AndreAsk, a scalable Search Engine (G22)

Zichuan Yu[1], Yuru Wang[2], Paul-Arthur Asselin[3], Lichang Xu[4]

Abstract— **In this paper, we present the architecture for our distributed search engine. We performed a crawl of a small subset of the internet in early May 2018 (more than $1,000,000$ pages from $24,000$ domains). The data we collected was than indexed and ranked in order to support the functionality expected from a basic search engine. We will evaluate our components with an eye towards potential future improvements.**

I. INTRODUCTION

Division of Work:

Paul-Arthur	Crawler
Zichuan	Indexer
Lichang	Page-Ranking
Yuru	Search Engine back-end and front-end

We experienced some delays getting started and fell slightly behind our proposed timeline but made up for it with our good work ethic. The last few days were spent integrating the various components to form a complete product.

AndreAsk is fast and responds well to all sorts of queries (single and multi-word queries). Results are augmented with information from 3^{rd} party APIs.

II. ARCHITECTURE

A. Major Components

Please see the architecture diagram attached at the end of this paper.

B. Data Structures

- **Web Frontier** — we used AWS Simple Queue Service (SQS) to store urls that needed to be processed. The distributed crawlers would request urls to process and push newly extracted ones back into SQS.
- **Metadata DB** — we used Redis (an in-memory key/value store) hosted on AWS ElasticCache to store individual page metadata (*e.g: title, description, language, image, time last accessed, location of html in S3*) as well as domain-level metadata (*e.g: time last accessed, number of pages retrieved*).
- **Document DB** — we used AWS S3 to store urls and their corresponding html content (with *script* and *img* tags removed in order to save space). Since S3 is built for large objects, we bundled the url, html contents in blocks of 100 using randomly-generated 128 *bit* names. In addition we used S3 to store urls and their corresponding normalized pagerank scores.
- **Indexer DB** — we used AWS DynamoDB to store the text-frequency information computed by the Map-Reduce jobs. The data-structure is as follows:

Partition Key	Sort Key	Secondary Idx (Local)...
word, String	*url, String*	*TF, Number*

Secondary Idx (Local)	Secondary Idx (Global)
TF/IDF, Number	*IDF, Number*

C. Justification

S3 and DynamoDB were chosen because they represent the services we needed in the AWS universe. We chose Redis to store metadata because it is a fast in-memory key/value store. Our crawler needs to access it multiple times per crawled page so speed was critical.

D. Scalability

Our architecture is highly scalable. Our crawler queue, Document DB and Indexer DB (SQS, S3, DynamoDB) can scale almost indefinitely on AWS (albeit at a cost). Our Redis instance was not set to automatically scale but handled the intensive crawling process well (the maximum CPU utilization was 18.9%). We could easily multiply the instance's capabilities by 20x if we wanted.

Our crawlers also scaled well. We ran them on 3 EC2 instances (with 16, 16, 64 cores respectively). We made sure to use every available processing core

[1]zichuany@seas.upenn.edu
[2]yuruwang@seas.upenn.edu
[3]passelin@seas.upenn.edu
[4]lichangx@seas.upenn.edu

by monitoring the resource utilizations using htop. We could scale to many more instances very easily (our main bottleneck is Redis and we can easily scale to handle 100x the current max throughput).

E. *Fault-Tolerance*

Our architecture is somewhat resistant to failures. The distributed nature of our crawlers mean that one can fail and the rest will continue working. By using fault-tolerant AWS services (S3 & SQS for example), we have mitigated most points of failures (AWS remains a single point of failure). Our Redis node (hosted on AWS ElasticCache) is the most likely cause of failure. This can be mitigated by running a Redis cluster instead of a single node (ElasticCache supports this functionality).

III. IMPLEMENTATION

A. *Crawler*

Seeding the Queue — We created a small program that seeds the queue with a selection of the 250 most popular hosts and the 20 most popular Wikipedia pages.

Talking to Redis — In order to test the crawler locally, we needed to talk to our Redis instance running on AWS's ElasticCache from outside of the security group. This required setting up a *micro* instance in the security group on EC2 to act as a pass-through node (this was done by editing the *iptables*'s NAT prerouting and postrouting attributes).

Politeness — We respect the *crawl-delay* and *disallow* arguments of domains. If we cannot find a *robots.txt* for a particular domain, we remain courteous and space out our request by 5 seconds.

