BOOLEAN LOGICAL OPERATOR DRIVEN SELECTIVE DATA FILTERING FOR LARGE DATASETS

Glenn Davidson
Shikharesh Majumdar

Dept. of Systems and Computer Eng.
Carleton University
1125 Colonel By Drive. K1S 5B6.
Ottawa, ON, CANADA
majumdar@sce.carleton.ca

ABSTRACT

Specific users of a system processing large data sets are often interested in only a small subset of the large volumes of available data. This paper presents research on a parallel processing based data filtering technique that filters out and stores only the subset of data that is of interest to a given user. A user's preferences reflecting her/his interest are captured in a set of keywords or phrases which may be used in conjunction with Boolean operators. An Apache Spark based prototype is built and deployed on an Amazon EC2 cloud to demonstrate the viability of the approach and to analyze the performance of the proposed technique.

Keywords: Apache Spark, Parallel Data Filtering, Boolean Operator based Selective Data Reduction, Big Data Processing on Clouds

1 INTRODUCTION

The financial value of big data applications is immense in the present day, when a single company such as Facebook may process over 500 terabytes of user data every day (Kambatla et al. 2014). A single Boeing jet engine alone can produce 10 terabytes of data for every thirty minutes of operation. With the capability of producing such huge amounts of data comes the importance of reducing the data to knowledge and useful insights. The modern world has become increasingly dependent on computing systems, and with this dependency, the importance of text-based data has risen. Personal and business practices are highly reliant on user records, medical reports, books, and news articles, and the efficacy of their access translates into user productivity and financial profits. Everyday consumers of text-based data face the challenge of finding relevant information from their data, which becomes more time consuming as the size of their data grows.

Parallel processing frameworks such as MapReduce, Apache Hadoop, and Apache Spark deployed on a cloud can be used to speed up data processing operations on large datasets using distributed and parallel computing, while also providing scalability and fault tolerance. The Google File System and the MapReduce programming model first published in 2003 provided the impetus for the development of general purpose cluster computing frameworks for processing big data (Ghemawat et al. 2003). A successor to Google's implementation of MapReduce, Apache Hadoop provides an open source implementation of the MapReduce model, and a big data storage solution in the Hadoop Distributed File System (HDFS) (Apache Hadoop 2021). Although Hadoop provides the ability to process large volumes of data in parallel, the performance of the framework suffers from the obligatory writing of all intermediate results to disk storage between operations. The big data analytics engine Apache Spark was

designed to overcome this bottleneck by keeping intermediate results in memory rather than writing them to disk storage, achieving a computation speeds 10 – 100 times greater than Apache Hadoop (Apache Spark Project-Unified 2021).

This paper concerns large textual datasets and presents a parallel processing technique with which a user may reduce the size of their data to contain only relevant information using a list of user specified keywords/topics, often combined by Boolean operators. The reduced user data is stored in a searchable format. The user may then retrieve relevant information from the filtered data as needed, reducing both the search latency of data retrieval, and the volume of data stored. Example use cases include the storing of selected classes of patient records handled by a specific doctor or administrator in a hospital, the storage of selected tweets that correspond to a specific topic of interest for a journalist and the storage of specific classes of research papers by a researcher.

Data filtering is a computation intensive operation. Apache Spark is used to implement a proof-of-concept prototype of the proposed technique because of its high performance parallel processing framework that performs in memory processing of data thereby alleviating the performance penalties incurred by Hadoop/MapReduce that used disks for the storage of intermediate results and its flexible python API. The technique includes two components: the first provides the Filter method which processes a large dataset and reduces it to relevant information based on user preference terms and Boolean operators, while the second provides the Search Method which serves selected Filtered Data based on user queries to users. The Boolean operators in the user specified preferences, such as 'NOT', 'OR' and 'AND' are used to capture the multiple topics of interest for the user. This paper describes the research on devising of these methods and presents a performance analysis of the proof-of-concept prototype deployed on a cloud. The contributions of this paper include:

- A parallel method of structuring and filtering large data sets based on user preference terms and logical operators.
- A parallel method of searching data that is filtered based on user preference terms and logical operators.
- A proof-of-concept prototype for the methods devised.

Insights into system behavior and performance based on experiments performed on the prototype.

The paper is organized as follows: Section 2 provides the necessary background information for the tools used in this paper. Section 3 discusses related research relevant to the topic of this paper. Section 4 describes the proposed techniques and their implementation. A performance evaluation of the proof-of-concept prototype is given in Section 5, and Section 6 provides a conclusion of the paper.

2 BACKGROUND

Apache Spark is a computing engine for parallel processing in big data applications, with a unified API in Java, Python, and Scala (Zaharia et al. 2016). By providing support for a wide spectrum of big data processing tasks under one API, and remaining impartial to the data storage solutions, Apache Spark provides an effective platform for high performance, scalable, and fault tolerant big data solution.

3 SPARK ARCHITECTURE

Spark applications are run as a set of processes on separate computing nodes, coordinated by a single Driver Program that utilizes the Spark control class SparkContext (Apache Spark Project-Cluster 2021). The Driver Program provides instructions to Spark Executors which are responsible for the execution of tasks that compose a single application. The Driver Program runs on a single Master Node, and the Executor tasks can run on multiple Worker nodes. The Driver and Executors receive computing resources from an underlying Cluster Manager, responsible for allocating and monitoring the usage of resources in a

cluster across applications. When a Spark application begins, the Driver Program dispatches user application code to each Executor, assigns tasks to Executors, and collects results from Executors upon completion of tasks. The Driver Program and each Executor are run in a separate Java Virtual Machine (JVM).

Parallel operations on distributed data are accomplished through the use of the Spark Resilient Distributed Dataset (RDD) structure (Zaharia et al. 2016). An RDD consists of a collection of Java, Scala, or Python objects, divided into logical partitions and located on several nodes according to the distributed file system used in the deployment. An RDD is an immutable, fault tolerant, programming abstraction that can be operated on in parallel. A Spark application consists of a set of operations performed on RDDs, coordinated by the Driver Program.

