A Large-Scale Distributed Text Processing Framework with Stanford CoreNLP[*]

Yuan Chen[†]
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
constant.chen@uwaterloo.ca

Kaisong Huang
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
kaisong.huang@uwaterloo.ca

Ji Xin
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
ji.xin@uwaterloo.ca

ABSTRACT

As the Internet becomes more and more integrated into all aspects of people's lives, the data generated by people using the Internet is increasing. Although most text processing related research requires a large amount of data support, at the same time, an oversized corpus poses a huge challenge to the processing power of natural language processing toolkits. In this research, We implemented two distributed versions of Stanford CoreNLP, using Hadoop MapReduce and Spark. We conducted five controlled experiments to explore the performance and efficiency of our distributed text processing framework against a real corpus from the Washington Post dataset. We concluded that when manipulating a large dataset, the Spark version of our distributed system outperformed the MapReduce one.

KEYWORDS

large scale text processing, natural language processing, Spark, MapReduce, Distributed System

ACM Reference Format:
Yuan Chen, Kaisong Huang, and Ji Xin. 2018. A Large-Scale Distributed Text Processing Framework with Stanford CoreNLP. In *Proceedings of ACM Conference (Conference'18)*

1 INTRODUCTION

As is known to all, the recent growth of the World Wide Web at increasing rate and the number of online resources represent a massive source of knowledge for various research and business interests [24]. However, single-machine, non-distributed architectures are proving to be inefficient for tasks like intensive text processing and analysis. To solve the problem, this paper proposes a Spark-version implementation of large-scale distributed text processing with Stanford CoreNLP (we will refer it as CoreNLP in the following sections). On the one hand, using distributed computing approaches to improve the performance and workload on available hardware is a common topic in computer science. On the other hand, current natural language processing (NLP) systems are growing in complexity, and computational power needs have been significantly increased, requiring solutions such as distributed frameworks and parallel computing programming paradigms [28]. Therefore, it is worth trying the integration of a distributed computing framework and an NLP toolkit to verify whether large-scale distributed text processing on a big data collection (e.g., the Washington Post dataset) is feasible. Hadoop is a popular open source distributed computing framework, providing a distributed file system named HDFS and distributed computing framework which is called MapReduce [2]. Our test collection (i.e., the Washington Post dataset) is stored on HDFS. Moreover, as an open source in-memory distributed cluster system developed in the UC Berkeley AMP Lab, Spark is designed to speed up data analysis operation. Spark is suitable for the treatment of iterative algorithm (such as machine learning, graph mining algorithm) and interactive data mining [3]. In this research, we implement both Hadoop MapReduce (we will refer it as MapReduce in the following sections) with CoreNLP and Spark with CoreNLP. CoreNLP toolkit is an extensible pipeline that provides core natural language analysis. This toolkit is quite widely used, both in the research NLP community and also among commercial and government users of open source NLP technology [25].

In this study, we conducted five controlled experiments to explore the performance and efficiency of our Spark-version distributed text processing framework. While conducting research, we also considered different running settings and contexts. More specifically, we were trying to find out how to optimize the configuration to achieve better performance in our distributed framework with Spark under different contexts and have a clear notion of how and why our Spark framework is able to perform better than other existing frameworks on a given task. The contributions of this paper are stated as follow:

- To understand how the number of partitions could exert influence on the performance of our Spark implementation, we designed an experiment having the framework run with different numbers of partitions.
- We assumed that a larger size of the dataset could compromise the performance. From our experimental analysis, we discussed the relation between the dataset size and the performance, and also the capability of our framework to process intensive data.

[*] All codes are available at: https://github.com/Constannnnnt/Distributed-CoreNLP
[†] Author names are listed in alphabetical order of the last name.

- We considered different contexts of the possible executions of our framework to capture the different performance for different properties in CoreNLP.
- To verify if the integration of Spark and CoreNLP could outperform the other two frameworks (i.e., integration of MapReduce and CoreNLP, CoreNLP running on a single machine with multiprocessing), we designed two experiments comparing Spark implementation with MapReduce implementation and multiprocessing implementation.

The rest of the paper is organized as follows. First, we discuss the related work in the next section. We then describe the methodology of our study. After that, we explain our experimental details in the *experiment* section, where the questions we intend to address and the experimental results along with the answers to the questions are also included. In the final section, we conclude our paper, state the limitations and express future work.

2 RELATED WORK

In recent years, many researchers have been focusing on how to take advantage of emergent Big Data and NLP solutions to better serve various domains, aiming to provide a general definition and framework for Big Data ecosystem [e.g. 1, 4, 18, 28, 37]. Although the enthusiasm of researchers in this field is very high and they have achieved fruitful results, there is still a lack of studies on the integration of Spark and CoreNLP, in particular, the performance and efficiency of this combination.

