A Parallel Processing Technique for Filtering and Storing User Specified Data

by

Bannya Chanda

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Users are often interested in a specific type of data (user-preferred data) from a large-volume dataset. An efficient system that only stores user-preferred data from the large dataset can reduce the search latency, which allows the users to search for relevant information in a timely manner. The motivation behind this thesis is to devise a technique that filters a large dataset and stores only the filtered data, thereby saving storage space for the user. Running the filtering operation can be CPU-intensive, which can lead to high latency in extracting preferred data from the dataset. To solve this problem, the technique employs parallel processing and machine learning. A proof-of-concept prototype for this technique has been built on Apache Spark. The performance of the prototype subjected to synthetic datasets is analyzed. The analysis of experimental results shows the viability of this technique and provides insights into the system behavior and performance.

Acknowledgements

First, I thank Almighty for the successful completion of this thesis. I would like to express my special gratitude to my supervisor, Dr. Shikharesh Majumdar, who has provided immeasurable support, both academically and personally, over the last 24 months. It would have not been possible to complete this thesis without his invaluable guidance and support. I am forever grateful to him for being the pilot of this scholarly journey.

Finally, I would like to thank my parents and loved ones, who have supported me throughout the entire process by helping me stay balanced and focused in putting the pieces together. Without their limitless love and support, I would not have reached my goal today. I am forever grateful for your love.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Program Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
EC2	Elastic Compute Cloud
FIFO	First in First out
GB	Gigabyte
Gz	Gigahertz
HDFS	Hadoop Distributed File System
IDF	Inverse Document Frequency
JSON	JavaScript Object Notation
KB	Kilobyte
LTS	Long Term Support
MB	Megabyte
ML	Machine Learning
MLlib	Apache Spark Machine Learning Library
NER	Named Entity Recognition
NLP	Natural Language Processing
NLTK	Natural Language Toolkit
RAM	Random Access Memory
REGEX	Regular Expression
RDD	Resilient Distributed Dataset
SQL	Structured Query Language

List of Symbols

B_I	Batch Interval
B_L	Length of Batch in Records
C_B	Total Number of Bytes Processed
C_{DP}	Centralized Partitioning
C_R	Raw Dataset Category
C_{R1}	Raw Dataset Category 1
C_{R2}	Raw Dataset Category 2
D_P	Data Partitioning Strategy
E_{DP}	Equal Distribution-Based Partitioning
E_F	Filtering Efficiency
E(N)	Efficiency
f_n	False Negative
f_p	False Positive
F_1	F-Measure
N	Number of Executor Cores
N_B	Number of Batches
N_F	Number of Files
N_W	Number of Worker Nodes
N_P	Number of Partitions in a Kafka Topic
S_F	File Size
S(N)	Speedup for N Processors
S_R	Raw Dataset Size

T_C	Computation Time of Filter Method
T_{CS}	Computation Time of the Search Method Performed within Filtered Data
T_{CT}	Computation Time of the Search Method Performed within Non-filtered Data
T_E	Elapsed Time
T_{EF}	Timestamp When the Filter Method Completes
T_P	Average Batch Processing Latency
t_p	True Positive
T_{P1}	Timestamp When Kafka Producer Finishes Sending a Batch
T_{P2}	Timestamp When the Filter Method Finishes Processing the Batch
T_S	Computation Time of Search Method
T_{S1}	Timestamp When the Filter Method Starts Processing the First Batch
T_{S2}	Timestamp When the Filter Method Completes the Processing of All the Batches
T_{SF}	Timestamp when the Filter Method Starts Execution
X	Throughput

Chapter 1: Introduction

1.1 Background

In recent years, there has been a significant shift in focus in finding useful information from large volumes of data, as large data continues to infiltrate our day-to-day lives. But what is meant by "large data"? Large data is a term that refers to large volumes of complex, unprocessed data. These data are difficult and time consuming to process using traditional processing methodologies. In today's data-driven world, large data applications are required in sectors like banking and securities, communication and media management, healthcare, education, manufacturing and natural resources, government, insurance, retail, and wholesale trades, transportation as well as energy and utilities [88]. Each of these industries heavily relies on programs that process large data to have continuous access to fast and reliable information. For example, banks are using large data to monitor their clients' accounts in real time so that they can take prompt action against any fraudulent activities that might happen in the accounts by using network analytics and natural language processors. The Canadian Food Inspection Agency needs to process large data to detect and study food-related illnesses and diseases, which in turn allows for faster responses to treatments. Other similar users of large data are typically heavily reliant on actions such as finding relevant information from articles, journals, meeting minutes, or logs, all of which contain a significant amount of text data. The user is often interested in a smaller subset of these data, and the effectiveness of their actions is often dependent on how accurately the data are derived from the source material. Considering this, the primary purpose of such a smaller subset of data is to assist users in making more informed decisions by deriving useful information from large datasets.

This issue highlights the challenge of easily finding the relevant information required by the user from a given dataset. If the given dataset is large, this becomes even more challenging due to the time requirement of searching through the large volume of data. Typically, there are several challenges associated with large data, such as insufficient understanding due to its complex nature, its high expense to manage, the complexity associated with managing the data quality, and the difficult process of converting large data into valuable insights [49].

To address these issues, this thesis focuses on devising a technique that can filter a raw dataset to find the desired data as per the user's preferences and store these filtered data. After filtering the user-preferred data, the user will be able to search through only this filtered dataset, which is significantly smaller in volume in comparison to the entire raw dataset. This technique is expected to reduce the latency during the search as well as the volume of data that needs to be stored for a specific user.

1.2 Problem Description and Motivation

Over 90% of the world's data has been created in the last two years, with 2.5 quintillion bytes of data being generated daily [46]. It is clear that the future will be filled with even larger amounts of data, which could potentially lead to more data problems. These problems might include the high cost of data solutions, complex systems for managing data, keeping up with the growth in data, data integration and the constant nature of changing data. While users can benefit from this growth in data, they must also be aware of the challenges such as collecting, storing, sharing, and securing these large amounts of data as well as creating and utilizing meaningful insights from them [47].

Often, a given user of large data is interested in a particular set of topics or keywords. Their queries may include the names of persons and places, a list of medicines, the dates of sporting events, temperatures, products, etc. To elaborate further on this, let us consider an example in which an environmental sustainability student is investigating how the temperature has been behaving for the past decade in the city of Ottawa and how it is being affected by climate change. To pursue this, the student primarily needs to find out what were the temperatures recorded in the city of Ottawa for the last 10 years (i.e., during each season, what were the fluctuations in daily temperature from one year to another). The student will also need to know what the trend has been for adjacent provinces for the same timeline (as they are likely to vary due to differences in total populations, cars, and industries' overall carbon emissions). This information will provide the student the data needed to determine the trend in the last 10 years. To get this information, the student would have to normally go through hundreds of articles, published journals, and research papers and then find the relevant information from each paper or journal. However, finding that information quickly from this large data is often time consuming and may require additional resources to find the data relevant to the research.

The problem addressed by this research focuses on filtering and storing only a subset of the data. The processing of user queries will be based on these filtered data. The data to be filtered are based on a set of preferences specified by the user describing his/her topic(s) of interest. The technique will store only the texts that capture the data described by the user's preferences and discard the rest of the raw data. By doing this, the proposed technique is expected to reduce data search latencies associated with users' queries and the resources required for data storage.

1.3 Proposed Solution

To find a solution to the problem discussed in the previous section, this thesis focuses on filtering the raw dataset and storing only the user "preferred" data specified by a user as filtered data. This approach is referred to as the "Filter Method". However, the filtering operation can require a significant amount of time to extract the preferred information from the large raw dataset. Hence, to reduce this filtering time, a parallel processing platform can prove beneficial. One of the most widely used parallel processing platforms is Hadoop, which is based on the concept of batch processing [50]. Hadoop Distributed File System (HDFS) is a distributed file system that handles large datasets running on commodity hardware. It is used to scale a single Apache Hadoop cluster to hundreds (or even thousands) of nodes [48]. Despite HDFS's ability to process a large amount of data in parallel, there are a few drawbacks of the Hadoop system such as its inability to efficiently support the random reading of small files, limited capability to process data in real time, inability to support cyclic data flow, and lack of an efficient caching system [51]. Apache Spark, a second-generation batch processing engine with stream processing capabilities, addresses most of these issues. A few key benefits of Apache Spark include its faster in-memory performance, suitability for iterative and live-stream data analysis, user-friendliness, and built-in tools for resource allocation, scheduling, and monitoring [52].

A proof-of-concept prototype of the proposed technique is built on Apache Spark. Its parallel processing engine, machine learning library, and high-level APIs for Python make Spark appropriate for researching this technique. Using Spark's machine learning library and the Python libraries in Spark's parallel processing engine, this technique filters the raw dataset based on user preferences and stores the filtered data to make them available to the

user. These filtered data contain only the user-preferred information from which the user can search for relevant information such as text containing specific names, dates, locations, and products.

1.4 Goal of the Thesis

This thesis focuses on investigating a parallel processing-based technique for filtering and storing users' preferred information from a raw dataset. By doing so, it identifies a way to reduce the time and resources required to search within a large dataset based on the user's queries and find the preferred information. The primary goal of the technique is to reduce the storage of data as well as speed up the data search operations performed by the user on the stored dataset. Additionally, this thesis performs a detailed performance evaluation of the proposed technique using a synthetic workload and presents an analysis of the performance results.

1.5 Contributions of the Thesis

The primary contributions of this thesis include the following.

- A parallel algorithm for filtering raw datasets based on the user's preferences and storing the filtered data for future use.
- A parallel algorithm for searching within the filtered data based on the user's queries.
- A proof-of-concept prototype for the proposed technique.
- Insights into system behavior and performance based on a rigorous performance analysis of the prototype. These include:
    - An experimental demonstration of the technique's ability to speed up data filtering and searching.

    - An experimental demonstration of the filtering efficiency.
    - A description of the relationship between system performance and workload/system parameters.
- A proof-of-concept prototype for a sample application of the technique that demonstrates the use of the proposed data filtering and storage technique.

Some preliminary high-level concepts underlying this technique and some of the research results are presented in a short paper [77].

1.6 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 presents the background on various tools and concepts used in this thesis and covers a key set of related works. The methodology, algorithms, and implementation of a proof-of-concept prototype for the technique are briefly explained in Chapter 3. Chapter 4 describes the experiments performed on the prototype and the experimental results. Chapter 5 concludes the thesis and discusses possible directions for future research.

Chapter 2: Background and Literature Review

This chapter starts by summarizing the parallel data processing techniques Apache Hadoop and Apache Spark in Section 2.1. Then, Apache Spark, its architecture, and components are presented in Section 2.2. Next, in Section 2.3 and Section 2.4, the internals of Apache Spark's machine learning library (MLlib) and Spark Structured Streaming are discussed. The internals and architecture of Apache Kafka are discussed in Section 2.5. A Python library used is discussed in Section 2.6. Lastly, the chapter is concluded by discussing related works in Section 2.7.

2.1 Parallel Data Processing System

Typically, the analysis of large data is often complicated, time consuming, and demands high computational resources [74]. The high speed at which these data are generated requires the data to be processed at a fast rate. It also requires the correct tools and techniques to be analyzed and processed efficiently [75]. With Apache Spark and Apache Hadoop, this process can be simplified through both parallel and distributed processing [74]. The main reason why Apache Spark and Apache Hadoop were created is that they can facilitate the larger volumes, velocity, and variety of data for collecting, storing, and processing of those large data [74].

Parallel processing typically uses the combination of two or more processors to attempt to solve a single issue. Parallel processing is dependent on the distribution of the processing task among multiple processor(s) cores that operate simultaneously [76]. Thus, parallel processing uses a set of simultaneously executing processes that communicate internally to achieve a common objective [76]. The purpose of parallel processing is to execute the codes efficiently, as this allows the execution of applications with a shorter clock time and

provides concurrency, especially when performing multiple tasks simultaneously. It also enables better resource management. While a single computer possesses limited memory resources, this problem can be easily solved by using multiple computers, as this allows larger and more complex problems to be solved [76].

Apache Spark and Apache Hadoop are both open-source parallel and distributed processing frameworks for large-scale data processing, though there are some major differences between them. While Hadoop typically uses MapReduce to process data, Apache Spark uses resilient distributed datasets (RDDs) for the same task [73]. Hadoop contains a distributed file system called HDFS, which allows data files to be sorted across multiple machines [50]. On the other hand, Spark does not provide any file system for distributed storage, which is why Spark mainly reads and then processes data from other file systems [73]. HDFS is one of the file systems that Spark supports [73].

The primary difference between Spark and Hadoop can be seen during their performance. While Hadoop is well-suited for batch processing, it is poor at iterative processing, and Spark was created to solve this problem. The Spark program repeatedly runs ~100 times faster than Hadoop in memory and ~10 times faster on disk [1]. Spark's speed relies on its in-memory processing [1]. On the other hand, Hadoop writes data on a disk that is read on the next iteration, which causes Hadoop to be significantly slower than Spark [73]. Spark was created to be faster and more efficient to avoid the limitations of Hadoop [73]. It is not only faster, but it also offers in-memory processing and contains several libraries that are built on top of it to accommodate large data analytics as well as machine learning [1]. That is the reason why Apache Spark is used in this research.

2.2 Apache Spark

Apache Spark is a fast, scalable, and reliable parallel processing framework for large datasets and machine learning workloads. It has become Apache's most active open-source project, with more than 1,200 active contributors from over 300 companies [1]. It is used by many companies to improve their user services. Some of the use cases of Apache Spark in real life are e-commerce, healthcare, social media, and entertainment. Many companies such as eBay, Alibaba, Netflix, MyFitnessPal, and Pinterest use Apache Spark to enhance their user services [78]. For instance, Netflix uses Apache Spark to recommend content based on the user's previous watch history [78].

Spark processes large datasets with high-level, relatively easy-to-use APIs [2]. Spark combines clusters of machines with a model for writing programs, which allows the logic of data transformations and machine learning algorithms to be written in a way that is parallelizable [2]. Spark is also called a hybrid framework because of its support for both batch and stream processing [3]. Spark runs on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud [1] with its own scheduler. Spark provides APIs in various high-level programming languages such as Scala [89], R [90], Python [64], and Java [91] [1], which makes it useful for solving data science and engineering problems.

Spark focuses on speeding up the application by offering full in-memory computation, low latency, and high-level APIs and tools. Despite using a similar principle as Hadoop's MapReduce engine, this in-memory computation allows Spark to perform faster than its competitive batch processing framework, Hadoop. Spark interacts with the disk only for initially loading the data into the memory and storing the final results at the end [3]. All other intermediate results are processed in memory [3].

2.2.1 Apache Spark Components

Apache Spark Core is the base component of the Spark ecosystem. Figure 1 depicts the Apache Spark ecosystem. All the functionalities of Apache Spark are built on Spark Core e.g., in-memory computation, fault recovery. Spark Core is embedded with a unique data abstraction layer using Resilient Distributed Datasets (RDDs) and a scheduling layer using a directed acyclic graph (DAG).

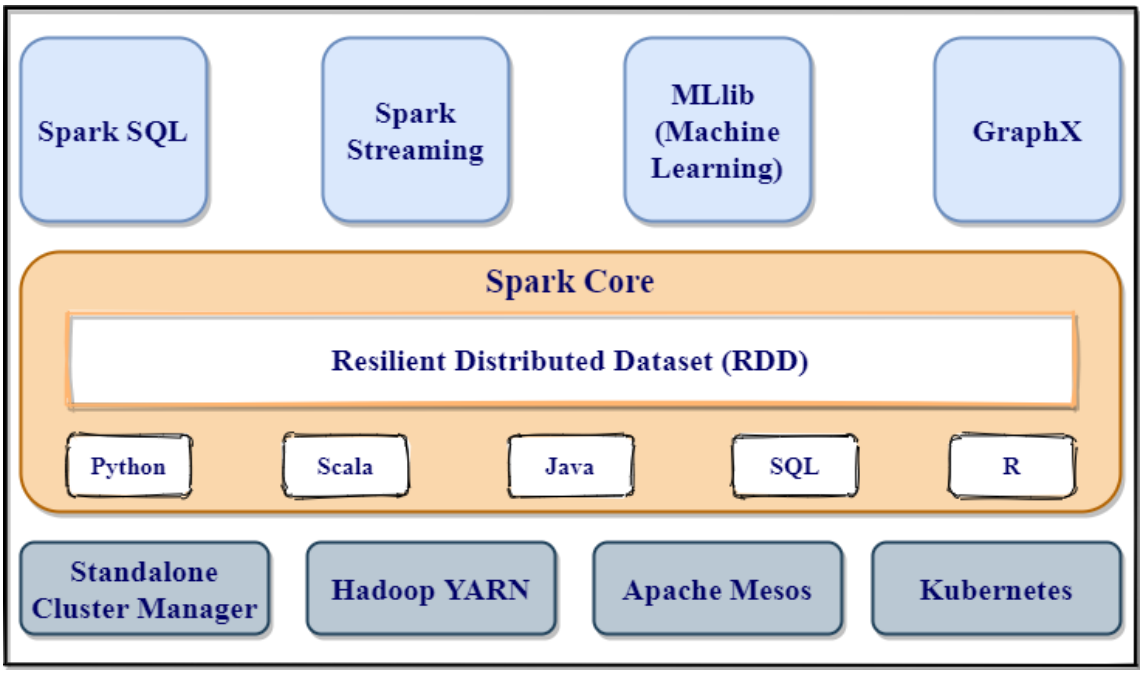


Figure 1: Apache Spark Ecosystem

- *Resilient Distributed Datasets (RDDs):* RDDs are the main read-only data structure for Apache Spark that exists within memory [2]. Spark represents large datasets as RDDs [2]. RDDs are distributed collections of objects, and the objects that comprise RDDs are called partitions [2]. Through RDDs, Spark eliminates expensive intermediate disk writes, which makes Spark a fault-tolerant framework [3]. RDDs are a representation of the lazy evaluation in Spark [2]. Transformations and actions are two types of operations on RDDs to process data and produce new

RDDs [37]. These transformations make Spark lazy in nature, meaning that when a transformation operation calls on an RDD, it does not execute immediately [37]; it creates a new RDD based on an existing one and does not return any value. Different transformations perform different activities on each element of an RDD. Some transformations such as 'map' and 'filter' do not require shuffling or reorganizing data between RDD partitions, as each partition can process independently. The Spark transformation function 'map' takes in any user-defined or built-in function and applies that function to every element of an RDD [37]. The Spark 'filter' transformation function returns a new RDD based on the filter conditions [37]. However, Spark has some transformations such as 'groupByKey' and 'reduceByKey' that require information from other partitions [37]. To implement these transformation functions, Spark performs shuffling by moving data across the cluster and collects the necessary data from each partition to form a new partition. When an action is called, these transformations get executed, the data are processed, and the result from RDDs is returned [37]. Examples of Spark actions include 'collect', 'reduce', 'take', and 'foreach' [37]. The action 'collect' is a common operation that returns the contents of the entire RDD to the driver [37]. The action 'reduce' aggregates the RDD contents using a function [37]. Figure 2 depicts the transformation and action operations on RDDs.

- *Directed Acyclic Graph (DAG):* Spark achieves high speeds in executing tasks by generating a DAG in advance, which implements stage-oriented scheduling. The DAG performs a sequence of computations on data, where each node is an RDD partition, and the edge is a transformation operation on the data [2].

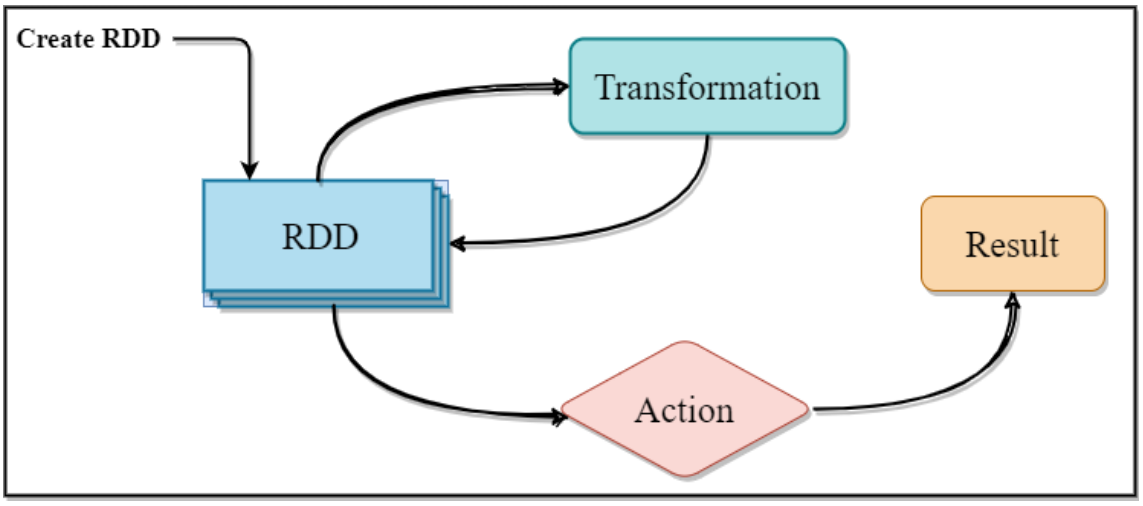


Figure 2: RDD Workflow

In addition to Spark Core, the Spark ecosystem includes several components such as Spark SQL, Spark Streaming, Spark MLlib, and GraphX [1] that allow Spark to provide a more specific data-processing functionality. The key components are described next.

- *Spark SQL*: Spark SQL is a component that provides a structured data-processing abstraction. It enables a powerful and analytical application for both chronological and streaming data that can be accessed from a variety of data sources such as JSON (JavaScript Object Notation), Parquet, and Hive table [3]. Spark SQL provides DataFrame and Dataset data abstractions in Apache Spark, which helps Spark execute SQL queries on these data sources [86]. A DataFrame is a distributed collection of data organized into columns with columns' names and type information. It is conceptually equivalent to the relational database [86]. A Dataset is an extension of DataFrame, which is also a distributed collection of a data structure with a collection of strongly typed objects that map to the relational schema [86].

- *Spark Streaming*: Spark streaming is a scalable, high-throughput, and fault-tolerant stream-processing extension of Spark Core that processes the data generated by various sources in real time [84]. Examples of these data include log files, sensor data, and status updates posted by users on social media. In this research, Spark Structured Streaming has been used, which is a stream-processing engine built on the Spark SQL [1] (discussed in Section 2.4).
- *Spark MLlib*: Spark MLlib is Spark's machine learning library comprised of common machine learning algorithms and high-level tools. [85]. Apache Spark MLlib is further discussed in Section 2.3.
- *GraphX*: GraphX is a graph-processing engine built on top of Spark Core. It is not a stable component of Spark. It allows Spark to build, transform and execute graph-structured data at a large scale [2].

Basic terms used to refer to Apache Spark cluster concepts are:

- *Driver Program:* The driver program is the process running the main function of the application [42].
- *Cluster Manager:* A cluster manager is responsible for acquiring resources on the cluster [42].
- *Worker Node:* A worker node, also known as a slave node, runs application code in the cluster [42].
- *Executor*: An executor is a process initiated on a worker node to run tasks and store data in memory or disk storage [42].
- *Job*: A job is a parallel computation comprised of multiple tasks that gets initiated when a Spark action operation is called [42].

- *Task:* Each job is divided into smaller tasks. A task is a unit of work executed by an executor [42].
- *Stage:* A stage is a set of tasks that depend on each other [42]. A new stage is formed when Spark performs a shuffle between RDD partitions.

2.2.2 Apache Spark Architecture

Apache Spark uses a master/slave architecture containing one master node and multiple worker (slaves) nodes [3] (see Figure 3). A central coordinator of the Apache Spark architecture called a driver runs in the master node [3]. Spark driver is the program that calls the main program of an application and is responsible for creating a Spark context [3]. The driver program translates the application into actual Spark jobs, which are further split into multiple smaller tasks [3]. Then, the driver program schedules the job execution on the cluster and negotiates with the cluster manager for the resource allocation [3].

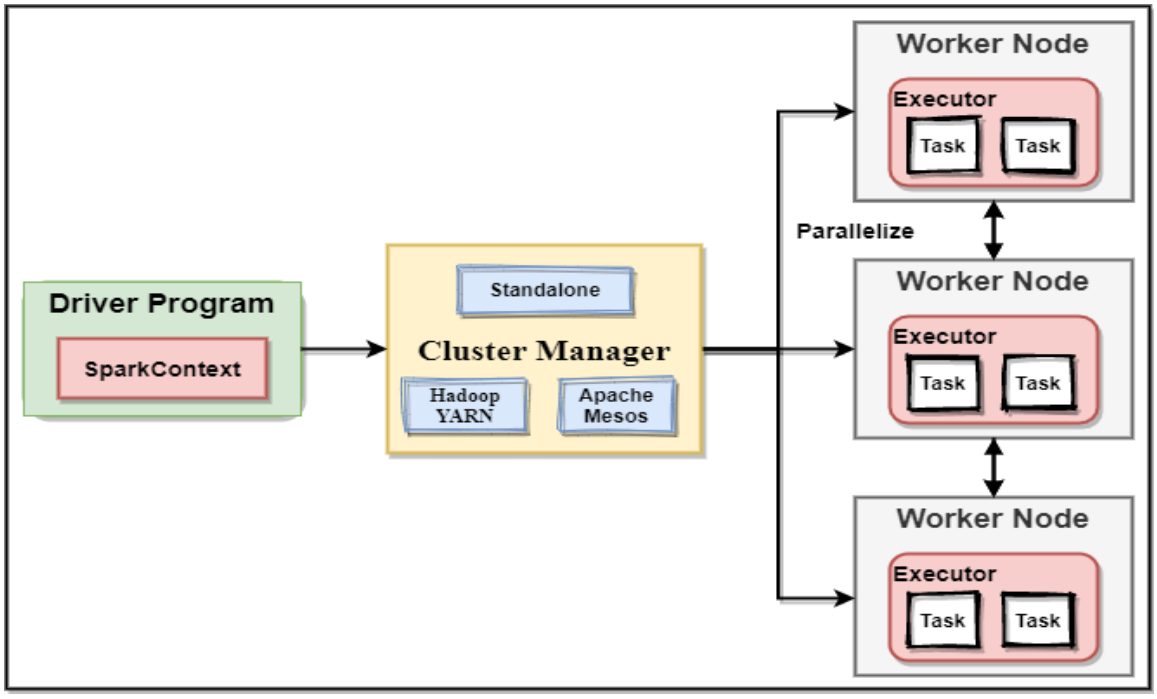


Figure 3: Apache Spark Architecture

Spark context is the main entry point to access all the Spark functionalities [42]. It is used to create RDDs that can be distributed across worker nodes and cached there. Spark context allows the driver program to access the Spark cluster with the help of a cluster manager. Spark context can connect to various kinds of cluster managers such as YARN [79], Mesos [80], Kubernetes [81], or its own standalone cluster manager [45] [42]. Cluster managers are responsible for allocating resources on the Spark cluster and distributing tasks to worker nodes [42]. The standalone cluster manager is the default built-in resource manager of Spark [45] that has been used for this research. With a standalone cluster manager, one executor can run on each worker node, and an application can get all the cores available in the cluster by default [45]. All the resource management and job scheduling are taken care of by Spark itself in a standalone cluster manager [45].

Worker nodes are responsible for executing the tasks assigned by the cluster manager. A set of processes called executors reside inside each worker node. An executor executes these tasks on the partitioned RDD, performs operations and returns the result to the Spark context. Executors are launched once at the beginning of a Spark application and then remain active for the duration of the application. This is referred to as static allocation. However, if the Spark application is configured to execute dynamically, executors can be added or removed by the cluster manager dynamically to match the overall workload. The number of concurrent tasks an executor can run is defined by the number of executor cores. With the increase in the number of worker nodes or executor cores of worker nodes, more RDD partitions can be executed in parallel, which will make the application faster.

