A Study on Hadoop, Spark, and Spark Optimization

Book Reviews Analysis Using Machine Learning

Fatema Tuz Zohora
University of Stavanger, Norway
ft.zohora@stud.uis.no

Konrad Krzysztof Jarczyk
University of Stavanger, Norway
kk.jarczyk@stud.uis.no

ABSTRACT

User reviews play a crucial role in understanding a user's perspective on a product. They are as essential to any business as the quality of the product itself. In this project, we aimed to utilize these reviews to gain insight into books from the users' perspective. First, we converted the understanding of the reviews into numerical ratings through sentiment analysis. Second, we recommended the top categorical book with similar reviews.

With this use case, we studied Hadoop and Spark. We used Hadoop to read and structure our datasets and Spark for further data processing to achieve the desired outcome. We optimized our implementation using various solutions available for Spark and evaluated the performance of different infrastructures in this big data project.

Further, we demonstrated the effectiveness of the pipeline in producing the desired category ranking and highlighted the scalability of the code. This project shows the potential of using user reviews and big data technologies to create tailored book recommendations based on sentiment analysis.

KEYWORDS

Hadoop, Spark, Spark Optimization, Natural Language Processing - Sentiment Analysis

1 INTRODUCTION

Data is now considered the fourth paradigm of science as the volume of data is increasing exponentially and they can help in decision-making efficiently. The world of computer science is incorporating different ways to use this large volume of data. The increase in the volume of data is not the only challenge. The nature of data and how they are stored is also changing. As a result, computing on local computers is not a suitable choice for the data nowadays. We need some practical solutions to this problem. That is where different data-intensive systems play a significant role.

Hadoop and Spark are two data-intense systems we are studying for this purpose using a use case of analysis on book reviews collected from the Amazon website. The reviews contain perspectives from which one can better understand a book. They also play an important role in decision-making for the service providers like Amazon, etc. Hence, we will analyze the sentiment of the user reviews available for a book and convert them to numerical values and combine the values of all the scores generated from the reviews of a book on average. We will recommend books by different categories after normalizing this score.

Supervised by Thomasz Wiktorski and Rui Paulo Maximo Pereira Mateu Esteves .

Project in Computer Science (DAT500), IDE, UiS
2023.

The use of Hadoop and Spark makes our implementation more scalable and parallelization makes the processing faster. We are exploiting the benefits of the MapReduce programming paradigm of Hadoop for reading and structuring our unstructured dataset. Then we are using Spark for processing the data to reach the recommendations. As the final study, we would optimize our implementation using different features of Spark.

We are making the following contribution to this paper:

- We are using HDFS for storing our data.
- We are using MRJob for structuring our dataset.
- We are using Spark for processing our dataset to enhance the behavior and reach the goal of our use case - Recommending a list of books by category on the generated scores from the review.
- We are using Sentiment analysis from SparkNLP to get the initial sentiment on the reviews.
- We are performing different optimization to tackle different performance issues observed throughout the implementation.

The code implementation of our process is presented on the GitHub Repository:
https://github.com/jarczykkonrad/dat500-project

2 BACKGROUND

This section contains background information necessary for different aspects of our project. We will begin by introducing the dataset we choose to work with. We will shed light on insights into Hadoop and Spark architecture, what benefits they hold, and relative information to our implementation choices. Additionally, we will discuss the sentiment analysis of natural language processing used for our review analysis and connect how the system developed will work together. Finally, we will discuss different performance issues related to Spark and how to optimize them.

2.1 Dataset

We have collected a dataset of book reviews on Amazon from Kaggale. The dataset is divided into two text files and connected as shown in Figure 1 below:

The dataset contains 3M text reviews in different languages on 212404 unique books in different languages. We are using the reviews in English contain about 3.04GB of unstructured text data.

2.2 Apache Hadoop

Apache Hadoop is an Open-source, Scalable, and Fault-tolerant Framework with features to handle big data. It can process large volume of data economically on cluster of commodity hardware. The Figure 2 [2] shows this intricate architecture works:

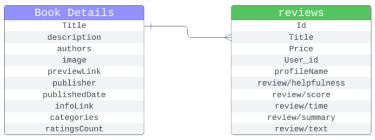

Figure 1: Amazon Book Reviews

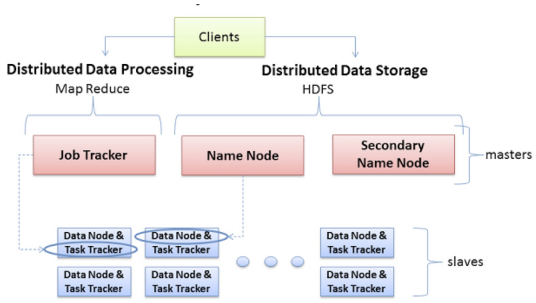

Figure 2: Hadoop Architecture

The architecture has two core components: HDFS, and MapReduce. Through this architecture design of the Hadoop, system is specifically suitable for parallel processing of large volumes of data as it can run in distributed mode on a cluster of multiple machines connected by a network [11].