The first attempts we find which employed parallel computing frameworks for NLP tasks were in the middle 90s. Chung and Moldovan proposed a parallel memory-based parser called PARALLEL [6], and PARALLEL was implemented on a parallel computer (i.e. the Semantic Network Array Processor). Later, Van Lohuizen proposed a parallel parsing method relying on a work stealing multi-thread strategy in a shared memory multiprocessor environment [38]. Hamon et al. implemented Ogmios, a platform for annotation enrichment of specialized domain documents within a distributed corpus [17]. The system provides NLP functionalities such as word and sentence segmentation, named entity tagging, POS-tagging and syntactic parsing. Jindal et al. developed a parallel NLP system based on LBJ [19], a platform for developing natural language applications, and Charm++ as a parallel programming paradigm [20]. Despite great contributions have been made by early studies, there is always a mismatch between these traditional data processing model and a vast amount of data generated by the rapid development of the Internet. To deal with such challenges, a variety of cluster computing frameworks have been proposed to support large-scale data-intensive applications on commodity machines [15]. Exner and Hugues recently presented a multi-language NLP processing framework (known as KOSHIK) for large scale-processing and querying of unstructured natural language documents distributed upon a Hadoop-based cluster [12]. It supports several types of algorithms, such as text tokenization, dependency parsers, coreference solver. The advantage of using the Hadoop distributed architecture and its programming model (known as MapReduce), is the capability to efficiently and easily scale by adding inexpensive commodity hardware to the cluster [28]. To the best of our knowledge, there has been a lot of research on the integration of MapReduce and NLP toolkits whereas only a few research has considered using Spark as their distributed computing framework.

Hadoop and Spark are top-level Apache projects designed to enable massive parallelization of data processing. Hadoop is a widely used processing framework with proven potential for very high throughput [37]. Spark is a distributed processing framework that makes use of in-memory primitives [40]. Spark introduces programming transformations on Resilient Distributed Datasets (RDD) [39], which are read-only collections of objects distributed over a cluster of machines providing fault tolerance by rebuilding lost data by using lineage information (without requiring data replication). RDD allows storing data on the memory, as well as to define the persistence strategy. Gopalani and Arora [14] have compared Spark and Hadoop performances on clustering K-means algorithms, showing that Spark outperforms Hadoop on different cluster configurations.

As mentioned before, current NLP systems are growing in complexity. The four most well-known and widely used toolkits by the NLP community are NLTK, Apache OpenNLP, CoreNLP and Pattern [31]. Neumer compared the performance of these four NLP toolkits from tokenizer's perspective [29]. The results of the quantitative measurements state that CoreNLP is not only the fastest open-source tokenizer, but it also uses the least amount of memory and an outstanding amount of 32 threads. Written in Java and having wrappers for many different programming languages it is a great choice for someone, who is trying to scale up his application. Additionally, CoreNLP is a relatively simple and straightforward pipeline to set up and run compared to other similar frameworks [25]. The library of CoreNLP was originally written in Java and runs on a local server [29]. We find that lots of research have been done on non-distributed versions of NLP toolkits, it is still very necessary to study a distributed version. The Apache OpenNLP and CoreNLP are quite similar to each other in terms of functionality and ease of use. However, the licensing terms are different. CoreNLP seems to have a slight edge in active maintenance and evolution [16]. Even so, prior studies mainly adopt OpenNLP for their NLP module, meaning that CoreNLP still needs more attention.

These days, processing a large amount of textual data has become a major challenge in the NLP research area. Due to the fact that the majority of digital information is present in the form of unstructured data like web pages or news articles, NLP tasks such as cross-document coreference resolution, event detection or calculating textual similarities often require processing millions of documents in a timely manner [5, 11, 27, 30, 34, 35]. CoreNLP provides highly accurate results, while it suffers from a slow processing time; adding additional annotators also makes it considerably slower. The naive "download and run" way to use CoreNLP is unrealistic when the input files become larger. Therefore, there is a tremendous need to run CoreNLP on a distributed framework, such as Spark, to improve the processing speed.

3 METHODOLOGY

In this section, we first briefly introduce CoreNLP, specifically its annotation pipeline and what functions have been implemented in our infrastructure. Then, we describe our distributed processing framework in detail and address some optimization contributing to the efficiency of this distributed framework.

3.1 Stanford CoreNLP

In this project, we have decided to use the Stanford CoreNLP toolkit. It is a JVM-based annotation pipeline framework, which provides most of the common core NLP tools. Before the toolkit was designed and proposed [26] to achieve a extensible pipeline that supports multiple NLP techniques, the Stanford Natural Language Processing Group had proposed and developed several techniques to enable machines to interpret text like humans. For Instance, the CoreNLP parser is a statistical, unlexicalized and NLP parser initially trained on the Wall Street Journal [10, 21, 22], which can identify the grammatical structure of sentences. In addition to the parser, the toolkit also includes the following: a named entity recognizer (NER), also known as CRFClassifier [13], which uses a liner chain sequence model to identify and label sequences of words in the text; a co-reference resolution architecture, consisting of three different co-reference systems to finds the mentions of the same entity in a text, including the deterministic co-reference system [23, 32, 33], the statistical co-reference system [7] and the neural coreference system [8, 9]; and sentiment analysis tools [36], which employ a recursive neural tensor network to build a tree-based representation of the whole sentence and then computes the sentiment in longer phrases. With these tools, the toolkit has empowered machines with abilities not only to chop long passages and paragraphs into several bags of words, but also analyze linguistic properties and semantic meanings and relations among words, which turns out to be a well-developed package for us to explore and analyze many text datasets, such as the Washington Post dataset.

