An Evaluation of Solr for Querying Terms of Differing Selectivity in Large Scale Document Collections

Ryan Clancy
University of Waterloo
ryan.clancy@uwaterloo.ca

Zeynep Akkalyoncu Yilmaz
University of Waterloo
zakkalyoncu@uwaterloo.ca

Jaejun Lee
University of Waterloo
j474lee@edu.uwaterloo.ca

ABSTRACT

In various data analytic workloads, searching for relevant documents is a fundamental operation. The most common platform for this is `Solr`, an open source enterprise search engine built on `Lucene`. Though `Solr` achieves low latency in identifying relevant documents by leveraging an inverted index, the impact of random I/O is apparent when fetching documents from disk for terms of low selectivity. Therefore, latency tends to increase as the document frequency of the search term increases when compared to filtering over the entire document collection, which takes advantage of sequential I/O. In this work, we aim to understand the performance characteristics of filtering over terms of varying selectivity within a large scale document collection. We compare search in Solr against filtering in Spark (on documents stored in `HDFS`) and implement a sentence detection workload simulating a typical pre-processing step of an analytic workload. Our comparison shows that for terms that appear in 2.2 million documents or less in our document collection, querying with Solr yields the best performance. However, for documents appearing in more than 2.2 million documents, filtering over the entire collection with Spark is more efficient.

KEYWORDS

Document Search, Performance Evaluation

1 INTRODUCTION

The Web has witnessed significant data proliferation over the last decade as a result of global increase in social media usage and business intelligence operations. While previous research efforts have primarily targeted structured data (such as a relational database), the focus has since shifted toward semi-structured and unstructured data given the nature and flux of data being generated (e.g: service logs) [6]. With increasing amounts of data, it has become unfeasible to preprocess all free-text to fit a certain schema prior to the actual data analytic workloads. Therefore, data scientists now find that they must optimize for non-structured in addition to structured data in their analyses. The problem introduced by this requirement is twofold: how to retrieve and process relevant unstructured data efficiently. This work explores natural operations associated with non-structured data: search and filtering.

Although digital storage has become considerably cheaper to accommodate data proliferation, processing power and data science tools initially struggled to keep up with this expansion. Domain-specific languages, such as SQL, designed to operate on structured data stored in RDBMSs are optimized for strictly structured data, and are ineffective at dealing with such large volumes of semi-structured or unstructured data.

For this reason, a number of frameworks and tools developed with this particular use case in mind have emerged over the last few years. MapReduce [3] was one of the earlier frameworks to address the problem of big data, and later became one of the most popular tools for processing large scale data collections. Its popularity may be attributed to its flexibility and extensibility, and ability to deal with ill-formed and/or unstructured data stored in HDFS in a distributed manner. Spark was then released in response to the limitations of MapReduce (e.g: strict linear dataflow structure, mandatory checkpointing to HDFS) [12]. It has enjoyed widespread adoption in the big data community by providing further flexibility with data processing.

Unfortunately, the brute force approach taken by data processing frameworks, such as Spark, for filtering terms in large scale document collections is not ideal for low selectivity terms (i.e. high frequency terms). Various search tools such as Apache Solr[1] and Elasticsearch[2] that leverage inverted indexes to search large document collections eliminate the need for full collection scans. The combination of these search tools and aforementioned frameworks allows the execution of powerful data analytic workloads.

Despite their undeniable convenience, the performance characteristics of these search tools on large scale document collections need to be considered carefully for reliable performance. In particular, there may exist many words that appear only in a few documents whereas a few, such as stopwords, that appear in many, following a Zipfian distribution. This discrepancy may lead to variable performance that is heavily dependent on the specific search term. The effect of search term selectivity poses an interesting problem in that it would impact high-level design decisions in data processing implementations.

In this project, we evaluate the effectiveness of Solr as a search tool for terms with different selectivity. We also compare its performance against filtering operations implemented with Spark to understand how the distribution of words in the dataset affects the data analytic workload.

Our findings show that Solr is efficient at retrieving relevant documents when the search term has low document frequency. However, the resource usage of Solr increases as search term frequency increases; for terms that appear in more than about 2.2 million documents, it is not as efficient as filtering with Spark.

