Acta Univ. Sapientiae, Informatica **12,** 1 (2020) 51–69

DOI: 10.2478/ausi-2020-0004

Word pattern prediction using Big Data frameworks

Bence SZABARI
Eötvös Loránd University
Budapest, Hungary
email: n0qsdc@inf.elte.hu

Attila KISS
J. Selye University
Komárno, Slovakia
email: kissae@ujs.sk

Abstract. Using software applications or services, which provide word or even word pattern recommendation service has become part of our lives. Those services appear in many form in our daily basis, just think of our smartphones keyboard, or Google search suggestions and this list can be continued. With the help of these tools, we can not only find the suitable word that fits into our sentence, but we can also express ourselves in a much more nuanced, diverse way. To achieve this kind of recommendation service, we use an algorithm which is capable to recommend word by word pattern queries. Word pattern queries, can be expressed as a combination of words, part-of-speech (POS) tags and wild card words. Since there are a lot of possible patterns and sentences, we use Big Data frameworks to handle this large amount of data. In this paper, we compared two popular framework Hadoop and Spark with the proposed algorithm and recommend some enhancement to gain faster word pattern generation.

1 Introduction

Expressing ourselves in writing can be a challenging task, especially if we are not a native speaker of the target language. Fortunately, nowadays a lot of

Computing Classification System 1998: H.2, C.2
Mathematics Subject Classification 2010: 68U15
Key words and phrases: word-pattern, word-prediction, big data, hadoop, spark, nlp, map reduce, snappy, lz4, data compression

utilities can help us to express ourselves in a very diverse way, for example smart phone keyboards with word recommendation, online synonym dictionaries, or even the Google Search engine has the functionality to recommend the proper topic.

In this article, we will focus on word generation or to be more precise word pattern generation, which means that users can express their thoughts and ideas with the combination of word(s), part-of-speech (POS) tags [11] and with any arbitrary word (wild card word). Those kind of queries are called word pattern queries. For example:

$$\mathsf{VB} * \mathsf{love}$$

is a word pattern query that needs to satisfy the following requirements: a verb at the VB position, any arbitrary word at * position then word 'love'. The proper answer to that query is consists of the matched word list to the given pattern along with their relative frequencies of appearance in a large corpora.

In general, construction of word patterns can take a considerable amount of time, since there are a huge number of potential word patterns can be created based on a text, that contains just a few or even millions of rows.

Based on Erin Gilheany's comparison idea [1] and Kritwara Rattanaopas' data compression improvement [2], we were also interested in the efficiency of generating word patterns, therefore we made different experiments with Apache Hadoop [9] and Apache Spark [10] since both of them allows distributed processing of large data sets. We also made suggestions, to make the pattern generation faster in case of Hadoop. For this, we applied the Hadoop Native Library that can use Snappy and LZ4 data compression algorithms to compress the intermediate output of the mapper task. With the help of those compression codecs, we were able to achieve faster pattern generation.

2 Big data frameworks and paradigms

As we mentioned above, we have to process large amount of text files with thousand and millions of lines therefore we use Big Data frameworks like Apache Hadoop and Apache Spark.

2.1 Hadoop

Apache Hadoop [9] is a bundle of open-source software utilities that can solve problems involving large amounts of data and computation using a net-

work of many computers which are built from commodity hardware. It provides a framework for distributed storage and processing of big data using the MapReduce programming model.

The main parts of Hadoop are the distributed storage system called Hadoop Distributed File System (HDFS in short), and the processing part which is using MapReduce programming model, and YARN (Yet Another Resource Negotiator). Hadoop splits files into large block and distributes them across nodes in the cluster, then it sends the runnable code to the nodes to process data in parallel.

HDFS [13] is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems, however it has also some key differences to the others. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware which makes it economical. It also provides high throughput access to application data and is suitable for applications that have large data sets.

YARN (Yet Another Resource Negotiator) [12] is a cluster management system. It has been part of Apache Hadoop since v2.0. With the help of YARN arbitrary applications can be executed on a Hadoop cluster. Therefore, the application has to consist of one application master and an arbitrary number of containers. Latter are responsible for the execution of the application whereas the application master requests container and monitors their progress and status.