The RDD, SparkContext, and the parallel processing engine constitute the Spark Core. Many Libraries have been built on top of the Spark Core API to provide convenient representations of big data tasks. The four Spark libraries provided by Apache are: Spark SQL, Spark Streaming, Spark Structured Streaming, and Spark MLlib (Machine Learning library).

3.1 Spark SQL

The Spark SQL API provides a Spark library used for the processing of structured data (Ambrust et al. 2015). Structured Query Language is a declarative programming language used to manage data stored in Relational Database Management Systems and is a standard of the International Organization for Standardization (ISO 2016). Spark SQL introduces a new data structure called DataFrame which organizes distributed data into a columnar format like that of Python pandas or Microsoft Excel. Spark SQL also acts as a distributed Structured Query Language engine, allowing queries with SQL syntax to be performed on the distributed DataFrame structure as if they were database tables. DataFrames are constructed by the Driver Program by reading common formats of structured data such as Comma Separated Values (CSV) or JSON, or by providing a structure to existing RDDs.

4 RELATED WORK

The use of parallel processing frameworks to address the computational challenges of big data processing is discussed in the literature. The MapReduce technique (Apache Hdoop 2021) that partitions the input data into multiple chunks that are processed concurrently is well known for processing large volumes of data in a timely manner. Managing resources to achieve deadlines of completion in performance sensitive MapReduce environments have been discussed in (Lim et al. 2017). A MapReduce based technique for creating a user profile using data on news items read by the user is described in (Gautam et al. 2015). The MapReduce based Rocchio relevance feedback algorithm is used in (Yang et al. 2016) for classifying documents. As discussed in Section 1 Apache Spark addresses some of the performance problems in MapReduce/Hadoop and is used in this research for achieving high system performance.

An overview of Spark SQL was given in (Armbrust et al. 2015) by its developers upon its addition to Apache Spark. The paper describes the motivations behind the new relational processing library Spark SQL, its new optimization engine Catalyst, and its implications for the Spark MLlib. The starting point of our paper originates from the work published in (Chanda et al. 2021), where the authors describe a method of data filtering using Spark, utilizing natural language processing and machine learning models to reduce large data sets to contain only relevant information for users. The paper includes a performance analysis of the technique running in a cluster, and the accuracy of five Spark MLlib models is evaluated. Similarly, the research presented in (Semberecki et al. 2016) outlines an Apache Spark based approach to text classification using natural language processing and the Spark MLlib. The paper describes how to distribute the tasks of NLP tokenization, text feature vector generation, and the use of Spark MLlib to train Machine Learning (ML) models for the classification of text. An analysis of the accuracy of the MLlib Native Bayes, Decision Tree, and Random Forests in text classification is given. Neither the

research in (Chanda et al. 2021) or (Semberecki et al. 2016) considers the use of user specified preferences to filter data using complex Boolean matching conditions that our paper focuses on. Moreover, they do not provide the flexibility of performing incremental filtering for handling changes in user preferences that is provided by our technique. A method of text classification for the detection of malicious URLs is demonstrated in (Lin et al. 2013), where large datasets containing safe and malicious URLs were filtered based on their text content to preprocess web requests. The technique however does not utilize distributed computing frameworks such as Spark or Hadoop used in our research.

5 THE PROPOSED TECHNIQUE

The data filtering technique consists of several stages, from raw data being uploaded to a cluster, to users receiving a smaller subset of data from the cluster based on their preferences. The components of the technique can be seen in Figure 1, and are described as follows:

- *Raw D*ata: Raw Data consists of a collection of one or more text-based files that arrive on the cluster to be processed by the Filter Method. The Raw Data size is typically very large making it time consuming to process on a conventional user's machine, thus motivating the requirement for filtering.
- *User Preferences*: The User Preferences consist of a set of keywords or phrases that the user is interested in, formed in a logical expression using the Boolean operators 'AND', 'OR', and 'NOT'. The User Preferences are used as an input to the Filter Method when a new Raw Dataset is to be filtered, so that only preferred data may be stored permanently.
- *Filter Method*: The Filter method processes the Raw Data into text structures specified by the User Preferences such as sentences, paragraphs, or file delimiters. The data resulting from this processing is referred to as Processed Raw data. The Filter method then tests the User Preferences against every item in this Processed Raw Data and saves all results that match the User Preferences as Filtered Data.
- *Filtered Data:* The Filtered Data is a reduced set of the Raw Data based on User Preferences (specified through keywords/sentences and Boolean operators), stored permanently so that it is available for the user to search using the Search Method.
- *Search Terms:* A Search Term consists of a set of key words or phrases to be used as an input to the Search Method, formed in a logical expression using the Boolean operators 'AND', 'OR', and 'NOT'. The Search Terms are used by the Search Method to search the Filtered Data for relevant information.
- *Search Method:* The Search Method accepts the Search Terms from a user and searches the Filtered Data for all elements that match the search terms, returning only Search Data to the user.
- *Search Data:* contains all elements from the Filtered Data for which the Search Method found a match.

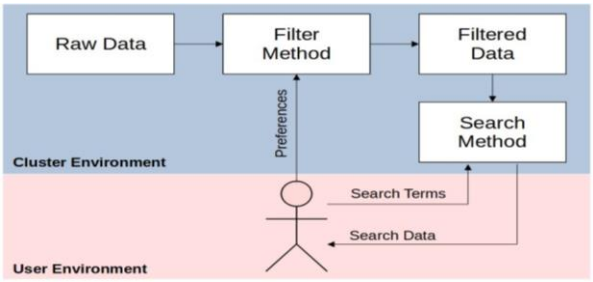

Figure 1: Components of the proposed technique.

