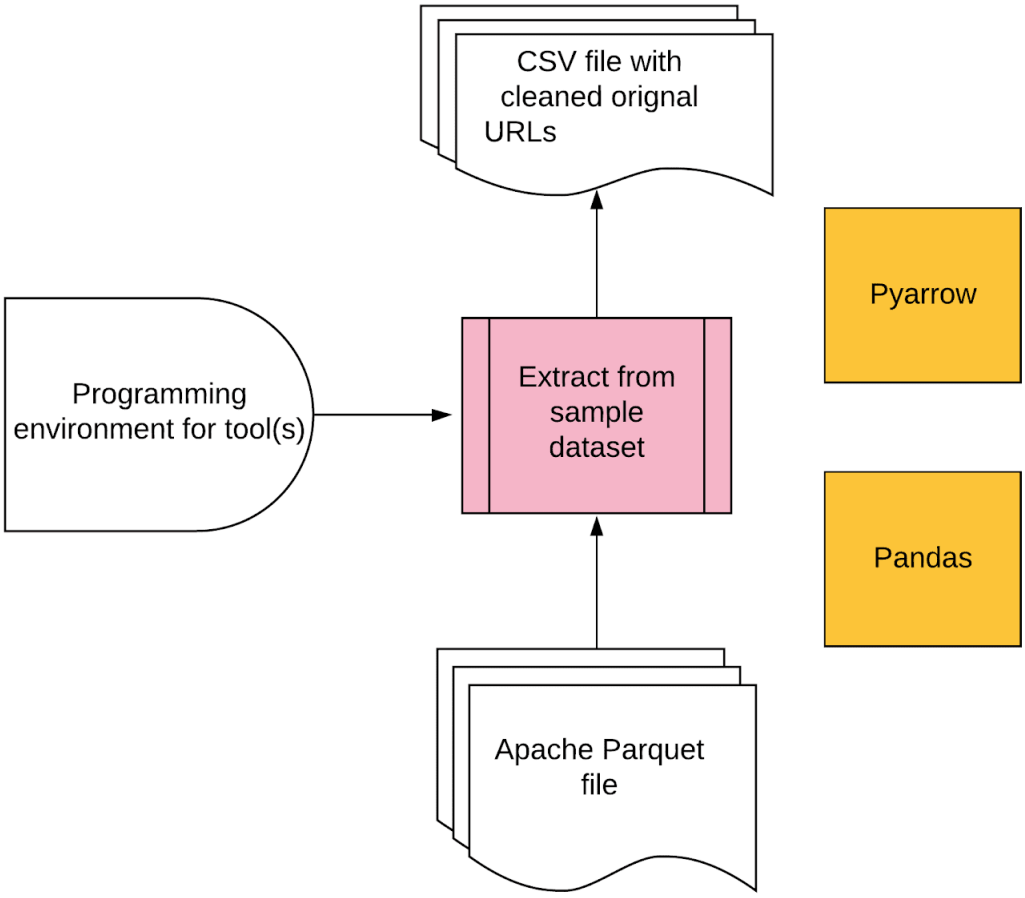

Figure 3.2: Data Extraction

Task: Data extraction (see Figure 3.2)

Input file: Sample data in Apache Parquet format

Output file: Extracted original URLs in CSV format

This task requires the Python API for Apache Arrow to extract information from the Apache Parquet file. It also requires Python data science libraries such as Pandas and Numpy to

process extracted data and send output to a CSV file for better accessibility and readability.