The source code and results can be found in this repo[3].

The rest of this paper is organized as follows: Section 2 presents the background information and summarizes relates work. Section 3 details the experimental setup and Section 4 presents the experimental results. Section 5 discusses future work and Section 6 concludes the paper with an overview of our results.

[1]http://lucene.apache.org/solr
[2]https://www.elastic.co
[3]https://github.com/ljj7975/CS848-project

2 BACKGROUND AND RELATED WORK

2.1 Solr

Apache Solr[4] is an open-source search platform built on top of Apache Lucene[5]. Lucene tokenizes arbitrary text, builds an inverted index over the tokens (mapping tokens to documents), and supports efficiently querying over the documents. Lucene is the de-facto indexing and search tool used in industry, but provides a low level API that may not be necessary to work with.

Solr is a search platform built around Lucene that exposes search as a service. Solr supports all the rich indexing and search functionality of Lucene and extends it by providing a management interface, supporting caching, exposing a REST API (XML and JSON) and providing `SolrJ`, a simplified client API in Java.

2.2 SolrCloud

SolrCloud supports sharding and replication of document collections by breaking down the collection into multiple shards and storing them in different nodes. SolrCloud supports search over both the whole collection and the subset of the collection in a single node. To minimize performance degradation caused by network partitioning, it also supports returning partial results.

SolrCloud has a different implementation for the client API that is aware of the data distribution across the system. When a `CloudSolrClient` is pointed to ZooKeeper, multiple queries are sent to a random replica of each shard in `distrib=false` mode (forces search within the specified shard only) and aggregated on the client side. On the other hand, when request is pointed to a single node (instead of ZooKeeper), Solr is responsible for forwarding the query to each `CloudSolrClient` and aggregating the results. The network usage increases with respect to the size of search results and affects performance of downstream operations.

2.3 Spark

Spark is general-purpose distributed data processing engine that achieves high performance by utilizing a data abstraction called resilient distributed dataset (RDD) [11]. In conjunction with a distributed file system such as HDFS, it tries to maximizes data locality by co-locating workers with input data [14]. Since the advent of Spark, its architecture has been studied and improved to support various types of workloads which has lead to the development of SparkSQL, MLib and SpartStreaming [1, 8, 13].

2.4 Word Frequency Distribution

Zipf's law is often used to describe distribution of words in a corpus. Zipf's law states that the frequency of a word is inversely proportional to its rank when it is ordered by frequency [15]. After the advent of this empirical law, numerous study has followed describing distribution of words on other languages [4, 10]. As a result, Zipfian distribution is often studied to understand the phenomena related to the frequency of a word. Especially in the field of computer science, this lead to numbers of heuristic which improves performance of a system for processing words with high selectivity [7, 9].

[4]http://lucene.apache.org/solr/
[5]http://lucene.apache.org/

Though Zipfian distribution introduces a nice heuristic that can be used to increase the performance of a system, it was found that the heuristic does not guarantee same benefit over words with low selectivity. In the work of Lin et al. [7], a Lucene full-text index is used to identify LZO blocks that contained search terms and only those blocks were decompressed and read by Hadoop. It is shown that integration of this step could yield a significant improvement in overall processing time when a term has low frequency. However, the correlation between performance and selectivity leads to extra overhead for terms that appear in almost every document because the amount of data retrieved from Lucene is similar to the amount of data processed by Hadoop.

Given Lucene is the main component of Solr that is responsible for document indexing and search, our assumption is that the aforementioned phenomena will also be observed in a system with Solr. Therefore, our goal in this paper is to understand the behaviour of Solr on large scale document collections when search term has different selectivity.

3 EXPERIMENTAL DESIGN

In our experiment, the system interact with a document collection with a size of 425.8 GB, searching for terms that vary in their selectivity ranging from 0.20% to 61.77% of the documents. Our implementation includes sentence detection to mimic a typical preprocessing step in a data processing pipeline on the search results.