2.2.1 HDFS. HDFS, abbreviated as Hadoop Distributed File System is the file system responsible for storing data in a distributed manner. It was developed to handle huge volumes of data. The data will be split into blocks and stored across multiple machines. These blocks are then replicated by the replication factor which gives the characteristics of fault-tolerance and reliability to Hadoop. The whole distributed system works in a master-slave manner. Master node with its daemon **namenode** stores the metadata and manages the data nodes. On the other hand, data nodes by the daemon **datanode** actually store data and performs the jobs.

2.2.2 MapReduce. MapReduce is what makes Hadoop work so efficiently. It is the programming Framework that allows us to perform distributed processing of large volumes of data. It divides work into a set of independent tasks and takes these tasks closer to data rather than moving data to the computation. In this way, it makes Hadoop efficient, economic and also scalability is ensured as the tasks are independent.

2.3 Apache Spark

Apache Spark is the successor framework to Hadoop. The architecture of this programming paradigm provides high performance. The Figure 3 [8] shows the Spark architecture in a simplified way:

The significant feature that makes Spark special is the ability to process data in memory. Additionally, Spark supported real-time processing by performing stream processing, as well batch processing can be performed when required. Spark supports interactive

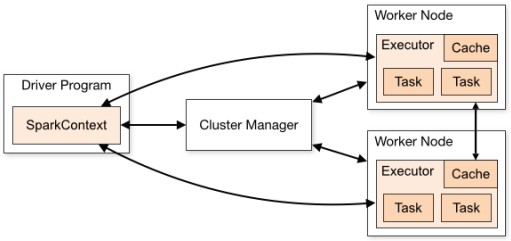

Figure 3: Spark Architecture

queries and iterative algorithms. It has its own cluster manager for hosting applications, while, it uses HDFS for storing purposes and YARN for running its application.

2.3.1 Spark RDD. Spark RDD stands for Resilient Distributed Dataset is the backbone of Apache Spark. This is the core data structure of Apache Spark. The concept of Spark datasets and data frames was introduced later in this data structure. RDD helps Spark to achieve efficient data processing [9] by dividing the dataset into logical partitions [9]. These logically partitioned data can be distributed over different nodes of the cluster for distributed computation.

2.3.2 PySpark. Apache Spark integrates Python with an API named PySpark. It combines Apache Spark's power of scaling data processing and analysis with Python's ease of use and learnability [7]. All of the features of Spark like, Spark SQL, Dataframes, and Machine Learning(MLib) are supported by PySpark.

2.3.3 Delta Table. Delta Lake is the optimized storage layer for storing data and tables. Delta Lake is fully compatible with Apache Spark APIs and was developed to support a single copy of data for batch and streaming operations with incremental processing at scale. Delta Lake is open-source software that extends Parquet data files with a file-based transaction log for ACID transactions and scalable metadata handling [1]. Delta Lake refuses to write with wrongly formatted data. It also allows the CRUD operations (insert, update, merge, and delete), which are usually not available in raw files [1].

2.4 Hadoop vs Spark

The MapReduce of Hadoop allows us to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance. the only way to reuse data between computations (e.g. between two MapReduce jobs) is to write it to an external stable storage system (e.g. HDFS). Figure 4 explains how the framework works.

As a result, this incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow. Most of the Hadoop applications, spend more than 90% of the time doing HDFS read-write operations.

The main feature of Spark is it is the ability to process data in memory. It means that the data processing happens in memory, rather than being written on the disk for better performance. These kinds of features outperform Hadoop in processing, but the framework still relies on file systems or databases for the permanent

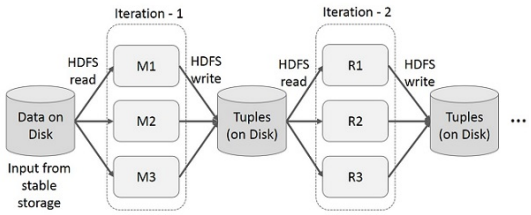

Figure 4: Iterative Operations on MapReduce

storage of data [11]. Figure 5 shows the iterative operations on Spark.

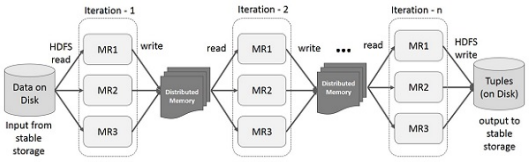

Figure 5: Iterative Operations on Spark

Spark will store intermediate results in a distributed memory instead of Stable storage (Disk) which makes the system faster. If the Distributed memory (RAM) is not sufficient to store intermediate results, then it will store those results on the disk [10].

2.5 Functional Choices - Data Structure and Processing

MapReduce works by directly reading and writing the data to the disk. Additionally, for performing different queries parallel, it reads and writes to the disk individually for each query as shown in Figure 6. Though, it ensures fault tolerance, each of these disks to memory read and write is expensive [10].

