UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION ENGINEERING AND ELECTRICAL AND APPLIED MATHEMATICS

Master's Degree in Computer Engineering

Project report Big Data

Hadoop MapReduce and Spark Exercises

Lecturer: Giuseppe D'Aniello gidaniello@unisa.it

Student and Creator: Agostino Cardamone 0622702276 a.cardamone7@studenti.unisa.it

ACADEMIC YEAR 2023/2024

Index

Figure index

1. Hadoop MapReduce Exercise

1.1 Problem Statement

Frequently, many online stores choose to reward customer loyalty by identifying those who stand out through specific parameters evaluated on their transaction data. Creating a classification of the most loyal customers becomes an interesting option for stores because it allows them to offer discounts or special promotions to these selected categories of customers. This approach not only incentivizes customer loyalty but also creates a business environment in which shoppers feel valued and motivated to maintain a long-term relationship with the online store.

By identifying the most loyal customers, the store can create personalized engagement strategies. This might include special offers, loyalty discounts, or exclusive events, fostering a sense of value and appreciation among these customers.

The staff can be informed about the most loyal customers, enabling them to provide exceptional, personalized service to these individuals. This could lead to increased customer satisfaction and further cement their loyalty.

Insights from the loyalty ranking can influence inventory control, ensuring that products favored by loyal customers are always in stock. It can also guide the sales strategy, focusing on products and services that appeal to this segment.

Loyal customers are often a great source of constructive feedback. Understanding who they are can help in gathering valuable insights for improving store offerings and the overall shopping experience.

1.2 Proposed Solution and Implementation: Mapping Customer Loyalty with Hadoop MapReduce

The proposed solution for analyzing customer loyalty in a single retail store involved the use of Hadoop MapReduce. The MapReduce exercise solution aims to create a system that accurately classifies customers based on their loyalty, so that the store is entrusted with a powerful tool for improving customer relationships. This ranking allows the store to focus its efforts on those who are most valuable to its business, ensuring that their needs are met and their loyalty is recognized. In turn, this strategy can lead to greater customer loyalty, increased sales, and a stronger brand reputation within the community.

The aim was to develop a program that could sift through the store's sales data, identify patterns in customer purchasing behavior, and ultimately rank customers based on their loyalty.

1.2.1 Key Components of the Implementation

The problem solution was implemented through the application of the skills acquired during the course, making use of the Hadoop MapReduce framework. Taking advantage of the MapReduce programming paradigm, we were able to divide the provided dataset into blocks during the map phase. Each block was processed by a specific map function in the program. Subsequently, through the Reduce phase, we grouped the results obtained after processing each block, thanks to the information mapping performed earlier.

The decision to use Hadoop is motivated by the efficient handling of large files that may exceed the processing capacity of a single node. Hadoop addresses this problem by dividing the work among numerous nodes in a cluster. Each node performs the map phase on local data, later sending the results to the reduce phase. This approach allows Hadoop to process large amounts of data in parallel. The Hadoop framework coordinates the execution of the MapReduce program, facilitating the simultaneous execution of the map and reduce phases.

A MapReduce program in Hadoop consists of three basic components, each implemented through a dedicated class:

- **Driver**: The Driver class contains the method or code that manages the job configuration and workflow of the application. An instance of this class is responsible for running the MapReduce program.
- **Mapper**: The Mapper class implements the mapping function. An instance of this class is the object responsible for executing the mapping phase of the MapReduce program.
- **Reducer**: The Reducer class implements the reduction function. An instance of this class is the object responsible for executing the reduction phase of the MapReduce program.

These components work together to process data in a distributed manner. In the context of Hadoop, the programming language used to write such MapReduce programs is **Java**.

The developed solution is structured around the three core classes of the MapReduce paradigm, flanked by an auxiliary class dedicated to serializing and deserializing information crucial to identifying loyal client characteristics.

