QATAR UNIVERSITY

COLLEGE OF ENGINEERING

SPARK-IR: A SCALABLE DISTRIBUTED INFORMATION RETRIEVAL ENGINE OVER SPARK

BY

SARA YAQOOB AL-RASBI

A Thesis Submitted to

the College of Engineering

in Partial Fulfillment of the Requirements for the Degree of

Masters of Science in Computing

January 2020

COMMITTEE PAGE

The members of the Committee approve the Thesis of
Sara Yaqoob Al-Rasbi defended on 03/12/2019.

Tamer Elsayed
Thesis/Dissertation Supervisor

Aiman Erbad
Committee Member

Omar BOUSSAID
Committee Member

Approved:

Khalid Naji , Dean, College of Engineering

ABSTRACT

Al-Rasbi, Sara Y., Masters : January : 2020:, Masters of Science in Computing Title: SparkIR: a Scalable Distributed Information Retrieval Engine over Spark. Supervisor of Thesis: Tamer M. Elsayed.

Search engines have to deal with a huge amount of data (e.g., billions of documents in the case of the Web) and find scalable and efficient ways to produce effective search results. In this thesis, we propose to use Spark framework, an in-memory distributed big data processing framework, and leverage its powerful capabilities of handling large amount of data to build an efficient and scalable experimental search engine over textual documents. The proposed system, SparkIR, can serve as a research framework for conducting information retrieval (IR) experiments. SparkIR supports two indexing schemes, document-based partitioning and term-based partitioning, to adopt document-at-a-time (DAAT) and term-at-a-time (TAAT) query evaluation methods. Moreover, it offers static and dynamic pruning to improve the retrieval efficiency. For static pruning, it employs champion list and tiering, while for dynamic pruning, it uses MaxScore top k retrieval. We evaluated the performance of SparkIR using ClueWeb12-B13 collection that contains about 50M English Web pages. Experiments over different subsets of the collection and compared the Elasticsearch baseline show that SparkIR exhibits reasonable efficiency and scalability performance overall for both indexing and retrieval. Implemented as an open-source library over Spark, users of SparkIR can also benefit from other Spark libraries (e.g., MLlib and GraphX), which, therefore, eliminates the need of using

other big data frameworks (e.g., Elasticsearch) for the search applications and experimental research.

DEDICATION

I dedicate this work to myself, family, friends, colleagues and to anyone who asked me" When will you finally finish this?"

ACKNOWLEDGMENTS

I want to thank my supervisor Dr. Tamer Elsayed for his guidance and patience, and Mr. Sajeer for administrating and fixing all the related issues in our experiment environment. And lastly, thanks to the bigIR research team for their help and kind words, and most especially Reem for her continuous support during my journey in completing this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 : Introduction

In recent years, a large volume of data has been generated daily at a variable rate, especially with the existence of social media. For instance, on Twitter alone, approximately 6000 tweets are posted in a second[1]; by the end of the day, the number of posts can easily exceed a million. This vast amount of data is commonly known as Big Data. Big data is defined as large data that can be structured, semi-structured or unstructured, and traditional data science techniques fail to process it due to its huge volume. Big data is generated from many sources, especially from mobile devices; such as smartphones and sensors. Also, it exists in many sectors like healthcare, businesses, the government sector, and more.

Data scientists usually characterize big data by the 3 V's: Volume, Variety, and Velocity. Volume refers to the size of the data. Variety refers to the different forms of the data, and velocity describes how fast the data is generated. For a long time, volume, variety, and velocity were used to describe big data and were known as the three main characteristics of big data. However, as big data got more and more popular, and further studies and research were conducted in this field, two new characteristics were added: value and veracity [1]. The value indicates if the data is worth analyzing and could potentially lead to exciting discoveries, while veracity represents if the data can be trusted or not.

As mentioned previously, the size and sometimes complexity of big data makes it extremely difficult for traditional data science techniques to process it; therefore, designated frameworks were developed for processing big data [2]. Google was one

[1] https://www.internetlivestats.com/twitter-statistics/

of the pioneers of big data processing frameworks with the introduction of MapReduce. After that, more and more frameworks and technologies were developed to process different types of big data; such as, Hadoop, Hive, HBase, and others that can be integrated to process big data and extract value from it. These technologies are used in different sectors to build different applications that process big data, such as; healthcare analysis software, recommendation systems, search engines, and more.

Search engines are one of the most trivial reasons for the need for big data processing frameworks. For example, the estimated size of the indexed web is approximately 5.03 billion pages[2]; moreover, these webpages are stored in a distributed environment. Thus, to index them a system must be able to work in a distributed environment and handle extensive data processing. Google originally built MapReduce to index the web, and the open-source version of MapReduce, Hadoop, is also used for indexing and other big data analytics purposes. However, since then, more technologies became available. For instance, Spark is an in-memory data analytics processing engine that outperformed Hadoop in the big data field in terms of speed[3].

Other than indexing, a standard Search engine has another stage called retrieval as shown in Figure 1. Indexing is an offline process that takes a collection of webpages as input and builds an index of words. In information retrieval, each record in a collection is referred to as a document. The system starts by reading the collection from disk. Then each document goes through a pipeline of preprocessing. The output of the preprocessing, are the words after applying text operations on them like

[2] https://www.worldwidewebsize.com

tokenizing, stemming, lowercasing, and more. In IR these are referred to as terms. Then, the indexer will create an inverted index for the collection.

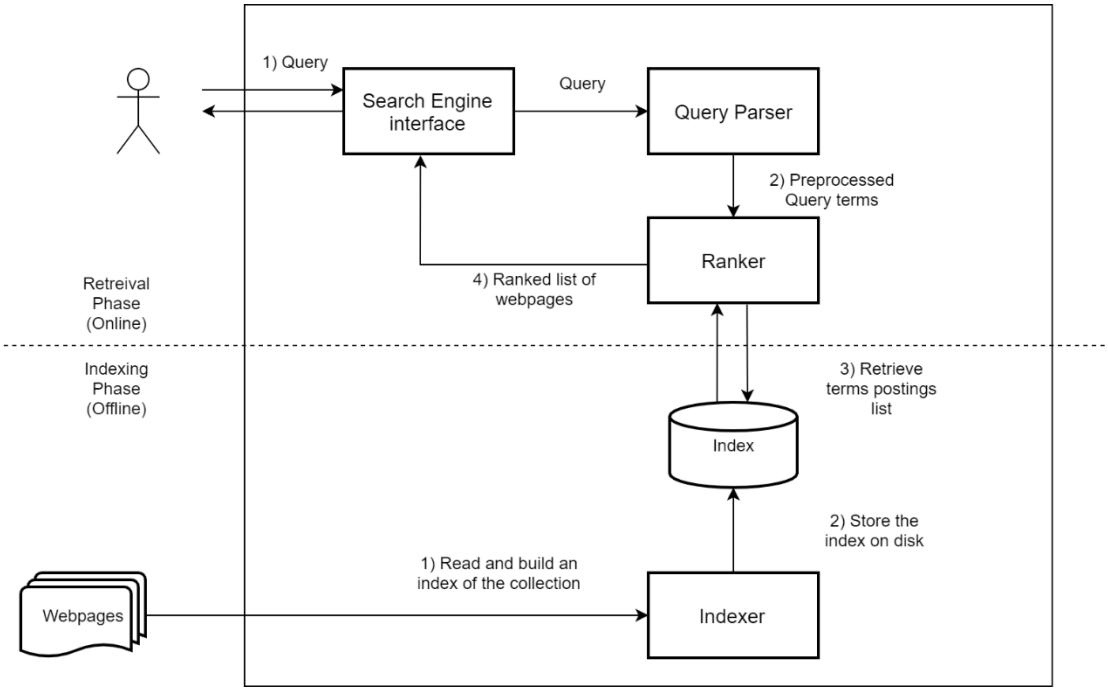

Figure 1. The architecture of an IR system

An inverted index - known merely as the index - has two parts: a dictionary and postings list. The dictionary is the terms that appear in the collection, while the postings list is a list of document ids that contain the associated term. Refer to Figure *2* for an illustration of the indexing process and an example of an inverted index. Usually, the webpage collection is distributed across many nodes; therefore, the indexer must work on parallel on a distributed environment to build the index in an efficient way that is both fast and guarantees the correctness of the built index.

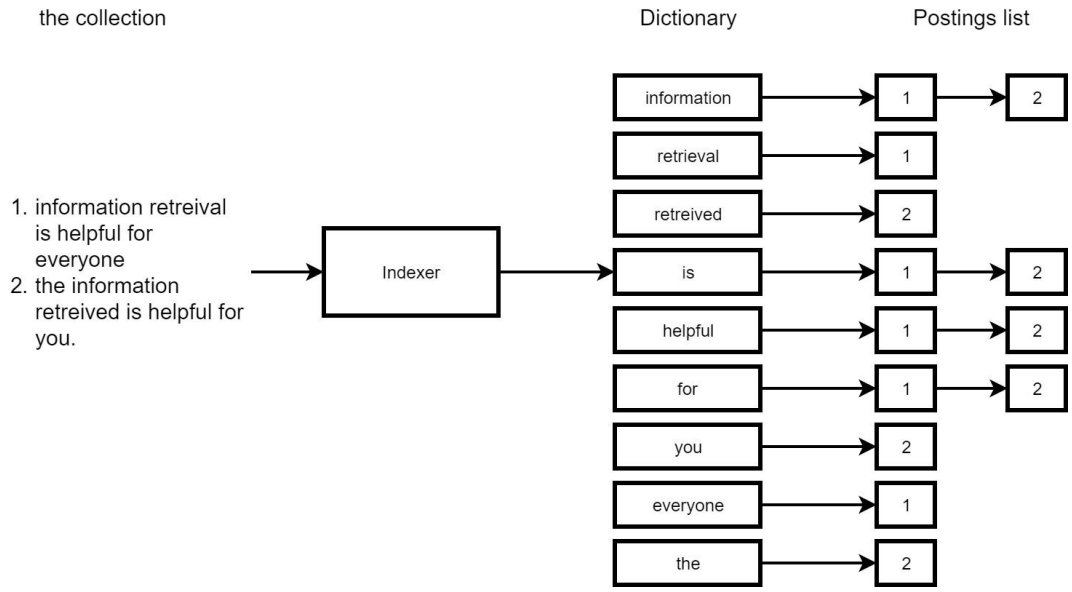

Figure 2. The indexing process

Once the index is built and saved to disk, the retrieval stage is ready to begin. In this stage, the user issues a query to the system; then, the query will go through a parser where it is reprocessed in the same pipeline as the collection. After that, the system will look up the query terms in the index. If the terms exist in the index, they are retrieved with their associated postings lists. Each document in the postings lists of the terms is given a score using a scoring method. This score indicates how relevant the document is to the user query. If the score is high, then the document is highly relevant; otherwise, it is not if the document score is low. Finally, the system will display the top ranking documents to the user. Retrieval is a real-time process, as results are returned to the users in a split of a second; thus, it is essential to have a model that is both efficient and accurate.

Search engines are meant to be fast, interactive, and handle a large number of webpages efficiently. A massive collection of documents needs first to be indexed,

and the retrieval is done in real-time. Therefore, it has to be able to get the relevant information to the user's needs while meeting the user expectation of the speed of the retrieval. Thus, in this thesis, we propose to use Spark framework and make use of its capabilities and efficiency of handling distributed big data to build an efficient and scalable search engine over textual documents. Spark is an in-memory big data analytics engine. Unlike Hadoop, Spark can keep the data in memory between jobs, whereas Hadoop has to read and write the data from disk. This property can be leveraged in building a retrieval system on Spark by loading the collection in memory and create the index from it. Moreover, the index can be loaded in memory in the retrieval and issue queries on it. The proposed system can be used as a library over Spark for Ad-hoc retrieval.

The proposed solution consists of an indexer that can index and partition the index by term or document. Moreover, to enhance the efficiency of the system, we employ different efficiency techniques. The techniques are classified as static pruning and dynamic pruning.

Static pruning techniques are done during the indexing phase, while dynamic pruning occurs in real-time while scoring. For the static pruning, we choose two techniques that are commonly used in retrieval systems which are Champion list and Tiering. In Champion list, the postings list is reduced to the top N postings with the highest term frequency (*tf*). In tiering, the index is divided into separate tiers; such that, the *tf* of a term in a document is decreasing in each tier. As for the dynamic pruning, we included MaxScore, where postings are skipped during scoring if they are not promising enough to appear in the final results.

The system was compared against Elasticsearch, a distributed IR system built over Lucene. Both systems were evaluated using a subset of ClueWeb12; a collection

of 733,019,372 English webpages collected in 2012. Although SparkIR did not outperform Elasticsearch in terms of effectiveness, where Elasticsearch achieved a precision of 0.2. While our proposed system scored 0.18, it did perform better than Elasticsearch in terms of response time after applying the efficiency methods. In SparkIR, by applying the efficiency methods reduced the response time but had a slight effect on the system's overall effectiveness.

We offer a research framework for conducting information retrieval (IR) experiments for researchers to conduct their experiments on Spark. To the best of our knowledge, no such system has been proposed over Spark within the information retrieval research community. Moreover, our distributed IR system offers efficiency techniques that distributed search engines like Solr and Elastisearch do not support.

