Node.js based Document Key Store for Web Crawling

CS 297 Project Report

Under the guidance of

Prof. Chris Pollett

Department of

Computer Science

San José State University

By

David Bui

Introduction

Web crawling is the process of programmatically browses the internet for the purpose of indexing web pages or archiving web resources such as images. While Node.js is an open-source JavaScript runtime environment that allows for JavaScript to run outside of the web browser. The goal of this project is to implement an efficient document key store that leverages the power of Node.js and can index the output files of these web crawls known as Web Archive or WARC files. To achieve this a portion of the Yioop! search engine's PHP data storage implementation will be migrated to Node.js and include various additional hashing techniques will be implemented and that will be in the final document key store.

The preliminary implementation work for this project includes four deliverables that tackle aspects of the final project. The first deliverable which was to set up a simple key value store as a basic Express.js server in Node.js to demonstrate basic routing and configurations. The second deliverable an extension of the first deliverable was a on disk implementation of the Linear Hashing dynamic hashing technique [1] to replace the simple key value store from deliverable 1. The third deliverable tackled the implementation of a WARC reader/writer that can read WARC files and its various other formats, filter out record based on an argument criterion and create new WARC files. The fourth and final deliverable was to create a on disk Consistent Hashing [4] implementation that leverages Node.js capability to setup up multiple server instances.

Deliverable 1

As a demonstration of the functions of Node.js and as a scaffolding project the first deliverable involved creating a simple key-value storage program that acts as a server that can be called as an API from a front facing client. While Node.js has a built in HTTP module that can be used to create basic connections most Node.js server programs make use of the Express.js server framework for managing all the capabilities of a server due built handling of certain mechanisms such as CORS. So, while Express.js is the backend portion of the program the client program used the React.js frontend framework to create a small website that interacts with the server.

Figure 1: Inserting KV pair function of Client Program of Deliverable 1

There are three routes that were implemented in the API that represent the basic functions of a key-value store which are put, get, and delete. Each of the routes manipulates a single JSON file that represents the basic simple key value store. With the structure of the program set the goal of the next deliverable was to turn the single JSON file instead into a data structure that performs the same operations but is more efficient.

Deliverable 2

Picking up from Deliverable 1, the goal of Deliverable 2 was to turn the previous simple key-value store into an efficient key-value store. Instead of using the routes to manage the store instead it would be replaced with a data structure. Since the goal is to make a key-value store some sort of hashing data structure would make the most sense which is where dynamic hashing comes in. Dynamic hashing techniques grow and expand by utilizing hash functions to generate more key bits as the table expands and less key bits as the table shrinks [1]. Dynamic hashing techniques fall under two schemes one with directories and one without. Extendible Hashing is the technique with directories and in short has a separate structure called a directory that each entry points to a bucket where the data is hashed. This hashing scheme grows exponentially as the number of indexable buckets double as the number of directory bits are doubled each time a criterion is met. On the other hand, Linear Hashing is the directory less scheme and expands it's table one bucket at a time [1]. Linear hashing is directory less because keys are hashed directly to the bucket. Linear Hashing get its name from this one bucket at a time linear growth, hence the name Linear hashing. One small advantage Extensible hashing has over Linear Hashing is that the due its criterion design it is possible to set an upper bound limit on access times [1]. However, the wasted memory from an exponentially growing directory is a downside that puts Extensible Hashing below Linear Hashing in most use cases.

So, data structure chosen was a physical hash table that implements Linear Hashing.

Linear Hashing has 4 variables that need to be kept track of to grow and shrink the table. N which is the number of initial buckets and must always be a power of 2, S a pointer to the current

bucket 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 load factor which is the number of items in the table divided by the current number of buckets times the average or set size of each bucket. Load factor is typically used as the threshold to grow or shrink the hash table. With the variables defined

A hash function will give typically give some number of bits. For example, some hash function gives a 32-bit output for some key. However, in Linear Hashing we will only use the first I bits, since the hash table only starts with N buckets. If we start with N = 2 buckets, then I = 1 bits. 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 exceed the load factor add 1 bucket to N. If N > (2^I - 1) we need to increment I to address to the new bucket. When any bucket is added we split the bucket at index S's keys with the new bucket, rehash all keys in the split bucket if I is incremented, and then increment S. Once N has doubled from where it was initially, we reset S to 0.