Running on EC2 — We used the default AWS Linux AMI image on our EC2 instances. We deployed our code to them as such:

1) install *apache-maven*
2) install *java 1.8* (and *javac*)
3) rsync the crawler code to the EC2 machine
4) use the *screen* terminal multiplexer to launch multiple instances of multi-threaded crawler, using *screen* allows us to disconnect from the terminal window / the *ssh* session

We can monitor our crawlers by login into the machines and reading system-out for each *screen*.

Multi-threading — We utilised the StormLite stream processing framework to run our crawler. We modified the *LocalCluster* file to have thread pools ranging from 40 to 80 threads depending on the machine we were using.

Obtaining Features — We obtain the following features from each url: *title, description, head (first 50 words), language, image link*. We also store when the page was accessed and its location in S3. The features are obtained in a robust way (if a tag is not present, we look for its open-graph equivalent).

Storing the HTML — We flatten the parsed HTML for each page to fit on one line and append it to a file. Once we have 100 pages, we upload the file to S3 and start with a new file. This is good for fault-tolerance (we save our work constantly) and it prevents the files in S3 from getting too big (which may be an issue when it is time to retrieve the HTML to display a cached version of a page). Our filenames are randomly-generated 128-bit string (unlikely to collide).

Robots.txt — We had initially not followed redirects when looking for *robots.txt* files. We fixed that and also had to strip the sub-domains to obtain the *robots.txt* file.

Diversity & Spider Traps — In the interest of saving a broad range of sources. We placed a limit on the number of pages we can retrieve from a particular domain (implemented by incrementing a domain count in Redis). This also partially solves the spider trap issue.

B. *Indexer*

The indexer's function is to store which urls contain a certain word. The word must therefore serve as the partition key. We used the url as the sort key in DynamoDB.

Two MapReduce jobs are used. The first generates all the entries of the indexer and writes them to S3 storage. The second one simply parses these entries and writes them to DynamoDB.

C. *PageRanker*

The pageranker's function is to give a normalized score to each url/host that indicates the "importance" of the page. the pagerank is implemented as a MapReduce job whose mapper will first parse in the links and value, separate the PR values and links. The reducer will store the page with the initial PageRank and the outgoing links.

Problems like ”dangling links” or ”sinks” are crucial. We tried the original PageRank paper approach, i.e., simply removing them from the system until all the ranks are calculated. We also attempted to solve the problem by forcing ”dangling” node to point to every other node but found this is really computationally heavy.

D. Search Engine

Front-End — When a user makes a query, the front-end server processes the query by splitting it into individual keywords using a set of splitter characters. The front-end server then sends a GET request using AJAX to the back-end server with the keywords and pagination parameters.

Back-End talking to the Indexer — When the back-end receives a GET request from the front-end, it itself sends a GET request to the indexer server. The indexer server responds with JSON containing the top 100 urls along with their TF/IDF scores.

Back-End talking to the Page-Ranker — The back-end then sends the 100 urls to the PageRank server via HTTP POST. The PageRank server sends back the pagerank score for each url.

Back-End talking to the Metadata DB — The back-end can then request information about each page from the Redis instance (*e.g:* title, description, language, first 30 words, etc...).

Back-End ranking function — The back-end decides in what order to display results based on its own ranking function that incorporates pagerank score, tf/idf data and relevance score calculated from the page's metadata. It then returns the top 10 results to the front-end server along with information obtained from other web APIs (Google Map API, Weather API, Ebay API).

E. Ranking

We use the pagerank score, TD/IDF score and a custom ”relevance score” to determine the ranking of pages.

Calculating the score:

- Calculate the weight of each keyword with respect to the query:

$$W_{i,q} = 0.5 + \left[0.5 * \frac{freq(i,q)}{maxfreq(I,q)}\right] * \log(\frac{N}{n_i})$$

  Where:
  - $W_{i,q}$ denotes the weight of the i^{th} keyword in document q
  - $freq(I,q)$ denotes the frequency of the i^{th} keyword in the query
  - $maxfreq(I,q)$ denotes the maximum frequency of all keywords in the query $\log(\frac{N}{n_i})$ denotes the IDF (Inverse Document Frequency) of the i^{th} keyword
- Calculate the weight of each query keyword with respect to the document that contains it. This is done by calculating the TF/IDF score of the keyword:

$$W_{i,j} = TF/IDF\ score$$

- Calculate the net IR score by combining $W_{i,j}$ and $W_{i,q}$ using the following formula:

$$IR\ Score = \sum_{i=1}^{k} W_{i,j} * W_{i,q}$$

- Calculate the custom ”Relevance Score” by considering the keywords found in the title and description tag of the document. For every keyword found in the title of the document, we increment the relevance score of that word by 5. We increment by 1 for words found in the description tag.