In order to execute these applications, YARN consists of two component types:

1. The **ResourceManager** is unique for a complete cluster. Its main task is granting the requested resources and balancing the load of the cluster. Furthermore, it starts the application master initially and restarts it in case of a failure.
2. On each computing node, one **NodeManager** is executed. It starts and monitors the containers assigned to it as well as the usage of its resources, i.e., CPU usage and memory consumption

2.2 MapReduce

MapReduce [8] is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

A MapReduce program is composed of a map procedure (or method), which performs filtering and sorting, and a reduce method, which performs a sum-

mary operation. The "MapReduce System" (also called "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

A MapReduce framework is usually composed of three operations:

1. **Map** each worker node applies the map function to the local data, and writes the output to a temporary storage. A master node ensures that only one copy of the redundant input data is processed.
2. **Shuffle** worker nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same worker node.
3. **Reduce** worker nodes now process each group of output data, per key, in parallel.

MapReduce allows for the distributed processing of the map and reduction operations. Maps can be performed in parallel, provided that each mapping operation is independent of the others; in practice, this is limited by the number of independent data sources and/or the number of CPUs near each source.

The Map and Reduce functions of MapReduce are both defined with respect to data structured in (key, value) pairs. Map takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

$$\mathtt{map}(\mathtt{k1}, v1) \rightarrow \mathtt{list}(\mathtt{k2}, v2)$$

The Map function is applied in parallel to every pair (keyed by k1) in the input dataset. This produces a list of pairs (keyed by k2) for each call. After that, the MapReduce framework collects all pairs with the same key (k2) from all lists and groups them together, creating one group for each key.

The Reduce function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

$$\mathtt{reduce}(\mathtt{k2}, \mathtt{list}(v2)) \rightarrow \mathtt{list}((\mathtt{k3}, v3))$$

Thus the MapReduce framework transforms a list of (key, value) pairs into another list of (key, value) pairs. This behavior is different from the typical functional programming map and reduce combination, which accepts a list of arbitrary values and returns one single value that combines all the values returned by map.

2.3 Spark

Apache Spark [10] is an open-source distributed general-purpose cluster-computing framework that can do in-memory data processing, while providing the ability to develop applications in Java, Scala, Python or even in R. Spark provides four main submodules which are SQL, MLib for machine learning, GraphX and Streaming.

In this paper, we focus on the SQL module, especially on the new data structure called DataFrame [16] which has been added in Spark 1.6. Dataframe was built on top of the previously used **RDDs** (Resilient Distributed Dataset) therefore, it is combining the benefits of RDDs (strong typing, ability to use lambda functions) and the benefits of Spark SQL's optimized execution engine.

It is also possible to use functional transformation such as map, flatMap, filter, etc. to manipulate these kind of datasets. Using those mapper functions are essentials during the word pattern generations.

2.4 Hadoop vs Spark

Spark is developed to run on top of Hadoop and this is an alternative to the traditional MapReduce model. The key differences are the following:

- Spark stores, a process data in-memory while Hadoop using the disk
- Hadoop uses the MapReduce paradigm and Spark uses the distributed data structure called datasets which are built on RDDs
- Hadoop merges and partitions shuffle spill files inte one big files, while Spark doesn't
- The MapReduce can be inefficient when the job has to reuse the same dataset, while Spark can hold the data in memory for efficient reuse

3 Word recommendation

3.1 Word recommendation problem

The applied method can help to choose the proper words in a certain context, by recommending a list of suitable words that can fit in the given words. A word pattern query is an ordered sequence of specific word(s), POS tag(s), and wild card word(s). For example:

$$VB * love$$

For each word pattern query, we want to get a list of frequently used words that match the POS tags. Furthermore, if we include the relative frequencies of the matched words that can help you to decide the right phrase for you.

The conventional language models can suggest the most appropriate words that can appear at a specific position of a sentence or phrase. However, those models are usually inappropriate for a word pattern query service. The problem with those models:

- they are not built to estimate the probability distributions for more than one words
- they are not able to understand the usage of POS tags

Therefore, they can not have any relationships between POS tags and words. In this case, we would like to model the following probability distribution $p(W_t|W_c)$ for matching word list W_t given context words W_c:

$$p(W_t|W_c) = \frac{C(W_t, W_c)}{C(W_c)}$$

where W_t denotes a list of words corresponding to POS tags in a word pattern query, $C(W_t, W_c)$ indicates the number of sentences that match the word pattern query with the words of Wt in the positions of POS tags, and $C(W_c)$ indicates the total number of sentences that match with the word pattern query. W_t part can be greater than 1, i.e., $|W_t| > 1$.

