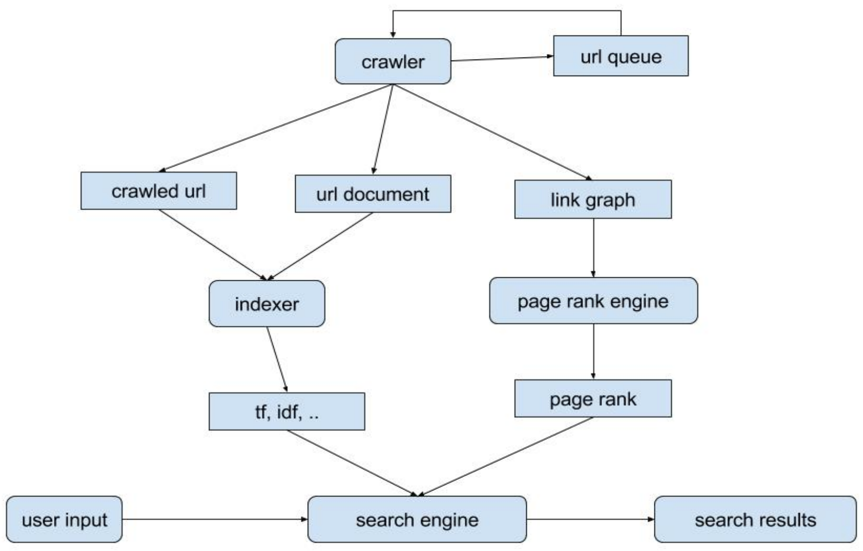
Search Engine Project Report

Yibang Chen | Tianle Yang | Yucong Li | Siyang Shu

I. Introduction

The search engine is composed of 4 independently implemented backend components, Crawler, Indexer, PageRank, and the Search Engine, and 1 frontend user interface for the Search Engine. Each component maintains its own database, which becomes data source for the rest components. Such design ensures that each component may run periodically independent of the rest for better efficiency and performance.



Design choices:

- CloudDB vs LocalDB: we have another option which is to store data in BerkeleyDB across different EC2 instances. We chose S3 and DynamoDB for simplicity and distributed features
- MapReduce vs Multithreading: we multithreaded indexing for portability. The efficiency feature of mapreduce has in fact become the bottleneck of indexing in a careful analysis (see indexing).

II. Evaluation

Our search engine is partly scalable:

1. The crawler as well as indexer has many threads and they work separately. We could add more machines to run more threads doing more work. The only common resource shared by crawler is the url queue, and for indexer is the idf file. These are not bottleneck because the time fetching data from url queue or writing data to idf file is relatively small to the time spent on the actually work.
2. The page rank is scalable because it use Spark to process the data.
3. DynamoDB has limited data read/write and became a major bottleneck (see indexing and search engine).

Crawler Implementation

The crawler is designed based on the Mercator crawler. It's a scalable, multithreaded and state-restorable crawler that is written completely in Java. Each crawler instance can be run simultaneously on multiple machines with the only interaction in AWS S3 database. Each instance has their own queue management system (UrlFrontier). The web pages are mainly in English by applying a language detection library: https://github.com/shuyo/language-detection/blob/wiki/ProjectHome.md.

The architecture of the crawler is composed of the following components:

- A UrlFrontier

The UrlFrontier contains the information of a queue to be crawled by the crawler. The queue is composed of four components which are wrapped in a java class with the following information: the parent URL that the to be crawled URL is extracted from, the current URL to be crawled, the depth of this URL from the root and the anchor text that leads to the current URL from the parent URL. This preserve the information needed to restore crawler state when the crawler is stopped from a crash or is manually stopped something went wrong.