This work would address the following research questions:

RQ1: Is Spark framework suitable for web search?

RQ2: Which partitioning technique is better suited for Spark?

RQ3: How does Spark perform with and without efficiency techniques?

RQ4: How does Spark compare to other distributed IR engines?

The following sections are as follows; first, we will discuss Spark and how it is used in recent research, as well as, a description of Hadoop, Solr, and Elasticsearch in chapter 2. Then in chapter 3, we will discuss in depth our proposed system. After that, in the next chapter, we will evaluate the proposed system against Elasticsearch. Finally, we end by concluding the significant lessons from this work and possible future plans in chapter 5.

Chapter 2 : Background and related work

2.1 Background

2.1.1 Hadoop

Although when Google revealed MapReduce, it was a huge step for big data analytics. Google did not release it for the public. A few years later, after publishing [4], Hadoop was publicly released as an open-source Java implementation of MapReduce that is now licensed by Apache.

The framework adopts a divide-and-conquer approach [2], the three main phases of MapReduce are: Map Phase, Shuffling Phase, and Reduce Phase. In the map phase, the data is read by multiple mappers at the same time, and the same programming code is used on all of them. The purpose of the map phase is to extract a common knowledge from the input data; for instance, if the programmer wanted to count the occurrences of the words in each input file, then the output of the mappers would be the word and the number of times it appeared in that input file. As implied from the example, the output of a single mapper is a list of key/value pairs. In the shuffling phase, these pairs are shuffled across the network and grouped by key, such that each key will have a list of all the different values it was assigned to by different mappers. After that, the reduce phase will reduce that list to at least one value per key. In the case of counting the word appearances, the goal of the reducer is to count the values assigned to the key to get the total occurrences across all data.

Aside from the MapReduce programming model, the Hadoop project licensed by Apache contains Hadoop commons, Hadoop HDFS, and Hadoop YARN [2].

1. Hadoop Commons: common utilities used to support Hadoop modules

2. Hadoop HDFS: A storage file system used by Hadoop. It provides reliable and distributed data access over a cluster of nodes.
3. Hadoop YARN: A cluster management framework that handles job scheduling and cluster resources management.

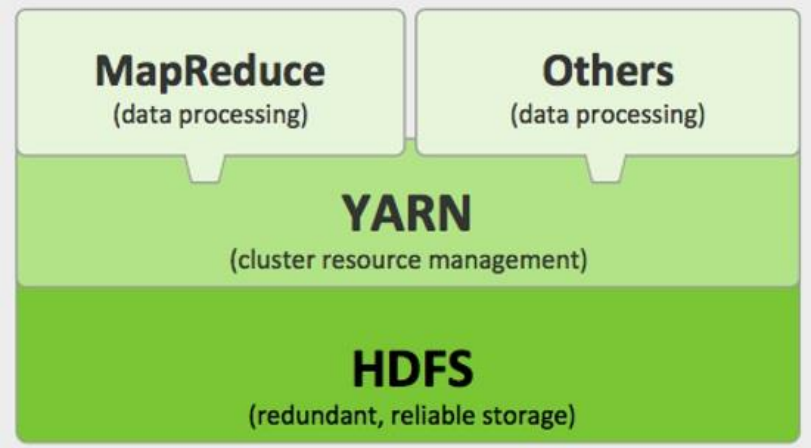

Figure 3. Hadoop Architecture

Hadoop was initially designed as a batch processing framework with the components defined earlier, and as shown in

Figure 3, but more and more tools have been developed by researchers that can be used on top of Hadoop. These tools are what makes Hadoop one of the most used framework for big data analytics. For instance, HBase [5] is a column-oriented NoSQL database. It is an open source project built to replicate BigTable, a distributed storage system for structured data developed by Google [6]. Mahout [2] is a machine learning library that includes most of the well-known algorithms for clustering, classification, data mining and more.

2.1.2 Spark

For many years Hadoop was the most popular framework for handling big data; however, it suffers from many limitations. Due to the nature of MapReduce, it does not support iterative nor interactive algorithms. Between MapReduce jobs, Hadoop has to write the data to disk. Thus, if the input of a job depends on the output of a previous job, the framework must write the output of the first job to disk so that the second job can read it and execute its processing. Moreover, Hadoop provides only the MapReduce framework; in order to perform other data analytics, external libraries that are compatible with Hadoop must be imported, such as Hive, Mahout, and HBase.

These issues led to the development of a new framework called Spark. Similar to Hadoop, Spark is also a framework to process big data, but Spark supports in-memory processing; as a result, Spark outperforms Hadoop in terms of speed. Zahiera et al. [3] proposed Spark with the intent to overcome Hadoop limitations. Firstly, Spark takes into consideration iterative algorithms; thus, in between jobs, the intermediate output is kept in memory for further processing, which allows Spark to be faster than Hadoop at accessing the data. Also, Spark comes with four components: SparkSQL, Spark MLlib, Spark Streaming, and GraphX. Considering Spark comes with these various libraries, Spark provides developers with a platform to perform and build different applications without any hassle and compatibility issues.

The primary abstraction of Spark is Resilient Distributed Datasets (RDD). They were proposed by Zaharia et al. [3] to overcome the weaknesses and shortcomings of current cluster computing frameworks. RDDs allow the reuse of data efficiently in a distributed environment. The authors argue that data reuse is very

common in machine learning and graph algorithms, where the intermediate data is reused in many iterations. In addition to data reusability, RDDs provide more benefits for the programmer. For example, they promise better fault tolerance in case of failures. Furthermore, the level of persistence is controlled by the programmer, as an RDD can be persisted in three different ways:

1. In memory as deserialized Java objects, which is the fastest of the three modes.
2. In memory as serialized data, less efficient in terms of time but a better solution if storage is limited.
3. On disk storage is better used when the RDDs are too big to be kept in RAM, and recomputing them from the start is costly, so they are stored to save the time needed to compute them.

In Spark, two kinds of operations are executed on RDDs, transformations, and actions [7]. Transformations are operations performed on RDDs to produce RDDs. and actions are used to materialize the transformations, compute a value and return it

In Spark, transformations are computed lazily, and spark will keep track of all the instructed transformation until an action triggers the computation. This allows Spark to optimize the transformations to achieve the action in the most efficient way possible. For example, if the programmer pipeline of transformations were map, group by key, then filter, it would be more efficient first to filter, then perform the map and grouping or performing the filtering before the grouping. This decision is made by Spark smartly, depending on the data and the transformation pipeline. Furthermore, unlike Hadoop, which provides two functions - map and reduce-, Spark includes more than 80 operations that give the user the freedom to build their applications easily. Some of the primary transformation used in this work are shown in Table 1.

Table 1. Some of Spark Transformations and Actions

	Method signature	Description
Transformations	flatMap[U](f: (T)⇒Seq[U]): RDD[U]	Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.
	filter(f:(T)⇒Boolean): RDD[T]	Return a new RDD containing only the elements that satisfy a predicate.
	map[U](f:(T)⇒U):RDD[U]	Return a new RDD by applying a function to all elements of this RDD.
	mapPartitions[U](f: (Iterator[T])⇒Iterator[U]): RDD[U]	Return a new RDD by applying a function to each partition of this RDD.
	aggregateByKey[U](zeroValue:U)(seqOp:(U,V)⇒U, combOp:(U, U)⇒U): RDD[(K, U)]	Aggregate the values of each key, using given combine functions and a neutral ”zero value”.
	groupByKey(numPartitions:Int): RDD[(K,Iterable[V])]	Group the values for each key in the RDD into a single sequence.
	join[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, W))]	Return an RDD containing all pairs of elements with matching keys in this and other.
	reduceByKey(f: (V, V)⇒V): RDD[(K, V)]	Merge the values for each key using an associative and commutative reduce function
Actions	count(): Long	Return the number of elements in the RDD.
	collect(): Array[T]	Return an array that contains all of the elements in this RDD.

Method signature	Description
reduce(f: (T, T)⇒T): T	Reduces the elements of this RDD using the specified com-mutative and associative binary operator.
saveAsTextFile(path: String): Unit	Save this RDD as a compressed text file
saveAsObjectFile(path: String): Unit	Save this RDD as a compressed text file

What has been discussed thus far is known as Spark Core, on top of it are four libraries that serve different purposes. The first library is SparkSQL, a library that provides structured data processing. The second library in Spark is MLlib, that adds machine learning functionalities to Spark. Furthermore, real-time data streaming is possible with Spark streaming, and finally, GraphX is a library for graph processing. Having these libraries on top of Spark Core gives the programmer most of what he needs under one platform relieving him from seeking external libraries. Together with Spark core, these libraries define Spark as a unified data analytical distributed system.

2.2 Distributed IR Systems

Elasticsearch and Solr are two existing tools that support distributed information retrieval. In the following section, we will discuss these frameworks in depth and explore any research conducted on them.

2.2.1 Solr

Solr is an enterprise distributed search platform provided by Apache and built on Apache Lucene [8]. Since it was publicly released as an open-source platform in

2006, Solr has now become a reliable platform that provides many out of the box features that are needed by the community. At heart, Solr uses Lucene and enriches it with extra features, as well as; wraps it as a RESTful service accessed over HTTP. Like Lucene, Solr is written in Java.

With Solr users can build an inverted index for a collection and query it to find some matching documents. The collection goes through a series of analyzers and tokenizers to process the documents and produce tokens to prepare the collection for indexing. Moreover, Solr provides a full-text search that supports phrase queries, fuzzy search, wildcards, spell checking, and autocomplete. As for the document matching, Solr uses the vector space model (VSM) and the Boolean model to determine the relevancy of documents.

Initially, Solr did not support distributed search. It was then possible after adding SolrCloud in Solr 4.x and later. SolrCloud made Solr the distributed highly scalable, fault tolerance search platform that we know now. Solr provides the ability to add and remove computing nodes as needed. In addition, the index is replicated across instances to protect the data against nodes' failure. Furthermore, SolrCloud saves the user a lot of trouble by handling how the data is distributed, and other processes like sharding and load balancing.

Solr has five major components. Request Handler that processes all the requests received by Solr. Search Component that defines the logic and the implemented Solr feature and provided by the search. Full-text search features like autocomplete and spell checking need to be registered in the search handler to work. Another component that needs to be registered in the search Component is the Query Parser component; this component maps the user query to instructions that lucence understands. The next component is the similarity component; it defines Lucene

weights and how it scores the documents. Lastly, the Response writer formats the response to the user as different response types have different response writer. The response types could be JSON, XML, CSV, and binary formats.

2.2.2 Elasticsearch

Elastic Search is an open source real-time distributed search engine built on Lucene [9]. It hides the complexity of Lucene indexing and searching mechanisms behind HTTP/JSON APIs. By doing so, the founders of Elasticsearch aimed to scale Lucene beyond its single machine limitation. Unlike Solr, Elasticsearch was built distributed, and it can be scaled horizontally. Other features include high availability by data replication. In Elasticsearch, the index is sharded and replicated across the cluster, and the shards are flagged as primary or secondary. If a node that contains primary shards fails, the secondary shards are promoted to primary.

Moreover, as implied earlier, Elasticsearch is based on REST architecture, where the users are provided with APIs to interact with the service. Also, Elasticsearch provides a JSON interface that enables the users to write queries even if they are not aware of Lucene query syntaxes. This is possible thanks to the Query DSL (domain-specific-language) feature. Lastly, Elasticsearch is schemaless. So, it does not need any fields or data types to be declared beforehand.

2.2.3 Ivory

The closest work to our proposed system is the work done by Lin et al. [10]. The work of the authors revolves around the idea of building a search engine using Hadoop framework. The proposed engine utilizes Hadoop's distributed architecture to build a full search engine that can index and search a large set of documents. The proposed system of the authors employs different design strategies to make the system. For example, they utilized Hadoop ability to automatically sorting the data

while shuffling to minimize the need to sort document ids manually in the reduce phase.

In addition, Lin et al attempted to use the same cluster for both indexing and search rather than having two separate architectures for each task. This approach successfully employed the available hardware and simplified the workflow and management of data; however, it led to a degradation in query processing. For evaluation, clueWeb09 was used for indexing and retrieval. The indexing took 145 minutes on average; meanwhile, the retrieval of relevant web pages for a query took 5.45s.

Aside from the work by Lin et al., Indexing is one of the popular uses of Hadoop, and many algorithms were implemented to index large collections. For instance, McCredie et al. [11] have explored different indexing strategies over MapReduce, including per term and per document. Each strategy has its advantages and disadvantages concerning scalability and efficiency, so it is based on the application and programmer to decide the best way to use MapReduce to benefit from what it has to offer.

2.2.4 Terrier

Terrier is a retrieval platform for the development of large scale retrieval systems [12]. Terrier is built after exploring many efficient and effective search techniques that lead to implementing new methods, such as hyperlink structure analysis, document length normalization methods, query expansion, and reformulation techniques. Moreover, Terrier supports powerful compression methods to enable Terrier the ability to process large collections. Lastly, Terrier was expanded by adding distributed architecture that allowed Terrier to be used in both a single and a distributed environment. Aside from that, an open-source version of Terrier is

available to the public. This version allows the developers to test their IR system in a robust, flexible, and transparent test-bed platform.

