# CIS555 G03 Final Report

**Zean Zhou, Zhe Cai, Jiayue Wu, Rui Li**

## 1. Introduction

### ○ Project Goal

- Create a Google-style search engine named "PennInSearch" consisting of 4 key components: Crawler, Indexer, PageRank, Search Engine and Web User Interface.
- Get familiar with AWS products including EC2, S3, RDS, EMR, etc.

### ○ High-level Approach

The Crawler adopts the Mercator-style and is designed to be both multithreaded and distributed. It parses HTML and PDF files and stores the raw content and metadata in S3 buckets and RDS tables, respectively. It is deployed on an EC2 instance.

The Indexer computes the tf-idf values of each term through the Hadoop framework. It uses the interfaces of the crawler to get the content of each document and export the result data into 2 different RDS tables. It is deployed on an EC2 instance.

The PageRank also uses the Hadoop framework to compute the PageRank values of each URL given the data crawled by the Crawler. It runs for more than 20 iterations and exports the result data to an RDS table. It is deployed on EMR.

The Search Engine uses the interfaces designed by the other 3 parts to get all the information. Then compute the final ranking score for every possible URL given the query. It will list the first 30 results in the web page. If the query contains some place name, weather information would be displayed as well. It is deployed on an EC2 instance.

### ○ Milestones

- Apr 14: Setup the S3 buckets, EMR, EC2, RDS, and other AWS things.
- Apr 24: All the parts could run on a small dataset.
- Apr 30: Started crawling final corpus.
- May 3: Integrated all the parts.
- May 6: Improved ranking and added several additional features

### ○ Division of Labor

- Zean Zhou: Crawler.
- Zhe Cai: Indexer.
- Jiayue Wu: PageRank.
- Rui Li: Search Engine and Web User Interface.

## 2. Architecture and Implementation

### ○ Crawler

#### ■ Components

The crawler is composed of three main parts. The first one is the crawling component. Each crawling component runs in a specific thread, it first query the robots.txt about the crawling-delay and "disallowed" restriction (supports wildcard now). If the check is passed, it then issues a HEAD request to the destination website and if the size is appropriate (< 10MB) and the content-type is of interest (html/pdf), a GET request is followed by. After that, the crawling component filters out non-English documents and extracts URLs using JSoup. These URLs with interested extensions will be passed to the next component.

The second part is the dispatching system. This part decides to which thread / remote node this url should be assigned. For now, we use the Java string hash of the domain name of each url to decide the destination of each crawling job, which naturally enables consistent assignment so that each crawling thread is able to follow a specific robots.txt restriction of some domains. If the URL is finally assigned to the local node, the dispatching system will put this url to the queue of that assigned thread only if the queue of that thread still has space available. If the in-memory queue is full (set to 50000 per crawling thread), the url will be redirected to an on-disk persistent queue, which is implemented based on Berkeley DB. A daemon thread is working in the background to move the URLs back from the persistent queue if the in-memory queue has some new slots. This persistent queue also enables the crawler to pause and continue. When the crawler is restarted, it will first fill the in-memory queue with the URLs from the persistent queue.

In order to control and monitor the behaviors of our crawler, the third part is introduced - the remote controlling system. This system consists of a worker, which runs on each crawling node and then a master, which runs independently on a master node. Administrators are able to dispatch new seed URLs, monitor each crawler node's runtime statistics (bytes downloaded, html crawled , pdf crawled, in-memory queue length, on-disk queue length, etc) and even send stop signal to a crawler node, through a web interface running on the master node. This system is implemented from scratch based on UDP. Each well-behaved crawler node is supposed to send out a HeartBeat Packet to the master node to register itself, and the master node will broadcast a MasterBroadcast Packet with all nodes' information to all crawler nodes. With this information in hand, each crawler node is able to dispatch URLs to other nodes directly through the dispatching system.

#### ■ Storage

During crawling, the crawling component running on each thread writes the crawled data to two separate storage systems, RDS and S3.

RDS is responsible for URLs' metadata, the associated table can be described as follows.

**DOCUMENT_TABLE**

| Url TEXT | urlId VARCHAR(40) | docId VARCHAR(40) | blockName VARCHAR(40) | blockIndex INT | lastCrawledTs BIGINT |
|---|---|---|---|---|---|

urlId and docId are SHA-1 hash values of the url and raw document bytes, respectively.
blockName is the name of the block stored in S3, and the blockIndex is the index of the document stored in that block. lastCrawledTs is the timestamp of the time when this url gets crawled.
urlId is used as the Primary Key, and we build additional BTree Index on docId to enable fast query of docId.

