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

May 11, 2020
Virginia Tech, Blacksburg, Virginia 24061

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 [6], 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 complexity.

The report introduces the design and implementation of various hashing algorithms to convert URLs to a sortable and shortened format, then demonstrates our findings by comparing benchmark results.

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 viii

1 Introduction 1

2 Requirements 3

3 Design 4

3.1 Goals in Detail . . . 4

3.2 Sub-tasks in Detail . . . 6

3.2.1 Data Extraction . . . 6

3.2.2 Hashing . . . 7

3.2.3 Benchmarking . . . 9

3.3 Workflow . . . 10

4 Implementation 12

4.1 SpookyHash . . . 12

4.2 Simple Hash . . . 13

4.3 xxHash . . . 15

4.4 MurmurHash . . . 16

4.5 cityhash . . . 17

5 Testing/Evaluation/Assessment 18

5.1 SpookyHash . . . 18

5.2 Simple Hash . . . 19

5.3 xxHash . . . 19

5.4 MurmurHash . . . 20

5.5 cityhash . . . 21

6 Benchmark & Conclusion **23**

7 Lessons Learned **25**

8 Future Works **26**

9 Users' Manual **27**

9.1 Pre-processing 27

9.2 SpookyHash & MurmurHash 27

9.3 Simple Hash 28

9.4 xxHash 28

9.5 cityhash 29

10 Developer's Manual **30**

10.1 System Compatibility 30

10.2 Installation and Setup 30

Bibliography **32**

List of Figures

3.1 Project Goals and Tasks 5
3.2 Data Extraction 6
3.3 Apply Hash Functions 7
3.4 Benchmarking 9
3.5 Project Workflow 11

4.1 URL Structure Example [15] 13
4.2 Simple Hash Process 15
4.3 Snippet of the MurmurHash3 16

5.1 Original URLs and SpookyHash Values Length Comparison 18
5.2 Original URL vs. Simple Hash Values's Size and Frequency 19
5.3 Original URL vs. xxHash Values's Size and Frequency 20
5.4 Original URL vs. MurmurHash Values's Size and Frequency 21
5.5 Original URL vs. cityHash Values's Size and Frequency 22

6.1 Benchmark Results 24

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. It is 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 [10].

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 and 18.5 petabytes of unique data [14]. We will give more details here about web archive services by introducing a public website called the "Wayback Machine". The Wayback Machine is part of a 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 [12]. Anyone with an account can upload media to the Internet Archive. They work with thousands of partners globally to save copies of their work into special collections [14].

As mentioned above, with the ever-increasing datasets, there seem to be demands for making some improvements regarding the searching efficiency [4]. An idea of making the URLs shorten and sortable to reduce time complexity and increase the searching speed was brought

out by our client Xinyue Wang.

We have been 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, if 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 so we can improve the organization of the datasets, and 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 a corresponding design and implementation. A benchmark method will be provided to measure the searching efficiency improvement in terms of time 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.

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 complexity during web archive searching, and (b) implementation of the method which converts URLs and reduces the time 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.