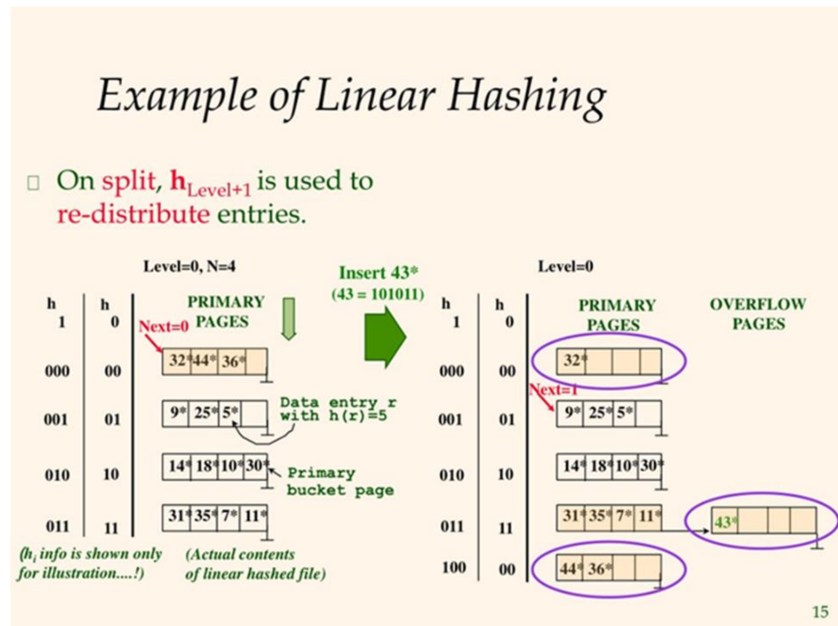

Figure 2. Linear Hashing example from [1]

While an in-memory implementation of a Linear Hash Table is simple by just making the buckets a linked list of arrays, However, a physical implementation is a little more complex. For this deliverable text files were used to represent each bucket. Each text file would include a header in the first line containing the keys mapped to this bucket and each subsequent line would be the key value pair.

```
429,475,481,526,534,556,561,584,603,647,648,775,915,927,960
{"429":"cat429"}
{"475":"cat475"}
{"481":"cat481"}
{"526":"cat526"}
{"534":"cat534"}
{"556":"cat556"}
{"561":"cat561"}
{"584":"cat584"}
{"603":"cat603"}
{"647":"cat647"}
{"648":"cat648"}
{"775":"cat775"}
{"915":"cat915"}
{"927":"cat927"}
{"960":"cat960"}
```

Figure 3. Physical Bucket Implementation for Deliverable 2

This implementation was chosen to utilize the power of Node.js streams which are able to quickly stream new line delimited files. However, this implementation could be further improved by separating the header from the file to prevent bit pushing from adding to the header but would sacrifice the ability to use continuous streams. With the physical implementation settled the Linear Hash data structure was added to the Express server in the backend and the routes became function calls to manipulate the data Linear Hash Table. Since, the hash table is physically implemented it is also able to persist beyond a server shutdown due to a parameters file that is saved just before a shutdown occurs. With Deliverable 2 finished we pivoted to WARC files for Deliverable 3.

Deliverable 3

For deliverable 3 the goal was to create a reader/writer for Web Archive Files or WARC files. WARC files are the output of web crawls and are used to preserve data from web crawls which are programmatically executed traversals of the internet. The WARC file consists of a concatenation of one or more WARC records. There are 8 types of WARC records, warcinfo, response, resource, request, metadata, revisit, conversion, and continuation. A WARC record consists of a header and a record content block. The header has mandatory named fields Date, Type, Length of the record, plus, other fields that assist in identification and retrieval. The content block contains resources in any format such as images and audio [2].

```
1  WARC/1.0
2  WARC-Type: warcinfo
3  WARC-Date: 2011-02-25T18:32:19Z
4  WARC-Filename: WIDE-20110225183219005-04371-13730~crawl301.us.archive.org~9443.warc.gz
5  WARC-Record-ID: <urn:uuid:88fbcbee-f24e-47c1-b0c4-f7a9530ceb74>
6  Content-Type: application/warc-fields
7  Content-Length: 442
8
9  software: Heritrix/3.0.1-SNAPSHOT-20110127.213729 http://crawler.archive.org
10 ip: 207.241.232.79
11 hostname: crawl301.us.archive.org
12 format: WARC File Format 1.0
13 conformsTo: http://bibnum.bnf.fr/WARC/WARC_ISO_28500_version1_latestdraft.pdf
14 operator: kenji@archive.org
15 isPartOf: wide
16 description: seeds.txt
17 robots: obey
18 http-header-user-agent: Mozilla/5.0 (compatible; archive.org_bot +http://www.archive.org/details/archive.org_bot)
19
20
```