The entire CoreNLP toolkit, including the code jar and model jar, takes around 500Mb. In order to reduce the size of the architecture, instead of downloading the entrie toolkit, we write related dependencies in *pom.xml.* The toolkit is written in Java, and the current version we use is Stanford CoreNLP 3.9.2, which requires Java 1.8+ and is licensed under the GNU General Public License. It supports a bunch of properties associated with the linguistic text analysis, ranging from giving basic form of words to co-reference resolution between words in a sentence and a paragraph. To achieve this, CoreNLP uses annotations, which structure and map the input data, and annotators, which server as workers with functions over annotations. The annotations supported by CoreNLP are listed in Table 1 in Appendix and the execution flow among annotation objects, input and output is listed in Figure 1.

3.2 Distributed Architecture

Functionalities of CoreNLP can be divided into two categories: sentence level and document level. An example of the former is *ner* (Named Entity Recognition) and an example of the latter is *ssplit* (sentence splitting). When we use CoreNLP to annotate our corpus, sentences[1] inside the corpus are independent, i.e., the annotation output of a sentence only depends on itself, not other sentences in

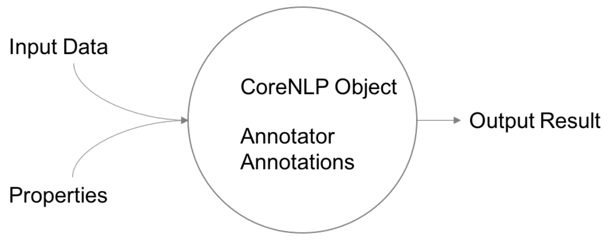


Figure 1: Execution flow between the data and annotation objects

the corpus. It is this independence that makes distributed annotation possible.

The core idea of the large-scale architecture is to distribute sentences over multiple processing threads in a machine or multiple machines in a cluster. Each individual thread or machine is a mapper. The most important implementation detail is that one and only one annotator object should be constructed for a mapper, and then be reused for each input sentence. On the one hand, such an annotator object is not serializable, and therefore cannot be constructed in advance and then broadcast to every mapper. On the other hand, if an annotator object is constructed for every input sentence, the processing speed will be considerably slow.

When the list of required functionalities is specified by the user, a corresponding configuration for CoreNLP is sent to all mappers, and each mapper constructs a CoreNLP annotator object accordingly in its own memory before any sentence arrives. When a sentence arrives, it is annotated by the annotator which yields one output value for each required functionality. The key for such output value is a tuple of both the sentence value and the functionality name. These key-value pairs are partitioned according to the functionality name within the key, and the number of reducers is exactly the number of required functionalities. After shuffling, these key-value pairs arrive at reducers, and they are sorted by sentences indices. The reducers finally write output values to the output path. Each functionality corresponds to a output file, and output values are in the order of input sentences within each file.

We provide two implementations of the above idea, using either MapReduce or Spark. Both implementations share the same architecture, but there is a minor difference due to their different nature. In the MapReduce implementation, the CoreNLP annotator object is constructed in the setup method of mapper. The list of required functionalities, which is needed to construct the annotator object, is passed to each mapper through the MapReduce context configuration. In Spark, however, there is no setup method of mapper, so we use the mapPartitions[2] method. The list of required functionalities is broadcast to each method, and the annotator object is constructed at the beginning of mapPartitions, before iterating over sentences in the partition. Pseudo codes for both implementations are provided in Algorithm 1 and 2.

[1]Or documents, if we are using document level functionalities. In this subsection we only use *sentences* for the sake of simplicity.

[2]Actually, we use the mapPartitionsToPair method, because Java, unlike Scala, distinguish JavaRDD from JavaPairRDD.

Algorithm 1: MapReduce: Methods of the Mapper Class

```
void setup(context)
    func = context.getConf().get("func")
    annotator = new StanfordCoreNLP(func)
void map(index, sentence)
    for f: func do
        output = annotator.parse(sentence)
        WRITE((index, f), output)
    end
```

Algorithm 2: Spark: RDD Transformation

```
corpus
    .zipWithIndex()
    .repartition(numMapper)
    .mapPartitions(partition -> {
        annotator = new StanfordCoreNLP(func.getValue())
        while partition.hasNext() do
            sentence, index = partition.next()
            for f: func do
                output = annotator.parse(sentence)
                WRITE((index, f), output)
            end
        end
    })
```

4 EXPERIMENT

In this section, we describe the design of five experiments that we carried out to learn about the performance and efficiency of our distributed processing framework under different settings and contexts. These five experiments intend to address the following questions:

- How does the number of partitions improve the efficiency?
- How does the size of dataset influence the performance?
- Which properties have larger impacts on the efficiency?
- Does the Spark framework outperform the MapReduce framework in the current context?
- Does the Spark framework outperform the framework that runs with multiple processors on a single machine?

