Search Engine Final Report

Xinyi Lu, Guanwen Qiu, Yuxiao Tang, Yulin Yu

I. Introduction

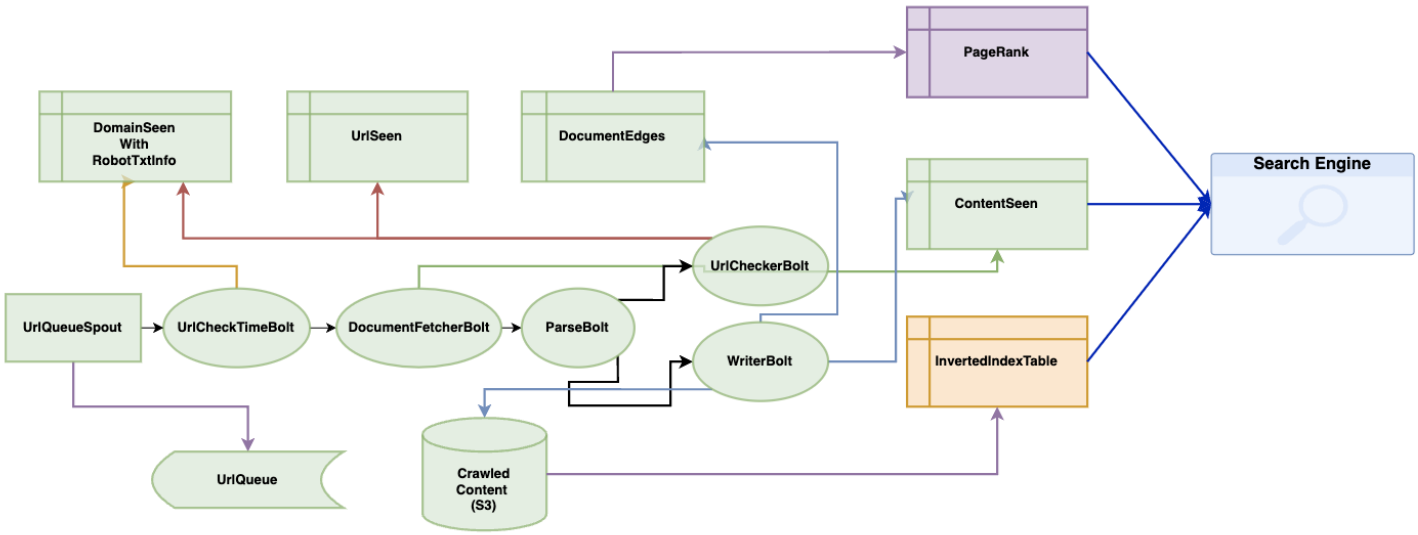
(1) Project Goals & High-level Approach

In our project, we aimed to build a powerful web search engine that utilizes what we have learned in this course. We designed a crawler that crawled one million diverse web documents using a distributed implementation of Stormlite, and used Hadoop to build a comprehensive lexicon that correctly maps words to documents to effectively answer queries. We also developed an effective PageRank algorithm to sort pages based on authoritativeness using Apache Spark and deployed a search engine that has a user-friendly interface and outputs search results indicative of authoritativeness & proximity to the given query within seconds. The search engine was deployed on AWS using Dynamodb and EC2, ensuring scalability of the system.

(2) Milestones

- 4/14 - 4/20
  - Understand each component of the architecture
  - Preliminary planning of API, database schema, ranking function and UI
- 4/21 - 4/26
  - Finalize database architecture and API
  - Functionality Implementation
  - Preliminary testing with sandbox
  - Start crawling and indexing
- 4/27 - 5/2
  - Continue crawling and indexing and monitor document quality
  - Integrate PageRank with crawler
  - Finalize ranking function and UI
- 5/2 - 5/7
  - Continue crawling and indexing and monitor document quality
  - Fully integrate PageRank and search engine with crawler and indexer
  - Final end to end testing
  - Deploy search engine on EC2
- 5/8 - 5/10
  - Finish Experimental Analysis and Final Report

II. Project Architecture



(1) Search Engine Architecture: After making a query, the front-end application will send the query terms to the search engine. The search engine will compute the TF/IDF score for each term and the page rank for each document. Based on the score and page rank, the search engine will query the database and send back the sorted result pages to the web app.