Typical OLAP workloads often utilize distributed data processing engines to improve performance with increased parallelism. Therefore, we explored the integration of Solr with Spark in two of our implementations. Overall, our experimental setup includes:

(1) `ThreadedSolr`
Query Solr for documents of interest with client side sentence detection + multi-threading
(2) `SparkSolr`
Query Solr for documents of interest with parallel sentence detection via Spark cluster
(3) `SparkHDFS`
Process raw documents in HDFS utilizing Spark operations for filtering documents of interest and sentence detection

3.1 Document Collection

We utilize the `GOV2` document collection which is used in the TREC Terabyte Track [2]. The collection consists of a crawl of `.gov` webpages (including the text of PDF and Word documents) from early 2004 and contains about 25 million documents. The size of the data collection is 425.8GB (84GB compressed). The documents are stored uncompressed on `HDFS` such that Spark can read the raw `HTML` files and Solr stores the documents in the Lucene file format leveraging an inverted index. It is found that decompression and uploading document to HDFS is very expensive operation as it took 40 hours while indexing into Solr took only 4 hours.

Figure 1 shows the document frequency distribution of GOV2. For simplicity, the graph only includes entries with document frequency greater than 10 and number of words greater than 150. Even though there exists some outliers with document frequency of 210, the distribution follows Zipfian distribution; there are a lot of words

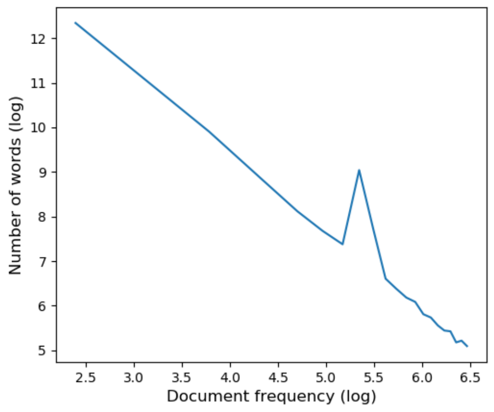

Figure 1: Document frequency distribution of GOV2.

with low frequency and only few words with high frequency. There are about 30 million unique words that appears only once. On the other hand, stop words tends to have high frequency. Stop words "the", "to", and "of" have highest frequency of 17, 18, and 19 million, respectively. Such stop words are removed during the preprocessing of raw documents and so are not included in the inverted index in Solr. Non-stop words that have the highest frequency are "public" and "inform" with 15 and 12 million appearance respectively, corresponding to about 60% and 50% of GOV2 documents.

For our experiments, we manually selected terms that appear in roughly 50,000 (0.20%) to 15,000,000 (61.77%) of the documents at varying intervals. Table 1 highlights the terms that were selected.

	Doc Freq (#)	Doc Freq (%)
interpol	50,551	0.20
belt	99,287	0.39
kind	506,423	2.01
idea	1,003,549	3.98
current	4,897,864	19.43
other	9,988,438	39.63
public	15,568,449	61.77

Table 1: Distribution of selected query terms.

3.2 Architecture

Hardware. In order to run our experiments, we utilize 6 machines from the Tembo[6] cluster. Each machine has 2x Intel E5-2620v2 CPUs, 32GB of RAM, 2x 1TB HDDs and is connected to other machines in the cluster with both 1 and 10 GbE networking. 5 of the machines are used as part of a Kubernetes cluster to run our services (Solr, Spark, HDFS, ZooKeeper) and 1 machine is used as the driver for experiments.

[6]https://cs.uwaterloo.ca/twiki/view/UW/Tembo

Kubernetes. Kubernetes[7] is a container orchestration engine created by Google. It serves to allow easy management of containers across a cluster of machines that have the Kubernetes daemon installed. Kubernetes supports features such as container scaling, service discovery, load balancing and self-healing (on container failure).

A Kubernetes cluster consists of multiple nodes where one (or multiple, in a highly available configuration) is designated as the master. Every node in the cluster, including the master, can run containers. For our experiments, we setup a Kubernetes cluster from scratch on Tembo using 5 nodes. We setup a local provisioning daemon on each machine for persistent storage which allows containers to keep its data after a restart (due to update or failure).