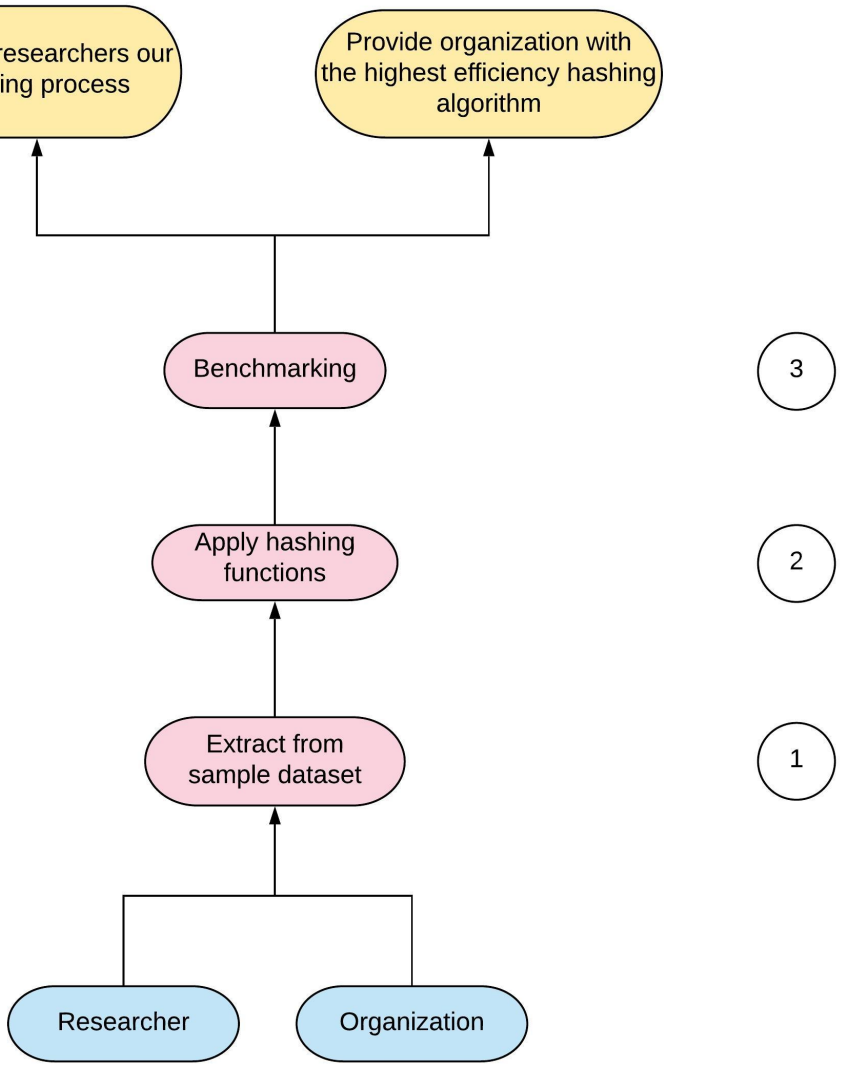

Figure 3.1: Project Goals and Tasks

3.2 Sub-tasks in Detail

3.2.1 Data Extraction

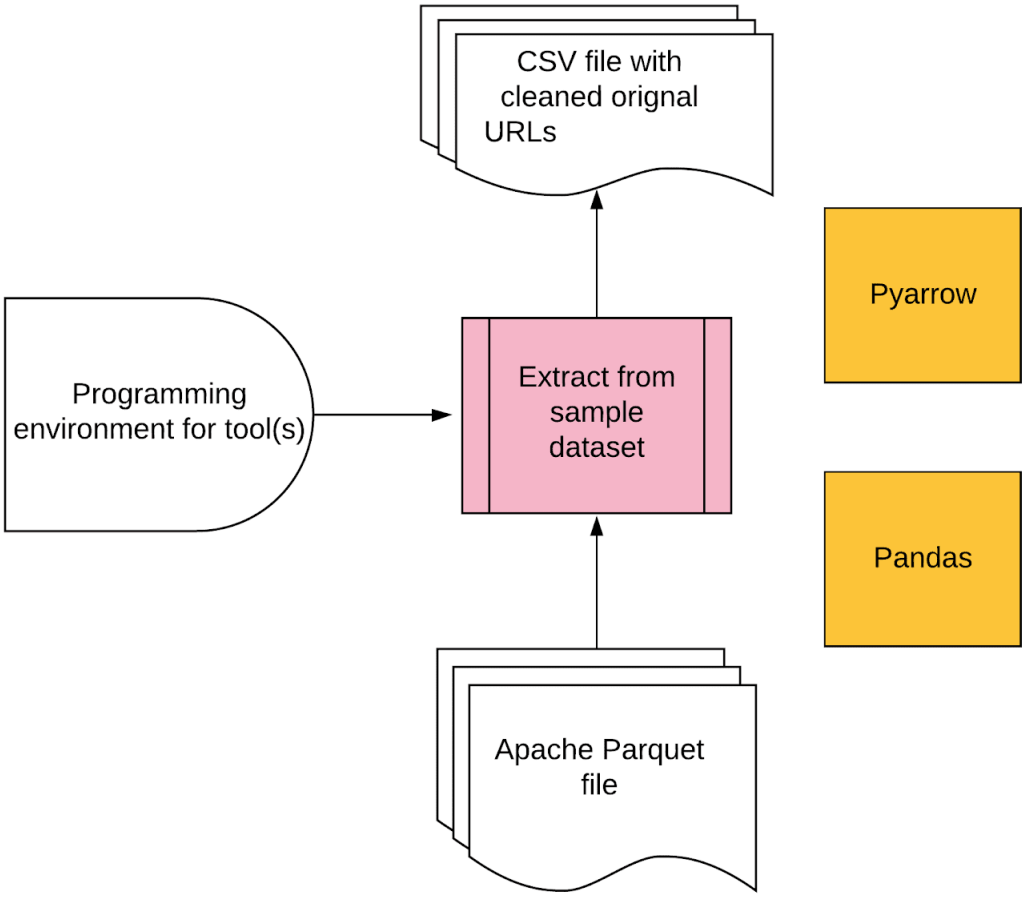

Figure 3.2: Data Extraction

- 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.

3.2.2 Hashing

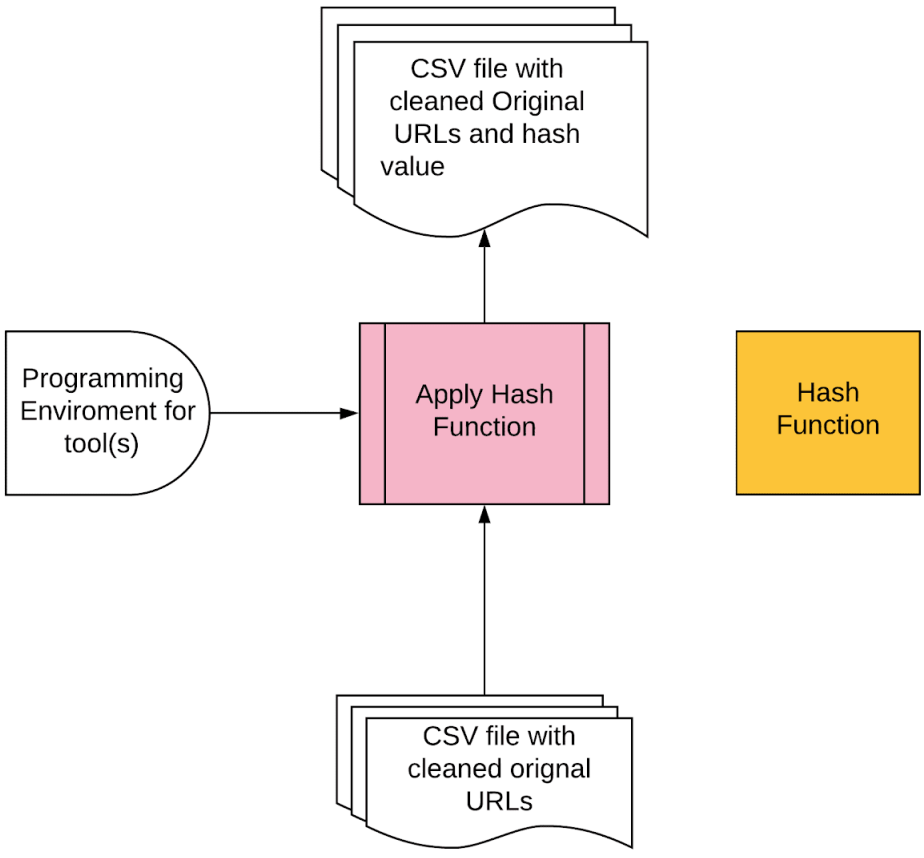

Figure 3.3: Apply Hash Functions

- 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 SpookyHash function is written in C++, the required libraries are stdlib, iostream, stdio, etc.