Figure 4: Example WARC record of type warcinfo from an Internet Archive crawl.

The WARC reader/writer for the deliverable 3 was implemented as a CLI program to parse these WARC files. WARC files are easy to parse as each record begins in its header with a WARC/1.0 which enables parsing a record by reading until this line is met. WARC files are usually gzipped compressed with a .warc.gz extension. This means that gzip decompress streams

are needed to decompress WARC file data as it is read. Most languages have a built-in library for gzip compression which in Node.js is its own Zlib module. With that information and Node.js built in decompress streams WARC files are simple to parse. As the WARC files are easy to parse themselves other functions were added to the cli program such as the filtering of WARC records from using date, url, and file type to newly created WARC files and reading and creation of CDX files which are used to index WARC files.

CDX files are used to index WARC records inside WARC files for quick access. CDX files start with a header legend specifying the format of subsequent lines. Each subsequent line is a new line delimited index into a WARC file. They can contain various information about the WARC record however, they all typically contain a URI of where the record was retrieved from, a file offset into the WARC file compressed or uncompressed, and a record length compressed or uncompressed [5].

```
CDX A b e a m s c k r V v D d g M n
0-0-0checkmate.com/Bugs/Bug_Investigators.html 20010424210551 209.52.183.152 0-0-0checkmate.com:80/Bugs/Bug_Investigators.html text/html 200
58670fbe7432c5bed6f3dcd7ea32b221 a725a64ad6bb7112c55ed26c9e4cef63 - 17130110 59129865 1927657 6501523 DE_crawl6.20010424210458 - 5750

0-0-0checkmate.com/Bugs/Insect_Habitats.html 20010424210312 209.52.183.152 0-0-0checkmate.com:80/Bugs/Insect_Habitats.html text/html 200
d520038e97d7538855715ddcba613d41 30025030eeb72e9345cc2ddf8b5ff218 - 47392928 145482381 4426829 15345336 DE_crawl3.20010424210104 - 6356

0-0-0checkmate.com/Hot/index.html 20010424212403 209.52.183.152 0-0-0checkmate.com:80/Hot/index.html text/html 200
52242643710547ff4ce2605ed03ed9e2 b06d037c06e7ffd7afc6db270aca7645 - 21301376 62305547 1855363 6627262 DE_crawl6.20010424212307 - 6317
```

Figure 5: Example CDX file from [5]

As CDX files always start with the header legend it must be decomposed into a mapped legend which in the CLI program created for this deliverable is a JSON from the letter to a full description. The filtering CDX files by the same parameters as WARC files and the creation of new CDX files was integrated into the CLI program as well.

Deliverable 4

The final deliverable was to implement consistent hashing on multiple key-value store instances. As Node.js can run multiple server instances Consistent hashing is useful for dividing data across instances for faster access. Consistent hashing is a hashing technique for distributing keys among nodes/servers. Originally developed to distribute webpages among caching servers [4]. Keys are hashed using a ranged hash function usually from 0 – 1 to form a radial pattern. Nodes/Servers are also hashed onto the circle. Keys are hashed and distributed to the nearest hashed node on this circle. The issues with the original design were that if two nodes are close to each other then one of them will take a larger share. Also, adding/deleting a node usually means the keys are distributed or remapped from 1 other node. A solution to this is to map each node to multiple virtual nodes distributed evenly around the range.