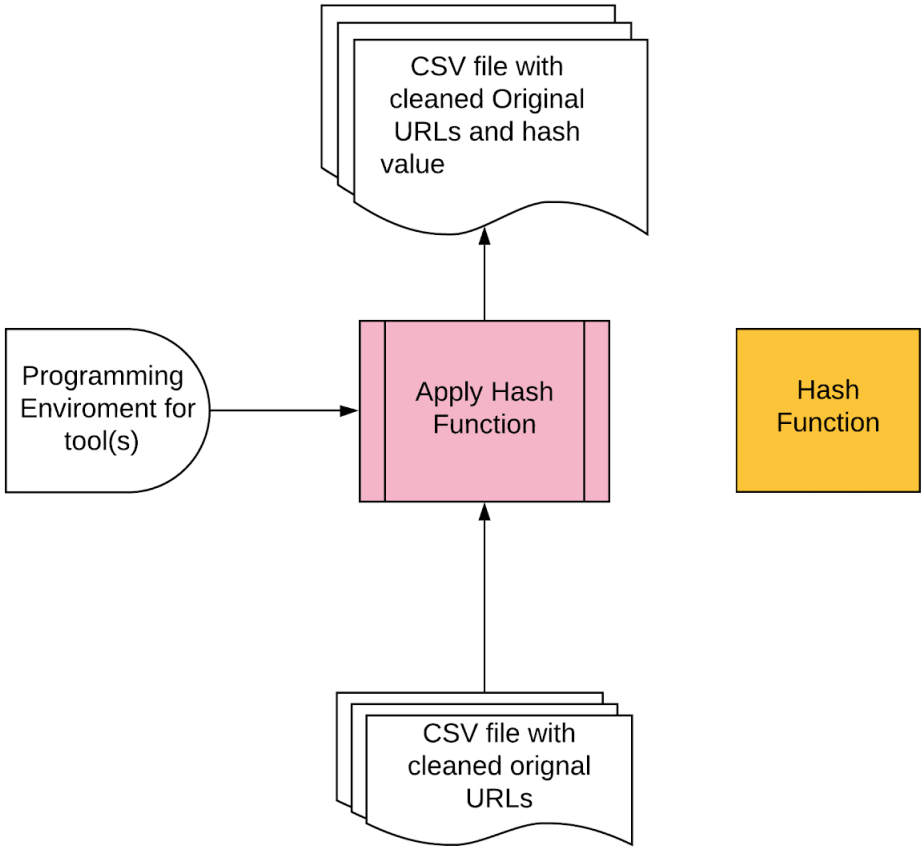

Figure 3.3: Apply Hash Function

Task: Apply hash function (see Figure 3.3)

Input file: Extracted original URLs in CSV format

Output file: CSV file with original URLs and hashed values

For the task of the apply hash function, the required input file is a CSV file with cleaned original URLs (URLs with no special characters). This CSV file is provided by the task of extracting from the sample dataset. The hash function takes the cleaned original URLs as input and outputs the hash values which will be written back to the CSV file. The libraries, functions, and the environment depend on which hash function we are using. For example, since the Spooky Hash function is written in C++, the required libraries are stdlib, iostream, stdio, etc.