3.2.3 Benchmarking

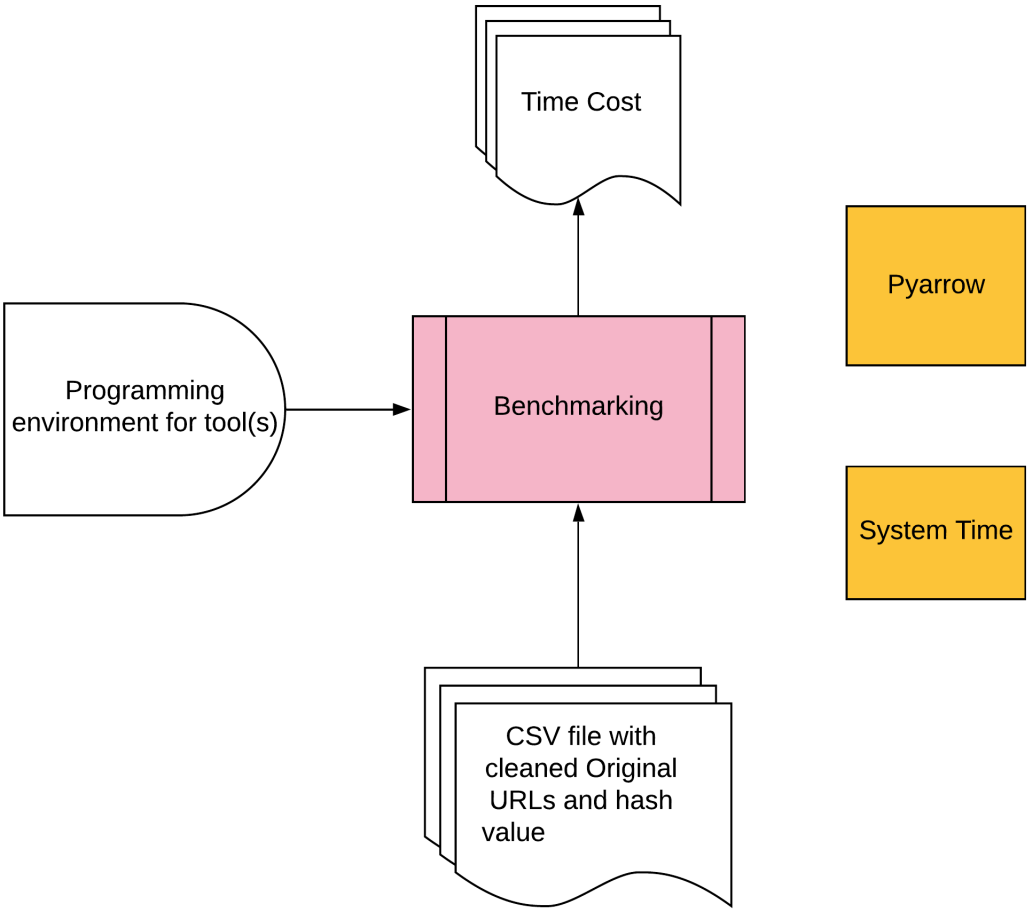

Figure 3.4: Benchmarking

- Task: Benchmarking (see Figure 3.4)
- Input file: CSV file with cleaned original URLs and hashed values
- Output file: Execution time 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 built-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 illustrates the relationship between project goals, system solutions, subtasks. Both of our project goals will be achieved by the same set of workflows. The system solution for both goals is divided into three parts which are extracting from the sample dataset, applying hashing functions, and benchmarking. Extracting information from the sample dataset part contains three subtasks which are understanding parquet file format, setting up system environment, and IO with Parquet. The applying hashing functions part also contains three subtasks which are reading CSV file with cleaned original URLs, calling hash functions with each URL, and writing the hash values of the URLs back to CSV file. There are four subtasks included in benchmarking; they are acquiring original URL and hash value from CSV file, generating a new parquet file, querying parquet file with Apache Spark, and comparing time complexity.

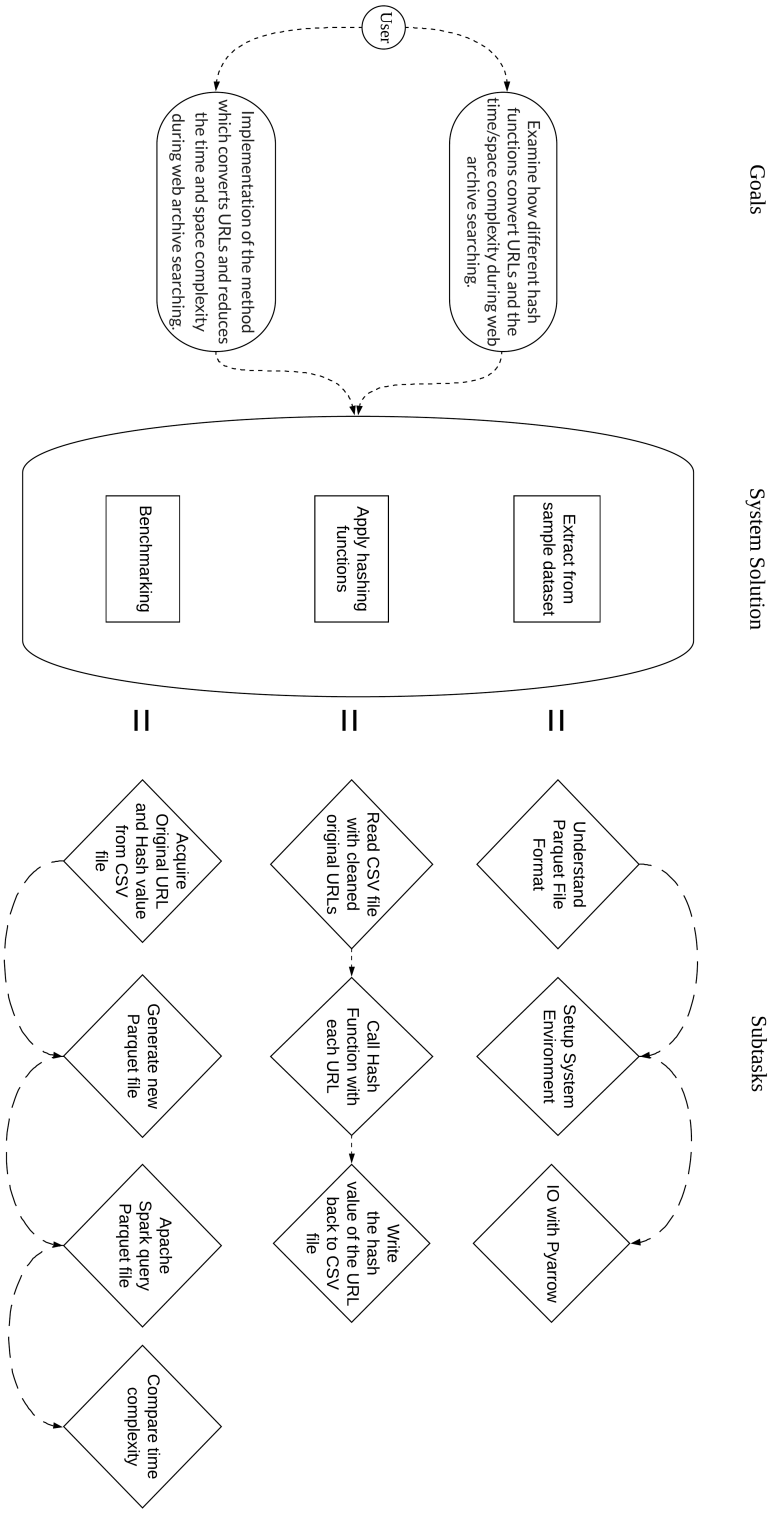

Figure 3.5: Project Workflow

Chapter 4

Implementation

4.1 SpookyHash

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. Each 128-bit hash value is stored in two uint64 variables. One is for storing the 64 low bits and the other is for storing the 64 high bits. 32-bit and 64-bit hash values are supported as well by using only the 64 low bits and casting to the correct size accordingly. The SpookyHash 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 [13]. 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 [17]. 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 [13].

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 SpookyHash consumes 8 bytes of the input and uses xor, rotation, and addition to manipulate that input [13]. 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 SpookyHash, libraries using 128-bit checksums should expect 1 collision once they hit 16 quintillion documents [13]. 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. Besides the 128-bit hash value, the 64-bit and 32-bit values are produced as well in order to test against each other and compare their performance.

4.2 Simple Hash

Our second implementation is our own design 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.

