DAT535 Project Report

Data Processing Pipeline in Spark

Rasmus Øglænd
University of Stavanger, Norway
ras.ogland@stud.uis.no

Filip B. Gotten
University of Stavanger, Norway
fb.gotten@stud.uis.no

ABSTRACT

This project leverages Apache Spark to build a scalable pipeline for analyzing large-scale Reddit comments on the COVID-19 pandemic. Using the Medallion Architecture, data is ingested (Bronze layer), cleaned (Silver layer), and analyzed (Gold layer) with Spark's distributed computing capabilities, ensuring efficient processing of massive datasets. Integrated with HDFS, the pipeline enables sentiment analysis, subreddit clustering, and trend tracking, demonstrating Spark's effectiveness in handling high-volume data for actionable insights.

1 INTRODUCTION

This project focuses on creating and exploring a robust data processing pipeline designed to analyze large-scale textual data from Reddit comments related to the COVID-19 pandemic [5]. The pipeline will generate structured datasets containing sentiment analyses and filtered insights, capturing public sentiment and discourse trends across the platform and within specific subreddit clusters during the pandemic. While these insights provide valuable perspectives on community discussions, the project's primary objective is to develop and optimize the pipeline itself. This involves leveraging Hadoop, HDFS, and Apache Spark through PySpark to efficiently manage and process large datasets [2], [3], [8].

The project is built around the Medallion Architecture, a layered design pattern for organizing and processing data in big data systems [6]. This architecture consists of three main layers: the Bronze layer stores raw ingested data, the Silver layer handles data cleaning and preprocessing, and the Gold layer contains refined and enriched data prepared for consumption. By adopting this approach, we ensure data integrity, scalability, and ease of use. This makes it especially effective for managing the complexities of the large-scale Reddit comment dataset.

The Github repository containing all our code can be found here: https://github.com/filigott/DAT535-2024.

1.1 Use Case Description

The COVID-19 pandemic has ignited extensive discussions on Reddit, where users share experiences, opinions, and information. This collection of user-generated content presents a unique opportunity to analyze public sentiment and discourse trends during a global crisis.

The use case for this project focuses on understanding public sentiment as a whole, examining how sentiment evolves over time and varies across different subreddit communities. By analyzing sentiment trends, identifying key topics of discussion, and uncovering similarities between subreddit clusters, we aim to provide a comprehensive view of how information spreads, opinions form, and engagement patterns emerge during significant events.

The insights derived from this analysis could inform stakeholders, such as pharmaceutical companies, on public attitudes, enabling them to make data-driven decisions in the event of future pandemics. For instance, understanding sentiment trends could help refine communication strategies, while identifying subreddit clusters with similar language and sentiment could highlight key communities for targeted outreach and engagement. Ultimately, this use case aims to facilitate efficient data processing for analyzing public discourse and to deliver actionable insights into societal responses during a health crisis.

2 DATASET

2.1 Dataset Description

To facilitate this analysis, we utilize the "Reddit COVID Dataset" available on Kaggle [5]. This dataset comprises Reddit comments that mention COVID-19, collected up to October 25, 2021. It includes the following key features:

- id (comment id)
- subreddit.id
- subreddit.name
- subreddit.nsfw
- created_utc
- permalink
- body
- sentiment (sentiment score of comment)
- score (upvotes)

The dataset is substantial, containing millions of comments, making it an ideal candidate for demonstrating the capabilities of Spark in handling large-scale data processing tasks. The dataset already contains the sentiment score, but this was removed and instead calculated in Part3 (link to section3) By leveraging this dataset, we can perform comprehensive sentiment analysis to uncover trends and patterns in public opinion during the pandemic utilizing Spark.

2.2 Data Scrambling

The original dataset was provided in a structured CSV format that had undergone initial cleaning during the scraping process from Reddit. For example, commas were likely removed from the body text to preserve CSV integrity. Despite these adjustments, the body text retained typical Reddit comment characteristics, such as special characters, inconsistent spacing, newlines, and formatting nuances.

Although the dataset could have been used in its existing form, simplifying the preprocessing tasks in the Silver Layer, we chose to further scramble the data to better align with the project's objectives. The goal was to simulate the challenges of processing unstructured, raw data and to demonstrate Spark's capabilities in handling such complexities. This added an extra layer of difficulty, allowing us to showcase the robustness and scalability of our pipeline.

The scrambling process transformed the dataset into a semi-structured format resembling raw, scraped data. Each comment was represented as a block of text with the following structure, where the number of lines varied depending on the comment length:

```
link:URL_TO_COMMENT
date:COMMENT_CREATED_UTC
body:TEXT_OF_COMMENT
```

This transformation produced a single large text file containing millions of comment blocks. The unstructured format introduced significant challenges, including parsing inconsistencies, handling multiline text, and managing irregular spacing. Reformatting the dataset into this scrambled structure allowed us to showcase Spark's capabilities in processing unstructured data within the Silver Layer.

3 PART 1: DATA INGESTION (BRONZE LAYER)

3.1 Cloud Setup

Our cloud setup was built on the OpenStack environment provided by the University of Stavanger (UiS), leveraging a template introduced during lab sessions to create a scalable and robust infrastructure tailored to our data processing needs. The setup consisted of four virtual machines (VMs), each configured with the "m1.large" flavor, providing 4 VCPUs, 8 GB of RAM, and 40 GB of storage per VM. This configuration was chosen to maximize the performance of each VM while adhering to the resource limitations imposed by our OpenStack account.