The techniques can utilize a parallel processing framework deployed on a cluster or cloud to speed up the computations performed. A proof-of-concept prototype is built using the PySpark API of Spark 3.1

(Apache Spark Project-PySpark 2021), and has been tested on the Amazon Elastic Compute Cloud (EC2) infrastructure (Amazon 2021) using the python library textblob for Natural Language Processing (NLP) tasks (Loria 2020). Textblob used in the proposed filter method provides an API for NLP tasks such as part of speech tagging and noun phrase extraction and tokenization through splitting the text into words or sentences. Texblob NLP is discussed further in the context of the filter method that is discussed in the following section.

5.1 Filter Method

Filter Method processes the Raw Data upon arrival on the cluster based on the User Preferences and saves the result to a file as Filtered Data. The algorithm begins by reading the User Preferences to determine how the Raw Data should be tokenized and creates a SQL filter expression from the User Preferences. The data may be first tokenized by a delimiter string, and then by textblob NLP, or either transformations may be omitted. The SQL filter expression is created by transforming the keywords/phrases specified in the User Preferences and their associated logical operators into a SQL syntax text string.

The Filter method then loads the Raw Data into an RDD, and tokenizes the RDD by delimiter string and/or by textblob NLP if specified. The tokenized RDD is processed to remove select punctuations and is converted exclusively to lower-case characters. The structured RDD is then converted into a DataFrame and is registered as a Table so that the distributed SQL engine may be used to filter the data. Finally, the Filter Method performs the SQL preference query, saves the resulting Filtered Data on the cluster. Suppose the user submits the following preferences:

```
delimiter='---END.OF.RECORD---'

nlp=sentences

terms='cancer' OR ('carcinogen' AND 'radiation')
```

In this example the Filter Method will first tokenize the Raw Data into sections using the delimiter, and then further into sentences using textblob. After these transformations the Filter Method will find all sentences that either contain the word cancer, or contain both the words carcinogen and radiation, and will save this result to a file as Filtered Data. If for example, these user preferences are used with a Raw Data set consisting of only the three sentences:

1. Both the carcinogen asbestos and gamma radiation are harmful to the human body.
2. Asbestos is a known carcinogen.
3. The patient was diagnosed with cancer.

then the Filter Method would save the first and third sentences and discard the second sentence. The algorithm of the Filter Method is given in the pseudocode of Algorithm 1. Line 1 reads the User Preferences from the user, determining any delimiter string or NLP to be used, and constructs the Filter SQL operation, saving it as a string value in the variable 'filter'. In line 2, all files located in a Hadoop Distributed File System (HDFS) directory called 'raw_data' are loaded into an RDD, effectively partitioning the Raw Data files among Executors. Lines 3-7 transform the Raw Data RDD depending on the User Preferences specified, operating on RDD partitions in parallel. If a delimiter string is specified, then a flat map operation is performed in line 4, where each RDD element is split into one or more new elements, separated by the delimiter. If an NLP transformation is specified, then another flat map operation is performed in line 6, splitting each RDD element into one or more natural language units. The RDD is then transformed in line 7 to convert all characters to lower case, and to remove select punctuation and special characters. After the operations of lines 3-7, the RDD can be converted into a DataFrame and registered as a Table, done in lines 8-9. Finally, Spark SQL is used to perform the filter operation on the DataFrame in line 10. The resulting DataFrame is saved as filtered data in line 11 with a collect action and a disk write.

5.2 Search Method

The Search Method searches a user's Filtered Data using search terms provided by the user and sends the Search Data result from the cluster environment to the user's local machine. The Search method begins by reading the search terms provided by the user and converting the search terms to a SQL language query. The search application loads the filtered data into memory and finds all elements that match the search terms. The Search Method then sends all this data resulting from the search to the user. The algorithm of the Search Method is shown in Algorithm 2.

Algorithm 1: Filter Method

```
1:  delimiter, nlp, filter = parse_user_preferences()
2:  rdd = load(raw_data)
3:  if(delimiter != none):
4:      rdd = rdd.flatmap(delimiter function)
5:  if (nlp != none):
6:     rdd = rdd.flatmap(nlp function)
7:  rdd = rdd.map(lowercase)
8:  df = rdd.toDF()
9:  df.createTable()
10: filtered_data = spark.sql(filter)
11: SavetoFile(filtered_data.collect())
```

In line 1, the Search Method converts the search terms into a SQL language query. The Filtered Data is then loaded into memory as a DataFrame (line 2), and the DataFrame is registered as a table (line 3). The Search Method executes the SQL search query in line 4 and sends the result to the user in line 5.

Algorithm 2: Search Method

```
1: search = parse_search_terms()
2: filtered_df = load(filtered_data)
3: filtered_df.createTable()
4: search_data = spark.sql(search)
5: send(search_data)
```

6 PERFORMANCE EVALUATION

A set of experiments was performed to assess the performance of the proposed technique in an Apache Spark cluster. The cluster consisted of one master node and three worker nodes running on the Amazon EC2 cloud infrastructure (Amazon 2021), and used HDFS for storage. Each node used Ubuntu Server 20.04 LTS as the operating system, with Spark version 3.1, Hadoop version 2.7, and the Spark standalone cluster manager.

A c5a.4xlarge instance type was used for the Master Node, consisting of 16 cores running at 3.4 GHz clock frequency and 32 GB of RAM. The c5.2xlarge instance type was used for all Worker Nodes, each consisting of 8 cores running at 3.4 GHz clock frequency and 16 GB of RAM. The Filter Method and Search Method are evaluated separately. The workload parameters, performance metrics, and raw dataset are reviewed in the following sections before discussing results.