(2) Crawler/Indexer/PageRank Architecture:

- The crawler engine will crawl the appropriate pages from the Web and store them in the S3 bucket.
- The indexer will get documents information from S3 bucket, and use MapReduce to generate data structures. The indexer will store all the information about the inverted index, including each term, related DocId and TF-IDF scores into the DynamoDB.
- The PageRank program will also get documents from the database, compute the pagerank score for each document and store them in the database.

III. Implementation

(1) Crawler

We choose to implement a serverless, fully distributed crawler using **StormLite** provided in hw2. All data structures used for bookkeeping and url filtering are stored on DynamoDB. A message queue used for saving current frontier is hosted on **AWS Simple Queue Service(SQS)**. The actual text content of webpages are written to **S3**.

Our crawler has a master-worker structure. A master node is hosted on AWS Elastic Cluster (**EC2**) used for bookkeeping and monitoring. It can stop and start all connected workers at once with different crawling configurations provided by the user. A worker node will send its performance metrics such as the number of pages that have been crawled, the number of crawled pages in a minute and worker start time to a configured master periodically. We utilize Spark Java, a REST microservice to build our servers and define their APIs. A remarkable advantage of our design is that a worker node can join this system and start crawling at any time. And workers with different hardware can be configured in different ways to optimize their performance. The

number of pages crawled in a given time period is linearly proportional to the number of workers given that the throughput of data structures hosted on the cloud are unlimited.

The schema of the DynamoDB tables used by the crawler are listed here:

1. **ContentSeen***(DocID, ContentType, CrawledTime, LastModifiedTime, Url, Prelude, S3Id, Title)*
2. **DocumentEdges***(FromDocId, ToDocId)*
3. **DomainSeenWithRobotTxtInfo***(Domain, BadRequestTimes, CrawlDelay, DomainLastCrawledTime, RobotTxtCrawledTime, TotalAccessTime)*
4. **UrlSeen***(Url, CheckedTime*)

A crawler's workflow can be described as below:

1. **UrlQueueSpout** keeps popping messages from the SQS UrlQueue and sends the url to UrlCheckTime Bolt.
2. **UrlCheckTimeBolt** then checks whether the url received has passed the required crawl delay time specified in its domain robot txt by checking the DomainSeenWithRobotTxtInfo table. If the url is qualified, it will be sent to the DocumentFetcherBolt
3. **DocumentFetcherBolt** downloads the received content and checks whether the content has been seen by checking its MD5 hash value against stored MD5 values in the ContentSeen table. If the content has not been seen, its hash value will be stored to the table and sent to the parseBolt.
4. **ParseBolt** utilizes the JSoup package to parse webpage content it receives. It first extracts the title, text and first 400 characters as prelude and sends them to the WriterBolt. Then it extracts all the hyperlinks from the document and sends them to the UrlCheckerBolt.
5. **WriterBolt** caches a number of pages in a HashMap and writes them to S3 when the limit (specified in the configuration file) is reached. Web contents and its corresponding DocID are written as a txt files with user defined text separator to distinguish separate documents. Before contents are written to the database, related records and entries are written to the corresponding tables first. So every Doc that exists in the database is guaranteed to find its metadata in the corresponding tables.
6. **UrlCheckerBolt** filters low quality urls. It first checks whether the url has been seen by checking the UrlSeen table. Then it checks whether the domain of the url has been added to the blacklist by calculating the ratio of bad pages over total pages accessed. A web page is defined as a bad page if its connection timeout or the url is unreachable. If the domain has not been seen, the bolt will try to download and parse its robot txt file and store such info in the DomainSeenWithRobotTxtInfo table. Then a HEAD request is sent to the url to validate its content type, modified date and file size. If all the above validations are passed, the url will be added to the UrlQueue.

(2) Indexer

We use **Hadoop MapReduce** to implement the Indexer and run the program on an **EMR cluster**. Initially, we proposed two ways to implement the Indexer:

1. Using two consecutive mapreduce jobs: the first job is to parse and clean/stem the page content and output the page ID with corresponding cleaned term into S3; the second job is to get the cleaned term from the first job in S3, then calculate the TF & IDF score, and send to the DynamoDB.