2.2.3 Run-time Architecture of a Spark Application

When a user's application code has been submitted to Spark, the driver program converts the application code containing transformations and actions into a logical DAG [2]. The driver program creates all the RDDs and performs no operations on data until the action is called. At this stage, the driver program also performs pipelining transformations. The logical DAG is then converted to a physical execution plan or Spark jobs through a set of stages. In each stage, the driver program creates small physical execution units by splitting up the jobs, which are referred to as tasks. Figure 4 depicts the run-time architecture of a Spark application.

At this point, the driver program connects with the cluster manager and negotiates resources. When the cluster manager runs the executors on the worker nodes, they register with the driver program. The driver program then sends the tasks to executors through the cluster manager. At this stage, the user's application code starts executing. The Spark application performs all the Spark transformation operations in parallel on each partition,

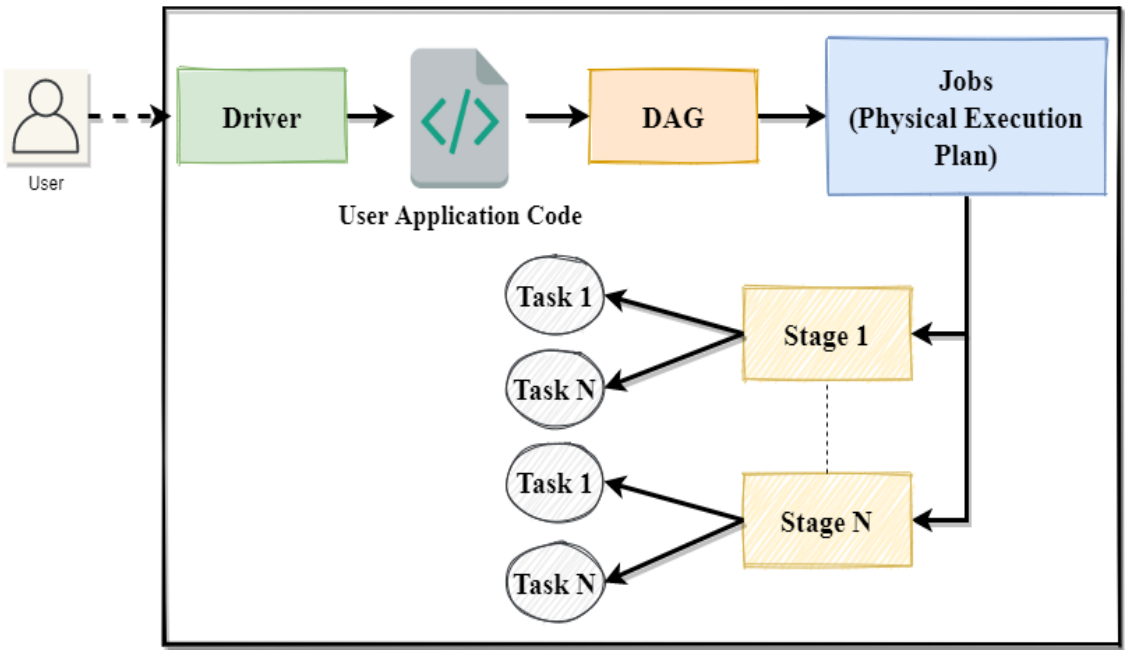


Figure 4: Run-time Architecture of a Spark Application

which derives new RDDs from the existing RDDs. When the Spark application performs an action on the data, it will trigger the entire DAG execution. After finishing the execution, results are sent back to the driver program for compilation. The driver program then calls the stop() method of the Spark context to terminate all the executors and release the resources from the cluster manager.

2.3 Apache Spark MLlib

Spark MLlib is the distributed machine learning library that provides large-scale learning settings to store and operate on data or models [5]. Spark MLlib is comprised of fast and scalable implementations of standard machine learning algorithms including classification, regression, collaborative filtering, and clustering [5]. It provides tools such as pipelines for constructing and evaluating ML pipelines, featurization for extracting features, transformation and selection, utilities for statistics gathering, linear algebra and data handling, persistence for saving and loading models and pipelines [85]. These algorithms are implemented as Spark operations performed on RDDs. Spark MLlib offers a set of multi-language tools at a high level. Among them, Pipeline [32] and K-fold Cross-Validation [87] are used in this research.

- *Pipeline:* In machine learning, Pipeline is a common workflow to use for running a sequence of algorithms involving a sequence of data pre-processing, feature extraction, model fitting, and validation stages [5]. Most machine learning libraries do not support various sets of functionalities required for this kind of workflow [5]. Spark MLlib includes a high-level API built on top of the DataFrames known as Pipeline to address this concern [5]. A Spark MLlib Pipeline consists of a sequence of pipeline stages to be run in a specific order to process and learn from data. This

sequence of stages is either a transformer or an estimator [32]. A transformer is an algorithm that calls the transform() method to transform one DataFrame into another DataFrame. An estimator is an algorithm that calls a fit() method to fit on a DataFrame to produce a transformer [32]. The input DataFrame is transformed as it passes through each stage. If the pipeline had more estimator stages, after producing the transformer by calling the fit() method, it would call the transform() method of the transformer on the DataFrame before passing the DataFrame to the next stage. For instance, a learning algorithm such as Logistic Regression is an estimator and calling fit() trains the Logistic Regression algorithm with a dataset to create a Logistic Regression model that is a transformer [32].

- *K-fold Cross-Validation:* K-fold Cross-Validation is a re-sampling method used to evaluate the machine learning model if the input data size is small [82]. Spark's K-fold Cross-Validation initiates by dividing the dataset into k folds, which are used as separate training and test datasets [87]. It runs the machine learning algorithms with different combinations of parameters and training and testing datasets to find the best model [82].

2.3.1 Machine Learning Algorithms

In this research, five different types of classification algorithms: Multinomial Naïve Bayes, Multinomial Logistic Regression, Multilayer Perceptron, Decision Tree, and Random Forest Classifier of Spark MLlib have been trained with the text classification dataset (this is discussed in Section 4.4). Among them, Multinomial Logistic Regression is used in this research, as it demonstrates the highest accuracy (discussed further in Section 4.4).

2.3.1.1 Multinomial Logistic Regression

A multinomial logistic regression [33] is a supervised classification algorithm that is used when the dependent variable is categorical with more than two classes (labels). It predicts the outcome by explaining the relationship between the class and the features extracted from the input data. Multinomial logistic regression calculates the probability of each specific outcome of the dependent variable using a linear combination of the features and parameters [34]. Multinomial logistic regression generates a matrix of dimension K×N, where K is the number of outcome classes, and N is the number of features [33]. While the algorithm is fit on a dataset with an intercept term, the intercepts are grouped into K-length vectors [33]. This is a multi-equation model, and it uses the SoftMax function to calculate the conditional probabilities of the outcome classes: [33].

$$P\left(Y = k \middle| X, \beta_k, \beta_{0k}\right) = \frac{e^{\beta_k . X + \beta_{0k}}}{\sum_{k'=0}^{K-1} e^{\beta_{k'} . X + \beta_{0k'}}} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [1]$$

where $P(Y=k \mid X, \beta_k\ \beta_{0k})$ is the probability of the class k, K is the total number of classes, X is the feature vector, β_k is the vector of weights or regression coefficients of X for the kth class, and β_{0k} is the bias. The denominator of the function normalizes the probabilities over all classes, ensuring that the sum of the probabilities is 1.

2.3.1.2 Multinomial Naïve Bayes Classifier

A multinomial Naïve Bayes classifier is a probabilistic and multiclass classification algorithm based on Bayes' theorem [35]. It applies the multinomial distribution on each feature and computes the conditional probability distribution of the label for the given features by applying Bayes' theorem for prediction. All Naïve Bayes classifiers make a "naive" assumption about conditional independence features. A naive Bayes classifier

considers each of these features to contribute independently and equally to the decision, regardless of any possible correlations between features.

2.3.1.3 Multilayer Perceptron Classifier

A multilayer perceptron classifier [36] is a feedforward artificial neural network that consists of multiple layers of nodes. The nodes in an input layer are for input data, whereas the nodes in the output layer are for providing predictions about the input data, and multiple hidden layers are placed in between the two layers for computation. Each layer is connected to the next layer of the network [36]. Nodes in the middle and output layers map inputs to outputs by using a linear combination of the inputs with the node's weights and bias and applying an activation function [36].

2.3.1.4 Decision Tree Classifier

A decision tree classifier [43] is one of the machine learning algorithms that is widely used because of its adaptation with categorical features. It is a greedy algorithm that divides the feature space by performing recursive binary partitioning [43]. This classifier predicts the label for each leaf partition [43]. It chooses each partition greedily by selecting the best split from a set of possible splits to maximize the information gain at a tree node [43].

2.3.1.5 Random Forest Classifier

A random forest classifier [44] is an ensemble learning method used for classification that combines many decision trees for classifying purposes. This classifier creates a set of decision trees by selecting subsets of training sets randomly. Then, it trains these decision trees separately by injecting randomness into the training process. This classifier gathers the final class label of the testing data from each decision tree. The testing data are assigned

to a class that is predicted by the maximum number of decision trees. Like decision trees, it is one of the most popular machine learning models for handling categorical features and adapting multiclass classification settings.

2.4 Spark Structured Streaming

Spark Structured Streaming is a stream-processing engine with fast, fault-tolerant, and scalable stream processing built on the Spark SQL engine [84]. It has been built by combining batch and streaming computation to simplify the development of continuous and real-time applications. Spark Structured Streaming processes a data stream as a series of small batches using a micro-batch processing engine, which allows it to achieve low end-to-end latencies [84]. The micro-batch processing engine groups the live data into small batches. The key idea in Spark Structured Streaming is to handle a live data stream as a table [84]. Every data item on a small batch that is arriving as a stream is appended to the input table as a row [84]. Spark uses the already existing Dataset and DataFrame (discussed in Section 2.2.1) data structure for the input table [83].

There are a few built-in input data sources available such as the file source, which reads files written in a directory as a stream of data; the Kafka source, which reads data from Kafka; and TCP Socket, which reads UTF8 text data from a TCP socket connection [84]. Apache Kafka has been used as a stream data source in this research. Figure 5 shows Spark Structured Streaming with Kafka source (discussed in more detail in the following section). Spark can access stream data from these sources to create the DataFrames/Datasets. Depending on a user-defined trigger interval, the input table is being appended with new data items arriving on the stream on which queries are run and results from the queries are saved on the result table [83]. There are several output modes available on Spark Structured

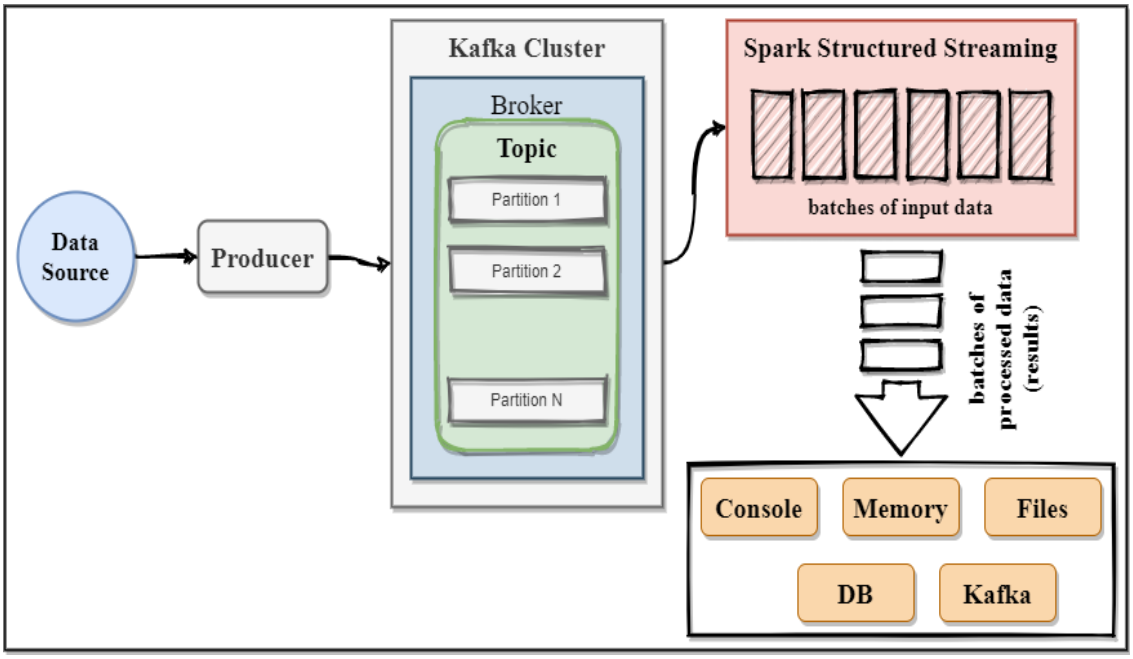


Figure 5: Apache Structured Streaming with Kafka

Streaming to use to write the changed result rows to an output sink [83]. The output modes define how the result table is written to the output sink. There are three types of output modes available on Spark: append mode, which is the default mode and only writes the new rows that arrive in the last batch to the output sink; complete mode, which writes the whole result table to the sink after each trigger; and update mode, which writes only those rows that are changed from the last trigger [84]. There are a few types of built-in output sinks to store the result rows: File sink to store the output in a directory, Kafka sink to store the output in one or more topics of Kafka, and ForeachBatch sink to specify a function to perform operations on each micro-batch of the streaming query [84]. Append mode as the output mode and ForeachBatch as the output sink have been used in this research.

2.5 Apache Kafka

Apache Kafka is an open-source distributed data streaming platform used by many companies to publish, subscribe, store, and process streams of records in real time [40].

Kafka is a fast, scalable, and durable publish-subscribe-based messaging system that can handle data streams from multiple sources and deliver them to multiple consumers. It is designed for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications [40]. Kafka acts as a middle layer that is built into streaming data pipelines and allows a user to send messages between systems and/or applications in distributed systems; it is also built into the systems and applications that consume that data [40]. Thus, the main idea is that the sender, who is known as a producer, sends messages to the Kafka server, and the receiver, who is known as a consumer, receives only messages that the receiver is interested in (as specified through Kafka topics) from the Kafka server. The main concepts underlying the Kafka architecture are topics, partitions, brokers, ZooKeeper, producers, and consumers. Figure 5 shows the Kafka architecture as a source of Spark streaming.

- *Kafka Topics:* A Kafka topic is a category name that is unique across the Kafka cluster. Each producer publishes their data to the respective topics, while the consumers read messages from their subscribed topic [40]. Hence, Kafka stores and organizes data across its topic.
- *Kafka Partitions:* Inside the Kafka cluster, topics are separated into individual partitions, and partitions are then replicated among the brokers [41]. This allows partitions to parallelize a topic across multiple Kafka brokers or Spark clusters. Multiple consumers can read from a single topic parallelly within each partition. The producers add a key to messages, where all messages containing the same key will go to the same partition. On the other hand, messages that do not contain any

key are usually written to partitions in a round-robin manner [41]. There is usually no limit on how many Kafka partitions can be created.

- *Kafka Brokers:* A Kafka cluster typically contains several brokers. These brokers usually work together to build the Kafka cluster, which helps achieve a balance in load [40]. For proper management and coordination of the cluster, the brokers usually utilize the Apache ZooKeeper system [41]. Every one of these brokers typically handles hundreds of thousands of read and write quantities every second [41]. Every one of these brokers has a unique ID that separates one from the other. Kafka brokers utilize ZooKeeper to elect a broker as a leader to handle client requests for an individual partition of a topic [41]. A minimum of three brokers needs to be utilized to achieve a reliable failover, where the reliability will significantly increase with a higher number of brokers [41].
- *Apache ZooKeeper*: ZooKeeper [95] is used by Kafka brokers to handle and coordinate the Kafka cluster. Whenever any broker and topic are being added or eliminated, ZooKeeper notifies all nodes [41]. Moreover, ZooKeeper also helps determine which broker will take the lead for a specific partition and which ones will hold replica(s) of the same data [41]. As soon as ZooKeeper notifies the clusters regarding the broker change(s), the brokers start to communicate with one another and choose a new partition leader, if needed [41]. This process helps protect in the event that a broker becomes absent.
- *Kafka Producers*: Kafka producers are external applications that write and publish messages to one or more Kafka topics. In addition, these producers also do the job

of serializing, compressing as well as load balancing the data amongst other brokers via partitioning.

- *Kafka Consumers*: Consumers read messages from their subscribed topics. These consumers belong to their respective consumer group, where each of these individual consumers maintains a responsibility to read a subset of the partitions from each subscribed topic.

2.6 A Python Library spaCy

spaCy is a pre-trained, free, open-source library for natural language processing (NLP) in Python that can analyze, identify, and derive meaning from human languages for computers [38]. It is used as an information extraction technique that automatically identifies named entities in a text and classifies them into predefined categories [38]. This information extraction is called named entity recognition (NER). Named entities are defined as objects of interest such as a person, organization, location name, date, time, percentage, or money. spaCy uses a lightweight deep learning library called Thinc as its backend [39]. Using Thinc, spaCy includes convolutional neural network models for NER, part-of-speech tagging, dependency parsing, and text categorization.

2.7 Literature Review

The proposed research focuses on a user preference-based filtering technique. Both parallel processing and machine learning to classify text datasets based on user preferences are used. In this research, the parallel processing engine Apache Spark's machine learning library and Python libraries are used for text classification to filter user-preferred data. Various researchers have published works on the text classification process using machine learning algorithms in general and the machine learning library of Apache Spark in

particular. To the best of my knowledge, little work exists on user preference-based filtering of data.

2.7.1 Text Classification using Machine Learning

Text classification is an important research orientation in the fields of information retrieval and data mining, as it has extensive applications in practical work and scientific research. Text classification is the process of categorizing texts in order to assign labels or classes to textual units such as sentences, documents, and paragraphs according to their content [30]. It has widespread applications in the real world such as sentiment analysis, topic classification, answering questions, and the categorization of news articles and academic papers [30]. Texts can be classified either manually or using automatic labeling techniques. In recent years, with the growing scale of text data, automatic text classification using machine learning-based methods has started receiving attention from researchers [30]. However, finding suitable machine learning algorithms for text classification has been a challenge for researchers [30].

In the research presented in [6], the authors used different machine learning techniques to classify research paper abstracts from the fields of science, business, and social science. The machine learning techniques employed in this research [6] are Support Vector Machine, Naïve Bayes, K-Nearest Neighbor, and Decision Tree. The experimental results showed that the Support Vector Machine achieves higher accuracy, precision, recall, and F1-score than the other methods, particularly the Decision Tree method, while K-Nearest Neighbor and the Naïve Bayes method do relatively better than the Decision Tree method [6]. The Naïve Bayes classifier is one of the classical and most commonly used machine learning algorithms for text classification [8]. In the research presented in [8], the authors

proposed a new classification algorithm based on a Naïve Bayes classifier using the Laplace distribution. Three different sizes of English movie review datasets were used for classification in [8]. The research in [8] showed that in the classification of those three different datasets, the accuracy of the proposed method is higher than the accuracy of the classical Gaussian Naïve Bayes classifier. While explaining the drawbacks of Hadoop MapReduce in performing text classification, the authors in [7] argued that their proposed machine learning approach to classifying text data is less time consuming than that achieved with Hadoop MapReduce. Classification in Hadoop uses K-Means Clustering, which requires a large amount of time to perform the classification, thereby increasing the latency [7]. Motivated by this, the authors in [7] proposed a machine learning method based on a Naïve Bayes classifier. A medical dataset was used for classification. Based on the disease, the proposed method checks for a class label that indicates whether a person is suffering from a disease or not [7]. The research presented in [28] proposed a novel classification method by improving the Naïve Bayes algorithm based on Improving Term Frequency–Inverse Document Frequency (ITF-IDF). The proposed ITF-IDF is based on the traditional Frequency–Inverse Document Frequency (TF-IDF) algorithm, which is a classical feature weight calculation method [28]. In the proposed method, the authors combined the TF-IDF algorithm with the characteristics of text classification [28]. This method categorizes text data from the power system [28]. The experimental result showed that the proposed method can achieve a higher classification accuracy than traditional Naïve Bayes, Logistic Regression, and Support Vector Machine methods using the same datasets [28]. In recent years, deep learning-based models such as Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and Long Short-Term

Memory (LSTM) have become more commonly used than classical machine learning-based approaches [30]. In the research presented in [9], the authors proposed a novel model to categorize news articles from news headlines and short text descriptions of the news. The authors used LSTM and Gated Recurrent Unit (GRU) as their proposed model's backbone. The authors stated that the accuracy of their proposed model showed an average performance in categorizing news articles, but this could be improved by applying other data optimization algorithms [9]. In the study presented in [10], the authors classified research articles based on a Bidirectional Encoder Representations from Transformers and Bidirectional Gated Recurrent Unit (BERT-BiGRU) model. The authors applied the BERT model to word vectorization based on feature extraction and used the bidirectional GRU Neural Network model as the classification model. According to this study's results, the algorithm can significantly enhance the accuracy of text classification in comparison with the commonly used Neural Network and BERT model [10]. The authors in [11] proposed a model to perform deep sentiment analysis for documents using a CNN and an LSTM network. The authors performed the experiments on two movie review datasets. The experimental results showed that the accuracy of the proposed model is better than traditional machine learning methods such as Support Vector Machine, Multinomial Naïve Bayes as well as the standard Neural Network model on the sentiment classification dataset [11].

2.7.2 Text Classification using Parallel Processing Engine

Several research works are being carried out on using parallel algorithms for text classification. In the research presented in [12], the authors used the MapReduce-based Rocchio relevance feedback algorithm for document classification. The authors showed

that traditional text classification algorithms such as K-Nearest Neighbor, Support Vector Machine, and plain Bias guarantee better accuracy by compromising the processing speed [12]. The dataset used for the experiments was earthquake information source text for Beijing. The experiments showed that the proposed solution based on the Rocchoi algorithm with MapReduce technology improved the speed of processing massive amounts of data and guaranteed better accuracy [12].

2.7.3 User Preference-Based Filtering Technique

User preference-based filtering is a popular approach that filters data based on user preferences to help users find their desired information in a shorter period. Several research works are being conducted to retrieve user-preferred data from various sources by using the user's preferences. The research presented in [14] introduced an integrated technique for user-filtering and query-refinement to retrieve user-preferred music. The authors in [15] refined the search results of the users based on a group of users' past searching behavior using an efficient Bayesian filter. The study in [29] showed that accurate recommendations for online advertisements are posing a major challenge to advertisers and agencies. The authors proposed a personalized advertisement recommendation system by analyzing the search engine data of users. The proposed personalized advertisement recommendation method can extract the user's preferences using the Latent Dirichlet Allocation (LDA) model and can generate a recommended list of advertisements based on the nearest neighbor and user behavior [29]. The authors in [13] built a system that shows the user specific content based on the profile created for a user. The authors used a MapReduce-based vector space model for user profiling [13]. The proposed model creates a user profile based on the news items read by the user [13]. The authors define user profiling as a process

of learning and analyzing the demands of a user from the data linked with that user [13]. The efficiency of the proposed system is achieved by using the MapReduce programming concept for large-scale data [13].

2.7.4 Searching within Large Datasets

Numerous research works have been conducted on developing scalable and efficient ways to search keywords or sentences within large datasets. The authors in [26] introduced a new graph-based indexing (GBI) technique that uses a directed graph structure to efficiently capture the concurrent occurrence of multiple keywords in the same document. The relationship among the search keywords captured in the graph structure was used to successfully retrieve all results of Boolean AND queries at once. The analysis of this research showed that for the set of data they experimented with, their proposed technique could improve the search latency when executing Boolean AND queries by an average of 69% to 99.9% in comparison to conventional inverted index-based techniques [26]. In [27], the authors introduced a document management system (DMS) that can index and search documents as well as perform full-text searches within a document using Apache Lucene. The results provided in [27] demonstrated that the proposed searching system can help users search efficiently and retrieve data promptly. In the research presented in [25], the authors addressed the challenge of building an efficient and scalable experimental search engine using an in-memory distributed big data processing framework called Apache Spark. They proposed an information retrieval engine called SparkIR that is built on top of Apache Spark. The proposed information retrieval system leverages Spark's in-memory processing by keeping the index in memory in the data retrieval stage and thus handles queries efficiently [25]. The authors evaluated the performance of SparkIR by conducting

experiments on different subsets of a ClueWeb12-B13 collection that contains approximately 50 M web pages [25]. They showed that SparkIR exhibits reasonable overall efficiency and scalability for both data indexing and information retrieval [25].

2.7.5 Research on Apache Spark

In recent years, Apache Spark has become one of the most popular data processing frameworks that is able to overcome Hadoop's shortcomings in implementing traditional data mining and machine learning algorithms. Several research works have been conducted in recent years based on Apache Spark's parallel processing engine and machine learning library (MLlib), both of which have been used in this thesis. A representation set is discussed in this section.

According to the research presented in [16], users use the Internet every day, from shopping for groceries to managing bank accounts. With this high usage, the potential risk of malicious users attempting to access users' information increases [16]. An intrusion detection system (IDS) is one way to tackle this problem, as it allows large amounts of network data to be processed accurately and quickly [16]. In this research [16], the authors have used Apache Spark's MLlib to classify whether a network packet is part of an attack. The authors have evaluated the performance of Random Forest, Support Vector Machines (SVMs), Logistic Regression, Naïve Bayes, Gradient Boosted Trees, and Multilayer Perceptron in the design of the Apache Spark-based IDS [16]. The results show that the Multilayer Perceptron algorithm produces favorable accuracy, precision, and recall, but it takes longer to analyze data than other machine learning algorithms. The authors in [17] developed five models using distributed machine learning based on Apache Spark for predicting diabetes. Among those models, the Logistic Regression Classifier achieved the

highest accuracy. Four MLlibs of Apache Spark have been evaluated to provide an effective solution for the prediction of heart diseases in the research presented in [18]. The authors built a scalable real-time health status prediction system in Apache Spark that is presented in [19]. The proposed technique receives user health data through tweet streams and predicts their health status by applying a Decision Tree model to the data [19]. In the research presented in [20], the authors used the Latent Dirichlet Allocation (LDA) algorithm of Apache Spark's MLlib to extract news topics. This paper included using a crawler to collect a large-scale news corpus on the Internet and preprocessing the data by removing stop words. According to [21], network monitoring systems (NMSs) are an important part of protecting users such as governments and corporate companies against emerging threats. Such organizations can maintain a series of custom regular expression (regex) patterns to run on NMS data [21]. However, with the increasing growth of network traffic, it is difficult to perform these regex operations quickly [21]. In this research [21], the authors proposed a novel algorithm that leverages Apache Spark to perform the regex patterns matching in parallel. The results from the study [21] show that the proposed parallel approach is able to process 1250 events in 1.047 seconds and 2500 events in 1.591 seconds on average using a single 36-core and 117-GB RAM Spark node. In the research presented in [22], the authors proposed an efficient sentiment prediction technique for Twitter data by utilizing different classification algorithms provided in Apache Spark's MLlib. The results indicated a significant improvement in the accuracy of Naive Bayes and Logistic Regression with respect to Decision Trees. The research presented in [23] introduced a new portable and scalable digital eScience toolbox called "defoe" that can extract knowledge from historical data by running text-mining queries across large

historical datasets in parallel using Apache Spark. The experiments were done using five different historical datasets and two different high-performance computing (HPC) environments as well as on desktop [23]. The authors showed that "defoe" can allow researchers to query multiple datasets in parallel from a single command-line interface [23]. The authors in [24] implemented a collaborative filtering (CF) recommender system on Apache Spark that can draw on users' past opinions for the process of prediction generation. This proposed recommendation system uses a web movies dataset from Amazon as a standard dataset [24]. It provides suggestions for improved ratings by evaluating the correctness of predictions in terms of mean absolute error, mean square error, and root mean square error [24].

