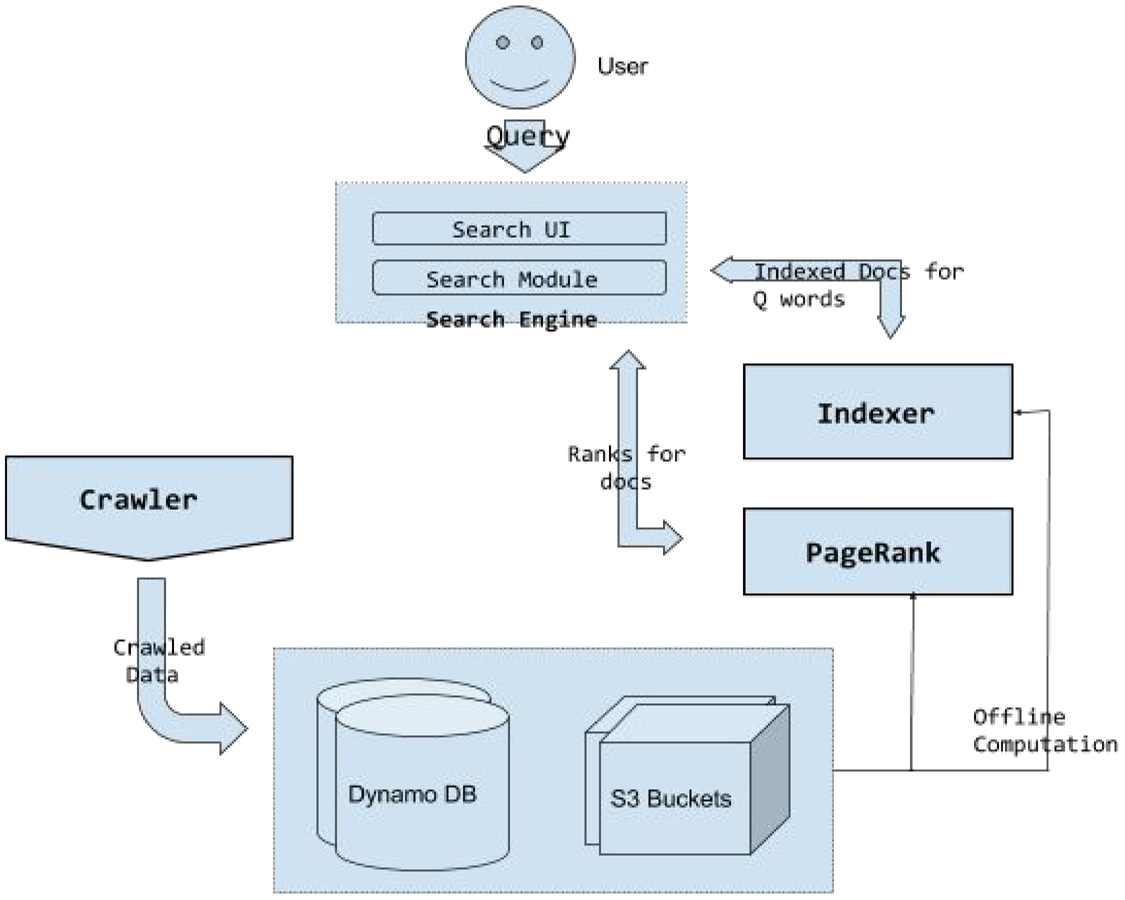
IWS Project Report

Dominik, Gouthaman, Ishan, Shreejit

Introduction:

Our project goal is to design a small, but fully functional implementation of a search engine, **"searchi"**, that runs in the Amazon Web Services cloud. The high-level design is as follows:



Searching for a query involves typing in the query on the Search engine UI, from which it is sent to the backend. The search engine first "purifies" the query linguistically before sending it as a list of keywords to the indexer. Based on the query words, the indexer determines all candidate documents that match any of the keywords as well as the document's TF-IDF-based similarity score to the search query. The ranked documents are then sent back to the search engine, that will fetch the pagerank scores for the ranked documents and produce a combined list of ranked documents that is finally sent back to the frontend module to be displayed to the user. The indexer and PageRank components both serve the search engine's requests based on precomputed tf-idf values and pagerank scores, respectively, that are computed using the crawled raw pages.

