Lab 02: Advanced MapReduce & Spark Structured APIs

April 2025

Table of Contents

1 Team's Contribution

StudentID	Member	Task	Contribution
22120210	Lê Võ Nhật Minh	Exercise 2.1	Reducer Logic to export to output file
		Report	Template Preparation
		Exercise 2.2	Write another program to verify the output
22120227	Nguyễn Hữu Nghĩa	Exercise 2.2	Weekyly Report for each SKU Logic
		Report	Write Exercise 2.2 Report
22120262	Nguyễn Lê Tấn Phát	Exercise 2.1	Mapper Sliding Window Logic
		Report	Write Exercise 2.1 Report
		Exercise 2.3	Write another program to verify the output
22120405	Nguyễn Thanh Tuấn	Exercise 2.3	Overlap detection logic
		Report	Write Exercise 2.3 Report

2 Calculate revenue in the last 3 days for each country

2.1 Mapper

The `mapper_parse` function in the `MRRevenueCalculating` class is responsible for parsing and preprocessing transaction data from the input CSV file. Its goal is to extract revenue information corresponding to each product category over a three-day period starting from the transaction date.

2.1.1 Input Format

Each line of input data is in CSV format, in which the important fields include:

- **Transaction date** (3rd column - index 2)
- **Transaction status** (4th column - index 3)
- **Category** (10th column - index 9)
- **Transaction amount** (16th column - index 15)

2.1.2 Mapper Processing Flow

The mapper operates as follows:

1. Parse the input line using `csv.reader`. If the line is a header (i.e., the first column is "index"), it will be skipped.
2. Check the transaction status. If the transaction is cancelled (`"cancelled"`) or pending (`"pending"`), the line will be ignored. Because with `"cancelled"`, the transaction has been cancelled by the buyer. With `"pending"` it mean the transaction is still waiting for processing, therefore it should also be excluded to avoid counting future or incomplete revenue.
3. Attempt to convert the date string into a `datetime` object, supporting two formats: `%m-%d-%y` and `%m/%d/%Y`. This is because the `Transaction Date` column contains both formats, such as `4-30-22 (%m-%d-%y)` – line 2, and `4/12/2022 (%m/%d/%Y)` – line 29927.
4. If the date is valid, the mapper will emit three output key-value pairs corresponding to:
   - Transaction date
   - The next day
   - The third day (after transaction)

   Each key-value pair has the format `(date, category)` and is mapped to the transaction revenue.

2.2 Reducer

In this program, Reducer includes two phases. Let's take a closer look at how each reducer works.

2.2.1 First reducer processing flow

The `reducer_revenue` function operates as these following steps:

1. Receiving key-value pair input (which is the output from `mapper_parse`).
2. Then, calculate total revenue for each corresponding `(date,category)`.

3. Upon completion, aggregate all the data row into a single data row by yielding with `key = None` and `value = (date,category,total_revenue)`.

Reasoning: This reducer performs essential local aggregation to ensure correctness in the next processing stage. Without summing up the revenue for each `(date,category)`, the following reducer would receive multiple fragmented values per key, making global sorting and selection impossible. Therefore, this step not only reduces intermediate data size but also guarantees data integrity for downstream processing.

2.2.2 Second reducer processing flow

The workflow of `reducer_last_3` is shown as below:

1. First, it receives a single key-value pair from `reducer_revenue` and converts `date_category_revenue_pairs` into a list.
2. Next, it converts the date-time values in `date_category_revenue_pairs` into a `datetime` format to facilitate comparison and sorting.
3. Then, it sorts the list in ascending order by date and category name (case-insensitive).
4. Finally, it sets `OUTPUT_PROTOCOL = TextValueProtocol` [1] inside the `MRRevenueCalculating` class to modify how MRJob exports data to a `.csv` file. The output is exported by yielding `key = None` and `value = (date,category,revenue)`.

Reasoning: Since the output from the first reducer contains aggregated values for all `(date,category)` pairs, this reducer is designed to sort and filter the relevant entries. The sorting step ensures the correct order as the requirements. Changing the output protocol allows for a cleaner, more usable CSV export that aligns with post-processing or reporting requirements.

3 Calculate the number of products sold in the last 7 days of each SKU, report every Monday

3.1 Detailed Description of the Solution

This task aims to calculate the total number of products (by SKU) sold in the last 7 days, reported every Monday.

Steps Taken in the Code:

1. **Read and Preprocess the Data:**
   - Read `asr.csv` using Spark with headers enabled.
   - Filter only the rows where the `Status` column contains the string `"shipped"` (case-insensitive).
   - Convert the `"Date"` column to Spark `DateType` using the format `"MM-dd-yy"`.
   - Cast the `"Qty"` column to `Integer` for aggregation.
2. **Date Range Calculation:**
   - Determine the minimum and maximum dates from the dataset to define the time window.
   - Add 7 days to the maximum date to ensure inclusion of a full week in case the data ends mid-week.
3. **Generate Mondays for Weekly Reports:**

- Compute the first Monday on or after the minimum date.
- Use Spark's `sequence()` function with `INTERVAL 7 DAYS` to generate a sequence of Mondays from `first_monday` to `end_date`.