Studies have been conducted on user preference-based recommendation systems [29], user preference-based music filtering [14], and user profiling based on news items read by the user [13]. The studies presented in [25][26][27] search keywords or sentences within large text datasets. Searching within large datasets is time consuming and gives rise to high processing latencies. More studies should be undertaken to address this gap so that users can search their preferred data from different large datasets in a timely manner. None of these existing studies have leveraged parallel processing with machine learning to filter large-text datasets based on user preferences and performed searches within the filtered data to get user-relevant data. The research presented in this thesis is aimed at addressing this gap. This research investigates techniques for filtering the large datasets based on user preferences using parallel processing and machine learning. It also investigates the degree of latency improvement achieved by using effective search techniques processing the filtered data.

Chapter 3: Proposed Technique

This chapter discusses the proposed technique for filtering and searching user-preferred data. Section 3.1 provides an overview of the components of the technique. Section 3.2 describes the internal details of filtering user-preferred data, and Section 3.3 explains the algorithm of the filtering system. Section 3.4 and 3.5 discuss the internal details and algorithms for searching relevant information within the filtered data. Section 3.6 describes the implementation details for the Filter and Search methods.

3.1 Methodology

The methodology is explained with the help of Figure 6. The main components of the system are the raw dataset, user preferences, filter, filtered data, and user search queries. Each component of the system presented in Figure 6 is described in the following sub-sections.

3.1.1 Raw Dataset

The raw dataset is comprised of data that needs to be filtered based on user-preferred information. The raw dataset can be retrieved when the user files are stored in a local storage system (see Section 3.2.1). It can also be retrieved from the stream source when stream processing is used (see Section 3.2.1). With batch processing, the raw dataset is a collection of one or multiple text files stored in a local storage system. For example, a raw dataset can be composed of multiple meeting-minutes files that are stored in the local storage system. In stream processing, the raw dataset is a collection of records that come from the Kafka stream source. Each record is comprised of a sentence. For instance, the raw dataset can be text sentences converted from an audio recording of a live meeting. The

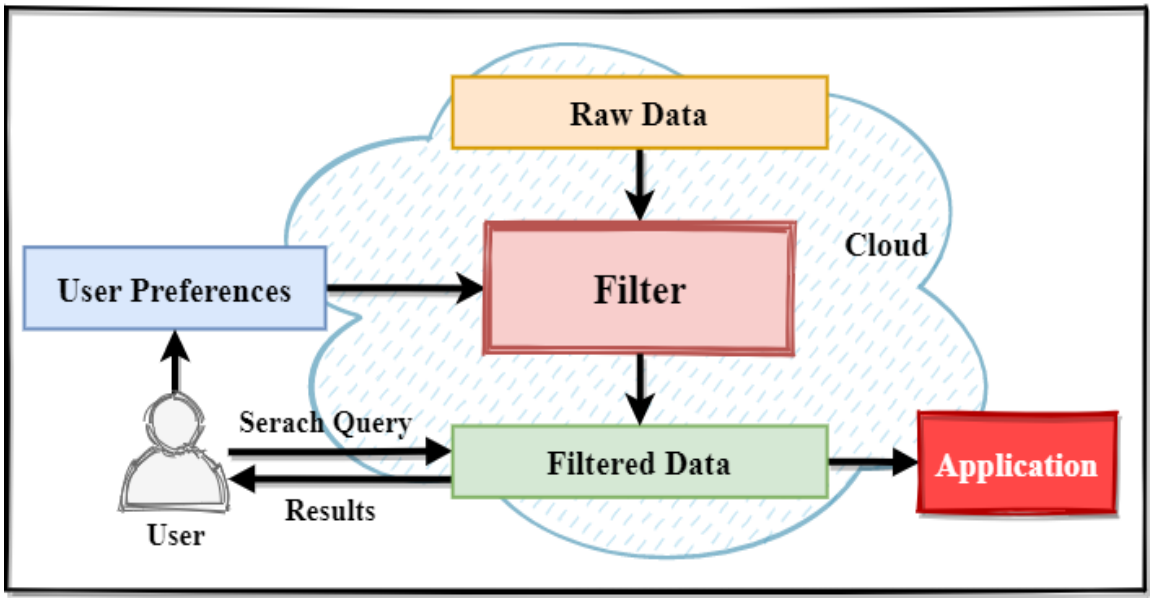


Figure 6: Proposed Technique

filtering system processes the raw text dataset and filters the processed data based on user preferences.

3.1.2 User Preferences

User preferences consist of a set of keywords, each of which describes a topic the user is interested in e.g., dates, locations, people's names, or products. User preferences can either be provided by the user or generated by using a history of previous queries sent by the user. The filtering technique uses the user preferences to filter raw data and store only the preferred data for handling user queries. Consider, for example, a system in which one of the user's interests is "people's names"-related data. Therefore, the user preference is set to "people's names" for the user to filter the raw data and store only "people's names"-related data for handling user queries. The user can query the stored data to get the desired data. The user's queries are used to generate more user preferences later. User queries are further discussed in Section 3.1.5.

3.1.3 Filter

The “Filter” module filters the raw dataset using the user-provided preferences and stores the related user-preferred data. This module is referred to as the “Filter method” in this research. The Filter method is further discussed in Section 3.2.

3.1.4 Filtered Data

The filtered data consist of the data related to user-specified keywords. They are stored in a file after running the filtering method on the raw datasets. The filtered data are the output of the Filter method. The format of filtered data is discussed in Section 3.2.

3.1.5 User Search Queries

A user can search within filtered data to get related data by providing a keyword or a sentence. Consider, for instance, a use case in which one of the user preferences is “people’s names”, and the user wants to see all the data related to a specific name such as “Donald Joe”. The user can search within the filtered data by providing “Donald Joe” as a keyword or by asking a question like “is Donald Joe’s name present anywhere?” A method referred to as the “Search method” is used in this research to handle user search queries. The Search method is discussed in Section 3.4.

3.1.6 Application

This an optional component that can coexist with the component that processes user queries. This is a program that processes filtered data for a specific intent. For example, an application can be developed to generate appointment schedules for the user based on the filtered data related to dates and times. Such an application is discussed in Section 4.5.

3.2 The Filter Method

The Filter method processes the raw dataset and filters the data based on user preferences. This method returns filtered data that contain data relevant to user preferences. The Filter

method requires a raw dataset and user preferences as input. First, it classifies the texts in the raw dataset using the machine learning model and the named entity recognition function. The machine learning model is created by running the machine learning algorithm on the training dataset. The machine learning model and named entity recognition function are discussed in Section 3.2.4. Then, it uses the user preferences to filter the categorized data.

The Filter method's functionality is explained with the help of an example in which the raw dataset contains the following text sentences and preferences provided by the user: "Date" and "Person".

> *Sentence 1:* "An appointment has been scheduled with Dr. John Watson on September 25, 2020 at 9:00 am."
>
> *Sentence 2:* "A nurse will confirm the appointment before the scheduled date."
>
> *Sentence 3:* "The hospital is located in Ottawa, ON."

After receiving the above raw dataset as input, the Filter method processes texts from the raw data and passes the texts as input to the named entity recognition function and the machine learning model. The named entity recognition function extracts the person's name - "John Watson", date - "September 25, 2020" and time - "9:00 am" from sentence 1 and classifies sentence 1 as "Person", "Date", and "Time" (classes). The machine learning model classifies sentence 1 as "Person", "Date". In the case of sentence 2, the machine learning model classifies it as "Person". But the named entity recognition function cannot extract any entities and classifies this sentence because there is no relevant entity e.g., person's name, date, product, etc. in sentence 2. Both the machine learning model and named entity recognition function classify sentence 3 as "Location". Next, the Filter

method compares these classes of each sentence with the user preferences. As mentioned before, the user preferences are "Date" and "Person", which is why sentence 1 and sentence 2 are stored by the Filter method as filtered data. The filtered data (shown in Figure 6) contain the following comma-separated sentences:

Sentence 1: "An appointment has been scheduled with Dr. John Watson on September 25, 2020 at 9:00 am.", "*Person, Date, Time*", "*John Watson, September 25, 2020, 9:00 am*", "*Person*"

Sentence 2: "A nurse will confirm the appointment before the scheduled date.", "*Person*"

The components of the Filter method are described with the help of Figure 7. Each component is described in a separate sub-section.

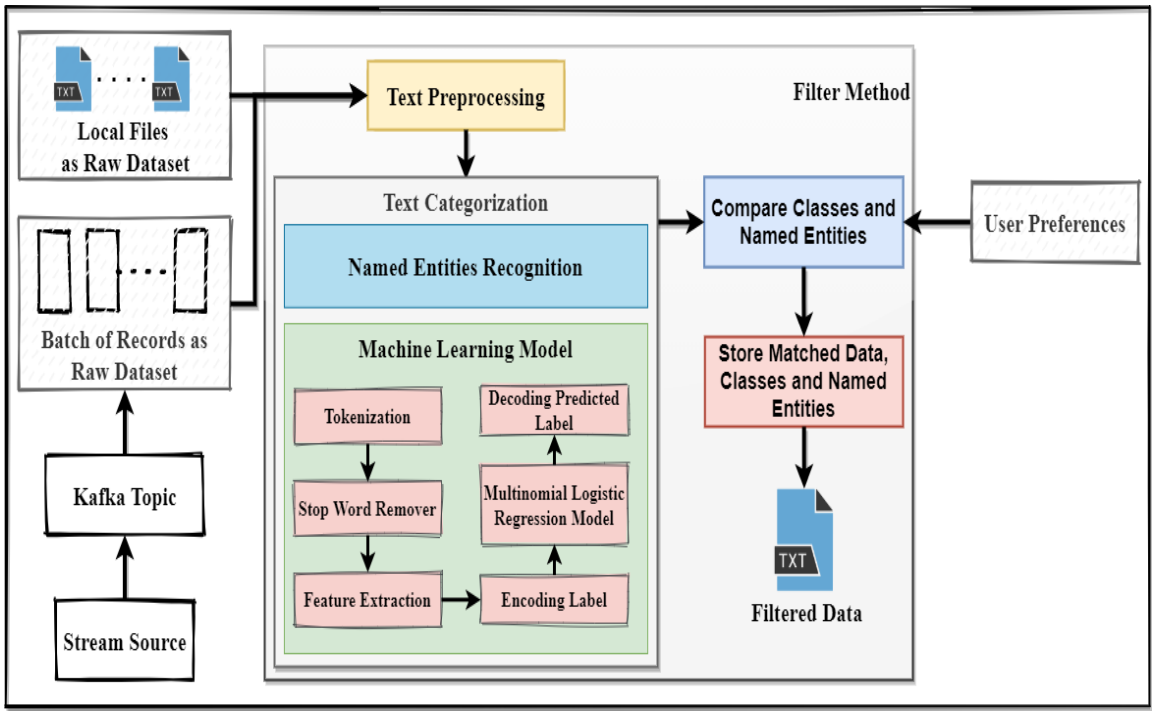


Figure 7: Filter Method

3.2.1 Raw Dataset

The Filter method requires the raw dataset as input. The Filter method can receive the raw dataset as a batch or as stream data.

- *Batch Processing:* For batch processing, the input raw dataset is a local directory where all the raw data files are located. First, the Filter method reads the files from the raw dataset and stores them in memory. Two types of data partitioning strategies (D_P) to distribute the raw dataset among the executor cores have been investigated. One is centralized partitioning (C_{DP}) where each executor core of the worker nodes handles a different number of files of the raw dataset. For C_{DP}, each executor core reads files from memory. As soon as any of the executor cores becomes available, it takes on the next file from the memory. Thus, the data size is not fixed for each executor core in C_{DP}. Another one is equal distribution-based partitioning (E_{DP}) where the raw dataset is distributed among executor cores equally. In E_{DP}, the data size for all the executor cores is equal. Thus, for E_{DP} the Filter method needs to pre-compute the size of the raw dataset and distribute the raw dataset equally among the executor cores. Hence, using C_{DP} as a data partitioning strategy will be more beneficial.
- *Stream Processing:* For stream processing, the raw dataset is comprised of batches of records. Each partition of a Kafka topic receives these batches of records from a streaming source. The Filter method reads the batches of records from each partition and stores them in memory.

3.2.2 User Preferences

Another input required by the Filter method is user preferences. The preferences are provided by the user as a set of comma-separated values. The Filter method also includes

the classes generated from the history of previous queries sent by the user in the user preferences. These user-provided classes (preferences) are used to match with the classes returned by the machine learning model and named entity recognition function (discussed in Section 3.2.5).

3.2.3 Text Preprocessing

The Filter method preprocesses the texts by eliminating punctuation and lowercasing the capital letters in the "Text Preprocessing" function. The preprocessed texts are passed as input to the named entity recognition function and machine learning model.

3.2.4 Text Categorization

The Python library "spaCy" (described in Section 2.6) is used to build the named entity recognition function. It classifies the preprocessed text data and extracts specific entities such as people's names, locations, dates, numbers, and money from the text data. A Multinomial Logistic Regression classifier (discussed in Section 2.3.1.1) of Apache Spark MLlib is used to classify the text according to its content. This classifier has been trained using the labeled dataset obtained from the Cognitive Computation Group of University of Illinois [56][57] (discussed in Section 4.4.1). This classifier of Apache Spark MLlib runs concurrently in multiple executor cores. The classifier runs a sequence of pipeline stages (discussed in Section 2.3). The pipeline stages are tokenization and stop words removal, converting each word into feature vectors, encoding labels to label indices, learning a logistic regression model from feature vectors and labels, and converting the predicted label to the original label. These pipeline stages are discussed next with the help of Figure 7:

- *Tokenization*: Tokenization is a process that takes text (such as a sentence) as input and divides it into a set of meaningful terms (usually words) [53]. A simple Tokenizer class named "RegexTokenizer" of Apache Spark MLlib is used to tokenize the data [53]. RegexTokenizer is a regex-based tokenizer that extracts words using the provided regular expression (regex) pattern [53].
- *Stop Words Remover:* Stop words are words such as "the", "an", and "a" that appear frequently and do not carry as much meaning [53]. These words are useless and should be excluded from the input [53]. Apache Spark MLlib provides a "StopWordsRemover" class that takes the output of a Tokenizer as input and drops all the stop words from the input sequences [53].
- *Feature Extraction:* Feature extraction is a process that reduces an initial set of raw data to more manageable groups for processing [100]. When the given input data of the algorithm is found to be large for processing, and in case they have any redundancies, they are reduced into feature vectors [100]. The large data sets that comprised of large number of variables require a lot of computing resources as well as memory to process [101]. While accurately describing the original data set, it selects and combines variables into features for reducing the amount of data that must be processed [101]. In the context of this thesis, feature extraction derives the numerical features from the raw text data which are usable in machine learning [97]. Thus, each raw text data component needs to be mapped into a vector space that can be used to train the machine learning model [96]. This mapping from the text data space to the real valued vector space is performed during feature extraction. CountVectorizer of Apache Spark MLlib is used for feature extraction.

CountVectorizer converts a collection of text data to vectors of token counts and generates a CountVectorizerModel [53]. The model produces sparse vectors with a length of the entire text data and an integer count for the number of times each word appeared in the text data [53], which are called feature vectors. Inverse document frequency (IDF) of Apache Spark MLlib is used to rescale the feature vectors; this usually improves the performance while using text as features [53]. Then, these feature vectors are passed to the learning algorithm. In this research, Multinomial Logistic Regression has been used as the machine learning algorithm.

- *Encoding Label:* StringIndexer of Apache Spark MLlib is a label indexer that encodes a string column of labels to a column of label indices [53]. It assigns a distinctive integer value to each label. The string labels are ordered in descending order, and the most frequent label is assigned 0, followed by 1, and so on [53].
- *Multinomial Logistic Regression:* The Multinomial Logistic Regression algorithm runs on the feature vectors and encoded labels to build the Multinomial Logistic Regression model. This model predicts the class of the text data and outputs the predicted indices. Multinomial Logistic Regression is discussed in Section 2.3.1.1.
- *Decoding Predicted Label:* IndexToString of Apache Spark MLlib maps a column of label indices back to a new column containing the original labels as strings [53]. It is used to retrieve the original labels of the predicted indices returned by the Multinomial Logistic Regression model.

To evaluate the model, these previously discussed pipeline stages have been run with different combinations of Logistic Regression parameters using Spark's 5-fold Cross-Validation (discussed in Section 2.3). Cross-validation is used to estimate the accuracy of

the machine learning model for the raw dataset. Spark's 5-fold Cross-Validation generates five (training, test) dataset pairs, each of which uses 4/5 of the data for a training dataset and 1/5 for a testing dataset [87]. The Cross-Validator produces 5 models by fitting the pipeline stages on the five different (training, test) dataset pairs. Then, it computes the accuracy of these five models to evaluate a particular set of Logistic Regression parameters [87]. After identifying the best set of Logistic Regression parameters, the Cross-Validator produces a final model by re-fitting the pipeline stages using the best set of Logistic Regression parameters and the entire training dataset [87]. This best pipeline model is used by the Filter method to classify the text data. The machine learning model and the named entity recognition function return the classes of the text data, which are used in the next step to filter the data based on user preferences.

3.2.5 Comparing and Storing the Filtered Data

The classes of the text data obtained from the previous step are compared with the user-specified classes (preferences). Next, the texts whose classes are matched with the user-specified classes are stored as filtered data (see Figure 7). Filtered data are a comma-separated file with the filtered text, entity classes recognized by the named entity recognition function, extracted name entities, and class predicted by the machine learning model.

3.3 Filter Method: Algorithm

The algorithm for the Filter method is described by Pseudo Code 1. This algorithm supports both batch and stream processing. The Filter method takes two inputs: the raw dataset and user preferences as multiple comma-separated values and outputs the filtered data. In line 1 of the pseudocode, the Filter method reads the raw dataset.

Pseudo Code 1: Filter Method

Input: User preferences, P[] and raw dataset, R
Output: User preferred information

```
1:  srdd <- read the raw dataset R.
2:  P <- getUserPrefHistory(P)
3:  data <- srdd.map {
4:      line <- sent_tokenize(line),
5:      line <- preprocessing(line),
6:      line <- NER(line, P)
7:      Row(Text=line[0], NERTokens=line[1], NERData=line[2])
8:  }.toDF()
9:  prediction <- MLPipelineModel.transform(data)
10: selected <- prediction.select("Text", "NERTokens", "NERData", "label")
11: for row in selected.collect():
12:         text, nertokens, nerdata, label <- row
13:         if compareClasses( P, label, nertokens):
14:                 SaveFilteredData(text, label, nertokens, nerdata)
```

- *Batch Processing:* In the case of batch processing, the Filter method reads the files from the local file system directory by using a Spark function that creates RDDs (described in Section 2.2.1).
- *Stream Processing:* In the case of stream processing, line 1 reads the batches of records from each partition of the Kafka topic (described in Section 2.5) using a Spark function that creates RDDs.

Line 2 calls the "getUserPrefHistory" function to get all the user's previous preferences that have been generated from the history of previous queries sent by the user. This function reads all the previous preferences, if any, from the user's preference history file and adds those preferences to the input array of the preferences. In lines 3-7, the text data from the raw dataset are processed using the "map" function. At first, the text data are divided into a list of sentences (line 4), and each sentence is preprocessed by lowercasing the capital letters and eliminating the punctuation in line 5. In line 6, the "NER" function is applied in each sentence to extract the named entities such as people's names, products, times, and

dates. The output from this function is used in line 7 to create row instances, where in each row there are three columns: “Text”, “NERTokens”, and “NERData” for the sentence, named entities classes, and the extracted names entities, respectively. Line 8 creates the DataFrame with the three columns. This DataFrame is passed as an input to the stages of the machine learning model to classify the sentences according to their content in line 9. The model returns each row with another additional column “label” for the predicted class in line 9. In line 10, four columns: “Text”, “NERTokens”, “NERData” and “label” are selected from the new DataFrame. Lines 11-14 iterate through each row and compare the predicted classes with user preferences using the function “compareClasses”. If the predicted classes of a specific row match with the user preferences, the row is stored in the filtered data using the “SaveFilteredData” function in line 14.

3.4 The Search Method

The Search method searches data within the filtered data concurrently based on a linear search algorithm described in [92]. Similar to this linear search algorithm, the search method searches each sentences of the filtered data until it finds the sentence that matches the argument specified in the user’s query. The search method equally distributes the filtered data among the executor cores and then, performs the sequential search in an executor core. This method receives the user query in two ways: by keywords and by a sentence. After getting the user query, first, this method processes the comma-separated filtered data concurrently. Next, it looks within the processed filtered data and only returns the results related to the user-provided keywords or a sentence. If the user query is by a sentence, first the sentence is classified using the machine learning model and named entity recognition. The classes associated with that sentence are saved as user preferences in a

user preference history file. The Filter method adds the preferences from this history file in the user-provided preferences list (Pseudo Code 1 line 2), and this preferences list is used to filter the raw dataset. Consider the example of the filtered data presented earlier in Section 3.2. Recall the two sentences,

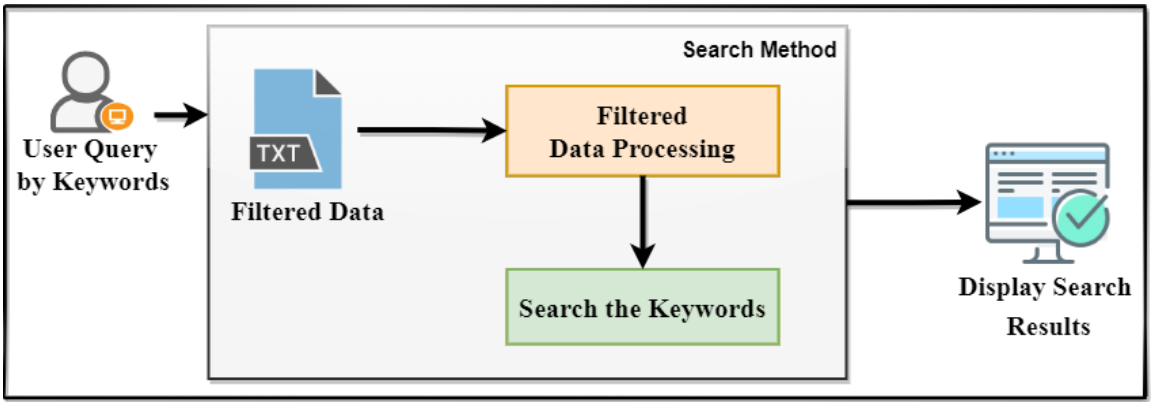
> *Sentence 1:* "An appointment has been scheduled with Dr. John Watson on September 25, 2020 at 9:00 am."*, "Person, Date, Time", "John Watson, September 25, 2020, 9:00 am", "Person"*
>
> *Sentence 2:* "A nurse will confirm the appointment before the scheduled date."*, "Person"*

If the user queries by a sentence: "When is my appointment with Dr. John Watson?", the Search method preprocesses the query and classifies it as "Person, Date" and extracts the named entity "John Watson". These classes "Person", and "Date" will be saved as user preferences in a user preference history file. Then, the Search method looks for that class and named entity in the filtered data and displays sentence 1: "An appointment has been scheduled with Dr. John Watson on September 25, 2020 at 9:00 am." as the result to the user. If the user searches for the keyword "appointment", it will display the full text contained in the sentences from both sentence 1 and sentence 2.

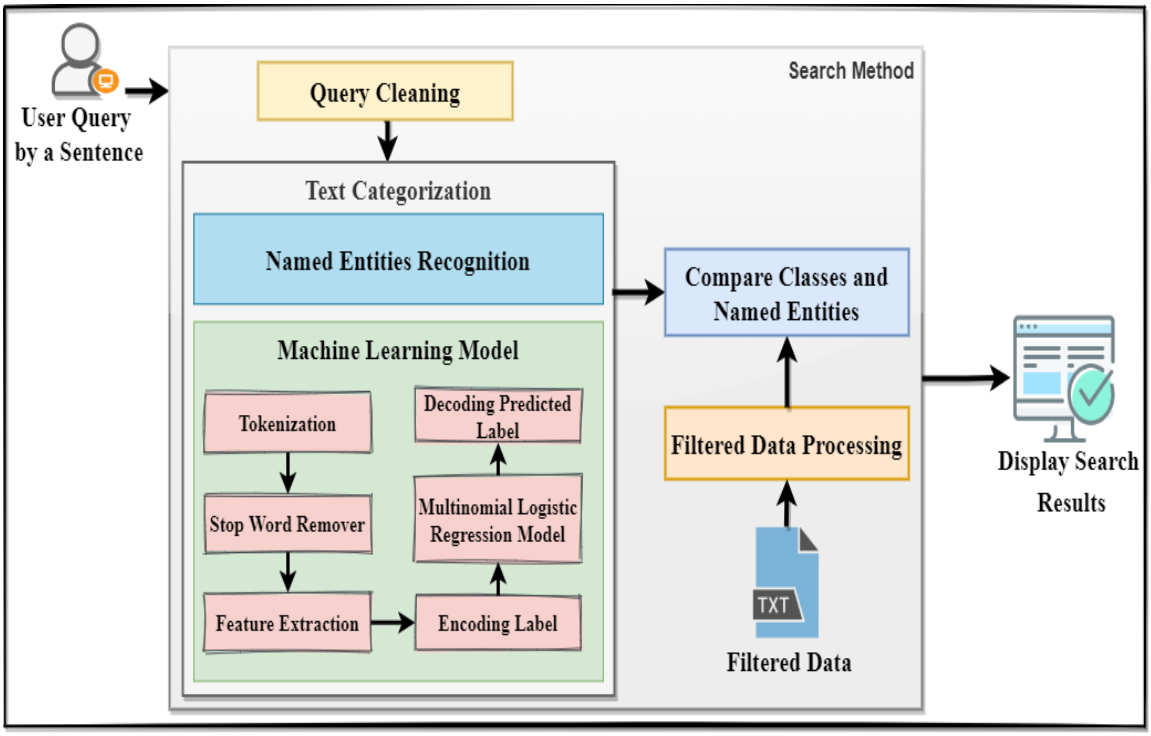
The components of the proof-of-concept prototype of the Search method are shown in Figure 8.

3.4.1 User Query by Keywords

In the method shown in Figure 8a, the user provides keywords to the Search method as input to get the results related to those keywords from the filtered data. First, the Search method processes the comma-separated lines of filtered data in the "Filtered Data



a) User query by keywords



b) User query by a sentence

Figure 8: Search Method

Processing" function. The Search method looks for the keywords in the sentences returned by this "Filtered Data Processing" function. Then, the Search method displays the relevant search results to the user.

3.4.2 User Query by a Sentence

In the second approach (see Figure 8b), the user provides a sentence to the Search method as input. First, the sentence provided by the user is preprocessed by the "Query Cleaning" function, which eliminates punctuation and converts the capital letters into lower case letters. Then, the processed sentence is classified by the machine learning model, and the named entities are extracted from the sentence using the named entity recognition function (see Figure 8b). The classes of the sentence are saved in a file as user preferences. The same machine learning model and named entity recognition function used in the Filter method (see Section 3.2) are used here as well. As in the case of the first method (Figure 8a), the "Filtered Data Processing" function is used to process the comma-separated lines of filtered data. In this method, this "Filtered Data Processing" function returns a list of sentences and their corresponding classes and named entities from the filtered data. Then, the classes and named entities of the user-provided sentence are compared with the classes and named entities of each row of the list returned by the "Filtered Data Processing" function. This function of the Search method returns the sentences from the filtered data whose classes and named entities match with the class and named entities of the user-provided sentence.