Github repository tracking progress: https://github.com/bollmann/searchi.

Project Components Overview:

Crawler (Owner: Shreejit). The crawler is the input to the search system and supplies url content and metadata to the system. We had initially chosen the Mercator style for it's many benefits, but we kept running into performance bottlenecks. We had an ambitious target of having at least 500,000 crawled url pages, and so after repeated attempts, we decided on a different crawler approach, in which each node would only crawl urls within a subset of domains, and discard all links going out of the domains it was initialized with. This way, we eliminated latency of inter worker communication, as well as removed the bottleneck of a central master, but still kept the crawler domain-friendly. Here however we had a stricter control of what was going to be crawled. In the end, with this architecture, over the whole course of the project, we were able to crawl more than **2 million** urls in total, although only 450,000 of them were indexed in the end. We went through multiple iterations of content and language filters, and in the end we arrived at a suitable system which left urls that let to good search results.
In addition to this, we also use disk-backed queues for greater resilience of the crawled state giving us the option of resuming crawls, as well as keeping the memory usage of the crawler nodes in check.
In the end, we also enhanced the crawler to use Apache Tika for parsing PDFs, and even though it could easily be extended to other types, we favored improving search results over extending content types our search engine would index.

Indexer (Owner: Dominik). The indexer consists of an offline and an online component. Based off the crawled raw pages, the offline component precomputes the inverted index and associated data structures using Amazon's Elastic Map-Reduce Framework. Upon completion, the precomputed tables are then imported into DynamoDB for easy and fast query access. The indexer's online component is built as a servlet-based interface that queries the precomputed inverted index based on the search engine's HTTP requests. Upon an incoming search query, it looks up all the candidate documents matching the query's keywords and sends these documents together with their TF-IDF scores as well as further associated features back to the search engine.

PageRank (Owner: Ishan). The overall pagerank system consists of a module that does an offline computation of page ranks as a MapReduce job and provides an interface backed up by a cache to get the computed pageranks. The system also computes similar scores on domain level - 'DomainRanks'. These serve as a better measure of domain authority. This system is responsible for ensuring importance of different links.