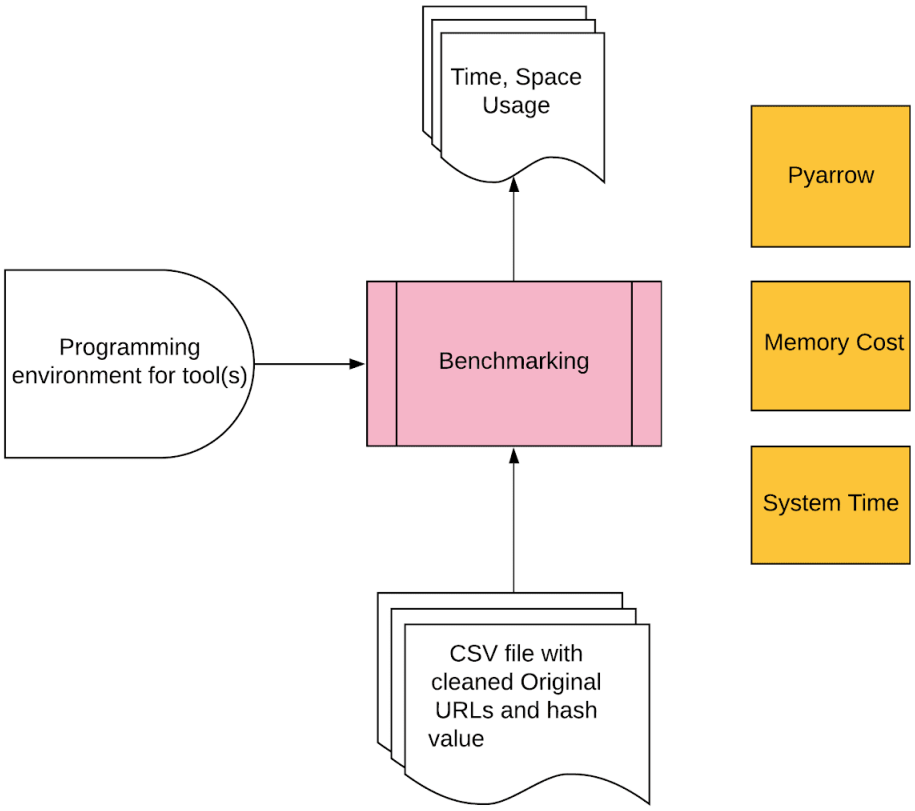

Figure 3.4: Benchmarking

Task: Benchmarking (see Figure 3.4)

Input file: CSV file with cleaned original URLs and hashed values

Output file: Execution time, Space usage

This task will take the CSV file with cleaned URLs and hash values from the previous task as an input. Then, Pyarrow and the Pandas library are used to generate a new Apache Parquet file. Next, we will call the Pyarrow build-in query function to search in the Apache Parquet file (Environment: Python). Meanwhile, we use the system time and the memory usage function to measure the query function execution time and memory usage for further comparison.

3.3 Workflow

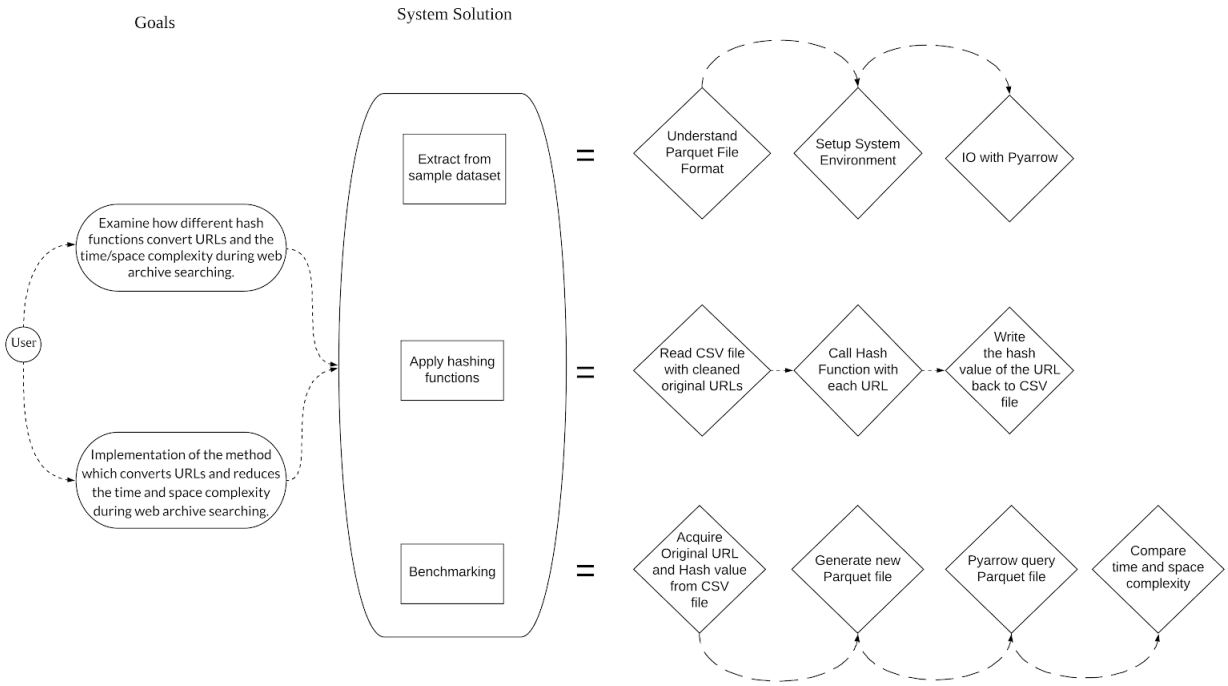

Figure 3.5: Project Workflow

Figure 3.5 summarizes our overall project workflow.