$$Relevance\ Score = \sum_{i=1}^{k} (S_{i,t} + S_{i,d})$$

  Where:
  - $S_{i,t}$ denotes the score the document can get for the i^{th} keyword in the query using data from the title tag
  - $S_{i,d}$ denotes the score the document can get for the i^{th} keyword in the query using data from the description tag
- Finally, for every candidate url, we multiply the pagerank score, IR net score and relevance score to obtain the final ranking score:

$$Total\ S = IR\ S * PageRank\ S * Relevance\ S$$

  where S stands for score.

IV. EVALUATION

A. *Crawler*

We believe our crawler is capable of scaling up to handle reasonable amounts of web data. With only three instances (2∗m4.4xlarge (16 cores) & 1∗m4.16xlarge (64 cores)), we were able to crawl roughly $290k$ pages per hour (4800 pages per minute).

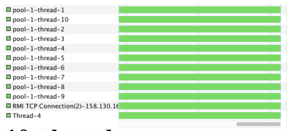
The load on the Redis instances was small despite this rate. The Redis instance never used more than 20% of its CPU and the network utilisation was $0.82GB$ for an hour at the peak ($290k$ pages).

We evaluated the performance of our crawler to determine how many threads was ideal on a given processor. We collected the following data for crawling 100 pages:

5 threads	*77s*
10 threads	*48s*
20 threads	*32s*
40 threads	*27s*
80 threads	*27s*

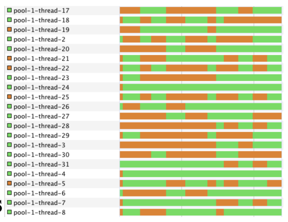
This highlights the diminishing returns from using many threads (managing threads adds over-head).

This can be shown using a profiler (i7 with 2 cores):



10 threads

80 threads



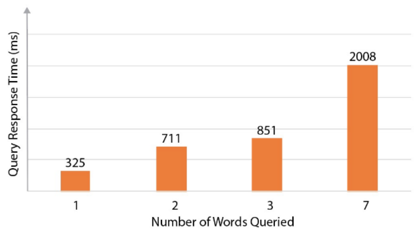
B. *Indexer*

Indexer Building

The indexer-building MapReduce job is run on 19 x m4.large machines, taking an average 1 hour and 12 minutes to handle 1 million datapoints. The DynamoDB-writing MapReduce job runs on 5 x m3.large EC2 machines, taking 15 hours to write all data to a single Dynamo Table.

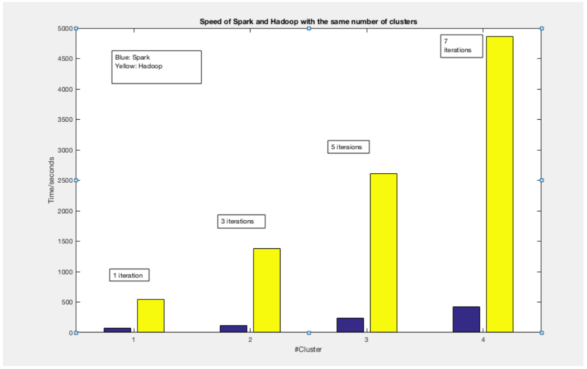
Indexer Server

The indexer servers performance is measured by query-time, as shown below:



C. *Ranking*

The PageRanker performance is measured by the total time to run it over all crawled urls with k=1,2,3,4 clusters and by the number of iterations until convergence. Both Hadoop and Spark approaches are attempted:



We tried both url-based and host-based ranking scheme. We noticed the former approach produces normalized ranks with little difference across many urls whereas the latter that regrads inter-domain urls as pure navigation produces normalzied ranks with obvious difference. The corresponding performances are not thoroughly evaluated, though, due to last-minute hacking incident of our redis server. Common SEO defenses such as blacklist of suspicious metatags are explored before the code was stolen unexpectedly.

V. TAKEAWAYS

The project was a success. The search engine behaves as expected and is very responsive (**fast!**).

We are proud to have built many components from scratch (*like the HTTP library and Robots.txt parser*).