- **DriverCustomerLoyalty.java**

  The main role of this class as with any driver class for Hadoop is to set up and control the execution of the MapReduce job. The class, through the main method, handles the configuration of the crucial parameters that govern the work. Responsibilities include specifying input and output formats, identifying classes for the mapper and reducer, and defining key types and output values. It is also responsible for managing the submission and execution of the MapReduce job within the Hadoop cluster; the class instantiates a Job object, which encapsulates the complete job configuration.

```java
public class DriverCustomerLoyalty {

    /**
     * The main method is the entry point of the program.
     *
     * @param args Command line arguments: [0] = input path, [1] = output path
     * @throws Exception If an error occurs during job execution.
     */
    public static void main(String[] args) throws Exception {
        // Verify the correct syntax of input parameters
        if (args.length != 2) {
            System.err.println( x: "Usage: DriverCustomerLoyalty <input path> <output path>");
            System.exit( status: -1);
        }

        // Configure Hadoop job
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, jobName: "Customer Loyalty Analysis");
        job.setJarByClass( cls: DriverCustomerLoyalty.class);
        job.setMapperClass( cls: MapperCustomerLoyalty.class);
        job.setReducerClass( cls: ReducerCustomerLoyalty.class);

        job.setMapOutputKeyClass( theClass: Text.class);
        job.setMapOutputValueClass( theClass: CustomerDataWritable.class);

        job.setOutputKeyClass( theClass: Text.class);
        job.setOutputValueClass( theClass: NullWritable.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // Execute the job and exit
        System.exit(job.waitForCompletion( verbose: true) ? 0 : 1);
    }
}
```

Figure 1: Source code of DriverCustomerLoyalty.java

- **MapperCustomerLoyalty.java: Mapper Class**

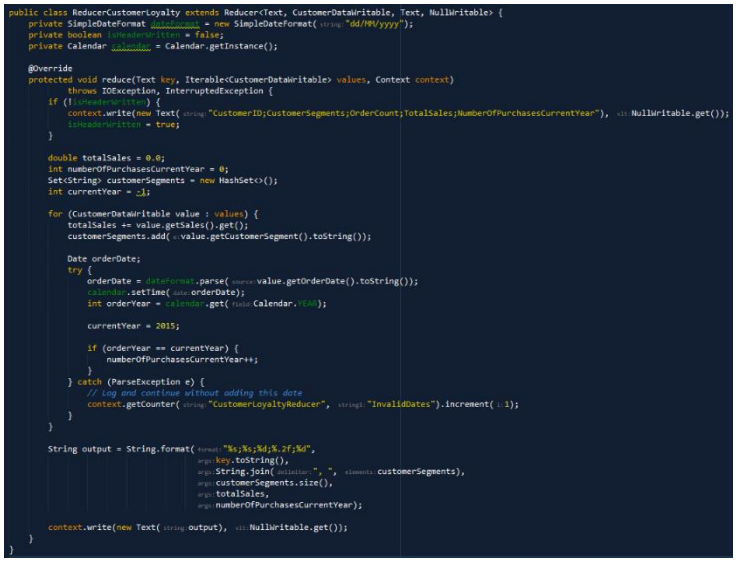
The class was designed to process the input data by extracting the detected information of each row and structure it into key-value pairs to facilitate the task of the later stages of the analysis. It extends Hadoop's *Mapper* class and overrides the map method to process each line of input data. The class starts with a setup method that initializes a HashMap (*columnIndexMap*) for mapping column names to their indices, which is critical for accurately parsing the data. In the map method, the first input line (header) is used to populate this map. Subsequent lines are processed to extract customer-related data like customer ID, sales, order date, and customer segment. This extraction is guided by the column index map, ensuring that the right data is extracted from each line. The mapper then creates a *CustomerDataWritable* object with this data and writes it to the context, keyed by the customer ID. This setup is essential for efficient data processing and accurate mapping of each customer's transaction data.

```java
public class MapperCustomerLoyalty extends Mapper<LongWritable, Text, Text, CustomerDataWritable> {
    private Map<String, Integer> columnIndexMap = new HashMap<>();
    private boolean isHeader = true;

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        columnIndexMap = new HashMap<>();
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        // Split the input line into fields
        String[] fields = value.toString().split( regex: ";");

        // Skip the header line and setup the column index mapping
        if (isHeader) {
            for (int i = 0; i < fields.length; i++) {
                columnIndexMap.put( key: fields[i].trim(),  value: i);
            }
            isHeader = false;
            return;
        }

        // Extract fields for each record
        String customerID = fields[columnIndexMap.get( key: "Customer ID")];
        double sales = Double.parseDouble(fields[columnIndexMap.get( key: "Sales")]);
        String orderDate = fields[columnIndexMap.get( key: "Order Date")];
        String customerSegment = fields[columnIndexMap.get( key: "Customer Segment")];

        // Create a new CustomerDataWritable object
        CustomerDataWritable customerData = new CustomerDataWritable(orderDate, sales, customerSegment);

        // Write the output key-value pair
        context.write(new Text( string: customerID),  v1t: customerData);
    }
}
```