TBD

Chapter 4

Implementation

4.1 Spooky Hash

SpookyHash is the first Hash Function chosen in this project. SpookyHash is a public domain non-cryptographic hash function written in C++ producing well-distributed 128-bit hash values for byte arrays of any length. 32-bit and 64-bit hash values are supported as well by using only the bottom bit. Spooky Hash has two main advantages. One is that the hash function is fast; long keys hash in 3 bytes per cycle; short keys take about 1 byte per cycle, and there is a 30 cycle startup cost [4]. In cryptography, the avalanche effect is a desirable property that stands for a slight change, in either the key or the plain-text resulting in a significant change in the cipher-text [1]. The second advantage is that it can achieve avalanche for 1-bit and 2-bit input; it can also produce different hash values based on the seed [4].

This hash function takes a pointer of the stream of bytes to be hashed, length of the message, and a seed. The inner loop of Spooky Hash consumes 8 bytes of the input and uses xor, rotation, and addition to manipulate that input [4]. The hash values which are generated at the end are used to create short URLs for the later part of the project.

The web archive is known to be huge, so any associated database will be very large. One of the reasons why SpookyHash is chosen for this project is because its hash values' chances

of collision are small. According to the creator of Spooky Hash, libraries using 128-bit checksums should expect 1 collision once they hit 16 quintillion documents [4]. Since the short URL is created based on the hash values, the fact that the hash values have a low chance of collision means the short URLs will also have a low collision chance. This works well with web archive databases.

4.2 Simple Hash

Our second initial implementation is called Simple Hash. The basic idea of this algorithm was inspired by Dr. Lenwood Heath from the CS department at Virginia Tech. All URLs are structured with at least one protocol and one domain. A fully structured URL example is shown in Figure 4.1.

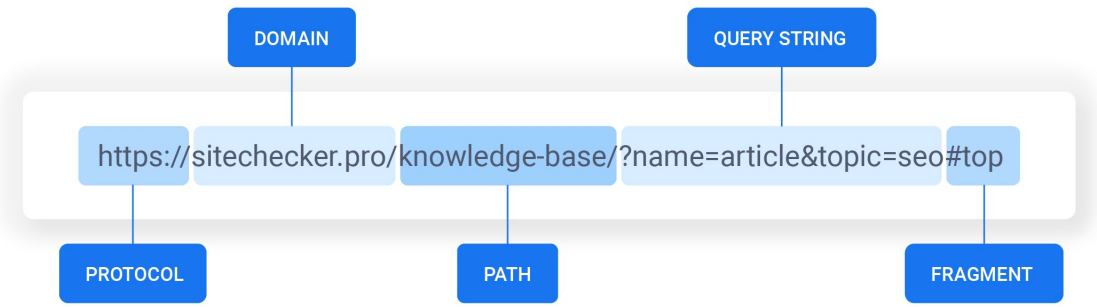

Figure 4.1: URL Structure Example [6]

According to our client, our goal is to not only shorten the URL but also make it sortable. Especially when it comes to looking for a specific URL in the dataset, without hashing the

server will need to look through the whole database to find the result. But if we can split the original URL into different categories, the server will be able to only look at one category instead of looking through the whole database.

Based on the requirement and our analysis, we made a decision on the first step we will do for the hashing: remove everything except protocol and domains. This step is presented in the diagram with the name "Protocol & Domain Only". For example, "https://vt.edu/", "https://vt.edu/innovationcampus/index.html" and "https://vt.edu/visit.html" will be shortened to "https://vt.edu". Without doing any hashing, this step already tremendously reduced the length of URLs.

Since we are planning to apply hashing on the URLs, special characters like dash and period aren't easy to deal with and most of them are meaningless, so the second step we did is to filter out all the special characters.