3.2 Used method for word recommendation

To create word pattern queries we used the presented algorithm in [3], which generates word patterns along with their associated information earlier on.

Word patterns can contain words, POS tags and wild card words denoted by a symbol (*). The wild card word can be any English word, so it can be useful when we are only focus on the other words within the word pattern, as they are just placeholders.

3.3 Workflow

As an overview, to construct word patterns the used method consists of the following steps:

Figure 1: Word pattern generation process

3.4 Preprocessing and tagging

To be able to create word patterns, first we have to transform the input sample texts. It may occur the sample texts contain non-English words or characters in this case they need to be removed. Numbers and punctuation marks can remain in the text.

After the transform phase has been finished, we are finally able to create POS Tagged sentences, using the Stanford POS Tagger [17]. This tool can read text and assign parts-of-speech to each word or token, e.g. noun, verb, preposition and so on. The possible POS Tags which is supported by tagger are listed in Table 2.

For example, if we have the following sentence part, after the preprocessing:

"mathematical notation widely used in physics and other sciences avoids many ambiguities compared to expression in natural language however for various reasons several lexical syntactic and semantic ambiguities remain"

Stanford POS tagger will create the output as shown below:

"mathematical/JJ notation/NN widely/RB used/VBN in/IN physics/NN and/CC other/JJ sciences/NNS avoids/VBZ many/JJ ambiguities/NNS compared/VBN to/TO expression/NN in/IN natural/JJ language/NN however/RB for/IN various/JJ reasons/NNS several/JJ lexical/JJ syntactic/NN and/CC semantic/JJ ambiguities/NNS remain/VBP"

3.5 Generating word patterns

After we got the tagged sentences, the used method creates the word patterns which consist of words, POS tags, and the wild card word symbol (*). It generates all word patterns for each n-gram of the POS-tagged sentences, and then aggregates them to get the information for word patterns.

The possible word patterns which can be generated from n-grams are listed in Table 3. To create the possible word patterns, we have to define constraints to reduce the amount of patterns to get a manageable amount of them. Therefore, we apply the following rules: there is at least one word in a pattern, if there is a wild card within the pattern it does not appear in the first or the last position in a pattern. Also, we reduce the number of possible n-grams to 5 for the reasons mentioned above.

After all word patterns of k-grams

$$k \in [2, 5]$$

are created for the provided input text, the used method clusters them into groups to have the same word pattern in each group. For each group the sentence Ids which have the same word lists are grouped together and their count is computed.

We are interested in the most frequent word list for each word patterns to be able to create a word recommendation service, so the used method constructs a word pattern database which maintains the frequent word lists along with their corresponding sentence Ids and their frequency for each word pattern.

3.6 Map reduce solution

The method use two cycles of Map-Reduce tasks. In the first Map-Reduce phase, the mapper receives sentences with their Ids and produces the key-value pairs where the key is made of a word pattern and its corresponding word list while the value is the sentence Id. Futhermore, the reducer of the first cycle aggregates the sentence Ids of word patterns generated by mapper. In the second cycle, the mapper computes the frequencies of the combination of a word pattern and its word lists, and then the reducer aggregates the results and also retains the top k-th word lists with the highest frequencies.

The following procedure is the algorithm for the first mapper in the first phase. It creates the word patterns for 2-grams to 5 grams for each sentence. The cands variable holds the possible word patterns, so $\mathtt{cands}[n][i][j]$ indicates the j-th value for the i-th word list of the n-gram.

In the first mapper-phase, we process POS tagged lines (assuming that we have the sentence Ids) and as an output the algorithm will create a pairs of

$$([\mathit{word\ pattern}, \mathit{word\ list}], \mathit{sentence\ id})$$

where the word pattern is generated based on the possible word patterns described in Table 3, while word-list contains the matched words corresponding to the pattern and the value will be the sentence id.

