Big Data ETL for Amazon Reviews using Spark

DAT535 - Data-Intensive Systems and Algorithms - Group 7

Bawfeh Kingsley Kometa
University of Stavanger, Norway
bk.kometa@stud.uis.no

Nourin Mohammad Haider Ali Biswas
University of Stavanger, Norway
nm.biswas@stud.uis.no

ABSTRACT

This project focuses on applying Spark methodologies to design compact and efficient Extract-Transform-Load (ETL) algorithms capable of handling large volumes of data. The case study examines customer review data from Amazon, covering the period from 2012 to 2023, amounting to over 75 GB of information. We aggregate customer review counts for each product across 29 product categories to derive monthly review counts. The analysis is conducted on a Hadoop cluster comprising four virtual machines hosted on the OpenStack cloud platform. The cluster consists of one master node and three worker nodes, each with specifications of 40 GB disk storage, 8 GB RAM, and 4 CPU cores. By deploying our algorithms on this cluster, we establish linear scalability in relation to data size. Furthermore, we demonstrate that pipelined transformations enhance the Spark engine's ability to optimize the ETL process, resulting in over a 30% computational gain compared to disintegrated computations.

KEYWORDS

Amazon Reviews, Hadoop, Spark, MapReduce, DataFrames

1 INTRODUCTION

This project aims to process large amounts of data using Apache Spark. The data consists of customer reviews on different categories of products from Amazon stores dating from 2012 to 2023. With minimal resources, we shall find ways to efficiently distribute, extract, and process useful information out of large volumes of data. This is achievable by exploiting the Hadoop file system (HDFS), a framework for efficient storage and access of large volumes of files on a cluster of commodity machines. Spark complementarily provides powerful high-level abstractions for parallel computing on distributed systems, such as resilient distributed databases (RDD), SQL, and Dataframes. RDD datasets are designed to support fast computations of course-grained (batch) operations in memory. Such operations typically consist of compositions of MapReduce functions. Details on the concepts of the RDD, MapReduce, and Dataframes can be found in their original articles [1, 2, 11]. See also [4, 9].

Our cluster setup consists of virtual machine (VM) instances on the OpenStack cloud network. To achieve the highest possible level of parallelism in our computations, three main resources must be optimized, namely, the storage capacity, the total memory (RAM), and the number of CPU cores available on each VM (worker node).

Supervised by Tomasz Wiktorski and Jayachander Surbiryala .

Data-Intensive Systems and Algorithms (DAT535), IDE, UiS
2024.

We shall experiment with two use cases and measure the performance of our algorithms in terms of computational time. In the first use case, we aggregate the number of monthly reviews for each product in a given year. In addition, for each product, we compute the total customer reviews and a linear regression coefficient of the reviews. The purpose is to reveal which product received the most attention from customers. To enhance performance, we fine-tune relevant spark configurations. In the second use case, we extend the previous steps for any two consecutive years and compute the seasonality trend for each product using a simple correlation coefficient.

In the next section, we shall describe our datasets, the cluster configurations, and the Hadoop file system and explain how to configure the spark engines. In section 3 we discuss the basic algorithm involved in the entire ETL process for Amazon reviews. The performance of the ETL algorithms used are presented via two use cases in Sections 3 and 4 respectively. We conclude in section 5 with the discussions and analysis of our results.

2 DATA AND CONFIGURATION

2.1 Hadoop architecture

To begin our project, we launched four VMs, henceforth called nodes, on the OpenStack cloud platform, guided by course material [8]. Each node includes 40GB of disk storage, 8GB of RAM, and 4 CPU cores. Remote access to the nodes is facilitated via SSH. After establishing the network interface, Hadoop is configured from scratch, designating one node as the master and the other three as worker nodes. The necessary libraries and environments for Java, Hadoop, Python, and PySpark are installed. A remote client-server (node) interacts with the master node through Hadoop commands. This setup closely resembles the architecture illustrated in Fig. 1 (sourced from the web[1]).

This architecture provides a comprehensive framework for managing data storage and processing in a distributed system. The client is only a temporal location for data (hence, it is called local). The master node orchestrates operations, hosting both the name node and the YARN resource manager, while data primarily resides in the data nodes inside the workers (also known as chunk servers). Communication between the data nodes and the client node is via the YARN resource managers. The client sends read/write requests to the YARN resource manager via Hadoop commands. The YARN resource manager plays a critical role in handling these client requests, ensuring resource availability, and directing data operations. It also oversees data replication across data nodes, a crucial mechanism for data recovery and integrity by creating multiple replicas

[1] https://i.sstatic.net/PSyy6.png

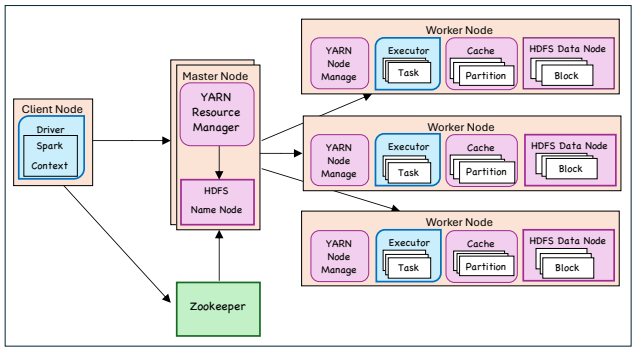

Figure 1: A YARN-based Hadoop file system [sourced from the web]

of data. All relevant metadata is maintained in the name node, ensuring efficient management and access. The role of each YARN node manager is to receive requests for task execution from the master node, activate the executors, assign partitions on which to perform the tasks, return feedback on completed tasks or error messages, etc. The master node is key to ensuring optimal resource allocation, load balancing, and data shuffling among data nodes, along with maintaining network connectivity and data integrity. For more details on how distributed file systems work, we recommend the Google File System (GFS) [3] and the Apache Hadoop documentation[2]. A close overview of our dataset would help throw more light on the essence of using a Hadoop system for the project.