2. Using a single mapreduce job to parse the content, filter the terms and send to the database directly.

After testing, we noticed that as the number of crawled pages increased, the difference in speed between the two implementations became smaller, so we ended up using the second implementation, which does not need to write output into S3.

The input directory (files) for the Indexer is stored in S3 by crawler, each S3 object contains multiple pages. In the drive class, we use an user-defined separator to split the input files from S3, which ensures that one mapper only receives a single page content with its unique ID. Additionally, we store the total number of pages crawled by crawler and store it in the mapreduce configuration.

In the Mapper class, we first parse the first line to get the page ID, and the rest are the text from the page. We then iterate over all the terms, remove the ones that are stopwords, stemm each word and keep track of their frequency using a Map. With the map, we are able to calculate the TF score for each term, and collect the **(Term, ID + TF)** pair to the reducer. To reduce the workload of the mapper, and to avoid out-of-memory errors, we only collect the first 30,000 words in one page.

In the Reducer class, we first get the total number of pages from the context configuration. Then we iterate over the value to count the number of pages associated with the term, so that we can calculate the IDF score. In the next step, as we get information about term, page ID, and TF from mapper, we then are able to create a DynamoDB Mapper object for each **(Term[key], ID, TF, IDF, TF*IDF)** tuple. We store all such pairs in a List and batch upload to the DynamoDB table.

In the DynamoDB table, we use Term as partition key, because *Term* may not be unique in the table, we use *PageID* as sort key, the schema is described below:

Term(String)	**PageID(String)**	**TF(Number)**	**IDF(Number)**	**TF*IDF(Number)**
pennsylvania	c10aaf2d0a9a90366cbce5ca225d3031	0.3452	0.564	0.19469

Table 1: InvertedIndexTable in DynamoDB

Additionally, to increase the searching speed for search engine, we create a Global Secondary Index table, which uses ***Term*** as partition key, and ***TF*IDF score*** as sort key.

(3) PageRank

The page rank algorithm was implemented in **Apache Spark** and run on an **EMR cluster**. It reads from a text file *(edges.txt)* stored in S3, which includes pairs of source page id and target page id, each forming an edge in the page network. The text file was converted from the *DocumentEdges(from, to)* table in dynamodb, since using Spark to read from S3 is much more efficient than from dynamodb that has unstable read throughout.

The first step is input conversion. Self-directed edges were removed to prevent pagerank hogs. Then, the algorithm created an adjacency list representation *(source page, list of target page)* of the graph.

Next step is to calculate pagerank. Each source page was populated with an initial rank of 1. Using mapreduce operations, initial ranks are propagated along edges and aggregated with a damping factor of 0.15 and decay factor of 0.85 to prevent page sinks,

$$PageRank^{(i)}(x) = \alpha \sum_{j \in B(x)} \frac{1}{N_j} PageRank^{(i-1)}(j) + \beta$$

And pagerank of previous iteration will be fed into the calculation of current iteration until convergence. Finally, each page with a final score was written to a new dynamodb table called *PageRank(docId, rank)* to serve the search engine.

(4) Search Engine

For the search engine part, we use **SparkJava Framework** for the frontend and use a load balancer class, which controls worker nodes.

To monitor the workers status, we use the "heartbeat" class to track if the current worker is working. When a user inputs a query, the query will be tokenized, cleaned by removing stop words and non-alphanumeric characters, stemmed and conveyed to an available worker. After the worker receives the request, it firstly gets the top 500 tf-idf score records from the *InvertedIndex* table, and their corresponding page ID. Then pagerank scores are collected from the pagerank table by querying each ID. After that we combine these scores and sort the records using the ranking function below:

$$Ranking\ Score\ =\ 0.6\ *\ TFIDF\ +\ 0.4\ *\ PageRank$$

Initially we tried to calculate the tf-idf fraction of the query term by the cosine function, but we found this to be less effective. After trying several methods, we found that directly accumulating the tf-idf values of each term can make the results more accurate. After tweaking different parameters, we use 0.6 and 0.4 to weight the TF-IDF score and pagerank score respectively. Finally we display the top 250 links based on our ranking score.