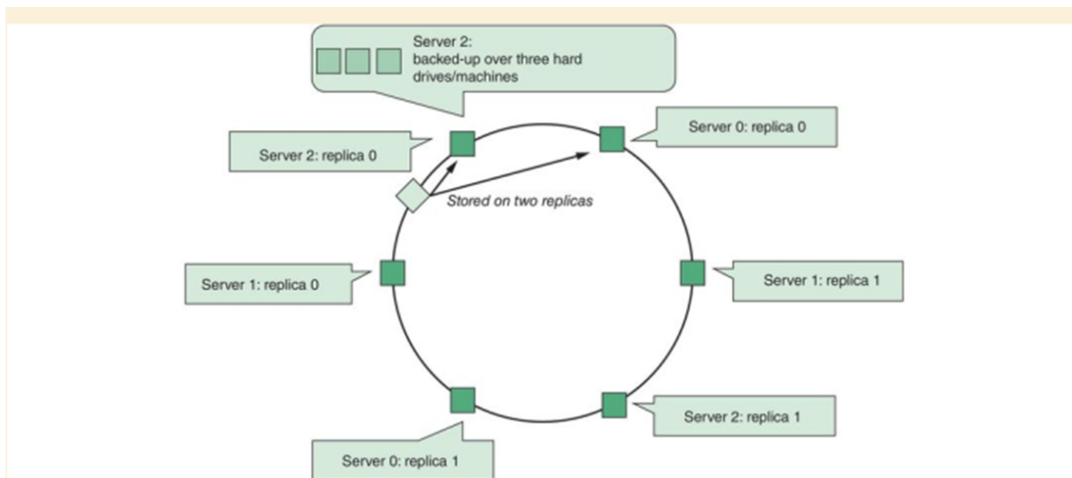

Figure 6. Virtual replicas in Consistent Hashing from *Principles of Database Management: The Practical Guide to Storing, Managing and Analyzing Big and Small Data*

The implementation for this deliverable due to time constraints is to have a root server handle all the consistent hashing operations rather than have gossiping style servers that communicate with one another. The root server would contain the replicas and references to each sub server representing nodes in the consistent hashing circle. Each sub server contains as its data store a Deliverable 2 Linear Hash Table from Deliverable 2. Adding a server would automatically generate some set number of replicas and would trigger a rebalancing of the keys within the server. Deleting a server triggers of move of all that server's data to other servers.

```
printing consistent hashing circle
{
  '0.7484796508436069': '5000',
  '0.05413429305834251': '5000',
  '0.676859276571685 2': '5000',
  '0.3756441584266907': '6000',
  '0.5464650329610465': '6000',
  '0.7418951887839189': '6000',
  '0.028702619999917293': '7000',
  '0.3970711957959883': '7000',
  '0.46563783596514885': '7000'
}
```

Figure 7. Consistent Hashing Circle with replicas.

```
Data of Server with port number: 5000
1: dog1
2: dog2
4: dog4
5: dog5
6: dog6
7: dog7


Data of Server with port number: 6000
0: dog0
3: dog3
8: dog8
9: dog9


Data of Server with port number: 7000
```

Figure 8. Data of each server assigned from the virtual nodes in Figure 7.

Conclusion

For this semester, basic implementation work was done on a key–value created in Node.js. Over the course of Deliverables 1 and 2, a persistent key-value store was implemented using a Linear Hashing Scheme. Deliverable 3 involved creating a WARC file reader/writer that can perform filtering and indexing operations as well. Deliverable 4 involved implementing Consistent hashing on multiple server instances containing Linear Hash table stores. These deliverables will serve as a platform of the final product which will be a Node.js data storage engine that will contain much more features as it will also be a Node.js port of the Yioop! database storage engine.

Next semester will be further refining the key value store and turning into a full-fledged Node.js based database system. The aim is to make this database insert and read efficient at the cost of updates and deletes. Other features as creating the ability to create indexes on tables like in SQl and a database driver interface for interacting with other languages like ODBC.

References

[1] Enbody, R. J., & Du, H. C. (1988). Dynamic hashing schemes. ACM Computing Surveys, 20(2), 850-113

[2] WARC, Web ARChive file format. (2009, August 31).

https://www.loc.gov/preservation/digital/formats

[3] Deliverable 3 github

https://github.com/bbdavidbb/warcfilter

[4] Karger, David & Lehman, Eric & Leighton, Tom & Levine, Matthew & Lewin, Daniel & Panigrahy, Rina. (2001). Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. ACM STOC. 10.1145/258533.258660.

[5] CDX file specifications (2015)

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