More specifically, we are mainly interested in how to optimize the configuration to achieve better performance in our distributed framework with Spark under different contexts and understanding how and why our framework is able to perform better than other existing frameworks on a given task.

The dataset we used is the Washington Post dataset, which takes 6.7 GB on disk space. Each line in the dataset is a JSON object, storing information associated a single post, including the title, the main content and the author etc. Each JSON object has a key *contents*, whose value is a list of key-value objects with each object represents a sentence in a post. Since most of the sentences have HTML tags around texts, we implemented a parser to strip all tags and concatenate all sentences in a post, joined with blanks. The outcome after parsing is a compressed and cleaned dataset, taking up only 2.5 GB on disk space and each line of which is a complete post. Regarding the following experiments, we partitioned the whole Washington Post dataset into different sizes: (1) a dataset with 1,000 posts, denoted as d_{1k}, (2) a dataset with 10,000 posts, denoted as d_{10k}, (3) a dataset with 100,000 posts, denoted as d_{100k}, and (4) the entire dataset, denoted as d_{600k}.

All the following experiments run on the *datasci* cluster[3].

4.1 Number of Partitions

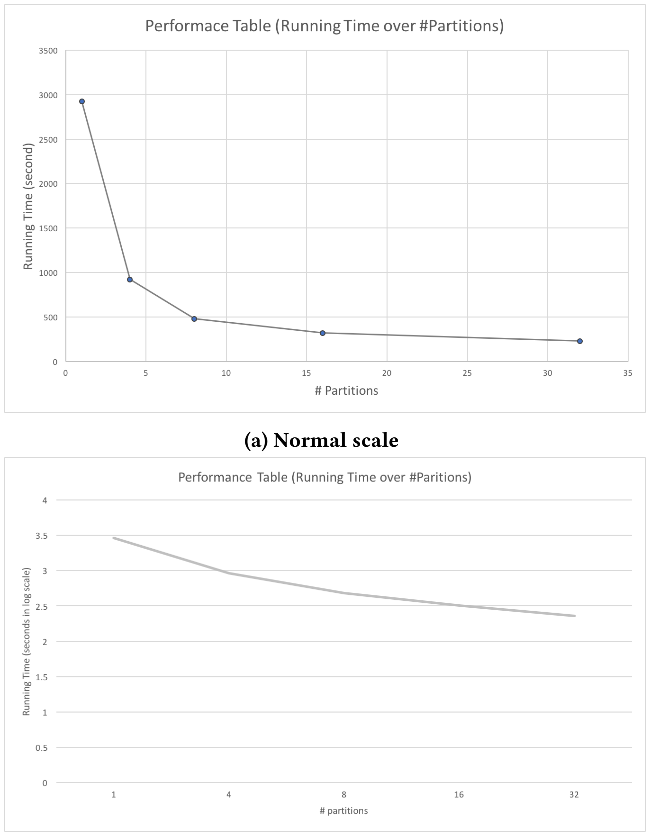
As mentioned in the Spark implementation in Section 3.2, since a CoreNLP object is not serializable when taking the RDD transformations and actions, it is not feasible to broadcast it to all workers. In addition to that, it is too resource-consuming and time-consuming to create a CoreNLP object per map, especially for properties either being loaded with a large model, such as *ner* or with a long and complex pipeline, such as *coref* or both, *quote*, see Table 1 in Appendix. Therefore, we decided to create a CoreNLP object per map-partition such that we can do initialization on per partition basis instead of per element basis. To understand how the number of partitions could exert influence on the performance, we designed the experiment as follows: we set the number of executors to be 10 and each executor had 4 cores, so there were totally 40 cores. Additionally, we set the driver memory to be 6GB and the executor memory to be 24GB. Then we set the number of partitions to be 1, 4, 8, 16, and 32 and ran the system with these configurations on the d_{10k} and the *ner* task respectively. The results are shown in Figure 2.

From the result, we notice that increasing the number of partitions can greatly improve the performance, particularly when increasing the number of partitions from 1 to 8. And when the number of partitions continuously increase, the performance improvement in speed is inconspicuous. The $\log - \log$ scale plot (Figure 2b) provides a clearer image. Ideally (when the cost of shuffling and communication between mappers can be ignored), the $\log - \log$ plot should be a straight line, since the product of number of mappers and running time should be a constant. The $\log - \log$ plot indeed demonstrates that when the number of mappers is not too big (e.g., not greater than 16), most of the time consumed is actually spent on parsing sentences.