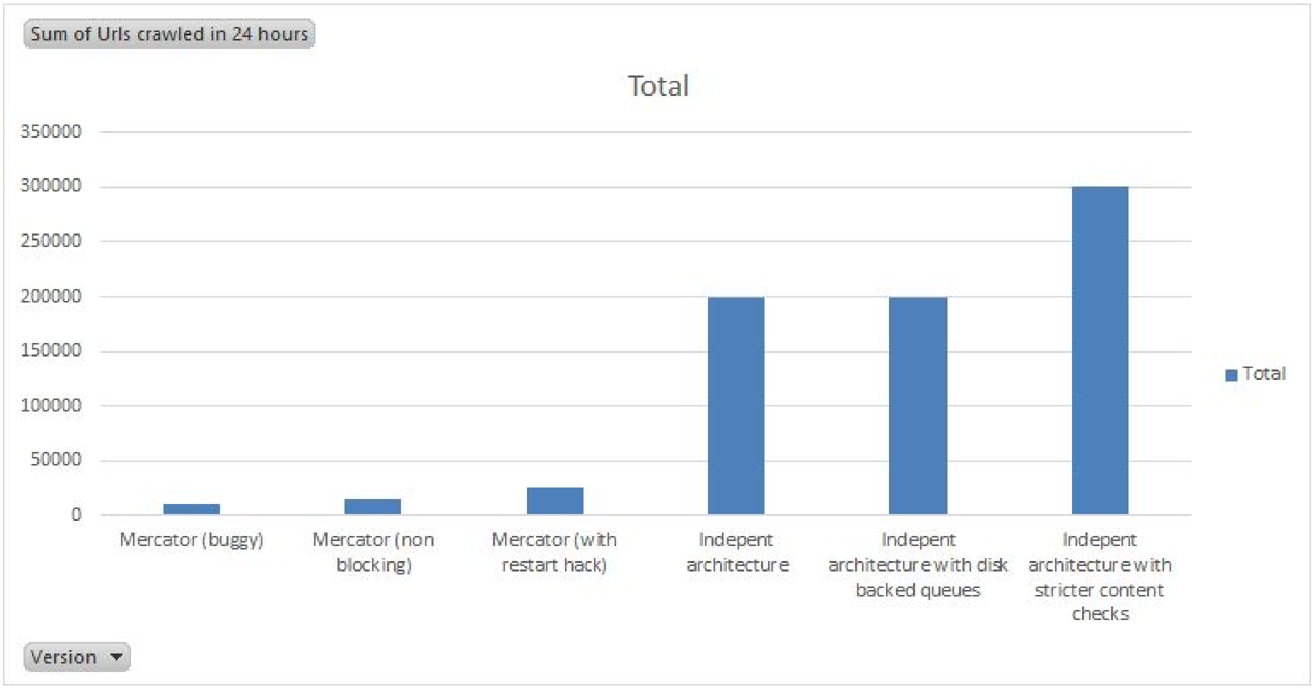
Search Engine (Owner: Gouthaman). The search engine is a Node.JS webapp backed with a Bootstrap framework. The search page accepts a query from the user, and the app performs linguistic cleanup on the search string (such as stop word removal, NLP to determine the keywords in the query) and calls the Indexer with the relevant keywords. In the case of image search, it passes the search string as tags of the image**.** With the document set returned by the Indexer, it calls upon the PageRank module to obtain the ranking of the documents. Using a combination of both the PageRank and TF-IDF scores and various other semantic features, the top ranked results are displayed to the user. AJAX calls are used to dynamically load relevant snippets of each URL.

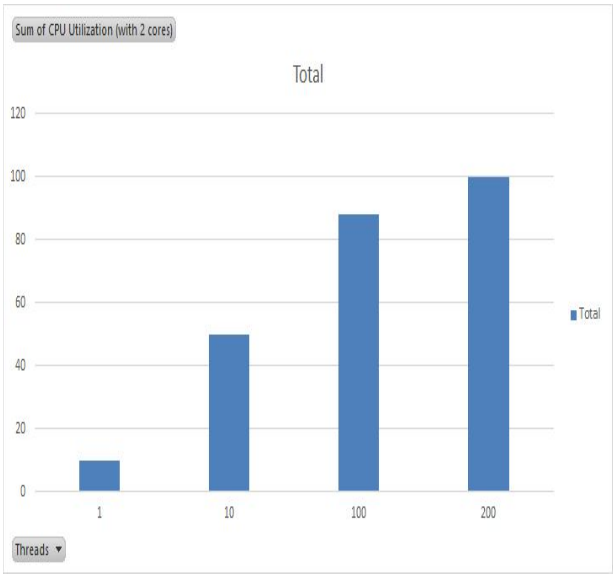
Architectural Details - Implementation & Evaluation