The core part of this hashing algorithm is the third step: Simple Hash. In our data set, there are only two kinds of protocols, "http" and "https". In order to shorten the URLs' length, we decided on using 0 and 1 to represent them. Since protocol can be used to divide URLs into subcategories, we are putting 0 and 1 as the first index for the hashed result. Other than protocols, as we know, the domain is structured by a sub-domain (optional), a domain name, and a top-level domain. An easy example would be "vt.edu". Here "vt" is the domain name, and "edu" is the top-level domain. There are other top-level domain names, for instance: .com, .org, .gov, etc. We realized that there are only about two thousand unique top-level domains, and most of them are more than two characters in length [3]. So we decided on hashing them into two bytes which could help to shorten the hash length. As for the method we used to hash the top-level domain into two bytes, we simply did an indexing hashing. Those two thousand top-level domains are listed and each of them has a unique index. We count numbers from 0 to 9, then letters a to z and capitalized letters A to Z; there are a

total of 62. Using them can accomodate 3844 unique strings which is more than the available number of top-level domain names. Each top-level domain is designated with a unique 2 byte string. This string will be appended next to the protocol's bit number since the top-level can divide URLs into the second-largest group other than protocols. Other than that, we just go ahead and append everything left of the tail of hashed URLs. See an example in Figure 4.2.

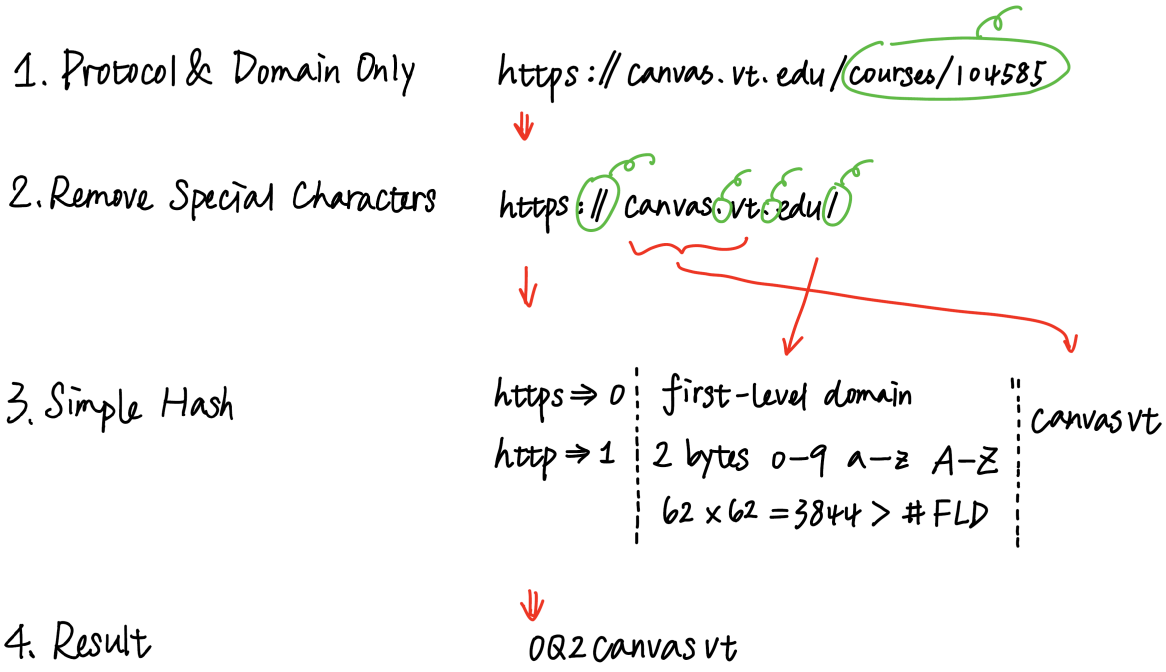

Figure 4.2: Simple Hash Process

Chapter 5

Testing/Evaluation/Assessment

5.1 Spooky Hash

Figure 5.1 compares the distribution of the original URL and SpookyHash Value in terms of character size. Since the number of SpookyHash values is much larger than the number of original URLs, the y-axis scale is rescaled to log base in order to display the skewness of the graph. Figure 5.1 shows that the original URL number is skewed to the right which means the data is more spread out in large URL/Value character size than in small URL/Value character. After the spooky Hash is applied to the original URL, the SpookyHash values it generates are concentrated between 25 and 50 character sizes. This shows that the overall character sizes of the original URL are greatly reduced after the hashing process.