4.2 Size of dataset

To test how the framework performs on different sizes of dataset, specifically verifying whether the system is capable of running on a large dataset, we designed the experiment below. First, based on the results of Section 4.1, we set the number of partitions to be 32. All other configurations and the *ner* task remain the same as in Section 4.1. Then, we ran the system on different datasets, i.e. d_{1k}, d_{10k}, d_{100k}. The reason we did not include the whole dataset d_{600k} here was that it needed a different configuration. It required a larger driver memory (8GB) and a larger executor memory (28GB), otherwise, the *SparkContext* would shutdown due to the huge load of data-shuffling and network I/O during the process. The results are shown in Figure 3.

[3]datasci.cs.uwaterloo.ca

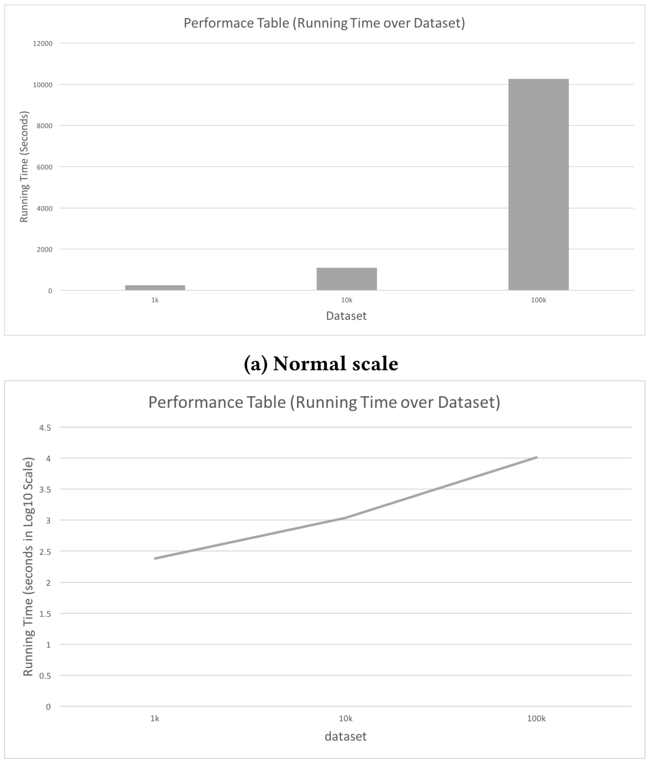


(a) **Normal scale**

(b) log-log **scale**

Figure 2: The system ran with 1, 4, 8, 16, 32 partitions respectively on d_{10k}.

The running time was around 240 seconds on d_{1k} and 1089 seconds on d_{10k}, which actually took much less time than that on d_{100k} (10275 seconds $\approx$ 2.85 hours). When a large dataset has a size 10 times bigger than a smaller one, the result indicated that the performance is approximately linearly related. Besides, the results imply that it might take many hours and resources to run CoreNLP on a large dataset, and it also verified the capability of our framework to process intensive data in a reasonable time. The log – log plot (Figure 3b) also demonstrates that the framework scales reasonably well as the size of dataset grows.



(a) **Normal scale**

(b) log-log **scale**

Figure 3: The system ran on d_{1k}, d_{10k} and d_{100k}.

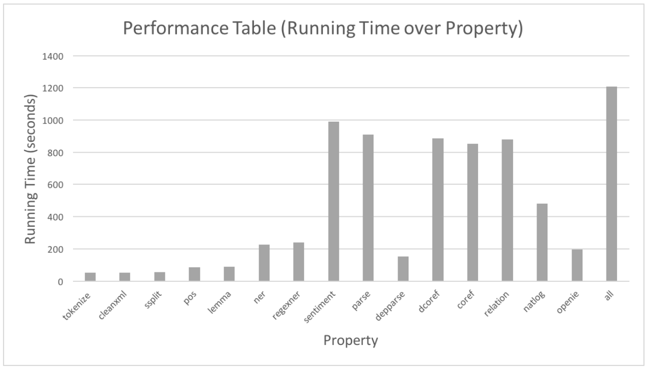


Figure 4: The system ran on each property first and ran all properties together, on d_{1k}.

4.3 Properties

Since a end-goal of this project is to support various NLP techniques as well as process document annotations at a large scale, we want to know how would the system perform using different properties in CoreNLP. In this experiment, we used the same configurations in Section 4.2, including setting the number of partitions to be 32, except changing the tasks. We ran the framework on each property first and then ran all properties together. This experiment was conducted on d_{1k}. The results are shown in Figure 4.

According to Figure 4, it is obvious that several properties (i.e., *sentiment*, *parse*, *dcoref*, *coref* and *relation*) would take longer time than other properties. One of the reasons is that unlike other properties which used annotations to manipulate a single word, these properties looked at either relations and interconnections between words or the semantic structure of a sentence. One interesting result is that running all properties together is computationally affordable as we can see that its running time is 22% larger than that of *sentiment* which is the computationally heaviest property among all properties, implying that running multiple annotations at the same time is actually an optimal task.