6.1 Raw Dataset

The raw dataset used in experimentation is the Westbury Lab Wikipedia corpus, a snapshot of all English language articles hosted on Wikipedia taken in 2010 (Shaoul et al. 2010). The 6 GB corpus was partitioned into segments of 100 MB and 700 MB, where the articles remain in alphabetical order of their title. Segments of the Wikipedia corpus were used to vary the size of raw data used in experimentation from 100 MB to 2.8 GB. Such raw dataset sizes were apt for capturing the performance characteristics of the proposed technique while keeping the experiment times and the concomitant cloud charges at reasonable levels.

6.2 Workload and System Parameters

A summary of the workload and system parameters used in the performance analysis of the Filter and Search methods are given in Table 1 in the following section. The experiments were conducted using a one-factor-at-a-time approach where all variables are held at their default values while a single variable (under test) is varied. The default values are indicated in bold text within the tables. For all experiments the Driver memory is 24 GB and the Executor memory on each Worker Node is 12 GB.

6.3 Performance Evaluation of the Filter Method

The workload and system parameters for the evaluation of the Filter Method are shown in Table 1. These parameters are chosen in such a way that they provided a range of different feasible values while keeping the cost of performing experiments on the EC2 cloud at a reasonable level. For the evaluation of the Filter Method, the number of cores allocated to each Worker mode was static, remaining at 8 cores, while the number of Worker Nodes was varied from 1 to 3. Some of the experiments require the use of boolean operators (OR, AND, NOT) in the user preferences used in filtering. Two parameters Filter Type (F_T) and Number of Terms (N_F) are used in these experiments. N_F is the number of terms included in the logical expression used in user preferences. Filter Type (F_T) is the type of the logical operators included in the expressions used in User Preferences. A set of tests were conducted using *single* word search terms. $F_{T\ for}$ these cases is denoted by the prefix 'Simple' (e.g. Simple OR and Simple AND). Another set of tests were conducted using complex logical expressions as filter terms, consisting of three search words in a boolean expression involving OR/AND/NOT operators. F_T for these cases is denoted by the prefix 'Complex'. Multiple such expressions were chained together by either logical 'OR' or logical 'AND' thus giving rise to an F_T of Complex OR and Complex AND respectively. Investigation of additional complex expressions of higher size using all the three operators OR, AND and NOT forms an important direction for future research

6.3.1 Performance Metrics for Performance Evaluation of Filtering

The performance metrics used are introduced next.

a) *Filter Completion Time (T_F):* The Filter completion time T_F is the difference between the time at which the Filter operation completes and the time at which the raw data transformation begins (Algorithm 1, lines 2-10).
b) Speedup (S(N)): The Speedup S(N) is the ratio of the completion time of a given application on a single Executor core, and the completion time of the application on N Executor cores. *Thus, S(N) provides an estimate of how fast n N-core system performs in comparison to a single core conventional system.*
c) *Efficiency E(N):* The Efficiency E(N) is defined as the fraction of time for which N Executor cores are usefully utilized when executing a given application. It is calculated as the ratio of an application's completion time when using a single Executor core, and the same application's completion time when using N Executor cores, multiplied by the number of Executor cores used (N).

d) *Logical Query Time (T_Q):* The Logical Query Time (T_Q) is defined as the difference between the time at which the Filter SQL operation results are collected on the Master Node, and the time at which the Filter SQL operation is initiated by the Master Node.

Table 1: Workload and System Parameters.

Parameter	Value
Raw Dataset Size (S_R)	*{0.1 GB, 0.7 GB, 1.4 GB,* **2.1 GB**, *2.8 GB}*
Filter Type (F_T)	*{***Simple OR***,* ***Simple AND****, Complex OR, Complex AND}*
Number of Terms (N_F)	*{***1***, 2, 3, 4, 5, 6, 7, 8}*
Number of Worker Nodes (N_w)	*{1, 2,* **3***}*
Total Number of Cluster Worker Cores (N)	*{1, 2, 3, 4,* **8,** *12,* ***24****}*

6.3.2 Effect of the number of Worker Nodes (N_W) on Tokenization Time (T_T) and Filter Completion Time (T_F)

The first set of experiments concern the effect of the number of Worker Nodes (N_W) and raw data size (S_R) on the performance of the Filter Method, where each Worker Node uses 8 cores. In this experiment, the Filter method tokenizes the raw data set first by string delimiter, and then by sentences using NLP. The effect of N_W is seen by examining T_F and S(N).

Figure 2 shows how T_F varies as N_W is changed from 1 to 3, using an S_R of 0.12 GB to 2.8 GB. As N_W increases, the completion time of the Filter Method decreases as more computing resources are utilized in parallel. Similarly, given an N_w the completion time of the Filter Method decreases as S_R decreases, as lower raw data sizes need less processing. However, at very small raw data sizes, the effect of adding additional workers is negligible, as a single node has sufficient computing resources to handle the workload. The tokenization step of the Filter Method (Algorithm 1, lines 2-7) was found to account for most of the Filter Method's completion time T_F.

While using three worker nodes and a raw data size of 2.8 GB, the raw data tokenization time was found to be 75 times greater than the time for doing the SQL Syntax Filter operation. Figure 3 shows precisely the effect that adding Worker Nodes has on the Filter Method's Speedup (S(N)). When only a single worker node equipped with 8 cores is used in the cluster, S(N) varies from 3.15 to 4.41 depending on S_R, where larger file sizes result in greater values of S(N). When three Worker Nodes are used in the cluster, the total number cores is 24 and S(N) varies from 3.15 to 12.57. This is because a single worker node has sufficient resources to compute the Filter Operation on the smaller S_R, but not on larger values of S_R. As S_R increases above 1.4 GB, for any given N_W, S(N) is observed to be less sensitive to changes in S_R. Efficiency E(N) achieved for different values of N_w is shown in Figure 4. At low S_R, E(N) decreases with an increase in the number of worker nodes each of which is equipped with 8 cores because a single node is sufficient to handle the data size. At higher S_R, E(N) decreases only slightly as N_w is changed from 1 to 2 and the curves tends to remain almost flat as N_w is changed from 2 to 3. This indicates that the useful utilization of the Executor cores does not change siginficantly when using 16 (N_w=2) and 24 (N_w=3) Executor cores at high S_R. The maximum Efficiency achieved is 0.55. E(N) is never close to 1 because of the time spent in scheduling tasks, increased result collection complexity with multiple cores, and unbalanced processing loads, which are not seen in the case of a single core.