2.2 Description of dataset

Our data is sourced from the MCauley Group data repository[3]. It comprises 34 zipped JSONL files, ranging from tens of megabytes to tens of gigabytes. Each file contains individual customer reviews on products in categories such as beauty, fashion, appliances, etc. It is worth noting that the Amazon Reviews datasets constitute an important benchmark actively used to validate state-of-the-art machine learning models for sentiment analysis [5, 6] and recommendation systems [7, 10]. Our focus, however, is to demonstrate the ETL processes of these datasets using basic Spark algorithms.

Using Python's Request and BeautifulSoup libraries, the URL for each file is extracted from the source repository. Following this, we dynamically retrieve the data from the web by applying the wget command on each URL. Files larger than 5GB are discarded to conserve space on our cluster with a capacity of 40GBx3, while smaller files are stored in Hadoop. In the end, 29 out of the 34 files are ingested. These files are automatically duplicated in the Hadoop system, amounting to a total of roughly 80GB of data. The Hadoop and wget commands used in the ingestion process are shown in the pseudocode in Algorithm 1. Once the ingestion process is completed, we read the data using the `read.json` method of Spark SQL. The output of the `read` method is a computational logic structure representing a DataFrame. This DataFrame is materialized by actions such as the `show` or `printSchema` method. The schema of the read DataFrame reveals that the files are semi-structured (see Fig. 2). Each file results in a DataFrame comprised of 11 wide columns and millions of rows. An enhanced summary of our datasets is provided in Table 1.

Algorithm 1 Data ingestion process

```
Get URL
Set file_limit = 5     # max file size to ingest
Set LIMIT = 100,       # max total data size to ingest
# check if file is already on the cluster
found = !hdfs dfs -find /user/ubuntu/project -iname
"{URL}"
if len(found) then
    Return 0
end if
# Check disk usage and abort if LIMIT has been exceeded
used = !hdfs dfs -du -s /user/ubuntu/project/
total_usage = used[0].split()[1] / 1e9
if total_usage > LIMIT then
    Return 0
end if
# Download file from web; outputs filename.gz
!wget "{URL}"
if getsize(filename) > file_limit then
    !rm "{filename}"
    Return 0
end if
# move downloaded file from local namenode into hadoop cluster
!hdfs dfs -moveFromLocal "{filename}"
/user/ubuntu/project/
Return 1
```

```
root
 |-- asin: string (nullable = true)
 |-- helpful_vote: long (nullable = true)
 |-- images: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- attachment_type: string (nullable = true)
 |    |    |-- large_image_url: string (nullable = true)
 |    |    |-- medium_image_url: string (nullable = true)
 |    |    |-- small_image_url: string (nullable = true)
 |-- parent_asin: string (nullable = true)
 |-- rating: double (nullable = true)
 |-- text: string (nullable = true)
 |-- timestamp: long (nullable = true)
 |-- title: string (nullable = true)
 |-- user_id: string (nullable = true)
 |-- verified_purchase: boolean (nullable = true)
 |-- date_time: string (nullable = true)
```

Figure 2: Data schema inferred by `json.read` method for each dataset. The *images* column makes the data unstructured.

2.3 Spark configurations

Going further, it is important to understand how a Spark session is configured and launched to maximize resource utilization. We

[2]https://hadoop.apache.org/docs/r3.3.6/hadoop-project-dist/hadoop-common/FileSystemShell.html#mkdir
[3]https://datarepo.eng.ucsd.edu/mcauley_group/data/amazon_2023/raw/review_categories

Table 1: Summary of datasets

Item	Value	Note
Format	JSONL	Archived (gz)
Size on disk	40G	Archived (gz)
Size on disk (with replicas)	80G	Archived (gz)
Structure	semi-structured	See Fig. 2
Schema	11 columns (6 String, 2 Integer, 1 Array, 1 Double, 1 Boolean)	Rows $\sim$ O (M) customers in each product category
Product categories	34	
Data source	MCauley Group data repository	Customer review data from 2012-2023

already gave some insight into the resources we have to manage. These include storage, memory, and CPU processing units. From the Hadoop configuration discussed earlier, we see that the more nodes we have, the more resources we have. Therefore, having more nodes will greatly enhance parallel computing on the cluster. So far, our storage requirements indicate that at least 3 nodes are sufficient to ingest 37.6GB of data. The replicated data (75.3GB) is shared among the 3 nodes so that each node hosts approximately 25.1GB. Other files, such as library installation and log files, also take up additional storage (roughly 8.8 GB on each datanode). After ingesting the data we observed (using `hdfs dfsadmin -report`) that the data got distributed into 29.8GB, 27.1GB, and 29.9GB among datanode1, datanode2, and datanode3, respectively. The reason for the slight skew in load balance might be attributed to the fact that YARN thrives to achieve the data locality property. Data from the same files are prevented from splitting across nodes unless when they are too large. The distribution of data across nodes, as facilitated by YARN, aims to maintain data locality, thereby minimizing data movement across the network and enhancing performance. Ensuring a balanced distribution of data across the nodes helps prevent bottlenecks.