4.4 MapReduce vs. Spark

In this experiment, we want to measure the performance differences between MapReduce and Spark with a similar implementation. MapReduce created a CoreNLP object during the setup in every mapper, while similarly, Spark created a CoreNLP object in every partition. To compare the performance in this controlled experiment, the number of mappers in MapReduce was supposed to be equal to the number of partitions such that we created the same number of CoreNLP objects to process document annotations. To do that, we decided to run on d_{10k}, which was 44MB in memory, so we could use more mappers to increase the processing speed. We didn't use a smaller dataset d_{1k} because the size was so small that we were not able to generate a proper amount of mappers. We set the maximum split size of input spills in MapReduce to be 4MB, resulting in 11 Mappers being created when running the experiment. Then, we set all other configurations to be the same as in Section 4.2 except taking the number of partitions to be 11. The results are shown in Figure 5.

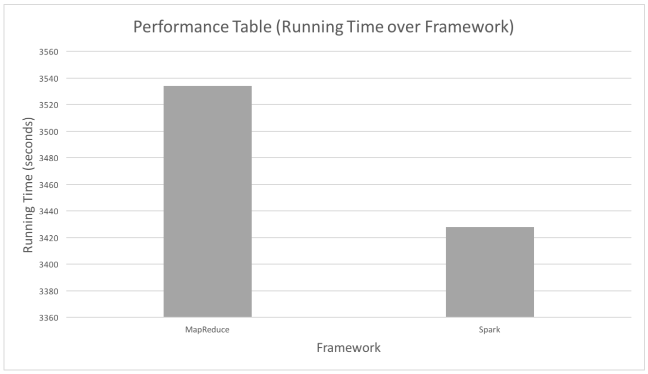


Figure 5: Performance comparison between MapReduce and Spark on d_{10k}

Based on the results in Figure 5, it took 3534 seconds and 3428 seconds to finish the *ner* task on MapReduce and Spark respectively, and it shows that the Spark version achieves 3% faster than the MapReduce version with almost the same implementation.

4.5 Spark vs. Multiple Processors On A Single Machine

We ran our Spark distributed framework on the entire dataset d_{600k} on the *ner* task. The configurations for this experiment were listed as follows: the number of partitions was 32; the number of executors was 10 and each of them had 4 cores; the driver memory was 8GB and the executor memory was 28 GB, which approximately reached the memory limit on the *datasci* cluster. The whole experiment took 57168 seconds ≈ 16 hours.

The single machine experiment was done by Peng Shi[4], on a CPU server with more than 80 CPU cores. His experiment, where he used python to run in multiple processes with 60 cores, took around 24 hours to process the whole dataset if the machine was fully available, or more than 72 hours if the machine was busy. We also replicated a similar experiment on the *Linux* cluster[5], where we used python with 60 processors to run the *ner* task on d_{1k}. This experiment took 2,397 seconds ≈ 40 minutes. As what we verified in the experiment in Section 4.2, the performance was linearly related to the size of dataset in a distributed framework, which suggested that it might take more than 160 hours to complete the *ner* task on the whole dataset on a single machine with multiple processors.

Compared with these results, our Spark distributed framework improves large-scale text processing efficiency with CoreNLP.

5 CONCLUSION AND FUTURE WORK

We implement two distributed versions of CoreNLP, using MapReduce and Spark. The Spark implementation is optimized so that (1) CoreNLP annotator object is not redundantly constructed, and (2) data shuffling is kept to minimum. We conduct experiments to examine the property of our Spark implementation in different settings, including (1) different number of mappers, (2) different size of dataset, and (3) different functionalities of CoreNLP. We also make comparisons between (1) MapReduce and Spark implementation, and (2) Spark and single machine implementation. Experiments show that our Spark implementation is efficient and scalable.

Limitations and future directions of the project include:

(1) All of our implementations and experiments are based on English, while CoreNLP also supports other natural languages as well. However, the robustness of other languages have not be considered and tested.
(2) The input format is limited to plain text, one sentence (document) per line. Other input formats such as JSON have not been supported.
(3) API calls of CoreNLP are complicated and somewhat coupled with the MapReduce (Spark) framework. It is worth trying to simplify and decouple these API calls.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Jimmy Lin for providing *datasci* system for building the framework. The authors would also like to thank Mr. Peng Shi for providing the Washington Post dataset and his experiment results for evaluation as well as discussing details and minor cases in the dataset.

REFERENCES

[1] Amal S Alblawi and Ahmad A Alhamed. 2017. Big data and learning analytics in higher education: Demystifying variety, acquisition, storage, NLP and analytics. In *2017 IEEE Conference on Big Data and Analytics (ICBDA)*. IEEE, 124–129.
[2] Apache. 2018. Hadoop Homepage. http://hadoop.apache.org/
[3] Apache. 2018. Spark Homepage. http://spark-project.org/
[4] Xabier Artola, Zuhaitz Beloki, and Aitor Soroa. 2014. A stream computing approach towards scalable NLP.. In *LREC*. 8–13.
[5] Seyed-Mehdi-Reza Beheshti, Srikumar Venugopal, Seung Hwan Ryu, Boualem Benatallah, and Wei Wang. 2013. Big data and cross-document coreference resolution: Current state and future opportunities. *arXiv preprint arXiv:1311.3987* (2013).