The account allowed a total of 16 VCPUs and 32 GB of RAM for all active VM instances, which dictated the constraints for our setup. We considered various configurations to optimize resource allocation further but faced a bottleneck due to the scaling properties of OpenStack flavors. For instance, while we had the option to double the number of VMs by reducing their allocated resources to the "m1.medium" flavor, this would not increase the total amount of RAM or VCPUs available. Instead, it would result in twice the number of VMs with half the individual computational power. We determined that maintaining four VMs with the "m1.large" flavor was more advantageous due to their higher specifications. This setup allowed for more efficient handling of memory-intensive tasks and simplified workload distribution.

Despite these optimizations, we identified that only 32 GB of the 50 GB storage allotment was utilized, leaving room for potential expansion in terms of storage. However, this surplus could not offset the bottleneck caused by the maximum allowable VCPUs. Given these constraints, our configuration struck a balance between maximizing computational resources and maintaining simplicity in managing the cluster.

The four VMs were used to distribute processing tasks across the cluster, while having the Spark executors run on three of the nodes to parallelize data ingestion, transformation, and analysis efficiently. This setup ensured that our pipeline could handle the large volumes of data associated with the project while maintaining scalability and performance. By carefully managing the available resources, we were able to implement a cloud-based solution that supported our project's requirements within the limitations of the OpenStack environment.

3.2 Hadoop & HDFS Overview

Hadoop Distributed File System (HDFS) formed the backbone of our distributed storage layer, enabling efficient data management across the OpenStack-based virtual machines. HDFS is well-known for its scalability and its ability to handle large datasets, making it the ideal storage solution for our data pipeline.

The HDFS cluster was configured in a distributed mode with one Namenode and three Datanodes, each hosted on identical "m1.large" VMs. This setup ensured data was distributed across multiple nodes, allowing for parallel access, fault tolerance, and scalability. Automatic replication of data blocks provided resilience against hardware failures, ensuring high availability and uninterrupted access to data during operations.

To optimize HDFS for our setup, we carefully configured replication factors and storage directories to align with the resource limitations of our cluster. This ensured a balance between redundancy and efficient use of storage resources. HDFS directories for Namenodes and Datanodes were managed to support scalability as the dataset evolved. By effectively integrating HDFS into our pipeline, we achieved seamless distributed storage that facilitated parallel processing and supported the high-performance requirements of subsequent computational stages.

3.3 Spark & PySpark Overview

Apache Spark served as a highly scalable and efficient framework for distributed data processing, offering the flexibility and performance required for large-scale analytics tasks. Its in-memory computation model minimized latency and data movement, making it ideal for both batch processing and iterative operations such as machine learning and clustering.

PySpark, Spark's Python API, provided a powerful interface for leveraging Spark's capabilities. For instance, instead of using traditional Hadoops MapReduce directly, Spark offers similar functionality in their own abstractions. The `sparkContext` API enabled low-level operations using RDDs, offering fine-grained control over distributed computations, while the DataFrame API facilitated SQL-like transformations and aggregations with ease. This combination balanced flexibility with simplicity, catering to a wide range of data processing needs.

Spark's integration with storage systems like HDFS streamlined workflows by enabling efficient data ingestion, transformation, and output storage. Its support for dynamic resource management and configuration tuning, such as optimizing memory allocation and shuffle partitions, ensured scalability and performance even with complex pipelines.

By combining Spark's distributed processing architecture with PySpark's accessible interface, we leveraged a robust framework capable of handling diverse data tasks while maintaining scalability, efficiency, and adaptability to evolving analytical needs.

3.4 Cluster Configs

Configuring Spark and the underlying Hadoop setup on the cloud cluster was a critical step to ensure efficient resource utilization and scalability. Initially, we applied standard configuration templates from lab exercises. These settings were adequate for small tasks but fell short during more complex workloads. For example, the default replication factor limited data distribution, and memory allocation caused frequent failures during intensive operations."

The initial Spark configuration in spark-defaults.conf for each node was as follows:

```
spark.master yarn
spark.driver.memory 512m
spark.yarn.am.memory 512m
spark.executor.memory 512m
```

This configuration resulted in several challenges. First, the cluster's worker nodes were not fully utilized. Specifically, only two of the three datanodes were active in processing tasks, which we traced to the dfs.replication setting in hdfs-site.xml, configured with a replication factor of 2. This limited data distribution and prevented the third node from participating effectively. Second, memory-intensive operations, such as joins or shuffle-heavy transformations, frequently failed due to the insufficient 512 MB memory allocation for executors, leading to out-of-memory errors during processing. Third, the overall resource utilization was suboptimal, as the minimal memory allocation and replication settings left significant computational power untapped, resulting in longer execution times and inefficiencies.

These initial configurations, while appropriate for basic lab scenarios, were not designed to handle the demands of large-scale data processing. As our use cases evolved to include larger datasets and more complex transformations, we recognized the need to optimize these settings. A deeper investigation into resource allocation and cluster utilization enabled us to unlock the full potential of Spark and improve overall performance significantly which are discussed in the later sections.