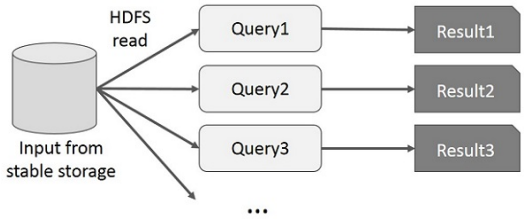

Figure 6: Interactive Operations on MapReduce

Therefore, for storing in a stable storage system (e.g. HDFS), and structuring the data, Hadoop MapReduce is a better choice.

On the other hand, if different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times for Spark as shown in Figure 7 [10].

As a result, it computes the data more efficiently, yet less expensively. Therefore, it is a better choice for data processing and algorithm implementation.

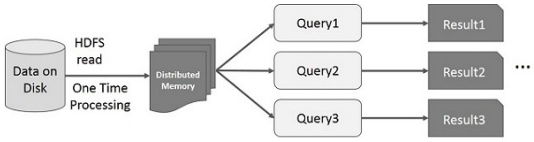

Figure 7: Interactive Operations on Spark

2.6 Natural Language Processing - Sentiment Analysis

Natural Language Processing (NLP) is the way to employ different computational techniques for the synthesis and analysis of natural texts available in different forms [4]. Sentiment Analysis is one of these computation techniques. It is used for understanding the emotion present in the text or speech. It is a subjective analysis of people's opinions, attitudes, and emotions towards an entity and an entity can be a product or service, etc. [4].

For analyzing the sentiments in the reviews for the books, we are using ViveknSentimentModel with support to SparkNLP. This model classifies text into negative and positive categories by examining individual words and short sequences of words (n-grams) and comparing them with a probability model [5]. This is a supervised sentiment classification model based on the Naïve Bayes algorithm. The Bayes theorem is,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{1}$$

This can be stated as that we can find the probability of A happening, given that B has occurred. Here, B is the evidence and A is the hypothesis. Simplifying the conditional independence assumption, this is modified to Naïve Bayes algorithm as stated below:

$$P(c_i|d) = \frac{(\prod P(x_i|c_j))P(c_j)}{P(d)} \tag{2}$$

Where c denotes a class, d denotes a document, and x_i are the individual words of the document.

This model enhances this simple Naive Bayes classifier to match the classification accuracy of more complicated models for sentiment analysis by choosing the right type of features and removing noise by appropriate feature selection [6].

2.7 Use Case

The use case we have selected to study the different properties of Hadoop and Spark is analyzing the book reviews we have collected. Upon analysis, we are going to rank them on the generated rating by categories.

Figure 8 delineates the major algorithm flow we are incorporating to reach this goal.

2.8 Possible Spark Performance Issues and Optimizations

We use different data-intensive systems like Hadoop, Spark, etc. to process big volumes of data more economically and efficiently. However, their performance can get hindered due to a lack of configuration management or, programming inefficiently. There are five most commonly observed performance issues related to Spark -

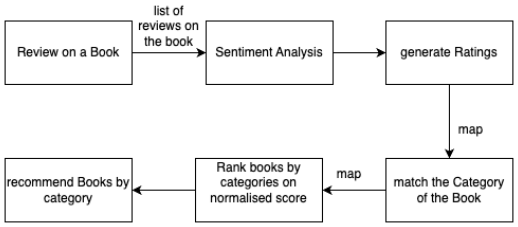

Figure 8: Use case: Book Recommendation by Categories

Storage, Spill, Skew, Shuffle, and Serialization. Some of these issues are explained below:

2.8.1 Storage. If the data is stored in a non-optimal way, storage issues arises [3]. Three of the main problems associated with Storage are - Tiny Files, Scanning, and Schemas - Inferring & Merging Schemas. Issue of the tiny files is observed while handling partition files less than the default value (e.g. 128 MB). Scanning issues may arise while scanning a long list of files in a single directory or in the case of highly partitioned datasets in multiple levels folders. Lastly, different schema issues can arise depending on the file format used. Most of these issues can be solved by proper hardware and software configuration [1].

2.8.2 Skew. If the partition size is highly imbalanced, the issue of skew is observed [3]. When using Spark, data is commonly read in evenly distributed partitions of 128 MB. Applying different transformations to the data can then result in some partitions becoming much bigger or smaller than their average as shown in Figure 9.

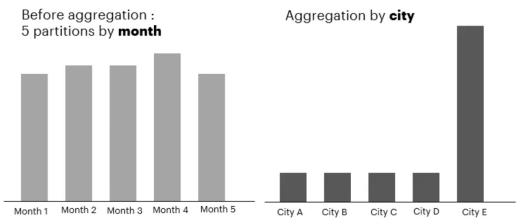

Figure 9: Spark Performance Issue - Skew

However, small amounts of skew are acceptable in some circumstances. Skew can result in Spill and Out Of Memory (OOM) errors. To mitigate skews, solutions like - salting, Adaptive Skew Join for Spark3.1, and Databricks' [proprietary] Skew Hint for Spark2.x are approached.