The indexing in Terrier has four stages; at each stage, developers can add plugins to customize the process. The four stages are Collection, Documents, Terms pipelines, and indexer. The collection stage is responsible for generating a stream of documents. Then, the document stage will parse each document using the different parsers supported by Terrier to produce a stream of terms. After that, the terms will go through the Term pipelines. Terrier provides some pipelines like Porter's stemming algorithm and stopword removal. New pipelines can be added as plugins if needed. Lastly, the terms after the pipeline are passed to the indexer where it writes the index using the data structures supported by Terrier. Terrier has four main structures: Lexicon, inverted index, document index, and direct index.

As for the retrieval, Terrier is so flexible that it allows the developers to choose from different weighting models, such as, BM25, TF-IDF, and the divergence from randomness framework (DFR) that offers parameter free probabilistic models. Moreover, in Terrier, users can use different defined modifiers to change how the documents are scores. Furthermore, Terrier uses an advanced query language that allows the user to define additional operations, as well as automatic query expansion.

Recently, the team behind Terrier released a new adaptation of Terrier using Spark [13]. This adaptation is built to make use of Spark pipeline feature and Spark machine learning MLlib fit and transformation. Initially, these functions are used to fit and train models in a machine learning setting.; however, in Terrier-Spark, the author builds on Spark MLlib to define his own version of fit and transformation. This adaptation overcame Terrier command line limitation.

Furthermore, Terrier-spark offers an agile experimental platform for IR, and using Terrier-Spark with Jupyter notebooks will make reproducibility of the experiments a lot easier by sharing the notebook, as well as aids at teaching information retrieval experiments in universities if needed.

2.3 Spark in IR

Ma et al. [14] use spark to build a media retrieval system. They utilize Search tress and caching the index in memory to achieve fast and cost effective retrieval. The system reduced the time of search; also, their system allows the user to choose the weights they want to increase the importance of a feature or more to customize their ranking results. The system is built with four components. The input for a system is a feature file where each file contains many instances of images.The first component partitions these feature files into an equal sized feature sets called chunks. Each feature file results in N or multiple of N chunks where N is the number of compute nodes.

The second component builds a local index from its chunks on each node. These indexes are saved in HDFS. The third component is responsible for retrieving the top k results for each feature group. Then the last component merges the top k results based on the weights set by the user to generate the final results. The authors use a tree structure index to increase the similarity matching processes within the nodes. Then the top hits are aggregated to the driver node to find the top of the top matches found on each node. The authors employ different transformations from Spark, more notably `MapPartitions` to work on a per-partition fashion and `ReduceBykey`.

This work has many things in common with our proposed solution, as both are retrieval systems built on Spark. However, our work is focused on Text and webpages

rather than images. Furthermore, this work also uses the per-partition approach that we are aiming to utilize in our work.

2.4 Spark in the Literature

Spark has been gaining much attention from researchers because of its speed and support for iterative algorithms and data reusability. The uses of Spark in the literature varied as some researchers used it for machine learning while others used it for graph processing. In this section, we explore how spark is used in different fields to build efficient and scalable systems.

Koliopoulos et al. [15] proposed extending the WEKA tool to be a distributed tool. WEKA [16] is a data mining tool that is widely used by data scientists. However, the authors argue that WEKA only supports single-node execution; thus, it is not suitable for Big data. Therefore, the authors propose to leverage from Spark to design a distributed version of WEKA called "DistributedWekaSpark." The proposed extends the WEKA framework while making use of Spark RDDs and their operations to make WEKA's processes run on a distributed environment rather than a single node allowing WEKA to be able to handle Big Data.

DNA sequences are another example of big data. NGS technology (Next generation Sequencing) is a technology used in the health field on DNA sequences that generates data in order of hundreds of gigabytes per experiment. This data has to be analyzed as quickly as possible to extract meaningful results from it. Thus, in [17] the authors propose a pipeline for DNA analysis using SPARK. SparkGA, the system proposed by the authors utilizes spark capabilities to produce a system that is scalable and supports parallel analysis. The proposed system managed to be 71% faster than the current state-of-the-art technology. Moreover, the authors experimented heavily

to reduce the cost of analysis while maintaining an impressive accuracy of 99.9981% by using a load balancer to optimize the solution memory requirements.

Han et al. [18] proposed turning DBCAN - a clustering algorithm - into an algorithm that can support parallelism using Spark. They were the first to use Spark; although, other implementations existed on Hadoop. One of the challenges they faced was how to design a scalable DBCAN that avoids shuffling since it is expensive as data is being written to disk and moved across the network. In Spark, results from each worker are propagated and merged in a module called Spark driver. Therefore, they assign each worker to create partial clusters locally for the data assigned to it; then, the partial clusters will be propagated to the Spark driver, where the merging happens between the partial clusters.

This approach is also adapted by [14], as the results are aggregated per node and then propagated to the driver node. As we can see, both systems are trying to avoid shuffling as it is very expensive. Similar to these solutions, our proposed architecture takes into consideration the shuffling issue, and we have employed Spark transformations to avoid it as much as we can.

Similarly to [18], Yang et al. [19] enhanced ZhihuRank, a topic sensitive expert finding algorithm that is based on Latent Dirichlet Allocation (LDA) and PageRank, by employing Spark MLlib's LDA and GraphX PageRank and made it a scalable method. The road to achieving this goal was not straightforward, as the operations in LDA and PageRank are of a high-order nature and require intensive and time-consuming computations.

Given that Databases by nature hold a vast amount of data, a system such as Spark could also demonstrate its potential in such environments. Using SparkSQL,

Sun et al. [20] built an in-memory distributed query processing system that achieved fast and efficient results. For Spark Streaming component, Chen et al. [21] built a distributed rule engine by converting a rule-based system and adapting it in Spark environment in addition to real-time streaming using Spark Stream.

Chapter 3 : SparkIR Architecture

The goal of this work is to build an efficient and scalable distributed search engine using Spark. In this chapter, we will discuss the four phases of the proposed system and the different decisions to consider when building such a system. The four phases are: preprocessing, pre-indexing, indexing, and retrieval.

The first phase of the proposed solution is preprocessing in which the data is cleaned and tokenized for the pre-indexing phase. In the pre-indexing phase, global statistics about the collection are generated. The statistics include document and term statistics. Next, the indexing phase will build an inverted index for the preprocessed documents. The created index could either be a document-based index or a term-based index. After that, the index is ready to be queried in the search phase using the provided API. The following subsections explain each phase in depth.

3.1 Preprocessing

The goal of the preprocessing stage is to clean the webpages from HTML tags and unnecessary characters to prepare them for indexing. The preprocessing component takes webpages as input. After that, each webpage will go through a pipeline to clean and extract the text from it. The preprocessing pipeline, shown in Figure 4, starts with removing the HTML tags; then, lowercasing the characters. After that, it removes digits, punctuation, and non-English characters. Then, the stopwords will be removed after tokenizing the text. Finally, the tokens will be stemmed using Porter stemmer.

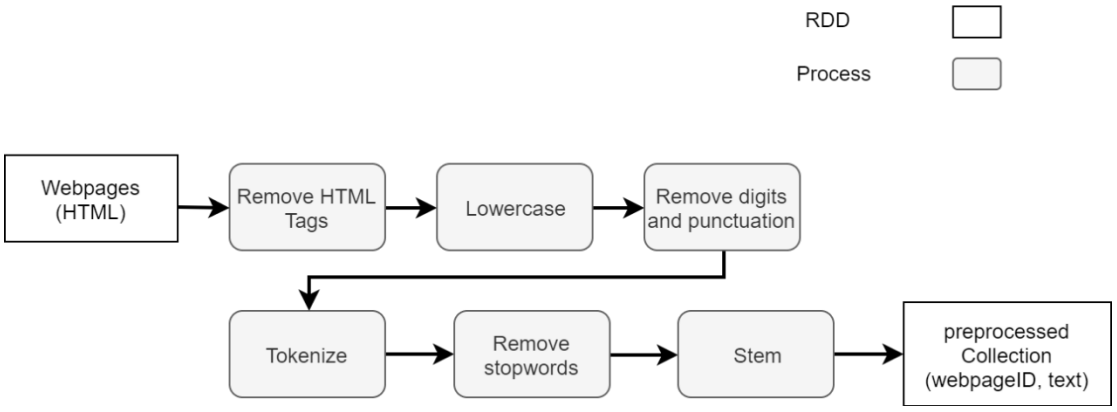

Figure 4. Preprocessing pipeline

Once the pipeline is completed, the preprocessed tokens are concatenated into a single string of text. We opt to save the content of the webpages in the preprocessed collection as a single string rather than tokens to save space when the collection is persisted in memory. So, the output of this stage is a key-value pair RDD where the key is the webpageID of the webpage and the value preprocessed content. From this point forward, the webpages will be referred to as documents.

3.2 Pre-indexing

Before building the index, we compute term and document statistics needed for scoring and reducing the size of the index. Figure 5 presents the steps of this stage that we refer to as the pre-indexing stage.

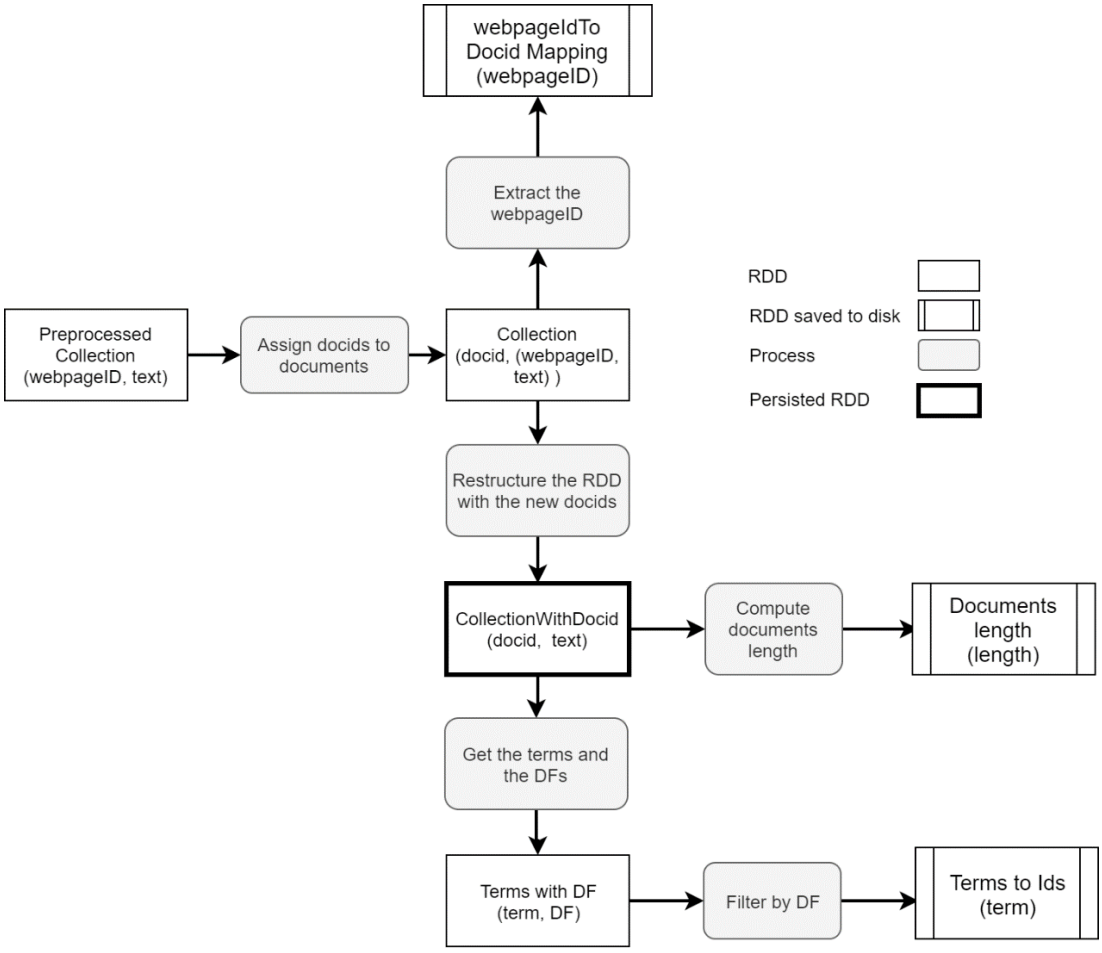

Figure 5. Pre-indexing stage

The process starts with assigning a sequence number - starting from 0 to N, where N is the size for the collection - to each document. Initially, the webpages are identified with a string ID; therefore, we assign an integer id, called docid, to each document to use in the postings to reduce the size of the index as much as possible. Then we save this mapping to disk as a key-value pair <docid, webpageId>.

As shown in the figure, a new RDD *CollectionWithDocid* is generated and persisted in memory. In the new RDD, the sequence of the docids will be used to

identify the documents throughout the system. Also, this RDD is the parent RDD for the later RDDs; thus, it is persisted in memory to avoid recomputing it.