3.5 Data Ingestion

The data ingestion process began with identifying and sourcing our Reddit COVID-19 dataset, which was retrieved from Kaggle [[5]]. The dataset was initially downloaded manually to a local machine to ensure the integrity and completeness of the raw data before transferring it to the cloud environment. After careful consideration, we decided to focus exclusively on the comments within the original dataset, which also included posts. The size of the posts was significantly smaller than the comments, and incorporating them would have introduced unnecessary complexity due to the differing schema.

To manage the dataset's size and complexity during testing, we implemented functionality to create smaller subsets. These subsets enabled us to validate the pipeline and troubleshoot issues efficiently without needing to process the full dataset, saving valuable time and computational resources during development.

Once the dataset was prepared, we used `SCP` (Secure Copy Protocol) to transfer the files from the local machine to the Namenode within our OpenStack cluster. The Namenode acted as the central entry point for all data ingested into the cloud-based HDFS (Hadoop Distributed File System). After copying the dataset to the Namenode, we added the files to HDFS, distributing the data across the Datanodes. This setup ensured that the data was accessible to all nodes in the Spark cluster, enabling parallel processing and efficient resource utilization.

By storing the raw dataset in HDFS, we established the Bronze layer of our Medallion Architecture. This layer served as a centralized repository for all incoming data, preserving the original structure and integrity of the dataset. HDFS provided high-throughput data storage and retrieval, supporting the scalability and fault tolerance required for handling large datasets.

This ingestion process not only ensured the secure and efficient transfer of data but also facilitated seamless integration with the Spark cluster. With the data accessible across all nodes, we laid the foundation for the subsequent cleaning and transformation stages, enabling the pipeline to process and analyze the data effectively.

4 PART 2: DATA CLEANING (SILVER LAYER)

4.1 Basic MapReduce Routines in Spark

MapReduce is a programming model widely used for distributed data processing. It involves two primary steps: map, which applies a transformation or function to each data element independently and reduce, which aggregates results across data partitions to produce a final output. [1]

While MapReduce is ideal for tasks involving aggregation or summarization (e.g., summing numbers or counting words), our use case is in the silver layer is different. Instead of aggregating data, we aim to structure and clean the input data by processing and organizing it without reducing or combining results. For this reason, mapPartitions was chosen as the appropriate Spark routine. Unlike map, which processes one record at a time, mapPartitions operates on an entire partition at once. [7] This allows for efficient parsing and restructuring of multiline comment blocks within the partition, making it suitable for handling the scrambled dataset where each comment spans multiple lines.

4.2 Cleaning and Preprocessing Implementation

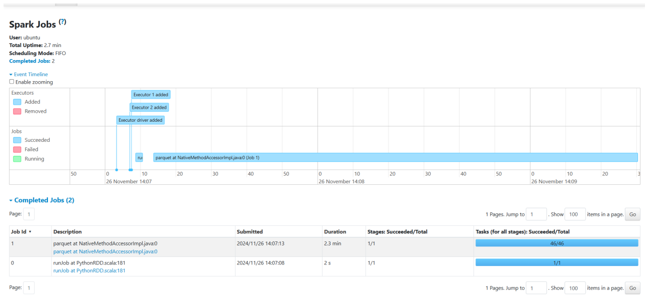
The dataset, stored as a large text file on HDFS, was processed as follows. A Spark context was initialized, and the file was loaded into an RDD. The processing logic was encapsulated in a custom "readerFunction", which transformed the scrambled multiline comments into structured records. The key tasks handled by "readerFunction" included identifying the start of a comment block, extracting relevant fields (e.g., link, date, and body), cleaning the comment body by removing unnecessary characters, and validating the completeness of the comment block. The processing logic in the "readerFunction" also ensured that incomplete or invalid comments were discarded.

The resulting RDD was converted into a Spark DataFrame with a well-defined schema and stored in Parquet format for easier access in the gold layer. Below is the pseudocode representation:

```
# Connect to existing spark cluster
sc = sparkContext.getOrCreate()

# Read the text file from hdfs
rdd = sc.textFile("path-to-file-on-hdfs")

# Apply the custom reader function to process each partition
structured_rdd = \
rdd.mapPartitions \
(lambda partition: readerFunction(partition))

# Convert the structured RDD into a DataFrame
df = structured_rdd.toDF \
(["link", "created_utc", \
"sub_reddit", "post_id", \
"comment_id", "body"])

# Save the DataFrame as a Parquet file
df.write.parquet("output-path")
```

4.3 Adjusting Spark Config

4.3.1 Default Configs: In the initial stages of the processing part of our pipeline, we utilized the default configuration parameters provided in the labs of our course presented in 3. These settings included limited memory allocations, and no advanced optimizations for parallelism or resource utilization. As seen in Figure 1, the execution of the proccessing pipline added only two executors, which processed the job with a total running (total uptime from spark logs) time of just under 3 minutes.