Furthermore, for extra credit features, we linked the search query to Amazon by using amazon-adsystem.com API in the /InitQuery path and implemented the weather forecast feature of 6 main cities by referring to the forecast7.com API in each page.

IV. Experiments and Evaluation

(1) Crawler

- **Hardware configuration:** EC2 and t3.medium, t2.large
- **DynamoDB capacity:** provisioned with auto-scaling mode
- **Performance:**

  At first, we choose to run the crawler continuously and the speed for each worker is around 10-15 pages per minute which is quite unsatisfying. We later discovered that the majority of time and computation resources are used to validate hyperlinks by the UrlCheckerBolt. This is because a modern url often contains hundreds if not thousands of outgoing links. In our initial attempt, over 600,000,000 links are validated and 10,000,000 html pages are added to the queue but only around 30,000 pages are actually crawled. To accommodate the project requirements and save computation costs we decide to make the crawler run in two phases, namely the Collect phase and ByPass phase

  In the Collect phase, a crawler will work normally, extract and validate all the hyperlinks of a web page. When the UrlQueue length reaches 10,000,000 we stop all the crawlers and restart them to run in the ByPass phase where a crawler only downloads and writes a page to S3 without extracting and validating any outgoing link of a webpage. This improvement causes a surge in crawling speed and a worker can crawl over 2100 pages on average in one minute. Finally we are able to crawl one million pages in 2-3 hours with 16 instances. The real time monitoring page for the two phases shown on the master node can be found on the appendix section. The metrics of these two phases are shown here:

Average Crawled Pages of a Single Worker in Two Phases	
Average Crawled Pages/Minute in ByPass phase	2100
Average Crawled Pages/Minute in Collect phase	14

Table 2: Average Crawled Pages of a Single Worker in Two Phases

(2) Indexer

Hardware configuration: EC2 and m5.xlarge

DynamoDB capacity: provisioned with auto-scaling mode

Time Performance:

1 master and 9 cores on EMR: 19hrs to index around 500,000 pages, on average, the indexer can index 7.3 pages per second with provisioned write capacity 15k per second. The speed is decreasing when more terms are indexed. In the end, we inserted over 500 million rows into the InvertedIndexTable.

(3) PageRank

Number of Iterations: A set of number of iterations from 1 to 10 to test convergence of pagerank. We eventually used 5 iterations at which pagerank barely changed.

Time to Run:

(a) **Hardware configuration:** EC2 and m5.xlarge

(b) **DynamoDB capacity:** provisioned with auto-scaling mode

(c) **Performance:** By varying the number of nodes, we found that using 1 master node and 2 cores takes 1.5 minutes, which is the fastest. Using 3 cores was less efficient than using 2 cores, probably because there is more overhead in the shuffling process with more instances. Thanks to the parallelism of mapreduce used in the Spark Framework, our algorithm gives impressive performance.

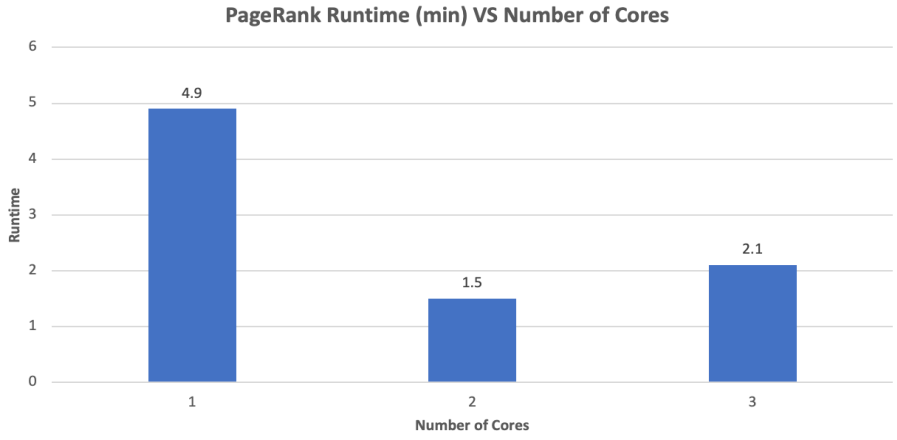


Figure 1: PageRank Runtime VS Number of EC2 nodes

(4) Search Engine

Query Length VS Query Time: Intuitively, using the example shown in the figure below, we can see that longer queries match on a larger set of documents in the corpus, which subsequently increases the query time. At first, our search engine needed at least 8 seconds to generate search results. We found that the bottleneck was iterating the tf-idf queried from dynamodb. We optimized it accordingly by creating a global secondary

index (GSI) using tf-idf as sort key and filtering search results before iteration, which greatly improved the query time to 929ms at the best.