In our setup, we have the following containers deployed:

- **5x** Solr containers (each shard with $\frac{1}{5}$ of the documents)
- **3x** ZooKeeper containers for Solr
- **3x** ZooKeeper containers for HDFS
- **1x** HDFS client container
- **2x** HDFS NameNode containers
- **3x** HDFS JournalNode containers
- **5x** HDFS DataNode containers

When used with Kubernetes as the master, Spark supports the dynamic deployments of workers. Therefore when a Spark experiment is running, 5 instances of Spark workers are managed by Kubernetes as well. Throughout our experiments, a Spark job is submitted in `client` mode, running driver on the client machine to be consistent with `ThreadedSolr`.

Apache ZooKeeper [5] is a distributed co-ordination service and is used to coordinate the Solr shards and HDFS nodes. We are running 3 ZooKeeper instance in Kubernetes with 1 leaders, 2 followers, and a quorum size of 2 which would allow failure of a single node.

3.3 Implementation

ThreadedSolr. SolrCloud alone often gets used to support the search functionality of an application. Therefore, we investigate querying the Solr cluster from a single machine and aggregating the results in the driver. We implement a program that utilizes the `CloudSolrClient` client implementation from the `SolrJ` library. We use the cursor functionality of the library to page through the results 1,000 at a time, passing each batch off to a `ThreadPool` consisting of 24 thread (the total number on the driver machine) to perform sentence detection.

SparkSolr. In the case of high volumes of search results, collecting the results and processing in a single machine is a bottleneck. The solution is to scale out by distributing the workload to multiple machines. This approach decreases resource usage of each worker and increases throughput of overall system. In this experiment, 5 Spark workers retrieve and process subsets of search results, reducing amount of data a single machine has to process.

We use the `spark-solr`[8] library to fetch documents as a RDD. `spark-solr` exploits data locality by being aware of Solr shards

[7]https://kubernetes.io/
[8]https://github.com/lucidworks/spark-solr

	Doc Freq	ThreadedSolr		SparkSolr		SparkHDFS
		Time (s)	Query (%)	Time (s)	Query (%)	Time (s)
interpol	50,551	191	71	152	69	2,256
belt	99,287	312	77	182	93	2,286
kind	506,423	1,000	91	625	84	2,296
idea	1,003,549	1,625	92	1,720	87	2,402
current	4,897,864	3,950	98	9,797	86	2,344
other	9,988,438	6,342	98	19,737	84	2,494
public	15,568,449	9,462	97	28,968	93	2,513

Table 2: Total execution time (query + sentence detection) and percent of time spent querying for each term.

that are located on the same node to eliminate unnecessary network traffic. Therefore, each request from Spark workers contacts its local Solr shard and retrieves results in batches of 1,000 forming a `partition`. Sentence detection is then performed over each `partition` in parallel.

SparkHDFS. The `SparkHDFS` implementation reads raw documents from the collection stored in HDFS. Each file in HDFS contains several documents enclosed within `<DOC></DOC>` tags and is parsed using Databrick's `spark-xml`[9]. Much like the `SparkSolr` implementation, Spark reads files from HDFS that are co-located on the same nodes in order to eliminate unnecessary network traffic. Sentence detection is then performed over each `partition` of the input data, just as in the `SparkSolr` implementation.

In order to ensure that returned results are consistent with Solr, this implementation uses the Lucene `EnglishAnalyzer` to process both the query and document before doing filtering (which the other implementations perform implicitly in Solr). For example a passage such as "The DOG's parents are barking" is transformed to "dog parent bark"; stop words are removed, text is converted to lowercase, possessives are removed, plurals are removed, and the words are stemmed.

4 EXPERIMENTAL RESULTS

Overall, the results show a positive correlation between document frequency and performance as previously observed in Lin et al. [7]. In addition to execution time, memory and CPU usages also increase as document frequency increases.