Amazon S3 is responsible for document contents of different URLs. Amazon S3 is a simple key-value storage system, where we use the blockName as the key, and the binary block file itself as the value. The structure of the binary block file can be described as follows.

**CONTENT_BLOCK**

| | | | | | |
|---|---|---|---|---|---|
| docTotalCount int | docId byte[20] | docLength int | docType short | urlId byte[20] | Offset int |
| | docId byte[20] | docLength int | docType short | urlId byte[20] | Offset int |
| | docId byte[20] | docLength int | docType short | urlId byte[20] | Offset int |
| | … … … … ... | | | | |
| | docId byte[20] | docLength int | docType short | urlId byte[20] | Offset int |
| | Document Content [0] byte[] (starting at Offset[0] of length docLength[0]) | | | | |
| | Document Content [1] byte[] (starting at Offset[1] of length docLength[1]) | | | | |
| | … … … … ... | | | | |

The content block is composed of a header section and data section. The header section contains the total number of documents in this block, and each document's docId (SHA1 of document content), each document's length in bytes, docType (0 for HTML and 1 for PDF), urlId (first seen urlId of this doc) and an offset indicating the starting position to read. The data section contains all document's binary data sequentially.

Each content block is at most 200 MB, usually containing about 800 - 1500 different documents.

Using the docBlockName and docBlockIndex in RDS, one can easily load the corresponding header info from S3, and then use the range header in the HTTP request to query a single document. Using the urlId in S3, one can easily query the URL metadata from RDS. Using the docId in S3, one can easily query a list of URLs metadata that is associated with the same docId from RDS.

(Quick facts: SHA1-value is 160 bits = 20 bytes = 40 hexadecimal characters, that's why RDS uses 40 bytes but content block needs only 20 bytes for the same thing)

- **Performance**

Due to the high throughput and performance requirement of the crawler itself and the following processing by Indexer and PageRank, some performance boosting techniques are introduced.

To accelerate the access of RDS, we use HikariCP - an SQL Connection Pool Library. HikariCP maintains a pool of MySQL connections internally and provides callers with idle connections on request. When there's no idling connection, the pool will automatically queue the incoming queries and assign them to connections after they're released by previous queries. A good pool size in our experience is 20.

As for the content blocks, we add documents into the block and upload right after it is full. One way to do this is to make all crawling threads write into the same block at a time, which hurts the performance a lot because the adding operation is required to be synchronized and each thread is waiting for the lock. The other way to do this is to make all crawling threads write into its own block, which could potentially consumes a lot of disk space (# of threads in worst case * block size). And most blocks may get uploaded at roughly the same time which occupies all upload bandwidth in an instant. The final decision is to make every 10 threads write into the same block to maintain the balance.

- ## Indexer

The Indexer uses the Hadoop MapReduce framework to compute the desired results. Before starting, a text file containing all the S3 buckets' names generated by the Crawler is exported. Every line in the file is a bucket name.

The mapper does the heavy job to fetch the bucket, iterate through the bucket to get the documents, and parse them. There are mainly 2 kinds of files, HTML files, and PDF files. They are handled by Jsoup, and PDFBox, respectively. The text string is generated, then all the punctuation, as well as non-English letters, or digits are removed. Stopwords were removed before and after stemming to ensure accuracy. In order to increase the weight in the head tag, including titles, keywords, etc, these words would be added to the whole corpus of this document for 20 times. In the end, the term-frequency value (TF) of every word in a specific document is computed. The intermediate key-value pair that the mapper sends to the reducer is the word, docid + "&" + TF value. For example, the key is "abc", the value is "a2c6f5f67fb116731abf2d7ae31a1e763ce589ca&0.26952820395717997".

The reducer takes care of the computation of inverse document frequency (IDF) and exports the result to AWS RDS tables and AWS S3 bucket. The useful information is actually in RDS. 2 Tables are created in RDS, one is for IDF, and the other if for TF. Below is the structure of the tables.

**IDF_TABLE**

| | |
|---|---|
| word VARCHAR(50) | idf DOUBLE |

**INDEX_TABLE**

| | | | |
|---|---|---|---|
| entryId INT PK | word VARCHAR(50) | docId VARCHAR(40) | tf DOUBLE |

The whole project was first intended to deploy on AWS EMR. However, there were several weird errors or exceptions that always interrupted the job. Thus, it is deployed on an EC2 instance instead.

The Indexer provides an interface for the Search Engine to get all the desired results. The method takes in a query string and returns a list of word index entities.