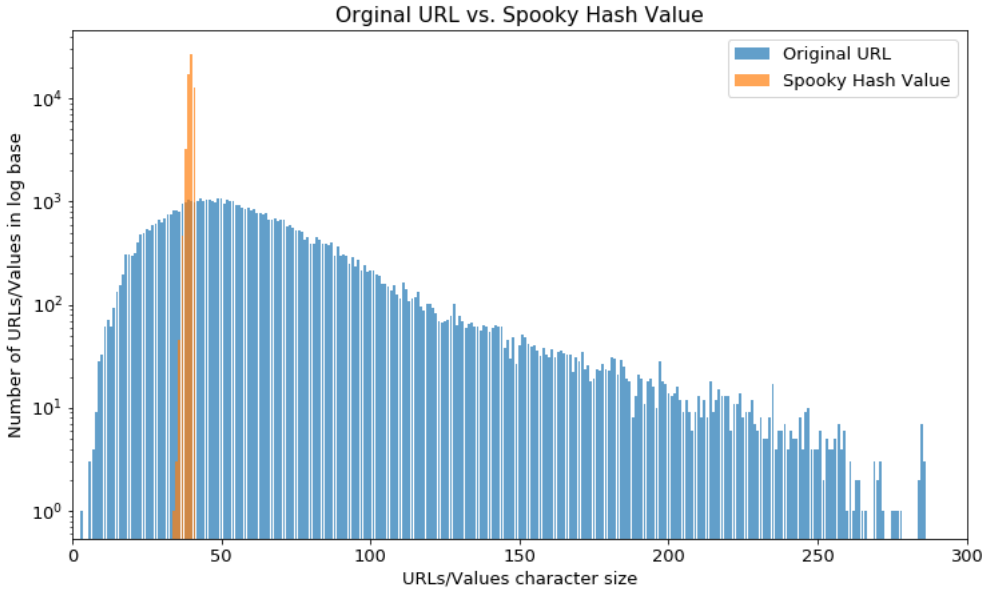

Figure 5.1: Original URLs and Spooky Hash Values Length Comparison

5.2 Simple Hash

Figure 5.2 compares the distribution of the original URL and Simple Hash Value in terms of character size. Figure 5.2 shows that the original URL number is skewed to the right which means the data is more spread out in large URL/Value character size than in small URL/Value character. After the Simple Hash is applied to the original URL, the SpookyHash values it generates are symmetrically concentrated between 0 and 40 character sizes whereas the majority of the original URLs have character size larger than 40. The data of the Simple Hash Value is located on almost the left side edge of the graph, compared to the original URL; this shows that the overall character sizes of the original URLs are tremendously reduced after the hashing process.