4.1 Latency

Figure 2 summarizes execution time in seconds. Given that sentence detection is a relatively simple operation, we observe that most of the execution time is spent to communicate with Solr when Solr is used for search. Starting from 100s of seconds with low frequent terms, `ThreadedSolr` requires 2.5 hours to process terms that appears in 15 million documents. Similarly, `SparkSolr` starts from 100s of seconds with low frequent terms. Processing documents in parallel, `SparkSolr` actually outperforms `ThreadedSolr` when searching for terms with document frequency up to around 2.2 million. However, starting from `current` which appears in 5 million documents, it is higher than the other two experiments and reaches

[9]https://github.com/databricks/spark-xml

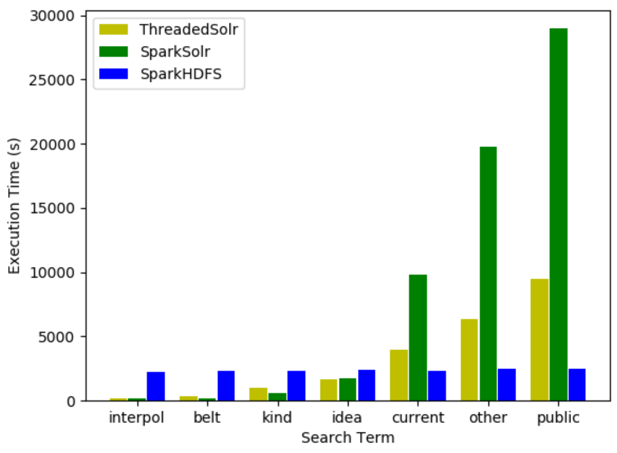

Figure 2: Total execution time (query + sentence detection).

latency of 8.3 hours with the highest frequency term (see Figure 2). On the other hand, minimal latency increase is observed over different selectivity with `SparkHDFS`. Due to the amount of data that `SparkHDFS` need to read in, it is about 20x slower than other two for terms with low frequency. However, latency for term that appears 15 million documents is 40 minutes, introducing only 250 seconds increase compared to the term with the lowest frequency (time which is likely spent performing additional sentence detection).

4.2 CPU and Memory

Figure 3 and Figure 4 show resource usage on the driver and the cluster. Given that the driver is running on a single machine, CPU usage is in percentage out of 24 cores and memory usage out of 32 GB.

To understand resource usage on the cluster, a script is created to collect CPU and memory usage every minute throughout the execution of each experiment. We calculate average resource usage for each experiment and aggregated over the cluster to get overall usage. Since there are 5 machines forming the cluster, we then divide overall usage by 120 cores and 160 GB respectively to calculate percentage.

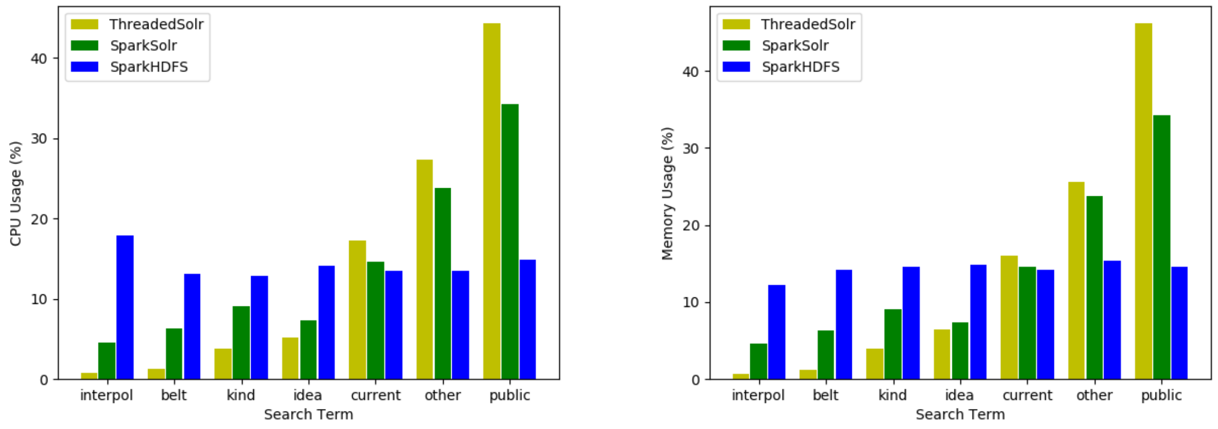

Figure 3: Average CPU and memory usage on client.