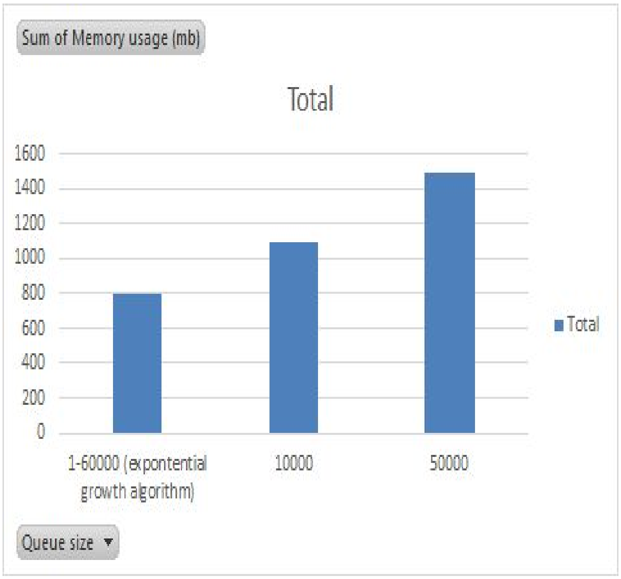
1.Crawler

The crawler was implemented as a distributed system made up of independent crawling units responsible for the list of domains each node is initialized with. It discards urls belonging to domains outside this list, as well as content that fails the preliminary english language and content filters (the indexer has a stricter check that is more resource intensive).
To identify the crawlable content types, the crawler builds upon the httpclient built earlier and uses Apache Tika to identify and read different content types, though we only support for html and PDFs for now.
To identify if a crawled page's content is adequate, we have a less resource intensive method of checking the content for language and pornographic content, and discard the url if it fails either of these.
We were running into memory constraints, and so to ensure longevity of runs, we had to build a generic disk-backed queue. Upon hitting the "push-limit" it would push the existing queue to s3 and update its queue info. If the push-limit is not reached, all queue operations are done in memory.

Performance and Evaluation







A number of different things were changed and tweaked during the course of crawling for the project. Different disk backed queue sizes were tried to optimize memory usage on medium instances of the crawler. A completely different architecture had to be used to crawl the number of web pages we were aiming to crawl. The level of multi-threading was also changed on each crawler node to maximize CPU utilization on medium nodes (to make the crawls cost effective, we used medium instances which were the best balance of memory and CPU vs cost).

2. Indexer

The indexer's offline component precomputes three central tables based off the crawled web pages:

1. The DocumentIndex, which maps a document's id to its URL.
2. The InvertedIndex, which associates each word with the list of document ids that this word occurs in together with a set of distinctive features for the (word, document) pair. As features we store the TF-IDF weight, the positions of the word in the document, as well as the number of times the word occurs in the document's url, headers, links, and meta tags, respectively.
3. The ImageIndex, which maps an image's "textual description" drawn from its title or alt tags to the image's URL.

All three tables are first precomputed using Amazon's Elastic Map-Reduce Framework and afterwards imported into DynamoDB. During the EMR computation, we enforce different filters on acceptable words and documents in order to maintain a quality index. To avoid indexing too many foreign words, we only index pages whose content is at least 50% english. Moreover, we exclude porn results using porn stop-word lists.

To make the import of the indexer's tables into DynamoDB and more importantly querying the inverted index from DynamoDB as fast as possible, we take various performance optimizations:

1. All data in DynamoDB is stored as compactly as possible. To this end, we use three tables and the InvertedIndex table stores the features for each (word, document) pair in a custom, little space consuming encoding. Moreover, each word in the inverted index only stores the top 8000 pages wrt TF-IDF scores in order to use just two DynamoDB rows of 400KB space. Restricting the number of candidate documents per word greatly helps to reduce query latencies.

2. A custom import program imports all data into dynamo in batches of many (e.g., 4000) rows at a time and in a multi-threaded fashion. That way, throughput is bound only by the write capacity assigned to the DynamoDB tables, which is highly scalable.
3. A query's word lookups to the indexer's tables are batched and multiple queries are executed simultaneously in a multi-threaded fashion.

Evaluation
The final inverted index has **1,646,185** unigrams and **24,251,404** bigrams and hence contains a total of around 25 million rows. The candidate pages for search queries with up to 5 words are fetched within 0.3 seconds of latency. Building different sizes of the indexer's InvertedIndex table offline using EMR first and then importing them into DynamoDB takes the following times:

#crawled pages	inv index size	EMR time	Db Import time	total time
120k	860k rows (unigrams)	25 mins	40 mins	65 mins
450k	1600k rows (unigrams)	84 mins	55 mins	139 mins
450k	25000k rows (unigrams + bigrams)	124 mins	4:20hours	5:44 hours

As shown by the table, the sheer size of the InvertedIndex makes its import into DynamoDB a bottleneck during the offline computation. Nonetheless, using our optimizations we were able to improve the total import time to less than 6 hours compared to initially 24 hours for just 500k rows.