3.5 Search Method: Algorithms

Pseudo Code 2 presents the algorithm for the first Search method (discussed in Section 3.4.1) that corresponds to the user query by keywords. The input of this algorithm is a set of keywords. The Search method reads the filtered data using a Spark function that creates RDDs in Line 1. In line 2-4, the map function processes the filtered data. Line 3 uses the "preprocessing" function to extract filtered sentences from the comma-separated filtered data. The output from this function is used in line 4 to create a row instance, where in each

Pseudo Code 2: Search by Keywords

Input: User query as keywords, K
Output: User query results, R

```
1. srdd <- read the filtered data.
2. filteredData <- srdd.map {
3.     line <- preprocessing(line),
4.     Row[Text=line[0]]
5.  }.toDF()
6. R <- searchKeywords(filteredData, K)
7. Display R
```

row there is one column: “Text” for the sentence. Line 5 creates the DataFrame with this one column. Line 6 searches for the keyword in each row of the DataFrame using the “searchKeywords” function. Line 7 displays the search results returned by line 6.

Pseudo Code 3 presents the algorithm for the second Search method (see Section 3.4.2). The input of this algorithm is the user query as a sentence. The Search method reads the sentence using a Spark function and creates an RDD in Line 1. In line 2-5, the map function processes the sentence provided by the user. Line 3 preprocesses the sentence by lowercasing the capital letters and eliminating the punctuation. Line 4 extracts the named entities from the preprocessed sentence using the “NER” function. Line 5 creates a row with three columns: “Text”, “NERTokens”, and “NERData” for the sentence, named entities classes, and the extracted names entities, respectively. Line 6 creates a DataFrame with the three columns. To classify the sentence according to its content, the DataFrame is passed through the multiple stages of the machine learning model and returns the row with another additional column “label” for the predicted class (see lines 7-8). Line 9 collects all the four columns of the query. Line 10 saves the classes returned in line 7 to a history file. This is used by the Pseudo Code 1 in line 2. The Search method reads the filtered data

Pseudo Code 3: Search by a Sentence

Input: User query as a sentence, S
Output: User query results, R

```
1.  qrdd <- S
2.  query <- qrdd.map {
3.      line <- preprocessing(line),
4.      line <- NER(line)
5.      Row(Text=line[0], NERTokens=line[1], NERData=line[2])
6.  }.toDF()
7.  prediction <- MLPipelineModel.transform(query)
8.  selected <- prediction.select("Text", "NERTokens", "NERData", "label")
9.  QResult <- selected.collect()
10. savePreftoHistory(QResult)
11. srdd <- read the filtered data.
12. filteredData <- srdd.map {
13.     line <- preprocessing(line),
14.     Row[Text=line[0], nertokens=line[1], nerdata=line[2], label=line[3]]
15. }.toDF()
16. for row in filteredData.collect():
17.         R <- contentMatch(row, QResult)
18.Display R
```

using a Spark function that creates RDDs in Line 11. In line 12-15, the map function preprocesses the filtered data using the "preprocessing" function to make it four columns: "Text", "nertokens", "nerdata", and "label" for the filtered text, named entities classes, extracted named entities, and class by machine learning model, respectively. Line 16-17 matches the result classes from line 11 with each class of the filtered data row. Line 18 displays the search results.

3.6 Implementation

The proof-of-concept prototypes of the Filter and Search method are implemented in an Apache Spark cluster. All the algorithms discussed in Section 3.3 and Section 3.5 are implemented using PySpark [69]. PySpark enables the use of Python in Apache Spark. In other words, it is a Python API for Spark [70]. Some Apache Spark libraries and Python

libraries are used to implement the methods. Apache Spark's machine learning library in Python (MLlib) discussed in Section 2.3 is used to perform classification using machine learning algorithms. Apache Spark's Structured Streaming library in Python (discussed in Section 2.4) is used to read the stream data from Kafka. Python libraries that are used to implement the methods are spaCy (discussed in Section 2.6) and Natural Language Toolkit (NLTK) [71]. NLTK is used to divide the text data into sentences. In stream processing, the raw dataset is sent to the Filter method using Kafka [40]. Section 4.2 discusses the implementation specific to Kafka in more detail.

Chapter 4: Performance Evaluation

A set of experiments is performed on the system prototype to evaluate the performance of the Filter and Search methods deployed on an Apache Spark cluster. This chapter describes two use cases of the Filter method as well as experiments that have been conducted to evaluate the performance of the Filter and Search method. Use Case-1 concerns operations on stored data whereas Use Case-2 focuses on streaming data. In the first, the raw dataset is at rest and is available as one or more stored files on the system. In the second, filtering of data is performed in real time as data is streaming into the system. The results of the experiments and insights into system behavior and performance resulting from the performance analyses are also summarized in this chapter. Section 4.1 and Section 4.2 describe the experiments conducted on two use cases of the Filter method. The workload and system parameters are outlined, and the performance metrics used to evaluate the Filter method are identified for both use cases in Section 4.1 and Section 4.2, respectively. The results of the experiments for both use cases of the Filter method are then presented. Section 4.3 identifies the performance metrics used to evaluate the Search method, presents the experiments conducted to evaluate the performance of the Search method, and summarizes the results of the experiments. Section 4.4 discusses the performance of the machine learning models used. Section 4.5 describes a sample application that is based on the processing of the filtered data.

4.1 Use Case-1 of the Filter Method: Raw Dataset Stored in a Local Directory

The first use case for the Filter method considers a raw dataset comprised of multiple files that are stored in a local directory. Some popular use cases similar to this scenario include the meeting minutes and agendas in municipal government, healthcare, business, etc. as

well as news articles, journals, and notes regarding doctors' sessions. As an example, consider a secretary whose primary task is to take meeting minutes from each meeting that occurs among the planning committee executives of a company. In a meeting, the secretary usually does this by typing the meeting minutes being discussed based on the agenda. At a later date, one or more executives of the company may need to look for relevant information from those meeting discussions (i.e., important dates, future schedules, plans discussed, courses of action and their timelines, and the list of executives present in the meeting). In such a case, they typically need to go through each of the files containing the meeting minutes to search for the relevant information. It would be helpful if they could use a method that filters all these meeting minutes data according to their preferences (e.g., important dates, the list of executives present in the meeting, etc.), and stores only the filtered data that are of interest to a specific person or a group of employees. Instead of going through all the meeting minutes files to search for the preferred information, the executive could then use the Filter method to filter the data according to their preferences. This would allow the executive to find and utilize the preferred information from those filtered data stored in a single and smaller file, which would save time and resources.

Use Case-1 in the experimental analysis considers a similar scenario involving meeting minutes data as a raw dataset. Synthetically generated meeting minutes data are considered as a raw dataset. Each such synthetically generated raw dataset is generated from three sets of real meeting minutes available from a city council website [54]. Depending on the raw dataset sizes used in the various experiments these sets of meeting minutes are copied multiple times to produce the raw data used in the respective experiments. The proposed

Filter method deployed on the Apache Spark cluster is used to filter the meeting minutes data. The details of the raw dataset and experiments are discussed in the next sections.

4.1.1 System Configuration

To run the experiments, an Apache Spark cluster comprised of one master node and three worker nodes is set up on an Amazon EC2 cloud infrastructure in the ca-central-1 region [55]. Amazon provides different types of EC2 virtual servers (instances) depending on the CPU and memory capacity [55]. One c5a.4xlarge-type EC2 instance is used for running the master node and one worker node. This EC2 instance type is comprised of 16 cores and 32 GB RAM running the Ubuntu 20.04 LTS operating system. Two c5.2xlarge-type EC2 instances are used for running two other worker nodes. Each of the c5.2xlarge-type EC2 instances is comprised of 8 cores and 16 GB RAM running the Ubuntu 20.04 LTS operating system. Each worker node is running with 16 GB RAM, and the driver memory is set to 12 GB. The rest of the memory of the c5a.4xlarge-type EC2 instance is not used in the experiments. The scheduling policy is set to FIFO (first-in-first-out), the default scheduler for Spark. Resources are allocated statically in the cluster. With static resource allocation, a fixed number of resources is given to Filter and Search methods, and both methods can hold onto them for the whole duration [72]. Table 1 shows the detailed system configuration for the Apache Spark cluster.

4.1.2 Workload and System Parameters

This section describes the various parameters used in the experiments of the Filter method for Use Case-1. The parameters are divided into two types: workload parameters and system parameters. A summary of the workload and system parameters is displayed in Table 2. The experiments are performed by following a factor-at-a-time approach where

one of the parameters is changed while others are held at their default values (indicated in bold in Table 2). It should be noted that fixed values of workload parameters (e.g., driver memory and memory of worker nodes) are used in each experiment. The parameter values are listed in the second column of respective rows. The default value is indicated in bold. Each of the workload parameters and system parameters is explained next in each sub-section.

Table 1: System Configuration for the Spark Cluster

Node Type	Configuration Parameter	Value
Master Node and a Worker Node	Type of EC2 virtual server (instance)	c5a.4xlarge
	Architecture	x86_64
	Number of CPU cores	16
	CPU clock frequency	3.4 Gz
	RAM	32 GB
	Operating system	Ubuntu 20.04 LTS
Second and Third Worker Nodes	Type of two EC2 virtual servers (instances)	c5.2xlarge
	Architecture per instance	x86_64
	Number of CPU cores per instance	8
	CPU clock frequency per instance	3.4 Gz
	RAM per instance	16 GB
	Operating system per instance	Ubuntu 20.04 LTS

4.1.2.1 Raw Dataset Size (S_R)

The Raw Dataset Size (S_R) is the total size of the raw dataset that is the input of the Filter method. The raw dataset sizes used for the experiments are 117 MB, 215 MB, 410 MB, 1.1 GB, 2 GB, 4.1 GB**,** 6 GB, and 8 GB. Among them, the default value of S_R is set to 4.1 GB for the experiments where other workload and systems parameters vary. The research presented in [58] used 1 GB to 100 GB data sizes to predict the execution time of Spark applications.

Table 2: Summary of the Parameters Used in the Experiments of Use Case-1

Parameter Type	Parameters	Value
Workload	Raw Dataset Size (S_R)	{117 MB, 215 MB, 410 MB, 1.1 GB, 2 GB, **4.1 GB,** 6 GB, 8 GB}
	File Size (S_F)	{**19.5 MB**, 39 MB, 78 MB, 119.9 MB}
	Number of Files (N_F)	See equation 2
	Raw Dataset Category (C_R)	{**C_{R1}**, C_{R2}}
	Data Partitioning Strategy (D_P)	{**C_{DP}**, E_{DP}}
System	Number of Worker Nodes (N_W)	{**1**, 2**,** 3}
	Number of Executor Cores (N)	**{**1**,** 2, 4, 6, **8**, 12**}**
	Driver Memory	**12 GB**
	Memory/Worker Node	**16 GB**

4.1.2.2 File Size (S_F)

File Size (S_F) is the total size of a file. Files are used to create the raw dataset. The combined size of all the files is the total size of a raw dataset. The file sizes used are 19.5 MB, 39 MB, 78 MB, and 119.9 MB. The default value of S_F is set to 19.5 MB.

4.1.2.3 Number of Files (N_F)

The number of Files (N_F) is the total number of files in a raw dataset. The number of files depends on the file and raw dataset sizes. Raw dataset sizes (S_R) are calculated by multiplying the file size (S_F) with the number of files (N_F).

$$S_R = S_F \times N_F \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [2]$$

Depending on the file and raw dataset sizes experimented with, the value of N_F changes from 6 to 420.

4.1.2.4 Raw Dataset Category (C_R)

Raw Dataset Category (C_R) defines how different file sizes and numbers of files are used to create a given raw dataset. Two distinct raw dataset categories are used for the experiments:

- Raw Dataset Category 1 (C_{R1}): The raw dataset is comprised of files of the same size. The raw dataset size has a default value of 4.1 GB. It is comprised of 215 files with the size of each file being 19.5 MB in the C_{R1}.
- Raw Dataset Category 2 (C_{R2}): The raw dataset is comprised of files of different sizes. The default raw dataset (4.1 GB) consists of files with four different file sizes – 19.5 MB, 39 MB, 78 MB, and 119.9 MB. The number of files with sizes of 19.5 MB and 39 MB is 17, and the number of files for the other two sizes is 16.

4.1.2.5 Data Partitioning Strategy (D_P)

Data partitioning strategy (D_P) refers to how the data are distributed among the executor cores. Two types of data partitioning strategy are used (discussed in Section 3.2.1):

- Centralized Partitioning (C_{DP}): In C_{DP}, the files of the raw dataset are distributed among executor cores in such a way that each executor core handles one file at a time.
- Equal Distribution-Based Partitioning (E_{DP}): The raw dataset is equally distributed among executor cores in E_{DP}.

4.1.2.6 Number of Executor Cores (N)

The number of executor cores (N) is the total number of executor cores in a worker node. The numbers of executor cores used in the experiments are 1, 2, 4, 6, 8, and 12 where the default value is set to 8. The research presented in [58] used up to 40 executor cores where

each worker node has 5 executor cores. The master node of Spark uses 1 core. When the number of executor cores varies from 1 to 12, the remaining cores are not used in the experiments.

4.1.2.7 Number of Worker Nodes (N_W)

The number of worker nodes (N_W) is the total number of worker nodes in the Apache Spark cluster. The numbers of worker nodes used in the experiments are 1, 2, and 3 where the default value is set to 1. The number of worker nodes used in the research presented in [22] was varied from 1 to 3.

4.1.3 Performance Metrics

The performance of the Filter method for Use Case-1 is evaluated using three performance metrics:

(1) Computation Time (T_C): The computation time is the difference between the time at which the Filter method completes and the time at which it starts execution. Therefore, it is the total time required by the Filter method to complete its processing of the raw dataset on the given number of worker nodes. The computation time of the Filter method is measured by taking two timestamps using time.time() from Python Library: one when the Filter method starts execution and one after the Filter method finishes by generating filtered data. The computation time computed at the end by the Filter method is the difference between these two timestamps (in seconds). The computation time is computed as follows:

$$T_C = T_{EF} - T_{SF} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [3]$$

Where T_{EF} is the timestamp when the Filter method completes, and T_{SF} is the timestamp when the Filter method starts execution.

(2) Speedup (S(N)): Speedup for a given raw dataset is defined as the ratio of the computation time of the Filter method on a single processing element to the time taken by the method to process the same dataset on N processing elements. Speedup measures the ratio of performance achieved with two versions of the same code with a different number of processing elements. It provides an estimate of how well the application speeds up when parallelism increases. Thus, the speedup of the Filter method for a given dataset is the ratio of the computation time required to execute the Filter method in one executor core and the computation time required to execute the Filter method in N number of executor cores. The speedup of the Filter method is measured by calculating the computation time (in seconds) of the Filter method on two different setups: one computation time is calculated by running the Filter method on one executor core, and another computation time is calculated by running the Filter method on N executor cores (the number of executor cores is discussed in Section 4.1.2). Then, the speedup is computed by dividing these two computation times. The speedup of the Filter method is defined as follows:

$$S(N) = \frac{T(1)}{T(N)} \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots [4]$$

Where T(1) is the computation time required to execute the Filter method in one executor core and T(N) is the computation time required to execute the Filter method in N executor cores.

(3) Efficiency (E(N)): Efficiency is the fraction of time for which the N processors are usefully utilized. It is computed as the ratio between the computation time achieved on a processor and the product of N and the computation time required to execute

the application on N processors. Thus, efficiency is the ratio of the speedup of the Filter method (S(N)) and the number of processors, N:

$$E(N) = \frac{S(N)}{N} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [5]$$

Where S(N) is the speedup of the Filter method and N is the total number of executor cores.

4.1.4 Raw Dataset

Experiments were performed on raw datasets of eight different sizes. Two different categories of raw datasets have been investigated (described in Section 4.1.2). As discussed in Section 3.1.1, a raw dataset is a collection of one or multiple text files. In the experiments of Use Case-1, raw datasets of each category were generated from a few council meeting minutes collected from a city council website [54]. Each set of meeting minutes is comprised of over 5000 words. The single file of a raw dataset is obtained by replicating the content of one set of meeting minutes multiple times so that a given raw data size is achieved. For the first category, the resulting single file size is approximately 19.5 MB. This single file was replicated from 6–420 times in the local directory. The resulting total size of the raw dataset is 117 MB–8.1 GB. For the second category, the resulting single file size varies from 19.5 MB to 119.9 MB. Then, these files of different sizes were replicated from 15–16 times in the local directory. The resulting total size of the raw dataset is 4.1 GB. The workload parameters used to control the size of the raw datasets are discussed in Section 4.1.2.

4.1.5 Performance Evaluation of the Filter Method for Use Case-1

The experiments used to evaluate the performance of the Filter method were run on the Spark Cluster by changing the number of worker nodes, executor cores, raw dataset size,

file size, the number of files, parallelism, and file categories. These workload and system parameters are discussed in Section 4.1.2. These workload and system parameters are observed to have a significant effect on the computation time, speedup, and efficiency by other researchers [58]. The selection of the appropriate number of worker nodes and the number of executor cores per worker node heavily influences the performance of the Filter method. Each experiment is repeated a sufficient number of times such that a confidence interval of +- 3% of the mean computation time (defined in Section 4.1.3) is obtained at a confidence level of 95%.

4.1.5.1 Effect of the Raw Dataset Size (S_R) and the Number of Worker Nodes (N_W)

As discussed before, the raw dataset size is the total size of the raw dataset, and the number of worker nodes is the total number of worker nodes used in the experiments. Figure 9 captures the computation time achieved with different numbers of worker nodes and different raw dataset sizes. This experiment is conducted by changing the number of worker nodes from 1 to 3 on each run for a given size of a raw dataset. Each worker node has a fixed number of executor cores, which is set to the default value of 8. For example, when the number of worker nodes is 2, the allocated number of executor cores is 16. Each executor core in each worker node handles one file of a given raw dataset at a time. The raw datasets are comprised of files with the same size. The number of files changes with different sizes of raw datasets. In Figure 9, for a given raw dataset size the computation time is observed to decrease with an increase in the number of worker nodes. With an increasing number of worker nodes, parallelization increases, and more tasks are executed simultaneously, which results in decreased computation time for a given raw dataset size. This decrease is greater for a raw dataset with a larger size. For example, for a raw dataset

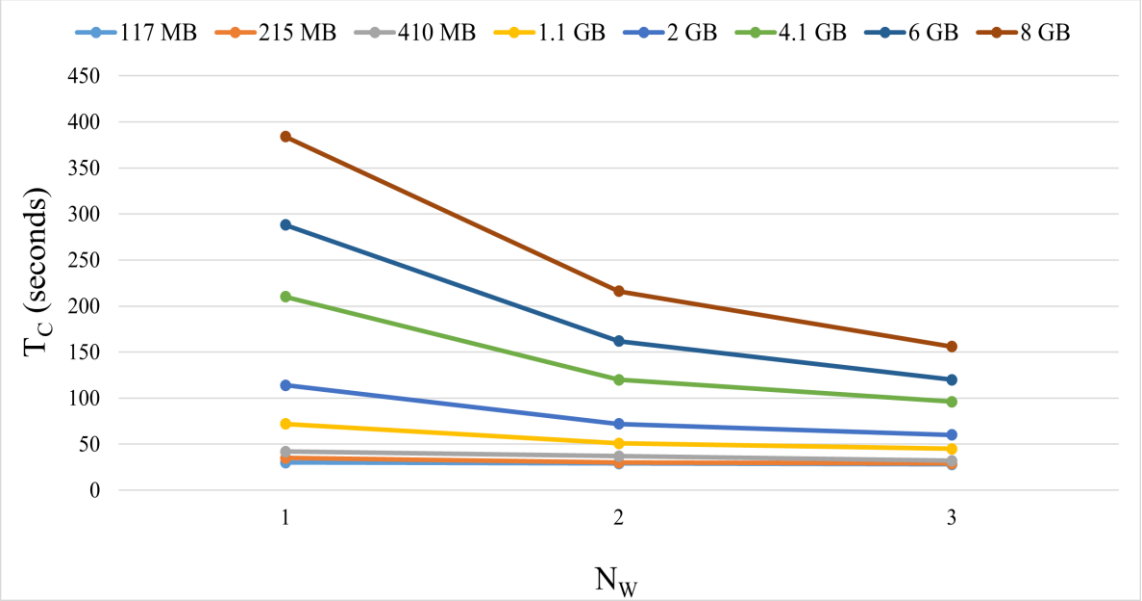


Figure 9: Computation Time for the Filter Method for Use Case-1 vs the Number of Worker Nodes for Different Raw Dataset Sizes

size of 8 GB, the completion time decreases by 59.375%: from 384 seconds to 156 seconds as the number of worker nodes changes from 1 to 3. For a raw dataset with a size of 117 MB, this decrease is observed to be only 6.667%. For smaller raw datasets (117 MB–410 MB), the single 8-executor-core worker node is apt for handling the workload demands. Thus, the increase in the number of worker nodes does not have an impact on the computation time.

Figure 10 shows the speedup (S(N)) achieved with given combinations of N_W and S_R. As expected, the speedup is increased with an increasing number of worker nodes and larger raw dataset sizes. The number of resources and parallelization increases with the increasing number of worker nodes, which results in an increase in the speedup for a given raw dataset size. For example, the speedup increases by 146.145%, from 7.657 to 18.846, as the number of worker nodes changes from 1 to 3 for the larger raw dataset size of 8 GB. Figure 11 displays the efficiency for different combinations of N_W and S_R. Figure 11 shows that

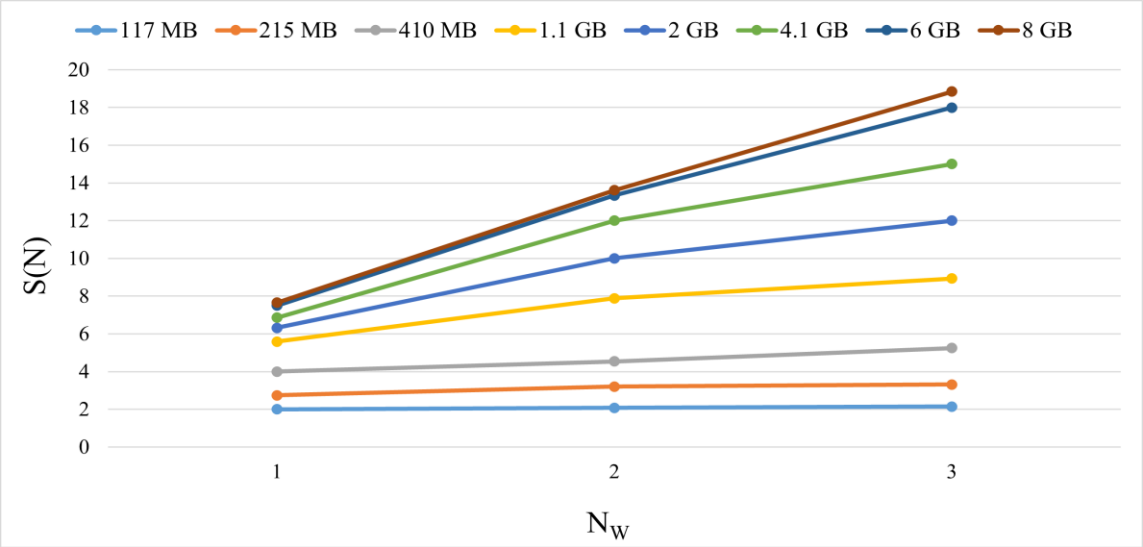


Figure 10: Speedup for the Filter Method for Use Case-1 vs the Number of Worker Nodes for Different Raw Dataset Sizes

the smaller raw datasets (117 MB–410 MB) use fewer resources, less than 29%, as the number of worker nodes increases from 1 to 3. Larger raw datasets (4.1 GB–8 GB) require more resources, resulting in an efficiency that is higher than 65%. For example, efficiency decreases from 0.25 to 0.089 and 0.957 to 0.785 as the number of worker nodes increases

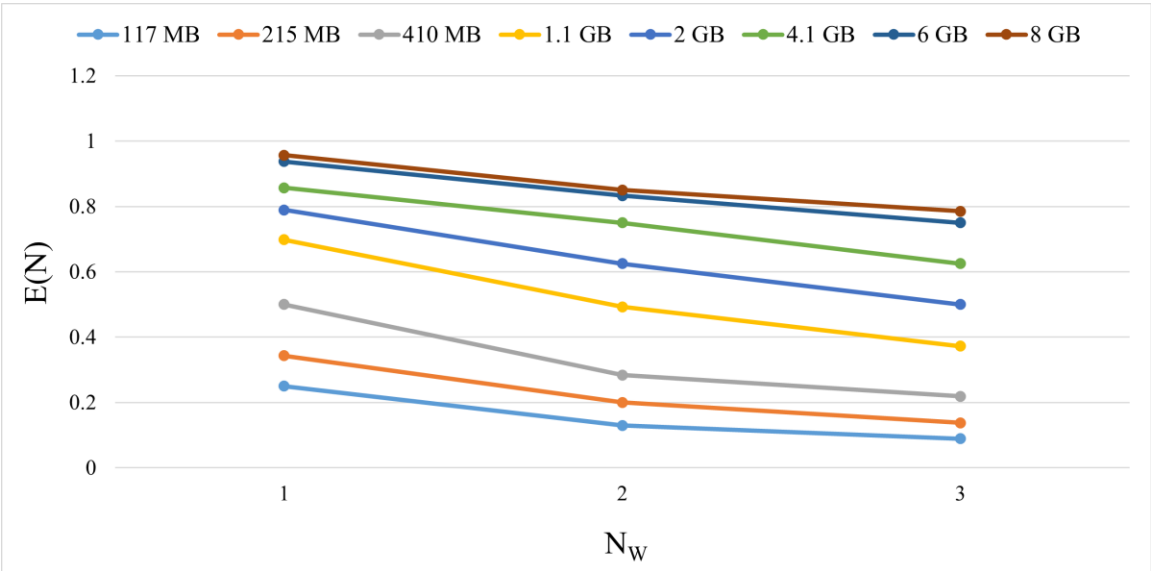


Figure 11: Efficiency for the Filter Method for Use Case-1 vs the Number of Worker Nodes for Different Raw Dataset Sizes

from 1 to 3 for the raw dataset of 117 MB and 8 GB, respectively. Figure 10 and Figure 11 capture the expected tradeoff between speedup and efficiency for parallel systems: the speedup increases while efficiency decreases with an increase in the number of worker nodes and the size of the raw datasets.

4.1.5.2 Effect of the Number of Executor Cores (N)

As shown in Figure 12, the computation time decreases with the increasing number of executor cores. This experiment is conducted by changing the number of executor cores of a worker node for the default raw data size value of 4.1 GB. The number of worker nodes is held at 1, which is the default value. The number of executor cores is changed from 1 to 12 for the worker node. The raw dataset (size 4.1 GB) is comprised of a fixed size of 215 files. The completion time decreases from 1440 seconds to 186 seconds (by 87.083%) as the number of executor cores changes from 1 to 12. More executor cores mean more tasks are executing parallelly in a worker node. A sharper decrease in computation time is observed when the number of cores is changed from 1 to 2 cores. For a higher number of

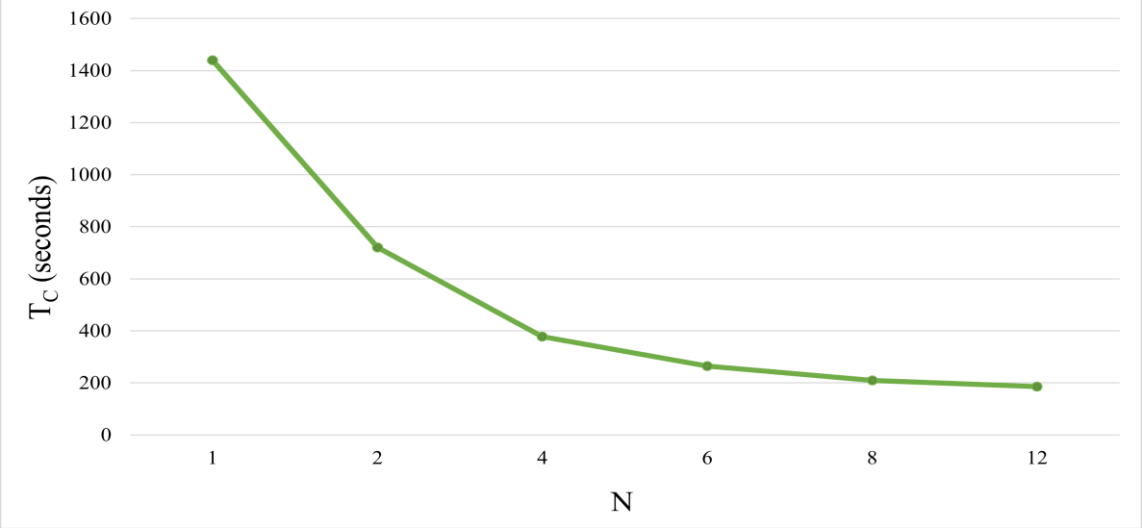


Figure 12: Computation Time for the Filter Method for Use Case-1 vs the Number of Executor Cores

executor cores, the decrease seems to be low. As each executor core handles one file of the given raw dataset at a time, all the executor cores do not finish at the same time. Each executor core handles a different number of files; thus, the decrease in computation time is non-linear.