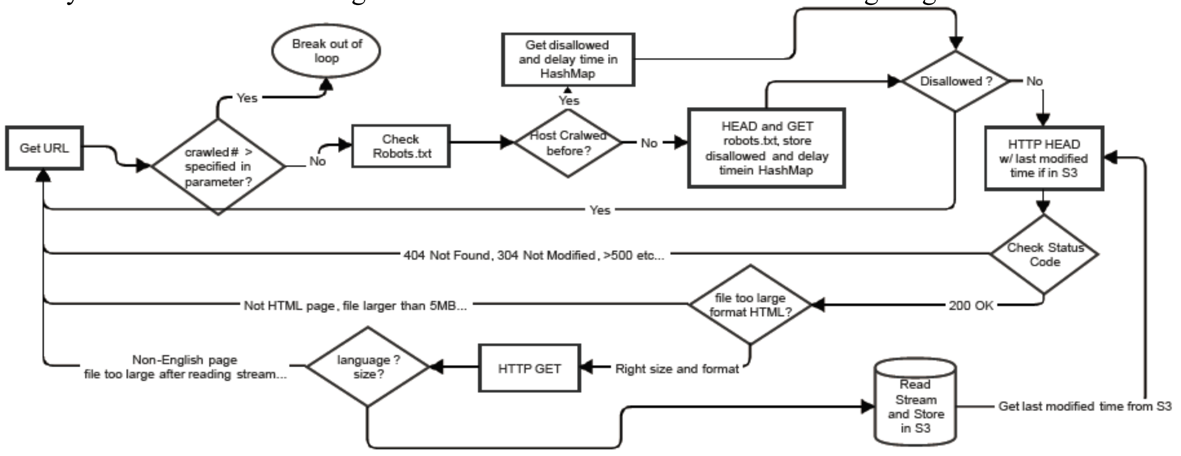
To ensure fast access to the queue, the queue is initially stored in Java memory. However, after some initial testing of the crawler, the queue has become too large to be stored in memory when the queue size is scaled up to gigabytes. Therefore, an alternative solution is proposed to store the queue, with the main memory only has 60,000 queue items. Once the queue has reached the limit, all the items extracted from the webpages are appended in a local queue file. The file is in the following format: <parent URL>\t<URL to be crawled>\t<depth>::::<anchor text>. At this point, when all the item in the queue is processed by the crawler, one of the crawler worker that detects the queue in memory is empty will load 60,000 items (or number of item in the queue file) into the memory by parsing the above format.

To start the crawler, a queue file is needed in the format same as above to be stored in aws S3 storage, with bucket name and filename stated in code. The crawler first pulls and read the queue and then load the first 60,000 into memory and the rest into a local file queue file. The URL frontier would check if item in queue is larger than 60,000 to determine future queue item to be listed in memory or appended to the queue file directly.

The UrlFrontier also includes a hashset of URL crawled in current run, so they don't get to crawl twice in very short period of time.

- A crawler worker

The crawler worker is the part that sends HTTP request and stores the crawled webpages into database. Each crawler worker implements the Runnable interface in java so that multithreading is made possible for the system. The flow of a single crawler worker is shown in the following diagram.



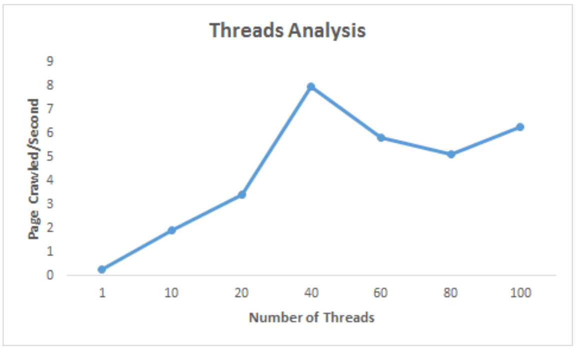
Break out of loop
Get disallowed and delay time in HashMap
Disallowed ?
No
Yes
Yes
Get URL
crawled# > specified in parameter?
No
Check Robots.txt
Host Cralwed before?
No
HEAD and GET robots.txt, store disallowed and delay timein HashMap
HTTP HEAD w/ last modified time if in S3
Yes
404 Not Found, 304 Not Modified, >500 etc...
Check Status Code
Not HTML page, file larger than 5MB...
file too large format HTML?
200 OK
Non-English page file too large after reading stream...
language ? size?
HTTP GET
Right size and format
Read Stream and Store in S3
Get last modified time from S3