Algorithm 1: Mapper phase 1

```
Input  :  (sentence id, sentence with POS tags)
Output:  ([word-pattern, word-list], sentence id)
value ← sentenceId;
tokenize the POS tagged sentence, into a collection of tokens;
tokens ← t1, t2, ... t3;
m ← tokens.size;
for n ← 2 to 5 do
    for s ← 0 to m−n do
        for i ← 0 to cands[n].size − 1 do
            pattern ← [];
            metWords ← [];
            for j ← 0 to n − 1 do
                word ← word at (s+j) th position of tokens;
                pos ← POS Tag at (s+j) th position of tokens;
                // j-th position is a word Tag in the pattern
                if cands[n/i][j] = 'w' then
                    pattern.add(word);
                // j-th position is a POS Tag in the pattern
                else if cands[n/i][j] = 'p' then
                    pattern.add(pos);
                // j-th position is a wildcard in the pattern
                else
                    pattern.add(_WC_);
                end
                if cands[n/i][j] > '0' and pos is a legal tag then
                    metWords.add(word);
            end
            if metWords.isEmpty() then
                metWords.add(_NONE_);
            key ←pattern + ";" + metWords;
            emit(key, value)
        end
    end
end
```

3.7 Example

Let's take an example, in that case when we have the following short sentence: *lincoln/NN practiced/VBD law/NN*. On the next page, we present some sample output. For the sake of simplicity and transparency, we categorized the output by n-grams and omitted the sentence ids.

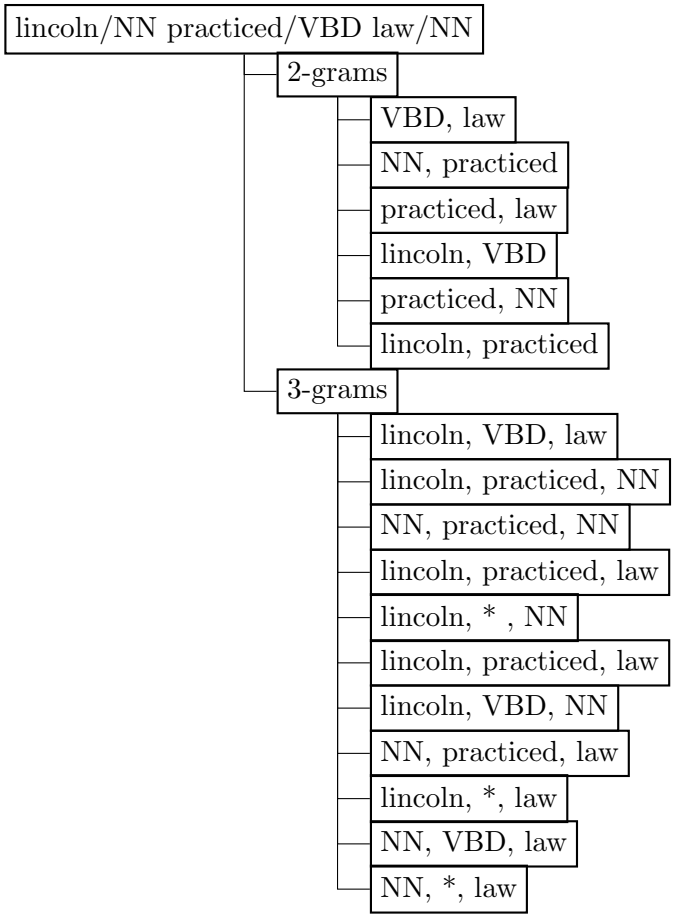

Figure 2: Word patterns example

3.8 Reducer phase 1

The first reducer phase aggregates the sentence ids according to the values of (pattern, word-list). It also removes the duplicated occurences of the same id for the same key.

3.9 Mapper phase 2

The second mapper calculates the relative frequencies of the different matched word list and also extract the pattern from the old key to use as a new key.

Algorithm 2: Reducer phase 1

Input : pairs of ((pattern, word list), sentence id)
Output: ([pattern, word-list]), sentence-id-list)
key ← [pattern, word list];
value ← sentence id list;
emit(key, value)

Therefore, we will get pairs like this:

$$word\ pattern \rightarrow (f_1, w_1, sid_1),\ \ldots\ , (f_n, w_n, sid_n)$$

where f_i denotes the ith relative frequency, w_i the ith matched word list, and sid_i is the ith the sentence id.

Algorithm 3: Mapper phase 2

Input : ([pattern, word-list]), sentence-id-list)
Output: pattern, [frequency, word list, sentence id list])
if *word-list.isEmpty()* **then**
  word list.add(_NONE_);
key ←pattern;
value ← [frequency, word list, sentence id list];
emit(key, value)

3.10 Reducer phase 2

In the last reducer phase we remain the top k-th (in our example 10) matched word list with the highest relative frequency. The received result can be the basis of a word pattern database.