- ## PageRank

The pagerank part also utilized the hadoop mapreduce framework to enable efficient computation. Before the actual job starts, I preprocessed the crawled data stored in S3 and convert them into <url, (list of outgoing urls)> pairs. This part could also be integrated into the map-reduce step and the reason to separate it is to ensure the integrity of the input data so we don't need to preprocess it again if one of the later steps crash. Actually the final implementation for the preprocess map is also achieved by a separate mapreduce job, in which the input is the block names of S3 and the output is <url, (list of outgoing urls)> pairs stored in another S3. During the preprocessing step, I designed the block size to be roughly 35MB each for 1M crawled pages and each block contains about 3000 url pairs.

After the preprocessing, we take in the new urls pairs data and start the pagerank algorithm The whole pagerank job comprises three map-reduce jobs. The first job takes in <line number, <url, (list of outgoing urls)> > and the mapper1 outputs <url, outgoing url> to the reducer1. The reducer 1 takes into <url, outgoing url> and appends the initial pagerank value 1to each url and it outputs <url, pagerank, (list of outgoing urls)>. Notice that it's easy to modify this part to be domain-based, where the final pagerank will be evaluated on each domain instead of each urls. The mapper2 takes in <line number,  <url, initial pagerank,  (list of outgoing urls)> > and outputs <outgoing url, fromUrl, rank, #of total outgoing urls>. The mapper2 also outputs <fromUrl, [outgoing urls]>, which is to save the information about the connectivity between urls for later use. The reducer2 will do 2 things: one is to calculate and update the url pagerank and the other is to check if the url is a dangling link. If the url is a dangling link, we will save it and not output it to the next iteration. The mapper2 and reduce2 is the main part for the iteration and we will iterate until converge, which took about 20 iterations for our task. The third mapper does light work. It first splits the value and gets the <pagerank, url> pairs and then generates a random value between 1 to 3000. It outputs <random number, <pagerank |url >>. The use of random keys here is that I want to split the job for writing to RDS to reduce the write request into RDS. About 500 <pr, urls> will be assigned to reducer 3 and the reducer3 will write them into RDS together. The schema of the RDS is URLID(sha1 of the urls, varchar(40), urls and pagerank.

**PAGERANK_TABLE**

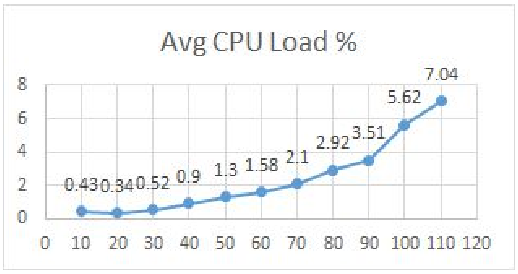
| | | |
|---|---|---|
| URL_Id VARCHAR(40) | URL(TEXT) | pagerank(FLOAT) |

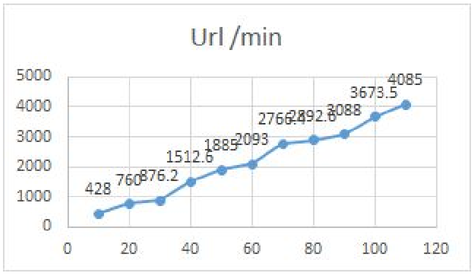
- ### Search Engine and Web User Interface

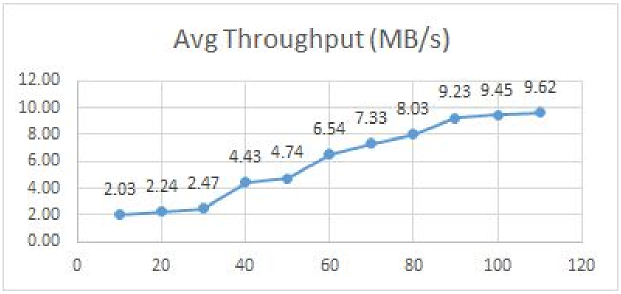
  The web interface was implemented using the Spark Java framework. The dynamic contents would be filled by the search engine. The search engine would take care of fetching results from all the other components of the project, and computing the final weight of each result to rank the results. First, the search query would be passed to the search engine through http get request made by the user. Then, the search engine would hand this query to the indexer to get TF and IDF scores. Since there are many pages which have exactly the same content, the content of the pages would be in the storage as an inverted index keyed with the document id. The indexer would perform a stemming on the words in the query to better match web contents, and send back the scores to the search engine. The search engine would maintain a list of objects with the IDF of each keyword in the query and also a map of each document id which contains the document responding to this keyword with its TF score. The search engine would pack all the document ids together and send them to the crawler database to consult the urls corresponding to the document ids. Once the search engine got all the urls, the urls would be sent to the PageRank database to get pagerank scores. The search engine would perform normalizations on the TF scores and IDF scores and multiply them together as the final search weight after. The search candidates with the top search weights could be picked up, and be sent to the crawler database to get the web content of each one. Then, the search engine would subtract the title of each url and check if there is any matching in the title and the keywords in the query. In all of the keywords that were matched, it would go to the top list, and is going to be displayed at the top of the page. Partial matched results would go into the middle part, and all the rest results would go to the bottom. Finally, the search engine would generate the dynamic contents and return them to the web interface, thus the web interface could have a whole html document.