[4] p8shi@uwaterloo.ca

[5] linux.student.cs.uwaterloo.ca

[6] Minhwa Chung and Dan I. Moldovan. 1995. Parallel natural language processing on a semantic network array processor. *IEEE Transactions on Knowledge and Data Engineering* 7, 3 (1995), 391–405.

[7] Kevin Clark and Christopher D. Manning. 2015. Entity-Centric Coreference Resolution with Model Stacking. In *Association for Computational Linguistics (ACL)*.

[8] Kevin Clark and Christopher D. Manning. 2016. Deep Reinforcement Learning for Mention-Ranking Coreference Models. In *Empirical Methods on Natural Language Processing*. https://nlp.stanford.edu/pubs/clark2016deep.pdf

[9] Kevin Clark and Christopher D. Manning. 2016. Improving Coreference Resolution by Learning Entity-Level Distributed Representations. In *Association for Computational Linguistics (ACL)*. https://nlp.stanford.edu/pubs/clark2016improving.pdf

[10] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. In *LREC*.

[11] Tamer Elsayed, Jimmy Lin, and Douglas W Oard. 2008. Pairwise document similarity in large collections with MapReduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*. Association for Computational Linguistics, 265–268.

[12] Peter Exner and Pierre Nugues. 2014. KOSHIK-A Large-scale Distributed Computing Framework for NLP.. In *ICPRAM*. 463–470.

[13] Jenny Finkel, Trond Grenager, and Christopher D. Manning. 2005. Incorporating non-local information into information extraction systems by Gibbs sampling. In *Association for Computational Linguistics (ACL)*. https://nlp.stanford.edu/pubs/finkel2005gibbs.pdf

[14] Satish Gopalani and Rohan Arora. 2015. Comparing apache spark and map reduce with performance analysis using k-means. *International journal of computer applications* 113, 1 (2015).

[15] Lei Gu and Huan Li. 2013. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 721–727.

[16] Venkat N Gudivada and Kamyar Arbabifard. 2018. Open-Source Libraries, Application Frameworks, and Workflow Systems for NLP. *Computational Analysis and Understanding of Natural Languages: Principles, Methods and Applications* 38 (2018), 31.

[17] Thierry Hamon, Julien Deriviere, and Adeline Nazarenko. 2007. Ogmios: a scalable NLP platform for annotating large web document collections. In *Corpus Linguistics*. 195–208.

[18] Youngsub Han. 2017. A study of aspect-based sentiment analysis in social media. (2017).

[19] Prateek Jindal, Dan Roth, and Laxmikant V Kale. 2013. *Efficient development of parallel NLP applications*. Technical Report.

[20] Laxmikant V Kale and Gengbin Zheng. 2009. Charm++ and AMPI: Adaptive runtime strategies via migratable objects. *Advanced Computational Infrastructures for Parallel and Distributed Applications* (2009), 265–282.

[21] Dan Klein and Christopher D. Manning. 2003. Association for Computational Linguistics (ACL). In *Accurate Unlexicalized Parsing*. https://nlp.stanford.edu/pubs/klein2003unlexicalized.pdf

[22] Dan Klein and Christopher D. Manning. 2003. Fast Exact Inference with a Factored Model for Natural Language Parsing. In *Advances in Neural Information Processing Systems 15 (NIPS 2002)*. https://nlp.stanford.edu/pubs/klein2003factored.pdf

[23] Heeyoung Lee, Yves Peirsman, Angel Chang, Nathanael Chambers, Mihai Surdeanu, and Dan Jurafsky. 2011. Stanford's Multi-Pass Sieve Coreference Resolution System at the CoNLL-2011 Shared Task. In *Conference on Natural Language Learning (CoNLL) Shared Task*. pubs/conllst2011-coref.pdf

[24] Xiuqin Lin, Peng Wang, and Bin Wu. 2013. Log analysis in cloud computing environment with Hadoop and Spark. In *Broadband Network & Multimedia Technology (IC-BNMT), 2013 5th IEEE International Conference on*. IEEE, 273–276.

[25] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[26] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. 55–60. http://www.aclweb.org/anthology/P/P14/P14-5010

[27] Richard McCreadie, Craig Macdonald, Iadh Ounis, Miles Osborne, and Sasa Petrovic. 2013. Scalable distributed event detection for twitter. In *Big Data, 2013 IEEE International Conference on*. IEEE, 543–549.

[28] Paolo Nesi, Gianni Pantaleo, and Gianmarco Sanesi. 2015. A Distributed Framework for NLP-Based Keyword and Keyphrase Extraction From Web Pages and Documents.. In *DMS*. 155–161.

[29] Tamas Neumer. [n. d.]. Efficient Natural Language Processing for Automated Recruiting on the Example of a Software Engineering Talent-Pool. ([n. d.]).