2.8.3 Spill. This issue of Spill is possibly the most significant contributor to poorly performing Spark jobs. Spill occurs when a given partition is simply too large to fit into RAM as shown in Figure 10. In this case, an RDD is first moved from RAM to disk and then back to RAM just to avoid OOM errors. As disk reads and writes can be quite expensive to compute, it should be avoided as much as possible.

In order to handle the spill, we have to address the root cause first. Generally, two approaches that can be used to mitigate spill are

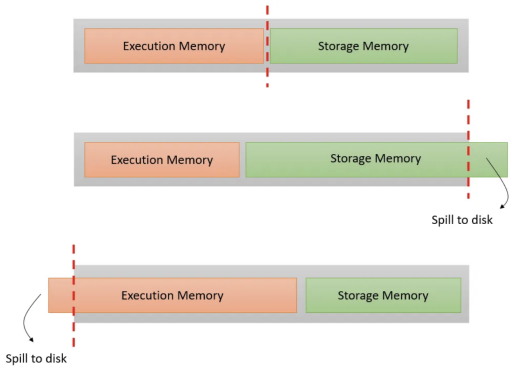

Figure 10: Spark Performance Issue - Spill

instantiating a cluster with more memory per worker or increasing the number of partitions (e.g. making partitions smaller).

2.8.4 Shuffle. Shuffle results from moving data between executors when performing wide transformations (e.g. joins, groupBy, etc.) or some actions such as count as shown in Figure 11. Mishandling of shuffle problems can result in other performance issues (e.g. skew). However, many shuffle operations are actually quite fast. So instead of focusing on these shuffle operations, targetting performance issues with greater impact would yield better results.

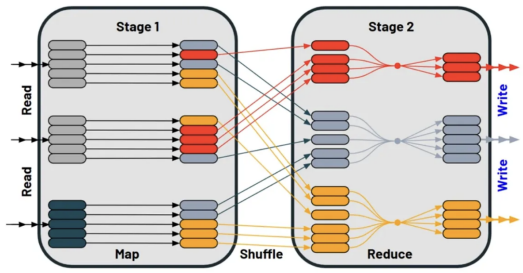

Figure 11: Spark Performance Issue - Shuffle

We can mitigate shuffling by reducing the amount of data being shuffled, reducing network I/O by using fewer and larger workers, denormalizing the datasets, etc.

2.8.5 Serialization. Serialization encompasses all the problems associated with the distribution of code across clusters. Serialization of Python codes can even be more complicated since the code has to be pickled and an instance of the Python interpreter has to be allocated to each executor. Serialization issues can arise when integrating codebases with legacy systems (e.g. Hadoop), 3rd party libraries, and custom frameworks [3]. This issue can be reduced by avoiding using UDFs or Vectorized UDFs, which act like a black box for the Catalyst Optimizer.

Therefore, to take full advantage of Spark's distributed computation capabilities, writing highly optimized code is an essential part of using Spark. As a result, having domain knowledge is fundamental in order to successfully make the best use of Spark capabilities, even after the release of Apache Spark 3 [3].

2.8.6 Caching. In Apache Spark, caching refers to the process of storing RDDs or DataFrames in memory or on disk, so that they can be quickly accessed and reused in subsequent Spark operations. It is useful when an RDD or data frame is used multiple times in different Spark operations because it can save the time and resources needed to recompute the RDD or data frame for each time it is used. However, caching every RDD or DataFrame is not always beneficial, as it can lead to excessive memory usage and slow down the overall performance of the Spark application.

2.8.7 Repartitioning. Repartitioning is a property in Apache Spark. It is the process of changing the number of partitions of an RDD or data frame. An RDD or DataFrame in Spark is divided into partitions, which are distributed across the nodes in a cluster. Repartitioning can be useful when the original number of partitions is not optimal for the Spark operations being performed, such as when the number of partitions is too low and not utilizing all available resources or too high and causing unnecessary data shuffling between nodes.

Nonetheless, repartitioning is an expensive operation, as it involves moving data between nodes in the cluster, and it should be used judiciously.

2.8.8 Broadcast Join. There are multiple ways available in Apache Spark to perform a join operation optimally. A broadcast join is a technique for joining two datasets where one of the datasets is small enough to fit into the memory of each worker node in the Spark cluster. In a broadcast join, the smaller dataset is broadcasted to all worker nodes in the cluster, so that each node can perform the join locally without having to shuffle the data between nodes. This can result in significant performance improvements, as shuffling data between nodes can be a time-consuming process, especially when dealing with large datasets.

3 IMPLEMENTATION

The section describes the most important parts of used methods to structure collected data and further processing to develop necessary outcomes and optimize in a data-intensive system paradigm. We used the Hadoop MRJob programming paradigm only to structure the data as desired, Spark is incorporated for processing the data to reach the goal of the use case for the reason described in 2.5 and later to study different optimization techniques.

3.1 Data Structuring

The collected data was unstructured 3.04GB of a dataset on book reviews from Amazon. For the purpose of our implementation, we structured the data and saved it in HDFS as a .CSV file. We are going to use this .CSV file for further implementation.