Extra Credit
1. Indexer support for image search.
2. Support for advanced indexing features in addition to TF-IDF weights. We index various contexts of a word hit in a document such a word's positions in a document, and its occurrences in a header, link, and meta tag. Furthermore, we support bigrams and trigrams in addition to unigrams in the indexer.

3. Search Engine / User Interface
The search engine consists of 2 modules:

Search Engine Backend
The search engine backend was implemented in Java to facilitate faster and more complex ranking, as well as making it a service so that the UI would always be up, but any changes in the ranking algorithm would only have to be updated on the backend. It is composed of the following sub-components:

Query Processor - This extracts information about the individual query words and bigrams of the query, and assigns each of them a weight based on the `Part of Speech` information. It then returns a weight for each token, and then queries the inverted index for each of these tokens. The query for the tokens is cached for faster load on subsequent queries.
Inverted Index Fetcher - After the tokens have been generated, they are sent to the multi threaded inverted index fetcher to retrieve the inverted index.

Ranking Engine - The ranking engine takes the documents fetched and executes weighted rankers like Tf-idf ranker, Url ranker, Meta and Links Ranker, Header ranker and so on, in parallel. These output scores for all the documents All rankers are pluggable in the ranking engine to facilitate easy modification to experiment with different weights
PageRank Fetcher - This module takes the ranked docs and uses the pagerank API to fetch pagerank scores from the cache or from the Dynamo
Combinator - The combinator then combines the indexer and page rank results with an empirically decided weighting factor.
Result Filter - After the combined result of the indexer and page rank scoring has been calculated, we run it through a result filter which only selects 2 urls belonging to the same domain, and checks for uniqueness of base url. This ensures diversity of the results to the user.

Search UI
The search engine frontend was implemented in Node.JS/Express, using the Bootstrap library for components. Node.JS was chosen primarily for its attunement with JSON, its highly scalable framework, and the vast array of non-blocking I/O operations available. The webapp consists of 2 main pages - a splash page and a results page. The splash page is static, but the results page loads dynamic content depending on the number and source of results. Apart from the crawled/indexed data stored in the cloud, support for the **Amazon Product Advertising API**, **Twitter Search API** and the **OpenWeatherMap API** is implemented.

Upon receiving a query from the user, the client-side JS analyses the query for certain keywords that would indicate if Amazon and/or Twitter search results would be appropriate. If the query contains words pertaining to weather, or climate, then the Weather API is called. The user can freely switch between all available sources of results. The local Searchi results (document or images) are always available.

If the user chooses to view documents or images from the Searchi database, the query is passed to the dedicated servlet. The servlet performs semantic analysis, querying, analysing and ranking of documents. We felt that separating the expensive ranking functions of the servlet from the lightweight UI functions of the webapp was essential.

The servlet returns a JSON object encoded with the results, which is then formatted by the frontend and dynamically updated to the user. Using AJAX, we then start client-side calls to the results to load the content snippets (with only the relevant portion being displayed to the user). Support for embedded media is provided as well.

Latency in answering some queries

Sample query	Indexer Fetch (ms)	Indexer Ranking (ms)	Total time to results (ms)
Barack Obama president	240	100	350
richest man on the planet	200	230	450
superman vs batman	200	100	330
functional programming	180	100	300

Extra Features Implemented

1. Third-party APIs: Amazon Product Advertising, Twitter Search, OpenWeatherMap (*displayed only when appropriate query is detected*)
2. Image Search: Complete with a gallery view/carousel
3. AJAX: Used to load results from different sources on the same page, paginate results, and to load relevant content snippets (*text or embedded media*) for each result. Adding a "favorite results" feature was considered, but we realised that it would be impractical without also implementing a user account/login system
4. Spell check: Checks spelling of query as it is typed, against a dictionary. Does not involve querying the database.