[30] Patrick Pantel, Eric Crestan, Arkady Borkovsky, Ana-Maria Popescu, and Vishnu Vyas. 2009. Web-scale distributional similarity and entity set expansion. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 2-Volume 2*. Association for Computational Linguistics, 938–947.

[31] Alexandre Pinto, Hugo Gonçalo Oliveira, and Ana Oliveira Alves. 2016. Comparing the performance of different nlp toolkits in formal and social media text. In *OASIcs-OpenAccess Series in Informatics*, Vol. 51. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[32] Karthik Raghunathan, Heeyoung Lee, Sudarshan Rangarajan, Nathanael Chambers, Mihai Surdeanu, Dan Jurafsky, and Christopher Manning. 2010. A Multi-Pass Sieve for Coreference Resolution. In *Empirical Methods in Natural Language Processing (EMNLP)*.

[33] Marta Recasens, Marie-Catherine de Marneffe, and Christopher Potts. 2013. The Life and Death of Discourse Entities: Identifying Singleton Mentions. In *North American Association for Computational Linguistics (NAACL)*.

[34] Luís Sarmento, Alexander Kehlenbeck, Eugénio Oliveira, and Lyle Ungar. 2009. An approach to web-scale named-entity disambiguation. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer, 689–703.

[35] Sameer Singh, Amarnag Subramanya, Fernando Pereira, and Andrew McCallum. 2011. Large-scale cross-document coreference using distributed inference and hierarchical models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*. Association for Computational Linguistics, 793–803.

[36] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher Manning, Andrew Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *EMNLP*.

[37] Connor Stokes, Anoop Kumar, Frederick Choi, and Ralph Weischedel. 2015. Scaling NLP algorithms to meet high demand. In *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2839–2839.

[38] Marcel P van Lohuizen. 2000. Parallel processing of natural language parsers. In *Parallel Computing: Fundamentals and Applications*. World Scientific, 168–175.

[39] 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.

[40] 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.

A APPENDIX

Properties	Annotators	Annotations	Description	Dependencies	Supported
tokenize	TokenizerAnnotator	TokensAnnotation etc.	Tokenizes the text	None	✓
cleanxml	CleanXmlAnnotator	XmlContextAnnotation	Removes XML tags from an input document	tokenize	✓
ssplit	WordsToSentencesAnnotator	SentencesAnnotation	Splits a sequence of tokens into sentences	tokenize	✓
pos	POSTaggerAnnotator	PartOfSpeechAnnotation	Labels tokens with their POS tag	tokenize, ssplit	✓
lemma	MorphaAnnotator	LemmaAnnotation	Generates the word lemmas for all tokens in the corpus	tokenize, ssplit, pos	✓
ner	NERClassifierCombiner	NamedEntityTagAnnotation etc.	Recognizes named entities in text	tokenize, ssplit, pos, lemma	✓
regexner	TokensRegexNERAnnotator	NamedEntityTagAnnotation etc.	RegexNER Implements a simple, rule-based NER system over token sequences using an extension of Java regular expressions	tokenize, ssplit, pos	✓
sentiment	SentimentAnnotator	entimentCoreAnnotations etc.	Analyse the sentiment of a sentence	tokenize, ssplit, pos, parse	✓
truecase	TrueCaseAnnotator	TrueCaseAnnotation etc.	Recognizes the "true" case of tokens	tokenize, ssplit, pos, lemma, ner	
parse	ParserAnnotator	TreeAnnotation etc.	Provides full syntactic analysis, minimally a constituency (phrase-structure tree) parse of sentences	tokenize, ssplit, parse	✓
depparse	DependencyParseAnnotator	BasicDependenciesAnnotation etc.	Provides a fast syntactic dependency parser	tokenize, ssplit, pos	✓
dcoref	DeterministicCorefAnnotator	CorefChainAnnotation	Implements both pronominal and nominal coreference resolution	tokenize, ssplit, pos, lemma, ner, parse	✓
coref	CorefAnnotator	CorefChainAnnotation	Finds mentions of the same entity in a text	tokenize, ssplit, pos, lemma, ner, parse (Can also use depparse)	✓
relation	RelationExtractorAnnotator	RelationMentionsAnnotation etc.	Find relations between two entities	tokenize, ssplit, pos, lemma, ner, depparse	✓
natlog	NaturalLogicAnnotator	PolarityAnnotation etc.	Marks quantifier scope and token polarity, according to natural logic semantics	tokenize, ssplit, pos, lemma, depparse (Can also use parse)	✓
quote	QuoteAnnotator	QuotationsAnnotation etc.	Deterministically picks out quotes from a text	tokenize, ssplit, pos, lemma, ner, depparse, coref	✓
openie	OpenIE	RelationTriplesAnnotation etc.	Extracts open-domain relation triples, representing a subject, a relation, and the object of the relation	tokenize, ssplit, pos, lemma, depparse, natlog	✓

Table 1: CoreNLP-supported properties and corresponding annotators, generated annotations and description; supported properties in our distributed processing framework are marked.