Similarly, in our system, we map the terms to integer ids, called termids, to further reduce the size of the index. To create this mapping, first, we make a pass over the collection to get the unique terms with their document frequencies (*df*) and sort them. When the terms are sorted in alphabetical order, later when this mapping is read from disk into an array, the ids of the terms will match the index of an array; therefore, we only need to store the list of terms without the ids.

After sorting, we filter terms based on their *df*. The *df* of a term indicates how many documents contain this term or simply how popular the term is in the collection. We use *df* to filter out terms with very low *df* or high *df*. The values for the minimum *df* and maximum *df* are specified by the user in a configuration file with other parameters. We remove these terms because if they have low *df* then they are probably typos, while high *df* means they frequently appear like stopwords, so removing these terms will not have a significant effect on the documents' scores. Finally, we compute the document lengths from the collection; likewise, this mapping is stored on disk without the docids.

3.3 Indexing

The indexing component of the proposed solution is responsible for building the inverted index of the collection. The index can be distributed over a cluster of nodes using two different schemes: document-based partitioning or term-based partitioning. In document-based partitioning, each node has a local index for a subset of the collection. In this case, each node has an independent index. In term-based partitioning, each node has an index of a subset of terms of the collection [22]. Figure

6 illustrates the difference between the two schemes, and how a collection of nine documents would be partitioned across the nodes in each scheme.

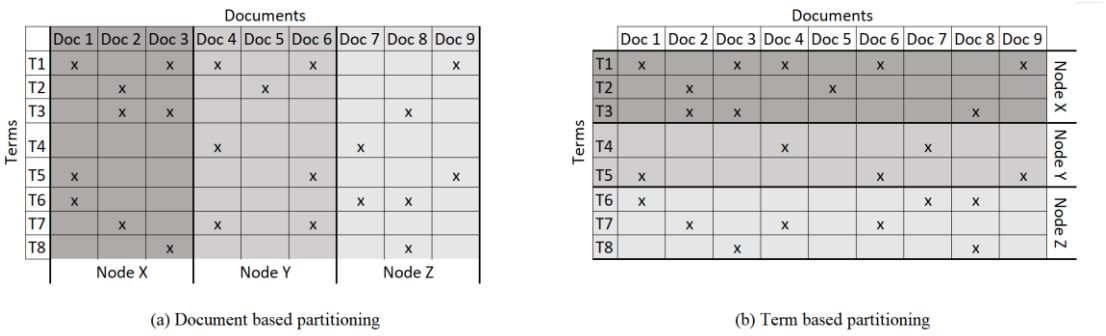

Figure 6. Index partitioning schemas

In the literature, document-based partitioning is preferred because it provides more benefits over term-based partitioning [23]. With document-based partitioning, the system can scale better with the increase of collection size and nodes. Moreover, fault tolerance is handled better in document-based partitioning because the quality of the results is hardly affected by the failure of some nodes. SparkIR supports both document-based and term-based partitioning schemes, and both schemes are possible thanks to Spark transformation and partitioning handling.

3.3.1 Document partitioning

In document-based partitioning, the documents are indexed per partition; therefore, using Spark, we must not shuffle the RDD of the collection and work on a per partition fashion. Figure 7 shows an overview of the document partitioning process, the RDDs created, and the dependencies between RDDs.

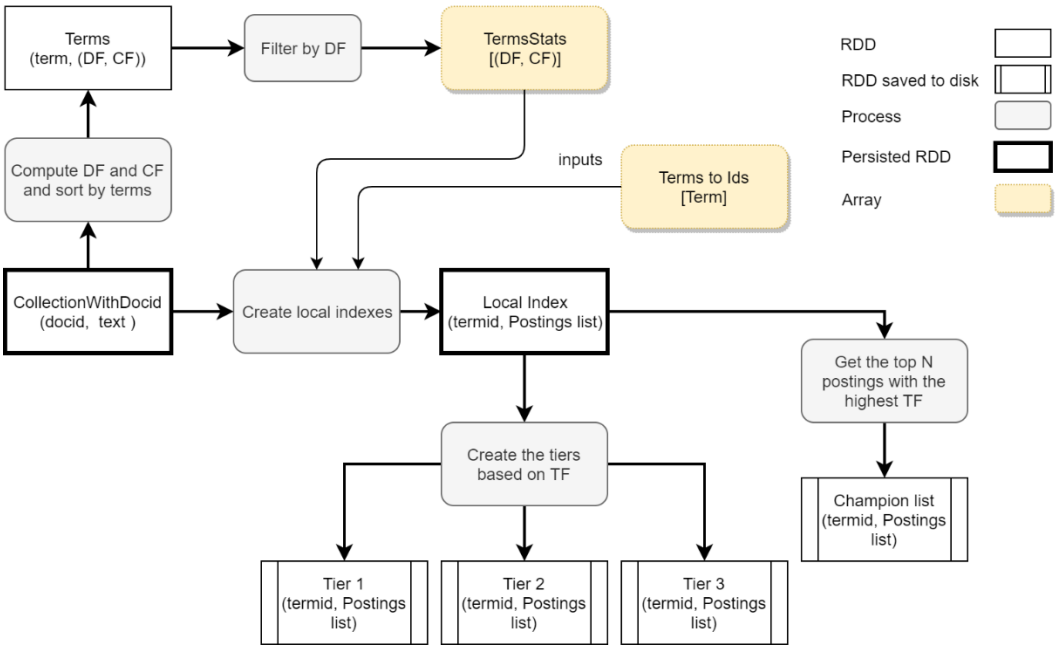

Figure 7. Document partitioning indexing process

The indexing starts immediately after computing the global statistics. All transformations are performed on the <docid, text> RDD that contains the preprocessed collection. First, we start creating the local index using `mapPartition`. `MapPartition` is a transformation that allows working per partition; thus, avoiding any shuffling through the network. Using this transformation, we create an inverted index in two steps. First, we get the tokens and their associated frequencies, then we create an index locally on each partition.

As shown in Algorithm 1, we start by tokenizing each document. For each token, we create a pair where the key is the term, and the value is one; then, we group on terms. After grouping, the output represents each term with its term frequency (*tf*) and docid. Term Frequency indicates how many times a term appeared within a

document. Once we grouped the terms, we map each term to its id using the terms to ids array we created in the pre-indexing phase. This step – lines 3 to 7 – is performed on all documents within the same partition. Once done, the output will be a list of pairs that has <term, <docid, *tf*>> from all documents within the partition.

Algorithm 1: Document partitioning Algorithm

Input: *collection*: an RDD of tuples(docid, tf), *termsToIds*: in alphabetically ordered array of terms; *termsStats:* an array of tuples (cf,df) ordered by termid, *output*: the output path of the index

```
1  termsPerDocument ⟵ collection.mapPartitions(          ▷ for each
   document get the terms and their tf
2    documents => {
3      documents.flatMap{ document =>{
4        document.text.split.groupby(term)
5          .mapValues(values => (document.docid, values.length))
6          .map(term => termsToIds[term])     ▷ map the terms to
           thier ids
7          .filter (termid != −1)        ▷ remove terms that arent
           found in the array
8        }
9      }
10   )
11
12   localIndex ⟵ termsPerDocument.groupby(termid)        ▷ group
     similar terms in the same partition
13     .mapValues(values => {      ▷ for each value of the terms
       compute the tf, df, cf and store them in an object
       postingList
14
15       docid[] ⟵ values.docid
16       tf [] ⟵ values.tf
17       df ⟵ termsStats[termid].df
18       cf ⟵ termsStats[termid].cf
19       postingslist(cf, df, docid, tf)
20     }
21
```

After that, we group the terms and create the index. In line 12, when we group by terms, we get a map of terms as keys and a list of values that are the pair < docid, *tf*>. Then, we convert the list of pairs of a term into two arrays; an array that holds the *tf*s and another to store the term's postings. We uses different arrays for *tf* and docids to reduce the size of the index. In Scala, each pair is known as a tuple, and a tuple is an object. After grouping the pairs by terms, we produce a list of tuples that share the same term. The length of this list might be long, and each element of this list is an object. Eventually, we will load the index to memory in the retrieval phase; so, we choose to use arrays to reduce the size of the index and object creation overhead.

Aside from the docid and *tf* arrays, we also store the collection frequency (*cf*) and document frequency (*df*) for each term. *cf* counts how many times a term appears in the whole collection, and *df* – as explained earlier – shows how many documents has the term. These two values are usually computed from the collection and used while creating the index; however, it does not work in our case.

If we compute them while building the index, we will face two issues. If we compute them after building the local index, the *df* and *cf* values will be local to the partition and do not reflect the true value of *df* and *cf* for the term across the collection. Moreover, global *df* and *cf* cannot be calculated easily across partitions. Also, before creating the index, the collection was already partitioned by docid; then, each local index was built independently of the other partitions. So, in each partition, the RDD creation starts with a pair RDD < docid, text> and ends with a pair RDD <Term, Postings list>. As a result, the original key that partitioned the data – the docid – is lost, and the new key term does not contribute to locating its partition. Usually, in Spark joins are used to combine RDDs with similar information. However, if we tried to join a Pair RDD <Term, <*df*, *cf*>> with the index, it will trigger a shuffling that will

shuffle the data based on the term; which will lead to losing the document partitioning of the index.

To solve this problem, we computed the *cf* and *df* a step before building the index using the collection. We hold the *cf* and *df* of each term in a key-value pair sorted by the term id. Then we broadcast this RDD as an array of <*df, cf*>, where the index of the array is the term id. This array is broadcasted to the workers and used while indexing the retrieve the *cf* and *df* for a term. This step is not part of the pre-indexing phase because it is only needed in document-based partitioning. As we will see later, this array will not be necessary for the term based partitioning.

The index created so far, is an index of the entire collection without pruning. To enhance the efficiency of the retrieval in the proposed system, we added two efficiency techniques that the user can select to use in the indexing stage. The user can choose one of the following static pruning techniques: champion list or tiering.

Champion List. The champion list is a static index pruning technique where the postings list is pruned during the indexing phase [24]. The indexer will take the top *n* postings list of each term according to their *tf* and prune the rest of the postings. The champion list aims to focus on the top contributing documents of a term rather than the full list. Although it does decrease the number of scored postings, this technique might harm the effectiveness of the system.

Tiering. On the other hand, tiering will divide the postings list into tiers [23]. The criteria can be *tf*, inverse document frequency (*idf*), or any weight according to the user. In our system, we use the *tf* to divide the postings list into three tiers given the *tf* cut-off values for each tier by the user. When tiering is used, the 1st tier is considered the main index, and the other tiers are only visited if the results retrieved

from the 1st tier does not reach the number of results (*k*) the user specifies. With tiering, we reduce the size of the index loaded into the memory by using only the highest contributing documents to avoid scoring documents that might not be in the final results for that term.

3.3.2 Term partitioning

In term-based partitioning, all postings of the same term must be in the same partition. As a consequence, the data will be shuffled across the network, so we need to minimize the size of the shuffled data to build the index efficiently and reduce the side effects of the shuffling process. Figure 8 shows the process of building a term partitioned index. Similar to document-based partitioning, the input to the term-based partitioning indexer is the preprocessed collection from the preprocessing stage. To create the index, we produce a key-value pair <Term, <docid, *tf*>> for each document. Then, we group similar terms across the partitions. Spark offers many transformations to accomplish this goal, such as `groupByKey`, `combineByKey`, `reduceByKey`, and `aggregateByKey`. Each transformation has different advantages and disadvantages. For our system, we choose `aggregateByKey`.

This transformation combines similar keys efficiently by aggregating the results within the partition first to create an intermediate result. Then, it will combine similar keys of the intermediate results across the partitions. This approach will reduce the amount of data shuffled across the network. Whereas `groupByKey`, for example, will combine on the cluster level by shuffling all the key-value pairs, so similar keys end up on the same partition. However, both transformations will output a key-value pair, where the value is a list of values that share the same key. The transformation signature was mentioned shown earlier in Table 1. For more details,

refer to lines 1-16 in Algorithm 2 to see the functions `aggregateByKey` uses to achieve its goal. The first parameter *initlalList* is an empty postings list that holds *df*, *cf*, and the docid and the *tf* arrays. `AddToList` is the function used to aggregate the postings list within the partition. Each pair is being added to the postings list of a term, and the values of the postings list are updated accordingly. Then, `mergePartitionList` decides how postings list across partitions are merged and updated.