## 3. Evaluation

For the crawler part, we mainly evaluate the performance vs number of threads. The crawler is deployed on EC2 t2.xlarge instance, with 4 cores and 16G memory. As the chart shown below, it shows a dramatic increase in the number of URLs crawled per minute over the number of threads. Some non linear gap in between might be caused by unbalanced hashing, ie. multiple famous domains are mapped to the same thread and that thread is busy handling these pages and some other threads are not fully occupied. We stopped the test as we found the CPU load is all 100% at number of threads = 110.







For the indexer part, the running time for about 1 million documents on an 8-core, 32G memory EC2 is about 8 hours. The IDF_TABLE contains more than 10 million entries and the INDEX_TABLE contains more than 500 million entries in the end.

For the pagerank part, we utilized EMR to run the main pagerank part with 8 X5.large instances(1 master and 7 slave nodes, each has 4-core, 16G memory ). And it took about 1 hours to run 6 iterations with 1M pages. However, the emr encounters strange errors when we later use the 1.45M-pages data. So the final version is deployed in one X5.2xlarge EC2 instance(8-core, 32G memory ) and took about 1.5hour to run 1 iteration for 1.45M data.

For the Frontend part, all operations except network connections could be completed within 0.5 s on an 8-core, 32G memory EC2. The final ranking of search results are reasonable.

## 4. Conclusions

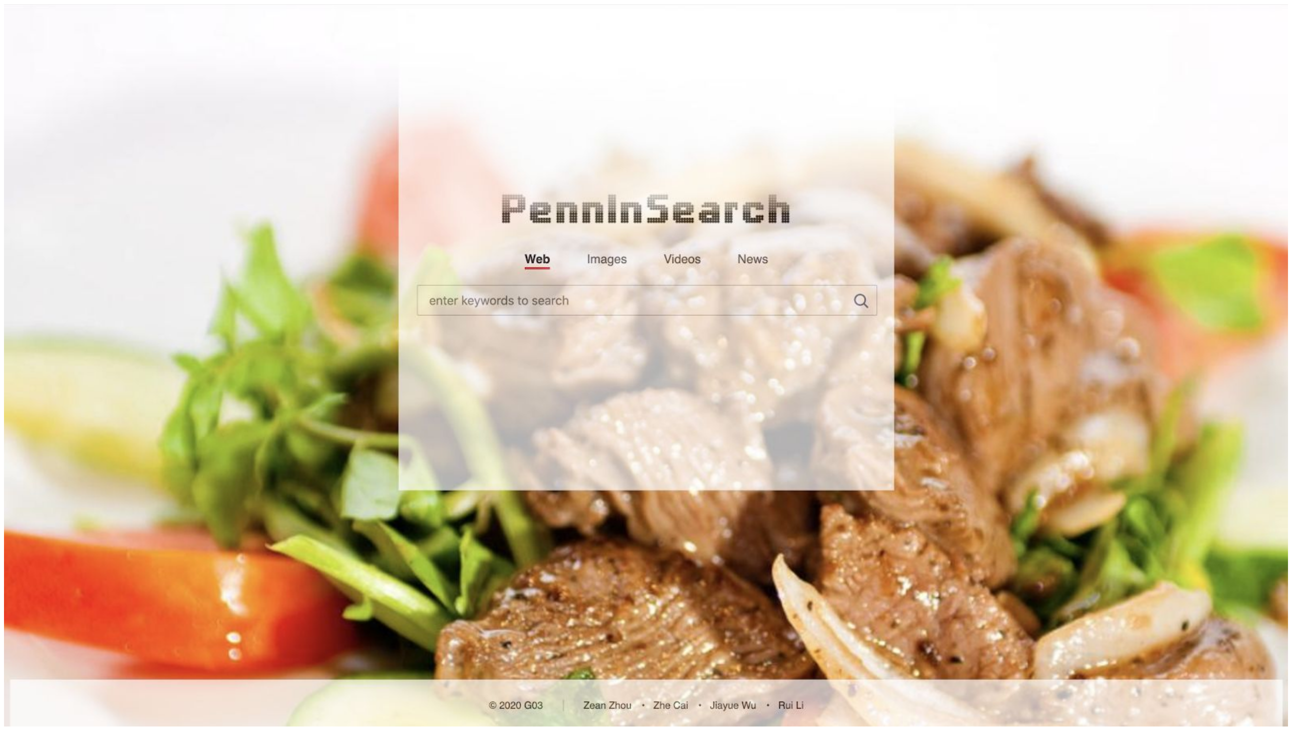
The project goals were achieved successfully. We have more than 1 million URLs crawled and processed. All the parts work well together. The results of queries are fairly good.