Spark Jobs (?)
User: ubuntu
Total Uptime: 2.7 min
Scheduling Mode: FIFO
Completed Jobs: 2

Completed Jobs (2)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/11/26 14:07:13	2.3 min	1/1	46/46
0	runJob at PythonRDD.scala:181 runJob at PythonRDD.scala:181	2024/11/26 14:07:08	2 s	1/1	1/1

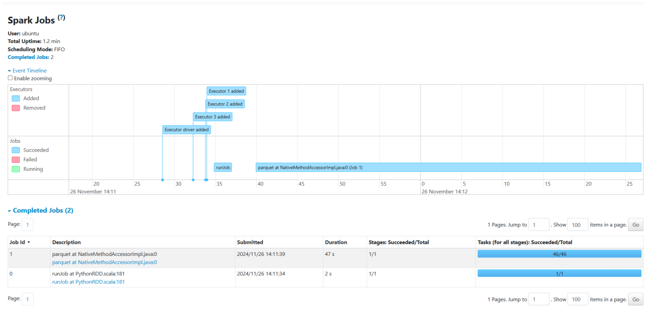
Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Figure 1: Spark history server processing with default configs

This indicates that the Spark cluster's computational power was underutilized, as a larger number of executors or adjusted configurations could have distributed the workload more effectively. This highlights the need for configuration optimization to achieve a better balance between available resources and processing speed.

4.3.2 Optimized Configurations. To address the inefficiencies observed with the default configurations, we initially optimized the Spark settings by allocating additional resources to the executors and the driver. The updated configurations allocated 5GB of memory to each executor, leaving 3GB available for other processes on an 8GB system. Each executor was assigned 3 out of 4 available cores, reserving one core for non-Spark tasks. Additionally, exactly three executor instances were deployed, ensuring one instance per data node for balanced resource utilization.

To further enhance cluster performance and fully utilize all three datanodes, we adjusted the Hadoop dfs.replication parameter in hdfs-site.xml, increasing the replication factor from 2 to 3. This ensured that data blocks were evenly distributed across all three nodes, maximizing cluster utilization and enabling better parallel processing during Spark jobs. These changes collectively allowed us to harness the full potential of the cluster, significantly improving processing efficiency and resource allocation.


Spark Jobs (?)
User: ubuntu
Total Uptime: 1.2 min
Scheduling Mode: FIFO
Completed Jobs: 2

Completed Jobs (2)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/11/26 14:11:39	47 s	1/1	46/46
0	runJob at PythonRDD.scala:181 runJob at PythonRDD.scala:181	2024/11/26 14:11:34	2 s	1/1	1/1

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Figure 2: Spark history server processing with optimized configs

With these adjustments, as seen in Figure 2, we achieved a significant reduction in processing time, bringing it down from approximately 3 minutes to just 1 minute. Importantly, all three data nodes in our network were actively utilized during this run, demonstrating a much-improved utilization of the cluster's computational capacity. This improvement validates the importance of tailoring Spark configurations to align with the cluster's resources and workload requirements, but it also raises the question of whether further optimization is possible.

4.3.3 Exploring Further Optimizations: In our efforts to further optimize the Spark cluster's performance for processing the raw data, we hypothesized that increasing the number of executors would lead to faster execution times by distributing the workload more evenly across the cluster. To test this, we enabled YARN dynamic resource allocation and configured the initial number of executor instances to 3 (the same as our previous optimized configuration). Additionally, we allowed the cluster to scale up to 6 executors, enabling each node to potentially run more than one executor given the number of cores available.

However, contrary to our expectations, this adjustment did not result in a noticeable improvement in execution time. Upon investigation, we realized that this was likely due to the memory constraints. Running multiple executors on a single data node requires reducing the memory allocation per executor to accommodate additional instances. To test this hypothesis, we further reduced the memory allocated per executor and allowed the cluster to scale up to 9 executors.

Despite these changes, the total processing time remained consistent with the prior optimized configuration. This suggests that the overhead introduced by running more executors on each node—such as increased inter-executor communication and reduced memory per task—offset any potential gains from increased parallelism. These findings highlight the complex interplay between memory allocation, executor count, and cluster utilization in Spark, and suggest that beyond a certain point, adding more executors does not necessarily translate into improved performance when we are limited to ceratain computers specs per node.

4.4 Considerations and Results

Using mapPartitions for processing introduced a notable challenge. If a comment block spanned multiple partitions, the block would become corrupted, making it unusable. To avoid such corrupted comment blocks, we decided to discard these incomplete entries entirely. While this approach risks losing up to one comment per partition, it was deemed acceptable given the dataset's large size, consisting of millions of comments.

If preserving every comment was essential, alternative strategies could be implemented to avoid data loss. One approach would involve processing the entire dataset on a single node. This method ensures that no comment blocks are split across partitions, thereby maintaining data integrity. However, it significantly increases execution time since it eliminates the parallel processing advantages of Spark, making it less suitable for large-scale or time-sensitive tasks.

Another option would be to pre-split the raw dataset into multiple smaller files, each containing only complete comment blocks. By organizing the data in this way, each node processes a self-contained file as a partition, avoiding the risk of splitting comment blocks. This strategy preserves the distributed processing efficiency of Spark while ensuring that no data is lost, making it a more practical solution for scenarios requiring both data integrity and high performance.

For scenarios involving periodic batch processing, such as a one-time job performed monthly, single-node processing might be acceptable despite the longer runtime. However, for real-time or high-frequency data processing, leveraging Spark's distributed architecture is essential to minimize latency. In such cases, the small risk of losing a few comments per partition is a reasonable trade-off for achieving high performance and scalability.