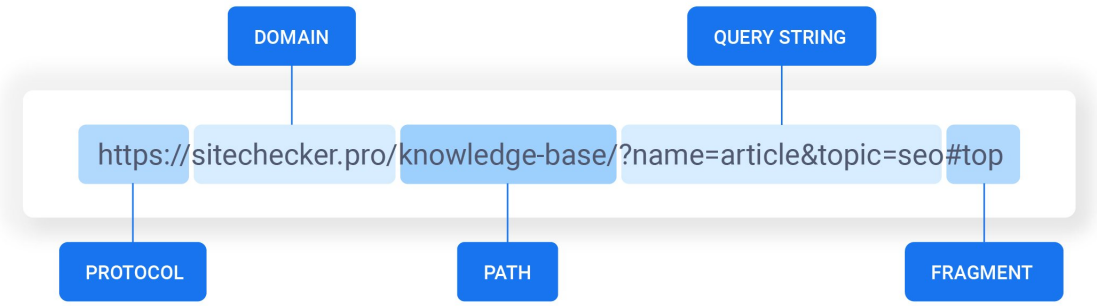

Figure 4.1: URL Structure Example [15]

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 first-level domain(FLD). An easy example would be “vt.edu”. Here “vt” is the domain name, and “edu” is the first-level domain. There are other first-level domain names, for instance: .com, .org, .gov, etc. We realized that there are only about two thousand unique first-level domains, and most of them are more than two characters in length [11]. 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 first-level domain into two bytes, we simply did an indexing hashing. Those two thousand first-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 accommodate 3844 unique strings which are more than the available number of first-level domain names. Each first-level domain is designated with a unique 2-bytes string. This string will be appended next to the protocol’s bit number since the first-level domain name 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. Then, we appended the domain name at the end of the hashed value, and ignore everything after the domain name. Since our goal is to improve the searching efficiency, after applying our Simple Hash, we will not be searching through the dataset for a specific URL. Instead, we will find all the URLs with the same first-level domains, domain names, and the same protocols. For instance, both https://canvas.vt.edu/courses/104585 and https://canvas.vt.edu/courses/104564 will be hashed into the same value “0Q2canvasvt”, we will first query with this hashed value, then perform a secondary search with the original URL. In this way, we will be able to reduce the time cost of searching through the dataset one by one. See an example in Figure 4.2.

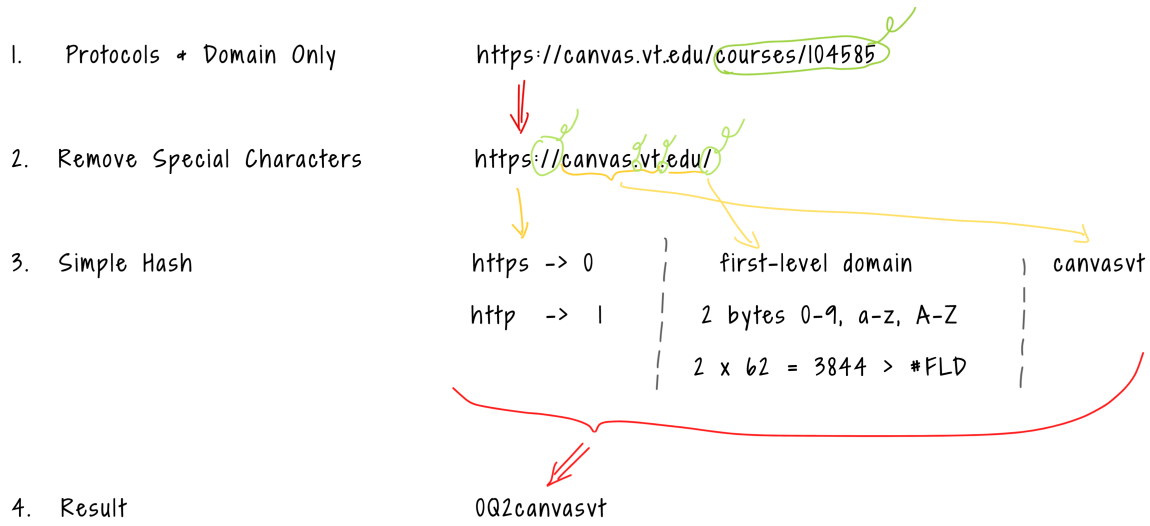

Figure 4.2: Simple Hash Process

4.3 xxHash

The xxHash is one of the hash functions we chose for this project. xxHash is an extremely fast non-cryptographic hash algorithm, working at speeds close to RAM limits. It is proposed in two flavors, 32 and 64 bits [5]. It is firstly written in C, and different language versions can produce the xxHash-compatible results, such as Java or Python. The two constructors for hash algorithms are xxh32() and xxh64(), which support well both 32-bit and 64-bit systems, separately.

The xxh32() function has three parameters: input, length, and seed. It first gets the sequence stored at memory address from "input" to "input +length of the sequence". Then, if the length of the sequence is not less than 16, adding the seed into four local variables for future hashing process, the "seed" parameter can be used to alter the hashing result predictably. After that, there will be several callings of different hash methods within a do-while loop to get a temp result value. If the input sequence is less than 16, simply add the seed and a constant variable as the temp result value. Now, it is time for further hashing processes. There will be two more while loops. Within each loop, the temp result value will be changed by calling the helper hashing methods. Finally, It will use $^=$ and $*=$ operators to change the temp result value with constant values and the shift values of itself, and return the 32-bit hash of sequences. Calling this function requires that the memory between input & input+length must be allocated and read-accessible. The xxh64() function returns the

64-bit hash of the sequence instead. It is to be noted that this function runs faster on 64-bit systems, but slower on 32-bit systems.

The main advantage of xxHash function is its proc essing speed. According to the Q-score (a measurement of quality of the hash function) table of different hash algorithms [5], it has a 5.4GB/s speed and 10 points on quality, which means a perfect score. Another advantage of this algorithm is that the return values are unsigned 32 and 64-bit integers, which makes it easy for future processes. Our project is efficiency-web-archive, and the goal of our project requires us to find relatively fast hashing algorithms, and also since we have a huge dataset, it would be better if we can have a simple and fast output format. That's the reason why we would consider xxHash as one of our hashing algorithm choices.

4.4 MurmurHash

The reason why the MurmurHash is chosen for this project is because it is a very fast, non-cryptographic hash suitable for general hash-based lookup [1]. Three MurmurHash versions are implemented in C++ and released publicly. MurmurHash3 is the latest version of MurmurHash. It contains algorithms which are optimized for each of the x86 and x64 platforms.

The function used for this project is MurmurHash3_x64_128, which is optimized for the x64 platform to produce 128-bit hash values. It takes a pointer to stream of bytes to be hashed, length of the stream, a seed, and a pointer to store the output value. Then they will be passed to the mix functions which are based on the following code:

```
k *= c1;
k = rotl(k,r1);
k *= c2;

h ^= k;

h = rotl(h,r1);
h = h*m1+n1;
```

Figure 4.3: Snippet of the MurmurHash3

As Figure 4.3 shows, k is a block of the key, h is a block of the hash state, and rN/mN/cN are constant. Each block of the key is pre-mixed using two constants and a rotate, xor

the block of key into the hash block, and then mix the hash block using a rotate and a multiply-add [2]. The function MurmurHash64A from MurmurHash2 (the second version of the MurmurHash) is called to produce 64 bit Hash Values in order to compare the results.