The speedup achieved with N for a raw data size of 4.1 GB is shown in Figure 13. The speedup is observed to increase with an increasing number of executor cores. For example, the speedup increases from 2 to 7.74 (by 287%) as the number of executor cores changes from 2 to 12. Figure 14 displays the fraction of time for which a processor is usefully utilized for N. Efficiency is observed to decrease from 1 to 0.65 as the number of executor cores changes from 2 to 12. All the cores are not utilized 100% and thus the speedup is not non-linear.

4.1.5.3 Executor Core Parallelism vs Worker Node Parallelism

Both the number of worker nodes and the number of executor cores determine the degree of parallelism in an application execution. An experiment where N_W is varied from 1 to 3 is conducted. In this experiment, 12 executor cores are allocated in a worker node (N_W =

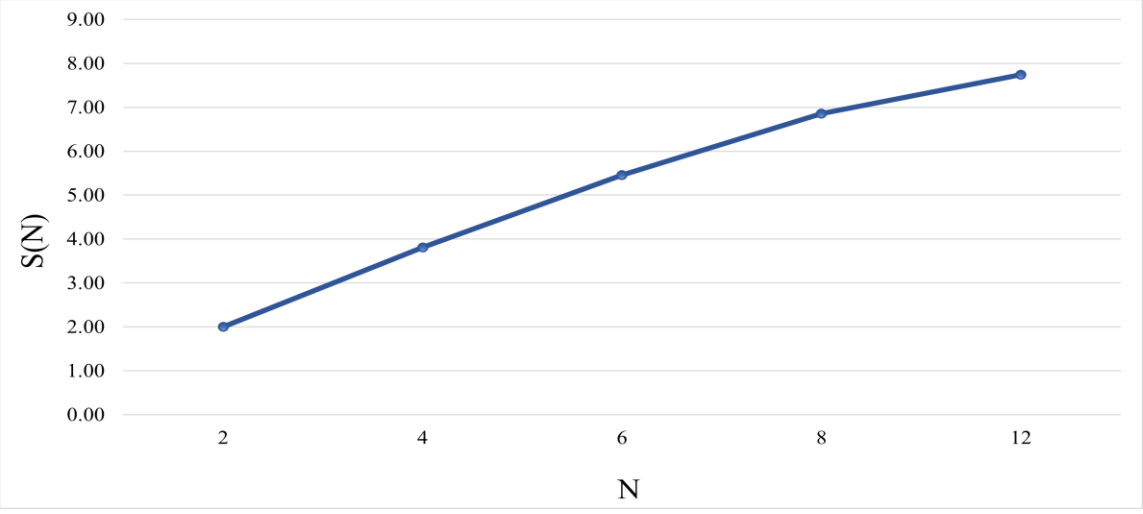


Figure 13: Speedup for the Filter Method for Use Case-1 vs the Number of Executor Cores

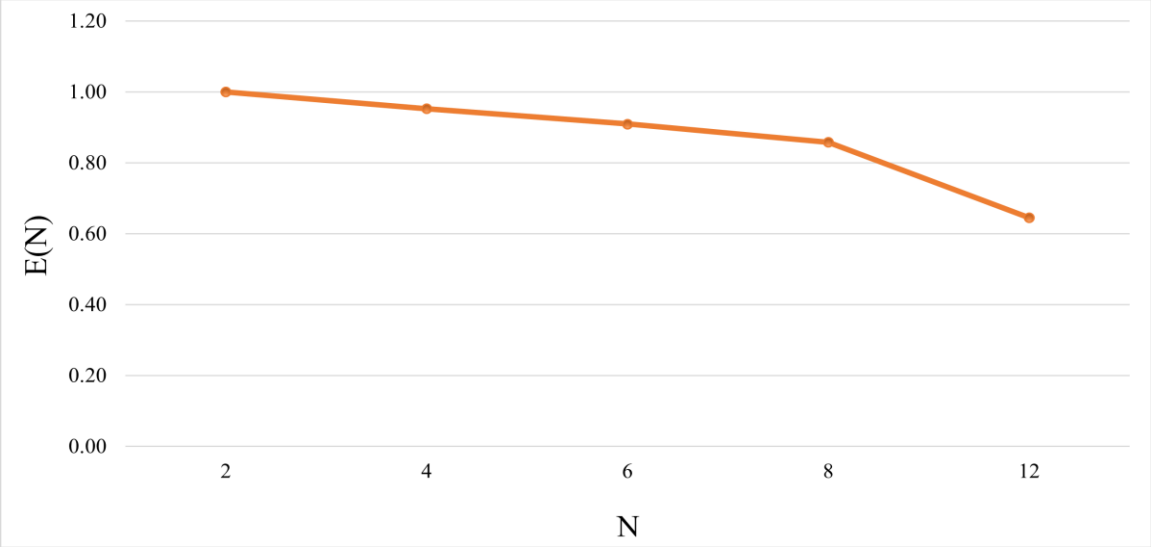


Figure 14: Efficiency for the Filter Method for Use Case-1 vs the Number of Executor Cores

1); 6 executor cores are allocated on each worker node when N_W is 2; and when N_W is 3, 4 executor cores are allocated on each. The total N used by all the worker nodes is fixed at 12 in this experiment. The raw dataset size is the default value of 4.1 GB: the raw dataset is comprised of 215 files of 19.5 MB each.

Although the total N in the cluster is 12, T_C decreases from the T_C achieved on the system with 12 executor cores in a worker node (Figure 15). The reason for this increase is the intercommunication delay between the worker nodes and driver node. When the number of executor cores in a worker node is high (the system where one worker node has 12 executor cores), it requires more computing time because of the time taken to gather the results from all the executor cores and send them to the driver node.

4.1.5.4 Effect of File Size (S_F) and the Number of Files (N_F)

Figure 16 displays the computation time achieved by changing the file size and number of files while holding the total size of the raw dataset (sum of the size of all files) at the default value of 4.1 GB. For example, when the file size is 19.5 MB, to make the raw data size 4.1

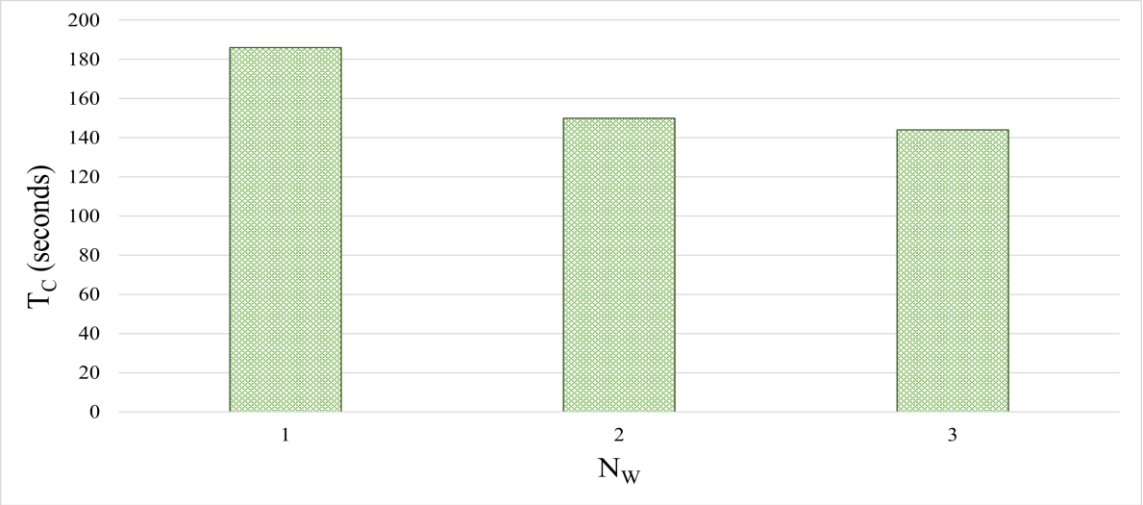


Figure 15: Computation Time for the Filter Method vs the Number of Worker Nodes While Total Number of Executor Cores is 12 in the Cluster.

GB, it requires 215 files with the size of each file being 19.5 MB. In Figure 16, the x-axis shows the file size and the number of files as "File size/Number of files (e.g., 19.5 MB/215)". This experiment runs on one worker node with eight executor cores, which are the default values. It is interesting to note that the number of files in the raw dataset has

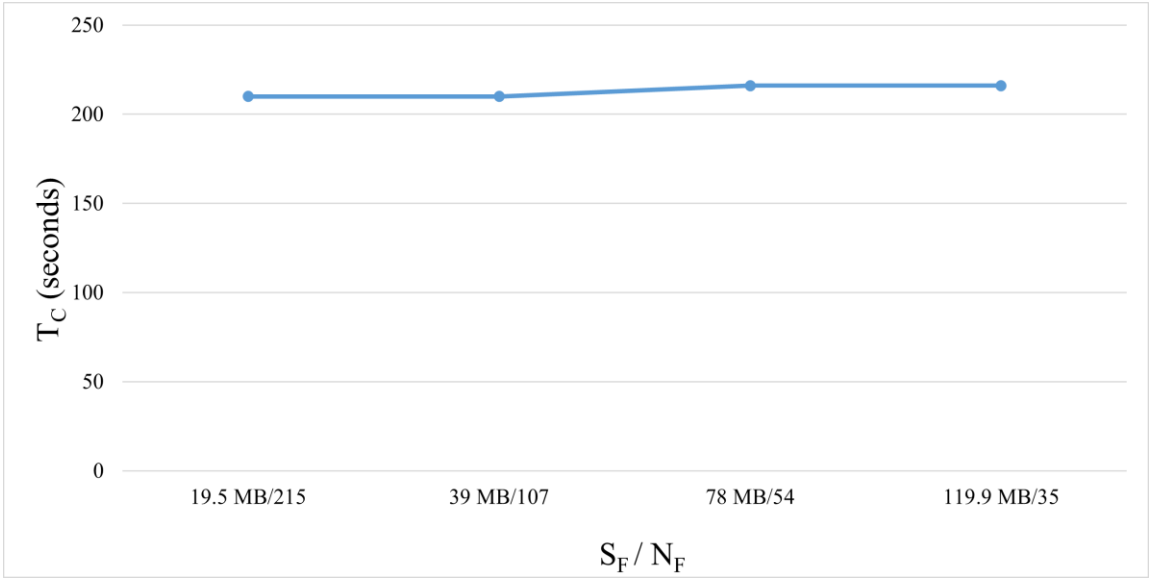


Figure 16: Computation Times for the Filter Method for Use Case-1 vs File Sizes and the Number of Files

only a small impact: the computation time seems to be insensitive to the number of files. The reason behind this is that the total raw dataset size is the same for each combination of file size and its corresponding number of files.

4.1.5.5 Effect of the Data Partitioning Strategy (D_P) and Number of Worker Nodes (Nw)

From Figure 17, it can be seen that the data partitioning strategy (D_P) has a significant impact on computation time. In the orange line of Figure 17, each executor core of the worker nodes handles a different number of files of the raw dataset, which is called centralized partitioning (C_{DP}) as discussed in Section 3.2.1. In the green line of Figure 17, the raw dataset is distributed among executor cores equally, which is called equal distribution-based partitioning (E_{DP}) as discussed in Section 3.2.1. The experiment was conducted by changing the number of worker nodes for a given data partitioning strategy (C_{DP} and E_{DP}). In the experiment, the number of worker nodes varies from 1 to 3, and each worker node has the default number of executor cores value of 8. The raw dataset size is

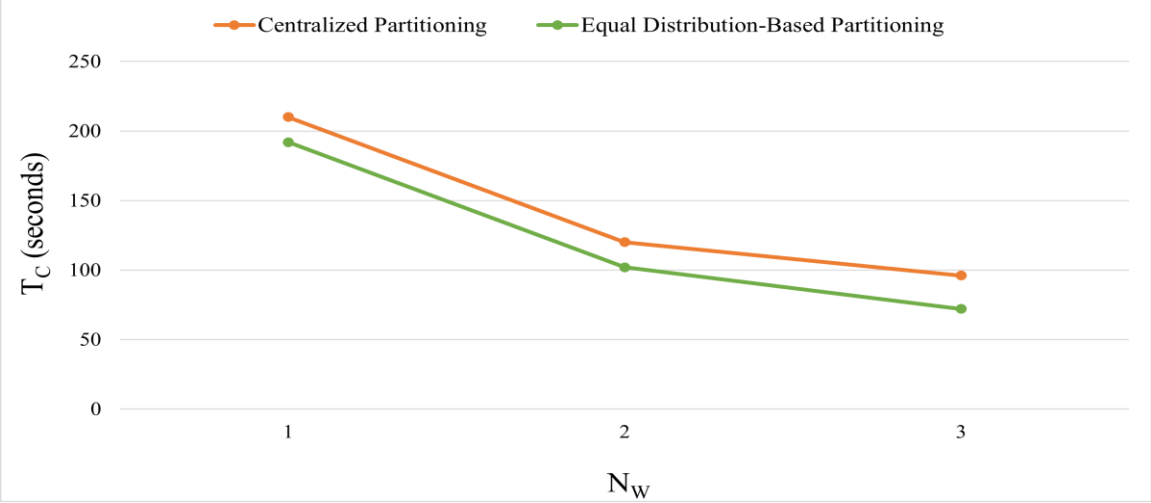


Figure 17: Computation Time for the Filter Method for Use Case-1 vs the Number of Worker Nodes for Different Data Partitioning Strategies

set to the default value of 4.1 GB. For a given number of worker nodes, the computation time achieved with E_{DP} is lower than that achieved with C_{DP}. The computation time decreases from 90 seconds to 66 seconds from C_{DP} to E_{DP} for the number of worker nodes equal to 3. The reason for this is that for the experiment shown in Figure 17, E_{DP} is balancing the load equally on each executor core of the worker node, and in C_{DP} each executor core handles a different number of files; thus, all the executor cores do not finish at the same time. This distinction in computation time seems to have a significant impact on performance.

Figure 18 displays the speedup achieved with D_P and N_W. The speedup achieved for a given N_W with E_{DP} is slightly higher than C_{DP} in Figure 18 because E_{DP} is balancing the load equally on each executor core of the worker nodes. The speedup increases from 16 to 17.143 (by 7.144%) from C_{DP} to E_{DP} for three worker nodes. Figure 19 shows the effect of D_P and N_W on efficiency. The efficiency achieved with C_{DP} is slightly higher than E_{DP}.

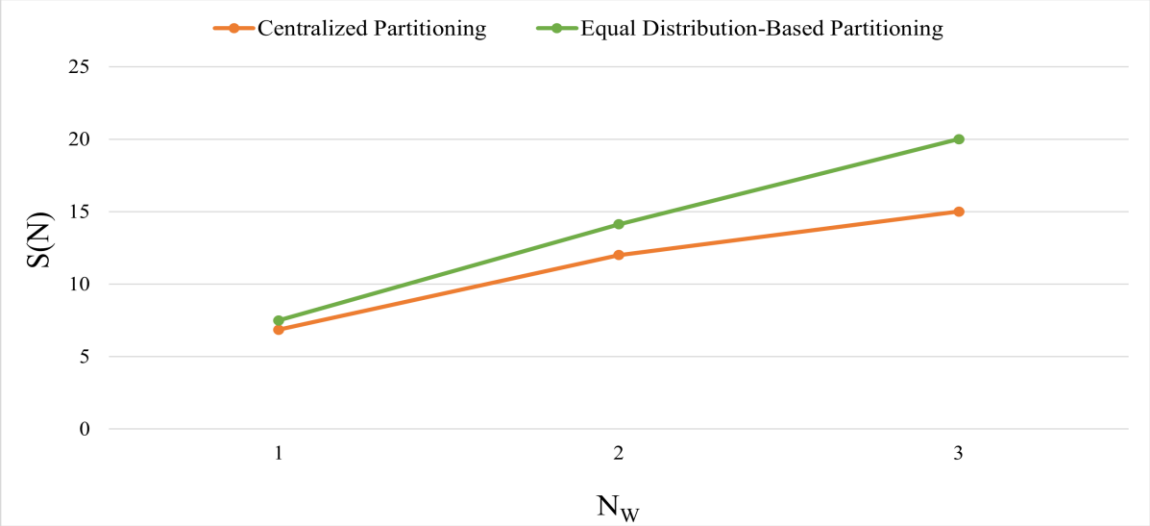


Figure 18: Speedup for the Filter Method for Use Case-1 vs the Number of Worker Nodes for Different Data Partitioning Strategies

4.1.5.6 Effect of the Raw Dataset Category (C_R)

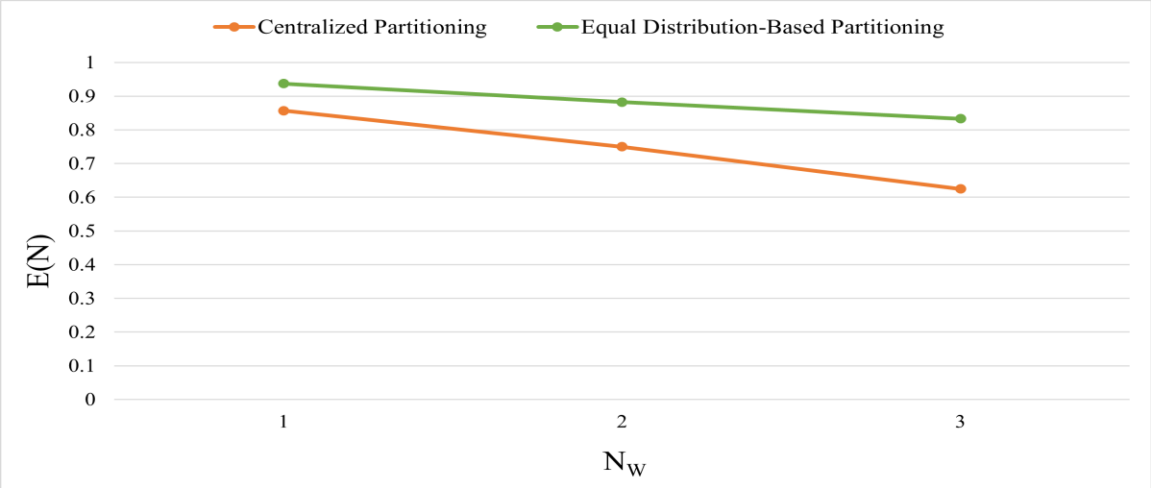


Figure 19: Efficiency for the Filter Method for Use Case-1 vs the Number of Worker Nodes for Different Data Partitioning Strategies

Figure 20 and Figure 21 display the computation time achieved with the raw dataset category C_{R1} and C_{R2} with respect to two different system parameters: N_W and N, respectively. For the C_{R1} category, the default raw dataset size value is 4.1 GB, and it contains multiple files of the same size (discussed in Section 4.1.2.4). For the C_{R2} category, the total raw dataset size is 4.1 GB, and the raw dataset contains multiple files of different sizes (discussed in Section 4.1.2.4).

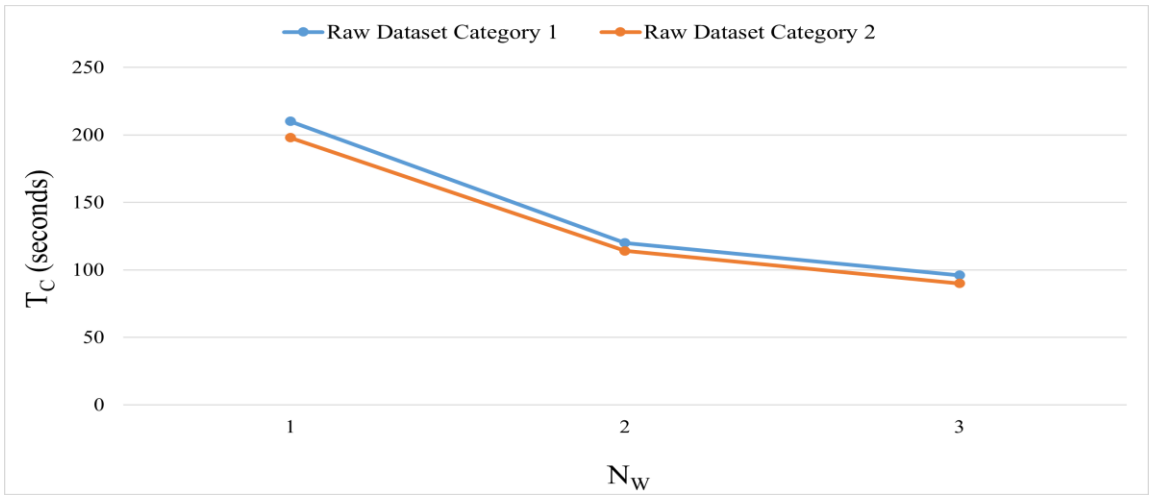


Figure 20: Computation Time for the Filter Method for Use Case-1 vs the Number of Worker Nodes for Different Raw Dataset Categories

In Figure 20, the experiment was conducted by changing the number of worker nodes for a given raw dataset category (C_{R1} and C_{R2}). The number of worker nodes varies from 1 to 3, and each worker node has a default number of executor cores value of 8. In Figure 21, the experiment was conducted by changing the number of executor cores for a given raw dataset category (C_{R1} and C_{R2}). The number of executor cores varies from 1 to 12 in a worker node.

Figure 20 and Figure 21 show that although the raw dataset is comprised of different file sizes, this has only a small impact when there is a varying number of worker nodes and number of executor cores, respectively. The computation time seems to be insensitive to the categories for the raw dataset. The reason for this is that the total raw dataset size is the same for both categories.

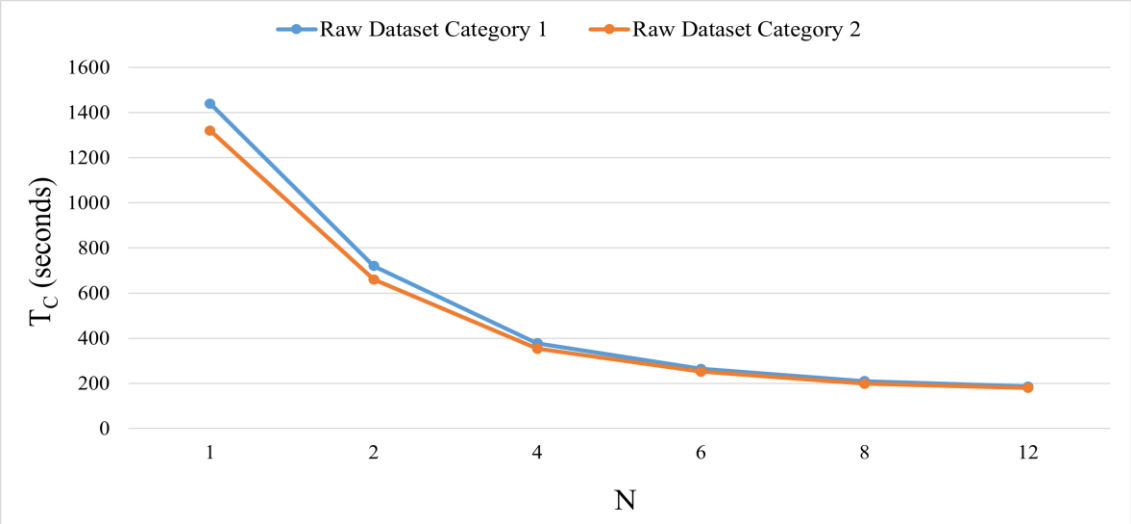


Figure 21: Computation Time for the Filter Method for Use Case-1 vs the Number of Executor Cores for Different Raw Dataset Categories

4.1.5.7 Additional Experiments on a Smaller System

In the beginning of this research, an experiment was conducted on a smaller system. The smaller system is comprised of three worker nodes with a raw dataset size of 6.2 MB. The

smaller Apache Spark cluster is set up on a local computer and is representative of a use case in which the user uses their own resources to filter meeting files. This cluster was created using four-core Intel i5 processors, 4 GB of RAM, and the Ubuntu 18.04 operating system. This Spark cluster consists of one master node and three worker nodes. Some modifications are applied to the default Apache Spark configuration [1], which includes one executor core for each worker node, 1 GB for each worker memory, and 1 GB for driver memory. The raw dataset is obtained by replicating the same meeting minutes (discussed in Section 4.1.4), resulting in 50 raw data files that are stored in a local file system. Each file contains the minutes of a meeting, and the file size is approximately 125 KB. The raw dataset is thus comprised of 50 files, and the total raw dataset size is 6.2 MB. Figure 22 shows the computation time required by the proposed Filter method in the Spark cluster residing on the local computer. In this experiment, the Filter method is run for different numbers of worker nodes (varies from one to three) with the 6.2 MB raw dataset. As expected, the computation time of the Filter method is observed to decrease by adding more worker nodes to the cluster. For example, the computation time is 210 seconds for one worker node, whereas it decreases to 84 seconds after adding two more worker nodes to the cluster. A 60-second computation time was achieved for the smaller dataset size of 117 MB on the EC2 cluster (Figure 9) whereas a 210-second computation time was achieved for the raw dataset size of 6.2 MB on a small system with one worker node and one executor core. This experiment on the smaller cluster with the number of worker nodes, $N_{W,}$ for the raw dataset size of 6.2 MB shows that the performance of the Filter method changes with the system configuration.

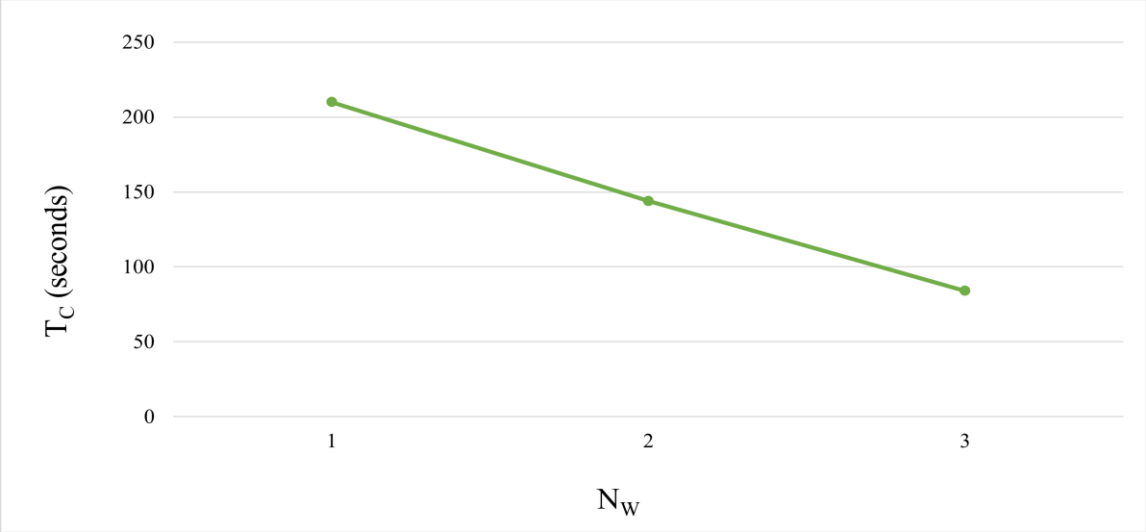


Figure 22: Computation Times for the Filter Method Residing on a Local Computer vs the Number of Worker Nodes

4.2 Use Case-2 for the Filter Method: Raw Dataset as Streaming Data

The second use case for the Filter method considers the raw dataset as streaming data, where a producer is sending stream data in small batches to partitions in a Kafka topic (described in Section 2.5). Each batch has a fixed number of records (sentences), and after sending a batch of records the producer waits for a predefined fixed amount of time before sending the next batch. The Filter method in Spark Cluster reads the records from Kafka topic partitions and then filters the records.