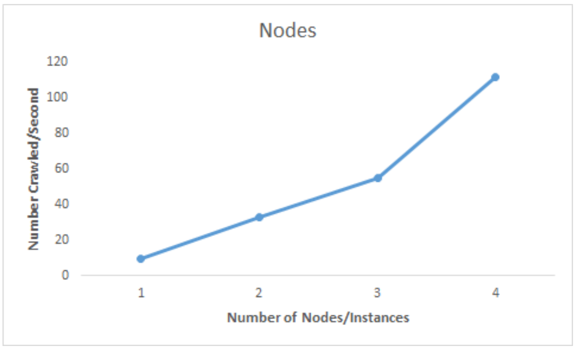
Each worker thread is wrapped in a while loop to consistently retrieving items from the queue, until it is interrupted by the stopping thread or the queue is empty. To serve the purpose of this project, only html pages are crawled.

- A stopper thread (A fault tolerance system)

The stopper thread is designed to smoothly and safely stop all crawler worker threads so that the queue and crawled urls files can be stored and uploaded to the cloud (S3). The thread has a Scanner that is always waiting for a ‘q’ input from the console/terminal. Unless a ‘q’ is entered, the whole system is running no matter whether all the crawler threads are stopped. The benefit of this implementation is that the state of the crawler can be restored easily by just run the program again without any additional operations. Also, it serves as a backup mechanism so that queue in memory can be eventually retrieved out to file when the crawler workers all crashes.

Evaluation:



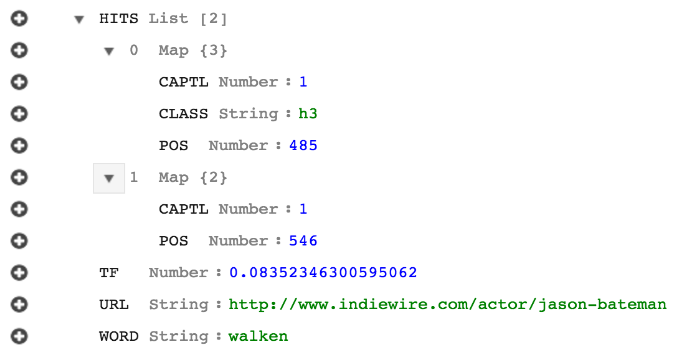


Two experiments about the crawler are performed with the results shown above. The figure on the left represents the effect of number of threads to the speed of crawling. The experiment is conducted on a local machine (with the vm provided by cis455) on Penn’s network. The starting root of the crawling is from http://en.wikipedia.org. By increasing threads on a single node, the overall performance on the crawling is increasing with linear speed. However, it seems that the increment encountered saturation around 40 to 60 threads. This is caused by a design decision that when the queue reaches 60,000 items, it will need some time to process the queue I/O operation as stated above. Therefore, when the threads reaches 100, it compensates for the I/O time and tends to increase the speed in a linear fashion again. The graph on the right represents the number of nodes to the speed of crawling. The experiment is set up across 4 EC2 medium instances with 100 threads each. With the first instance starts with http://en.wikipedia.org, second with http://www.nytimes.com/, thrid with http://www.cnn.com/ and the last from https://www.yahoo.com/. Apparently, the crawling speed also depends on the server speed. As shown in the graph, the relationship between number of nodes/instances and number crawled per second is also linear. Hence, the crawler system, although don’t have interaction with each other, is scalable on the level of number of instance.