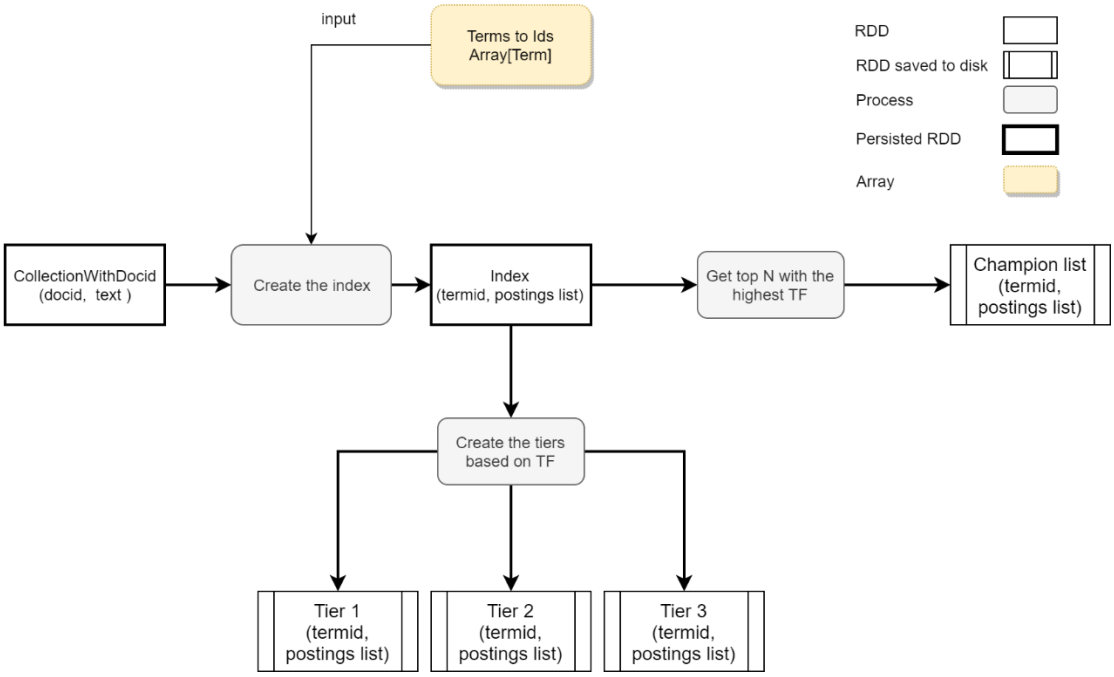

Figure 8. Term partitioning indexing process.

The structure of the index RDD created in term-based partitioning is the same as document-based partitioning; therefore, the same design decisions made earlier are also applicable here. The only information we need from the pre-indexing stage is the

term to id mapping. Other term statistics; such as *cf* and *df* are computed directly after creating the index. Therefore, we do not need to compute them from the collection like the document-based partitioned index. In other words, the length of the postings list is the *df*, while the sum of *tf*s is the *cf*. Lastly, both the champion list and tiering can be used during indexing if the user chooses to use them, and both are built similarly to how they are built-in document-based partitioning.

Algorithm 2: Term partitioning Algorithm

Input: *collection*: an RDD of tuples(docid, tf), *termsToIds*: in alphabetically ordered array of terms, *output*: the output path of the index

```
1  initialList ⟵ PostingsList(0, 0, Array[Int], Array[Int])      ▷ the value
     used for aggregateByKey
2  addToList ⟵ (s: PostingsList, v: (Int, Int)) => {
       ▷ Add a new tuple to the postings list in a single partition
3      s.cf ⟵ s.cf + v._2
4      s.df ⟵ s.df + 1
5      s.docid.+=(v._1)
6      s.tf.+=(v._2)
7      s
8  }
9  mergePartitionList ⟵ (p1: PostingsList, p2: PostingsList) => {
                    ▷ How to merge 2 postings list across partitions
10     p1.cf ⟵ p1.cf + p2.cf
11     p1.df ⟵ p1.df + p2.df
12     p1.docid.++=(p2.docid)
13     p1.tf ++= p2.tf
14     sort(p1.docid, p1.tf)
15     p1
16 }
17 index ⟵ collection.flatMap(                          ▷ building the index
18     document => {
19         document.text.split
20             .map(word =>{ ((word, document.docid),1) })
21         })
22         .reduceByKey(_+_)
23         .map( ((word, docid),count) => (word, (docid,count)) )
24         .aggregateByKey (initialList)(addToList, mergePartitionLists)
25         .map(term => termsToIds[term])      ▷ map the terms to thier
             ids
26         .filter (termid != −1)      ▷ remove terms that arent found in
             the array
27         index.SaveAsObject(output)
```

3.4 Retrieval

In the retrieval phase, the user can search the index by loading the index in memory, issuing queries, and receiving the results efficiently. Both efficiency and effectiveness are important in the retrieval stage; therefore, we focused on exploiting Spark efficiency to support our system. To meet this goal, we employ both static and dynamic pruning techniques to enhance the retrieval process.

Figure 9 shows the process of retrieval in the proposed system. First, the user provides the path to the index and its related files saved in the indexing stage, such as the global statistics, and the mappings of webpageIDs and terms. As mentioned in chapter 2, Spark computes RDDs lazily; so, it will not load the RDD until the first query is issued. To avoid reissuing queries, we force the index to be loaded using the `count()` action. As for the global statistics, they are read into arrays and broadcasted to the workers, except for the webpageIDs mapping. By broadcasting these arrays, each worker will have a copy of this data locally, and it will not need for the driver to ship them each time they are needed. As for the webpageID to docid mapping, we read it into an RDD. Unlike the previous arrays that hold int numbers, this array will have String objects. Therefore, as the collection grows, the size of this mapping will grow larger into GBs. In an effort to avoid storing all this mapping into an executor, we save it as an RDD to distribute it in the cluster.

Preprocessing. When the user issues a query, the query will be preprocessed with the same pipeline used for the documents. Then the processed query tokens are mapped to their ids from the terms to ids array using binary search, which will return the index of the term if it is found, and -1 if it is not. Recall that terms to ids mapping has initially been sorted alphabetically; so, the ids will be the indices of the terms in

the array. Then, a map that holds the termId and the *tf* of each term is broadcasted to the workers, so each worker has a copy of the query.

Query evaluation. The next stage is query evaluation, where the documents are scored. The query evaluation process depends on the schema the user chooses in the indexing. Depending on the partitioning, the query evaluation can be done document at a time (DAAT) or term at a time (TAAT). In DAAT, the postings lists of the query terms are processed in parallel by advancing one docid at a time; thus, assigning a score for each docid as it processed. This approach is suitable for a document-based partitioning because all the document terms are within the same partition; so, a full score of a document can be computed within a partition. On the other hand, TAAT is more suitable for term-based partitioning, where it processes each query term sequentially, going through its postings list and keeping track of the partial scores of the docids in accumulators.

Scoring. When a map of query terms reaches the worker, it will first load and persist the index to memory if it was the first query; then, it will filter the query terms from the index to get their postings list (Line 11 in Algorithm 3). A filter transformation in Spark does not trigger shuffling. So, the system works per partition to get the scores by performing either a TAAT or DAAT. The score of a document is computed using BM25. Within each partition, the system keeps track of the top *k* scores. Once the scoring of a partition is completed, the results are collected at the driver node – the master node in Spark that is responsible for communicating with the cluster nodes and collecting the results – where the top *k* of the local top *k* from the partitions is collected and the docid is replaced with the webpageID. Since the webageID is an RDD with docid as the key, we efficiently join this RDD with the

collected results using map side join by broadcasting the results to the workers and joining with docid as key. After that, the results are presented to the user (Lines 28-31).

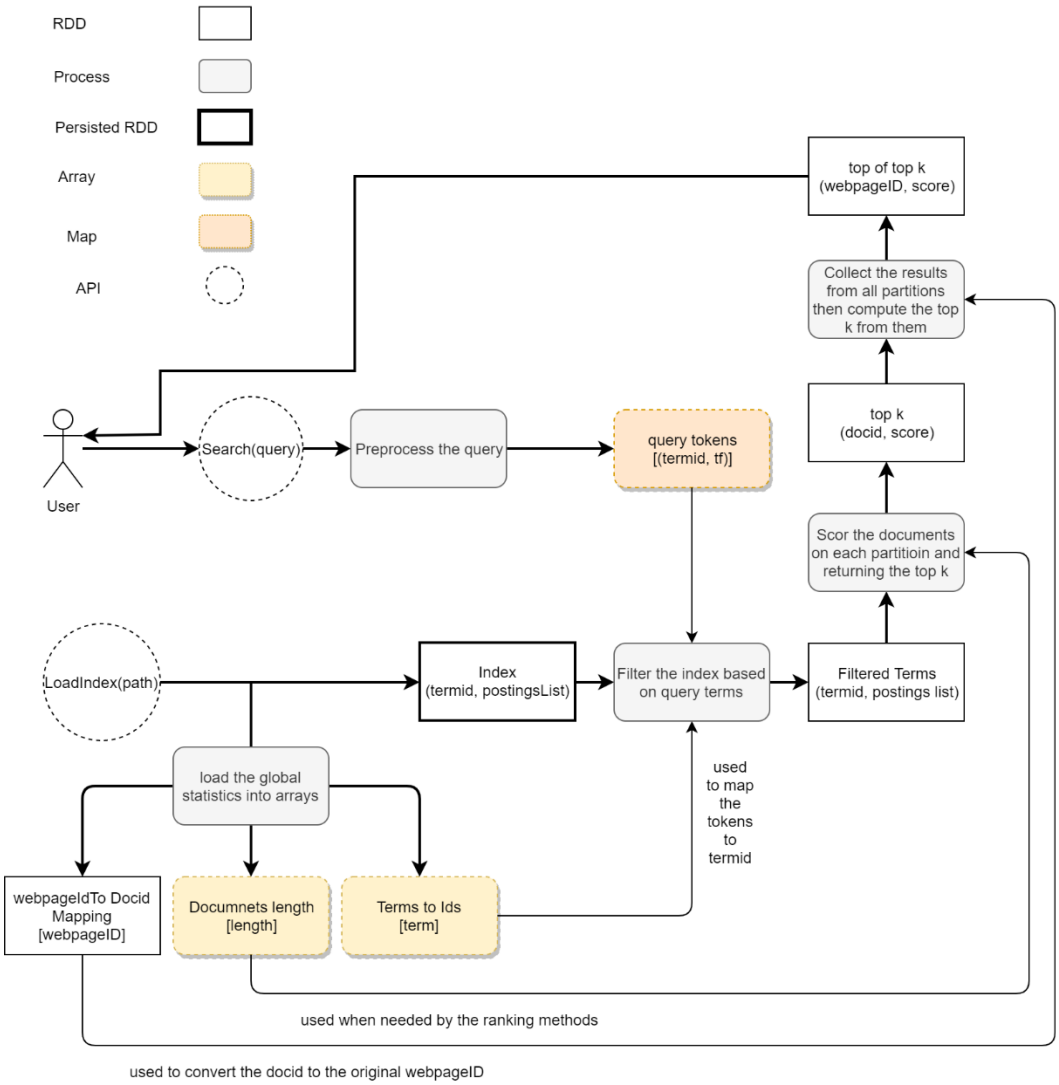

Figure 9. Retrieval process.

Algorithm 3: Search Algorithm

Input: *sc*: SparkContext used to execute transformations and actions,
query: query to run against the index
Output: top k documents as a tuple (webpageId, score)

```
queryTerms ← Map[Int, Int]()          ▷ a Map to hold term id and tf
if query.split.length > 1 then
    preprocess(query) .split.foreach(term => termId ← getTermId(term))
     if termId != -1 then
        queryTerm.add(termId,tf)
else
    queryTerm.update(query)
end
broadcastQueryTerms ← sc.broadcast(queryTerms)
terms ← index.filter(entry =>
 broadcastQueryTerms.value.contains(entry.termid)).persist()
local_hits ← terms.mapPartitions(iter => {
    termCont[] ← getTermContribution(iter.map(term => term.tfs.max))
     docids[] ← iter.map(t => t.docids)
    tfs[]← iter.map(t => t.tfs)
    dfs[] ← iter.map(t => t.df)
    cfs[] ← iter.map(t => t.cf)
    if (DAAT) then
        if maxscore then
            sort(termCont, tfs, docids, dfs, cfs)    ▷ sort all the arrays
             based on term contributions
            maxScore(docids, tfs, termCont, dfs, cfs, k)
        else
            documentAtATime(docids, tfs, dfs, cfs, k)
        end
                                  ▷ If the query evaluation is TAAT
    else
        termAtATime(docids, tfs, dfs, cfs, k)
    end
    }
all_hits ← local_hits.collect()
q ← PriorityQueue[(Int, Float)](k)
for each hit in all_hits do
    q.enqueue(hit)
mapToWebPageID(q)              ▷ mapping the docids to webpageID
```

3.4.1 Dynamic Pruning

Although these query processing techniques are state of the art, they are not efficient. For example, TAAT consumes memory as it must keep track of seen docids so far. Whereas DAAT is slower compared to TAAT [24]. Therefore, in our proposed system, we add efficiency methods aiming to enhance the retrieval phase. In addition

to tiering and the champion list used in the indexing stage, we included MaxScore in the retrieval.

MaxScore is a dynamic pruning algorithm that skips postings as the evaluation is ongoing [26]. By adding MaxScore, we can reduce the time needed for scoring the postings list.

The main point of MaxScore is to make use of terms contribution to compute an upper bound for each postings list of the query term. While scoring a document, the sum of the current document score and the contribution of the next unseen terms is used to decide if the document should be skipped or not. A threshold – which is the lowest score of a document in the current top k – is used to determine if a document can be considered a candidate for the final results list. If the sum of the current score and the contribution of the unseen terms exceeds the threshold, it means the document is a candidate for the final top k, but if it fails to surpass the threshold, then the document is skipped from all terms as there is no expectation for it to be in the top scoring documents.