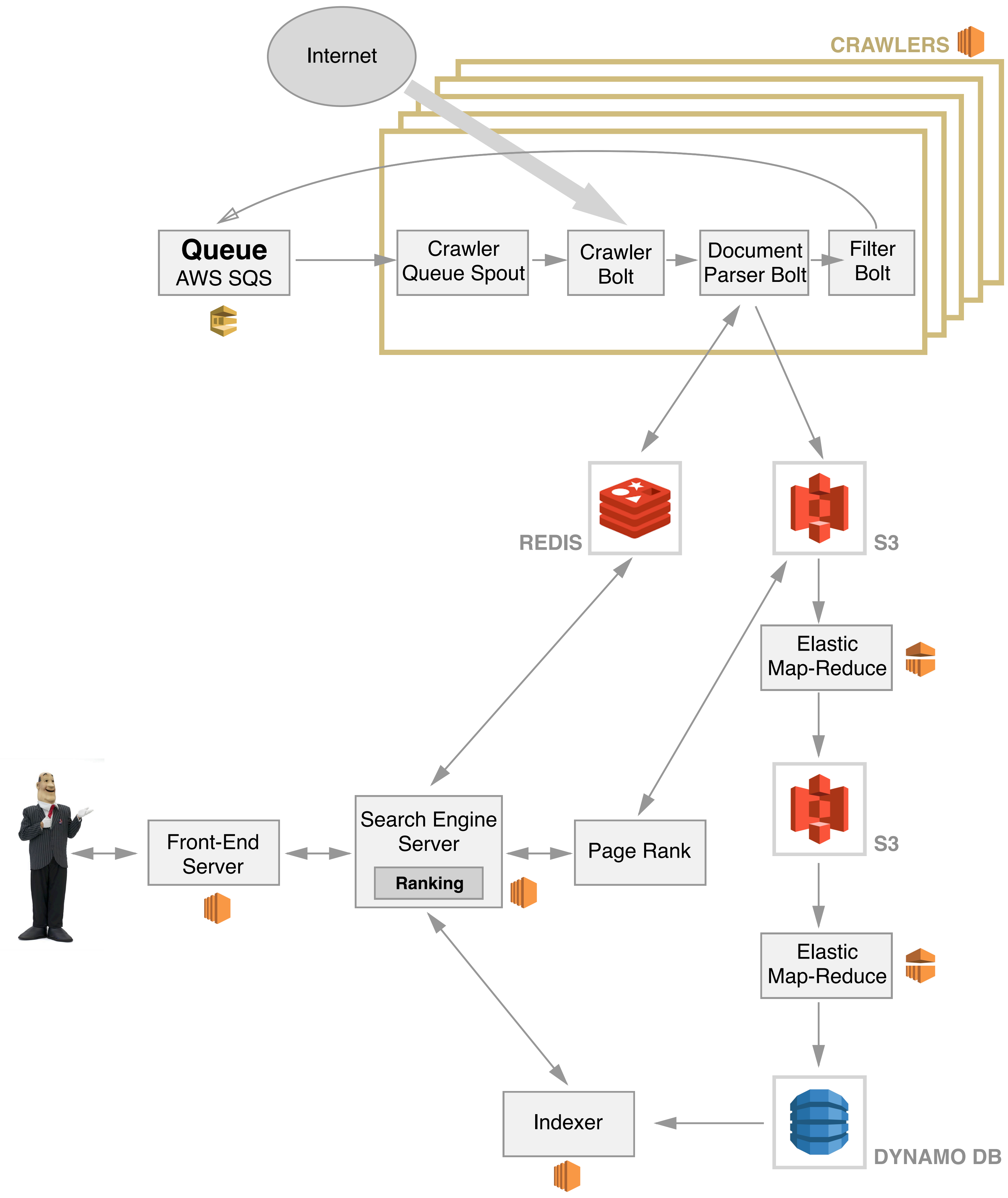
On the ranking side, our search engine performs very well for single-word queries but sometimes fails to provide very accurate results for multi-word queries. The problem could be fixed by having a larger corpus of documents (which would require more financial resources).

We faced a few setbacks. One of our team-members got their laptop stolen. It contained important code that had not been backed-up. Secondly, we opened the Redis port (6379) on the proxy to make it easier to develop and do integration (given that we weren't in geographic proximity). An attacker infiltrated our Redis instance and flushed all the keys, leaving only the key *crackit*. This taught us to value backups and never to never leave open ports on servers.