Indexer Implementation

The indexer processes data from 2 sources, a S3 database of 800k crawled html documents and an URL queue of the crawled documents, stored as local text file (58MB).
Indexing outputs the following data:

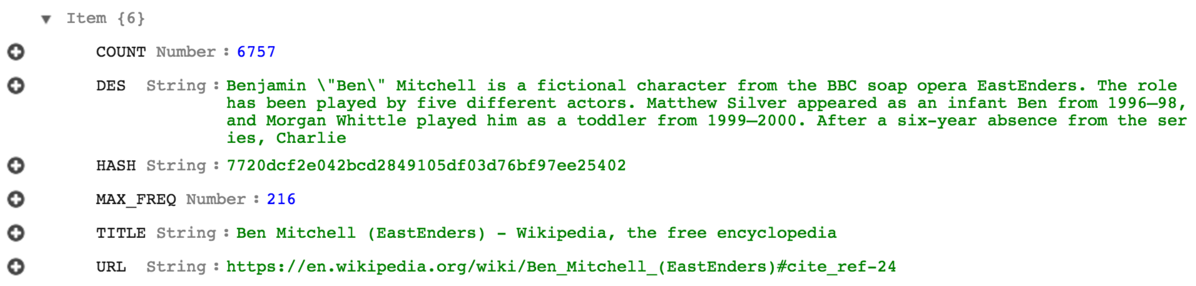
1. A hits list and a term frequency for each word in a document. See figure 3.
2. A text file of idf scores, i.e. the count of documents where each indexed term appeared in. The file is updated periodically.
3. For every document the indexer will generate some useful information when doing the parsing like document title, description, hash code and so on, and store it in dynamodb. See figure 4.
4. The search engine needs relevant urls for any given word. To make the searching process faster, we have the function to return only a subset of the most relevant urls. This is done by store the tf value with url as the sort key so that dynamodb can sort and return top results.



```
▼ HITS List [2]
  ▼ 0  Map {3}
       CAPTL  Number : 1
       CLASS  String : h3
       POS    Number : 485
  ▼ 1  Map {2}
       CAPTL  Number : 1
       POS    Number : 546
  TF    Number : 0.08352346300595062
  URL   String : http://www.indiewire.com/actor/jason-bateman
  WORD  String : walken
```

1.

Figure 3: A typical hits data stored in dynamodb.



```
▼ Item {6}
    COUNT Number : 6757
    DES   String : Benjamin \"Ben\" Mitchell is a fictional character from the BBC soap opera EastEnders. The role
                   has been played by five different actors. Matthew Silver appeared as an infant Ben from 1996–98,
                   and Morgan Whittle played him as a toddler from 1999–2000. After a six-year absence from the ser
                   ies, Charlie
    HASH String : 7720dcf2e042bcd2849105df03d76bf97ee25402
    MAX_FREQ Number : 216
    TITLE String : Ben Mitchell (EastEnders) - Wikipedia, the free encyclopedia
    URL   String : https://en.wikipedia.org/wiki/Ben_Mitchell_(EastEnders)#cite_ref-24
```

Figure 4: A typical document information stored in dynamodb.

The phrase search is done by storing two-word phrase in addition to single words. We remove meaningless phrase after parsing by requiring each phrase should occur at least twice in one document. We also use multithreading instead of MapReduce to do indexing. This choice is for performance after side-by-side comparison. Then We found the indexer's bottleneck is DynamoDB but not CPU power so we didn't write a new MapReduce version.

The tf algorithm is as follows:

$$f(i,j) = a + (1-a) * freq(i,j) / max(freq(l,j))$$

Instead of native counting, we implemented Term Frequency by assigning term weight depending on where it appears in a document. For example title terms are weighted 3 times that of body while h1/h2/h3-tagged terms are worth 2 times of body. We also take word's capitalization into account whose weight is between 1 and 2. We have also made another tweak: if the squared sum of all the words in one documents exceeds 5, we'll make them smaller linearly. This is because for any document, if every word occurs only a small times, then `max(freq(i, j))` becomes very small thus every word has a large tf value.