Figure 2: Source code of MapperCustomerLoyalty.java

- **ReducerCustomerLoyalty.java: Reducer Class**

The fundamental objective of the class is to aggregate data emanating from the Mapper phase. This class extends Hadoop's *Reducer* and focuses on processing grouped data (by customer ID). Leveraging the *Iterable* collection of values emitted by the Mapper, the class consolidates relevant information for further analysis. In its reduce method, the class aggregates total sales and counts the number of purchases made by each customer in the current year. Once the aggregation is complete, the class emits the final aggregated data for each customer. This structured output serves as valuable input for subsequent stages of the analysis, contributing to the overall understanding of customer loyalty dynamics.



```
public class ReducerCustomerLoyalty extends Reducer<Text, CustomerDataWritable, Text, NullWritable> {
    private SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
    private boolean isHeaderWritten = false;
    private Calendar calendar = Calendar.getInstance();

    @Override
    protected void reduce(Text key, Iterable<CustomerDataWritable> values, Context context)
            throws IOException, InterruptedException {
        if (!isHeaderWritten) {
            context.write(new Text("CustomerID;CustomerSegments;OrderCount;TotalSales;NumberOfPurchasesCurrentYear"), NullWritable.get());
            isHeaderWritten = true;
        }

        double totalSales = 0.0;
        int numberOfPurchasesCurrentYear = 0;
        Set<String> customerSegments = new HashSet<>();
        int currentYear = -1;

        for (CustomerDataWritable value : values) {
            totalSales += value.getSales().get();
            customerSegments.add(value.getCustomerSegment().toString());

            Date orderDate;
            try {
                orderDate = dateFormat.parse(value.getOrderDate().toString());
                calendar.setTime(orderDate);
                int orderYear = calendar.get(Calendar.YEAR);

                currentYear = 2015;

                if (orderYear == currentYear) {
                    numberOfPurchasesCurrentYear++;
                }
            } catch (ParseException e) {
                // Log and continue without adding this date
                context.getCounter("CustomerLoyaltyReducer", "InvalidDates").increment(1);
            }
        }

        String output = String.format("%s;%s;%d;%.2f;%d",
                                      key.toString(),
                                      String.join(", ", customerSegments),
                                      customerSegments.size(),
                                      totalSales,
                                      numberOfPurchasesCurrentYear);

        context.write(new Text(output), NullWritable.get());
    }
}
```

Figure 3: Snippet code of ReducerCustomerLoyalty.java

- **CustomerDataWritable.java: Custom Writable Class**

This class is a custom implementation of Hadoop's Writable interface, specifically designed to encapsulate and manage data related to customer transactions in the context of the MapReduce job. The class includes three primary fields: *orderDate*, *sales*, and *customerSegment*, each represented by appropriate Hadoop data types (*Text* for orderDate and customerSegment, and *DoubleWritable* for sales). These fields are essential for collecting the transaction information of each customer, such as the amount spent (*sales*) and customer category (*customerSegment*). The serialization methods of the class (*write* and *readFields*) are implemented to manage the data so that it can be used properly during the MapReduce process.

```
public class CustomerDataWritable implements Writable {
    private Text orderDate;
    private DoubleWritable sales;
    private Text customerSegment;

    // Default constructor
    public CustomerDataWritable() {
        set(new Text(), new DoubleWritable(), new Text());
    }

    // Parameterized constructor
    public CustomerDataWritable(String orderDate, double sales, String customerSegment) {
        set(new Text(orderDate), new DoubleWritable(sales), new Text(customerSegment));
    }

    // Setter method
    public void set(Text orderDate, DoubleWritable sales, Text customerSegment) {
        this.orderDate = orderDate;
        this.sales = sales;
        this.customerSegment = customerSegment;
    }

    // Getter methods
    public Text getOrderDate() {
        return orderDate;
    }

    public DoubleWritable getSales() {
        return sales;
    }

    public Text getCustomerSegment() {
        return customerSegment;
    }

    // Serialization method
    @Override
    public void write(DataOutput out) throws IOException {
        orderDate.write(out);
        sales.write(out);
        customerSegment.write(out);
    }

    // Deserialization method
    @Override
    public void readFields(DataInput in) throws IOException {
        orderDate.readFields(in);
        sales.readFields(in);
        customerSegment.readFields(in);
    }
}
```