Table 2: Spark configuration choices for a 4-node cluster comprised of 3 worker nodes, each hosting 4 cpu cores

Number of cores per executor	1	2	3
Number executors per node	3	2	1
Executor memory	2GB	4GB	6GB
Executor description	thin	medium	thick

The Spark engine is run by a program called the Spark driver, usually hosted in the namenode. Its role is to create Spark executors, analyze user code, and distribute tasks to the executors. The executors are hosted in the datanode, and use the node memory and CPU cores to perform tasks concurrently. The level of parallelism achieved is typically proportional to the number of executors available. However, having too many executors for a given tasks can also create processing overheads resulting from resource contentions. Therefore the number of executors to use is mainly guided by: the size and complexity of the task, the number of CPU cores available, and total available memory. For our cluster resource contention might never be an issue, given that we cannot have beyond 12 executors (assuming we assign 1 cpu core per executor) and the tasks is time-consuming (given the large data involved). Nevertheless we have with 4 cores in each node, we have at least three options: Assigning 1-3 cores per executor, which will lead to the configuration choices enlisted in Table 2. The configuration choices have been classified as thin (1 core per executor, 2G memory), medium (2 cores per executor, 4G memory), and thick (3 cores per executor, 6G memory). Note that we cannot assign 4 cores per executor, as this might lead to memory overhead. We assume that at least 20% of available memory should be reserved to caching, heap, and other processes like driver and YARN manager interactions, data shuffling etc.

The following steps illustrates how we plan the resource allocations for a thin Spark configuration:

Given

$$\begin{aligned} \text{nodes} &= 3 \\ \text{memory per node} &= 8 \\ \text{cores per node} &= 4 \\ \text{cores per executor} &= 1 \end{aligned}$$

We get

$$\begin{aligned} \text{executors per node} &= (\text{cores per node}) \,/(\text{cores per executor}) \\ &= 4/1 \\ &= 4 \\ \text{memory per executor} &= (\text{memory per node})/(\text{executors per node}) \\ &= 8/4 \\ &= 2 \end{aligned}$$

To use only 80% of memory for Spark executors:

$$\begin{aligned} \text{total availabe memory} &= 0.8 \times (\text{memory per node}) \times \text{nodes} \\ &= 0.8 \times 8 \times 3 \\ &= 19.2 \\ \text{total number of executors} &= (\text{total availabe memory})/(\text{memory per executor}) \\ &= 19.2/2 \\ &= 9.6 \\ \text{executors per node} &= (\text{total number of executors}) \,/\text{nodes} \\ &= 9.6/3 \\ &= 3.2 \approx 3 \end{aligned}$$

Once we decide how many cores to assign for each executor, similar mathematical calculations can be used to determine the optimal

number of executors to assign in each node. The Spark configuration documentation[4] also provides the option to enable dynamic allocation of executors, so that the driver adapts the number of executors in each node according to the stage of the task currently running.

We shall proceed to discuss the Spark algorithms implemented in our use cases and also briefly discuss how Spark analyzes the steps of these algorithms.

3 METHOD

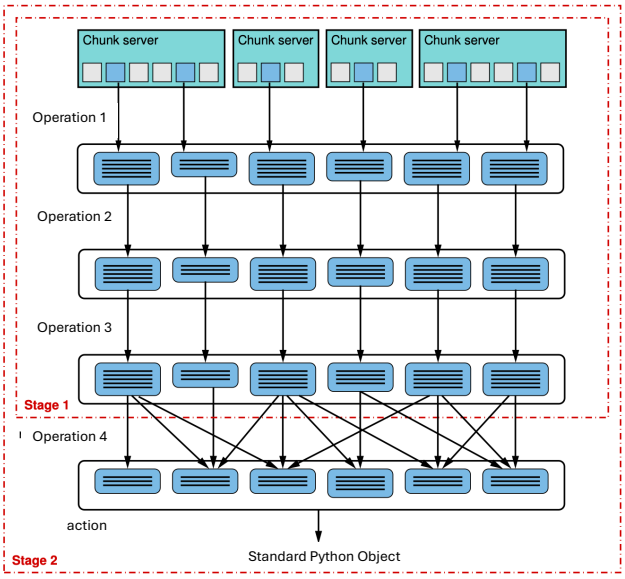

Figure 3: A lineage graph comprised of 4 operations/transformations and 1 action, represented by 2 stages. Operations 1-3 have narrow dependencies, but operation 4 has wide dependency.

The first step entails reading the files and extracting relevant fields (productID/asin, data, rating). The files are read into a DataFrame (`df`) using the `read.json` method. Alternatively, we could have read the files directly into an RDD and preprocessed the data before converting it into a DataFrame. The steps taken in this direction are detailed in the appendix section A.1. However, we found it more efficient to begin with the DataFrame read method, as our tasks involved minimal text processing.