Listing 1: Data Structuring using MRJob

```
from mrjob.job import MRJob
from mrjob.protocol import RawValueProtocol

class TabToPipeSeparator(MRJob):
    OUTPUT_PROTOCOL = RawValueProtocol

    def mapper(self, _, line):
        pipe_sep_line = line.replace('\t', '|')
        yield None, pipe_sep_line

if __name__ == '__main__':
    TabToPipeSeparator.run()
```

The reviews we collected were textual data, and they contained commas in the text. As in .CSV file the default delimiter is the comma, separating columns by comma divides the reviews with commas in them as different columns. To avoid this situation, we used **tab** instead of **comma** to separate our columns.

3.2 Spark - Helper Libraries

For this study, we are using PySpark. These are the helper and machine learning libraries we are going to use that we used for our implementation.

Listing 2: Imported Libraies

```
from pyspark import *
from pyspark.sql import functions as F
from pyspark.ml import Pipeline

import sparknlp
from sparknlp.base import *
from sparknlp.annotator import *

from delta import *
```

PySpark libraries included for this study are the helper libraries, while we are including SparkNLP library and its helper library for Sentiment Analysis on our collected reviews.

3.3 Spark Configuration

In Spark configuration, besides the necessary lines for using delta tables and sentiment analysis model we added amount of executor cores equal to four, which was the most optimal solution for our code.

Listing 3: Spark Configuration

```
builder = pyspark.sql.SparkSession.builder.appName("
 MyApp").master("yarn").config("spark.kryoserializer.
 buffer.max", "2000M") \
.config("spark.sql.extensions", "io.delta.sql.
 DeltaSparkSessionExtension") \
.config("spark.sql.catalog.spark_catalog", "org.
 apache.spark.sql.delta.catalog.DeltaCatalog") \
.config("spark.executor.cores", 4) \

spark = configure_spark_with_delta_pip(builder,
 extra_packages=["com.johnsnowlabs.nlp:spark-nlp_2
 .12:4.4.0"]).getOrCreate()
```

3.4 Create and Enforce Data Schema

To store and work with our processed data from HDFS, we are defining the schema structure we need to load the data, as well as the path to load the data as the initial step to work with Spark.

Listing 4: Creating and Enforcing Schema in Spark

```
books_data2 = "hdfs:/project/books_data2/part*"
books_ratings2 = "hdfs:/project/books_ratings2/part*"

schema_books2 = "Title STRING, Description STRING,
 Authors STRING, Image STRING, PreviewLink STRING,
 Publisher STRING, PublishedDate DATE, InfoLink
 STRING, Categories STRING, RatingsCount INT"
schema_ratings2 = "ID STRING, Title STRING, Price INT
 , User_id STRING, User_name STRING, Helpfulness
 STRING, Score FLOAT, Timestamp INT, Summary STRING,
 Review STRING"

df_books2 = spark.read \
    .format("csv") \
    .schema(schema_books2) \
    .option("header", "false") \
    .option("delimiter", "|") \
    .load(books_data2)
df_ratings2 = spark.read \
    .format("csv") \
    .schema(schema_ratings2) \
    .option("header", "false") \
    .option("delimiter", "|") \
    .load(books_ratings2)
```

After that, we are loading the data with schema enforcement for further use from .CSV file present in HDFS.

3.5 Data Pre-processing

Our goal is to recommend books by category. Therefore, if a book does not have a category available, we are dropping those books from our dataset in our preprocessing step.

Listing 5: Data Preprocessing

```
df_books = df_books.repartition(150)
df_ratings = df_ratings.repartition(150)

df_books = df_books.na.drop(subset=["Categories"])
```

Further processing to clean the data is performed before sentiment analysis in 3.6.

3.6 Sentiment Analysis

In this section we are focusing on sentiment analysis of our collected reviews on books. However, we are using models recommended tokenizer to clean and tokenize the words. As the model is enhancing simple Naïve Bayes algorithm as described in 2.6, we are also using the recommended normalizer so that we can ensure the best performance of the model.

Listing 6: Sentiment Analysis Model Implementation

```
document = DocumentAssembler() \
    .setInputCol("Review") \
    .setOutputCol("document")

token = Tokenizer() \
    .setInputCols(["document"]) \
    .setOutputCol("token")

normalizer = Normalizer() \
    .setInputCols(["token"]) \
    .setOutputCol("normal")

vivekn = ViveknSentimentModel.pretrained() \
    .setInputCols(["document", "normal"]) \
    .setOutputCol("result_sentiment")

finisher = Finisher() \
    .setInputCols(["result_sentiment"]) \
    .setOutputCols(["Sentiment"]) \
    .setCleanAnnotations(True)

pipeline = Pipeline().setStages([document, token,
 normalizer, vivekn, finisher])

pipelineModel = pipeline.fit(df_ratings)
df_ratings = pipelineModel.transform(df_ratings)
```

The whole procedure is carried out in the described pipeline, and generated sentiments, either **positive** or **negative**.