4. PageRank

The pagerank design has been modified from the initial idea to consist of two parallel components -

a. **PageRank** - computes and provides api access to individual page scores
b. **DomainRank** - computes and provide access to domain level pagerank scores

Each of the above components has 2 modules -

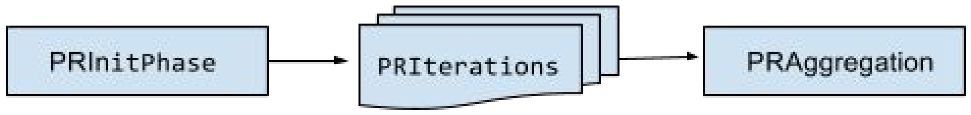
a. Offline Computation
b. API and Cache

Design

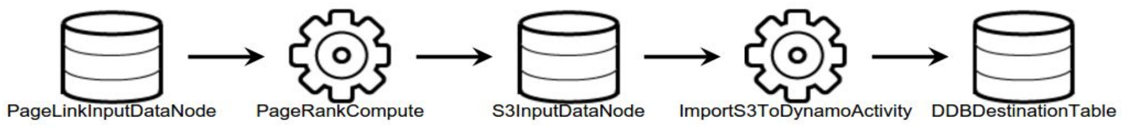
PageRank Component

Our final design is a **3 phase iterative** MapReduce Job. The 1st phase reads data from S3 in json, formats, *removes all self links* and assigns initial pagerank scores to the links. The 2nd phase has 2 mapreduce jobs, one for iterating and computing the pagerank score using the formula -

$$(1-d) + d * \left(\sum_{inlinks} inPageRank / outLinks \right)$$



The 2nd mapreduce job 2nd phase *calculates the accumulated sink scores for all sinks and distributes it to all the links in each iteration.* The 3rd phase aggregates the final pagerank scores. There is an AWS data pipeline that manages everything from EMR MapReduce Jobs to importing all the data in DynamoDB.



This component also provides APIs to access pagerank scores both individually and in batch. All the pageranks are stored in DynamoDB which are **cached in memory** on PageRank Component Bootup. The APIs internally do a **local cache lookup** for a request.

DomainRank Component
This component has a fairly similar architecture to pagerank component but also has logic to do all of the above steps on domain level. It also adds other domain level info in the final aggregation phase of the mapreduce jobs.

Performance Analysis & Evaluation

DomainRank- Offline Computation and Dynamo Import

# crawled pages	# links	# iterations	# instances	time - EMR Job	time - DB import	time -total
10000	157917	15	2 large EC2	8 min	7 min	15 min
60000	78593	15	3 large EC2	13 min	10 min	25 min
100000	140418	15	4 large EC2	18 min	15 min	35 min
200000	271298	15	4 large EC2	25 min	15 min	45 min
457961	601851	15	6 large EC2	38 min	25 min	70 min

The complete MapReduce Job was pretty fast and scaled efficiently for more data. We revised the code implementations to have efficient data formats to reduce any extra overhead other than pagerank computation. Also, initially the **API access** used to read from the Dynamo directly **(5-10 ms)** for 1 million entries but after in-memory caching we reduced it a lot to **<0.05 ms**. It takes 8-10 seconds of boot up time initially when the pagerank service is loaded.

Extra Credit

1. Domain Rank Computations in addition to Page Rank. Adding domain level features in the module. (implemented)
2. Extended HW3 MapReduce to take num of iterations as input and run the job for automatically for those iterations. Ran it on a sample size of 5000 pages. Didn't use it for the project as it was very slow. (discarded)
3. Wrote the pagerank as a Spark application handling self links and ran it on EMR using the spark configuration which was lightning fast for moderate amount of data as it was able to handle everything in memory but couldn't scale it up to the amount of data we had. (implemented)

Conclusions:

In the course of this project, we built a small but fully functional search engine running in the Amazon cloud. While we were initially struggling with performance and scale bottleneck across all four components, we improved our system over multiple iterations to crawl, index and pagerank a corpus of 500k documents and to efficiently answer queries over it. To handle scale on the crawler side, we had to overcome the synchronization issues between multiple nodes. To index huge corpora, we had to experiment with several different database layouts to efficiently store the data in order to allow for good import and query facilities. Lastly, to allow for low-latency query answering, we made extensive use of data batching and caching in the search engine frontend.
The exploration of the design space continuously improved our search engine and let us learn a lot!