Our datasets are very clean, requiring little or no preprocessing steps except for converting the time-stamp field into a date format. For sanity checks, a line is added to drop any record (row) from the Dataframe with no review records (null text and title fields). The extraction process is summarized using the pseudocode in Algorithm 2. After reading, it is observed that `df` has been partitioned into 29 parts, corresponding to the 29 product categories. This confirms that YARN has enforced data locality by putting all rows of the same category into the same node.

[4]https://spark.apache.org/docs/3.5.1/configuration.html

Partitions are blocks that facilitate how the Spark catalyst optimizes computations of transformations in a task. A *transformation* is a DataFrame operation (sequence of mapReduce tasks) that takes as input a DataFrame (parent) and returns another DataFrame (child, possibly with different partitioning from the input). A task typically involves a composition of transformations chained into a pipeline. This pipeline is usually broken into stages using a lineage graph based on the complexity of the transformations. See Fig. 3. This happens before any actual computation takes place. A transformation is said to have narrow network complexity if no two child partitions derive from the same parent partition. Otherwise, it has wide network partitioning. Wide transformations are costly as they require shuffling data across the network. Examples include: `groupBy, OrderBy, join` etc. Therefore, it is advisable to avoid using many such operations when designing efficient Spark algorithms. Stage boundaries in the lineage graph are marked by transformations with wide dependencies. Unlike transformations, *actions* are DataFrame operations that trigger the materialization of the DataFrame as defined by the lineage graph. Such actions include `show, count, collect, rdd.getNumPartitions, write`, etc. See the Spark documentation[5] for more examples.

Algorithm 2 Data extraction

```
# Read file with json library
Get file_path      # to category or (category list)
df = spark.read.json(file_path)
# Select relevant columns and clean up
df = (df.withColumn('date',
     F.to_date(F.from_unixtime(df.timestamp / 1000),
     'yyyy-MM-dd HH:mm:ss'))
     .drop('images').drop('timestamp')
     .na.drop(how='all', subset=['text', 'title']))
# Extract asin (product id), date, and rating to for given YEAR
if YEAR is None then
   df = df.select(['asin', 'date', 'rating'])
          .to(schema)
else
   df = df.select(['asin', 'date', 'rating'])
          .where(F.year(df.date)==YEAR).to(schema)
end if
```

The primary task in the transformation step involves aggregating the reviews to calculate review counts for every product each month. This is achieved through the `groupBy` operation, which is an example of a wide transformation and is computationally intensive. Refer to Algorithm 3 for more details. Another demanding transformation in this algorithm is `OrderBy`, but it is significantly less costly because the data has already been markedly reduced in terms of row and partition count at this stage. Lastly, the output DataFrame is stored using the `write.csv` operation (as depicted in Algorithm 4).

To determine the efficiency of each algorithm—extraction, transformation, and storage—the `show` method is used to initiate each process. The execution time for each algorithm is recorded to assess their performance.

[5]https://spark.apache.org/docs/3.4.1/api/python/index.html

Algorithm 3 Data transformation

```
# Map each month using one-hot encoding
for k, col in months.items() do
   df = df.withColumn(col,
   (F.month(df.date) == k).cast(IntegerType()))
end for
# Aggregate data using groupBy and agg_func
df = df.groupBy(df['asin']).agg(agg_func)
# Select columns using a short_listed columns
df = df.select(short_listed + list(months.values()))
# Compute custom-defined
# - total monthly reviews (total_reviews)
# - linear regression coefficient (linRegCoef) on monthly reviews
df = (df.withColumn('linReg', linRegCoef(df))
   .withColumn('total_reviews', total_reviews(df)))
# Keep rows with at least 12 total reviews,
# and sort in descending order of total reviews
df = (df.filter(df.total_reviews >= 12)
       .orderBy('total_reviews', ascending=False))
```

Algorithm 4 Data storage

```
df.write.csv(f"amazonReviews{YEAR}.csv",
                                mode="overwrite")
```

4 MOST REVIEWED PRODUCTS OF THE YEAR

The first use case focuses on identifying products with the highest number of reviews in 2023. In addition, the transformation step is adjusted so that the 'date' column is substituted with the dates of the first and last review for each product. This experiment is confined to the year 2023, with the goal of tracking which products garnered the most customer attention and measuring the trend of customer reviews during this period.

Using Algorithm 2, we iterate through all product categories and apply the *union* operator to aggregate all customer reviews for each category into a single DataFrame. This results in a DataFrame with approximately ($O(14M)$) rows and 4 columns: 'asin' (product ID), 'date', 'rating', and 'prod_category'. A snippet of the top 20 rows can be seen in Fig. 4. This combined DataFrame then undergoes the transformation step outlined in Algorithm 3. The transformed DataFrame consists of approximately ($O(200K)$) rows, each representing unique products, and contains 19 columns: 'asin', 'avg_rating', 'prod_category', 'first_review_date', 'last_review_date', and monthly review counts from 'Jan' to 'Dec', as well as 'linRegCoeff' and 'total_reviews'. See Fig. 5. The performance for each processing step with various Spark configurations, including one labeled as *dynamic* for dynamically allocated executors, is summarized in Table 3. The last column (ETL) shows the results obtained by applying extraction, transformation, and loading algorithms within a unified pipeline.