Listing 7: Convertion of Sentiment to Numerical Value

```
# replacing positive/negative with value
df_ratings = df_ratings.withColumn('Sentiment', F.
 when(df_ratings.Sentiment[0] == "positive", 1).
 otherwise(0)).cache()
```

From these generated sentiments, we are converting them into numerical values for further processing and score generation.

3.7 Score Generation

Initially, we were calculating the score from all the reviews present for a book using the following formula:

$$score = 0.04 * Count + 0.3 * 2 * \frac{1}{n} \sum_{i=i}^{n} Score + 0.3 * 10 * \frac{1}{n} \sum_{i=i}^{n} Sentiment \tag{3}$$

The score calculated here is a normalized score over a scale of 0 to 10.

Listing 8: Score Generation

```
df_ratings = df_ratings.repartition("Title")

# First, calculate the count for each title
df_count = df_ratings.groupBy("Title").agg(F.count("*
 ").alias("Count"))
df_ratings.unpersist()

# Filter the DataFrame based on the count
df_count_filtered = df_count.filter(df_count["Count"]
  > 50).cache()

#using brodacast as optimization
df_ratings_filtered = df_ratings.join(F.broadcast(
 df_count_filtered), "Title")

# Now, calculate the average scores and sentiments

df_ratings = df_ratings_filtered.groupBy("Title", "
 Count").agg(
    F.mean("Score").alias("Scores"),
```

```
        F.mean("Sentiment").alias("Sentiments")
    )

    df_count_filtered.unpersist()

    # upper limit for amount of reviews
    df_ratings_updated = df_ratings.withColumn('Count', F
     .when(F.col("Count") > 100, 100).otherwise(F.col("
     Count")))

    # calculating weighted values to get the final score
     of each book
    final_ratings = df_ratings.withColumn(
        "Final_score", 0.04 * F.col("Count") + 0.3 *2* F.
     col("Scores") + 0.3 *10* F.col("Sentiments")
    )
```

In order to optimize the final score generation. We are doing first filtering on minimal amount of reviews for a particular movie to be included in the final ranking. We are choosing 50 as minimal and 100 as maximal amount of reviews.

3.8 Books with Category

Up until now, we only focused on generating a rating on each book from the reviews available. However, we intend to recommend top books for each category.

Listing 9: merging schemas to know the category of the book

```
    final_ratings = final_ratings.select("Title", "
     Final_score").cache()

    second_df = df_books.select("Title", F.col("
     Categories").getItem(0).alias("Category"))

    # joining tables
    final_ratings = final_ratings.join(F.broadcast(
     second_df), on="Title")

    merged_df = final_ratings.withColumn(
        "Title_final_score",
        F.concat(F.col("Title"), F.lit(" - "), F.
     format_number(F.col("final_score"), 2))
    )
```

The information of the generated ratings on a book and the categorical information is on different schemas. For that reason, we are merging them together when the title matches.

3.9 Ranked List of Books by Category

This is the final step in our proposed algorithm. The books are arranged by the information available in each book. We are changing this arrangement by pivoting the schema by the category of the book.

Listing 10: Ranked List of Books

```
    # making ranking
    window = Window.partitionBy('Category').orderBy(F.
     desc('Final_score'))

    # using extra counter column
    df = merged_df.withColumn('counter', F.row_number().
     over(window))

    # pivoting
    pivot_df = df.groupBy('counter').pivot('Category').
     agg(F.first('Title_final_score'))
    pivot_df = pivot_df.drop('counter')
```

We are recommending the books in descending order depending on the scores they hold.

3.10 Updating Delta Table

We are merging the new result generated with the previously recorded result. In our merge operation, we are updating the delta table only when titles are matching.

Listing 11: Updating Delta Table with Merge

```
    delta_table_ratings.alias("ratings1") \
    .merge(df_ratings_updated_temp.alias("
     ratings1_updated"),
           "ratings1.Title = ratings1_updated.Title AND
     ratings1.Scores = ratings1_updated.Scores") \
    .whenMatchedUpdate(set={"Count": "ratings1_updated.
     Count"}) \
    .whenNotMatchedInsertAll() \
    .execute()

    delta_table_ratings.toDF().write.format("delta").mode
     ("overwrite").save("/tmp/ratings_count")
```

3.11 Spark Optimization

Spark is optimizing the use of components related to the cluster and configurations by enabling *AdaptiveQueryExecution* for Spark3. As we are using Spark3.3.2, our implementation is running in the default optimal way Spark considers. However, it is not guaranteed to always optimize to run the programs in the best environment. There are a lot of possible ways to custom-optimize while programming. Some of the ways we have included are:

3.11.1 Caching. We are caching the data frames in memory as indicated in code listing 6. However, after completing the processing on the data frames on which, keeping them in the cache would restrict us from using the memory optimally for further use. That is why we are *unpersisting* as shown in code listing 8 the cached data frames when they are not in use for further implementation.

3.11.2 Repartitioning. We are initially repartitioning our datasets for equal distribution of the data over executors. We initially implemented this *repartition* with the default value of 200. Later on, we found that our algorithm works best when repartitioned by a value of 150 as shown in code listing 5.