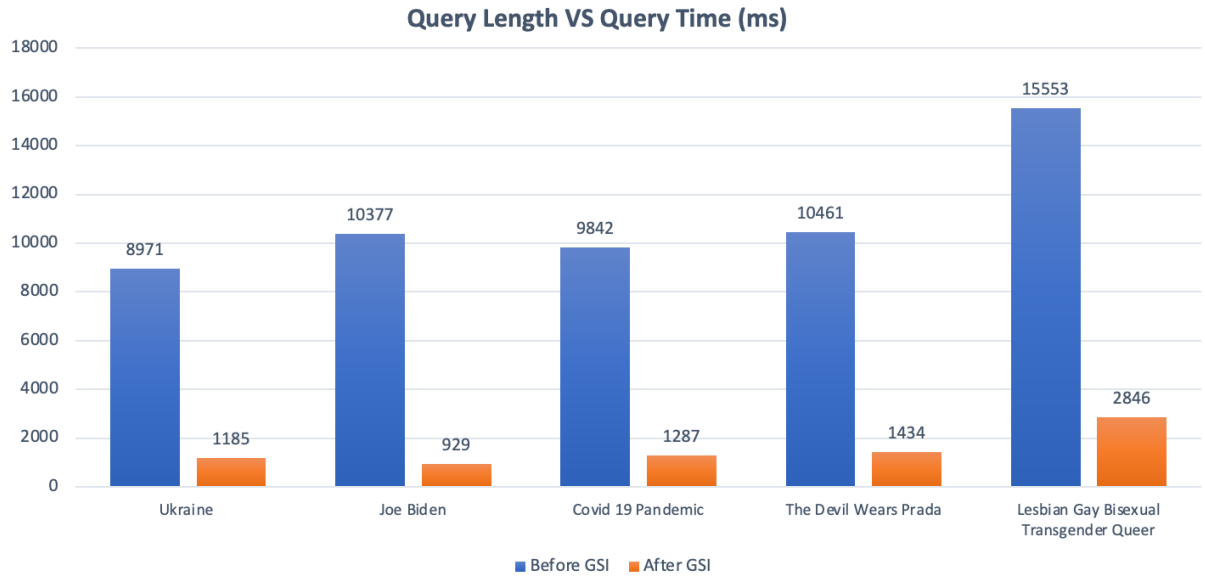


Figure 2: Query Time Performance

V. Conclusion

In this project, we successfully designed and developed a distributed web system that consists of a crawler, indexer and pagerank calculator, which make up the foundation of the search engine that allows users to search queries and get highly relevant results within seconds. By implementing this project, we have gotten a much better understanding of the internet and web system and been able to design an efficient and scalable solution to handle the complex system. We have also become more familiar with the modern web service tools such as S3, EC2, EMR and DynamoDB, as well as honed our teamwork skills.

While we are satisfied with our work, it definitely has much space to improve. For example, to better evaluate the ranking score of the results of a search query, we may further include other metrics such as term position, semantic similarity, and style. Additionally, we can incorporate other features such as spell check into the search engine. To further increase the scalability, we may try different structures other than mapreduce, such as Apache Storm. Finally, we may also crawl images, audios, and videos beside html documents to enrich search results.

VI. Acknowledgement

We would like to thank Professor Zachary Ives and all teaching assistants for all of their guidance and instruction throughout this rewarding semester.