Ultimately, given the dataset's scale and the minimal impact of losing a few comments, we concluded that the trade-off was acceptable. The resulting processed dataset was stored as parquet for easy use in the gold layer.

5 PART 3: DATA SERVING (GOLD LAYER)

5.1 Business Use Cases and Objectives

The primary objective of the Gold Layer is to leverage the Reddit comments dataset to derive actionable insights related to public sentiment and subreddit dynamics during the COVID-19 pandemic. This involves analyzing sentiment trends over time, identifying patterns in subreddit discussions, and uncovering similarities between different subreddit communities [5].

These insights aim to provide a deeper understanding of public discourse, enabling targeted messaging, improved engagement strategies, and preparedness for addressing future crises. Key outcomes include identifying shifts in sentiment due to significant events, exploring thematic relationships between subreddits, and recognizing emerging topics of interest [6].

5.2 Identified Algorithms and Methodologies

To achieve these objectives, the Gold Layer employs a combination of algorithms and methodologies. Sentiment analysis techniques are used to assign sentiment scores to each comment, forming the foundation for tracking trends and analyzing shifts in opinion over time [4]. Clustering algorithms are applied to identify similarities between subreddits, uncovering thematic or linguistic relationships and enabling the grouping of communities with overlapping interests or concerns [9].

These methodologies ensure consistency and reliability in the generated insights, with sentiment analysis providing granular perspectives on public opinion and clustering revealing broader connections between subreddit dynamics. Together, they enable a comprehensive approach to understanding community discourse and its evolution during the pandemic.

5.2.1 Sentiment Calculation. To effectively analyze public sentiment in Reddit comments, we utilized VADER (*Valence Aware Dictionary and sEntiment Reasoner*), a sentiment analysis tool well-suited for social media text [4]. VADER's ability to handle informal language, slang, emojis, and abbreviations made it an ideal choice for capturing the nuanced sentiment in user-generated content. The compound sentiment score, which ranges from -1 (most negative) to +1 (most positive), was calculated for each comment to measure overall sentiment.

Before applying VADER, the text underwent rigorous preprocessing to ensure consistency and suitability for analysis. Using PySpark native functions, the text was converted to lowercase, and non-essential elements such as URLs, mentions, hashtags, and special characters were removed. This preprocessing step standardized the dataset, improving the reliability of sentiment analysis [11].

To efficiently scale sentiment calculation across a large dataset, we initially attempted to use PySpark's User-Defined Functions (UDFs). However, we found that UDFs introduced significant performance overhead due to serialization and deserialization costs when data was passed between the JVM and Python processes. As a result, we shifted to PySpark's RDD (Resilient Distributed Dataset) API, which proved to be faster and more efficient for our use case [10].

VADER's sentiment analyzer was broadcast across the cluster, allowing each worker node to process comments locally with minimal resource overhead. A custom function calculated the compound sentiment score for each comment and returned the results as an RDD. This approach avoided the performance penalties associated with PySpark UDFs and ensured a highly efficient distributed computation.

The sentiment scores were subsequently integrated into a structured DataFrame containing only the essential fields, specifically the id of the comments and the calculated sentiment score. The processed results were stored in Parquet format to minimize storage while retaining accessibility for downstream analysis.

5.2.2 Sentiment Trend Analysis. To analyze the evolution of sentiment over time, we utilized PySpark's DataFrame API to efficiently process and compute daily sentiment trends from the timestamped Reddit comments dataset [11]. This implementation leveraged PySpark's SQL-like capabilities for scalability and optimization, enabling us to handle large datasets while deriving actionable insights into how sentiment fluctuated in response to significant events during the COVID-19 pandemic.

The process began with integrating the calculated sentiment scores into the original dataset by performing an `inner join` on the `comment_id`. This merged DataFrame included key fields such as `body_clean`, `created_utc`, `sub_reddit`, and the computed `sentiment`. The timestamps in the `created_utc` column were then converted into a human-readable `date` format using PySpark's `from_unixtime` function, forming the basis for temporal analysis.

To compute overall sentiment trends, the data was grouped by `date` using the `groupBy` method, and the daily average sentiment was calculated with the `avg` function. This provided a high-level view of how sentiment shifted over time across the entire dataset. For subreddit-specific trends, we grouped the data by both `date` and `sub_reddit` and calculated the daily average sentiment for each subreddit. These operations leveraged PySpark's distributed execution, ensuring scalability even with large-scale data [8].

A comparative analysis was performed by joining the subreddit-specific trends with the overall daily averages on the `date` column using PySpark's `join` operation. To quantify deviations in sentiment for each subreddit relative to the overall trend, we added a derived column, `sentiment_diff`, which calculated the difference between the subreddit's average daily sentiment and the overall average sentiment. This computation was performed directly on the DataFrame, avoiding iterative processing and ensuring efficient utilization of PySpark's distributed architecture.

For storage and downstream analysis, we optimized the final dataset by selecting only the essential columns: `date`, `sub_reddit`, `avg_daily_sentiment_subreddit`, and `sentiment_diff`. The results were written to Parquet format, ensuring minimal storage requirements and high performance for subsequent queries.