We learned a lot from this project and there are still several parts could be improved:

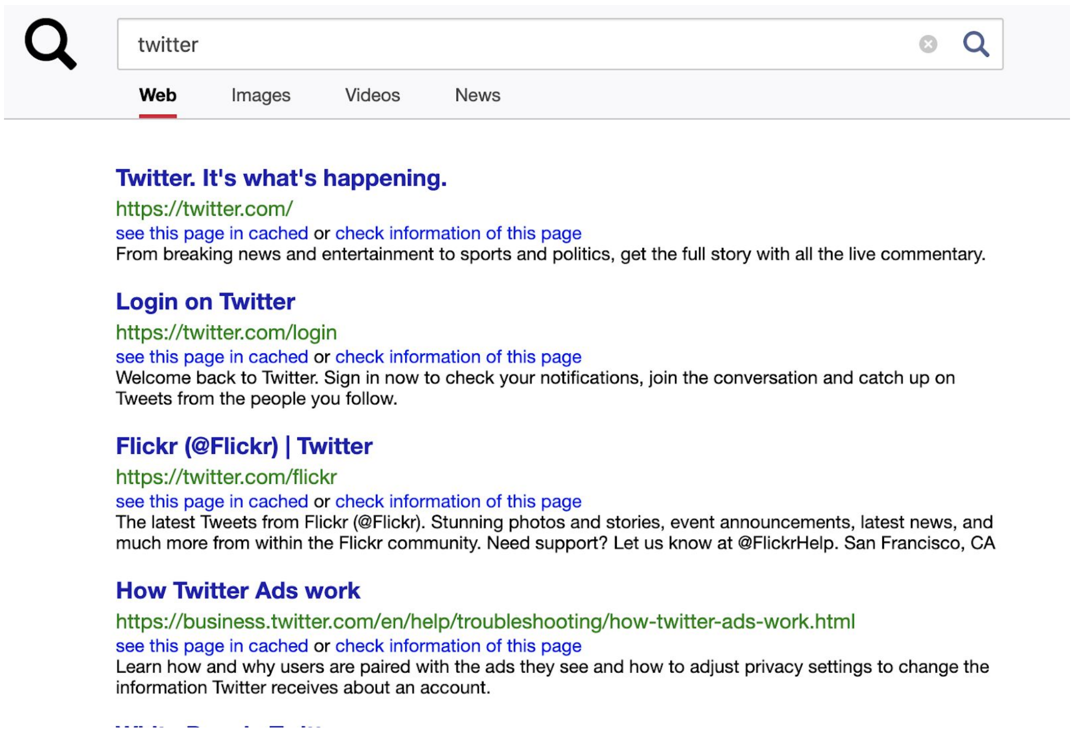
- Communication is important. We wrote down some similar code in different parts, which is unnecessary and might increase the processing time during indexing and responsive time during querying.
- Think big before implementation. Some of our code is not working on a larger scale, problems like out of memory / out of disk storage happen a lot at the beginning, which wasted a lot of our precious time to tune the actual algorithm.
- Adjusting the ranking is tricky given only index and pagerank values.
- Mini-batch testing is important, which can avoid a lot of bugs when deployment in large volume. However, even with testing locally in advance, you should still prepare for the unexpected error when dealing with large data, such as memory leak or connection failure.
- Backup plan is important. For example, it may be better to keep the old data unless you are sure that the new data is correct and useful.