4.5 cityhash

The cityhash is another hash function we chose for our project. It is implemented in C++ by Geoff Pike and Jyrki Alakuijala. The cityhash has a relatively good performance at smaller data sizes which are between 10-500 bytes [16]. Based on Figure 5.5, cleaned original url's maximum length is about 300 characters which makes cityhash suitable for our project. The functions used in this project are cityhash32 and cityhash64 which can produce 32-bit and 64-bit hash values. They both take a pointer to the stream of bytes and the length of the stream as input. This hash function was designed with heavy reliance on previous work by Austin Appleby and Bob Jenkins [3] who are the developers of MurmurHash and SpookyHash. This makes the way that cityhash handles and processes the stream similar to both MurmurHash and SpookyHash.

Chapter 5

Testing/Evaluation/Assessment

5.1 SpookyHash

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 re-scaled 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 SpookyHash is applied to the original URL, the 32-bit, 64-bit, and 128-bit SpookyHash values it generates are concentrated on range 1-15, 15-25, and 30-50 character sizes. This shows that the overall character sizes of the original URL are greatly reduced after the hashing process.

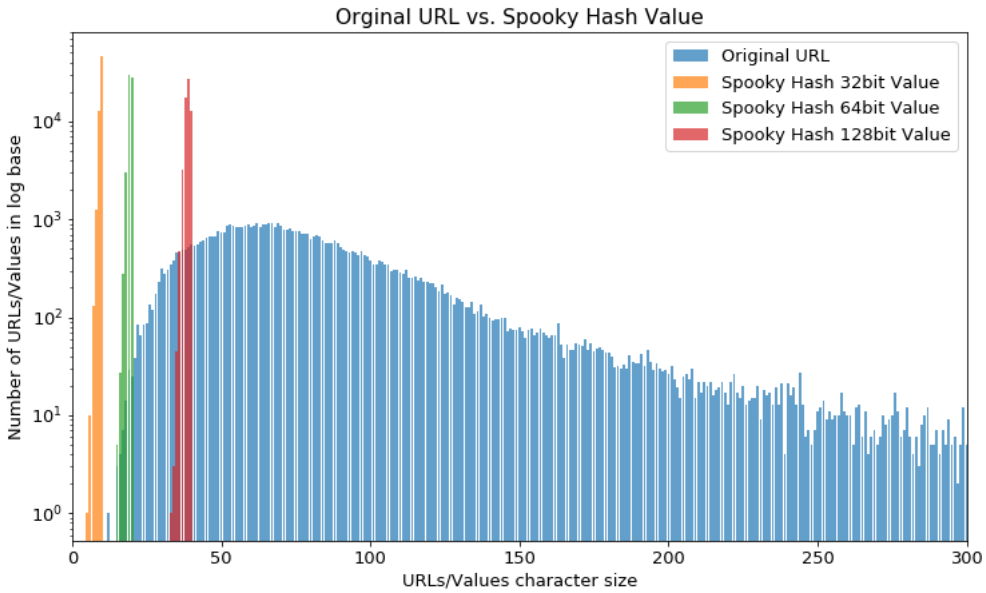

Figure 5.1: Original URLs and SpookyHash Values Length Comparison

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.

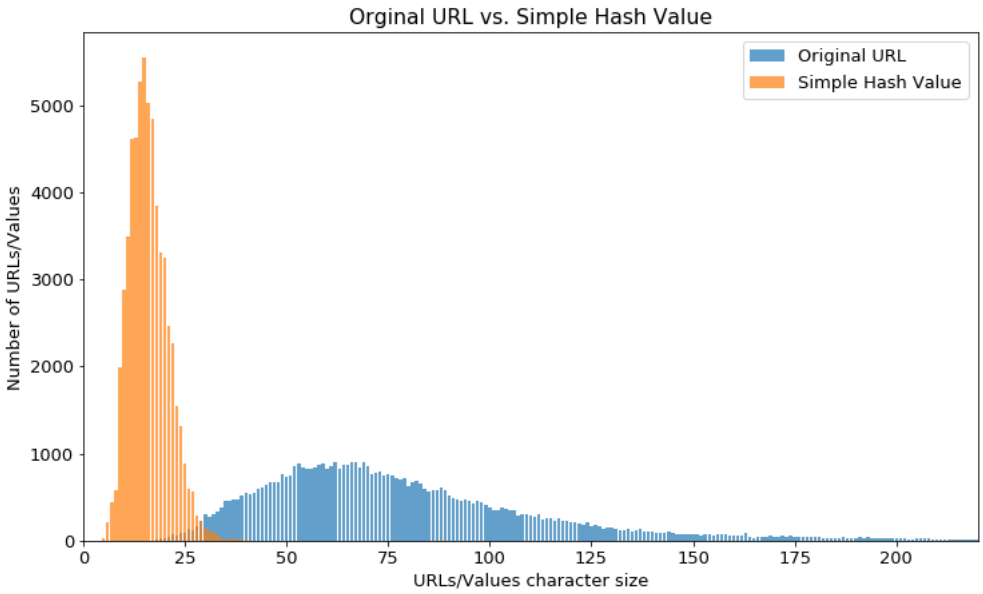

Figure 5.2: Original URL vs. Simple Hash Values's Size and Frequency

5.3 xxHash

Figure 5.3 compares the distribution of the original URL and xxHash Value in terms of character size. As we can see, the original URLs are relatively bigger, and after applying the xxHash algorithm, the returned 32-bit and 64-bit values are mostly distributed around 10. They are much shorter than the original URL and meet our expectation. Also, what needs our attention is that the returned 32-bit and 64 bit values have a fixed length. We can see that from the graph, the two lines representing xxHash32 and xxHash64 values are vertical

to the x-axis.

In conclusion, the testing/evaluation result shows that the xxHash algorithm reduces the length of original URLs, and compared with xxHash64, xxHash32 can generate shorter new URLs.