The use of PySpark's DataFrame API and SQL-like transformations allowed us to build a robust, scalable pipeline for sentiment trend analysis. Caching intermediate results further optimized performance by reducing recomputation overhead during iterative development and execution. This implementation enabled rapid extraction of insights, supporting both broad temporal sentiment analyses and focused investigations into subreddit-specific patterns.

5.2.3 Subreddit Similarity and Clustering. To uncover thematic relationships and similarities among subreddit communities, we employed a combination of text-based similarity analysis and clustering techniques. This allowed us to group subreddits with similar discussion patterns and sentiments, providing insights into shared interests and overlapping topics across communities. While effective, this process significantly increased the runtime of the Gold Layer pipeline to approximately 1.5 hours from the usual 30 minutes, though we opted to retain it in the pipeline due to its possible value.

The analysis began with preprocessing the body text of comments using a text-processing pipeline. We tokenized the `body_clean` field to split it into individual words and removed stopwords such as "the" and "and," which are common but provide little thematic significance. These steps ensured the focus remained on meaningful terms relevant to subreddit-specific themes.

Next, we calculated TF-IDF (*Term Frequency-Inverse Document Frequency*) vectors for the preprocessed text, which transformed the filtered words into numerical representations reflecting the importance of terms within each subreddit [9]. This was implemented using PySpark MLlib's `HashingTF` and `IDF` components within a pipeline, ensuring scalability and efficient processing of the large dataset. The resulting TF-IDF vectors represented the textual characteristics of each comment, which were subsequently aggregated by subreddit to compute average vectors, forming subreddit-level representations.

Using these subreddit-level TF-IDF vectors, we applied `KMeans` clustering to group subreddits into five distinct clusters, although this parameter (k) can be tuned for different use cases [9]. The clustering results assigned each subreddit to a cluster, enabling the identification of communities with similar discussion topics or shared thematic focus.

To enrich these clusters with sentiment insights, we joined the clustering results with the sentiment data and calculated the average sentiment for each cluster. This step highlighted how sentiment trends aligned with thematic similarities within clusters, providing a nuanced understanding of cross-community dynamics.

Finally, we stored the clustering results and sentiment analyses in Parquet format to minimize storage while maintaining accessibility for downstream analysis. The clustering results included subreddit assignments to clusters, while the sentiment analysis provided insights into average sentiment and subreddit counts per cluster.

Despite its significant impact on runtime, this clustering methodology added substantial analytical depth to our pipeline. By combining TF-IDF vectorization, KMeans clustering, and sentiment analysis, we gained valuable insights into thematic and sentiment-based relationships among subreddits. These findings support the discovery of shared themes, emerging topics, and cross-community sentiment trends, justifying the additional computational overhead required for this analysis.

5.3 Challenges Faced and Solutions

Implementing the Gold Layer posed several technical and logistical challenges, particularly concerning data volume, performance, scalability, and resource constraints. Below, we outline the key challenges and the solutions we implemented to overcome them.

5.3.1 Limited Virtual Machine (VM) & Distributed File System (DFS) Space. A significant challenge was the limited storage capacity of the VMs, which directly restricted the DFS storage available on the nodes. The original dataset, over 14GB in size, consumed much of the available space, further strained by the need to store multiple versions of the data—including scrambled, cleaned, and processed results. This demand was particularly high during the Gold Layer's execution, where intermediate and final results required additional storage.

To mitigate this, we reduced the dataset size by retaining only every other row in the scrambled dataset. This halved the size of both the scrambled and cleaned datasets, significantly reducing storage demands. The reduced dataset also minimized the size of the output data from the Gold Layer, easing space constraints while preserving analytical integrity.

Another storage bottleneck arose from the accumulation of temporary files in the Hadoop user cache, often causing failures when saving large Parquet files during the Gold Layer's execution. We resolved this by regularly clearing the Hadoop user cache, freeing up space and ensuring adequate capacity for writing results.

By combining dataset reduction with proactive HDFS management, we successfully addressed storage limitations, enabling reliable and efficient pipeline execution within the available infrastructure constraints.

5.3.2 *PySpark Configuration Challenges.*

Optimizing PySpark for the Gold Layer posed significant challenges due to the varying computational demands of its tasks and the extended runtime. The length and complexity of the pipeline made iterative optimization difficult, requiring configurations that balanced performance with resource constraints. We utilized YARN's dynamic resource allocation to manage the number of executors based on workload demands, ensuring efficient use of cluster resources during computational peaks [8].

Attempts to fine-tune PySpark parameters, such as increasing `spark.executor.memory` to handle larger partitions and adjusting `spark.sql.shuffle.partitions` to minimize shuffle overhead, yielded only marginal performance gains. These findings reinforced the reliance on YARN's adaptive resource management to dynamically allocate resources based on workload demands, making manual configurations less impactful for our use case. This reliance highlighted the value of PySpark's integration with distributed resource managers, which facilitated scalability and efficiency even for resource-intensive tasks [11].

By combining dynamic resource allocation with selective parameter tuning, we established a robust and scalable PySpark configuration capable of supporting the computational intensity of the Gold Layer. This configuration not only ensured reliable pipeline execution but also provided a foundation for handling future data volume increases.