4 Hadoop native library

Hadoop can support native libraries [14] out of the box, which includes components like compression codecs (e.g.: bzip2, lz4, snappy and zlib), native io utilities for Centralized Cache Management in HDFS or even CRC32 checksum implementation.

From this library, we mainly focus on the compression codecs that we can use to compress the Mapper job's intermediate output in a Hadoop MapReduce.

Algorithm 4: Reducer phase 2

Input : pattern, [[frequency, word-list, sentence-id-list] ...]
Output: pattern, [[frequency, word-list, sentence-id-list] ...] (top k-th)

key ←pattern;
value ← [];
list ← [[frequency, word − list, sentence − id − list]...];
sort the list by frequency, in decreasing order;
put the first k-th element from list to value;
emit(key, value)

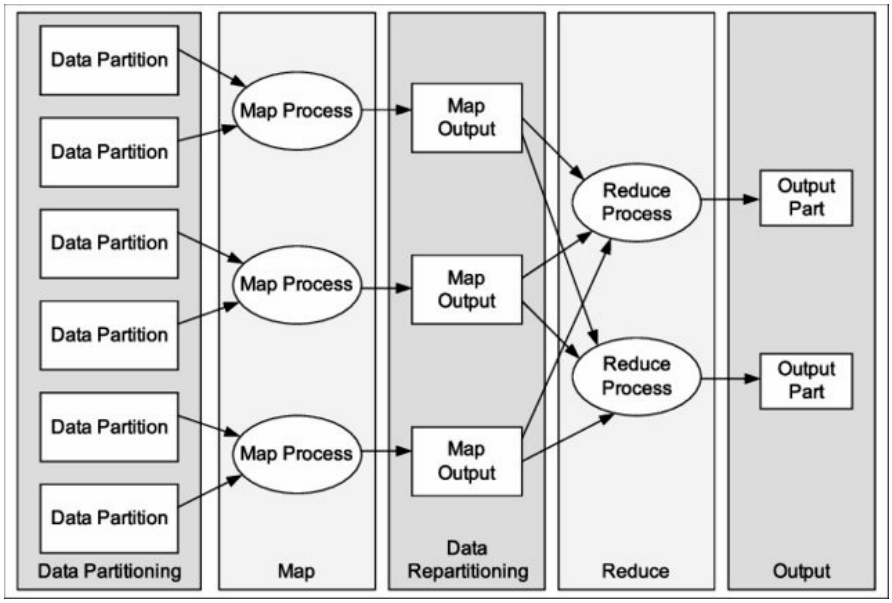

Figure 3: Computing Stages of the MapReduce model [4]

4.1 Compression codecs

4.2 Snappy

Snappy (previously known as Zippy) [7] is a fast data compression and decompression library written in C++ by Google based on ideas from LZ77 and open-sourced in 2011. It does not aim for maximum compression, or compatibility with any other compression library; instead, it aims for very high speeds

and reasonable compression. The compression ratio is 20–100% lower than gzip.

4.3 Lz4

LZ4 [5] is a lossless data compression algorithm that is focused on compression and decompression speed. It belongs to the LZ77 family of byte-oriented compression schemes.

The LZ4 algorithm represents the data as a series of sequences. Each sequence begins with a one-byte token that is broken into two 4-bit fields. The first field represents the number of literal bytes that are to be copied to the output. The second field represents the number of bytes to copy from the already decoded output buffer (with 0 representing the minimum match length of 4 bytes). A value of 15 in either of the bitfields indicates that the length is larger and there is an extra byte of data that is to be added to the length. A value of 255 in these extra bytes indicates that yet another byte to be added. Hence arbitrary lengths are represented by a series of extra bytes containing the value 255. The string of literals comes after the token and any extra bytes needed to indicate string length. This is followed by an offset that indicates how far back in the output buffer to begin copying. The extra bytes (if any) of the match-length come at the end of the sequence.