We used repartitioning after score generation as listed in code listing 8 to distribute the partition over the different titles of the books found as the number of reviews observed over the book titles were giving us the optimal behavior.

3.11.3 Reducing Unnecessary Data. Before joining the schema which has the final rating generated for each book to the schema with the categories of the book, we selected the columns necessary

from the $final_ratings$ Schema to reduce the overload in comparing and joining those columns as shown in the code listing 9.

3.11.4 Broadcast Join. The number of categories of the books observed was 540, which is very small in comparison to the number of books available in our dataset(212404 unique books) we used broadcast join to optimize the joining over partitions as shown in code listing 9.

4 EXPERIMENTAL RESULT AND EVALUATION

In this section, we are presenting the final output of our use case which is a recommendation of a list of books by category based on the normalized score we have generated and the effect of optimization we have observed while running our implementation.

4.1 Categorical Book Recommendation

An example of categorical recommendation can be clearly observed in Figure 12.

```
+-----------------------------------------------------------------------
|Fiction
+-----------------------------------------------------------------------
|Prodigal Son (Dean Koontz's Frankenstein, Book 1) - 8.63
|Cruel and Unusual (G K Hall Large Print Book Series (Cloth)) - 8.48
|King Rat - 8.39
|The Dharma Bums - 8.02
|From Potter's Field - 7.99
|COCKTAILS FOR THREE - 7.88
|Nappily Ever After - 7.77
|Pushing Ice - 7.51
|Edge of Danger - 5.82
```

Figure 12: Recommended books of Fiction Category

4.2 Comparison Results on Performance Optimization

The properties of Hadoop and Spark provide us with a scalable means to perform our computations optimally. We evaluated our system for three runs each with optimization and without optimization and calculated the average execution time. Further, we experimented with different volumes of data to see the effect of scalability.

For a volume of 1GB of our collected data, the performance we observed is listed in Table 1. The results captured in this table are observed in minutes.

Run	Without Optimizations	With Optimizations
1	62	16
2	59	17
3	50	14
Avg.	57	~ 16

Table 1: Performance observed with and without optimization - 1GB Data

For a volume of 3GB of our collected data, the performance we observed is listed in Table 2. The results captured in this table are observed in minutes.

Run	Without Optimizations	With Optimizations
1	157	39
2	140	35
3	151	37
Avg.	~ 150	~ 37

Table 2: Performance observed with and without optimization - 3GB Data

5 DISCUSSION

This section includes an overview of our journey with this project - what was our cluster setup, a summary of the challenges we have faced while implementing this project, what are the boundaries we still have not handled, our observations, and finally, the limitations of our system.

5.1 Cluster Setup

To work in this Project, a cluster of four VMs is configured with one master node and three slave nodes. On the cluster, Hadoop 3.2.1 is deployed, and we use Yarn as our resource management. We installed Spark 3.3.2 as it supports the Hadoop cluster we have set up for this project. As our cluster manager, we are using Yarn for both Hadoop and Spark.

5.2 Observations

The core idea of this implementation was to study the behavior of Hadoop, and Spark and the effect of optimization on Spark. The optimizations we have incorporated include repartitioning by a value of 150, caching, broadcast-join, and dropping unnecessary data columns and filtering.

As a study, we evaluated the performance and scalability of our implementation by comparing the execution time with and without optimizations for two different dataset setups: 1 GB and 3 GB of the collected dataset. From Table 1, we observed that for the 1GB dataset, the average execution time without optimizations was 57 minutes, whereas, with optimizations, it significantly decreased to 16 minutes, demonstrating a remarkable improvement in performance. Similarly, Table 2 showcases the results for the 3GB dataset, where the average execution time without optimizations was 150 minutes, and with optimizations, it was reduced to 37 minutes. The runtime increase from 1GB to 3GB dataset was roughly **2.6** times without optimizations and **2.4** times with optimizations, indicating that the code maintained its performance while handling a larger dataset. We also observed optimization has a huge effect on the performance of Spark execution time. This consistent performance enhancement, along with the effective utilization of Spark's parallel processing capabilities through the applied optimizations, serves as strong evidence that our code is scalable and capable of handling increasing amounts of data efficiently.

5.3 Challenges

During the development of our PySpark code, we encountered several challenges that required us to explore different solutions and make adjustments to our approach:

5.3.1 Hadoop and Spark synchronization. : Establishing the correct configuration for seamless synchronization between Hadoop and Spark was a significant challenge. We faced issues with the output raw protocol for the Hadoop application, resulting in non-encoded data at the beginning.

5.3.2 Package conflicts in Spark configuration. : We encountered conflicts between the Delta Table package and the Spark NLP package, which prevented us from running sentiment analysis and Delta Tables in the same file. As a workaround, we initially divided our code into two separate files. The first file handled data ingestion from CSV files and performed sentiment analysis, saving the output to a Parquet file. The second file read data from the Parquet file and carried out further data operations, including the use of Delta Tables and merging.