VII. Appendix

(1) Collect Phase Monitoring Page

- ip: 44.195.46.119 url dequeue: 310 url add to queue: 57972 url requeue: 0 total url checked: 189400
- crawled pages: 253 parsed pages: 186 pages save to S3: 180
  averge saved page per minute: 9.274350752510516
- last report received on: 1652141597757 reporter starting report on: 1652140433619
- ip: 44.202.194.78 url dequeue: 135 url add to queue: 19554 url requeue: 0 total url checked: 72595
- crawled pages: 106 parsed pages: 75 pages save to S3: 70
  averge saved page per minute: 10.109227799413665
- last report received on: 1652141597762 reporter starting report on: 1652141182659
- ip: 3.235.151.94 url dequeue: 42 url add to queue: 982 url requeue: 0 total url checked: 4413
- crawled pages: 38 parsed pages: 20 pages save to S3: 10
  averge saved page per minute: 19.01863826550019
- last report received on: 1652141597349 reporter starting report on: 1652141566573
- ip: 44.198.185.120 url dequeue: 627 url add to queue: 106989 url requeue: 0 total url checked: 316992
- crawled pages: 590 parsed pages: 394 pages save to S3: 390
  averge saved page per minute: 11.306259512477956
- last report received on: 1652141597782 reporter starting report on: 1652139528471
- ip: 3.220.167.75 url dequeue: 35 url add to queue: 1231 url requeue: 0 total url checked: 7039
- crawled pages: 31 parsed pages: 27 pages save to S3: 10
  averge saved page per minute: 13.663068725235687
- last report received on: 1652141597988 reporter starting report on: 1652141554207
- ip: 3.235.84.117 url dequeue: 34 url add to queue: 2037 url requeue: 0 total url checked: 12094
- crawled pages: 30 parsed pages: 27 pages save to S3: 10
  averge saved page per minute: 10.584810796507012
- last report received on: 1652141598071 reporter starting report on: 1652141541436
- ip: 3.238.70.89 url dequeue: 34 url add to queue: 580 url requeue: 0 total url checked: 4144
- crawled pages: 30 parsed pages: 23 pages save to S3: 10
  averge saved page per minute: 30.70938683590951
- last report received on: 1652141597122 reporter starting report on: 1652141578583
- ip: 3.216.125.48 url dequeue: 376 url add to queue: 69853 url requeue: 0 total url checked: 231666
- crawled pages: 304 parsed pages: 222 pages save to S3: 200
  averge saved page per minute: 9.066683188178253
- last report received on: 1652141597474 reporter starting report on: 1652140274594
- ip: 44.200.250.27 url dequeue: 205 url add to queue: 25948 url requeue: 0 total url checked: 108133
- crawled pages: 159 parsed pages: 109 pages save to S3: 100
  averge saved page per minute: 11.405152848056753
- last report received on: 1652141597149 reporter starting report on: 1652141072043
- ip: 44.202.233.9 url dequeue: 45 url add to queue: 2013 url requeue: 0 total url checked: 14728
- crawled pages: 41 parsed pages: 34 pages save to S3: 30
  averge saved page per minute: 25.026764734507736
- last report received on: 1652141596478 reporter starting report on: 1652141526198
- ip: 18.215.187.103 url dequeue: 606 url add to queue: 113326 url requeue: 0 total url checked: 360783
- crawled pages: 578 parsed pages: 459 pages save to S3: 450
  averge saved page per minute: 11.471120180376992
- last report received on: 1652141596669 reporter starting report on: 1652139244384
- ip: 44.201.30.102 url dequeue: 253 url add to queue: 50014 url requeue: 0 total url checked: 171624
- crawled pages: 216 parsed pages: 178 pages save to S3: 170
  averge saved page per minute: 10.156752547652097
- last report received on: 1652141595247 reporter starting report on: 1652140593863
- ip: 3.236.190.216 url dequeue: 119 url add to queue: 8234 url requeue: 0 total url checked: 50339
- crawled pages: 103 parsed pages: 71 pages save to S3: 50
  averge saved page per minute: 14.63621683067361
- last report received on: 1652141597775 reporter starting report on: 1652141393150
- ip: 3.236.183.149 url dequeue: 202 url add to queue: 31891 url requeue: 0 total url checked: 115581
- crawled pages: 159 parsed pages: 116 pages save to S3: 110
  averge saved page per minute: 9.773115639650419
- last report received on: 1652141597121 reporter starting report on: 1652140922799
- ip: 3.231.157.146 url dequeue: 429 url add to queue: 80937 url requeue: 0 total url checked: 260176
- crawled pages: 413 parsed pages: 321 pages save to S3: 320
  averge saved page per minute: 11.946494637143891
- last report received on: 1652141597566 reporter starting report on: 1652139990955
- ip: 44.195.32.91 url dequeue: 214 url add to queue: 39064 url requeue: 0 total url checked: 133966
- crawled pages: 180 parsed pages: 135 pages save to S3: 130
  averge saved page per minute: 9.183785203038184
- last report received on: 1652141597436 reporter starting report on: 1652140748798
- total crawled from all workers: 2240