6.3.3 Effect of Filter type (F_T) and Number of Terms (N_F)

As discussed in Section 5.3 experiments were performed to test the effect of different types of logical expressions in the SQL filter operation, in conjunction with the number of such terms in the expression.

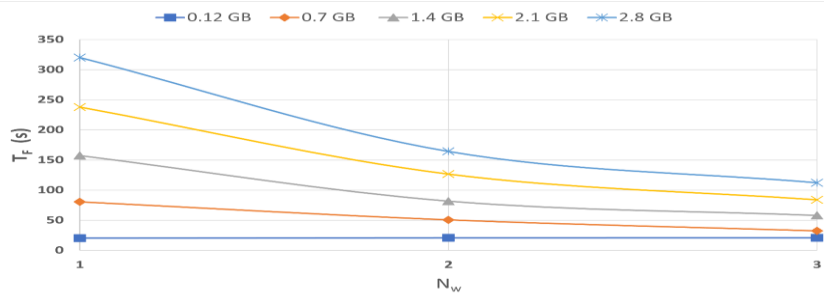

Figure 2: Filter Completion Time versus the number of Worker Nodes for various raw data sizes.

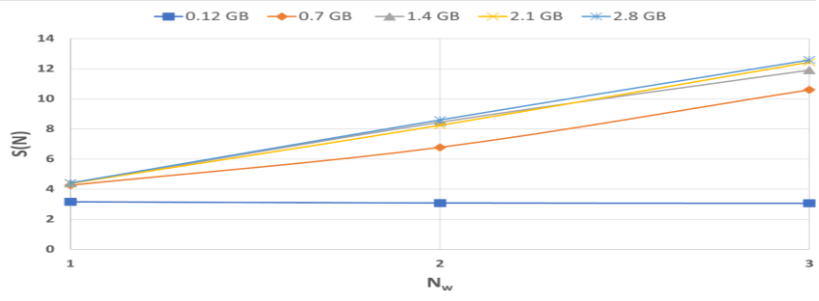

Figure 3: Speedup versus the Number of Worker Nodes for various raw data sizes

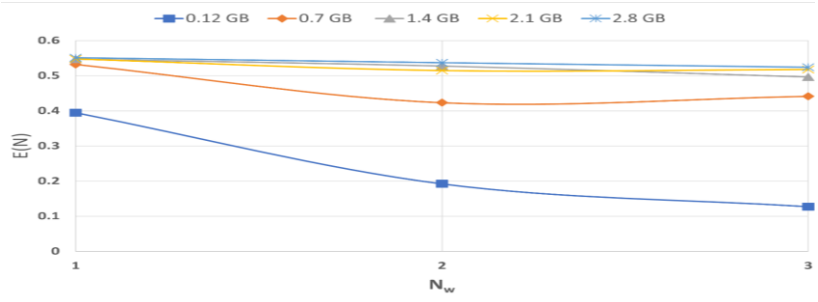

Figure 4: Efficiency versus the Number of Worker Nodes for various raw data sizes.

The default raw data size of 2.1 GB was used, and raw data was tokenized by NLP into paragraphs before performing SQL filter operations. Four types of filter expressions were tested; these can be divided into two categories: 'OR' structured expressions consisting of the logical 'OR' of all filter terms, and 'AND' structured expressions consisting of the logical 'AND' of all filter terms. The results of the tests for both cases: FT = Simple and FT = Complex are included in the figure.

Figure 5 shows the effect of adding additional Simple and Complex search terms (leading to an increase in N_F) in a SQL filter operation (Algorithm 1, line 10). In the 'OR' structured filter expressions, T_Q was found to increase linearly as N_F increases. Note that T_Q has two important components: result collection time required to collect the results from multiple cores and the pattern matching test time that corresponds to the matching of the terms in user preferences with that in the raw data. This increase in T_Q is because increasing N_F in an 'OR' structured expression always results in more matches found, and a greater number of Boolean expressions that must be evaluated, increasing both the result collection time and pattern matching test time. The same effect was not seen in the 'AND' structured filter expressions, where T_Q was seen to decrease slightly as N_F increases. This is because the addition of filter terms in an 'AND' structured filter expression tends to decrease the number of matches found, effectively reducing the result collection time, while the pattern matching test time only increases slightly as N_F increases resulting in a slight decrease in the overall value of T_Q. The average pattern matching test time only increases slightly

as N_F increases in this case because the top level structure of the Boolean expression is composed exclusively of 'AND' statements, only one of which needs to test false to prove the entire statement false, allowing for faster evaluation. In all cases the 'complex' versions of the expressions took 1.6-2.0 times longer than their 'simple' counterparts, while containing 3 times the number of logical operations to be performed. The execution time of Spark SQL statements can be seen as dependent on their level of specificity, where more specific queries are able to execute faster than general queries.

Incremental Filtering: If the user preferences change it is not always necessary to redo the entire filter operation starting with the raw data. When the change is such that the new preferences will lead to the "narrowing" of the existing filtered data the second filtering operation can be performed on the previously generated filtered data. Consider for example the situation in which original user preference includes the word 'computer'. If the user wants to change this preference to 'computer' AND 'architecture' the new filtering operation can reuse the previous filtered data and perform a filtering for 'architecture' thereby leading to a smaller filtering time in comparison to starting the filtering from scratch using the raw data with the new user preference.