ACKNOWLEDGMENT

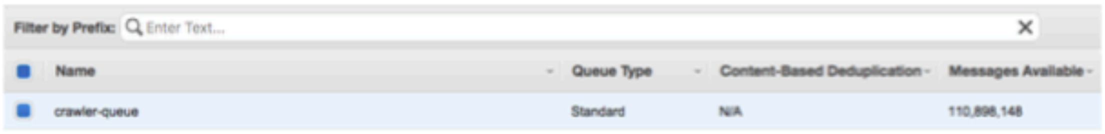
Grateful to Dr. Haeberlen and TAs for a wonderful semester.

Architecture Diagram:



APPENDIX:

Figure 1: Our url queue. It grew to a very ambitious size.



Filter by Prefix: Enter Text...

Name	Queue Type	Content-Based Deduplication	Messages Available
crawler-queue	Standard	N/A	110,898,148

Figure 2: Running the crawler on some big machines (note the high CPU utilisation).

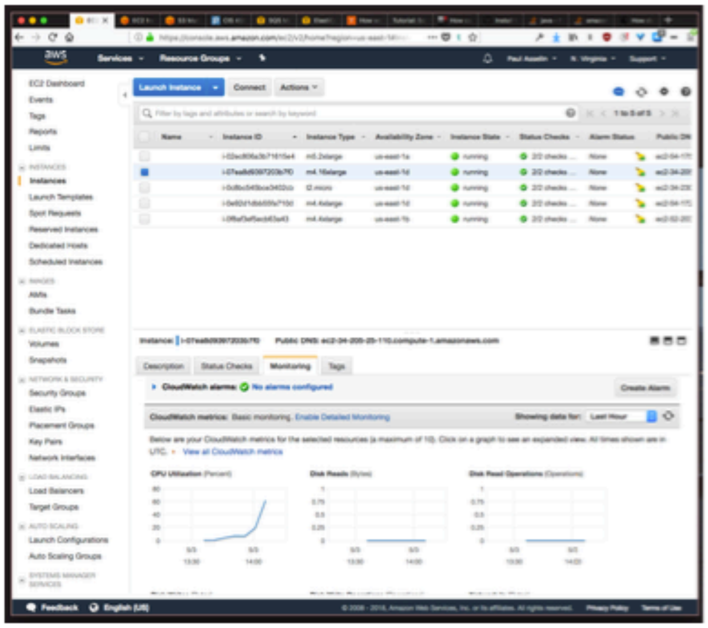
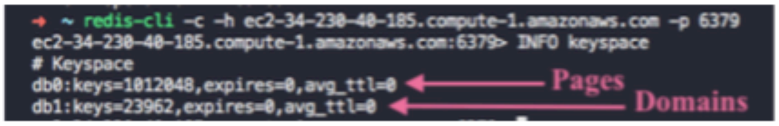
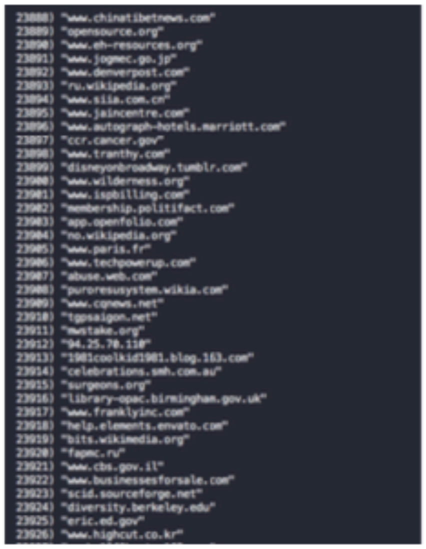


Figure 3: Redis stats (INFO keyspace)

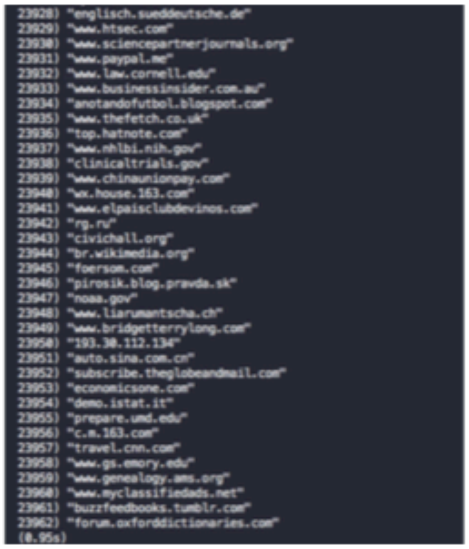


```
→ ~ redis-cli -c -h ec2-34-230-40-185.compute-1.amazonaws.com -p 6379
ec2-34-230-40-185.compute-1.amazonaws.com:6379> INFO keyspace
# Keyspace
db0:keys=1012048,expires=0,avg_ttl=0
db1:keys=23962,expires=0,avg_ttl=0
```