This architecture is scalable if we assume the Dynamodb is not the bottleneck. The indexer has many threads and they work separately. We could add more machines to run more threads doing more work. The only common resource shared by indexer is the idf file. It is not the bottleneck because the time writing data to idf file is relatively small to the time spent on actually work, and we could also write the idf locally and merge these files at the end.

Other implementation choices includes:
We use porter algorithm as the stemmer.
We use stop words provided by http://www.ranks.nl/stopwords.
We didn't do indexing on numbers. The pros is many number is meaningless and the cons is we can't search by number like "1776".
We use jsoup to parse the html.

From performance point of view, the framework indexes 15 documents per second, with each document containing 550 hits items.

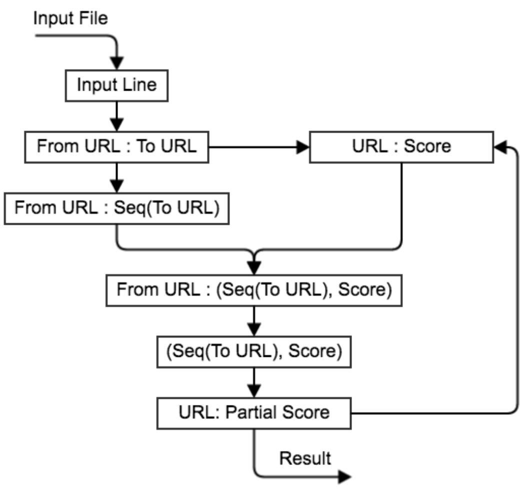
PageRank Implementation

I. Architecture

Under our implementation, the final PageRank score of a web page is based on three factors: the ranking of the page (PageRank), the ranking of the host server (Host Rank), and the ranking of the root domain (Domain Rank). Score is injected into the entire graph every iteration to offset the sunk scores, and hogs situation is handled by removing self-pointing links. To boost the ranking speed, our PageRank program is written in Java as a Spark application.

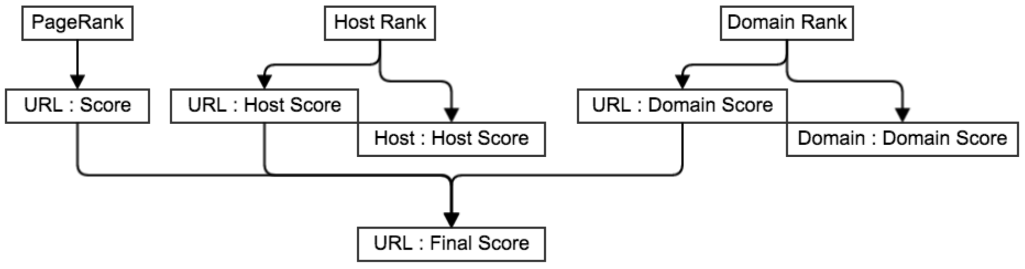
II. Framework Selection & Algorithms

The single iteration of PageRank fits Hadoop MapReduce well, but for multiple iterations, massive I/O operation from Hadoop will dramatically slow down the entire process. Spark, on the other hand, caches data aggressively, and with other optimizations, it is about ten times faster than Hadoop MapReduce implementation.



Our PageRank implementation starts by following the Random Surfer Model described in the PageRank paper. The decay factor d is set to typical value 0.85. As we proceed, we found the number of dangling pages is huge. So we inject scores to each page in every iteration to offset the score lost. We tuned the score to inject to 0.6, so that the score after 20 iterations of each page is in a reasonable range, though it will not converge.

For Host Rank and Domain Rank, links from same source pointing to the same destination is counted repeatedly to better reflect the influence. We combined the three ranks by taking the logarithm of each of them and calculate the sum, which magnified the difference near zero but compressed the difference between larger numbers. We also added a constant 0.15 to each logged result to make the number positive.