Figure 4: Source Code of CustomerDataWritable.java

1.2.2 MapReduce Design Pattern and Configuration Choices

The MapReduce job for analyzing customer loyalty in a retail setting primarily utilized the *Numerical Summarization* pattern. This pattern is adept at grouping records based on specific key fields and calculating numerical aggregates for each group. In our case, the focus was on aggregating sales data to derive insights into customer loyalty.

Mappers' Role (from MapperCustomerLoyalty.java)

- The Mappers were tasked with processing each input record, extracting key information such as customer ID, sales, and other relevant details.
- They produced key-value pairs, with the customer ID serving as the key and a custom CustomerDataWritable object (containing sales and customer segment data) as the value.
- This step aligns perfectly with the Numerical Summarization pattern, where the goal is to prepare data for aggregation based on a grouping key.

Reducers' Function (from ReducerCustomerLoyalty.java)

- The Reducer received collections of CustomerDataWritable objects grouped by customer ID.
- It computed final statistics for each group, such as total sales and the number of purchases within the current year.
- This process of aggregating numerical data for each key is the essence of Numerical Summarization, providing a high-level view of the dataset.

Single Reducer Choice

The decision to use a single Reducer was influenced by the size of the dataset. Given its manageable size, it was feasible to process and aggregate the data without the need for distributed computing across multiple Reducers. A single Reducer was capable of efficiently handling the computation, making it a practical choice for this scenario. The choice ensured the centralization of the calculation process, avoiding the complexities and overheads associated with multiple Reducers.

Decision Against Using Combiners

Combiners, often used in MapReduce for intermediate aggregation, were not utilized in this job. The rationale was based on the nature of the aggregation tasks and the dataset's characteristics. The aggregations (like summing sales and counting transactions) were straightforward and not computationally intensive. In such cases, the introduction of Combiners might not lead to significant performance improvements, especially considering the dataset's modest size. Moreover, the final aggregation logic in the Reducer required a complete view of each customer's data.

1.3 Execution Results

The execution of the MapReduce job on the Hadoop cluster resulted in an output that provides a detailed view of customer loyalty metrics. This output, obtained through the aggregation and analysis of sales data, offers valuable insights into customer purchasing behavior.

The output data, as saved in *output_local.csv*, consists of aggregated information for each customer, including metrics like total sales, the number of purchases in the current year, and customer segments. Each line in the output represents a unique customer, providing a consolidated view of their interaction

with the store. This data is important for understanding which customers are most loyal and their purchasing patterns, thereby enabling the store to tailor its marketing and sales strategies accordingly.

CustomerID	CustomerSegments	OrderCount	TotalSales	NumberOfPurchasesCurrentYear
11	Home Office	1	211.15	1
14	Small Business	1	1214.93	4
15	Small Business	1	413.00	2
16	Small Business	1	964.79	2
18	Small Business	1	455.77	1
19	Small Business	1	231.79	1
21	Small Business	1	3084.04	3
2198	Small Business	1	771.39	2
2201	Small Business	1	27.96	1
2202	Home Office	1	1733.91	3
2203	Home Office	1	1924.66	2

Figure 5: A part of the output_file.csv result of the program execution.