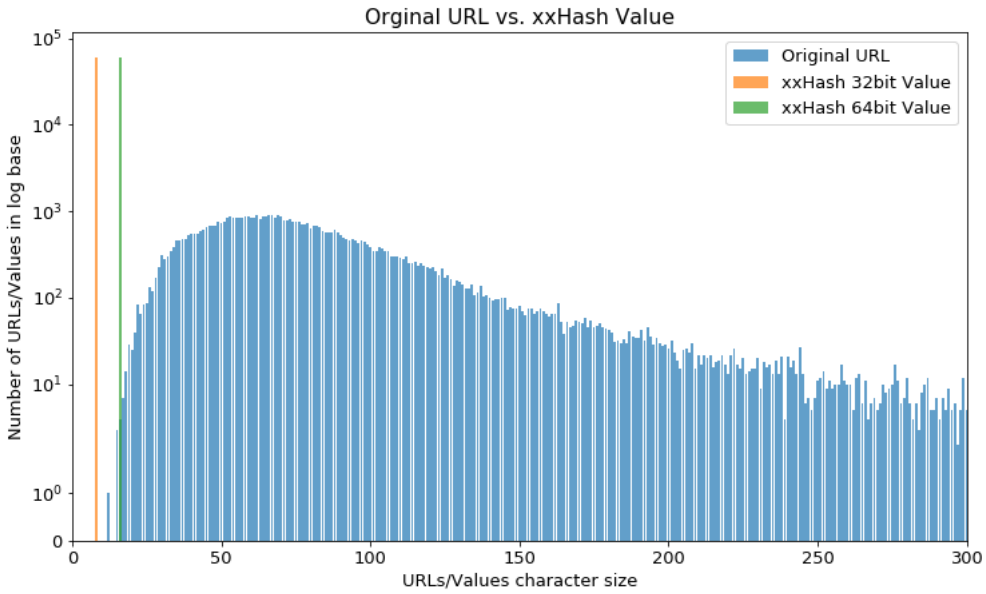

Figure 5.3: Original URL vs. xxHash Values's Size and Frequency

5.4 MurmurHash

Figure 5.4 compares the distribution of the original URL and MurmurHash Value in terms of character size. We can see from the graph that the length of most original URLs are more than 50, ranging from 50 to 300.

After applying MurmurHash, the new generated hash 32-bit and 64-bit values are closely fixed in a very narrow area around 20 and 35 separately in terms of character size, which is far shorter than most of the original URLs' length. There are some few original URLs that have smaller character size than the new hash values, but overall, MurmurHash does a good job on reducing the URLs.

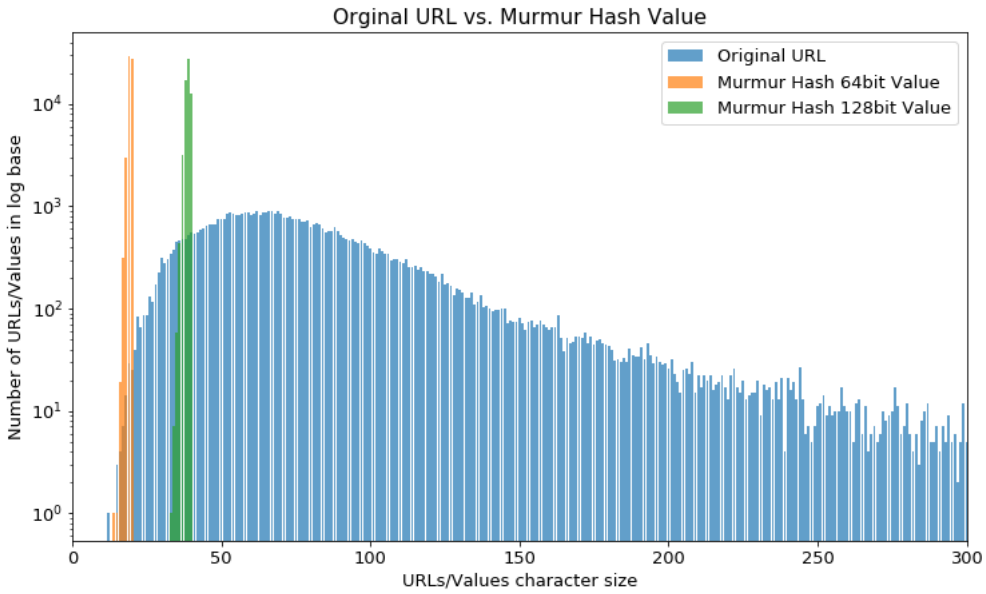

Figure 5.4: Original URL vs. MurmurHash Values's Size and Frequency

5.5 cityhash

Figure 5.5 shows the comparison of the distribution of the original URL and cityhash Value in terms of character size. As we can see that the original URLs' character sizes range from 50 to 300, which are all relatively long. Then, let's take a look at the performance of cityhash 32-bit and 64-bit. From the graph, we can easily see that both of the two values are less than 25, which are far shorter than the original URL. Also, the length of new URLs are relatively fixed to one specific value, and the character size after applying cityhash 32-bit are shorter than cityhash 64-bit.

The test and evaluation result shows that the cityhash function can generate shorter URLs and meet the basic requirement of the algorithms chosen for our project.

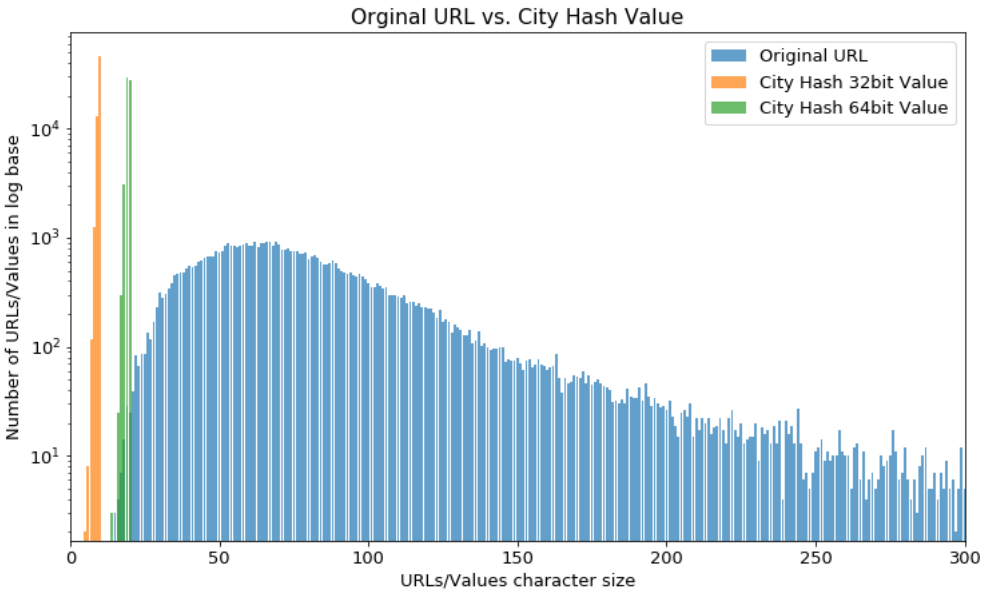

Figure 5.5: Original URL vs. cityHash Values's Size and Frequency

Chapter 6

Benchmark & Conclusion

We used Apache Zeppelin [9] with integrated Apache Spark [8] for our benchmark. First of all, we generated a new Apache Parquet file [7] by combining all calculated hash values with the data from the original Parquet file. That is the new database we will use for querying. Then we go to the Apache Zeppelin web server and use the Apache Spark script to load our new parquet file and perform queries for each hashing algorithm. This Parquet file consists of 60745 URLs associated with metadata such as date and time the URL was archived alongside with their hashing results.

In order to make our results more accurate, we randomly pick 250 records and query their original URLs in the entire data frame. Next, we query the corresponding hash values from each hash algorithm. Finally, we divide the total running time by 250 to get the average query time. This represents the average time cost for a single query.