4. **Join with Sales Data:**
   - For each Monday (`report_date`), join with the sales data to collect records that happened in the 7-day window before that Monday (excluding the Monday itself).
   - This uses the condition:
     `(sale.Date < report_date) AND (sale.Date >= report_date - 7 days)`
5. **Aggregate Sales Data:**
   - Group by both `report_date` and `SKU`.
   - Aggregate the total quantity sold (`Qty`) over the 7-day window.
   - Sort the output by `report_date` and `sku`.
6. **Output:**
   - Write the result to a CSV file with headers and overwrite mode.

3.2 Reasoning Behind the Solution

- **Why generate a sequence of Mondays instead of relying on existing dates in the dataset:**
  The dataset may not contain a record for every Monday. Generating a complete sequence ensures that every potential weekly report is produced, even for weeks with no sales.
- **Why join using date conditions:**
  Instead of using `window()` functions (which may be more concise), the solution uses manual date-based joins to maintain transparency and control over the 7-day window, ensuring only the 7 days prior to each Monday are counted.
- **Why add 7 days to the max date:**
  Adding 7 days ensures that the generated sequence of Mondays includes the final week. This guarantees that no weekly report is missed near the end of the dataset.

4 Find pairs of overlapping shapes

The task aims to detect two overlapped quadrilaterals using the separated axis theorem (SAT) idea for the detection of overlap. The task is resolved in the following steps:

1. **Read data:** Read `shapes.parquet` file storing in HDFS using Spark.
2. **Process input data:** As the points given are in arbitrary order, sort the points in the following order: top left, top right, bottom right, and bottom left. This ensures that two adjacent elements form a convex edge.
   - Sort all four points along the y-axis in descending order. The first two points are the top points and the remaining points are the bottom points.
   - Sort the top points along the x-axis in descending order. Sort the bottom points along the x-axis otherwise.

3. **Self join** Using `crossJoin` with the `Dataframe` itself to create a pair of quadrilaterals for checking. To avoid duplication as the task only needs **sets** of quadrilaterals, use the condition: `col("x.shape_id") < col("y.shape_id")` to both ensure uniqueness and ascending order of `id`.

4. **Overlap detection**

   - In overall, according to SAT, two closed convex sets are disjoint if and only if there exists a hyperplane between the two [2]. The idea is to iterate all edges of the two quadrilaterals and see if there is a line separating the two quadrilaterals. If such a line exists, two quadrilaterals do not overlap. Otherwise, they do.
   - There is a property that if there exists a line separating two quadrilaterals, one of the edges is such a line [3]. Hence, iterating all of the available edges is sufficient.
   - Implementation is carried out as follows:
     - For each row after self join step, the algorithm receives two quadrilaterals. There are eight edges in total so the number of iterations is eight times.
     - For each iteration, the algorithm takes two adjacent points of one quadrilateral to form an edge (this is already ensured as a convex edge in the sorted step).
     - Construct the line function of Ax + By + C with the information of two points. Calculate the perpendicular vector of the vector connect two current points and find the A, B and C parameters for the function.
     - To check if the current line is the separating line, calculate all four points of one quadrilateral on the Ax + By + C function and then all four points of the other. By using the line function, we test if all four points of a quadrilateral all gives negative value or positive values. We mark `1` as the four points give all positive values, `-1` as the four points give all negative values, and `0` as the four points give mixed values. Note that if a point gives the value of 0, we skip it as it is not important. If one quadrilateral gives `1` and one quadrilateral gives `-1`, we say that the two quadrilateral do not overlapped. Otherwise, they do.

5. **Format and write the result**

   - Format the output result from the format *Shape_ {id}* to *{id}* only.
   - Write the result to HDFS.

Initially, the team aimed to solve the problem in the general case (detecting overlaps among polygons). However, after reviewing the input data, the team discovered that the inputs consist only of upright rectangles, and the assignment is limited to the use of Structured APIs. Therefore, the team developed an additional algorithm that applies Structured APIs to this specific case.

1. **Read data:** Read `shapes.parquet` file storing in HDFS using Spark.
2. **Overlap detection:**
   - Computes the bounding box for each shape based on its vertices: extracts all x and y coordinates from each shape's vertices, then calculates the bounding box (minimum and maximum x/y values) for every shape.
   - Perform a self cross-join to generate shape pairs: We do this by perform a self cross-join on the bounding box DataFrame and filters out duplicate and self-pairs by keeping only pairs where a.shape_id < b.shape_id.
   - Check for overlapping bounding box: We filters the shape pairs to keep only those whose bounding boxes overlap, based on the axis-aligned rectangle overlap condition.
   - Choose the overlapping pair: This step selects the overlapping shape pairs, removes the "Shape_" prefix from their IDs, converts them to integers, and sorts the results by shape ID.

References

[1] MRJob Team. *Protocols — mrjob v0.7.4 documentation.* Accessed: 2025-04-06. 2024. URL: `https://mrjob.readthedocs.io/en/latest/protocols.html#module-mrjob.protocol`.

[2] Manuel R. Lugo. *Separating Axis Theorem.* Accessed: 2025-04-10. n.d. URL: `https://personal.math.vt.edu/mrlugo/sat.html`.

[3] StackExchange user. *Proof of Separating Axis Theorem for Polygons.* Accessed: 2025-04-10. 2016. URL: `https://math.stackexchange.com/questions/2106402/proof-of-separating-axis-theorem-for-polygons`.