5 Experiments

As we introduced, we used Apache Hadoop and Apache Spark to measure execution times generating word patterns. During these experiments we also applied several data compression libraries to achieve faster pattern generation. To measure those generation times and see how efficient can be a selected profile we used Monte Carlo method. [6] In our experiments, the following scenarios were compared: standard Hadoop Mapreduce job without any compression codecs, Hadoop Mapreduce job with Snappy compression codec, Hadoop Mapreduce job with LZ4 compression codec, and standard Spark job using dataframes. Furthermore, we used Wikipedia dumps to provide a suitable input for our measurements. [18]

5.1 Experiment settings

The experiments were run on a 21 node cluster with the following configuration: master node has 32 Gb RAM, 12 vCPU while slaves have 15 Gb ram, 8

vCPU. In both cases, Yarn was used as a resource manager and its configuration Node Managers were set to allocate **13** Gb for Containers. The Mapper Container's and the Reducer Container's memory size size were set to **4** Gb. Additional configuration and settings, and even more technical detail can be found at the project git repository [15]

5.2 Results

In the first measurements we compared the standard Hadoop MapReduce jobs against MapReduce jobs with compression codecs.

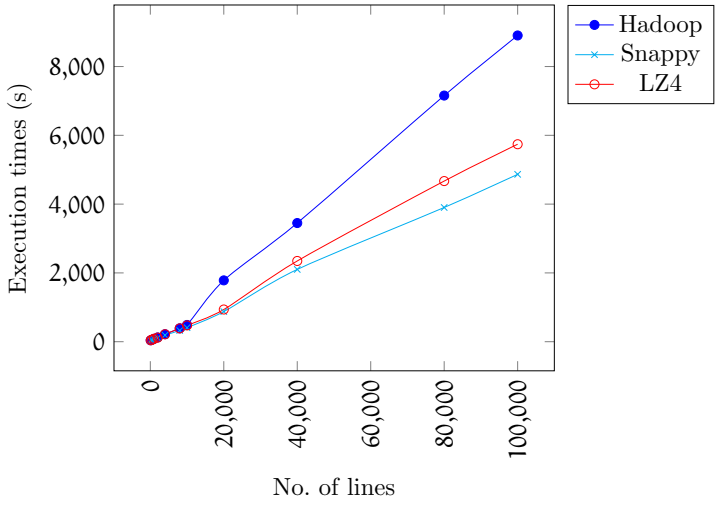

Figure 4: Hadoop Word pattern generation

Either using LZ4 or Snappy, both gave us significantly better results, but Snappy was the most spectacular codec for improvement. Therefore we conclude that, using those compression libraries can reduce the overhead during the word pattern generation and we can get better execution times. In the next phase, we used the Spark with Scala implementation of the word pattern generation and compared against a standard Hadoop job.

As we can see there is a significant difference between the two run times. In the last measurement, we compared all the previously used profile.

In summary, Snappy and the LZ4 compression codecs brought a convincing improvement during the pattern generation. Also, using the Spark framework was able to beat his rival, however it is still possible to have different SparkSQL query optimization so we can reduce word pattern generation time.

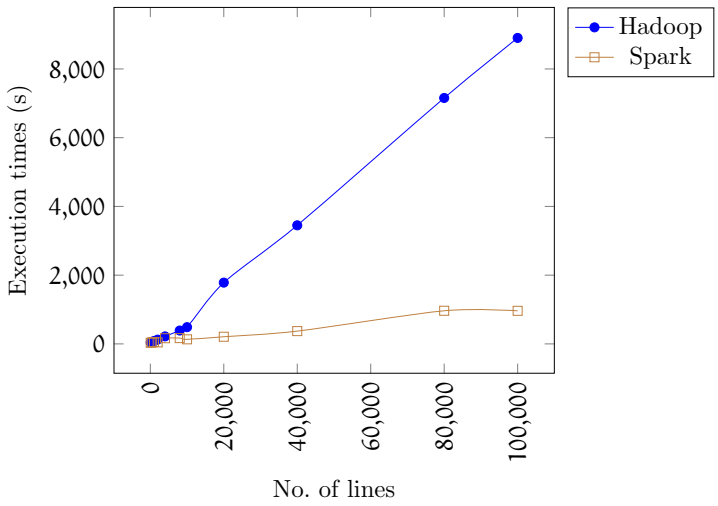

Figure 5: Hadoop compared to Spark

No. of lines	No. of words	No. of unique words	Hadoop	Hadoop with Snappy	Hadoop with LZ4	Spark
100	7045	2578	41 s	40 s	42 s	33 s
500	34 206	9351	58 s	53 s	59 s	44 s
1000	73 963	17 241	84 s	75 s	85 s	56 s
2000	143 092	28 502	124 s	108 s	119 s	44 s
4000	305 789	50 010	218 s	189 s	216 s	165 s
8000	588 912	79 933	389 s	326 s	386 s	169 s
10 000	738 533	93 828	487 s	407 s	473 s	133 s
20 000	1 497 350	151 253	1783 s	876 s	936 s	208 s
40 000	2 912 093	237 531	3450 s	2101 s	2347 s	372 s
80 000	5 858 805	374 986	7156 s	3902 s	4670 s	964 s
100 000	7 192 730	438 023	8903 s	4866 s	5741 s	962 s

Table 1: Experiment results

6 Conclusion

This paper introduced a word recommendation problem [3] where the user can express themselves using word pattern queries that contain words, POS tags and wild cards and as a result they get back matched word lists along with their relative frequencies.