In a distributed environment, it was challenging to implement MaxScore for term-based partitioning because some information like the current top k has to be known between nodes, and in Spark, it is not possible to communicate within nodes with a variable that allows both reading and writing. Therefore, TAAT implementation does not include MaxScore. However, in document-based partitioning, each partition has a local index; so, we can implement MaxScore per partition and then propagate the final results to the driver to get the final ranking.

3.5 Implementation Issues

3.5.1 API

The created library has two main APIs, an indexing API and a retrieval API. For the Indexing API, the available methods include:

- `setPartitioningSchema(partitionSchema, pathToConfigurationFile)`: to set it to either document based partitioning or term based partitioning

- `createIndex(inputPath, OutputPath)`: create the index as shown in the previous section

As for the retrieval API, the user has access to the search method to submit the queries.

- `loadIndex(indexPath, SparkContext)`: to load the index and the related global statistics into memory

- `search (query, SparkContext)`: to search for a single query and return the results to the user.

- `batchSearch(queries[], SparkContext, output)`: given a list of queries and an output path, this method writes the results of the queries in the output path.

3.5.2 Required Configurations

To run SparkIR the user has to include a configuration file that the user has to include. The configuration file has the different parameters the system depends on in both indexing and retrieval. These configurations include:

- tier-ubound and tier-lbound: the tiers *tf* cutoff values.

- number-of-partition: number of partitions of the index
- min-df: minimum value of *df*
- max-df: maximum value of *df*
- no-prunning: A flag that indicates if the index is with or without pruning.
- champion-list: A flag that indicates if the champion list technique is selected or not
- n: The size of the champion list
- tiering: A flag that indicates if tiering is included or not
- ranking-method: The ranking method
- maxscore: A flag that indicates if maxscore should be implemented or not
- k: The size of the results returned to the user
- query-evaluation: The query evaluation technique - DAAT or TAAT - .

3.6: Experimental Evaluation

In this chapter, we will evaluate the performance of the proposed system. This chapter is divided into two sections. In the first section, we discuss the experimental setup where we disclose the dataset and the services used for evaluation. The second section shows and discusses the results of the different stages of the system, and how the system performs against the baseline.

3.7 Experimental Setup

3.7.1 Dataset

The dataset used is a subset of ClueWeb12[3]. ClueWeb12 contains about 733 million English web pages collected from February to May of 2012. Carnegie Mellon University was responsible for collecting this dataset and distributing it. It took four months of crawling to collect the dataset. Once the crawling was over, the data was cleaned by removing non-English content, and other pages or websites that are not usable as part of a research dataset. The uncompressed size of the dataset is 27.3 TB. ClueWeb12-B13 is a 7% sample that was created by taking the 14th document from each file of the full dataset. The number of documents in this collection is 52,343,021. The collection was distributed on HDFS with three replications as compressed files. This sample is the dataset we use for our evaluations. Therefore, from this point forward, 'dataset' refers to ClueWeb12-B13 unless otherwise stated.

In the upcoming sections, we evaluate the scalability and response time of the SparkIR architecture. Another important aspect is to make sure that the effectiveness of the proposed system does not fall when we perform the efficiency methods.

[3] https://lemurproject.org/clueweb12/

The queries used for this experiment are from the TREC2013 and TREC2014 web track. Each set has 50 different queries of varying lengths. Figure 10 shows the length distribution of the 100 queries. These queries will be used for testing the effectiveness and efficiency of the proposed system.

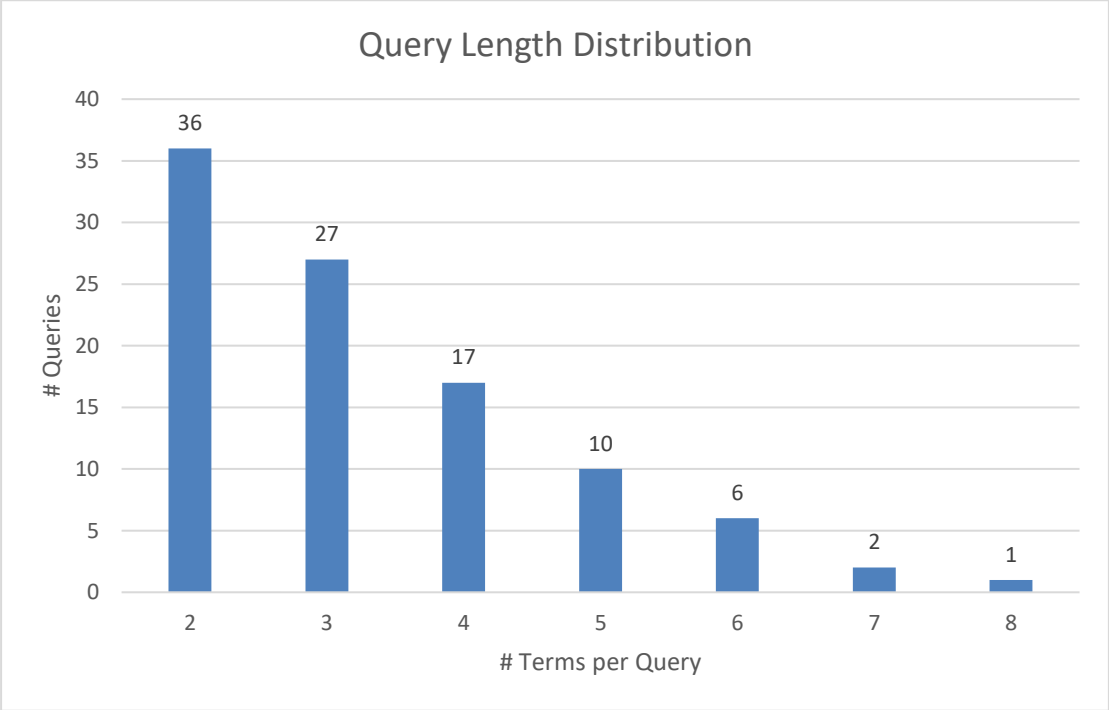

Figure 10. Query length distribution.

3.7.2 Evaluation environment

We test the system on a three-server cluster. Each server has 120 cores and 128 GB memory. However, only 100 GB is configured for Hadoop on each node. The

cluster has Hadoop 2.6.0 and Spark 1.6.0 installed. All the below experiments were run on this cluster using Yarn as the resource manager.

In both indexing and retrieval, we use the same number of executors and memory per executor. Empirically, we fixed the memory per executor to 22 GB and the number of cores to three cores per executor. By default, Spark assigns 128 MB for java heap overhead, a memory portion added to an executor to avoid Java heap errors. We added about 3 GB for each executor as an overhead; therefore, in total, each executor has about 25 GB reserved for its use. Aside from the executor, we also need to specify the memory size of the driver and its number of cores. Since most of the work is done on the executors, the driver does not need to have much memory, but it needs enough in case we collect data to it. Thus, we assigned 5 GB for the driver, and the number of cores is one which is the default value assigned by Spark. More details on how and why we choose these numbers will be mentioned in the subsequent sections.

3.7.3 Performance measures

Since the main field this work tackles is Big data, high efficiency is essential for SparkIR. Therefore, we will be focusing on time in both the indexing and retrieval stages. In the indexing phase, we will evaluate how fast the system indexes a collection as it grows. As for the retrieval, we will conduct experiments to measure the response time of the system. For both stages, we will test how to fine-tune Spark to enhance its performance. However, this does not mean we will neglect the effectiveness, as we need our system to be efficient without harming its effectiveness.

3.8 Indexing Performance

In this section, we evaluate the time it takes for SparkIR to build the index. Aside from the number of executors and cores assigned for Spark, the number of partitions is the only parameter in SparkIR that affects the indexing time. In Spark, it is recommended to either have the number of partitions to be 2 or 3 times the number of CPUs or by dividing the collection size on the block size [27]. For our system, we decided to choose the number of partitions based on the size of the data. We split the data into six different sizes; the smallest data size has about 7 million documents, while the largest size is the full 52 million documents collection.

As mentioned earlier, the other parameters that affect the indexing are Spark parameters. The speed of execution in Spark is greatly affected by the number of executors and cores. The more executors and cores per executor we employ, the faster the job will be if we assigned enough memory to support the job execution. However, these two parameters are limited by the available resources and the size of the data. For a fair comparison, we fixed the number of cores and the size of memory to accommodate the largest size of the dataset. We choose the memory per executor to be 22 GB, this forced Spark to create 11 executors with about four executors on two nodes while the last node has three executors and the driver. As for the cores, we set it to three cores per node. We choose these values empirically by indexing the full collection and choose the configuration that led to the best results. However, we change the number of partitions according to the size of the data. Moreover, the driver needs enough memory to collect the mapping RDDs that are needed for retrieval. Similarly, we tried different driver size starting from the default value of 1 GB until we reached 5 GB where the RDDs were collected without causing an out of memory error in the driver.

3.8.1 Preprocessing

In our system, preprocessing can either be pipelined along with the indexing stage or ran as a separate stage where the intermediate preprocessed collection is saved to disk. For the experiment, we opt to save the intermediate results on disk. Then we based the number of partitions on the size of the dataset after preprocessing. Table 2 shows the rounded partition number for each dataset. We round up the number partitions to a few more than the original number given by division to reduce the chances of out of memory problems.

Table 2. The size and Number of Partitions for each dataset

No. of documents in millions	7	15	23	31	36	51
Size in GB	20.3	39.5	63.1	85.6	101.8	151.9
No. of partitions	150	310	500	700	900	1200

Since the preprocessing is shared between the document and term partitioning indexing, we excluded that time from the indexing. It is also worth noting that the reported time for preprocessing also includes writing to disk as SparkIR will execute the preprocessing transformation pipeline after calling the action `saveAsObjectFile`. Thus, it is difficult to measure the real-time taken by

preprocessing. Figure 11 shows the time taken by preprocessing for the different collection sizes.

The preprocessing is approximately linear in the size of the data and as the size of the collection increases, the longer it takes to preprocess the collection. It is also worth noting that the number of partitions affects the speed of processing significantly. Naturally, more partitions mean more tasks to schedule; therefore, the longer it takes to finish a job. On the other hand, with fewer partitions, we will need enough memory to fit the partitions in the executor storage memory and save some memory for computation.

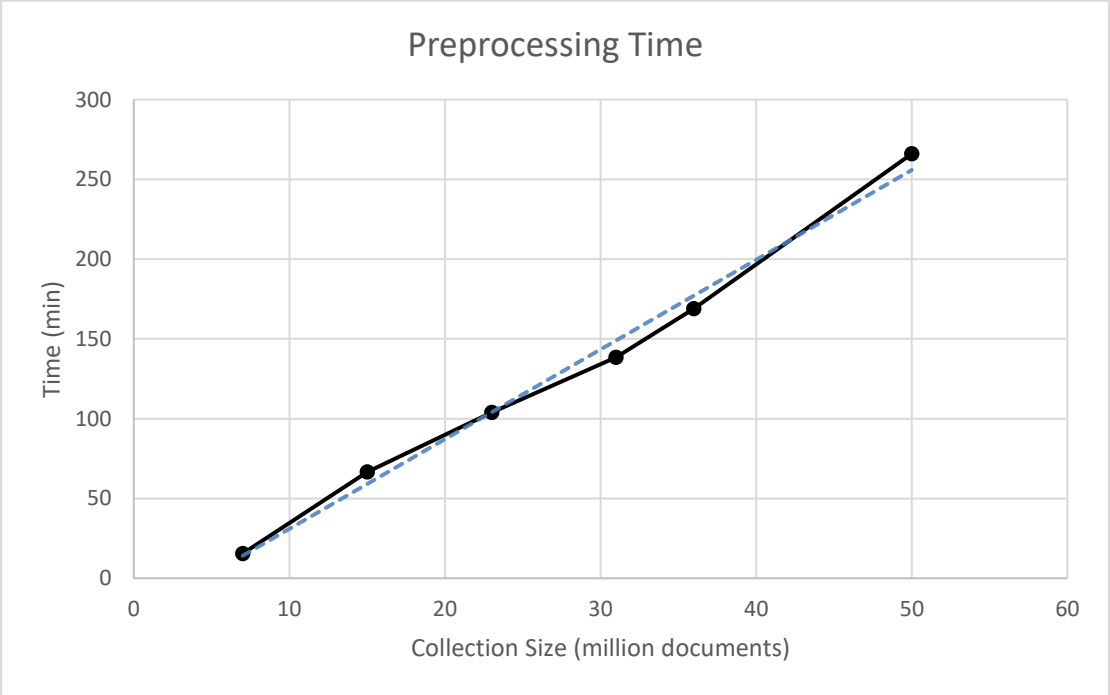

Figure 11. Preprocessing time across different dataset sizes

3.8.2 Indexing

After preprocessing, we indexed the different sizes using both document partitioning and term partitioning. Figure 12 shows the indexing time only across the datasets.