5.3.3 *Sentiment Calculation Challenges.*

Sentiment calculation was particularly challenging due to the need to process millions of comments using VADER, a sentiment analysis tool optimized for social media text [4]. Initial implementations relied on PySpark UDFs, which proved inefficient, as UDFs execute native Python code instead of leveraging Spark's optimized JVM-based execution framework [11]. This inefficiency resulted in runtimes of approximately 8 minutes for processing.

To address this, we transitioned to RDD-based processing, reducing runtime to 7 minutes by distributing computations more effectively across the cluster. A significant optimization involved broadcasting the `SentimentIntensityAnalyzer` across worker nodes. By preloading the VADER analyzer in memory on each node, we eliminated the need for repeated initialization, reducing memory consumption and improving execution consistency [8].

Intermediate result caching further enhanced performance by avoiding redundant computations during iterative development and debugging. These solutions underscored the importance of aligning Spark operations with workload-specific needs to maximize efficiency. While further gains could be achieved by integrating Spark-native libraries for sentiment analysis, the current configuration effectively balanced speed and resource usage.

5.3.4 *Sentiment Trends Analysis Challenges.*

Analyzing sentiment trends required managing large-scale joins, aggregations, and temporal computations efficiently. Joining sentiment scores with the main dataset introduced memory-intensive operations, particularly when calculating daily averages for platform-wide and subreddit-specific trends. To mitigate this, we partitioned the dataset strategically and cached intermediate results, reducing recomputation and ensuring scalability [11].

Group-by operations, such as computing `avg_daily_sentiment_all` and `avg_daily_sentiment_subreddit`, required careful tuning of `spark.sql.shuffle.partitions` to minimize shuffle overhead. This optimization was crucial for maintaining performance during large-scale aggregations [10].

Calculating sentiment differentials (`sentiment_diff`) added complexity by requiring multi-level joins and comparisons between subreddit-specific trends and overall averages. These operations were optimized to execute within memory constraints while maintaining accuracy. To reduce storage demands without compromising accessibility, we selected essential columns for output and stored results in Parquet format with compression, ensuring efficient storage utilization [3].

These optimizations enabled effective sentiment trend analysis, providing actionable insights into temporal sentiment patterns and subreddit-specific dynamics, supporting deeper understanding of community behaviors.

5.3.5 *Subreddit Clustering Challenges.*

Clustering subreddits using KMeans introduced significant computational challenges due to the high-dimensional TF-IDF vectors and the iterative nature of the algorithm. Preprocessing steps, including tokenization, stopword removal, and TF-IDF vectorization, were computationally expensive, particularly for large-scale subreddit data. These transformations were implemented using PySpark MLlib, leveraging its scalability for large datasets [9].

The clustering process, configured with `k=5`, required substantial memory for storing and processing high-dimensional vectors and centroids, significantly contributing to the overall runtime of approximately 1.5 hours. To mitigate this, we pre-aggregated TF-IDF vectors by subreddit, reducing the dataset size passed to the clustering algorithm while preserving thematic granularity.

Evaluating the quality of clustering using the Silhouette score added additional computational overhead but was critical for validating results. The final step of saving clustering outputs and sentiment analyses required significant storage resources. By selecting actionable fields for output and storing results in Parquet format with compression, we minimized storage demands while maintaining accessibility [8].

Despite these challenges, clustering revealed valuable thematic relationships between subreddits, highlighting cross-community dynamics and emerging discussion patterns. These insights justified the computational cost and emphasized the importance of incorporating advanced clustering techniques into the Gold Layer to enhance analytical depth and cross-community understanding.

5.4 Results and Insights

The Gold Layer of our medallion architecture pipeline produced critical outputs, powering a real-time dashboard for sentiment trends, subreddit clustering, and thematic analysis. These outputs offered both a platform-wide view of sentiment dynamics and detailed insights into community-specific behaviors, providing actionable intelligence for decision-making.

Aggregate sentiment analysis identified significant shifts in public discourse, often aligned with major events such as policy changes or global developments. Subreddit-level trends highlighted diverse reactions, with some communities showing increased positivity and others heightened negativity, reflecting contrasting perspectives within the user base.

Temporal sentiment patterns revealed key moments of heightened activity or controversy, often coinciding with spikes in engagement. These patterns offered predictive insights into user behavior, enabling targeted outreach and the identification of potential flashpoints.

Subreddit clustering, leveraging TF-IDF and KMeans, uncovered thematic connections between seemingly unrelated communities. For instance, clusters linked subreddits on mental health and personal finance, suggesting shared concerns during the pandemic. These insights supported cross-community engagement and strategies tailored to overlapping interests.

Despite challenges such as computational intensity and storage demands, the pipeline was optimized for scalability and efficiency. Techniques like caching, dynamic resource allocation, and data reduction allowed for processing large datasets without compromising performance.

The Gold Layer demonstrated the power of distributed computing and advanced analytics to generate meaningful insights, laying the groundwork for future extensions such as real-time monitoring, predictive analytics, and deeper exploration of user behavior patterns.

6 AI USAGE

We leveraged AI tools, such as ChatGPT and GitHub Copilot, to assist with various aspects of our project, from generating ideas to refining our code and text. However, it's important to emphasize that the final code and content were carefully crafted and thoroughly reviewed by us, ensuring the highest standards of quality and optimization.