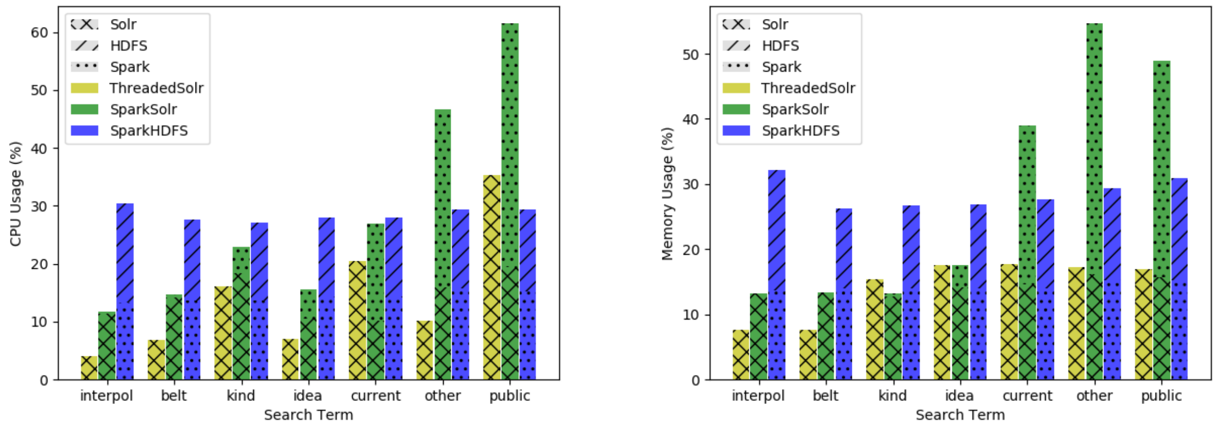

Figure 4: Average CPU and memory usage on cluster.

Client Resource Usage. When the search term has low frequency, both CPU and memory usage of ThreadedSolr is smaller than the other two experiments because it does not suffer from the overhead introduced by Spark. However, as the amount of data to process increases, client side sentence detection requires more resources. As a result, ThreadedSolr uses the most resources starting from terms with document frequency of 5 million. Similar to ThreadedSolr, SolrSpark resource usage increases as term frequency increases. Compared to SparkHDFS, the driver for SolrSpark uses much fewer resources when the amount of documents to process is small but SparkHDFS the driver eventually consumes less resource because resource usages of SparkHDFS driver does not change much even though document frequency of search terms changes.

Cluster Resource Usage. Figure 4 shows the resource usage of each service in the cluster. Given Solr is the only component playing a role in ThreadedSolr, we observe increasing CPU and memory usage following the increase of document frequency. Comparing its usage against Solr of SparkSolr, it uses more CPU when the document frequency is high.

Similar to our previous observation, the resource usage of SparkSolr workers start small but increases as document frequency increases. On the other hand, the resource usage of the SparkHDFS workers are consistent throughout the experiments. This pattern is also observed with HDFS resource usage and this must be true since the amount of being retrieved from HDFS is same for every experiment.

4.3 Random vs. Sequential I/O

Given that spark-solr and HDFS both try to maximize data locality by contacting local nodes, we suspect that the difference in performance of SparkSolr and SparkHDFS is coming from other factors than network usage. Our investigation shows that difference in disk I/O pattern introduces this behavior.

Solr maintains an inverted index for each collection it stores, mapping terms to a (sorted) list of document IDs for each document the term is present in. When executing a query, random seeks are required to fetch each document from disk as the documents aren't physically ordered by the search term. When using Spark with HDFS, Spark reads input files sequentially. Obviously, this will be much faster when reading the same amount of data.

Inspecting the read I/O during the experimental runs, it is found that `SparkSolr` have an average read speed between 15-20 MB/s whereas `SparkHDFS` has an average read speed of between 65-80 MB/s by taking advantage of sequential I/O. The difference in I/O speed of random vs. sequential access is what we believe to be the main difference between the results of `SparkSolr` and `SparkHDFS`.