Some popular use cases similar to Use Case-2 included the analysis of network and webserver logs, sensor data, tweets, live recordings, and many more. As an example, consider a voice recording application [59][60][61] that does the live recording of the conversations between a doctor and patient. The dialogues between the doctor and patient are recorded using this application. The recording application records and converts this recording into text in real time. However, there might be some data that are not useful for the patient later. The patient might need only key information like the time and date of the

next appointment, a list of prescribed medications, or any tests that need to be done before the next appointment. Therefore, instead of storing everything from the recording, the application can store only those converted conversations that are relevant to the patient in real time. This will be helpful for the patient in searching for the key information and will also save time.

Use Case-2 in the experimental analysis considers a similar application involving doctor-patient conversations. The producer sends batches of synthetically generated data to the partitions in a Kafka topic, and then the Filter method processes these batches. The details of the data producer, system configuration, and workload parameters are discussed in the next sections.

4.2.1 Data Producer

For the experiments run for Use Case-2, a producer application sends batches of 80 synthetically generated raw data to the topic of Apache Kafka. As discussed in Section 2.5, Apache Kafka is a popular distributed publish/subscribe-based messaging system. The internals of Apache Kafka is discussed in Section 2.5 of this thesis. To send data into partitions of a Kafka topic, the producer API of "kafka-python", a Python client for the Apache Kafka distributed stream processing system [63], has been used in this thesis. A Kafka topic is created with multiple partitions. A producer application is created using the Python programming language [64], which pushes records to the Kafka topic using the producer API [63]. The producer sends streams of records (sentences) to multiple partitions of a topic in parallel (see Figure 23). Then, the Filter method in Apache Spark filters the records in parallel. The raw dataset source from where the producer application reads records is discussed in Section 4.2.5. For a given number of batches, the producer

application divides the total number of records in this batch by the number of partitions in the topic and sends an equal number of records to each partition. The number of partitions in the topic is set to be equal to the number of executor cores of the Apache Spark cluster for all the experiments. Various workload and system parameters used in the experiments e.g., the number of records in each batch, the time difference between two consecutive batches sent by the producer application, and the number of executor cores and worker nodes of Spark Cluster are discussed in Section 4.2.3.

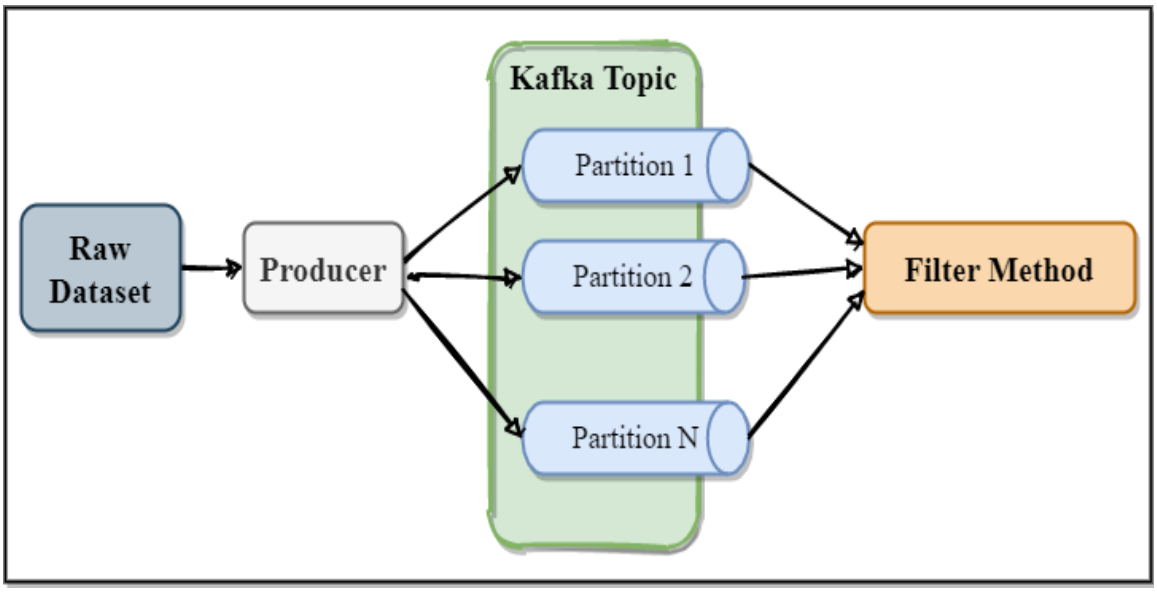


Figure 23: Kafka Architecture

4.2.2 System Configuration

To run the experiments, an Apache Spark cluster, a Kafka cluster, and ZooKeeper are set up on an Amazon EC2 cloud infrastructure in the ca-central-1 region [55]. Kafka uses ZooKeeper for cluster synchronization and state management. The Apache Spark cluster setup used for the experiments of Use Case-2 is similar to the setup described in Section 4.1.1. The Kafka and ZooKeeper are set up on the same c5a.4xlarge-type EC2 instance where the master node and one worker node reside. The producer application is run on a

t2.micro-type EC2 instance equipped with 1 GB of RAM, one CPU core, and a clock speed of 2.5 GHz running on the Ubuntu 20.04 LTS operating system.

4.2.3 Workload and System Parameters

This section describes the various workload and system parameters used in the experiments for Use Case-2. A summary of the workload and system parameters used in the experiments of Use Case-2 is displayed in Table 3. The experiments for Use Case-2 are also performed by following a factor-at-a-time approach, where one of the parameters is changed while others are held at their default values (indicated in bold in Table 3). Note that fixed values of workload parameters (e.g., the number of batches and Kafka topic) and system parameters (e.g., driver memory and worker nodes memory) are used in each experiment.

4.2.3.1 Length of Batch in Records (B_L)

The length of Batch is the number of records (sentences) in each batch sent by the producer. Each record is a single sentence. The values used in the experiments are 200, 500, 1000, and 2000 where the default value of B_L is set to 1000. Different ranges of batch lengths have been used by previous researchers [93][94]. The value of B_L chosen lies within the ranges of small and large values of B_L used by the researchers.

4.2.3.2 Batch Interval (B_I)

After sending a batch of records, the producer waits for a specific time period before sending the next batch of records. The Filter method completes the filtering of the batch of records within that time interval before the next batch of records arrives. This time interval is referred to as the Batch interval. Thus, the Batch interval is the time difference in seconds between consecutive batches of records sent by the producer. The values of B_I used in the

experiments are 15, 20, and 30 seconds where the default value of B_I is set to 15 seconds. This is in line with the research presented in [93] that used batch intervals between 10 seconds and 50 seconds.

Table 3: Summary of the Parameters Used in the Experiments of Use Case-2

Parameter Type	Parameter	Value
Workload	Length of Batch in Records (B_L)	{200, 500**, 1000**, 2000} records
	Batch Interval (B_I)	**{15,** 20, 30**}** seconds
	Number of Batches (N_B)	**80**
	Kafka Topic	**1**
	Number of Partitions in a Kafka Topic (N_P)	**{**1, 2, 4, 6, **8**, 12, 16, 24**}**
System	Driver Memory	**12 GB**
	Number of Worker Nodes (N_W)	{**1**, 2**,** 3}
	Number of Executor Cores/Worker Nodes (N)	**{**1**,** 2, 4, 6**, 8,** 12**}**
	Memory/Worker Nodes	**16 GB**

4.2.3.3 Number of Batches (N_B)

N_B is the total number of batches is sent by the producer in a given experiment. The N_B is set to the fixed value of 80. The producer application sends 80 batches to the Filter method. In the research presented in [94], the number of batches was set to 80. Each Kafka partition is a log file on the disk, which means records from the producer application are written to the log in the order in which they are received by the Kafka broker. In addition, Kafka also stores logs in the disk for garbage collection and other debugging messages. The disk space available in the virtual machine (used in this research) where Kafka resides is 100 GB. With an increasing number of batches, the log files generated by Kafka increases, which fills up the disk space. Due to this disk space limitation, the number of batches is set to 80.

4.2.3.4 Kafka Topic

The producer sends batches of records to the Kafka topic. Kafka topic is discussed in Section 2.5.

4.2.3.5 Number of Partitions in a Kafka Topic (N_P)

The number of partitions is the total number of parts a Kafka topic is divided into. The multiple partitions in the Kafka topic allows Spark to process data in parallel. In the experiments, the number of partitions varies depending on the number of executor cores and worker nodes. For instance, if the number of worker nodes is 2 and the number of executor cores is 8 for each worker node, then the number of partitions will be 16 in the Kafka topic. Kafka partitions are described in Section 2.5.

4.2.4 Performance Metrics

The performance of the Filter method for Use Case-2 is evaluated using two performance metrics:

(1) Average Batch Processing Latency (T_P): Average batch processing latency is the average time in seconds a batch takes to be processed by the Filter method on a given number of resources on the Apache Spark cluster. To compute T_P, two timestamps are taken using time.time() from Python Library: one after the Kafka producer finishes sending a batch and one after the Filter method finishes processing the batch from Kafka by generating the filtered data. The difference between these two timestamps (in seconds) is used to compute the batch processing latency. The average batch processing latency is then computed by taking the average of all the computed batch processing latencies:

$$T_P = \frac{T_{P2} - T_{P1}}{N_B} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [6]$$

where T_{P1} is the timestamp taken after the Kafka producer finishes sending a batch, T_{P2} is the timestamp taken when the Filter method finishes processing the batch, and N_B is the total number of batches.

(2) Throughput (X): Throughput is the number of batch records in bytes being processed per second. To compute the throughput, the total number of bytes completed is divided by the elapsed time (in seconds) of the Filter method. The total bytes completions is measured by adding the total bytes of each batch. The elapsed time (T_E) is the total time in seconds taken by the Filter method to complete the processing of all the batches. This is the time that the Filter method spends on the processing of all the batches from the beginning to the end. T_E is measured by taking two timestamps (in seconds) using time.time() from Python Library: one when the Filter method starts processing the first batch and the other after the Filter method finishes processing all the batches. The difference between these two timestamps (in seconds) is used to calculate the elapsed time (T_E). The throughput (X) is computed as follows:

$$X = \frac{C_B}{T_E} \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots [7]$$

$$T_E = T_{S2} - T_{S1} \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots [8]$$

where C_B is the total number of bytes processed and T_E is computed from two timestamps. T_{S1} is the timestamp in seconds when the Filter method starts processing the first batch, and T_{S2} is the timestamp in seconds when the Filter method completes the processing of all the batches.

4.2.5 Raw Dataset

Experiments were performed on synthetically generated stream data. As discussed in Section 3.1.1, in stream processing the raw dataset is a collection of records that comes from the Kafka stream source. In the experiments of Use Case-2, a raw dataset was generated synthetically by using several sample medical conversations between doctors and patients that were collected from two websites [65][66]. Each set of conversations is comprised of over 200 words. The raw dataset is obtained by replicating the content of the conversations. The conversations were replicated 800 times in a file of the local directory. The resulting total size of the raw dataset is 10 MB. The producer application reads the sentences from this raw dataset file to create batches of records and sends the batches to the Kafka topic.

4.2.6 Performance Evaluation of the Filter Method for Use Case-2

To evaluate the performance of the Filter method for Use Case-2, several experiments were run on the Spark and Kafka Cluster by changing the number of worker nodes, executor cores, length of batch records, and batch intervals. In each experiment, the producer application sends 80 batches containing synthetically generated records to a topic of Apache Kafka. After sending a given batch of records, the producer application has to wait a batch interval to send the next batch. The total time a producer application needs to send 80 batches is 80 multiplied by the batch interval in each experiment. The workload and system parameters are discussed in Section 4.2.3. These workload and system parameters are observed to have a significant effect on the average batch processing latency and throughput.

4.2.6.1 Effect of the Length of Batch in Records (B_L)

To see the effect of the length of batch (B_L) on the performance of the Filter method, an experiment was conducted by changing B_L. The number of executor cores in a worker node was held at the default value of 8 and a default batch interval (B_I) value of 15 seconds is used. The B_I is varied from 200 to 2000 records in this experiment. Figure 24 shows the average batch processing latency (T_P) achieved with different lengths of batches in the records. Figure 24 shows that T_P increases with an increase in B_L. The Filter method requires more time to process the increased length of batch records. So, for a default N

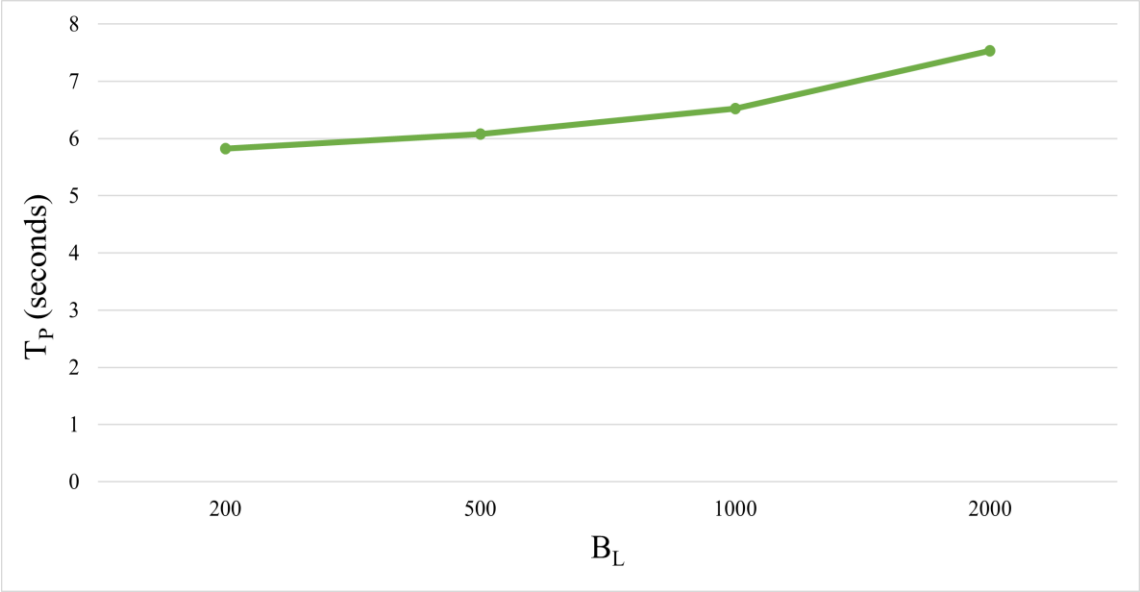


Figure 24: Average Batch Processing Latency for the Filter Method for Use Case-2 vs the Length of Batch in Records

value of 8 and default B_I value of 15 seconds, with an increase in B_L, the Filter method takes more time to complete the processing of 80 batches.

Figure 25 shows that the throughput (X) increases with an increase in the length of batch in records. The total time required by the Filter method is nearly the same for all B_L. Since the batch interval (B_I) is fixed to a default value of 15 seconds, the throughput increases as the total number of bytes in records increases.

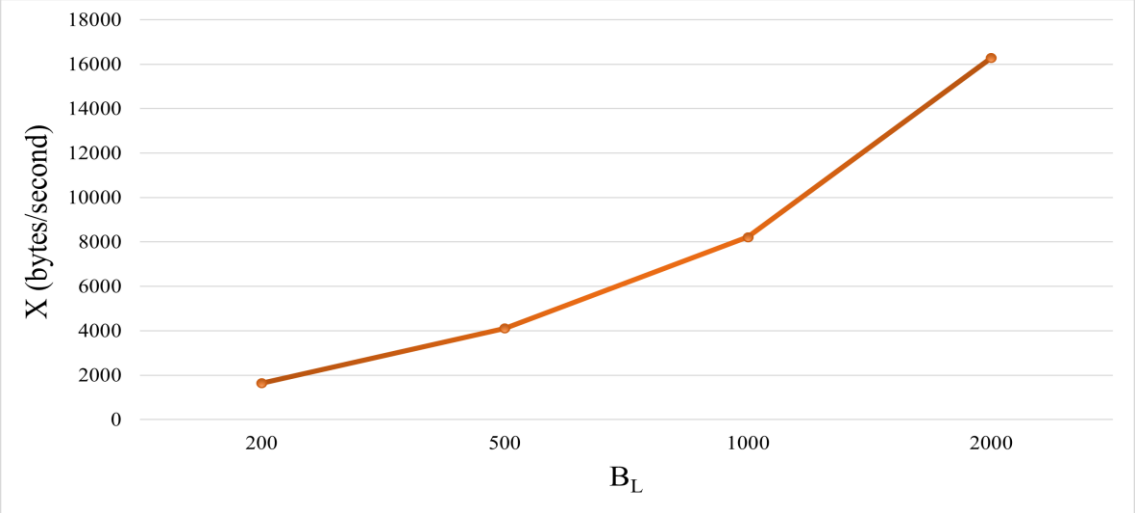


Figure 25: Throughput for the Filter Method for Use Case-2 vs the Length of Batch in Records

4.2.6.2 Effect of Batch Interval (B_I)

As discussed before, the time difference (in seconds) between two continuous batches sent by a producer is referred to as the batch interval (B_I). The Filter method needs to complete its processing within this time frame so that the filtered data from the current batch are available before the next batch of records arrives on the Spark cluster. To see the effect of the batch interval (B_L) on the Filter method, the experiment was conducted by changing the value of B_I. The number of executor cores is set to the default value in a worker node and the default value of batch records of 1000 is used. The value of B_L changes from 15 to 30 seconds in this experiment.

Figure 26 shows the T_P achieved with different batch intervals (B_I). Increasing the value of B_I has no effect on the T_P. The reason for this is that while the length of a batch in records (B_L) varies, the batch length remains the same. Thus, the time required by the Filter method to process each batch remains the same. The average batch processing latency is not

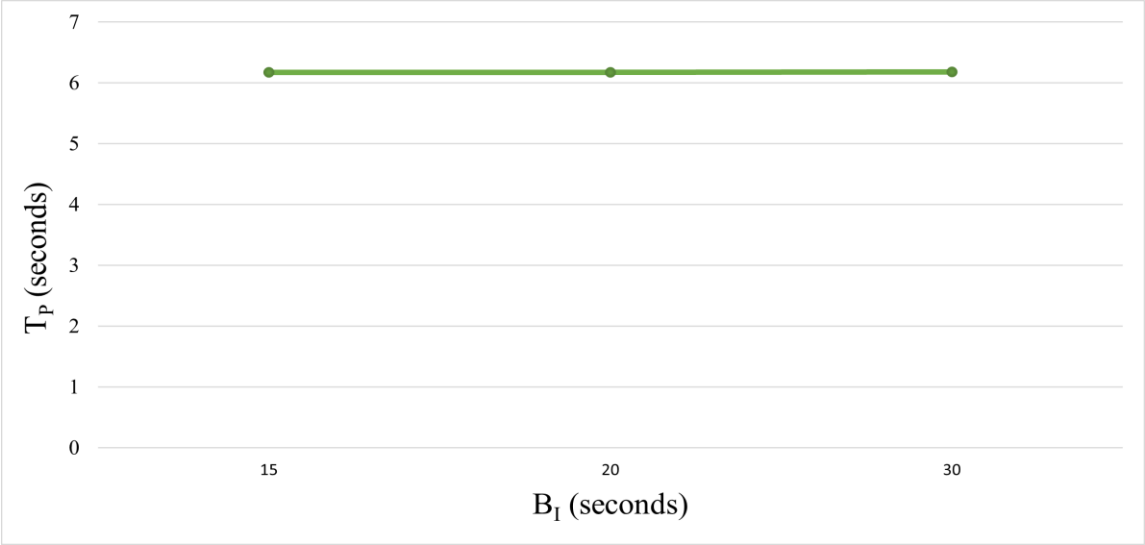


Figure 26: Average Batch Processing Latency for the Filter Method for Use Case-2 vs Batch Interval

impacted by a higher value of B_I. The higher B_I only provides a longer time window for the Filter method to complete processing of the current batch before the next batch arrives. Figure 27 displays the throughput (X) achieved with a different B_I. The throughput is observed to decrease with an increase in the B_I. This is because although the B_L is fixed, the total time taken by the Filter method to complete the processing of 80 batches increases with a higher value of B_I. Thus, the throughput changes with a higher value of B_I.

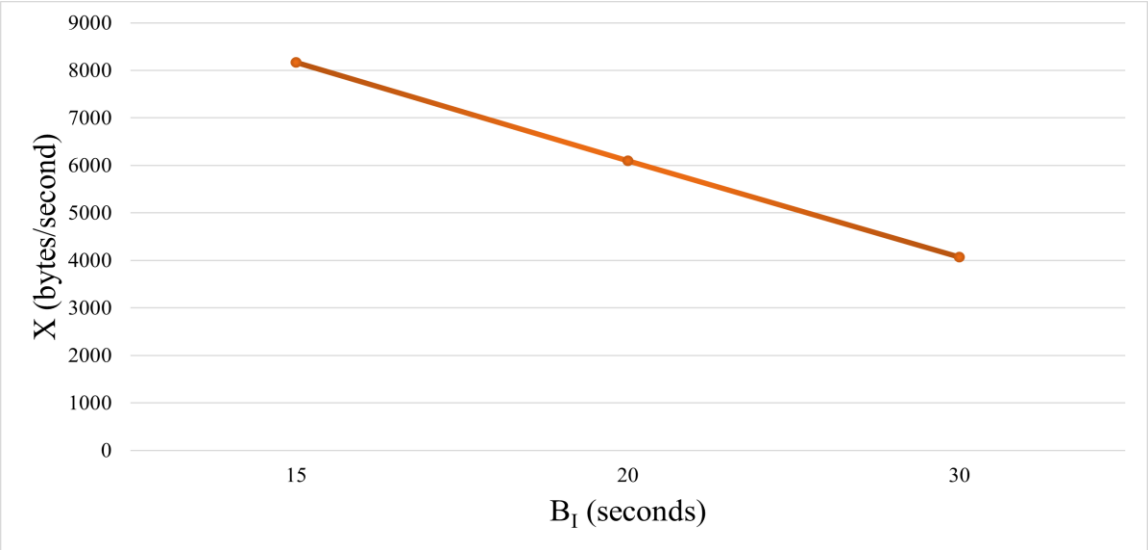


Figure 27: Throughput of the Filter Method for Use Case-2 vs Batch Interval

4.2.6.3 Effect of the Number of Executor Cores (N)

To see the effect of the number of processing resources on the Filter method, an experiment was performed by changing the number of executor cores from 1 to 12 in a worker node for a default batch interval (B_I) and a default length of batch in records (B_L). Figure 28 displays the average batch processing latency (T_P) achieved with different numbers of executor cores (N). A low average batch processing latency was achieved with an increase in the number of executor cores. For example, when the number of executor cores is 1, T_P is 12.793 seconds, whereas T_P decreases to 6.524 seconds when the number of executor cores is 8. This is because with an increasing number of executor cores in a worker node more batch records execute in parallel.

Figure 29 shows that increasing the number of executor cores does not have any impact on the system throughput. The throughput is observed to be the same with an increase in the number of executor cores. For the default B_L value of 1000 records, the total number of bytes remains the same. As a result, for an increasing number of cores, a given micro batch

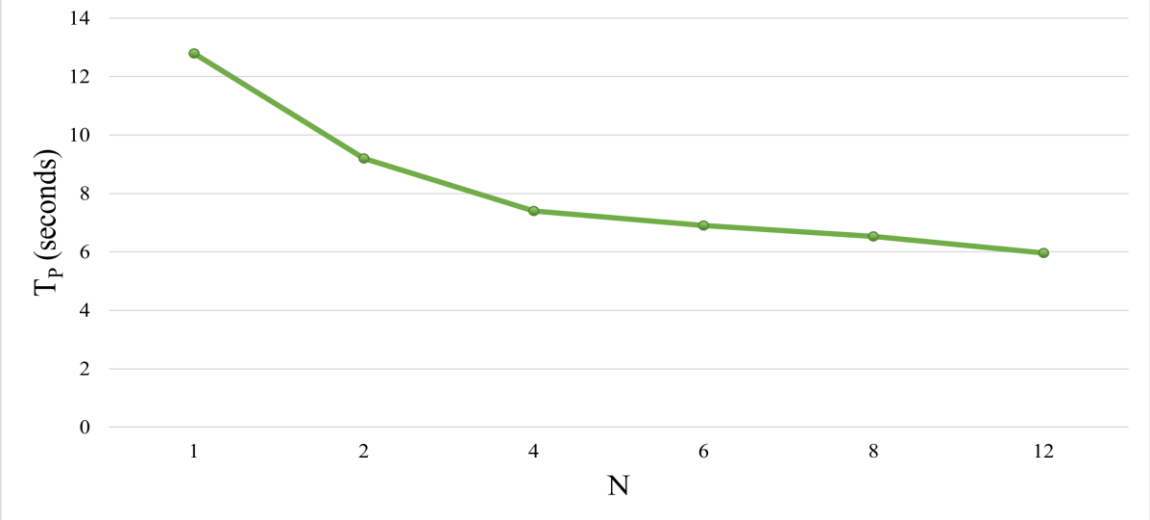


Figure 28: Average Batch Processing Latency for the Filter Method for Use Case-2 vs the Number of Executor Cores

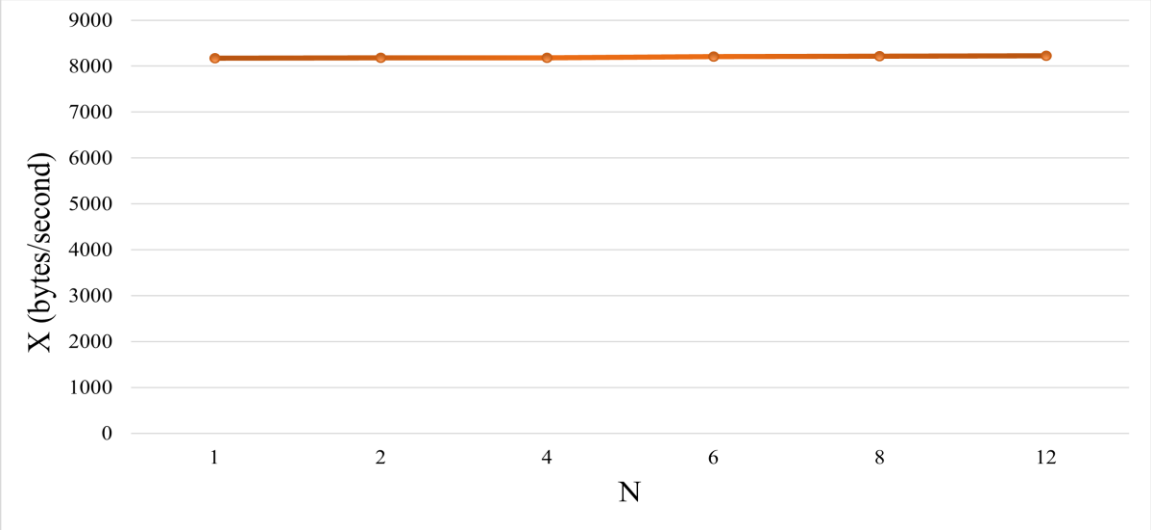


Figure 29: Throughput for the Filter Method for Use Case-2 vs the Number of Executor Cores

processes quickly, but the total time (T_E) required by the Filter method to process all batches does not change for a given number of executor cores.