The results indicate minimal differences in performance across the four configurations evaluated. However, the dynamic configuration consistently outperforms the others in all three scenarios, with the medium configuration following closely behind. There is no significant performance difference between the thin and thick

```
+----------+----------+------+-------------+
|      asin|      date|rating|prod_category|
+----------+----------+------+-------------+
|B0BFR5WF1R|2023-02-08|   1.0|   All_Beauty|
|B0BL3HSBZB|2023-01-22|   1.0|   All_Beauty|
|B0BSR6WK1Q|2023-03-11|   4.0|   All_Beauty|
|B07TT8JK51|2023-01-05|   4.0|   All_Beauty|
|B09XBSDCXP|2023-02-12|   1.0|   All_Beauty|
|B08RRSPNWV|2023-02-15|   2.0|   All_Beauty|
|B0BNV4WRF3|2023-01-21|   1.0|   All_Beauty|
|B07FKVTYJX|2023-02-13|   3.0|   All_Beauty|
|B005FH2SOO|2023-02-13|   5.0|   All_Beauty|
|B0BHR26GQJ|2023-01-08|   5.0|   All_Beauty|
|B017BGJLBE|2023-02-09|   5.0|   All_Beauty|
|B0BNQ5D7J2|2023-02-19|   5.0|   All_Beauty|
|B0BSR6RRJR|2023-03-07|   4.0|   All_Beauty|
|B09G22586Y|2023-03-07|   2.0|   All_Beauty|
|B09N9TZLG3|2023-01-05|   5.0|   All_Beauty|
|B0BQWTXV2Q|2023-02-06|   4.0|   All_Beauty|
|B0BVM6DL7C|2023-03-09|   4.0|   All_Beauty|
|B01N547VYL|2023-02-21|   5.0|   All_Beauty|
|B0BPCFJV2J|2023-02-10|   5.0|   All_Beauty|
|B0BMTXX727|2023-02-08|   5.0|   All_Beauty|
+----------+----------+------+-------------+
only showing top 20 rows
```

Figure 4: View of top 20 rows extracted for the year 2023

Table 3: Execution time (in minutes) of the each segment of the ETL algorithms for different Spark configurations (See Table2).

Spark Configuration	Extraction	Transformation	Load	ETL
thin	28.54	32.20	32.17	57.99
medium	26.53	31.75	31.91	61.10
thick	28.49	32.83	32.67	60.56
dynamic	25.82	31.41	31.55	60.60

configurations. This suggests that, for this use case—and possibly numerous others—it's advantageous to let the YARN manager dynamically allocate resources and find the optimal balance between thin and thick configurations. Additionally, the results from the final column (ETL) indicate that an overall performance gain of over 30-37% is achieved through the pipelining (chaining) of the algorithms.

For the subsequent use cases, the resource allocation will solely rely on the dynamic configuration.

In today's rapidly evolving world of information, the robustness of data processing algorithms is increasingly crucial. Therefore, for this use case, we are expanding the dataset to include data from 2017 to 2023 and applying extraction and transformation algorithms within a unified pipeline. This approach allows for streamlined processing and efficiency. We plan to iterate over the years 2023

asin	avg_rating	prod_category	first_review_date	last_review_date	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	linReg	total_reviews
B0BMBC1FJX	4.8	Pet_Supplies	2023-01-01	2023-09-10	402	365	782	937	667	454	356	188	4	0	0	0	-0.01	4155
B00T0C9XRK	3.5	Beauty_and_Person...	2023-01-01	2023-09-07	663	730	723	370	205	104	117	62	2	0	0	0	-0.01	2976
B09C6LW4XW	4.3	Kindle_Store	2023-01-01	2023-09-06	951	651	563	272	112	87	75	16	2	0	0	0	-0.01	2729
B08DVFZTTG	4.5	Health_and_Household	2023-01-01	2023-09-11	801	655	516	251	144	70	65	27	1	0	0	0	-0.01	2530
B0BCP3JP6F	4.8	Kindle_Store	2023-01-10	2023-08-31	1461	509	241	73	42	8	10	5	0	0	0	0	-0.01	2349
B08GF7YGCD	4.1	Health_and_Household	2023-01-01	2023-09-10	459	474	614	245	185	104	61	44	15	0	0	0	-0.01	2201
B00BAGTNAQ	4.5	Pet_Supplies	2023-01-01	2023-09-08	478	457	455	261	128	77	94	57	4	0	0	0	-0.02	2011
B0B76XG3B8	4.5	Cell_Phones_and_A...	2023-01-01	2023-08-31	856	462	291	140	72	46	29	17	0	0	0	0	-0.01	1913
B098NSHBQK	4.5	Beauty_and_Person...	2023-01-19	2023-09-08	76	282	807	247	178	115	103	84	4	0	0	0	-0.01	1896
B07N7PK9QK	4.3	Beauty_and_Person...	2023-01-01	2023-09-06	393	376	455	308	136	87	77	60	3	0	0	0	-0.02	1895
B09W8HQYP9	4.4	Health_and_Household	2023-01-01	2023-08-16	860	677	238	52	22	11	9	4	0	0	0	0	-0.01	1873
B09R1G4MSR	4.6	Pet_Supplies	2023-01-01	2023-08-27	580	455	421	109	100	96	58	37	0	0	0	0	-0.02	1856
B07Q1DLKBG	4.7	Patio_Lawn_and_Ga...	2023-01-01	2023-09-08	672	488	298	184	46	68	51	26	4	0	0	0	-0.01	1837
B0B76RKBLJ	3.1	Movies_and_TV	2023-01-27	2023-08-05	1093	588	42	7	6	3	1	1	0	0	0	0	-0.01	1741
B071D4DKTZ	3.8	Health_and_Household	2023-01-01	2023-09-08	518	398	421	197	97	27	22	19	2	0	0	0	-0.02	1701
B09TKRFWYZ	4.9	Beauty_and_Person...	2023-01-01	2023-09-02	728	505	188	80	80	47	30	14	2	0	0	0	-0.01	1674
B00DU5SRIY	3.9	Health_and_Household	2023-01-01	2023-09-07	351	357	446	216	122	55	58	52	4	0	0	0	-0.02	1661
B0B2ZG8QFW	4.3	Pet_Supplies	2023-01-01	2023-09-04	938	229	184	113	75	37	34	21	1	0	0	0	-0.01	1632
B0BG9TB26K	4.9	Sports_and_Outdoors	2023-01-01	2023-07-28	1314	182	91	16	1	3	5	0	0	0	0	0	-0.01	1612
B0BFPM5BF7	4.5	Health_and_Household	2023-01-01	2023-08-29	464	178	342	181	66	68	78	83	0	0	0	0	-0.02	1460