III. Input and Output & MapReduce

Since MapReduce framework works best with the key-value pair input, we store the input graph as a list of edges, which is represented as <URL FROM><TAB><URL TO>. The list is stored across 79 files in the same directory on Amazon S3. Each line of the file includes only one edge, and act as the basic unit of input to JavaRDD.

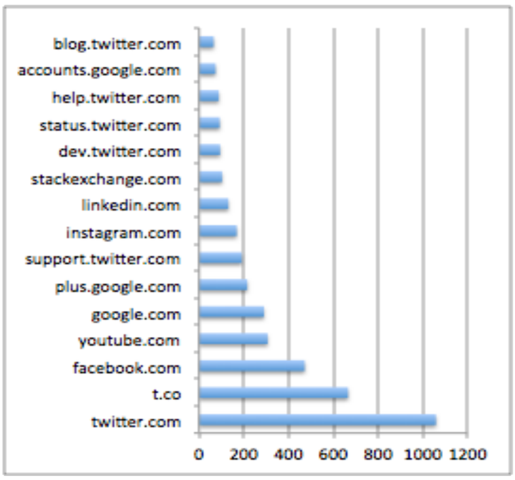
Six output are generated during these steps. They are stored in S3 buckets, all of which are candidate ranks for the Search Engine. PageRank score and Host Rank score are transferred to AWS DynamoDB. The Domain Rank score file is loaded by Searcher to achieve better access efficiency.

The PageRank progress can be described as the graph. Each arrow represents a transformation or an action. Host Rank and Domain Rank uses the similar flow.

IV. Evaluation

Some sample high score pages: http://programmers.stackexchange.com(297.58), http://espn.go.com/nhl(248.65), and https://twitter.com/bloomberg (113.00). We can see they are all well known pages.

Root domain rank describes the inter-website relations well. Since host/root domain rank uses the similar score injection technique to handle sinks, the minimum score of a host/root domain is also 0.75. After 20 iterations, 30 out of 87,945 root domains and 58 out of 159,109 hosts have scored more than 15.0 which is very high. As shown in table, almost all of these websites are very well known. One thing should be noticed is that social network websites get the highest scores because of the share links on a great amount of web pages.



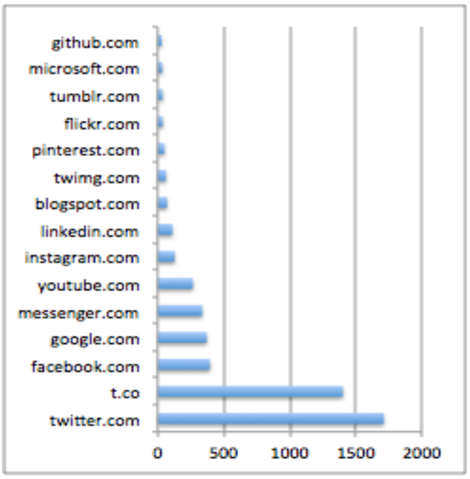


Figure: Top 15 hosts and domains

Spark Application Execution Time: The PageRank step consumes the most time. With one m3.xlarge master node and three m3.xlarge worker node, it takes less than 30 minutes.

Search Engine & User Interface Implementation

I. Design and result in brief

The User Interface is written in HTML and CSS and have Java Servlets to generate dynamic contents upon user's request. In addition, we implemented wrapper classes for easy access to database, language processing, and common functionalities.

II. Query intake and processing

In addition to stemming and stop-word check, the user's query is broken down it into sets of terms:
- Set of all terms: all single words and adjacent 2-word terms.
- Essential terms: subset of all terms ranked by idf terms limited to 6
- Single words: non stop-word list

A document's relevance score takes consideration of all terms' tf and idf while document-selection process only considers pages with at least one essential terms and all single words.