For this problem, we proposed several fine tuning to reduce the overhead during the word pattern generation. To achieve this, we first introduced the Hadoop Native Library which contains compression codecs like Snappy, LZ4,

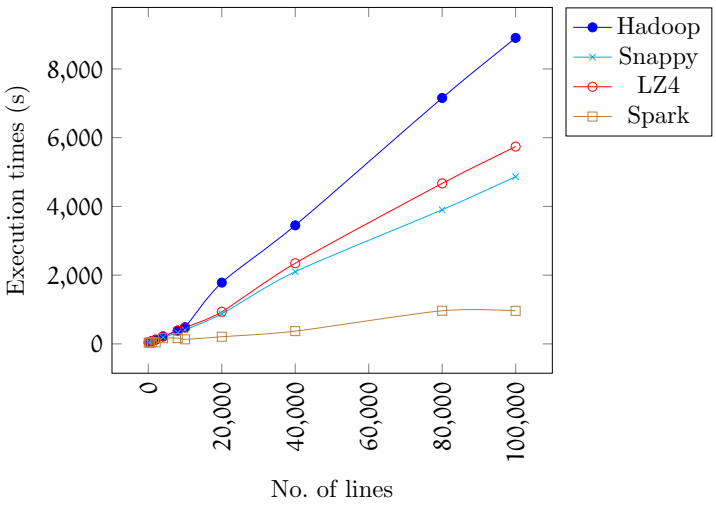

Figure 6: Word pattern generation summary

ZLib then we chose the first two of them, to see how they act during pattern generation, and then we compared the Spark framework with Hadoop.

7 Future works

As we pointed out in case of Spark there is still options to get better execution times than Hadoop. For example, fine tuning the built-in Spark Catalyst query optimizer. From the perspective of Hadoop, there are several compression codecs that we have not tested, and we used the default configuration, therefore a more advanced configuration / profile can be a good starting point for further examinations.

8 Acknowledgements

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

This publication is the partial result of the Research & Development Operational Programme for the project "Modernisation and Improvement of Technical Infrastructure for Research and Development of J. Selye University in the Fields of Nanotechnology and Intelligent Space", ITMS 26210120042, co-funded by the European Regional Development Fund.

References

[1] G. Erin. *Processing time of TFIDF and Naive Bayes on Spark 2.0, Hadoop 2.6 and Hadoop 2.7: Which Tool Is More Efficient?*, Msc Thesis, National College of Ireland Dublin, 2016. ⇒52

[2] K. Rattanaopas, S. Kaewkeeree. Improving Hadoop MapReduce performance with data compression: A study using wordcount job, *2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology* (ECTICON). IEEE, 2017. p. 564-567 ⇒52

[3] KM. Lee, CS. Han, KI. Kim, SH. Lee, Word recommendation for English composition using big corpus data processing, Cluster Computing, (2019), 1911-1924. ⇒56, 65

[4] M. Kontagora, H. Gonzalez-Velez, Benchmarking a MapReduce Environment on a Full Virtualisation Platform, *The 4th International Conference on Complex, Intelligent and Software Intensive Systems*, 433-438. 10.1109/CISIS.2010.45. ⇒62

[5] M. Bartík, S. Ulbik, P. Kubalik Matěj. LZ4 compression algorithm on FPGA, *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS). IEEE, 2015* ⇒63

[6] RY Rubinstein, DP. Kroese, Simulation and the Monte Carlo method. Vol. 10. John Wiley & Sons, 2016. ⇒63

[7] R Lenhardt,J Alakuijala, Gipfeli-high speed compression algorithm. *2012 Data Compression Conference (pp. 109-118). IEEE* ⇒62

[8] H. Karloff, S. Suri, S. Vassilvitskii, A model of computation for MapReduce. *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2010.* ⇒53