only showing top 20 rows

Figure 5: View of top 20 rows transformed from the year 2023

down to 2017, incrementally adding one year of data at a time. By doing so, we can closely monitor and measure execution times, providing valuable insights into how well the algorithm scales as data volume grows. The data sizes will be gauged by counting the rows of the extracted data. By using row counts as a measure, we can effectively estimate the volume of data being processed each year. This strategy not only helps in assessing the performance and scalability of the algorithms but also aids in anticipating potential challenges and improving efficiencies in data handling processes as the dataset continues to expand.

The results of this experiment are illustrated in Figure 6. From this plot, we can confidently conclude that the Extract-Transform algorithm demonstrates linear scalability with respect to the data, as indicated by the approximately linear relationship between execution time and the number of rows.

To further verify this assertion, we maintain the year as 2023 and iteratively apply the ETL pipeline while adding chunks of product categories to increment the data by approximately 5 GB with each iteration. The plot of execution times against raw data sizes, shown in Fig. 7 (produce from Table 4), confirms the linear scaling of the algorithm with respect to data volume.

Table 4: Performance of the ETL algorithm: Values plotted in Fig. 7

Data size (GB)	4.63	10.8	15.5	20.5	24.7	29.4	35.2	37.8
Exec. times (min)	7.71	17.6	25.5	32.8	39.1	46.9	56.4	60.1

Our next use case introduces a seasonality measure based on yearly periods of monthly review counts, which is essential given that our data represents a time series.

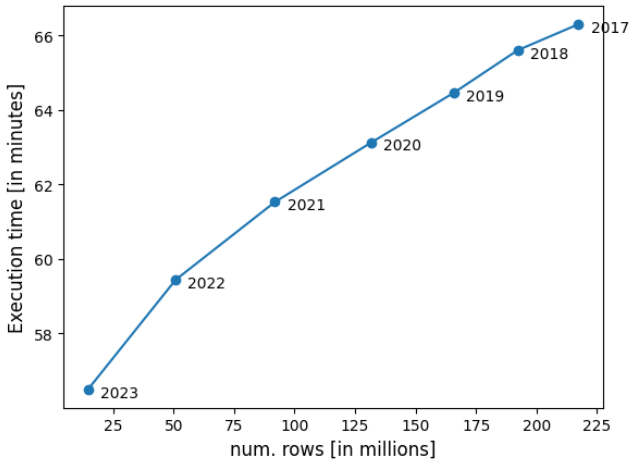

Figure 6: Performance of the Extract-Transform algorithm as data volume increases by iteratively adding years from 2023 down to 2017.

5 SEASONALITY

The concept of seasonality is pivotal in time series analysis as it reveals trends expected to recur over fixed time intervals. In this use case, we design a Spark algorithm by leveraging the previous two algorithms, 2 and 3. Initially, the extraction algorithm is applied to generate a DataFrame, denoted as `df`, comprising products across all categories. Subsequently, for any two consecutive years, `df` is filtered for each year and transformed to produce DataFrames `df1` and `df2`. These two DataFrames are then combined using the `join` operator. The resulting joined DataFrame is manipulated to include a column that contains the correlation coefficients for monthly reviews sourced from both years. The correlation coefficient provides

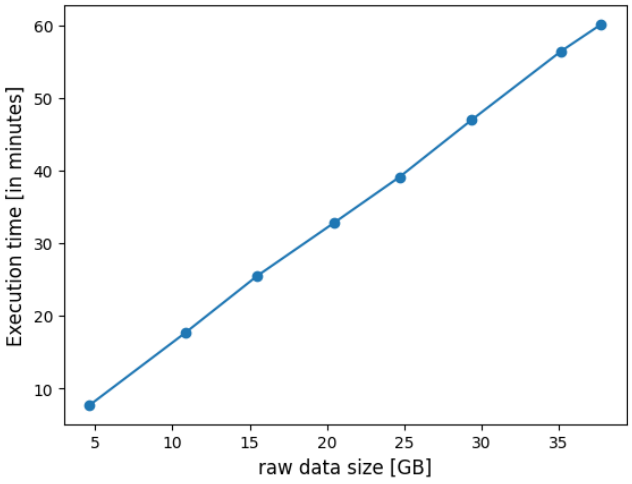

Figure 7: Performance of the Extract-Transform-Load (ETL) algorithm as data volume increases by iteratively adding 5GB of product category data.