Figure 4: Redis sample domains (keys * in db1)

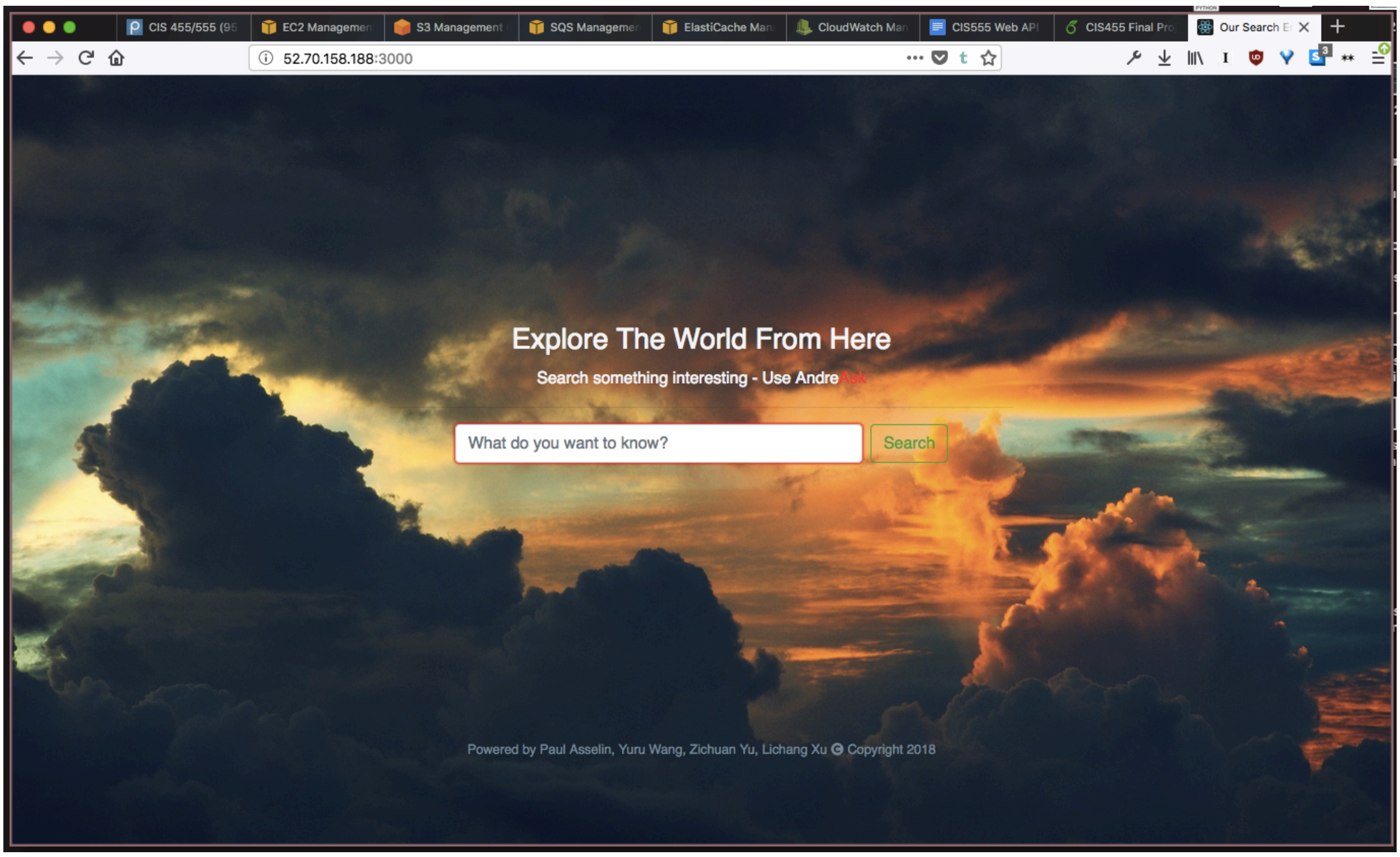


```
23888) "www.chinatibetnews.com"
23889) "opensource.org"
23890) "www.eh-resources.org"
23891) "www.jogmec.go.jp"
23892) "www.denverpost.com"
23893) "ru.wikipedia.org"
23894) "www.siia.com.cn"
23895) "www.jaincentre.com"
23896) "www.autograph-hotels.marriott.com"
23897) "ccr.cancer.gov"
23898) "www.tranthy.com"
23899) "disneyonbroadway.tumblr.com"
23900) "www.wilderness.org"
23901) "www.ispbilling.com"
23902) "membership.politifact.com"
23903) "app.openfolio.com"
23904) "no.wikipedia.org"
23905) "www.paris.fr"
23906) "www.techpowerup.com"
23907) "abuse.web.com"
23908) "puroresusystem.wikia.com"
23909) "www.cqnews.net"
23910) "tgpsaigon.net"
23911) "mwstake.org"
23912) "94.25.70.110"
23913) "1981coolkid1981.blog.163.com"
23914) "celebrations.smh.com.au"
23915) "surgeons.org"
23916) "library-opac.birmingham.gov.uk"
23917) "www.franklyinc.com"
23918) "help.elements.envato.com"
23919) "bits.wikimedia.org"
23920) "fapmc.ru"
23921) "www.cbs.gov.il"
23922) "www.businessesforsale.com"
23923) "scid.sourceforge.net"