Detailed Explanation of the Commands Executed

The following sequence of commands was used to configure, execute, and manage the MapReduce job:

1. Setting Up the Environment:
    - *cd C:\Users\agost\Documents\Universita\HPC\hadoop-cluster-3.3.6-amd64*: Navigating to the Hadoop cluster directory.
    - *docker build -t hadoop-new*: Building a new Docker image for the Hadoop setup.
    - *docker network create --driver bridge hadoop_network*: Creating a new network in Docker using the bridge network driver.
    - *docker compose up -d*: Starting all services defined in the Docker Compose file in detached mode and creating the Docker container.
2. Initializing and Managing Hadoop Services:
    - *docker container exec -ti master bash*: Accessing the master node of the Hadoop cluster via Docker.
    - *hdfs namenode -format*: Formatting the HDFS namenode.
    - *$HADOOP_HOME/sbin/start-dfs.sh*: Starting the Hadoop Distributed File System (DFS) services.
    - *cd data/*: Navigating to the data directory.
    - *hdfs dfs -put store3.csv hdfs:///input*: Uploading the dataset to HDFS.
    - *$HADOOP_HOME/sbin/start-yarn.sh*: Starting YARN for resource management and job scheduling.
3. Executing the MapReduce Job:
    - *hadoop jar Exercise1_BIGDATA.jar /input /output*: Running the MapReduce job with the specified input and output paths.
    - *hdfs dfs -cat /output/part-r-00000 > output_file.csv*: Retrieving the output from HDFS and saving it locally.
4. Cleanup and Shutdown Procedures:
    - *hdfs dfs -rm -r hdfs:///input*: Removing the input directory from HDFS.
    - *hdfs dfs -rm -r hdfs:///output*: Removing the output directory from HDFS.
    - *$HADOOP_HOME/sbin/stop-dfs.sh*: Stopping the DFS services.
    - *$HADOOP_HOME/sbin/stop-yarn.sh*: Stopping YARN.
    - *exit*: Exiting the master container.
    - *docker stop hadoop-new*: Stopping the Docker container.

2. Spark Exercise

2.1 Problem Statement

The problem statement for the second exercise focuses on utilizing Apache Spark to analyze sales data from a retail store. The task is to determine the regions with the highest and lowest sales figures. Spark, a powerful engine for Big Data analysis, excels beyond the capabilities of Hadoop MapReduce by offering fast, in-memory data processing and a suite of high-level tools.

In this context, Spark's resilience, efficiency, and the ability to process iterative computations quickly make it an ideal platform for this analysis. The problem involves reading the dataset, aggregating sales data by region, and computing the sum of sales in each region to identify the regions with the maximum and minimum sales. This exercise will leverage Spark's ability to cache data in memory across multiple parallel operations, ensuring efficient iterative processing and optimizing the execution plan.

2.2 Proposed Solution and Implementation

The objective of the second exercise centers around leveraging Apache Spark for the analysis of sales data obtained from a retail store. The goal is to identify the regions characterized by the highest and lowest sales figures.

The proposed solution makes use of Spark which is known to be a powerful tool for big data analysis and relies on its advanced in-memory processing capabilities, which offer a significant speed advantage over Hadoop's MapReduce. While MapReduce writes intermediate data to disk, Spark performs the calculations in memory and stores the intermediate results in distributed collections known as Resilient Distributed Datasets (RDD). This in-memory data sharing is faster and more efficient, particularly for applications involving iterative algorithms and interactive analysis.

2.2.1 Key Components of the Implementation

In the context of the retail sales data analysis, Spark is particularly well-suited for the task due to its ability to quickly aggregate large datasets and perform complex calculations. The goal is to gain insights into sales patterns across different regions, identifying both high-performing and underperforming areas. By leveraging Spark's fast data processing capabilities, the *RegionSaleAnalysisDriver* program can perform these computations with high efficiency, providing valuable information that can influence strategic business decisions.

In the *RegionSaleAnalysisDriver* Spark application, the architecture of the program reflects a deep integration of RDD transformations and actions, which are foundational to Spark's processing model. An RDD, or Resilient Distributed Dataset, is an immutable distributed collection of objects. It is the primary abstraction in Spark around which the data processing is centered, allowing for fault-tolerant operations across a distributed dataset.