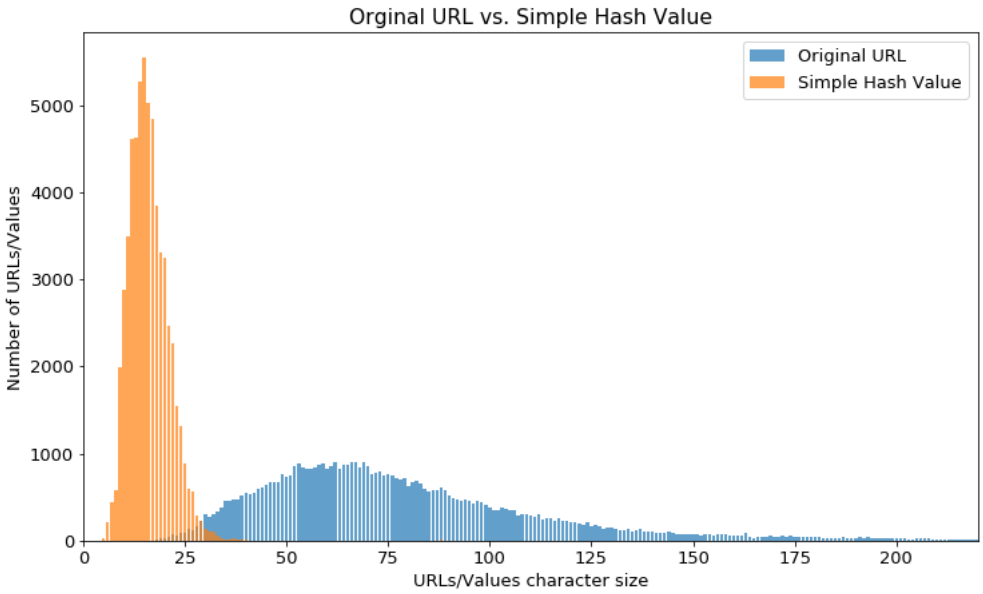

Figure 5.2: Original URLs and Simple Hash Values Length Comparison

TBD

Chapter 6

Users' Manual

6.1 User Experience

This section will go over how this will help the users, why it helps, and then go into detail on how a user can use this.

The implementation of our project will help users to shorten their URLs by using a specially designed hashing algorithm. Users can download a dataset and then search for a specific URL from the dataset.

TBD

Chapter 7

Developer's Manual

7.1 System Compatibility

Our project supports Windows, macOS, and various Linux distributions (including Ubuntu 16.04, Ubuntu 18.04).
A 64-bit system is required.

7.2 Installation and Setup

1. 64-bit version of Python 3.7 or higher (32-bit will lead to a PEP517 Error during the installation process).
2. Go to GitHub and clone the repository by following the link below
    - https://github.com/GBBKN/EWAS
    - If the link does not work, please contact our project contact person, Ming Cheng at ming98@vt.edu
3. Install requirements by using **python -m pip install -r python_requirement.txt**

- Window: Command Prompt
- Linux: Terminal
- macOS: Terminal

4. Run hashing
    - Spooky Hash
        (a) Go to SpookyHash_V2 directory
        (b) Run command:
            g++ -o Spooky.o SpookyV2.cpp Main.cpp
        (c) ./Spooky.o
    - Simple Hash
        (a) Go to SimpleHash directory
        (b) Run command:
            python simple_hash.py Dataset.csv
5. Benchmark: **TBD**

Chapter 8

Lessons Learned

- During the initial phase of implementing the design of our project, we first thought of using a two-way hashing algorithm. However, during the process of searching for information on how to use two-way hashing, we realized that this method does not fit for our project. After that, we went to Dr. Heath and asked him about the idea. He explained to us that this approach is not feasible. Because of the enormous size of the dataset, the result would be too much collision.
- We also tried various one-way hashing algorithms, and we found that lots of these algorithms will make URLs longer instead of shorter.
- **TBD**

Bibliography

[1] Avalanche effect in cryptography, Feb 2020. URL https://www.geeksforgeeks.org/avalanche-effect-in-cryptography/.

[2] Helge Holzmann, Vinay Goel, and Avishek Anand. Archivespark: Efficient web archive access, extraction and derivation. In *Proceedings of the 16th ACM/IEEE-CS on Joint Conference on Digital Libraries*, JCDL '16, page 83–92, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342292. doi: 10.1145/2910896.2910902. URL https://doi.org/10.1145/2910896.2910902.

[3] ICANN. List of top-level domains, Apr 2020. URL https://www.icann.org/resources/pages/tlds-2012-02-25-en.

[4] Robert John Jenkins Jr. Spookyhash: a 128-bit noncryptographic hash, 2020. URL http://www.burtleburtle.net/bob/hash/spooky.html.

[5] Internet Archive Organization. Internet archive: About ia, Apr 2020. URL https://archive.org/about/.

[6] Ivan Palii. Url address: Definition, structure and examples, Dec 2017. URL https://sitechecker.pro/what-is-url/.