[9] Apache Hadoop, Apache, `https://hadoop.apache.org/` ⇒52

[10] Apache Spark, Apache, `https://spark.apache.org/` ⇒52, 55

[11] E. Brill, A simple rule-based part of speech tagger, *Proceedings of the third conference on Applied natural language processing. Association for Computational Linguistics, 1992.* ⇒52

[12] Apache Yarn, Apache, `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html` ⇒53

[13] Apache HDFS docs, `https://hadoop.apache.org/docs/r1.2.1/` ⇒53

[14] Hadoop Native Library, `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/NativeLibraries.html` ⇒61

[15] Project repository, `https://gitlab.com/thelfter/word-prediction` ⇒64

[16] Spark Sql, `https://spark.apache.org/docs/latest/sql-programming-guide.html` ⇒55

[17] Stanford part-of-speecg tagger, `https://nlp.stanford.edu/software/tagger.html` ⇒57

[18] Wikipedia dumps, `https://dumps.wikimedia.org/` ⇒63

9 Appendix

Table 2: Part-of-speech (POS Tags)

CC coordinating conjunction	POS possessive ending
CD cardinal number	PRP personal pronoun
DT determiner	PRP$ possessive pronoun
EX existential there	RB adverb
FW foreign word	RBR adverb, comparative
IN preposition or subordinating conjunction	RBS adverb, superlative
JJ adjective	RP particle
JJR adjective, comparative	SYM symbol
JJS adjective, superlative	TO to
LS list item marker	UH interjection
MD modal	VB verb, base form
NN noun, singular or mass	VBD verb, past tense
NNS noun, plural	VBG verb, gerund or present participle
NNP proper noun, singular	VBN verb, past participle
NNPS proper noun, plural	VBP verb, non-3rd person singular present
PDT predeterminer	VBZ verb, 3rd person singular present
WDT Wh-determiner	WP$ possessive wh-pronoun
WP Wh-pronoun	WRB Wh-adverb

Table 3: Possible word pattern: w word, p POS tag, * wild card

2-gram (3 cases)	w w	w p	p w
3-gram (10 cases)	w w w	w p p	
	p w w	p * w	w w p
	w * w	p w p	w p w
	w * p	p p w	
4-gram (32 cases)	w w w w	w w w p	w w p w
	w w p p	w w * w	w w * p
	w p w w	w p w p	w p p w
	w p p p	w p * w	w p * p
	w * w w	w * w p	w * p w
	w * p p	w * * w	w * * p
	p w w w	p w w p	p w p w
	p w p p	p w * w	p w * p
	p p w w	p p w p	p p p w
	p p * w	p * w w	p * w p
	p * p w	p * * w	

5-gram (100 cases)	w w w w w	w w w w p	w w w p w
	w w w p p	w w w * w	w w w * p
	w w p w w	w w p w p	w w p p w
	w w p p p	w w p * w	w w p * p
	w w * w w	w w * w p	w w * p w
	w w * p p	w w * * w	w w * * p
	w p w w w	w p w w p	w p w p w
	w p w p p	w p w * w	w p w * p
	w p p w w	w p p w p	w p p p w
	w p p p p	w p p * w	w p p * p
	w p * w w	w p * w p	w p * p w
	w p * p p	w p * * w	w p * * p
	w * w w w	w * w w p	w * w p w
	w * w p p	w * w * w	w * w * p
	w * p w w	w * p w p	w * p p w
	w * p p p	w * p * w	w * p * p
	w * * w w	w * * w p	w * * p w
	w * * p p	w * * * w	w * * * p
	p w w w w	p w w w p	p w w p w
	p w w p p	p w w * w	p w w * p
	p w p w w	p w p w p	p w p p w
	p w p p p	p w p * w	p w p * p
	p w * w w	p w * w p	p w * p w
	p w * p p	p w * * w	p w * * p
	p p w w w	p p w w p	p p w p w
	p p w p p	p p w * w	p p w * p
	p p p w w	p p p w p	p p p p w
	p p p * w	p p * w w	p p * w p
	p p * p w	p p * * w	p * w w w
	p * w w p	p * w p w	p * w p p
	p * w * w	p * w * p	p * p w w
	p * p w p	p * p p w	p * p * w
	p * * w w	p * * w p	p * * p w
	p * * * w		

Received: January 31, 2020 • Revised: February 28, 2020