1. When the application begins, it initializes the ***Spark context***, which is the conduit through which all Spark functionality is accessed. It then reads the dataset and performs a series ***transformation on the RDD***. The *textFile* method reads the input data into an RDD of strings, each string representing a line from the file. The *filter* transformation is then applied to exclude the header row, which is essential for accurate data analysis, especially when determining indices dynamically.
2. Then the ***mapToPair*** transformation takes each line of input and maps it into a tuple, the outcome is a pair RDD where each element is represented as a pair of (region, sales). This step is important as it sets the stage for aggregation.

- Spark then performs a ***data shuffling operation***, aligning all values bound to the same key on a single node. This operation is designed to bring together all values associated with the same key onto a unified node within the Spark cluster.
- For each distinct key, the ***reduceByKey*** method performs a reduce operation, specifically summing the sales data. By summing up the sales figures for each distinct key, this operation contributes significantly to the overall aggregation of data and the derivation of insights from the original dataset.
- Then a ***parallel reduction*** operation is performed whose process diverges from the traditional single-node execution and unfolds concurrently across diverse nodes within the Spark cluster. Instead of relying on a singular computing node, each node takes charge of processing a specific subset of keys along with their associated values. This parallelized approach enables simultaneous and distributed reduction operations, enhancing the efficiency of data processing.
- The result of this aggregation is a new RDD where each record represents a region and its corresponding total sales.

3. Once the data is aggregated, the application uses the ***max*** and ***min*** actions to identify the regions with the highest and lowest sales. These actions are notable for their utilization of Spark's parallel processing capabilities. A custom comparator serializable is used to ensure that these operations can be serialized and sent across the network for execution on different nodes.
   - The max and min operations are executed in parallel in the cluster thanks to the ***parallelize*** function. Spark distributes the task of comparing sales data among different nodes, ensuring a search process.
   - To perform these comparisons, a custom ***serializable comparator*** is used. This comparator defines how two sales figures are compared to identify the maximum and minimum values. The comparator must be serializable to ensure it can be sent over the network to different nodes.
   - Once the max and min values are identified on each node, Spark consolidates these results to determine the overall maximum and minimum sales regions.

4. The results of these operations are then collected and written out to the ***filesystem*** as text files. The final step of saving the output in text files uses the *saveAsTextFile* method, which distributes the task of writing the data to the file system among various nodes in the cluster. This action reflects the culmination of the data flow from input to processed output. Finally, the Spark context is closed, signaling the end of the program's execution, and releasing the resources held by the context.

2.2.2 Spark Design Pattern

Throughout this process, Spark's design patterns, such as ***lazy evaluation*** and ***in-memory computation***, allow for efficient execution of complex data processing tasks. Lazy evaluation means that transformations on RDDs are not computed immediately but are instead executed when an action requires a result to be returned to the Spark driver. This allows Spark to optimize the overall data processing pipeline, reducing the number of passes over the data and thus improving performance. In-memory computation, another cornerstone of Spark's design, enables fast data processing speeds, as intermediate data can be held in RAM rather than written to disk after each operation, contrasting with the more I/O-intensive approach of Hadoop's MapReduce.

2.3 Execution Results

The output from the RegionSaleAnalysisDriver Spark program, is represented in the *max.txt* and *min.txt* files, provides a focused analysis on the sales data across different regions. These files are the result of Spark's ability to process and analyze large data sets efficiently, offering valuable insights into regional sales performance.

The ***max.txt*** file contains the information about the region with the highest sales. The data encapsulated here includes the region's name and its corresponding total sales figure. This region represents the highest revenue-generating area in the dataset.

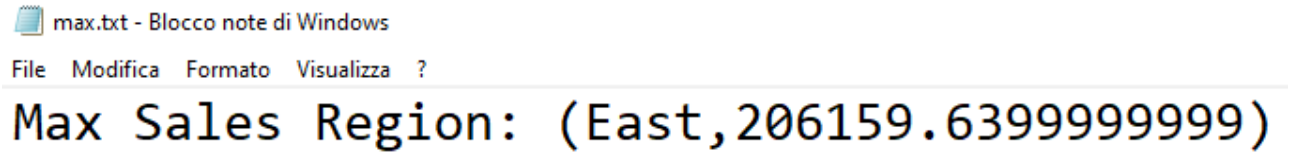


Figure 6: Content of the max.txt result file of the launch_single.sh execution

In contrast, the ***min.txt*** file identifies the region with the lowest sales. Similar to max.txt, it includes the name of the region and its total sales amount. This region represents the lowest revenue-generating area in the dataset.