Another thing to be aware of is when we generate the Parquet file, we combined all the hash results in a single data frame. Therefore, when we do the benchmark, we will have a comparison between different hash algorithms by querying in the same Parquet file.

Figure 6.1 shows all the performance of hash results we got from Apache Spark. It is clear that City Hash and Simple Hash made obvious improvements compared with querying the original URL. if we take a look at our Simple Hash implementation as we mentioned before, it is clear that this hashing algorithm may lead to a larger number of collisions, which would require a secondary search. But the size of the database we feed in here is comparatively small, Simple Hash algorithm does not produce a large number of conflicting results, so the secondary search will take O(1) time complexity which can be ignored. This factor might be the biggest overlooked point in future testing.

To summarize, from our benchmark results, City Hash result has a better query performance and provides a different selection of hashed result character length: 32 bit and 64 bit. Depending on the size of the database, different lengths of hashed results can be generated by selecting different hashing algorithms. Based on the result, XXHash and Spooky Hash do not have a good performance, for 32 bit, 64 bit or 128 bit. Finally, Simple Hash has the best efficiency when ignoring the secondary searches and returning multiple similar results.

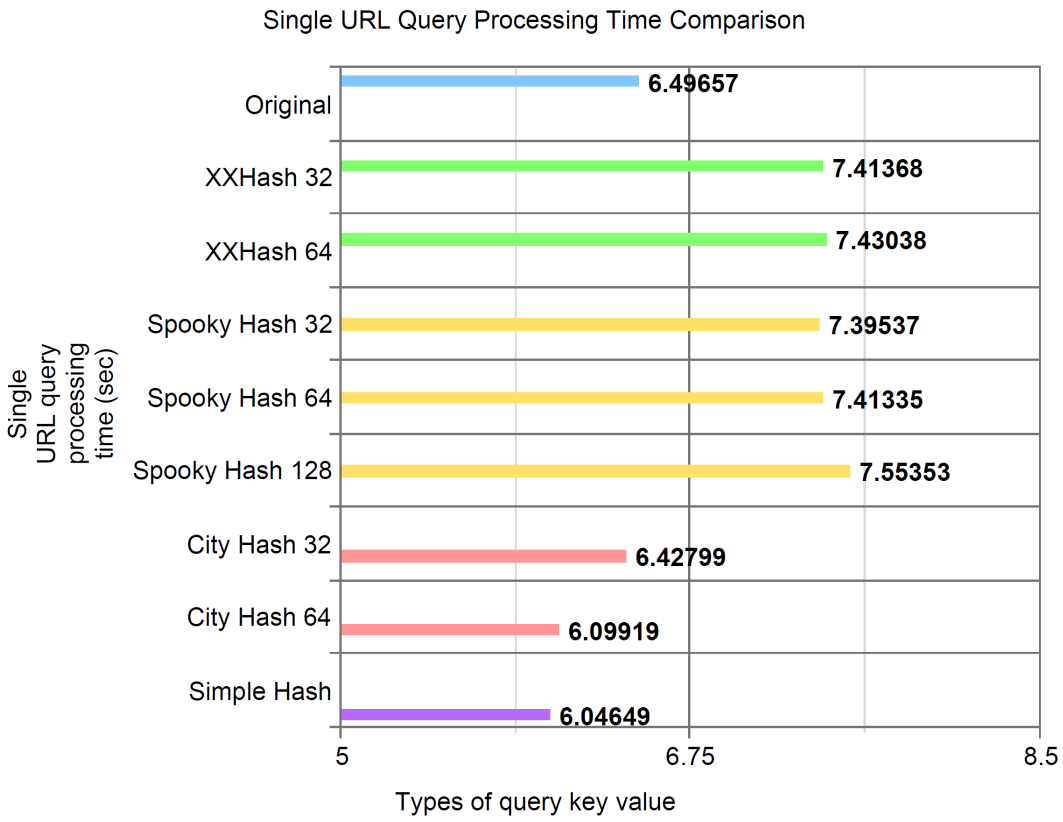

Figure 6.1: Benchmark Results

Chapter 7

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.
- We learned how to use Spark scripts in a Zeppelin web server, and how to query in a dataframe by using Scala. We also learned how to use Pyarrow and Pandas to deal with a database, like converting from Parquet to CSV, or CSV to a Pandas table.

Chapter 8

Future Works

Since our dataset is comparatively small, we did not consider collision resolution in our implementation, but when it comes to a larger set, the collision might have a huge impact on searching time, and in our project, we only consider the first searching time. If there are collisions, we will need to search through the collision data again.

We ran 250 samples once and calculated the total query time. In future work, we recommend to run multiple samples, multiple times, and calculate the average. This will result in a more accurate result for the benchmark.

Chapter 9

Users' Manual

This section will go into detail on how a user can apply our project.

9.1 Pre-processing

The pre-processing file is called basic_clean_url.py which takes origin_surt_url.CSV that passed from the previous (Parquet) step. This file only contains one column: original URLs. The output for this file is a CSV file named origin_simpleclean.CSV, which contains two columns. The first one is the original URL, while the other is the cleaned URL. The cleaned URL is the original URL without special characters.

To generate a new origin_simpleclean.CSV output file, users need to create a new origin_surt_url.CSV which contains the new URLs as the one provided above and replace the original one under the current folder. Open the terminal and call command "python basic_clean_url.py"in the current directory. A new origin_simpleclean.CSV file for the new input file should be generated.

9.2 SpookyHash & MurmurHash

The SpookyHash and Murmur Hash are called by the same Main.cpp under the SpookyMurmurHash folder. SpookyHash takes the originalURL.CSV as input. This file only contains one column of cleaned original URLs without any column title. Each row of the column contains one cleaned original URL.

The output file is called hash128bit.CSV and it contains 6 columns. The cleaned original URL column stores the input value and the rest of the columns are the hash value generated by different hash functions. Columns 2-4 are 128 bit, 64 bit, and 32 bit hash values generated by SpookyHash. Columns 5 and 6 are 64 bit and 128 bit hash values generated by MurmurHash.

To generate a new hash128bit.CSV output file, users need to create a new originalURL.CSV which contains the new URLs as the one provided above and replace the original one under the SpookyMurmurHash folder. Open the terminal and call command "g++ -o

Hash.o SpookyV2.cpp MurmurHash2.cpp MurmurHash3.cpp Main.cpp" under the spooky-MurmurHash directory. Then call command "./Hash.o" when the last step is finished. A new hash128bit.CSV file with hash values for the new input file should be generated.

9.3 Simple Hash

The Simple Hash file is called simple_hash.py which takes the origin_surt_url.CSV that passed from the previous (Parquet) step. This file contains two columns. One is for the original URLs. The other column is for the same URL with domain name being split by slashes, surrounded by braces, and still being placed at the original position in URLs. Another input file it takes is tld-list-basic.txt, which is a list of all top-level domains in the world. The output for this file is origin_simplehash.CSV which has two columns. The first column is still the original URL, while the second column is the Simple Hash value for the URL in the first column.