23924) "diversity.berkeley.edu"
23925) "eric.ed.gov"
23926) "www.highcut.co.kr"
```



```
23928) "englisch.sueddeutsche.de"
23929) "www.htsec.com"
23930) "www.sciencepartnerjournals.org"
23931) "www.paypal.me"
23932) "www.law.cornell.edu"
23933) "www.businessinsider.com.au"
23934) "anotandofutbol.blogspot.com"
23935) "www.thefetch.co.uk"
23936) "top.hatnote.com"
23937) "www.nhlbi.nih.gov"
23938) "clinicaltrials.gov"
23939) "www.chinaunionpay.com"
23940) "wx.house.163.com"
23941) "www.elpaisclubdevinos.com"
23942) "rg.ru"
23943) "civichall.org"
23944) "br.wikimedia.org"
23945) "foersom.com"
23946) "pirosik.blog.pravda.sk"
23947) "noaa.gov"
23948) "www.liarumantscha.ch"
23949) "www.bridgetterrylong.com"
23950) "193.30.112.134"
23951) "auto.sina.com.cn"
23952) "subscribe.theglobeandmail.com"
23953) "economicsone.com"
23954) "demo.istat.it"
23955) "prepare.umd.edu"
23956) "c.m.163.com"
23957) "travel.cnn.com"
23958) "www.gs.emory.edu"
23959) "www.genealogy.ams.org"
23960) "www.myclassifiedads.net"
23961) "buzzfeedbooks.tumblr.com"
23962) "forum.oxforddictionaries.com"
(0.95s)
```