a measure of the seasonally trend for each product across the two years compared. The entire procedure requires the definition of two custom functions: `extraction_transformation(year=YEAR, Df=None)`, which pipelines the extraction and transformation of data for a specified year, and `rowwise_corr(df, year1, year2)`, which calculates the row-wise correlations between the two years. Code snippets for these functions provided in section A.2. The entire use case is implement as follows:

```
year1 = 2022
year2 = 2023
df = extraction(categories)
df1 = extraction_transformation(year1, df)
df2 = extraction_transformation(year2, df)

df = df1.join(df2, on='asin', how='inner')

df = df.withColumn(f'corr({year1}, {year2})',
                   rowwise_corr(df, year1, year2))
df.show()
```

Additional cost in this algorithm comes from the use of SQL `join` operator, being an operator with wide network complexities. The results did show high correlations some products when comparing 2022 with 2023. However, the algorithm appears computationally untenable to run on our customized Hadoop cluster as it takes about 166.48 minutes to execute (nearly tripple the amount of time consumed for the ETL process for one year).

6 CONCLUSIONS

Throughout this project, we explored various Spark methodologies and algorithms for analyzing and processing large volumes of data on a Hadoop cluster. We also learned how to configure a Hadoop cluster within cloud computing systems. Key considerations were made to optimize code for effective utilization of computing resources, including storage and computational power. We successfully wrote compact code using Spark DataFrames and SQL commands, while also evaluating the scalability of the code in relation to data size. Additionally, we addressed significant use cases that provide meaningful value for the business chain. However, due to limitations in the number of worker nodes and computing cores, we were unable to thoroughly assess the impact of repartitioning datasets to increase parallelism within the distributed system. Operations on DataFrames such as `join` and `repartition`, as well as RDD methods like `rdd.getNumPartitions` proved to be extremely resource-intensive due to their complex lineage.

During this project, we occasionally encountered issues when attempting to initiate a new Spark session. After consulting ChatGPT, we found that the problems were primarily due to an overcrowded cluster as the log data increased. This caused challenges for YARN when it tried to connect with data replicas on some data nodes. Upon checking the cluster with `hdfs dfsadmin -report`, we chose to delete some of the data. Specifically, data from the Books and Electronics categories, each exceeding 5GB, was removed. Additional categories not included are Clothing, Shoes, and Jewelry (7.23GB), Home and Kitchen (8.42GB), and Unknown (8.44GB). These five omitted categories are crucial to the business chain. However, to include them, we would need to provide 36.44 GB of storage (or 72.88GB, with replicas), necessitating the expansion of the cluster by at least 3 extra VMs. In the future, it would be good to start with at least 10 VMs.

Additionally, we encountered another issue where log data in the `.sparkStaging` directory grew to over 10GB within four days. This issue surfaced when Spark failed to save the results. We consulted ChatGPT to adjust the Spark configuration, enabling the periodic deletion of log history. The configuration settings applied were `spark.history.fs.cleaner.enabled=true` and `spark.history.fs.cleaner.maxAge=1d`, which helped manage the log history by ensuring regular clean-up and maintaining system efficiency.

We also attempted to repartition the extracted data to increase the level of parallelism in the code. However, the additional cost of repartitioning outweighs any advantages that the increased parallelism may offer. This might be due to our limited computing resources in relation to the size of the data involved.

ACKNOWLEGEMENT

We hereby acknowledge that in this project we made use of ChatGPT as stated in the conclusion section. Grammarly's AI Chat (BETA) was also instrumental in proper structuring of sections of the report for improved readability.

A APPENDIX

A.1 MapReduce operations for data cleaning and preprocessing

In this section, we illustrate the alternative steps taken to extract and preprocess the data. This alternative approach utilizes SparkContext's `textFile` (or `wholeTextFiles`) from core Spark. The output of this method is an RDD composed of key-value pairs of strings. However, a significant drawback of this choice is that considerable effort must be expended on subsequent cleanup and data extraction.

To begin, the data is read using the following code snippets:

```
from pyspark.context import SparkContext
sc = SparkContext.getOrCreate()
rdd = sc.textFile(join(PROJECT_DIR, filenames[0])) \
    .map(lambda d: cleanText(d))
```

The custom function `cleanText` employs Python's regular expression library (re) to filter through each `rdd` record, transforming them into dictionaries that display the various fields of the dataset along with their corresponding values. The function is implemented as in shown in section A.2.