5.3.3 Delta Table limitations. : We discovered that Delta Tables were not well-suited for our use case, as our pipeline was primarily focused on performing operations on the structure of data frames, such as adding or selecting columns, rather than manipulating the data within the columns. The need to assign and save to a new Delta Table path every time we altered the data frame structure resulted in extended execution times. To meet the project's requirements, we devised a case where we used a merge of Delta Tables as shown in the code listing 11.

5.3.4 Sentiment analysis inaccuracies. : We opted for the Vivekn model from SparkNLP for sentiment analysis, primarily due to its relatively small pre-trained model size. However, after analyzing the sentiment of the reviews, we observed that some of them were misclassified as positive or negative, indicating potential limitations in the chosen model's accuracy.

5.4 Limitations

The scores we are generating is depending on the small dataset we have collected from Kaggle. With more data (e.g. more reviews) we could contribute to the generation of more accurate score.

As we observed that we are using a small dataset, we set up our nodes with medium size clusters both for the namenode and the datanodes. However, we observed with the increase in network overload in the server, this setup is causing some errors. Therefore, a better-configured setup (e.g. medium name node with larger data nodes, overall large cluster, etc.) would allow us to run code on bigger datasets and do more tests freely.

Although programming in Python is easier to understand and learn, the Python API on Spark works in a straightforward way. It performs the executions by allocating an executor for each partition. As a result, it hampers the best behavior Spark could give even after optimization. Therefore, any application program that can perform better and use the properties of Spark more effectively would give better results. As Spark is written on Scala, it would be the best-recommended language.

6 CONCLUSION

In this work, we have explored the use of data-intensive systems, Hadoop and Spark, with a use case of analyzing book reviews collected from the Amazon website. We performed various studies on different properties available in Spark and ways to optimize Spark. We observed, the optimization techniques applied significantly improved the system's performance and scalability. Although the system developed is not without any limitations, it was an effective case study for learning about the benefits of using data-intensive systems like - Hadoop and Spark. We would reflect on our studies in our further work.

In conclusion, we have learned about the benefits of different data-intensive systems, their behavior, and ways to optimize them, which was the goal of this study.

7 FURTHER WORK

There is considerable scope for further development and optimization of our book recommendation pipeline. One possible enhancement would be to personalize book recommendations based on a specific reader's preferences and reading history. For instance, we could suggest books from the same category as the previously read books, thus ensuring the recommendations are tailored to the reader's interests. Moreover, we could refine the final ranking system by displaying categories with a minimum of 10 books, providing users with a more diverse and comprehensive list of recommendations.

In addition to these improvements, further work could be undertaken to experiment with different cluster setups to optimize the code's performance more effectively. This experimentation may involve assessing various configurations and resource allocations that suit the specific needs of the recommendation pipeline.

Also future research could investigate alternative sentiment analysis models to enhance the accuracy of the recommendation system. By exploring and comparing different models, it may be possible to identify a more precise method of sentiment analysis.

Lastly, incorporating more data variants and larger datasets would enable us to evaluate the scalability and robustness of our recommendation system more thoroughly. By using a more extensive and diverse dataset, we can better understand the effectiveness of our pipeline in handling real-world data and make further refinements to improve its accuracy and performance.

REFERENCES

[1] Databricks. 2023. What is Delta Lake? https://docs.databricks.com/delta/index.html

[2] Brad Hedlund. 2011. Understanding Hadoop Clusters and the Network. https://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/

[3] Pier Paolo Ippolito. 2023. Apache Spark Optimization Techniques. https://towardsdatascience.com/apache-spark-optimization-techniques-fa7f20a9a2cf

[4] Walaa Medhat, Ahmed Hassan, and Hoda Korashy. 2014. Sentiment analysis algorithms and applications: A survey. *Ain Shams Engineering Journal* 5, 4 (2014), 1093–1113. https://doi.org/10.1016/j.asej.2014.04.011

[5] Vivek Narayanan. 2014. Web interface to sentiment analyzer. https://github.com/vivekn/sentiment-web

[6] Vivek Narayanan, Ishan Arora, and Arjun Bhatia. 2013. Fast and Accurate Sentiment Classification Using an Enhanced Naive Bayes Model. In *Intelligent Data Engineering and Automated Learning – IDEAL 2013*. Springer Berlin Heidelberg, 194–201. https://doi.org/10.1007/978-3-642-41278-3_24

[7] PySpark. 2023. PySpark Overview. https://spark.apache.org/docs/latest/api/python/#:~:text=PySpark%20is%20the%20Python%20API,for%20interactively%20analyzing%20your%20data.

[8] Spark. [n. d.]. Understanding Hadoop Clusters and the Network. https://spark.apache.org/docs/latest/cluster-overview.html

[9] Dhanya Thailappan. 2021. Understanding the Basics of Apache Spark RDD. https://www.analyticsvidhya.com/blog/2021/08/understanding-the-basics-of-apache-spark-rdd/

[10] TutorialsPoint. [n. d.]. Apache Spark - RDD. https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm
[11] Tomasz Wiktorski. 2019. *Data-intensive Systems: Principles and Fundamentals using Hadoop and Spark).* https://doi.org/10.1007/978-3-030-04603-3_2