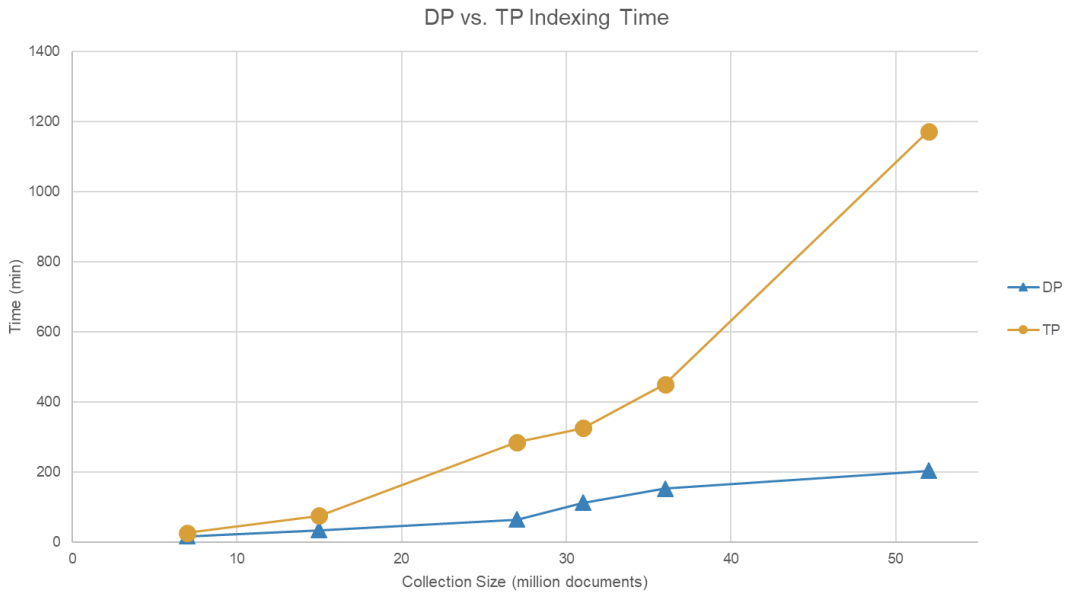

Figure 12.Document and term partitioning indexing time

The indexing time for both term and document partitioning is increasing with the dataset size. However, what is striking in this figure is the difference in how the increase happens in both techniques. As the figure shows, the increase in time for term partition is much larger than document partitioning. This difference could be due to the shuffling that happens in the term partitioning. Unlike term partitioning, document partitioning is designed to avoid unnecessary shuffling by working per

partitioning. Meanwhile, in term partitioning, Spark has to shuffle the whole dataset to group similar terms in the same partition to build the indexes. Although we tried to employ the recommended Spark method `AggregateByKey`, the shuffling proved to be very expensive. As a consequence, as the number of partitions and data increases, the time to build the index increases significantly.

This issue caused some problems for us. First, when indexing the 36 million with 900 partitions the `AggregateByKey`, ends up failing as some executors reach the maximum allocated memory for them. Therefore, we tried increasing the number of partitions. We tried different numbers and ended up with 1000 partition to the dataset of size 36 million document and 1400 for the full dataset. The indexing time kept increasing sharper compared to document partitioning, however we did not get any errors. However, it will be unfair to compare the performance with the different numbers of partitions for each schema since more partitions mean more time needed to schedule and compute them.

This experiment answers our research question about the partitioning technique best suited for Spark. Although we cannot claim that Spark cannot index the full collection, we can conclude that Spark is more suitable for document based partitioning as it does not need to shuffle huge amount of data over the network. The total time for preprocessing and indexing is available in Table 3

Table 3. Total Indexing time with Preprocessing for Document Partitioning and Term Partitioning in minutes

	Datasets					
	7	15	23	31	36	51
DP	33	101.4	165	251.3	322.6	483.3
TP	42.2	142.7	389	464.3	618.9	1438.1

3.9 Retrieval Performance

For the retrieval phase, we measure the response time of the system. We conducted two different experiments to test our system. In the first experiment, we measure the response time across the dataset sizes and check the effects of changing the executor memory and cores have on the response time if we fixed the dataset size. The last section compares the performance of our system against Elasticsearch. For both experiments, we retrieve the top k where k = 10 and use BM25 as our ranking method.

3.9.1 Response time across different dataset sizes

In this experiment, we used the same configurations for the executor memory and cores as we did in the indexing. Figure 13 shows the response time across different datasets – excluding the 51 million documents dataset – for different techniques. In

this test, we focused on DAAT, TAAT, Champion list with DAAT, and Maxscore. For the champion list the length of the postings list was set to 1000.

As expected with small datasets, the retrieval is fast; however, the response time increases as the dataset size increases. We can also see that in most cases, Maxscore is faster than DAAT, but with a small margin. Furthermore, most noticeably, the TAAT response time increases at a faster pace than the rest. This high increase could be due to the skewed data that occurs due to the architecture of the term partitioning.

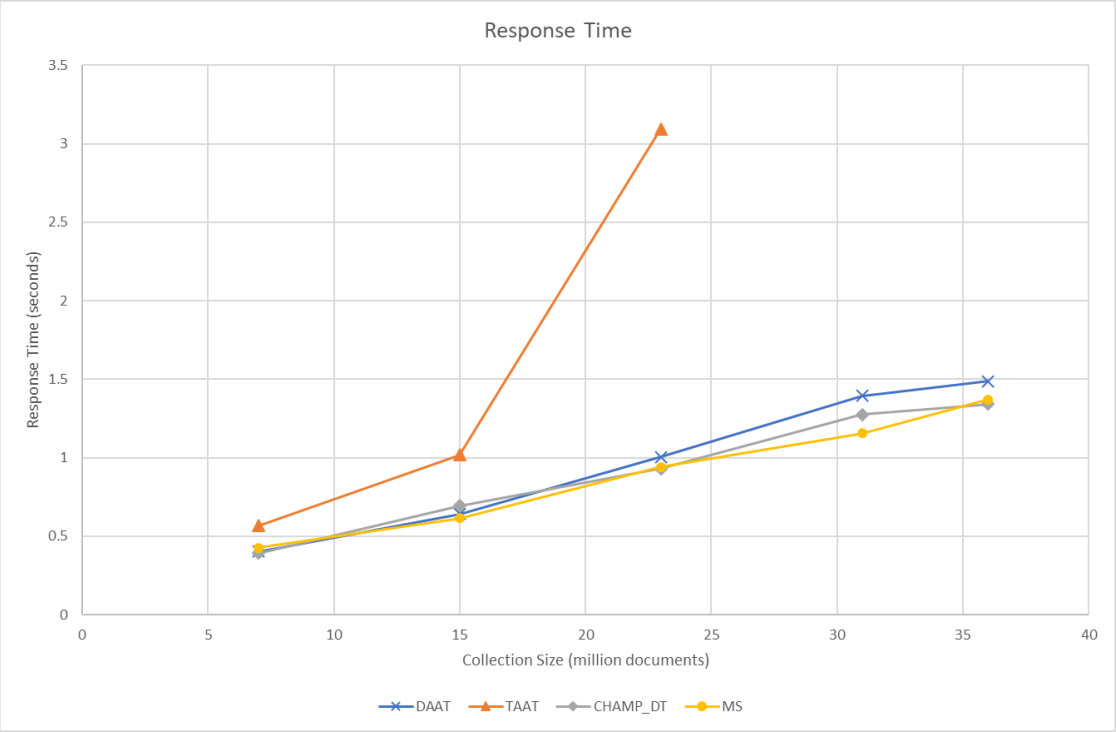

Figure 13. Response time for different techniques

By definition, the postings of a term must all be on the same partition. Therefore, terms with very long postings list may appear on the same partition; as a result, some partitions will be larger than the rest. Thus, execution might take longer on some partitions than others. This case was evident after inspecting the size of the partitions of term partitioning. While the size of the partitions was acceptably uniform in document partitioning, it was visibly skewed with term partitioning. For example, in one index for the same dataset, the size of partitions ranged from 300 MB to 70MB in term partitioning, but for document partitioning, the sizes were between 116 MB to 90 MB, and they had more fair distribution than term partitioning. Due to this, some tasks were taking longer than others since in Spark a task is the whole partition itself.

3.9.2 Effects of number of executors and cores on the retrieval

As mentioned earlier in this section, the execution time of a job in Spark is affected by the number of cores, executors, and partitions. In this experiment, we will focus solely on the 23 million records dataset. We conducted two tests; in the first, we fixed the number of executors and varied the number of cores per executor. While in the second, we fixed the number of cores and executors and varied the number of partitions. In both experiments, we used 11 executors. *Figure 14* shows how response time is affected by the number of partitions. In this test, we fixed the number of cores to 5 per executor, and in total, we had 11 executors. The figure shows an increase in time as we increase the number of partitions. This proves that in order to get an optimal response time, we need to make sure that the number of partitions is suitable for the job.

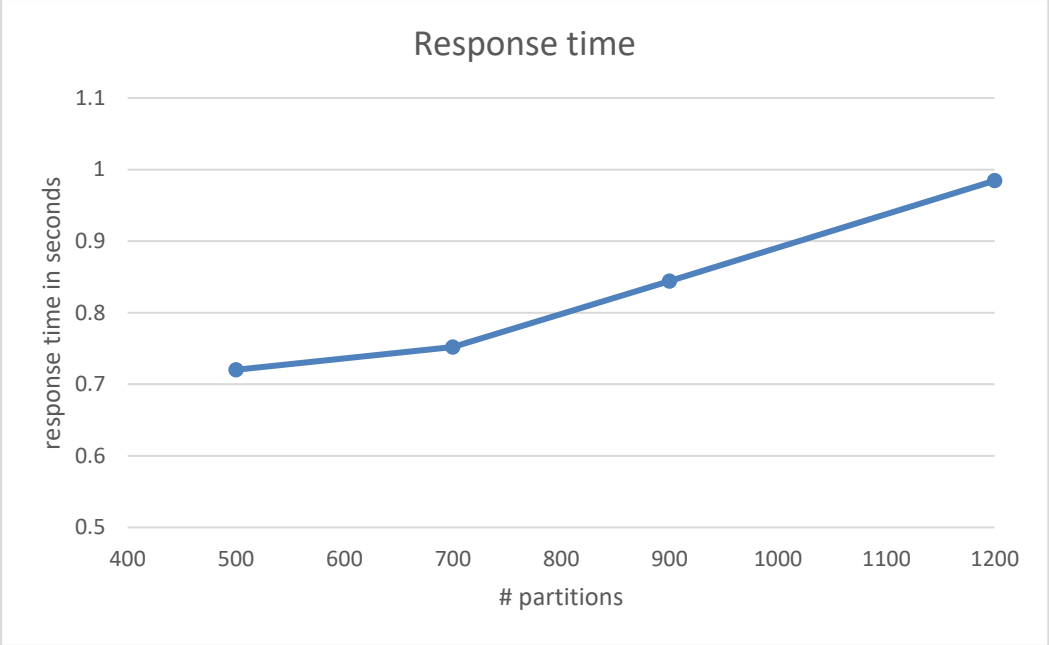

Figure 14. The effect of number partitions on response time

In the next experiment, we also fixed the number of executors to 11 while changing the number of cores to see how it affects the response time for the different partitions. From Figure 15, we note that as we increase the number of cores from 1 to 5 the response time decrease; however, the change in response time was insignificant after five cores. From this test, we can conclude that the performance of retrieval will have a slight change after hitting a specific number of cores. For this small collection and number of executors, five cores would be enough to carry out the retrieval job.