The RDD now comprises of a list of dictionaries. One item of this RDD looks as in the following example (using `rdd.take(1)`):

```
[{'rating': '5.0',
  'title': 'Such a lovely scent but not overpowering.',
  'text': "This spray is really nice. It smells really...",
  'images': '[]',
  'asin': 'B00YQ6X8EO',
  'parent_asin': 'B00YQ6X8EO',
  'user_id': 'AGKHLEW2SOWHNMFQIJGBECAF7INQ',
  'timestamp': '1588687728923',
  'helpful_vote': '0',
  'verified_purchase': 'true'}]
```

Converting this RDD into a DataFrame, we observe the following schema

```
root
 |-- asin: string (nullable = true)
 |-- attachment_type: string (nullable = true)
 |-- helpful_vote: string (nullable = true)
 |-- images: string (nullable = true)
 |-- large_image_url: string (nullable = true)
 |-- medium_image_url: string (nullable = true)
 |-- parent_asin: string (nullable = true)
 |-- rating: string (nullable = true)
 |-- text: string (nullable = true)
 |-- timestamp: string (nullable = true)
 |-- title: string (nullable = true)
 |-- user_id: string (nullable = true)
 |-- verified_purchase: string (nullable = true)
 |-- date_time: string (nullable = true)
```

The schema does not clearly indicate that the original data is semi-structured. However, from this DataFrame, the extraction step outlined in Algorithm 2 can be implemented with minimal modifications. Additionally, it is feasible to apply an extra map function on the RDD to filter the relevant fields of each dictionary. It is noteworthy that most of the methods employed in transforming our data are MapReduce methods that can also be directly applied to RDDs.

A.2 Some useful custom functions

This section contains definitions of a few custom functions mention in the rest of the report.

```
import  pyspark.sql.funtions as F

def cleanText(d):
    """ returns dictionary of key-value pairs
    after cleaning text d """
    # remove closing braces at ends of text,
    # and start text with ', '
    d = d.replace('{', ', ', 1)[:d.rfind('}')+1]
    # extract keys following the pattern: ', "key":'
    pattern = r",\s+\".*?\":"
    keys = re.findall(pattern=pattern,string=d)
    # clean up the keys
   keys = [re.sub(r'[\s",:]*', '', k).strip() for k in keys]
    # extract values
   vals = re.sub(pattern,'_key_', d).split('_key_')[1:]
    # clean values
    vals = [re.sub(r'"', '', v).strip() for v in vals]
    keyval = dict(zip(keys, vals))
    return keyval

# Monthly reviews map
months = {
    1 : 'Jan', 2 : 'Feb', 3 : 'Mar', 4 : 'Apr',
    5 : 'May', 6 : 'Jun', 7 : 'Jul', 8 : 'Aug',
    9 : 'Sep', 10 : 'Oct', 11 : 'Nov', 12 : 'Dec'
}

def linRegCoef(df, Sy=sum(months), n=12):
    Sx = sum([df[col] for (_, col) in months.items()])
   Sxx = sum([F.pow(df[col], 2) for (_, col) in months.items()])
   Sxy = sum([df[col]*k for (k, col) in months.items()])
    coef = (n*Sxy - Sx*Sy) \
        / (n*Sxx - F.pow(Sx, 2))
    return F.round(coef, 2)

def total_reviews(df):
   total_reviews = sum([df[col] for (_, col) in months.items()])
    return F.round(total_reviews, 1)


def rowwise_corr(df, year1, year2):
    suffix1 = str(year1)
    suffix2 = str(year2)
    innerprod = sum([df[col+suffix1]*df[col+suffix2]
                     for (_, col) in months.items()])
    L1 = sum([df[col+suffix1]*df[col+suffix1]
              for (_, col) in months.items()])
    L2 = sum([df[col+suffix2]*df[col+suffix2]
              for (_, col) in months.items()])
    corr = innerprod / F.sqrt(L1*L2)
    return F.round(corr, 2)

def extraction_transformation(year=YEAR, Df=None):
    # read method applied only if Df==None
    df = extraction(categories, year, Df)
    df = transformation(df)

    for col in df.columns[1:]:
        df = df.withColumnRenamed(col, col+str(year))

    return df
```

REFERENCES

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[2] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 29–43. https://doi.org/10.1145/945445.945450

[4] Damji JS and Denny Lee. 2020. *Learning Spark: lightning-fast data analytics (2. ed.)*. Sebastopol,O'Reilly Media, CA.

[5] Jianping Mei, Yilun Zheng, Qianwei Zhou, and Rui Yan. 2021. TaskDrop: A Competitive Baseline for Continual Learning of Sentiment Classification. arXiv:cs.CL/2112.02995 https://arxiv.org/abs/2112.02995

[6] Lei Pan, Yunshi Lan, Yang Li, and Weining Qian. 2024. Unsupervised Text Style Transfer via LLMs and Attention Masking with Multi-way Interactions. arXiv:cs.CL/2402.13647 https://arxiv.org/abs/2402.13647

[7] Theresia Veronika Rampisela, Tuukka Ruotsalo, Maria Maistro, and Christina Lioma. 2024. Can We Trust Recommender System Fairness Evaluation? The Role of Fairness and Relevance. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2024)*. ACM, 271–281. https://doi.org/10.1145/3626772.3657832

[8] Jayachander Surbiryala. 2024. Laboratory exercises for DAT535 Data-Intensive Systems and Algorithms.

[9] Wiktorski T. 2019. *Data-intensive systems: principles and fundamentals using Hadoop and Spark (2. ed.)*. Cham: Springer.

[10] Jinpeng Wang, Ziyun Zeng, Yunxiao Wang, Yuting Wang, Xingyu Lu, Tianxiang Li, Jun Yuan, Rui Zhang, Hai-Tao Zheng, and Shu-Tao Xia. 2023. MISSRec: Pre-training and Transferring Multi-modal Interest-aware Sequence Representation for Recommendation. In *Proceedings of the 31st ACM International Conference on Multimedia (MM '23)*. ACM, 6548–6557. https://doi.org/10.1145/3581783.3611967

[11] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, USA, 2.