# Appendix (Screenshots)

## Main Page



## Sample Query 1



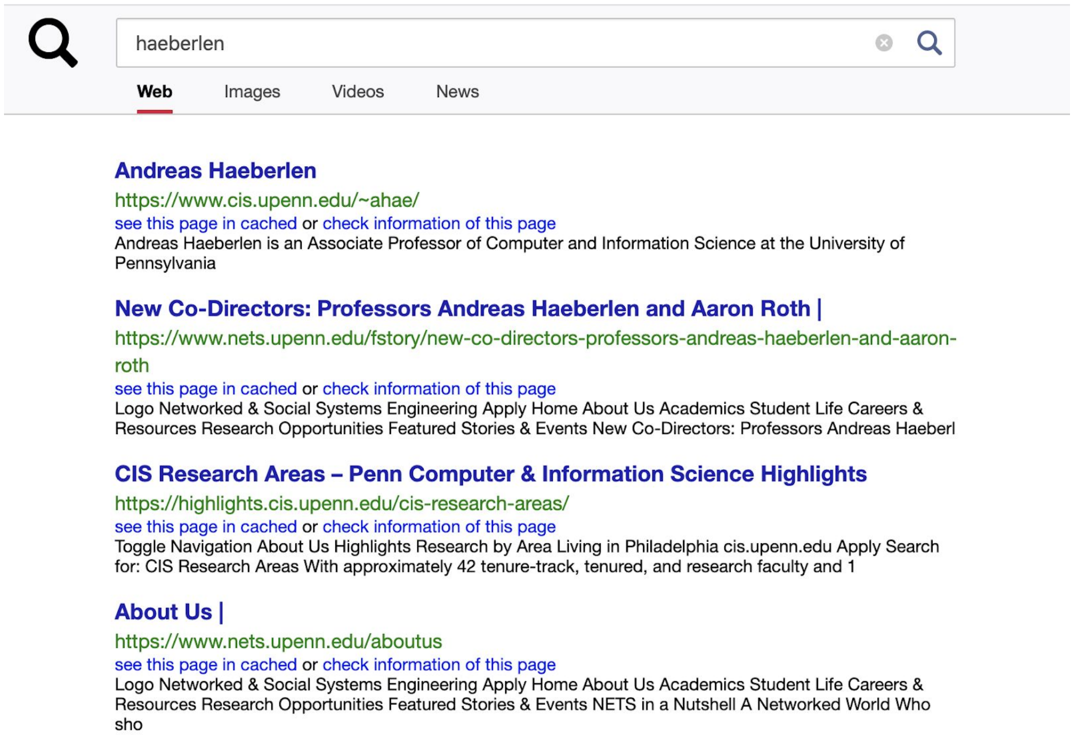
**Twitter. It's what's happening.**
https://twitter.com/
see this page in cached or check information of this page
From breaking news and entertainment to sports and politics, get the full story with all the live commentary.

**Login on Twitter**
https://twitter.com/login
see this page in cached or check information of this page
Welcome back to Twitter. Sign in now to check your notifications, join the conversation and catch up on Tweets from the people you follow.

**Flickr (@Flickr) | Twitter**
https://twitter.com/flickr
see this page in cached or check information of this page
The latest Tweets from Flickr (@Flickr). Stunning photos and stories, event announcements, latest news, and much more from within the Flickr community. Need support? Let us know at @FlickrHelp. San Francisco, CA

**How Twitter Ads work**
https://business.twitter.com/en/help/troubleshooting/how-twitter-ads-work.html
see this page in cached or check information of this page
Learn how and why users are paired with the ads they see and how to adjust privacy settings to change the information Twitter receives about an account.