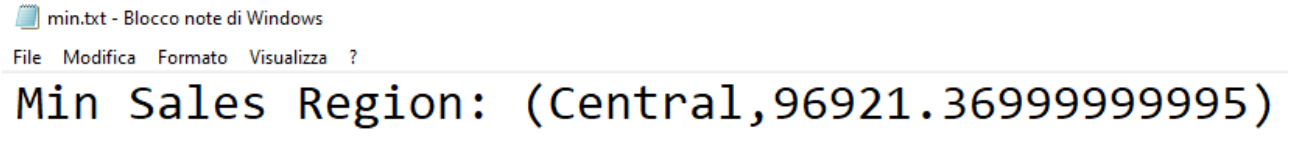


Figure 7: Content of the min.txt result file of the launch_single.sh execution.

Detailed Explanation of the Commands Executed

The following sequence of commands was used to set up the environment, executed the program, and managed the output for the execution on the cluster and in local:

1. Setting Up the Spark Environment:
   - *cd C:\spark-cluster*: Navigates to the directory containing the Spark cluster setup.
   - *docker compose up -d*: Starts the Spark cluster services using Docker Compose in detached mode.
2. Accessing the Spark Master Container:
   - *docker container exec -ti sp_master bash*: Opens a bash shell in the Spark master container for command execution.
3. Navigating and Preparing for Job Execution:
   - *cd bin/* and *cd testfiles/*: Changes the current directory to where the Spark job .jar file and launch scripts are located.
4. Executing the Spark Job and saving results:
   - *./launch_single.sh*: Executes the Spark job in a single-node environment.
   - *./launch_cluster.sh*: Executes the Spark job across the Spark.
   - *cd output1/max_sales/*, *cat part-00000 > max.txt*: Move to the correct folder and copies result in max.txt, same was done for min output file.

launch_cluster.sh:

```
/opt/bitnami/spark/bin/spark-submit \
  --class it.unisa.diem.hpc.spark.exercise2.RegionSaleAnalysisDriver \
  --master spark://spark-master:7077 \
  --deploy-mode client \
  --supervise \
  --executor-memory 1G \
  ./Exercise2_BIGDATA.jar \
./input ./output2
```

- */opt/bitnami/spark/bin/spark-submit *: This command is the standard way to submit a Spark application for execution. It tells the system to use the spark-submit script located in the specified directory.
- *--class it.unisa.diem.hpc.spark.exercise2.RegionSaleAnalysisDriver *: This option specifies the main class of the application, i.e., *RegionSaleAnalysisDriver*, which is part of the package *it.unisa.diem.hpc.spark.exercise2*.
- *--master spark://spark-master:7077 *: This sets the master URL for the cluster. spark://spark-master:7077 specifies that the Spark master node is running on the host spark-master and listening on port 7077.
- *--deploy-mode client *: This option indicates that the application is running in 'client' mode. In client mode, the driver is launched in the same process as the spark-submit command.
- *--supervise *: This flag requests that the Spark cluster manager restarts the driver should it fail.
- *--executor-memory 1G *: This sets the amount of memory to use per executor process, in this case, 1 gigabyte.
- *./Exercise2_BIGDATA.jar *: Specifies the path to the JAR file containing the application to run.
- *./input ./output2*: These are the command-line arguments passed to the Spark application, specifying the input and output directories for the job.

launch_single.sh:

```
/opt/bitnami/spark/bin/spark-submit \
  --class it.unisa.diem.hpc.spark.exercise2.RegionSaleAnalysisDriver \
  --master local \
  ./Exercise2_BIGDATA.jar \
  ./input ./output1
```