ChatGPT played a significant role in transforming our original content into more polished and professional text. While the core ideas and descriptions were all authored by us, ChatGPT helped enhance the language for improved readability and clarity, making our writing more formal and cohesive.

Additionally, ChatGPT was a valuable resource for clarifying complex technical concepts, such as Apache Spark and HDFS. It served as a supplemental learning tool, helping us gain a deeper understanding of specific topics and answering questions that emerged during the implementation process.

Both ChatGPT and GitHub Copilot were used as aids in writing and debugging code. They offered suggestions for syntax, code snippets, and best practices, which helped streamline the development process. That said, the logic, structure, and functionality of the code were meticulously designed by us. T hese AI tools were integral in ensuring syntactical accuracy and optimizing our implementation, but the code's overall direction and decisions were the result of extensive trial, error, and hours of testing.

In conclusion, while AI was a helpful assistant throughout the process, we dedicated considerable time and effort to ensure that all aspects of the project—both the code and the written content—were the result of our own careful consideration and hands-on work.

7 FURTHER WORK

Future work could transform the current implementation into a fully operational pipeline for real-world applications. This includes automating data ingestion from sources like Reddit's API using tools such as Pushshift or PRAW, enabling periodic or trigger-based data fetching. Processed data and insights could be seamlessly delivered to downstream systems, such as data warehouses or visualization platforms.

Scalability improvements could focus on expanding the cloud cluster with additional datanodes to process larger datasets without compromising performance. Dynamic scaling on platforms like AWS EMR, GCP Dataproc, or Azure Synapse Analytics could balance cost and compute demands. Integrating PySpark with cloud-native tools like S3 or Google Cloud Storage and serverless execution could further streamline operations.

In the Silver Layer, a key challenge is data loss due to incomplete comments being discarded during partitioning. Future work could develop a more robust processing framework to retain all comments and support real-time or near-real-time processing, moving beyond the current batch-oriented design.

To improve the Gold Layer's execution time, advanced caching mechanisms, checkpointing, and optimized serialization formats could reduce recomputation overheads. Incremental updates for clustering and sentiment analysis would enable processing of new data without reprocessing the entire dataset, lowering latency. Alternative clustering algorithms, such as DBSCAN or graph-based methods, could provide better insights into subreddit diversity, while transformer-based NLP models like BERT or GPT could enhance sentiment analysis and topic modeling, surpassing current TF-IDF and VADER capabilities.

Further optimization of PySpark could include advanced data partitioning to reduce shuffling, leveraging PySpark's pandas API as a UDF alternative, and deploying containerized applications with Kubernetes for portability and fault tolerance. These improvements would enhance scalability and performance under high workloads.

By addressing these areas, the pipeline could evolve into a robust, efficient system capable of delivering real-time insights while maintaining scalability and adaptability for diverse use cases.

8 CONCLUSION

This project successfully implemented a scalable data processing pipeline using the Medallion Architecture to analyze Reddit comments related to COVID-19. By leveraging Apache Spark and HDFS, we efficiently ingested, cleaned, and processed large-scale data, uncovering insights on public sentiment and community dynamics.

The Bronze Layer provided robust raw data storage, the Silver Layer tackled semi-structured data cleaning, and the Gold Layer delivered actionable analyses such as sentiment trends and subreddit clustering. Despite challenges like storage limitations and computational overhead, our solutions demonstrated the pipeline's scalability and adaptability.

Future work could focus on real-time data integration, advanced clustering techniques, and enhanced NLP methods for deeper insights. This pipeline serves as a strong foundation for large-scale data analysis, with potential for broader applications in crisis response and beyond.

REFERENCES

[1] dikshantmalidev. 2023. Map Reduce and its Phases with numerical example. https://www.geeksforgeeks.org/mapreduce-understanding-with-real-life-example/
[2] Apache Hadoop. 2024. Apache Hadoop 3.4.1. https://hadoop.apache.org/docs/current/
[3] Apache Hadoop. 2024. HDFS Architecture. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
[4] Hutto and Gilbert. 2014. VADER (Valence Aware Dictionary and sEntiment Reasoner). https://github.com/cjhutto/vaderSentiment
[5] Lexyr. 2022. The Reddit COVID dataset. https://www.kaggle.com/datasets/pavellexyr/the-reddit-covid-dataset/data?select=the-reddit-covid-dataset-comments.csv
[6] Microsoft. 2024. What is the medallion lakehouse architecture? https://learn.microsoft.com/en-us/azure/databricks/lakehouse/medallion
[7] Naveen Nelamali. 2024. PySpark mapPartitions() Examples. https://sparkbyexamples.com/pyspark/pyspark-mappartitions/
[8] Apache Spark. 2024. Apache Spark - A Unified engine for large-scale data analytics. https://spark.apache.org/docs/latest/
[9] Apache Spark. 2024. MLlib: Scalable Machine Learning on Spark. https://spark.apache.org/mllib/
[10] Apache Spark. 2024. PySpark Overview. https://spark.apache.org/docs/latest/api/python/index.html
[11] Apache Spark. 2024. PySpark SQL and DataFrame Guide. https://spark.apache.org/docs/latest/sql-programming-guide.html