**Sample Query 2**



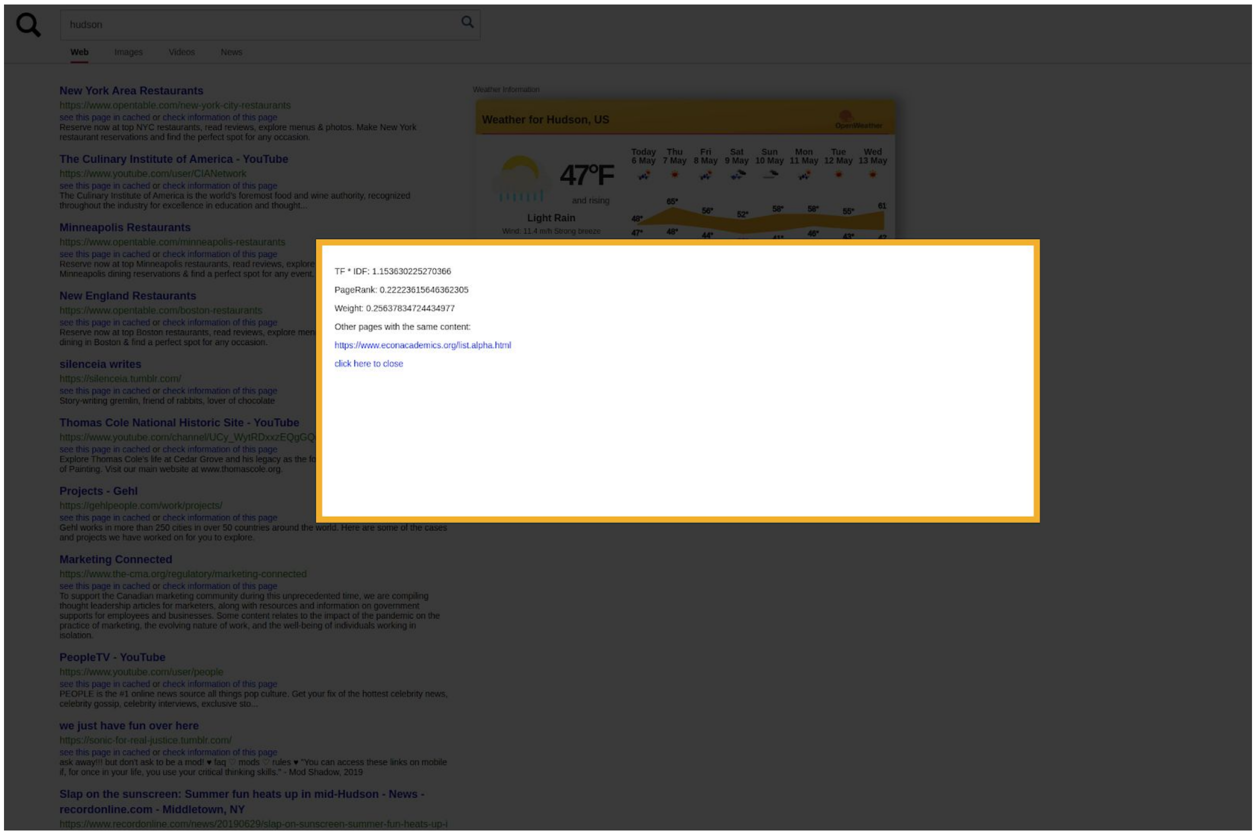
**Andreas Haeberlen**
https://www.cis.upenn.edu/~ahae/
see this page in cached or check information of this page
Andreas Haeberlen is an Associate Professor of Computer and Information Science at the University of Pennsylvania

**New Co-Directors: Professors Andreas Haeberlen and Aaron Roth |**
https://www.nets.upenn.edu/fstory/new-co-directors-professors-andreas-haeberlen-and-aaron-roth
see this page in cached or check information of this page
Logo Networked & Social Systems Engineering Apply Home About Us Academics Student Life Careers & Resources Research Opportunities Featured Stories & Events New Co-Directors: Professors Andreas Haeberl

**CIS Research Areas – Penn Computer & Information Science Highlights**
https://highlights.cis.upenn.edu/cis-research-areas/
see this page in cached or check information of this page
Toggle Navigation About Us Highlights Research by Area Living in Philadelphia cis.upenn.edu Apply Search for: CIS Research Areas With approximately 42 tenure-track, tenured, and research faculty and 1

**About Us |**
https://www.nets.upenn.edu/aboutus
see this page in cached or check information of this page
Logo Networked & Social Systems Engineering Apply Home About Us Academics Student Life Careers & Resources Research Opportunities Featured Stories & Events NETS in a Nutshell A Networked World Who sho

**Debug Info After Clicking on "check information of this page"**



TF * IDF: 1.153630225270366

PageRank: 0.22223615646362305

Weight: 0.25637834724434977

Other pages with the same content:

https://www.econacademics.org/list.alpha.html

click here to close

**New York Area Restaurants**
https://www.opentable.com/new-york-city-restaurants
see this page in cached or check information of this page
Reserve now at top NYC restaurants, read reviews, explore menus & photos. Make New York restaurant reservations and find the perfect spot for any occasion.

**The Culinary Institute of America - YouTube**
https://www.youtube.com/user/CIANetwork
see this page in cached or check information of this page
The Culinary Institute of America is the world's foremost food and wine authority, recognized throughout the industry for excellence in education and thought...

**Minneapolis Restaurants**
https://www.opentable.com/minneapolis-restaurants
see this page in cached or check information of this page
Reserve now at top Minneapolis restaurants, read reviews, explore
Minneapolis dining reservations & find a perfect spot for any event.

**New England Restaurants**
https://www.opentable.com/boston-restaurants
see this page in cached or check information of this page
Reserve now at top Boston restaurants, read reviews, explore men
dining in Boston & find a perfect spot for any occasion.