- */opt/bitnami/spark/bin/spark-submit *: This is the command used to submit the application to Spark. It specifies the path to the spark-submit script in the Spark installation directory.
- *--class it.unisa.diem.hpc.spark.exercise2.RegionSaleAnalysisDriver *: This option specifies the main class of the application to be executed. The class RegionSaleAnalysisDriver belongs to the package it.unisa.diem.hpc.spark.exercise2.
- *--master local *: This sets the master URL to local, indicating that the application will run on a single node in a local mode. This mode is typically used for development and testing purposes where the application runs locally on a single JVM.
- *./Exercise2_BIGDATA.jar *: This is the path to the JAR file containing the Spark application. It tells Spark where to find the application to run.
- *./input ./output1*: These arguments specify the input and output paths for the application. The application will read data from the ./input directory and write the results to the ./output1 directory.

2.3.1 Comparing Single-Node and Cluster Performance in Spark Applications

During the initial phases of the RegionSaleAnalysisDriver Spark application development, I employed **launch_single.sh** for running the program. This approach was used because:

- Running the application in a single-node environment greatly simplifies the debugging process. It allows for rapid identification and resolution of issues without the complexities of a distributed system.
- For initial testing, especially when working with smaller datasets or testing individual components of the application, a single-node setup is more resource-efficient.
- The local mode enabled quick turnaround times for code changes, as it did not require the overhead of setting up and communicating with a distributed cluster.
- This stage focused on validating the core logic, data transformations, and functionalities of the application, ensuring that it behaved as expected in a controlled environment.

After ensuring the application's correctness and stability in a single-node setup, I transitioned to using **launch_cluster.sh** for advanced testing and performance evaluation. This choice was made because:

- Running the application on a Spark cluster more accurately simulates real-world scenarios where the application will be deployed in a distributed environment.
- This approach allowed for testing and optimizing the application's performance under distributed computing conditions, including network latency and data shuffling between nodes.
- It was essential to understand how the application scales with increased data volume and cluster size, ensuring that it can handle large-scale data processing efficiently.
- The cluster mode enabled me to evaluate the application’s resource utilization, including memory and CPU usage, and tweak configurations for optimal performance.

Execution Time

- **Single-Node**: Exhibited an execution time of approximately 0.2 to 0.3 seconds for each job or stage. This is indicative of the efficiency in handling small-scale data processing tasks within a contained environment.
- **Cluster**: Showed a slightly variable execution time, ranging from 0.1 to 0.6 seconds per stage. While some stages performed faster, the overall time did not significantly improve compared to the single-node execution.

For our small dataset, the single-node environment was almost as efficient as the cluster mode. The cluster mode did not markedly improve execution times, suggesting that the benefits of distributed computing become more pronounced with larger datasets.

Resource Utilization

- **Single-Node**: Utilizes a single executor with limited CPU and memory resources. This setup is straightforward and effective for smaller datasets but lacks scalability.
- **Cluster**: Offers distributed computing resources, with tasks shared across multiple executors. This setup is scalable and can handle larger datasets more efficiently.

The cluster mode's ability to distribute tasks across multiple nodes with dedicated resources is a clear advantage for larger, more complex tasks. However, for smaller datasets, as in our case, single-node execution may offer a more resource-efficient approach.

Task Distribution and Load Balancing

- **Cluster**: The load was evenly distributed across two executors, demonstrating the cluster's ability to balance tasks effectively.

While the cluster mode ensured balanced load distribution, the impact was less significant for our small-scale task. This feature is more beneficial for larger datasets where parallel processing can significantly reduce execution times.

Data Shuffling and Network Overhead

- **Cluster**: The process involved data shuffling between executors, introducing network overhead.

For small datasets, the network overhead associated with data shuffling in cluster mode might outweigh the benefits of distributed processing. Larger datasets would benefit more from this aspect of cluster execution.

Cost vs. Performance Trade-off

The cost of running a cluster, particularly in cloud-based environments, needs to be justified by the performance gains. In our case, the small dataset did not fully leverage the advantages of distributed processing, making the single-node approach more cost-effective.

In conclusion, while the cluster mode in Spark offers scalability, improved resource utilization, and fault tolerance, its advantages become more evident with larger datasets and more complex processing tasks. For smaller datasets, as in our study, a single-node setup can offer comparable performance with reduced complexity and cost.