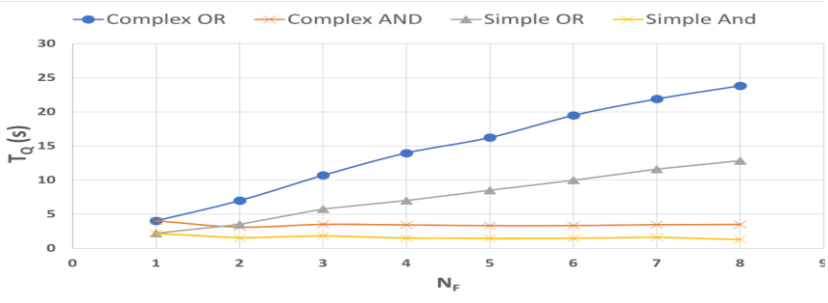

Figure 5: Logical Query Time versus the Number of Terms (N_F).

6.4 Performance Evaluation of the Search Method

The workload and system parameters for the experiments with the Search Method are given in Table 1, where the total number of Worker Node cores used in the cluster (N) is varied from 1-12. In measurements where less than 3 cores are used for the Search Method, less than 3 Worker Nodes must be used for the computation. The performance of the Search Method was evaluated using two filtered data files that are the product of single word filter operations on the raw data file with 2.1 GB Raw Data size. The two filtered data files were then individually searched for a single term, while the performance metrics of the Search method were measured. The description for the two independent filter and search operations can be written as "Filter('england')->Search('london')" and "Filter('america)->Search('washington')", where '->' denotes a precedence relationship: that is, the operation at the tail of -> is performed with the result of the operation indicated at the head of ->. This convention based on -> is used in the figures to reduce the size of labels. The performance metrics measured during these operations are given in the next section.

6.4.1 Performance Metrics

1. Raw Search Latency (T_{SR}): The Raw Search Latency is defined as the difference between the time at which search results are collected on the Master Node, and the time at which the search job is submitted to the cluster when running the Search Method on the entire Processed Raw Data.
2. Filtered Search Latency (T_{SF}): The Filtered Search Latency is defined as the difference between the time at which search results are collected on the Master Node, and the time at which the search job is submitted to the cluster when running the Search Method on a filtered data file.
3. Filtered Data Size (S_F): The Filtered Data Size used in the performance evaluation of the Search Method is defined as the size of text in bytes resulting from a single word filter operation on the Processed Raw Data.

4. Search Data Size (S_S): The Search Data Size in the Search Method tests is defined as the size of the text in bytes resulting from a single word search operation on the Filtered Data.
5. Filter Reduction Ratio (R): The Filter Reduction Ratio is defined as the ratio of S_R to S_F and is an effective metric of the reduction in data size performed by the Filter Method.
6. Search Efficiency (E_S): The Search Efficiency is defined as the ratio of T_{SR} to T_{SF}, and is an effective metric for the reduction in search time when searching Filtered Data instead of Processed Raw Data.

6.4.2 Data Sizes and Filter Reduction Ratio (R) in Search Method Experiments

A summary of the data sizes S_F, S_R, S_S, and the Filter Reduction Ratio from the experiments analyzing the performance of the Search Method, are given in Table 2. For the filter operation that finds all sentences from the Raw Data containing the word 'england', the resulting S_F is 8.31 MB. For the filter operation that finds all sentences from the Raw Data containing the word 'america', the resulting S_F is 55.25 MB. The 'england' filter operation reduces the Raw Data size by factor of 252.71, while the 'america' filter operation reduces the Raw Data size by a factor of 38.01. The resulting size of search data (S_S) in both operations is in the range of approximately 150-250 kB. These two filter and search operations are used throughout the performance evaluation of the search method.

Table 2: Search Method Performance Metrics: File Sizes and Reduction Ratio.

Number	Operation	S_R (MB)	S_F (MB)	S_S (MB)	R
1	*Filter('england') -> Search('london')*	*2100*	*8.31*	*0.147*	*252.71*
2	*Filter('america') -> Search('washington')*	*2100*	*55.25*	*0.257*	*38.01*

6.4.3 Searching Processed Raw Data (T_{SR}) versus Searching Filtered Data (T_{SF})

Figure 6 presents the comparison of the time to perform search operations on filtered data with that achieved with raw data. It shows the improvement in performance achieved by using the data filtering algorithm over a conventional system where search is performed on the raw data. Such a comparison of the times to search Filtered Data and Processed Raw Data was done using the two filter and search operations described in Section 5.4. Figure 6 shows a 3 axis plot of $T_{SR}(N)$ and $T_{SF}(N)$ versus N in the two operations, where T_{SR} and T_{SF}have separate axes and scales. As described in Section 5.4 operation 1 corresponds to Filter('england')->Search('london')" and operation 2 corresponds to Filter('america)->Search('washington')". T_SR1 and T_SF1 correspond to operation 1 whereas T_SR2 and T_SF2 correspond to operation 2. The plot shows that all T_{SR} values across N are greater than 1 second, while all T_{SF} values across N are less than 225 milliseconds. As expected, as N increases, all search times decrease. The largest difference in search time between raw and filtered data is seen at an N of 1 in operation 1, where T_{SR} is 8.66 s, and T_{SF} is 70.77 ms. This points to an improvement in search latency by a few orders of magnitude when the Filtered Data is searched instead of the Processed Raw Data.

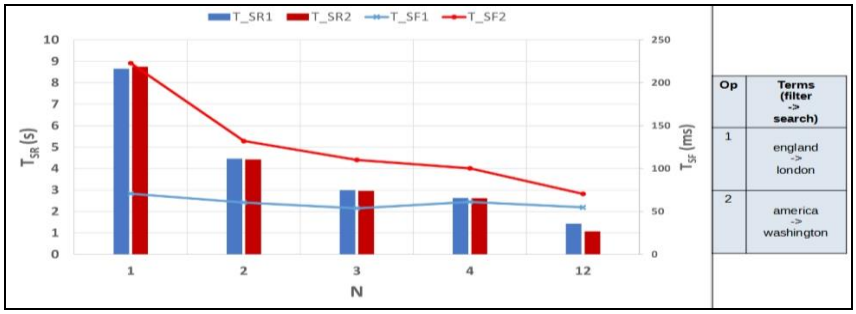

Figure 6: $T_{SR}(N)$ (left axis) and $T_{SF}(N)$ (right axis) for two Search Method Tests (Filter(england)->Search(london) (blue), Filter(america)->Search(washington) (red)).