**silenceia writes**
https://silenceia.tumblr.com/
see this page in cached or check information of this page
Story-writing gremlin, friend of rabbits, lover of chocolate

**Thomas Cole National Historic Site - YouTube**
https://www.youtube.com/channel/UCy_WytRDxxzEQgGQ
see this page in cached or check information of this page
Explore Thomas Cole's life at Cedar Grove and his legacy as the fo
of Painting. Visit our main website at www.thomascole.org.

**Projects - Gehl**
https://gehlpeople.com/work/projects/
see this page in cached or check information of this page
Gehl works in more than 250 cities in over 50 countries around the world. Here are some of the cases and projects we have worked on for you to explore.

**Marketing Connected**
https://www.the-cma.org/regulatory/marketing-connected
see this page in cached or check information of this page
To support the Canadian marketing community during this unprecedented time, we are compiling thought leadership articles for marketers, along with resources and information on government supports for employees and businesses. Some content relates to the impact of the pandemic on the practice of marketing, the evolving nature of work, and the well-being of individuals working in isolation.

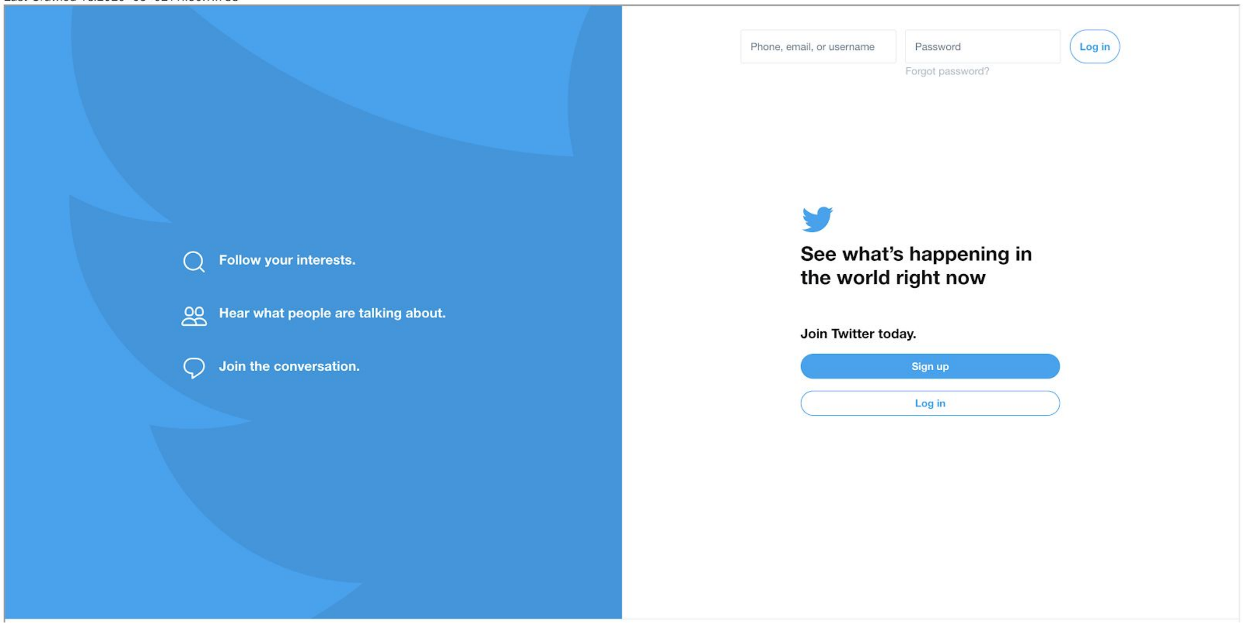
**PeopleTV - YouTube**
https://www.youtube.com/user/people
see this page in cached or check information of this page
PEOPLE is the #1 online news source all things pop culture. Get your fix of the hottest celebrity news, celebrity gossip, celebrity interviews, exclusive sto...

**we just have fun over here**
https://sonic-for-real-justice.tumblr.com/
see this page in cached or check information of this page
ask away!!! but don't ask to be a mod! ♥ faq ♡ mods ♡ rules ♥ "You can access these links on mobile if, for once in your life, you use your critical thinking skills." - Mod Shadow, 2019

**Slap on the sunscreen: Summer fun heats up in mid-Hudson - News - recordonline.com - Middletown, NY**
https://www.recordonline.com/news/20190629/slap-on-sunscreen-summer-fun-heats-up-i

**Cached Page**

URL:https://twitter.com/
Last Crawled Ts:2020-05-02T11:30:11.788



Follow your interests.

Hear what people are talking about.

Join the conversation.

See what's happening in the world right now

Join Twitter today.