5 FUTURE WORK AND LESSONS LEARNED

One of the goals of our project was to evaluate the use of Kubernetes to orchestrate containers for use in future research projects. Although Kubernetes is very powerful, it's not a one size fits all solution. Kubernetes is so configurable and extensible that it's often hard to discover the correct way to do things; the documentation was fragmented and was hard to navigate. We encountered quite a few problems with managing persistent storage for stateful applications (mainly Solr) as well as with ensuring containers to be correctly located on different nodes. More specifically, controlling which nodes to instantiate Spark workers on is not supported with the latest version of Spark and multiple Spark workers may be scheduled on same node, affecting data locality of Solr and HDFS. Our solution was to request high resources for each worker forcing Kubernetes to locate each workers on different machines. Kubernetes was a great learning experience, but for smaller setups where you aren't trying to manage a data center or huge application deployment, we believe it is an overkill and falling back to something simpler like Docker Swarm (or even normal Docker) may be a better path to take.

We found that often times during the experiments we were bottle-necked by various hardware resources such as disk I/O or not having enough memory to fit the entire Solr index into memory. For example, we initially tried using the `cw09b` collection, but it was far too large for the machines we were using. In the future, we would like to re-run the experiments on a cluster with additional (and more powerful) machines so that we can evaluate performance of the system when we aren't as resource constrained.

Given the time constraints of the project, it was difficult to do multiple runs of the experiments for averaging runs and calculating the standard deviation. Each complete run of the experiments took several days. Given more time, we would like to perform additional runs of the experiments for averaging and having more confidence in the results.

One final note is that we'd like to dig deeper into the CPU usage in 3 for the driver program. For `SparkHDFS`, the CPU usage across different terms is constant. This is what we'd expect given that the driver does very little work. For `SparkSolr` on the other hand, the CPU usage grows with the number of terms that are returned and consumes more resources than `SparkHDFS` after the document frequency of search term reaches certain point. Our hypothesis was that it will never consume more resources than `SparkHDFS` because the number of partition involved is much smaller in `SparkSolr`. We believe that digging deeper into the implementations of the Solr libraries (SolrJ and spark-solr) can be beneficial for understanding this behavior.

6 CONCLUSIONS

In this work, we conducted a number of experiments to study the performance of search and filtering over terms of varying selectivity in the `GOV2` collection by comparing search with Solr using an inverted index, and filtering on documents stored in HDFS with Spark. For this purpose, we set up a Kubernetes cluster from scratch (including installing and configuring Solr, ZooKeeper, and HDFS) and implemented three Scala programs that perform search and filtering designed to work seamlessly with our architecture.

According to our experiments, Solr is not only faster but also more efficient, in terms of CPU and memory usage, when searching for terms that occur in relatively few documents (up to roughly 2.2 million documents for our particular document collection). However, filtering with Spark over documents stored in HDFS is much efficient than utilizing Solr when terms have high frequency.

REFERENCES

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and others. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1383–1394.

[2] Charles LA Clarke, Nick Craswell, and Ian Soboroff. 2004. Overview of the TREC 2004 Terabyte Track.. In *TREC*, Vol. 4. 74.

[3] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150.

[4] Abduelbaset Goweder and Anne De Roeck. 2001. Assessment of a Significant Arabic corpus. In *Arabic NLP Workshop at ACL/EACL*.

[5] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX-ATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[6] Jimmy Lin. 2018. Data Intensive Distributed Computing. (September 2018).

[7] Jimmy Lin, Dmitriy Ryaboy, and Kevin Weil. 2011. Full-text Indexing for Optimizing Selection Operations in Large-scale Data Analytics. In *Proceedings of the Second International Workshop on MapReduce and Its Applications (MapReduce '11)*. ACM, New York, NY, USA, 59–66. DOI:http://dx.doi.org/10.1145/1996092.1996105

[8] M Seddon. 2015. Natural Language Processing with Apache Spark ML and Amazon Reviews. (2015).

[9] Catriona Tullo and James Hurford. 2003. Modelling Zipfian Distributions in Language. In *Proceedings of language evolution and computation workshop/course at ESSLLI*. 62–75.

[10] Hang Xiao. 2008. On the Applicability of Zipf's Law in Chinese Word Frequency Distribution. *Journal of Chinese Language and Computing* 18, 1 (2008), 33–46.

[11] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[12] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 423–438.

[14] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, and others. 2016. Apache Spark: a Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65.

[15] George Kingsley Zipf. 2016. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology.* Ravenio Books.