To generate a new origin_simplehash.CSV output file, users need to create a new origin_surt_url.CSV which contains the new URLs like the one provided above and replace the original one in the current folder. Open the terminal and call the command "python simple_hash.py" under the current directory. A new origin_simplehash.CSV file with hash values for the new input file should be generated.

If there are new top-level domains being created, in that case, the user can generate a new tld-list-basic.txt to replace the original one with the same format as in the original folder. Open the terminal and call the command "python simple_hash.py" under the current directory. A new origin_simplehash.CSV file with hash values for the new input file should be generated.

9.4 xxHash

The XXHash file is called xxhash_parse.py, which takes the origin_simpleclean.CSV that was passed from the previous (pre-processing) step. This file contains two columns. The first one is the original URL. The other is the cleaned URL. The cleaned URL is the original URL without special characters. The output file is named xxhash3264.CSV which contains three columns. The first column is still the original URL, the second column is the 32 bits xxHash value, and the third column is the 64 bits xxHash value.

To generate a new xxhash3264.CSV output file, users need to create a new origin_surt_url.CSV which contains the new URLs like the one provided above, and replace the original one in the current folder. Open the terminal and call the command "python xxhash_parse.py" under current directory. A new xxhash3264.CSV file with hash values for the new input file should be generated.

9.5 cityhash

The cityhash is called by the same main.cc under the cityHash folder. The cityhash takes the originalURL.CSV as input which has the same format as the input of SpookyHash and MurmurHash which are mentioned above. The output file is called hashValue.CSV. This file contains 3 columns. The cleaned original URL column stores the input value. The second column stores the 64 bit Hash values and the third column stores the 32 bit Hash values.

To generate a new hashValue.CSV output file, users need to create a new originalURL.CSV which contains the new URLs as the one provided above, and replace the original one under the cityHash folder. Open the terminal and call command "g++ -o city.o city.cc Main.cc"under the cityHash directory. Then call command "./city.o" when the last step is finished. A new hashValue.CSV file with hash values for the new input file should be generated.

Chapter 10

Developer's Manual

10.1 System Compatibility

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

10.2 Installation and Setup

1. 64-bit version of Python 3.7 or higher (32-bit will lead to a PEP517 Error during the Pyarrow installation process).
2. Go to GitHub and clone the repository by following the link below
   - https://github.com/GBBKN/EWAS
   - You can also find a ZIP file of the source code on our project page in VTechWorks.
   - Please contact our project contact person, Ming Cheng at ming98@vt.edu, if you still cannot locate the source code.
3. Install requirements by using **python -m pip install -r python_requirement.txt**
   - Window: Command Prompt
   - Linux: Terminal
   - macOS: Terminal
4. Run hashing
   - SpookyHash & MurmurHash
     (a) Go to SpookyMurmurHash directory
     (b) Run command:
     g++ -o Hash.o SpookyV2.cpp MurmurHash2.cpp MurmurHash3.cpp Main.cpp

   (c) ./Hash.o

- Simple Hash
   (a) Go to SimpleHash directory
   (b) Run command:
   python simple_hash.py
- xxHash
   (a) Go to XXHash directory
   (b) Run command:
   pip install xxhash
   python xxhash_parse.py
- cityhash
   (a) Go to cityHash directory
   (b) Run command:
   g++ -o city.o city.cc Main.cc
   (c) ./city.o

5. Benchmark:

- Environment:
   (a) Java: jdk-8u251-windows-x64.exe
   (b) Apache Zeppelin: Zeppelin-0.8.2-bin-all.tgz
- Required Path:
   (a) Add “JAVA_HOME”
- Create the required file:
   (a) Combine all hash results and original variables to a new Parquet file
- Open Zeppelin web server and run script:
   (a) Run “/bin/Zeppelin.cmd” to start the web server
   (b) Go to “localhost:8080/”
   (c) Run query script

Bibliography

[1] Austin Appleby. Web Archive: MurmurHash API, Jul 2011. URL https://web.archive.org/web/20120112023407/http://hbase.apache.org/docs/current/api/org/apache/hadoop/hbase/util/MurmurHash.html. [Accessed May 11, 2020].

[2] Austin Appleby. Murmurhash3, April 2020. URL https://github.com/aappleby/smhasher/wiki/MurmurHash3. [Accessed May 11, 2020].

[3] Bob Jenkins Austin Appleby. Cityhash, April 2020. URL https://github.com/google/cityhash. [Accessed May 11, 2020].

[4] Anat Ben-David and Hugo Huurdeman. Web archive search as research: Methodological and theoretical implications. *Alexandria*, 25(1-2):93–111, 2014. doi: 10.7227/ALX.0022. URL https://doi.org/10.7227/ALX.0022. [Accessed May 11, 2020].

[5] Yann Collet. xxHash - Extremely fast non-cryptographic hash algorithm, March 2020. URL http://cyan4973.github.io/xxHash/. [Accessed May 11, 2020].

[6] Miguel Costa, Daniel Gomes, Francisco Couto, and Mário Silva. A survey of web archive search architectures. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, page 1045–1050, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320382. doi: 10.1145/2487788.2488116. URL https://doi.org/10.1145/2487788.2488116. [Accessed May 11, 2020].

[7] The Apache Software Foundation. Apache Parquet, April 2020. URL https://parquet.apache.org/. [Accessed May 11, 2020].

[8] The Apache Software Foundation. Apache Spark™ - unified analytics engine for big data, April 2020. URL https://spark.apache.org/. [Accessed May 11, 2020].

[9] The Apache Software Foundation. Apache Zeppelin, April 2020. URL https://zeppelin.apache.org/. [Accessed May 11, 2020].

[10] 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. [Accessed May 11, 2020].

[11] ICANN. List of top-level domains, Apr 2020. URL https://www.icann.org/resources/pages/tlds-2012-02-25-en. [Accessed May 11, 2020].

[12] Elliot Jaffe and Scott Kirkpatrick. Architecture of the internet archive. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605586236. doi: 10.1145/1534530.1534545. URL https://doi.org/10.1145/1534530.1534545. [Accessed May 11, 2020].

[13] Robert John Jenkins Jr. Spookyhash: a 128-bit noncryptographic hash, Sept 2012. URL http://www.burtleburtle.net/bob/hash/spooky.html. [Accessed May 11, 2020].

[14] Internet Archive Organization. Internet archive: About IA, Apr 2020. URL https://archive.org/about/. [Accessed May 11, 2020].

[15] Ivan Palii. URL address: Definition, structure and examples, Dec 2017. URL https://sitechecker.pro/what-is-url/. [Accessed May 11, 2020].

[16] Aras Pranckevičius. More hash function tests · aras' website, Aug 2016. URL https://aras-p.info/blog/2016/08/09/More-Hash-Function-Tests. [Accessed May 11, 2020].

[17] swethavazhakkat. Avalanche effect in cryptography, Feb 2020. URL https://www.geeksforgeeks.org/avalanche-effect-in-cryptography/. [Accessed May 11, 2020].