Figure 5: Search Engine

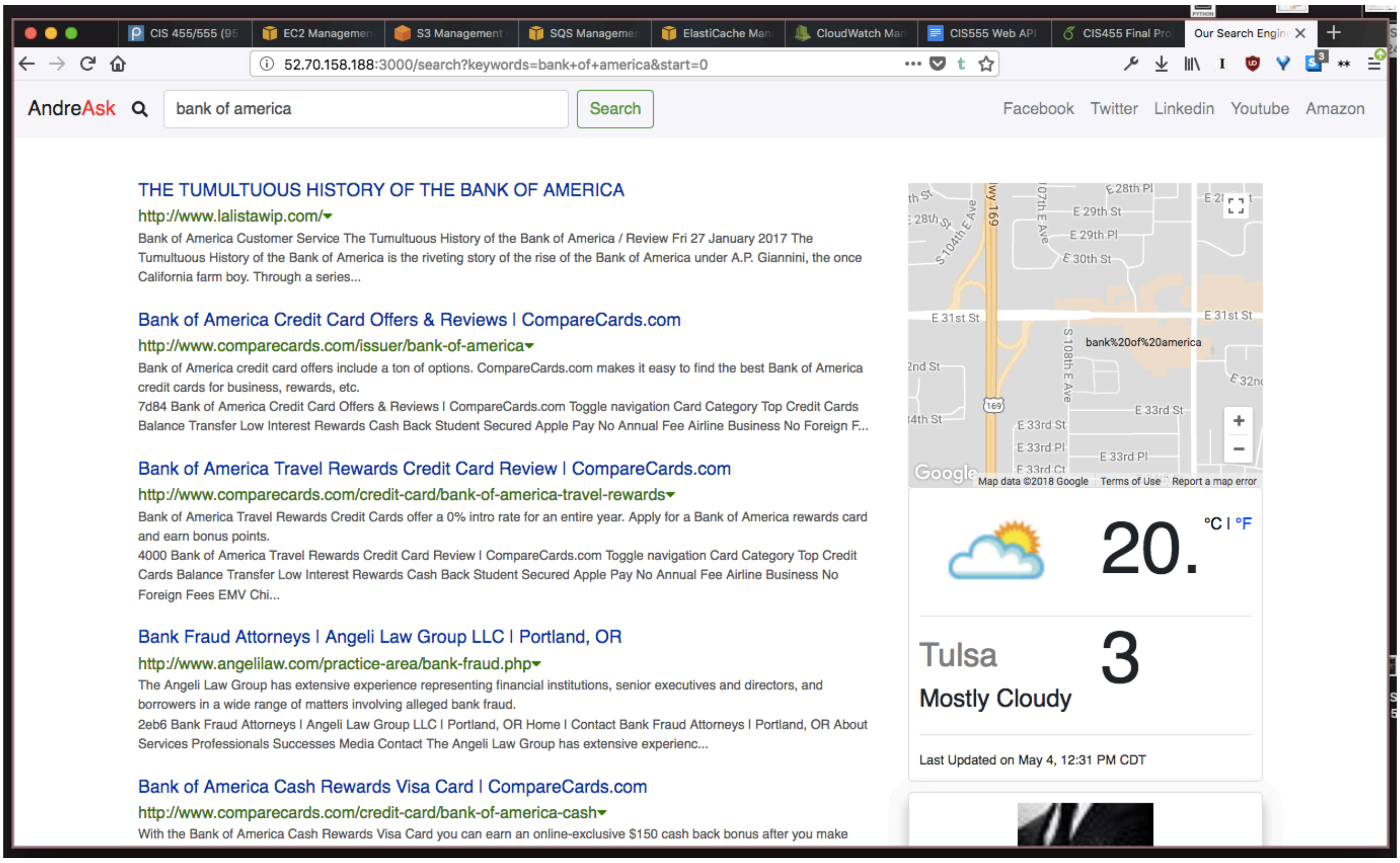


Explore The World From Here

Search something interesting - Use AndreAsk

What do you want to know?

Powered by Paul Asselin, Yuru Wang, Zichuan Yu, Lichang Xu © Copyright 2018



THE TUMULTUOUS HISTORY OF THE BANK OF AMERICA
http://www.lalistawip.com/
Bank of America Customer Service The Tumultuous History of the Bank of America / Review Fri 27 January 2017 The Tumultuous History of the Bank of America is the riveting story of the rise of the Bank of America under A.P. Giannini, the once California farm boy. Through a series...

Bank of America Credit Card Offers & Reviews | CompareCards.com
http://www.comparecards.com/issuer/bank-of-america
Bank of America credit card offers include a ton of options. CompareCards.com makes it easy to find the best Bank of America credit cards for business, rewards, etc.
7d84 Bank of America Credit Card Offers & Reviews | CompareCards.com Toggle navigation Card Category Top Credit Cards Balance Transfer Low Interest Rewards Cash Back Student Secured Apple Pay No Annual Fee Airline Business No Foreign F...

Bank of America Travel Rewards Credit Card Review | CompareCards.com
http://www.comparecards.com/credit-card/bank-of-america-travel-rewards
Bank of America Travel Rewards Credit Cards offer a 0% intro rate for an entire year. Apply for a Bank of America rewards card and earn bonus points.
4000 Bank of America Travel Rewards Credit Card Review | CompareCards.com Toggle navigation Card Category Top Credit Cards Balance Transfer Low Interest Rewards Cash Back Student Secured Apple Pay No Annual Fee Airline Business No Foreign Fees EMV Chi...

Bank Fraud Attorneys | Angeli Law Group LLC | Portland, OR
http://www.angelilaw.com/practice-area/bank-fraud.php
The Angeli Law Group has extensive experience representing financial institutions, senior executives and directors, and borrowers in a wide range of matters involving alleged bank fraud.
2eb6 Bank Fraud Attorneys | Angeli Law Group LLC | Portland, OR Home | Contact Bank Fraud Attorneys | Portland, OR About Services Professionals Successes Media Contact The Angeli Law Group has extensive experienc...

Bank of America Cash Rewards Visa Card | CompareCards.com
http://www.comparecards.com/credit-card/bank-of-america-cash
With the Bank of America Cash Rewards Visa Card you can earn an online-exclusive $150 cash back bonus after you make

20.3

Tulsa
Mostly Cloudy

Last Updated on May 4, 12:31 PM CDT