The search engine features a relevance ranking among all crawled documents basing on tf - the importance of a term is in a document, idf - the importance of a term in Internet, and pagerank, the importance of websites in Internet. The tf score is calculated and maintained along with word-documents hits in the indexing database; terms' idf scores and document pagerank scores are updated periodically through a text file. Those text files are read at the initialization of program and persists in Java Map for key-value searching.

In the searching phase, the engine first queries the indexing database in batch to match documents that contain at least one of the essential terms (see above). The count of documents to return is inverse proportional to the number of terms. From that initial document pool, the engine matches those that contains each single word (see above) in query at least once as "qualified documents". In the scoring phase, the engine queries the indexing, pagerank, and idf databases for a relevance score, base on which documents are ranked and cached in memory. The formula is:

```
relevanceScore = factor * indexScore * pageRank/ Math.sqrt(urlString.length());
```

* factor: a compound factor that takes consideration of term appearance in url, metadata, and title

Over all indexed url and words in the database, we have 2 filtering processes before relevance scores (i.e.) are calculated:

1. Pages of a word in the inverted index database are ranked by tf score (see indexing)
2. In the search engine, pages are classified by number of terms matched. URLs with all terms matched are given priority in scoring.

In this way, we are more likely to achieve better precision; recalled docs may be smaller in number.

III. User interface and usage analysis:

There are 2 major webpages in the user interface, each of which is achieved with a java servlet.

1. The /interface page (user entrance page) has a simple design with 1 user query input box and a list of clickable term list extracted from Twitter API. The Twitter API sends http requests to twitter and returns a random word out of 5 that are currently in the trend. The link directs to a /result page to search for it.
2. The /result page is directed from either from /interface page or another /result page with the previous page's information.

- The page navigator at the bottom of page allows the user to navigate through the search results by counts of 10. On click, the search engine reads results from memory instead of searching again.
- A search query input box is provided at top of page and displays the user's previous search query. The use can change query and submit a new search request from any page of results.
- The count of matched documents and time spent on searching is shown at the top of first search result. However, the count does not reflect the actual number of matched documents since we only rank top tf-scored documents of essential terms as explained in section II.
- To render formatted html result back to client, the search engine queries the indexing database for page titles and descriptions.

IV. Robustness experiment and analysis:

To test robustness of the search engine, we have conducted number crunching experiment with 100 top search queries from google trend of 2015* (https://www.google.com/trends/). On average, our search engine is able to generate 1289 relevant results in 3.2 seconds**, out of which scoring process accounts for 88%. That indicates that cloud database read is the largest bottleneck of performance and can be improved through better database choice and design. The length of query in fact has little impact on the search time while average result count is smaller when query is longer. That is because database overhead happens mostly in the staging (fetching documents) and scoring (filtering staging documents and ranking) period.

Query Length	1-word	2-word	3-word	Longer	Total
Avg Search Time***	2.4114	2.4702	2.5758	4.8618	3.1854
Max Search Time	6.444	4.1184	3.864	7.7568	7.7568
Min Search Time	0	1.0128	1.134	2.451	0
Avg Result Count	3543	689	534	370	1289
Max Result Count	5916	2490	1853	1742	5916
Min Result Count	0 ****	31	0	3	0
Avg Staging Count	3908	3507	3413	4678	3948
Avg Staging Time	0.2892	0.432	0.40668	0.43566	0.3954
Avg Scoring Time	2.2194	2.0358	2.1642	4.4226	2.8188
Total Count	25	33	12	30	100

* Complete search queries and intermediate processing code and logs are available upon request

** DynamoDB read capacity = 10000; running machine 1.86 GHz Core 2 Duo, 2GB memory

*** Time is calculated as second; count is calculated as per occurrence

**** Out of 100 search term, we are unable to get results for 3 queries: "George W. Bush", "Franklin D. Roosevelt", "Alprazolam".