(2) ByPass Phase Monitoring Page

- ip: 44.195.46.119 url dequeue: 17003 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 14337 parsed pages: 0 pages save to S3: 6320
  averge saved page per minute: 2344.8079693789846
- last report received on: 1652164461417 reporter starting report on: 1652164299779
- ip: 44.202.194.78 url dequeue: 16670 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 13923 parsed pages: 0 pages save to S3: 6180
  averge saved page per minute: 2302.3047883965824
- last report received on: 1652164461090 reporter starting report on: 1652164300442
- ip: 3.235.151.94 url dequeue: 16116 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 13526 parsed pages: 0 pages save to S3: 6020
  averge saved page per minute: 2252.080930261558
- last report received on: 1652164461068 reporter starting report on: 1652164301113
- ip: 160.72.126.70 url dequeue: 14908 url add to queue: 0 url requeue: 10574 total url checked: 0
- crawled pages: 4182 parsed pages: 0 pages save to S3: 2820
  averge saved page per minute: 1049.2502697540588
- last report received on: 1652164461433 reporter starting report on: 1652164300240
- ip: 3.220.167.75 url dequeue: 17230 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 14270 parsed pages: 0 pages save to S3: 6490
  averge saved page per minute: 2426.047299823062
- last report received on: 1652164461208 reporter starting report on: 1652164300990
- ip: 3.238.70.89 url dequeue: 17314 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 13894 parsed pages: 0 pages save to S3: 6250
  averge saved page per minute: 2339.8891828483006
- last report received on: 1652164461010 reporter starting report on: 1652164301234
- ip: 3.216.125.48 url dequeue: 17314 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 14533 parsed pages: 0 pages save to S3: 6460
  averge saved page per minute: 2393.360831871959
- last report received on: 1652164461384 reporter starting report on: 1652164299550
- ip: 44.200.250.27 url dequeue: 16247 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 13186 parsed pages: 0 pages save to S3: 5820
  averge saved page per minute: 2166.5219009802704
- last report received on: 1652164460329 reporter starting report on: 1652164300318
- ip: 44.202.233.9 url dequeue: 17348 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 14146 parsed pages: 0 pages save to S3: 6940
  averge saved page per minute: 2589.745503395776
- last report received on: 1652164461408 reporter starting report on: 1652164300710
- ip: 3.236.190.216 url dequeue: 17237 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 13825 parsed pages: 0 pages save to S3: 6840
  averge saved page per minute: 2550.4306648271745
- last report received on: 1652164460724 reporter starting report on: 1652164300584
- ip: 3.236.183.149 url dequeue: 16724 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 14028 parsed pages: 0 pages save to S3: 6310
  averge saved page per minute: 2346.655427183022
- last report received on: 1652164460857 reporter starting report on: 1652164300162
- ip: 3.235.84.117 url dequeue: 17784 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 14233 parsed pages: 0 pages save to S3: 7070
  averge saved page per minute: 2640.4078253670864
- last report received on: 1652164460760 reporter starting report on: 1652164300841
- ip: 18.215.187.103 url dequeue: 10712 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 9506 parsed pages: 0 pages save to S3: 4730
  averge saved page per minute: 1749.3573977846404
- last report received on: 1652164460858 reporter starting report on: 1652164299267
- ip: 44.195.32.91 url dequeue: 16626 url add to queue: 0 url requeue: 0 total url checked: 0
- crawled pages: 13731 parsed pages: 0 pages save to S3: 6160
  averge saved page per minute: 2288.6157466175423
- last report received on: 1652164459978 reporter starting report on: 1652164300003
- total crawled from all workers: 84410