4.2.6.4 Effect of the Number of Worker Nodes (N_W)

This experiment was performed by changing the number of worker nodes. The batch interval (B_I) and length of batch in records (B_L) are set to the default value. The number of worker nodes changes from 1 to 3, and each node has a default number of executor cores (N). The number of partitions in Kafka topic changes depending on the total number of executor cores in the cluster. Figure 30 shows that a low average batch processing latency (T_P) was achieved with an increase in the number of worker nodes. For example, when the number of worker nodes is 1, T_P is 6.524 seconds, whereas T_P is 1.092 seconds when the number of worker nodes is 3. That is an 83.2557% decrease in T_P when N_W is changing from 1 to 3. This is because more tasks are executing in parallel when the number of worker nodes is increased.

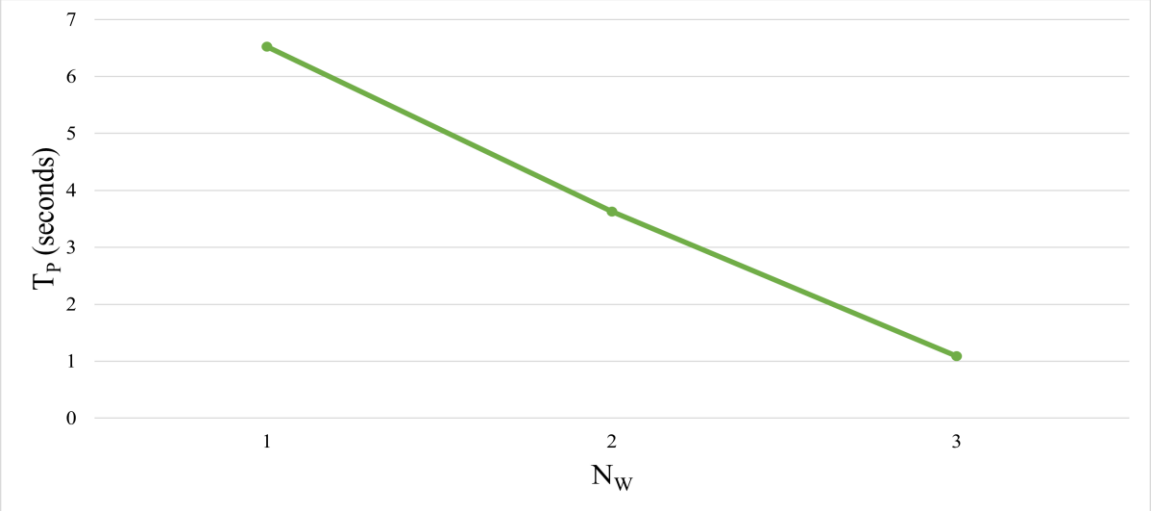


Figure 30: Average Batch Processing Latency for the Filter Method for Use Case-2 vs the Number of Worker Nodes

Figure 31 shows that the throughput remains the same with an increase in the number of worker nodes. This is because when the value of B_L and B_I are set to the default value, the total time (T_E) taken by the Filter method is the same for a given number of worker nodes. For the default B_L value, the number of bytes is constant for 80 batches. Thus, the Filter method takes an equal amount of time to process the 80 batches for a given number of worker nodes.

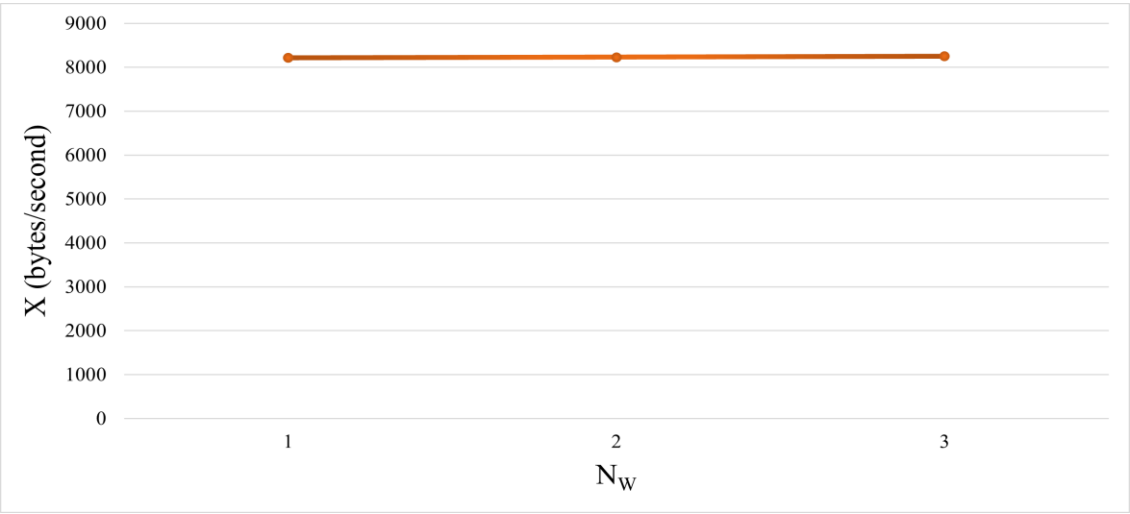


Figure 31: Throughput for the Filter Method for Use Case-2 vs the Number of Worker Nodes

4.3 Performance Evaluation of the Search Method

The experiments used to evaluate the performance of the Search method were run on the Spark Cluster. The Search method that searches within the filtered data is compared with the Search method that searches within the non-filtered data. In this research, experiments were performed by performing searches in two ways: sequential and parallel. Four different experiments were conducted to evaluate the performance of the Search method: both searching based on keywords and a sentence are investigated (see Table 4). In the first experiment, a sequential search based on keywords is performed for the filtered and non-filtered data. In the second experiment, the Search method searches within the filtered and non-filtered data based on keywords in parallel. In the third experiment, a sequential search based on a sentence runs within the filtered and non-filtered data. In the fourth experiment, a parallel search based on a sentence is performed within the filtered and non-filtered data. For the non-filtered raw dataset, a 4.1 GB raw dataset was used. The filtered data were retrieved using the Filter method by filtering the 4.1 GB non-filtered raw dataset based on the user preferences: “DATE”, “LOCATION” and “PERSON”.

In the sequential Search method, the number of worker nodes is set to 1 with 1 executor core. In parallel executions, the number of worker nodes is set to 2 with 2 executor cores each. This is because the Search method searches within a smaller volume of data (filtered data) and thus, does not require that many resources as used in the case of the Filter method. As discussed in Section 3.1.4, the filtered data are the preferred data retrieved from the Filter method by filtering the raw dataset based on user preferences. The configuration of Spark Cluster used in these experiments is discussed in Section 4.1.1. The performance metrics used to evaluate the performance of the Search are discussed next.

Table 4: Experiments for Search Method

Experiment No.	Search Method	Data Type	Search Based on
1	Sequential	Filtered Data	Keywords
		Non-filtered Data	
2	Parallel	Filtered Data	Keywords
		Non-filtered Data	
3	Sequential	Filtered Data	A Sentence
		Non-filtered Data	
4	Parallel	Filtered Data	A Sentence
		Non-filtered Data	

4.3.1 Performance Metrics

The performance of the Search method is evaluated by using two performance metrics:

(1) Computation Time (T_S): The computation time is the total time required by the Search method to complete the searching based on keywords or a sentence on a given number of resources. The computation time is measured by taking two timestamps using time.time() from Python Library: one when the Search method starts execution and one after the Search method finishes searching by generating the results. The computation time computed at the end by the Search method is the difference between these two timestamps (in seconds).

(2) Filtering Efficiency (E_F): Filtering Efficiency (E_F) is a ratio that illustrates the Filter method's efficiency**.** E_F is measured as the ratio between the computation time (T_S) achieved with non-filtered data and the computation time (T_S) achieved with filtered data. The E_F is calculated as follows:

$$E_F = \frac{T_{CT}}{T_{CS}} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [9]$$

Where T_{CT} is the computation time of the Search method performed within non-filtered data and T_{CS} is the computation time of the Search method performed within filtered data. A higher efficiency is preferable; it indicates the degree of reduction in T_S achieved by a search with the filtered data from that achieved by a search with the non-filtered data.

4.3.2 Results

Figure 32 and Figure 33 display the computation time (T_S) achieved with the sequential and parallel Search methods.

- *Sequential Search Method:* The first (blue) and second (red) bars in Figure 32 capture T_S, where a sequential search based on keywords was performed within non-filtered data and filtered data, respectively. The first (blue) and second (red) bars in Figure 33 show T_S achieved with the search based on a sentence, where a

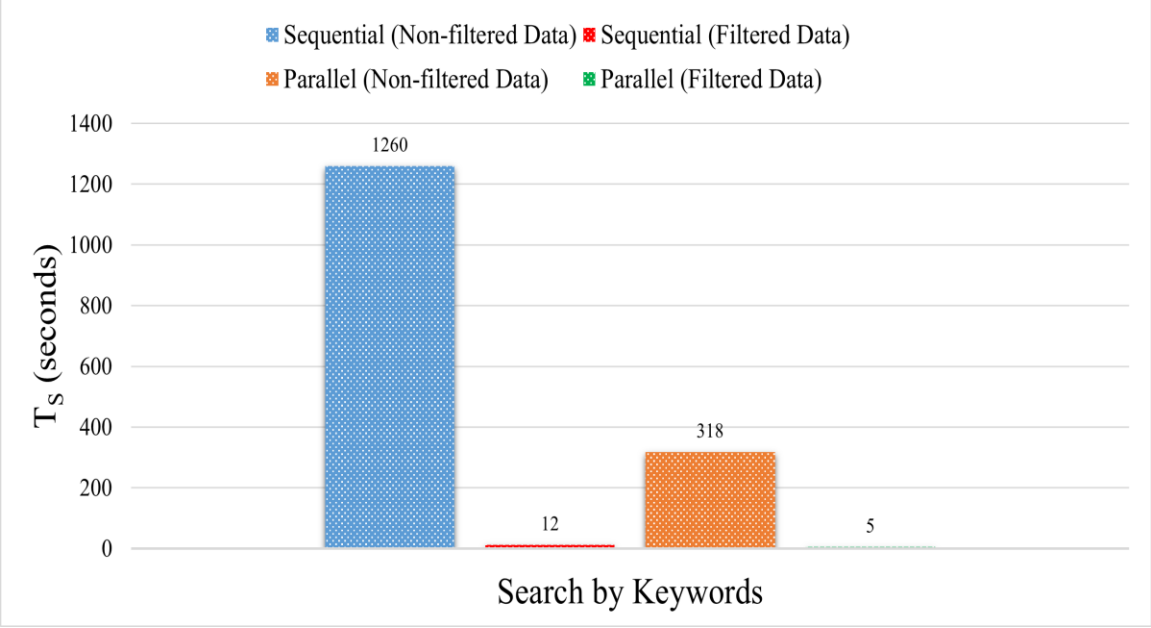


Figure 32: Computation Time for the Search Method Based on Keywords

sequential search is conducted within non-filtered and filtered data, respectively. From these first (blue) and second (red) bars in Figure 32 and Figure 33, it is shown that T_S achieved with a sequential search within filtered data is much lower than the sequential search within non-filtered data.

- *Parallel Search Method:* The third (orange) and fourth (green) bars in Figure 32 show T_S achieved for the keywords search, where the search is conducted in parallel within non-filtered and filtered data, respectively. The third (orange) and fourth (green) bars in Figure 33 capture T_S achieved, where the search based on a sentence is conducted in parallel within non-filtered and filtered data, respectively. These third (orange) and fourth (green) bars in Figure 32 and Figure 33 depicts that T_S achieved by searching within filtered data is much lower than searching within non-filtered data in parallel.

Figure 34 and Figure 35 capture the filtering efficiency (E_F) achieved while searching by keywords and by a sentence, respectively.

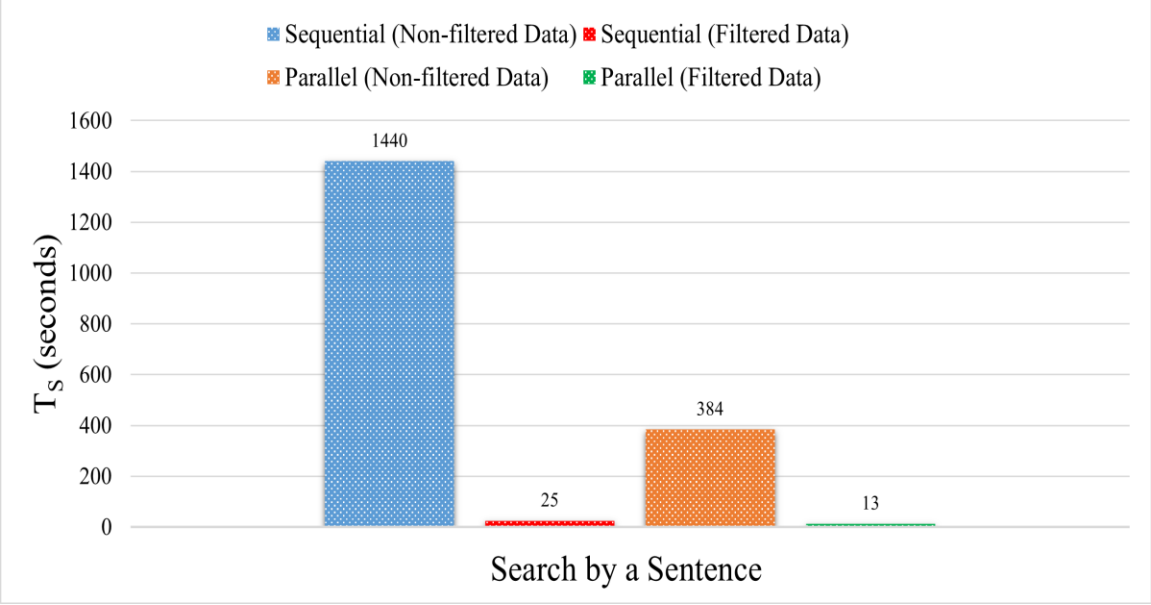


Figure 33: Computation Time for the Search Method Based on a Sentence

- *E_F Achieved with Sequential Search:* The orange bar in Figure 34 and Figure 35 capture E_F when a sequential search is performed based on keywords and a sentence, respectively. E_F achieved with the search by keywords is 105, which means that T_S for the search within non-filtered data is 105 times that of T_S for the search within filtered data. E_F is 57.6 when the search is performed by a sentence; thus, T_S for the search within non-filtered data is 57.6 times that of T_S for the search within filtered data.
- *E_F Achieved with Parallel Search:* The green bar in Figure 34 and Figure 35 capture E_F when a search is performed based on keywords and a sentence in parallel, respectively. E_F is 63.6 when the search is performed by keywords; thus, T_S for the search within non-filtered data is 63.6 times that of T_S for the search within filtered data. However, T_S for the search within non-filtered data is 29 times that of T_S for the search within filtered data when the search is performed by a sentence.

The high E_F achieved demonstrates the high filtering efficiency both in the case of sequential and parallel search.

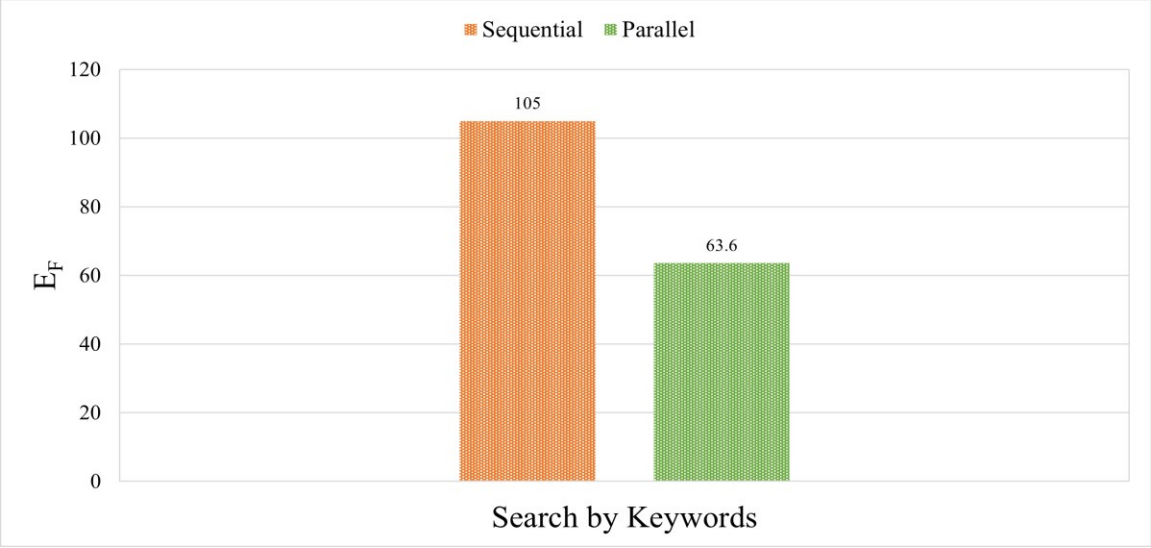


Figure 34: Filtering Efficiency of the Filter method While Searching by Keywords

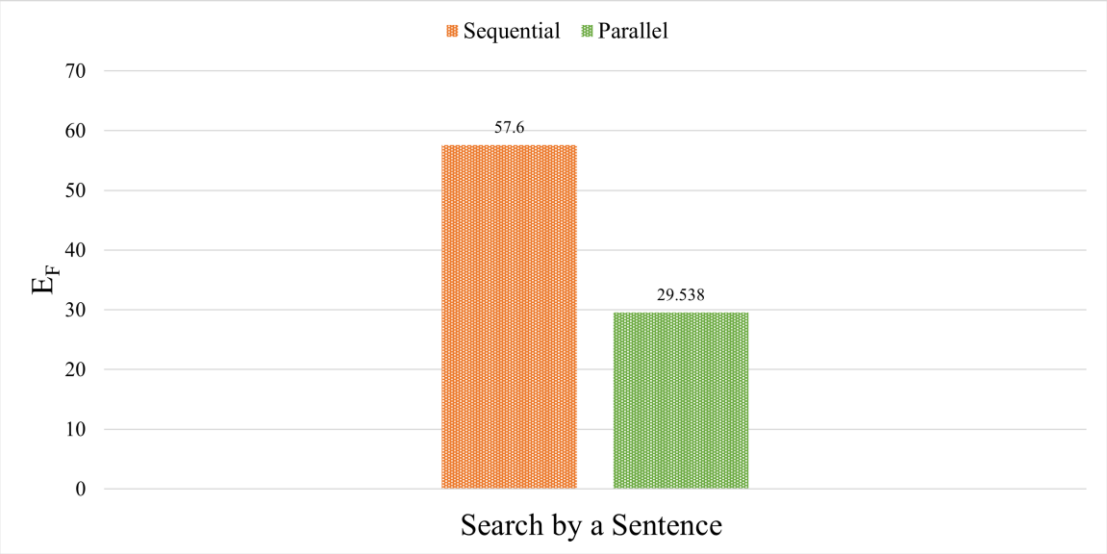


Figure 35: Filtering Efficiency of the Filter method While Searching by a Sentence

4.4 Performance of Machine Learning Models

As discussed in Section 2.3.1, five different kinds of machine learning algorithms: Multinomial Naïve Bayes [35], Multinomial Logistic Regression [33], Multilayer Perceptron [36], Decision Tree [43], and Random Forest [44] Classifier of Spark MLlib have been trained with the text classification dataset for creating classification models. Each of the five classifiers runs a sequence of pipeline stages (discussed in Section 2.3). The pipeline stages are tokenization and stop words removal, converting each word into feature vectors, encoding labels to label indices, learning the ML model from feature vectors and labels, and converting the predicted label to the original label. These pipeline stages are discussed in Section 2.3. Then, Spark's five-fold cross-validation is used to evaluate the model performance and find the best model (discussed in Section 2.3). The model evaluation is intended to estimate the accuracy of a model on future data [68]. To measure the model performance, model evaluation metrics are required [68]. The training

dataset used to train these algorithms and a model evaluation metric used to evaluate these machine learning models are discussed next.

4.4.1 Training Dataset

A labeled text classification dataset from the Cognitive Computation Group of University of Illinois [56][57] was used as a training and testing dataset for the machine learning algorithms. The labeled dataset has two attributes. One of the attributes is the label, which specifies the class of the second attribute which is text attribute. There are 50 hierarchical non-uniform distribution of class labels in this dataset to identify the text semantically, and these classes are modified to 10 labels in this research. The primary goal of this research is to filter the user-preferred data from a raw dataset using Apache Spark. Therefore, the classification dataset used in this research can be easily replaced by any other relevant text classification dataset.

4.4.2 Model Evaluation Metric

The performance of the five different machine learning models has been assessed based on F-Measure (F_1).

F-Measure (F_1): F-Measure considers both precision and recall. It is the harmonic mean of the model's precision and recall [67]. *F*-Measure is calculated as follows:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots [10]$$

In this thesis, F-measure is used to measure the accuracy of the five different machine learning models because of two reasons. First reason is that a good F-Measure indicates low false positives and low false negatives [99]. This means that the model is correctly identifying the class of the text [99]. Another reason is that the training dataset used to train the machine learning algorithms has non-uniform

distribution of class labels. The F-Measure metric finds a balance between precision and recall, which is useful in the scenarios when the dataset has a non-uniform distribution of class labels [98].

- *Precision*: Precision is the proportion of positive results that are truly positive [67]. It is calculated by dividing the number of true positives (t_p) by the number of false positives (f_p) plus the number of true positives (t_p) [67]. Precision is calculated as follows:

$$precision = \frac{t_p}{t_p + f_p} \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots [11]$$

- *Recall*: Recall is the ability of a model to correctly measure the percentage of actual positives that are correctly identified [67]. It is measured by dividing the number of true positives (t_p) by the number of true positives (t_p) plus the number of false negatives (f_n) [67]. Recall is calculated as follows:

$$recall = \frac{t_p}{t_p + f_n} \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots [12]$$

4.4.3 F-Measure of ML Models

Table 5 displays the F-Measure obtained for the five different machine learning models. According to Table 5, Multinomial Logistic Regression outperforms the other algorithms. Therefore, Multinomial Logistic Regression was used in the experiments to classify the raw dataset in the Filter method and to classify the user query by a sentence of the Search method.

4.5 Sample Application

Filtering and storing user-specified data can be useful for the user when the user wants to look at content that, for example, may be comprised of tweets containing certain keywords

Table 5: F-Measure of ML Models

ML Models	F-Measure
Multinomial Naïve Bayes Classifier	0.74
Multinomial Logistic Regression	0.94
Multilayer Perceptron Classifier	0.44
Decision Tree Classifier	0.55
Random Forest Classifier	0.51

or meeting minutes containing several names or topics of interest to the user. As indicated in Figure 6, the filtered data can also be further processed by an application program. A sample proof-of-concept prototype for such an application that uses filtered data extracted by the Filter method to generate a meeting schedule was implemented as a Python program. This program uses the user preferences "DATE" and "PERSON" to generate a meeting schedule and adds this schedule to the user's calendar of events. The methodology of the sample application is explained next.

4.5.1 Sample Application: High-Level Approach

The methodology of the sample application is explained with the help of Figure 36. First, the filtered data are processed by the "Filtered Data Processing" function. As mentioned earlier (Section 3.2), filtered data are stored as a comma-separated file with four columns: the filtered text, the entity classes recognized by the named entity recognition function, the extracted name entities, and the class predicted by the machine learning model. The "Filtered Data Processing" function shown in Figure 36 extracts those four columns and stores the data in a data structure implemented as a Python list. Next, the data structure is passed as an input to the "Find Class" function, which returns the sentences that belong to

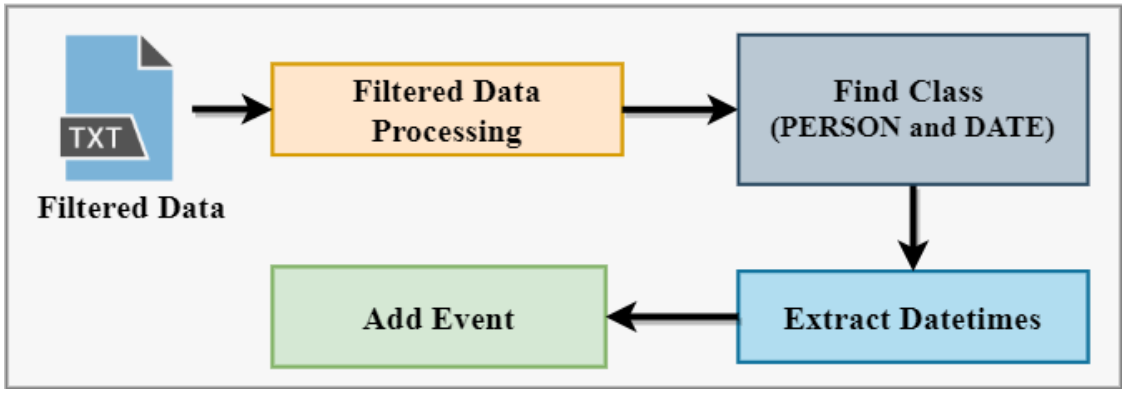


Figure 36: Methodology of the Sample Application

the "PERSON" and "DATE" classes. These sentences are then passed as inputs to the "Extract Datetimes" function. This function extracts the future dates and times from those sentences using regular expressions (regex) and returns the future dates, times, and corresponding sentences. The "Add Event" function adds those sentences as events in the user's Google calendar (using the user's Google Calendar credentials) or stores the events in a calendar (.ics) file so that the user can use the calendar file to import the events in any calendar.

4.5.2 Results

The result of a sample run of the program is shown in Figure 37. The sentences containing the date and times and the word "meeting" or "meet" were extracted from the filtered data, which were then used to add meeting events to the person's Google calendar. These filtered data were generated from the 4.1-GB raw dataset using the Filter method. The 4.1-GB raw dataset is discussed in Section 4.1.4. Figure 37 displays the content from the calendar file generated by the application. In the filtered data, two sentences belong to user preferences: the PERSON and DATE classes. The application extracts those two sentences, and these are added as events in the calendar file (on April 1 and April 15).

```
BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//user.preference//Meeting Schedule//EN
CALSCALE:GREGORIAN
X-WR-CALNAME;VALUE=TEXT:User Meeting Schedule
BEGIN:VEVENT
UID:16-297712031-user.preference
DTSTAMP;TZID=America/Toronto:20210415T100000
DTSTART;TZID=America/Toronto:20210415T100000
SUMMARY: Meeting is scheduled with John on April 15, 2021 at 10:00.
DESCRIPTION: Meeting is scheduled with John on April 15, 2021 at 10:00.
END:VEVENT
BEGIN:VEVENT
UID:73-1462290652-user.preference
DTSTAMP;TZID=America/Toronto:20210401T110000
DTSTART;TZID=America/Toronto:20210401T110000
SUMMARY: We will meet next on 2021-04-01 at 11 am.
DESCRIPTION: We will meet next on 2021-04-01 at 11 am.
END:VEVENT
END:VCALENDAR
```

Figure 37: Output of the Sample Application - Calendar File (.ics)

Chapter 5: Conclusions

Quintillion bytes of data are typically generated every day [46], and users of these data often need to extract useful information from them. To accurately derive the information a user is interested in, the user needs to process these data. This is challenging for the user due to the large time requirements of searching through large volumes of data. This thesis addresses this issue by proposing a Filter and Search method that reduces the data volume and speeds up the searching operations. The Filter method enables users to filter large amounts of data based on user preferences. This allows the user to discard all the unnecessary data and store only the required data to be used for various purposes while significantly minimizing the search time. However, for very large raw datasets this becomes a very time-consuming process. This thesis devises a technique that efficiently filters user-preferred data. This technique leverages the parallel processing platform as well as the machine learning library for achieving its goals.