6.4.4 Impact of Search Data Size (S_S) in Filtered Data Search Time (T_{SF})

While it is clear that searching filtered data is faster than searching raw data, an unexpected factor was found to dominate the search times on Filtered Data (T_{SF}). The major portion of the computation time (greater than 85%) in T_{SF} was found to be in the result collection stage of the computation. This is confirmed by the Spark cluster manager console that can be checked manually after the run of an application. The relationship between T_SF and S_S for two operations is captured in Figure 7. Operation 1 corresponds to Filter('america')->Search('america')" and operation 2 corresponds to Filter('america)->Search('washington')". Figure 7 demonstrates how T_{SF} is directly proportional to the size of the resulting search data (S_S) for each operation (op1 and op2): a higher T_SF is observed to accompany a higher S_S. In this example, the Filtered Data file is searched once for the same single term that was used in the Filter stage (operation 1) , and once for a new term that was not used in the filter stage (operation 2). This results in the S_S of operation 1 being the same as the size as that of the Filtered Data. In Figure 7 as N increases, parallelism in execution increases and both T_{SF1} and T_{SF2} are observed to decrease. The size of the search result is independent of the number of processing elements used and S_S is observed to remain invariant for a given operation when the value of N is changed. Similar results for other test cases that used a different set of keyword for filtering and for searching were obtained.

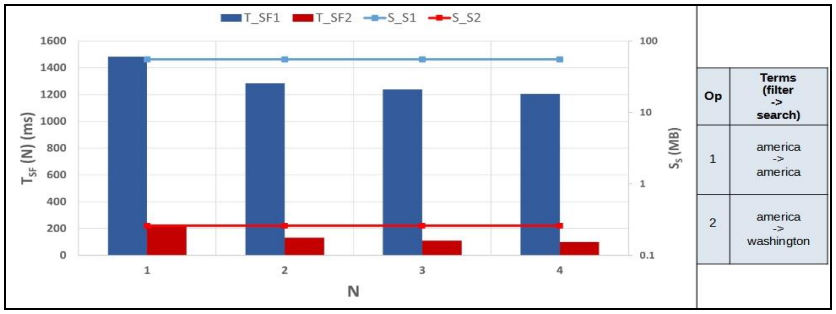

Figure 7: T_{SF}(N) and S_S for two Search Method Tests, (Filter(america)->Search(america) (blue), Filter(america)->Search(washington) (red)).

6.4.5 Effect of Search Data Size (S_S) on Search Efficiency (E_S)

Using the comparison data of T_{SR} and T_{SF} shown in Figure 7, a trend of E_S can be plotted, displaying a single metric for the reduction in search time when searching Filtered Data instead of searching Processed Raw Data. Figure 8 shows E_S and S_S versus N in a three axis plot, showing that E_S is higher when S_S is smaller, as a smaller S_S results in a lower result collection time. The maximum E_S observed is 122 at N = 1. This means that a reduction in search time by a factor of 122 is achieved by searching the flitered data. Operation 1 in Figure 8 gives rise to an S_S of 0.15 MB, while S_S of operation 2 is 0.25 MB. Also shown is the effect of N on E_S, where fewer cores used in the search operation results in higher E_S. This indicates that the impact of filtering on search time is more pronounced for smaller values of N.

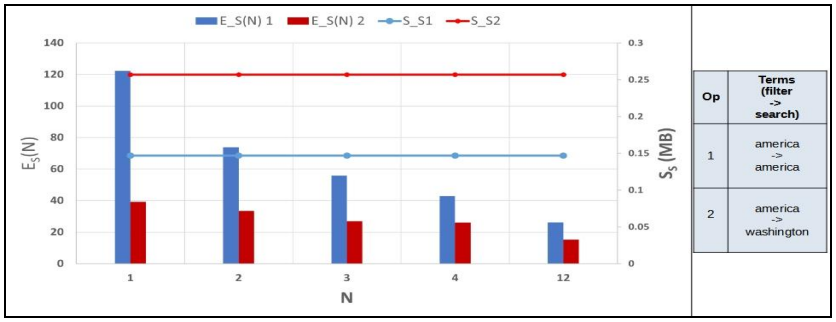

Figure 8: The impact of N on E_S and S_s.

7 CONCLUSIONS

A parallel filtering technique for large data sets based on user preferred terms and boolean logical operators is presented, including the performance analysis of a proof-of-concept prototype running in a four-node Spark cluster on an Amazon EC2 cloud. The key findings of the paper include the following:

- The efficacy of the technique is demonstrated through the proof-of-concept prototype. For the workloads experimented with, the search latency was shown to be reduced up to 122 times by searching filtered data instead of raw data, where filtering had a stronger effect on Search Efficiency for lower numbers of Executor cores.
- *Incremental Filtering*: As noted in Section 5.3.3 previously filtered data can be reused to efficiently handle changes in user preferences.
- An increase in parallelism through the use of additional worker nodes (N_W) gave rise to a significant improvement in the filtering time (shown in Figure 2).
- Filter operations having many 'AND' terms were found to execute faster than Filter operations having many 'OR' terms. In other words, more specific Filter operations were found to execute faster. The increase in the number of AND terms did not seem to affect the filtering time significantly.
- A significant part of the computation time was found to be in structuring and tokenizing the raw data to generate Processed Raw Data. This processing was sped up by adding Worker Nodes.
- *Cost Reduction:* A hybrid deployment model can be used for cost reduction in a multi-user system. For reducing the charges of the cloud service provider text data filtering can be performed on local user machines using the preferences of the specific users each of whom can use the same Processed Raw Data produced on the cloud. This will enable the sharing of the same Processed Raw Data generated on the cloud by multiple users that are interested in the same raw data there by reducing the cost for usage of the public cloud for data processing.