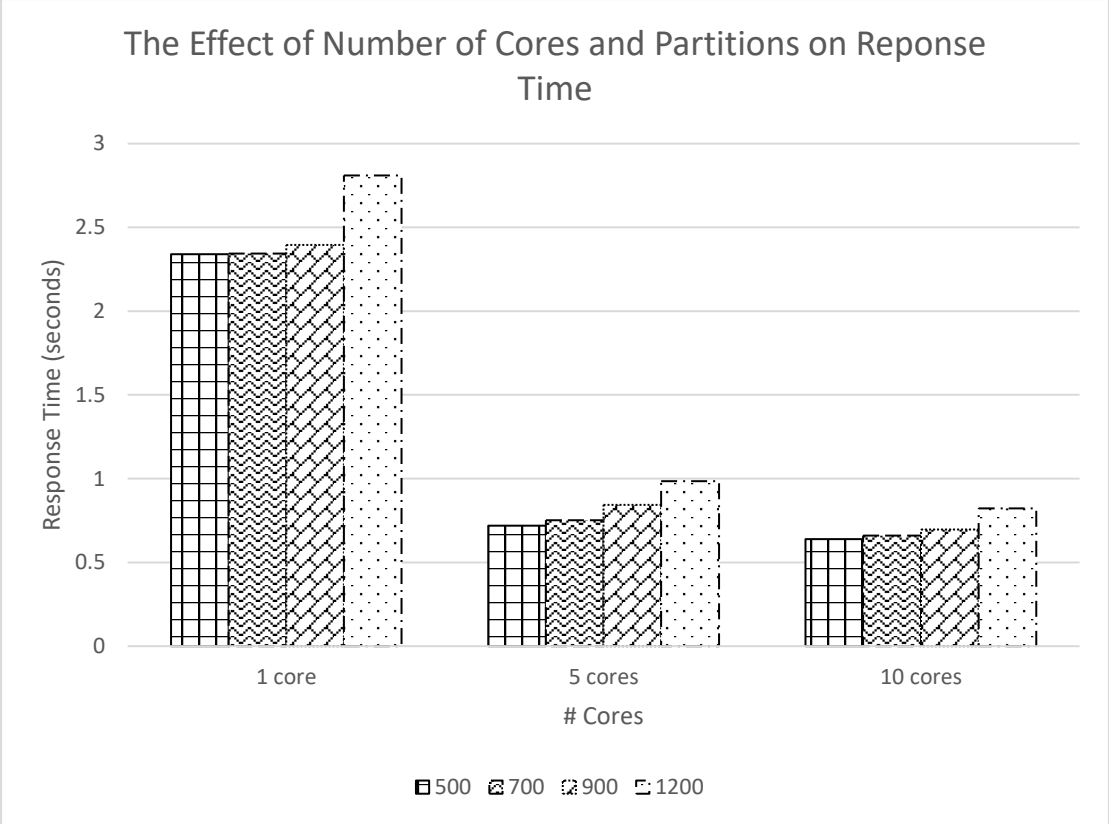

Figure 15. The effect of the number of cores and partitions on response time

3.10 SparkIR vs. Elasticsearch

3.10.1 Effectiveness

In this section, we use the full dataset to evaluate the performance of the system against Elasticsearch. For SparkIR, we will only focus on document partitioning with different efficiency techniques. Table 4 shows the effectiveness of document partitioning with the champion list, MaxScore, Tiering, and standard DAAT. The length of the postings list for the champion list is 1000 as before, and for Tiering, the upper and lower bounds of the tiers are 7 and 3. Thus, we end up with three tiers: tier 1 has postings with *tf* greater than or equal 7, tier 2 has postings with

tf less than 7 and greater than or equal to 3, the last tier has postings with *tf* less than 3.

The results reported in the table represent the effectiveness of SparkIR using BM25. It is noticeable that Elasticsearch has better performance than our system, although both systems use BM25. However, Elasticsearch uses a Lucene scoring function that is a modified version that differs slightly from the original BM25 that we use. Moreover, while indexing, our system faced some issues processing some of the WARC files due to encoding; thus, we had to catch and skip these records. In total, the number of skipped records was about 600,000 records. Meanwhile, Elasticsearch takes as an input raw text in JSON format. Therefore, Elasticsearch had no issues while indexing the full dataset. Due to our decision to skip these records, we might have skipped potential relevant documents, which led to a decrease in the effectiveness of Spark IR.

However, our original intent was to make sure that our system's effectiveness is not significantly affected by the efficiency techniques and this is what we achieved. Looking at the figures obtained we can see that the effectiveness was maintained across the different efficiency methods. We can see that with the efficiency techniques applied in SparkIR do not harm the effectiveness significantly.

It is also worth noting that the Tiering achieved the lowest results out of all the techniques. This could be due to how tiering chooses the top k scores from the tiers. It is possible that some terms do not exists in the same tier; therefore, a document that might be relevant might not have a high score when scoring terms from the first tier which might lead to it being excluded.

Table 4. The effectiveness of the different techniques against Elasticsearch

	TREC2013		TREC2014	
Techniques	P@10	nDCG@10	P@10	nDCG@10
DAAT	0.1820	0.1238	0.1840	0.1163
MaxScore	0.1820	0.1210	0.1840	0.1115
Champion list +DAAT	0.1880	0.1234	0.1900	0.1174
Tiering + DAAT	0.1820	0.1188	0.1760	0.1223
Elasticsearch	0.204	0.1323	0.276	0.1481

3.10.2 Response time

It was challenging to compare the performance of our proposed system against Elasticsearch because the systems differ in the concept of sizes of the partitions. Elastic search has a similar concept, but it is referred to as shards. In Elasticsearch, each shard is a Lucene index, and the Elasticsearch team recommends that a single shard should be large enough to reduce the overhead of searching many small shards. In the official documentation of Elasticsearch, they recommended the size of a shard should be around 20 to 40 GB, which is the contrast of the size of a partition in Spark. In Spark, it is recommended that the size of a partition to be small to fit in an executor while the computation is ongoing.

To conduct a fair comparison, we attempted to index Elasticsearch on 1024 shards - the maximum number of shards accepted by Elasticsearch-, but the indexing faced many errors as the rate of requests to the index exceeded the threshold of requests that Elasticsearch can handle for this small collection. This error caused Elasticsearch to skip some records during the indexing processes.

So, we opt to index Elastic search on 25 shards because the total size of the 50 million documents in JSON format is 447 GB. Thus, by having 25 shards, each shard should be around 20 GB or more. This approach puts our system at a disadvantage since it was tested earlier that the number of partitions affects the retrieval process and given the size of the index we cannot increase the number of cores as it will either force the executor to spill the persisted RDDs or to waste time in garbage collection instead of actual computation.

Nevertheless, we compared the response time of Elasticsearch with the different techniques of SparkIR to see the difference in response time if both systems were indexed to the recommended sizes of shards/partitions. Table 5 shows the average response time of Elasticsearch, and SparkIR with DAAT, champion list, and Tiering. Compared to Elasticsearch, SparkIR response time was around 8.8 to 9.2 for the different techniques. The response time got better as we applied the efficiency techniques, and this is due to the reduction of the index size. Although in all cases the index is in memory, without any efficiency methods most of the space in the executors is used for storage. Thus, the executor will need to free memory by doing garbage collection while scoring. This behavior was noticed with long postings lists, as with shorter postings lists, no garbage collection was needed; therefore, the response time was shorter for these queries. For the Champion list, the response time decreased

while maintaining its effectiveness, as shown in Table 4. However, by using Tiering, we achieved high response time in favor of effectiveness.

Table 5. Average response time of SparkIR and Elasticsearch

IR system	**Average response time(sec)**
SparkIR – DAAT	9.2239
SparkIR – Maxscore	8.75581
SparkIR – Champion list	1.90208
SparkIR – Tiering	0.74703
Elasticsearch	2.32715

Chapter 4 : Conclusion and Future Work

This work aimed to build a scalable IR engine over Spark framework. We tried to include all the basic features of an IR engine and its architecture and test how they can be implemented on this framework. The results showed that this framework has a promising performance in the IR field. Although the current results show that Spark is not a competitor to the current distributed search engine if it only supported state-of-the-arts techniques, but by including some efficiency techniques, it might be a promising competitor. Finally, as stated in earlier sections, we managed to create a library over Spark core that can work together with the other spark libraries such as MLlib and Streaming.

However, this work still has room for growth. In the future, we would like to include more advanced ranking methods like vector models and language models. Furthermore, we want to focus on efficiency and implement other retrieval efficiency methods like WAND. We also think that a framework like Spark could be used for batch query processing by issuing different queries at the same time. We also want to study new alternatives to indexing the collection as a term -based partitioned index to get overcome the shuffling bottleneck.

Furthermore, we based our system on RDDs, which are now considered low level as new data structures are added in the newer versions of Spark. Thus, we would like to examine the difference between implementing our system on RDDs and Datasets. In Spark 2.0+, Datasets are marketed as a combination of an RDD and Dataframes, making it the most efficient structure in the new versions of Spark

because it combines the best features of both data structures. Lastly, due to the time constraints, we were not able to test the limits of Spark in indexing more than 50 million documents. Therefore, in the future, we will aim to index the full ClueWeb12 dataset to see how Spark performs under such a huge collection in both indexing and search.

References

[1] R. Dollah and H. Aris, “A review of sector-specific big data analytics models,” *2017 IEEE Conf. Big Data Anal.*, pp. 72–80, 2017.

[2] A. Oussous, F. Z. Benjelloun, A. Ait Lahcen, and S. Belfkih, “Big Data technologies: A survey,” *J. King Saud Univ. - Comput. Inf. Sci.*, 2017.

[3] M. Zaharia, M. Chowdhury, T. Das, and A. Dave, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” *Nsdi*, pp. 2–2, 2012.

[4] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Proc. 6th Symp. Oper. Syst. Des. Implement.*, pp. 137–149, 2004.

[5] D. Carstoiu, E. Lepadatu, M. Gaspar, D. Carstoiu, E. Lepadatu, and M. Gaspar, “Hbase - non SQL Database, Performances Evaluation,” in *in Computer Science (1986), Master in Computer Science (1990), and PhD in Computer Science*, 2010, pp. 42–52.

[6] F. Chang *et al.*, “Bigtable: A distributed storage system for structured data,” *ACM Trans. Comput. Syst.*, vol. 26, no. 2, p. 4, 2008.

[7] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-fast Data Analysis*, 1st Ed. O’Reilly Media, Inc., 2015.

[8] D. Shahi, “Apache Solr: An Introduction,” in *Apache Solr: A Practical Approach to Enterprise Search*, Berkeley, CA: Apress, 2015, pp. 1–9.

[9] B. Dixit, R. Kuc, M. Rogozinski, and S. Chhajed, *Elasticsearch: A Complete Guide*. Packt Publishing Ltd, 2017.

[10] J. Lin, D. Metzler, T. Elsayed, and L. Wang, “Of Ivory and Smurfs: Loxodontan

MapReduce experiments for web search," *proceeding Proc. Eighteenth Text Retr. Conf.*, 2009.

[11] R. McCreadie, C. MacDonald, and I. Ounis, "MapReduce indexing strategies: Studying scalability and efficiency," *Inf. Process. Manag.*, vol. 48, no. 5, pp. 873–888, 2012.

[12] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma, "Terrier: A high performance and scalable information retrieval platform," in *Proceedings of the OSIR Workshop*, 2006, pp. 18–25.

[13] C. Macdonald, "Combining Terrier with Apache Spark to create Agile Experimental Information Retrieval Pipelines," pp. 1309–1312, 2018.

[14] Y. Ma, D. Liu, G. Scott, J. Uhlmann, and C.-R. Shyu, "In-Memory Distributed Indexing for Large-Scale Media Data Retrieval," *2017 IEEE Int. Symp. Multimed.*, pp. 232–239, 2017.

[15] A. K. Koliopoulos, P. Yiapanis, F. Tekiner, G. Nenadic, and J. Keane, "A Parallel Distributed Weka Framework for Big Data Mining Using Spark," *Proc. - 2015 IEEE Int. Congr. Big Data, BigData Congr. 2015*, pp. 9–16, 2015.

[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, 2009.

[17] H. Mushtaq, F. Liu, C. Costa, and P. Hofstee, "SparkGA : A Spark Framework for Cost Effective , Fast and Accurate DNA Analysis at Scale," *Proc. 8th ACM Int. Conf. Bioinformatics, Comput. Biol. Heal. Informatics*, pp. 148–157, 2017.

[18] D. Han, A. Agrawal, W. K. Liao, and A. Choudhary, "A novel scalable DBSCAN algorithm with spark," *Proc. - 2016 IEEE 30th Int. Parallel Distrib.*

Process. Symp. IPDPS 2016, pp. 1393–1402, 2016.

[19] T. Yang, K. Qian, D. C.-T. Lo, Y. Xie, Y. Shi, and L. Tao, "Improve the Prediction Accuracy of Naïve Bayes Classifier with Association Rule Mining," *2016 IEEE 2nd Int. Conf. Big Data Secur. Cloud (BigDataSecurity), IEEE Int. Conf. High Perform. Smart Comput. (HPSC), IEEE Int. Conf. Intell. Data Secur.*, pp. 129–133, 2016.

[20] J. Sun, Z. Shang, G. Li, D. Deng, and Z. Bao, "Dima: A distributed in-memory similarity-based query processing system," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1925–1928, 2017.

[21] Y. Chen and B. Bordbar, "DRESS: A Rule Engine on Spark for Event Stream Processing," *Proc. 3rd IEEE/ACM Int. Conf. Big Data Comput. Appl. Technol.*, pp. 46–51, 2016.

[22] C. L. A. Clarke, G. V. Cormack, and S. Büttcher, "Parallel Information Retrieval," in *Information Retrieval: Implementing and Evaluating Search Engines*, MIT Press, 2010.

[23] B. B. Cambazoglu and R. Baeza-Yates, *Scalability Challenges in Web Search Engines*. Morgan & Claypool, 2016.

[24] N. Tonellotto, C. Macdonald, and I. Ounis, "Efficient and effective retrieval using selective pruning," *Proc. sixth ACM Int. Conf. Web search data Min. - WSDM '13*, p. 63, 2013.

[25] C. D. Manning, P. Raghavan, and H. Schutze, *An introduction to information retrieval*, Online ed., vol. 21, no. c. Cambridge: Cambridge UniversityPress, 2009.

[26] H. Turtle, J. Flood, W. P. Co, and O. Drive, "QUERY EVALUATION :

STRATEGIES AND OPTIMIZATIONS," vol. 31, no. 6, pp. 831–850, 1995.

[27] H. Karau and R. Warren, *High performance Spark: best practices for scaling and optimizing Apache Spark*. " O'Reilly Media, Inc.," 2017.