The proposed Filter method is developed using the PySpark API on an Apache Spark cluster. It classifies the large raw dataset using a machine learning model and named entity recognition. Then, it generates the filtered data by filtering the categorized data based on user preferences. The user can then use the proposed Search method to search within the filtered data, which are significantly smaller in volume in comparison to the raw dataset. By using the Search method, the user can perform queries using keywords or a sentence to retrieve the preferred information from filtered data. If the user queries by a sentence, the sentence is classified by the machine learning model and named entity recognition. Then, the relevant data from filtered data containing these classes are returned by the Search method. These classes of this user-queried sentence are also saved as user preferences,

which are used by the Filter method in the future to filter raw datasets. When the user performs queries using keywords, the relevant data from the filtered data containing these keywords are returned by the Search method.

A proof-of-concept prototype was built and used to evaluate the performance of the Filter and the Search methods. Several experiments were performed on an Apache Spark cluster using various synthetically created raw data files. The experimental results for the Filter method demonstrate the viability of the Filter method while demonstrating the reduction of filtering latency and speedup achieved by the filtering technique due to its parallel execution. The experimental results for the Search method demonstrate the efficacy of the Filter method by comparing the search conducted within filtered data with the search performed within non-filtered data.

The experiments in the Filter method were conducted for two use cases: Use Case-1 and Use Case-2. The experiments for the Search method were conducted using the filtered data generated by the Filter method for each use case. A date extraction application demonstrating the effective use of the proposed data filtering and storage technique is also presented (see Section 4.5). A summary of key observations is presented next for the two use cases. Directions for future work are also included at the end of the chapter.

5.1 Use Case-1 of the Filter Method

The first use case considers that the raw dataset is comprised of multiple files in a local directory. The Filter method in a Spark cluster distributes this raw dataset among the executor cores of the cluster and generates filtered data based on user-provided preferences by filtering the raw dataset in parallel. A Spark cluster consisting of one master node and three worker nodes is configured on Amazon EC2 cloud infrastructure (see Section 4.1.1).

To evaluate the Filter method for Use Case-1, experiments were performed based on various workload and system parameters. Three performance metrics: computation time, speedup, and efficiency are used to evaluate the performance of the Filter method for Use Case-1. Key insights resulting from the performance experiments performed with the Filter method using Use Case-1 include the following.

- *Effect of Raw Dataset Size:* The total raw dataset size (S_R) has a significant impact on the computation time (T_C) for the Filter method (see Figure 9). T_C is observed to increase with an increase in S_R. Thus, the Filter method requires more time to process the larger raw dataset.
- *Effect of File Size and the Number of Files:* T_C depends only on S_R and seems to be insensitive to the size of the individual files and the number of files constituting the raw dataset (see Figure 17). T_C increases with an increase in file sizes and their corresponding number of files in a raw dataset.
- *Effect of Raw Dataset Category:* The raw dataset category (C_R) seems to have no impact on T_C. The reason behind this is that the total raw dataset size is the same for both categories (raw dataset comprised of multiple files of the same size and multiple files with different sizes).
- *Effect of Parallelism in Execution:* The degree of parallelism in the Filter method is determined by both the number of worker nodes and the number of executor cores. The speedup (S(N)) increases with an increase in the number of worker nodes (N_W) and executor cores (N). This is accompanied by a decrease in efficiency (E(N)). The reason behind this is that with a higher value of N_W and N, more tasks can be executed concurrently by the Filter method, thus the decrease in T_C.

- *Executor Core vs Worker Node Parallelism:* There are two ways to change parallelism: one is achieved by varying the number of executor cores in a worker node and the other one is achieved by varying the number of worker nodes while keeping the number of executor cores same in a cluster. T_C seems to be higher when the value of N in a worker node is higher in comparison to a higher N_W with lower N on each. This is due to the intercommunication delay between the worker nodes and driver node; a worker node needs more time to gather the results from all the executor cores when N is higher.
- *Effect of Raw Dataset Partitioning:* Data partitioning in executor cores has a significant impact on T_C. T_C achieved with the equal distribution-based partitioning (E_{DP}) of a raw dataset among executor cores is lower than that achieved with centralized partitioning (C_{DP}), where each executor core handles one file at a time. This is because E_{DP} is balancing the load equally on each executor core of the worker node. In C_{DP}, all the executor cores do not finish at the same time because each executor core handles a different number of files.

5.2 Use Case-2 of the Filter Method

The second use case focuses on streaming data. The data producer sends the raw data in small batches to an Apache Kafka topic at fixed time intervals. The Filter method in the Spark cluster reads these batches from the Kafka topic and filters them to generate filtered data based on user preferences. The Spark and Kafka cluster is configured on an Amazon EC2 cloud infrastructure (see Section 4.2.2). Experiments were performed based on various workload and system parameters to evaluate the performance of the Filter method for Use Case-2. Average batch processing latency (T_P) and throughput (X) are used to

evaluate the performance of the Filter method. Key insights from the results of the experiments performed are briefly discussed.

- *Effect of the Length of Batch in Records:* An increase in the length of batch in records (B_L) increases the processing time of each batch. A higher value of B_L indicates that a greater number of records (sentences) in a batch needs to be processed by the Filter method. Thus, the average batch processing latency (T_P) and throughput (X) increase with an increase in B_L.
- *Effect of Batch Interval:* An increase in the batch interval (B_I) does not seem to have any impact on the T_P achieved by the Filter method. But the throughput is observed to decrease with a higher value of B_I because the elapsed time increases with a higher value of B_I.
- *Effect of Parallelism in Execution:* T_P decreases with an increase in the number of worker nodes (N_W) and executor cores (N). This is because more records of a given batch can be processed concurrently by the Filter method with a higher value of N_W and N; thus, T_P decreases. However, an increase in N_W and N does not have any impact on throughput. As B_L and B_I are fixed, the throughput remains the same for different values of N_W and N.

5.3 Search Method

Experiments have been presented to evaluate the performance of the Search method with respect to searches based on keywords or a sentence. A sequential and parallel search have been performed within the filtered data and non-filtered data. These experiments are performed in the same Spark cluster configured on Amazon EC2 cloud infrastructure that is discussed in Section 4.1.1. The Search method has been evaluated based on computation

time (T_S) and filtering efficiency (E_F). T_S achieved with the Search method when a search is performed within filtered data is observed to be much lower than when a search is performed within non-filtered data, which leads to a higher filtering efficiency (E_F). This demonstrates the efficacy of the proposed filtering technique.

5.4 Future Research

This section presents a few directions for future research.

- In this thesis, a synthetically generated raw dataset and stream data have been used to evaluate the Filter method. The Search method searches within the filtered data containing all the user-preferred data from a synthetically generated raw dataset. Investigations based on real-world datasets form an important direction for future research.
- Converting live speech to text and filtering the text based on user preferences using the Filter method warrants investigation.
- This thesis used fixed batch intervals for the experiments. Investigating other distributions for batch intervals such as exponential distribution for batch intervals (B_I) is worthy of research.
- FIFO, the default scheduler for Spark, has been used as the scheduling policy. The investigation of Fair Scheduler Pool, another job scheduling technique supported by Spark, can form a future research direction.
- Filtering efficiency is measured based on the filtered data achieved using the machine learning model and the named entity recognition function. The investigation of how filtering efficiency changes based on the filtered data achieved without the machine learning model and the named entity recognition function can

form a future research direction. Comparing the filtering efficiency achieved with machine leaning and that achieved without machine learning can provide important insights into the impact of machine learning on the techniques described in this thesis.

References

[1] Apache Spark, “Apache Spark - Unified Analytics Engine for Big Data”, [Online]. Available at https://spark.apache.org/. [Accessed: 31-Mar-2021].

[2] H. Karau and R. Warren, *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. Beijing: O'Reilly, 2017.

[3] E. Shaikh, I. Mohiuddin, Y. Alufaisan and I. Nahvi, "Apache Spark: a Big Data Processing Engine," *2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*, 2019, pp. 1-6.

[4] K. Aziz, D. Zaidouni, and M. Bellafkih, “Leveraging Resource Management for Efficient Performance of Apache Spark,” *Journal of Big Data*, vol. 6, no. 1, pp 78 – 101, 2019.

[5] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Made, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia and A. Talwalkar, “MLlib: Machine Learning in Apache Spark,” *Journal of Machine Learning Research*, vol. 17, no. 34, pp 1-7, 2016.

[6] S. Chowdhury and M. P. Schoen, "Research Paper Classification using Supervised Machine Learning Techniques," *Intermountain Engineering, Technology and Computing (IETC) Conference*, 2020, pp. 1-6.

[7] Venkatesh and K. V. Ranjitha, "Classification and Optimization Scheme for Text Data using Machine Learning Naïve Bayes Classifier," *IEEE World Symposium on Communication Engineering (WSCE) Conference*, 2018, pp. 33-36.

[8] N. Kalcheva and N. Nikolov, "Laplace Naive Bayes Classifier in the Classification of Text in Machine Learning," *International Conference on Biomedical Innovations and Applications (BIA)*, 2020, pp. 17-19.

[9] R. Abyaad, M. R. Kabir and S. Hasan, "A Novel Approach to Categorize News Articles from Headlines and Short Text," *IEEE Region 10 Symposium (TENSYMP) Conference*, 2020, pp. 162-165.

[10] J. Dai and C. Chen, "Text Classification System of Academic Papers Based on Hybrid Bert-Bigru Model," *12th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, 2020, pp. 40-44.

[11] M. Ghosh and G. Sanyal. 2018, “Document Modeling with Hierarchical Deep Learning Approach for Sentiment Classification,” *2nd International Conference on Digital Signal Processing (ICDSP)*, 2018, pp. 181–185.

[12] W. Yang, Y. Fu, and D. Zhang, "An Improved Parallel Algorithm for Text Categorization," *International Symposium on Computer, Consumer and Control (IS3C)*, 2016, pp. 451-454.

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

[14] J. Su, T. Hong, J. Li and J. Su, "Personalized Content-Based Music Retrieval by User-Filtering and Query-Refinement," *Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, 2018, pp. 177-180.

[15] J. Zhang and P. Pu, “Refining Preference-Based Search Results Through Bayesian Filtering,” *12th international conference on Intelligent user interfaces (IUI), Association for Computing Machinery*, 2007, pp. 294–297.

[16] S. V. Siva reddy and S. Saravanan, "Performance Evaluation of Classification Algorithms in the Design of Apache Spark based Intrusion Detection System," *5th International Conference on Communication and Electronics Systems (ICCES)*, 2020, pp. 443-447.

[17] H. Ahmed, E. M. G. Younis and A. A. Ali, "Predicting Diabetes using Distributed Machine Learning Based on Apache Spark," *International Conference on Innovative Trends in Communication and Computer Engineering (ITCE)*, 2020, pp. 44-49.

[18] A. Ed-Daoudy and K. Maalmi, "Performance Evaluation of Machine Learning Based Big Data Processing Framework for Prediction of Heart Disease," *International Conference on Intelligent Systems and Advanced Computing Sciences (ISACS)*, 2019, pp. 1-5.

[19] A. Ed-daoudy and K. Maalmi, "Application of Machine Learning Model on Streaming Health Data Event in Real-Time to Predict Health Status Using Spark," *International Symposium on Advanced Electrical and Communication Technologies (ISAECT) Conference*, 2018, pp. 1-4.

[20] J. Gui and Q. Wang, "Topic Modeling of News Based on Spark MLlib," *14th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, 2017, pp. 224-228.

[21] S. Deaton, D. Brownfield, L. Kosta, Z. Zhu and S. J. Matthews, "Real-Time Regex Matching with Apache Spark," *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1-6.

[22] H. Elzayady, K. M. Badran and G. I. Salama, "Sentiment Analysis on Twitter Data Using Apache Spark Framework," *13th International Conference on Computer Engineering and Systems (ICCES)*, 2018, pp. 171-176.

[23] R. Filgueira et al., "defoe: A Spark-Based Toolbox for Analysing Digital Historical Textual Data," *15th International Conference on eScience (eScience)*, 2019, pp. 235-242.

[24] A. Alexopoulos, G. Drakopoulos, A. Kanavos, S. Sioutas and G. Vonitsanos, "Parametric Evaluation of Collaborative Filtering over Apache Spark," *5th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, 2020, pp. 1-8.

[25] S. Al-Rasbi and T. Elsayed, "Can We Build a Search Engine over Spark?," *IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*, 2020, pp. 345-350.

[26] A. K. Mohideen, S. Majumdar, M. St-Hilaire and A. El-Haraki, "A Graph-Based Indexing Technique to Enhance the Performance of Boolean AND Queries in Big Data Systems," *20th IEEE/ACM International Symposium on Cluster, Cloud, and Internet Computing (CCGRID)*, 2020, pp. 677-680.

[27] J. Jabbar, W. JunSheng, L. Weigang and I. Urooj, "Implementation of Search Engine with Lucene in the Document Management System," *IEEE 2nd International Conference on Electronics and Communication Engineering (ICECE)*, 2019, pp. 1-4.

[28] G. Liang, Y. G. Yan, M. Wang, X. L. Lian, M. S. Li and W. H. Tang, "Classification for Text Data from the Power System Based on Improving Naive Bayes," *12th IEEE PES Asia-Pacific Power and Energy Engineering Conference (APPEEC)*, 2020, pp. 1-6.

[29] X. Liu and Y. Zhang, "A Kind of Personalized Advertising Recommendation Method Based on User-Interest-Behavior Model," *8th International Symposium on Next Generation Electronics (ISNE) Conference*, 2019, pp. 1-4.

[30] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu and J. Gao, "Deep Learning Based Text Classification: a Comprehensive Review," *arXiv.org*, [Online]. Available at https://arxiv.org/abs/2004.03705. [Accessed: 09-Feb-2021].

[31] K. Kowsari, K. Jafari Meimandi, M. Heidarysafa, S. Mendu, L. E. Barnes and D. E. Brown, "Text Classification Algorithms: a Survey," *Information Journal*, vol. 10, no. 4, pp. 150-218, 2019.

[32] Apache Spark, "ML Pipelines", [Online]. Available at http://spark.apache.org/docs/3.0.1/ml-pipeline.html. [Accessed: 11-Jan-2021].

[33] Apache Spark, "Multinomial Logistic Regression", [Online]. Available at http://spark.apache.org/docs/3.0.1/ml-classification-regression.html#multinomial-logistic-regression. [Accessed: 11-Jan-2021].

[34] R. Dua, M. S. Ghotra, and N. Pentreath, *Machine Learning with Spark*, Birmingham: Packt Publishing, 2016.

[35] Apache Spark, "Naïve Bayes", [Online]. Available at http://spark.apache.org/docs/3.0.1/ml-classification-regression.html#naive-bayes. [Accessed: 11-Jan-2021].

[36] Apache Spark, “Multilayer Perceptron Classifier, [Online]. Available at http://spark.apache.org/docs/3.0.1/ml-classification-regression.html#multilayer-perceptron-classifier. [Accessed: 11-Jan-2021].

[37] Apache Spark, “RDD Programming Guide”, [Online]. Available at http://spark.apache.org/docs/3.0.1/rdd-programming-guide.html. [Accessed: 12-Jan-2021].

[38] Spacy, “spaCy · Industrial-Strength Natural Language Processing in Python”, [Online]. Available at https://spacy.io/. [Accessed: 11-Jan-2021].

[39] Thinc, “Thinc”, [Online]. Available at https://thinc.ai/docs/. [Accessed: 11-Jan-2021].

[40] Apache Kafka, “Apache Kafka”, [Online]. Available at https://kafka.apache.org/. [Accessed: 20-Feb-2021].

[41] M. Carter, “Your Complete Guide to Apache Kafka Architecture”, Instaclustr. [Online]. Available at https://www.instaclustr.com/apache-kafka-architecture/. [Accessed: 04-Mar-2021].

[42] Apache Spark, “Cluster Mode Overview”, [Online]. Available at https://spark.apache.org/docs/latest/cluster-overview.html. [Accessed: 15-Jan-2021].

[43] Apache Spark, “Decision Trees - RDD-Based API”, [Online]. Available at http://spark.apache.org/docs/3.0.1/mllib-decision-tree.html. [Accessed: 12-Jan-2021].

[44] Apache Spark, “Random Forests”, [Online]. Available at http://spark.apache.org/docs/3.0.1/mllib-ensembles.html. [Accessed: 12-Jan-2021].

[45] Apache Spark, “Spark Standalone Mode”, [Online]. Available at http://spark.apache.org/docs/3.0.1/spark-standalone.html. [Accessed: 05-Feb-2021].

[46] J. Bulao, “How Much Data Is Created Every Day in 2021? [You'll be shocked!]”, TechJury. [Online]. Available at https://techjury.net/blog/how-much-data-is-created-every-day/. [Accessed: 20-Mar-2021].

[47] Solvexia, “15 Big Data Problems You Need to Solve”, [Online]. Available at https://www.solvexia.com/blog/15-big-data-problems-you-need-to-solve. [Accessed: 20-Mar-2021].

[48] IBM, “What is HDFS? Apache Hadoop Distributed File System”, [Online]. Available at https://www.ibm.com/analytics/hadoop/hdfs. [Accessed: 01-Feb-2021].

[49] A. Bekker, “The 'Scary' Seven: Big Data Challenges and Ways to Solve Them”, ScienceSoft. [Online]. Available at https://www.scnsoft.com/blog/big-data-challenges-and-their-solutions. [Accessed: 20-Mar-2021].

[50] Apache Hadoop, “Apache Hadoop”, [Online]. Available at https://hadoop.apache.org/. [Accessed: 01-Mar-2021].

[51] DataFlair, “13 Big Limitations of Hadoop & Solution to Hadoop Drawbacks”, [Online]. Available at https://data-flair.training/blogs/13-limitations-of-hadoop/. [Accessed: 01-Mar-2021].

[52] G. Jevtic, “Hadoop vs Spark: Detailed Comparison of Big Data Frameworks”, phoenixNAP. [Online]. Available at https://phoenixnap.com/kb/hadoop-vs-spark. [Accessed: 28-Mar-2021].

[53] Apache Spark, “Extracting, Transforming and Selecting Features”, [Online]. Available at http://spark.apache.org/docs/3.0.1/ml-features.html. [Accessed: 06-Feb-2021].

[54] City of Ottawa, “City Council & Committee Agendas & Minutes”, [Online]. Available at https://app06.ottawa.ca/cgi-bin/docs.pl?lang=en. [Accessed: 05-Dec-2019].

[55] Amazon, “Amazon EC2”, [Online]. Available at https://aws.amazon.com/ec2/. [Accessed: 12-Mar-2021].

[56] X. Li and D. Roth, “Learning Question Classifiers,” *19th International Conference on Computational Linguistics*, 2002, pp. 556-562.

[57] X. Li and D. Roth, “Learning Question Classifiers: The Role of Semantic Information,” *Journal of Natural Language Engineering*, vol. 12, no. 3, pp. 229-249, 2005.

[58] S. Shah, Y. Amannejad, D. Krishnamurthy and M. Wang, "Quick Execution Time Predictions for Spark Applications," *15th International Conference on Network and Service Management (CNSM)*, 2019, pp. 1-9.

[59] Rev, “Voice Recorder App: Audio Recording App”, [Online]. Available at https://www.rev.com/voicerecorder. [Accessed: 20-Feb-2021].

[60] Medcorder, “Record & Share Doctor Appointments”, [Online]. Available at https://www.medcorder.com/. [Accessed: 20-Feb-2021].

[61] Alberta Health Services, “My Care Conversations App”, [Online]. Available at https://www.albertahealthservices.ca/info/page16144.aspx. [Accessed: 20-Feb-2021].

[62] S. Kozlovski, “Thorough Introduction to Apache Kafka™”, Hacker Noon. [Online]. Available at https://hackernoon.com/thorough-introduction-to-apache-kafka-6fbf2989bbc1. [Accessed: 20-Feb-2021].

[63] Kafka Python, “KafkaProducer”, [Online]. Available at https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html. [Accessed: 20-Feb-2021].

[64] Python, “Python Programming Language”, [Online]. Available at https://www.python.org/ [Accessed: 01-Jan-2021]

[65] Anil, “Conversation Between Doctor and Patient [Five Scenarios]”, Lemon Grad. [Online]. Available at https://lemongrad.com/conversation-between-doctor-and-patient/. [Accessed: 01-Sept-2020].

[66] Prasanna, “English Conversation Between Doctor and Patient in Four Simple Scenarios”, A Plus Topper. [Online]. Available at https://www.aplustopper.com/conversation-between-doctor-and-patient/. [Accessed: 01-Sept-2020].

[67] DeepAI, “F-Score”, [Online]. Available at https://deepai.org/machine-learning-glossary-and-terms/f-score. [Accessed: 15-Feb-2021].

[68] S. Mutuvi, “Introduction to Machine Learning Model Evaluation”, Medium. [Online]. Available at https://heartbeat.fritz.ai/introduction-to-machine-learning-model-evaluation-fa859e1b2d7f. [Accessed: 10-Mar-2021].

[69] Apache Spark, “Welcome to Spark Python API Docs!”, [Online]. Available at https://spark.apache.org/docs/3.0.1/api/python/. [Accessed: 11-Mar-2021].

[70] Databricks, “What is PySpark?”, [Online]. Available at https://databricks.com/glossary/pyspark. [Accessed: 11-Mar-2021].

[71] NLTK, “Natural Language Toolkit”, [Online]. Available at https://www.nltk.org/. [Accessed: 11-Mar-2021].

[72] Apache Spark, “Job Scheduling”, [Online]. Available at https://spark.apache.org/docs/3.0.1/job-scheduling.html. [Accessed: 11-Feb-2021].

[73] Hackr.io, “Hadoop vs Spark: a Head to Head Comparison in 2021”, [Online]. Available at https://hackr.io/blog/hadoop-vs-spark. [Accessed: 11-Mar-2021].

[74] N. Diep, “Big Data Analytics: Apache Spark vs. Apache Hadoop”, Medium. [Online]. Available at https://towardsdatascience.com/big-data-analytics-apache-spark-vs-apache-hadoop-7cb77a7a9424. [Accessed: 11-Mar-2021].

[75] U. Sivarajah, M. M. Kamal, Z. Irani, and V. Weerakkody, “Critical Analysis of Big Data Challenges and Analytical Methods,” *Journal of Business Research*, vol. 70, no. 29, pp. 263–286, 2017.

[76] R. Buyya, C. Vecchiola, and S. T. Selvi, *Mastering Cloud Computing: Foundations and Applications Programming*, Waltham, MA.: Morgan Kaufmann, 2013.

[77] B. Chanda and S. Majumdar, “Filtering and Storing User-preferred Data: an Apache Spark Based Approach,” *IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress*, 2020, pp. 679-685.

[78] DeZyre, “Top 5 Apache Spark Use Cases”, [Online]. Available at https://www.dezyre.com/article/top-5-apache-spark-use-cases/271. [Accessed: 01-Mar-2021].

[79] Apache Hadoop, “Apache Hadoop YARN”, [Online]. Available at https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. [Accessed: 23-Jan-2021].

[80] Apache Mesos, “Apache Mesos”, [Online]. Available at https://mesos.apache.org/. [Accessed: 23-Jan-2021].

[81] Kubernetes, “Kubernetes”, [Online]. Available at https://kubernetes.io/. [Accessed: 23-Jan-2021].

[82] J. Brownlee, “A Gentle Introduction to K-Fold Cross-Validation”, Machine Learning Mastery. [Online]. Available at https://machinelearningmastery.com/k-fold-cross-validation/. [Accessed: 22-Mar-2021].

[83] T. Ivanov and J, Taaffe, “Exploratory Analysis of Spark Structured Streaming,” *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2018, pp 141–146.

[84] Apache Spark, “Structured Streaming Programming Guide”, [Online]. Available at http://spark.apache.org/docs/3.0.1/structured-streaming-programming-guide.html. [Accessed: 02-Mar-2021].

[85] Apache Spark, “Machine Learning Library (MLlib) Guide”, [Online]. Available at https://spark.apache.org/docs/3.0.1/ml-guide.html. [Accessed: 02-Mar-2021].

[86] Apache Spark, “Spark SQL, DataFrames and Datasets Guide”, [Online]. Available at https://spark.apache.org/docs/3.0.1/sql-programming-guide.html. [Accessed: 02-Mar-2021].

[87] Apache Spark, “ML Tuning: Model Selection and Hyperparameter Tuning”, [Online]. Available at https://spark.apache.org/docs/3.0.1/ml-tuning.html. [Accessed: 05-Mar-2021].

[88] Simplilearn, “Top 10 Big Data Applications Across Industries”, [Online]. Available at https://www.simplilearn.com/tutorials/big-data-tutorial/big-data-applications. [Accessed: 25-Mar-2021].

[89] Scala, “The Scala Programming Language”, [Online]. Available at https://www.scala-lang.org/. [Accessed: 31-Mar-2021].

[90] R, “The R Project for Statistical Computing”, [Online]. Available at https://www.r-project.org/. [Accessed: 31-Mar-2021].

[91] Oracle, “Java”, [Online]. Available at https://www.oracle.com/java/. [Accessed: 31-Mar-2021].

[92] Programiz, “Linear Search”, [Online]. Available at https://www.programiz.com/dsa/linear-search. [Accessed: 03-Mar-2021].

[93] M. T. Tun, D. E. Nyaung and M. P. Phyu, "Performance Evaluation of Intrusion Detection Streaming Transactions Using Apache Kafka and Spark Streaming," *International Conference on Advanced Information Technologies (ICAIT)*, 2019, pp. 25-30.

[94] R. Chakraborty and S. Majumdar, “Priority Based Resource Scheduling Techniques for a Resource Constrained Stream Processing System,” *4th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT)*, 2017, pp 21–31.

[95] Apache ZooKeeper, “Apache ZooKeeper”, [Online]. Available at https://zookeeper.apache.org/. [Accessed: 12-Mar-2021].

[96] J. Nabi, “Machine Learning — Text Processing”, Towards Data Science. [Online]. Available at https://towardsdatascience.com/machine-learning-text-processing-1d5a2d638958. [Accessed: 03-Mar-2021].

[97] S. S. Kumar and A. Rajini, "Extensive Survey on Feature Extraction and Feature Selection Techniques for Sentiment Classification in Social Media," *10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2019, pp. 1-6.

[98] S. Raschka, “How can the F1-score help with dealing with class imbalance?”, Sebastianraschka. [Online]. Available at https://sebastianraschka.com/faq/docs/computing-the-f1-score.html. [Accessed: 10-May-2021].

[99] A.I. Wiki, “Evaluation Metrics for Machine Learning - Accuracy, Precision, Recall, and F1 Defined”, Pathmind. [Online]. Available at https://wiki.pathmind.com/accuracy-precision-recall-f1. [Accessed: 10-May-2021].

[100] K. M. Sagayam, P. M. Bruntha, M. Sridevi, M. Renith Sam, U. Kose, and O. Deperlioglu, “A cognitive perception on content-based image retrieval using an advanced soft computing paradigm,” *Advanced Machine Vision Paradigms for Medical Image Analysis*, 2021, pp. 189–211.

[101] DeepAI, “Feature Extraction”, [Online]. Available at https://deepai.org/machine-learning-glossary-and-terms/feature-extraction. [Accessed: 11-May-2021].