Directions of future research include the filtering of text by Machine Learning based text classification, and the filtering of pdf data and video files.

ACKNOWLEDGMENTS

Financial support for this research was provided by NSERC of Canada.

REFERENCES

Amazon Web Services Inc. 2021. "Amazon EC2". https://aws.amazon.com/ec2/. Accessed August 22, 2021.

Armbrust, M., R. S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, A. Ghodsi, and M. Zaharia. 2015. "Spark SQL: Relational Data Processing in Spark". In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, pp. 1383–1394. Melbourne, Australia. Association for Computing Machinery.

Apache Hadoop Project, 2021. "Apache Hadoop". http://hadoop.apache.org/. Accessed August 22, 2021.

Apache Spark Project-Cluster 2021. "Cluster Mode Overview - Spark 3.1.2 Documentation". https://spark.apache.org/docs/latest/cluster-overview.html. Accessed August 29, 2021.

Apache Spark Project-PySpark, 2021, "PySpark Documentation — PySpark 3.1.2 Documentation". https://spark.apache.org/docs/latest/api/python/index.html. Accessed August 29, 2021.

Apache Spark Project-Unified, 2021. "Apache Spark – Unified Engine for Big Data". https://spark.apache.org/. Accessed August 22, 2021.

Chanda. B., and S. Majumdar. 2021. "A Parallel Processing Technique for Extracting and Storing User Specified Data", In *Proceedings of the 8th International Conference on Future Internet of Things and Cloud (FiCloud 2021)*, pp. 241-249. Rome, Italy.

Ghemawat, S. H. Gobioff, and S.-T. Leung. 2003. "The Google file system", In *Proceedings of the Nineteenth ACM Symposium on Operating systems Principles (SOSP '03)*, pp. 29-43.New York, U.S.A., Association for Computing Machinery.

ISO/IEC JTC 1/SC 32. 2016. ISO/IEC 9075-1:2016. *Information Technology — Database Languages — SQL — Part 1: Framework (SQL/Framework)*. International Organization for Standardization.

Gautam, A., and P. Bedi. 2015. "MR-VSM: Map Reduce Based Vector Space Model for User Profiling-an Empirical Study on News Data", In *Proceedings of the International Conference on Advances in Computing, Communications, and Informatics (ICACCI)*, pp. 355-360. Kochi, India .

Kambatla, K., G. Kollias, V. Kumar, and A. Grama. 2014. "Trends in Big Data Analytics", *Journal of Parallel and Distributed Computing* vol. 60, July 2014, pp. 2561–2573.

Lim,N., S. Majumdar, P. Ashwood-Smith. 2017. "MRCP-RM: a Technique for Resource Allocation and Scheduling of MapReduce Jobs with Deadlines", *IEEE Transactions on Parallel and Distributed Systems*, 2017, Vol. 28, pp. 1375-1389.

Lin, M.-S., C.-Y. Chiu, Y.-J. Lee, and H.-K. Pao. 2013. "Malicious URL Filtering — a Big Data Application", In *Proceedings of the 2013 IEEE International Conference on Big Data.* pp. 589-596. Santa Clara, USA, International Electrical and Electronics Engineering.

Loria, S. 2020. "TextBlob: Simplified Text Processing". https://textblob.readthedocs.io/en/dev/. Accessed August 29, 2021.

Semberecki, P., and H. Maciejewski. 2016. "Distributed Classification of Text Documents on Apache Spark Platform". In *Proceedings of the 15th International Conference on Artificial Intelligence and Soft Computing*, pp. 621-630. Zakopnae, Poland.

Shaoul,C., and C. Westbury. 2010. "The Westbury Lab Wikipedia Corpus"., http://www.psych.ualberta.ca/~westburylab/downloads/westburylab.wikicorp.download.html. Accessed June 17th, 2021.

Yang, W., Y. Fu, and D. Zhang.. 2016. "An Improved Parallel Algorithm for Text Categorization," In *Proceedings of the International Symposium on Computer, Consumer and Control (IS3C)*, pp. 451-454. Xi'an, China.

Zaharia M., R.S. Xin., P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng,, J. Rosen, S. Venkataraman, M.J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, 2016. "Apache Spark: a Unified Engine for Big Data Processing", *Communications of the ACM*. Vol.59, pp. 56–65.

AUTHOR BIOGRAPHIES

GLENN DAVIDSON received his Master of Engineering in Systems and Computer Engineering from Carleton University in 2022. After working as a Power Electronics Engineer for several years, he turned to Carleton University to learn about software development, Operating Systems, and Embedded Systems. He is currently an Embedded Software Engineer using a Yocto Linux platform, and likes solving problems with Python and C++.

SHIKHARESH MAJUMDAR is a Chancellor's Professor and the Director of the Real Time and Distributed Systems Research Centre at the Department of Systems and Computer Engineering in Carleton University in Ottawa, Canada. He holds a Ph.D. degree in Computational Science from University of Saskatchewan, Saskatoon, Canada. His research interests are in the areas of cloud and grid computing, smart systems, operating systems and performance evaluation. Dr. Majumdar actively collaborates with the industrial sector and has performed his sabbatical research at Nortel and Cistech. He

has been the area editor for the Simulation Modelling Practice and Theory journal published by Elsevier (2008-2017). Dr. Majumdar is a Fellow of the Institution of Engineering and Technology (FIET), a member of ACM, a senior member of IEEE and a Professional Engineer (Ontario, Canada). He was a Distinguished Visitor for the IEEE Computer Society from 1998 to 2001.