Big Data - Parking data analysis

David Hartl, Manzo Vincenzo

November 6, 2025

1 Assignment

1.1 Task 1

Task 1 required to develop a Python program, that streams parking information every minute and display the parking code and available spaces.

1.2 Task 2

The second Task was to extend the Program, with additional processing, which are defined by the student (Example: display only parking facilities that have changed in the last 5 minutes, display a graph showing the evolution of the number of available parking spaces, etc.)

For this project, we chose to implement the following additional processing features:

- A change-detection filter that prints a message to the console only for parking facilities that have reported a new number of available spaces.
- A real-time line graph, saved to an interactive HTML file, that visualizes the evolution of free spaces for all parking facilities over the last 5 minutes.

2 Solution

2.1 Task 1

Before the downloader is initialized, the program first cleans the input directory by removing any existing old files in the `data/input/` folder. After that, the `downloader` function is started in a background thread. This thread requests the parking data every minute. If the download is successful (no timeout and a 200 status code), the data is first saved to a temporary file (e.g., `tmp_parking_12345.csv`). This file is then immediately renamed to its final, timestamped name (e.g., `parking_20251106_214900.csv`). This two-step process ensures that Spark's streaming query only sees a complete, fully-written file, which prevents processing errors.

In the main thread, the program has two streams:

- **streamDF**: This is the input stream that reads the CSV data. It is configured as follows:
  - `option("header", True)`: Specifies that the CSV files contain a header row.
  - `schema(CSV_SCHEMA)`: Applies the predefined schema to the incoming data.
  - `csv(INPUT_DIR)`: Sets the monitored directory to `data/input/`.
  - `select("id", "libres")`: Selects only the `id` and `libres` columns for processing.
- **query_console**: The output stream, uses `streamDF` as input and writes the data to the console. The configuration looks as follows:
  - `format("console")`: Output format is console.
  - `outputMode("append")`: New rows are appended to the result table.
  - `Option("truncate", "false")`: Do not truncate the output.
  - `trigger(processingTime=f"PERIOD seconds")`: Process the data every 60 seconds.

This final `try...except...finally` block manages the application's lifecycle by using `spark.streams.awaitAnyTerminatio` to keep the main thread alive, catching a `KeyboardInterrupt` (like `Ctrl+C`) to exit, and then executing the `finally` block to guarantee a clean shutdown of both streaming queries and the Spark session.

2.2 Task 2

The optional second task is fulfilled by a second, parallel `writeStream` which processes the same `streamDF` input. This stream is responsible for the custom analysis and visualization.

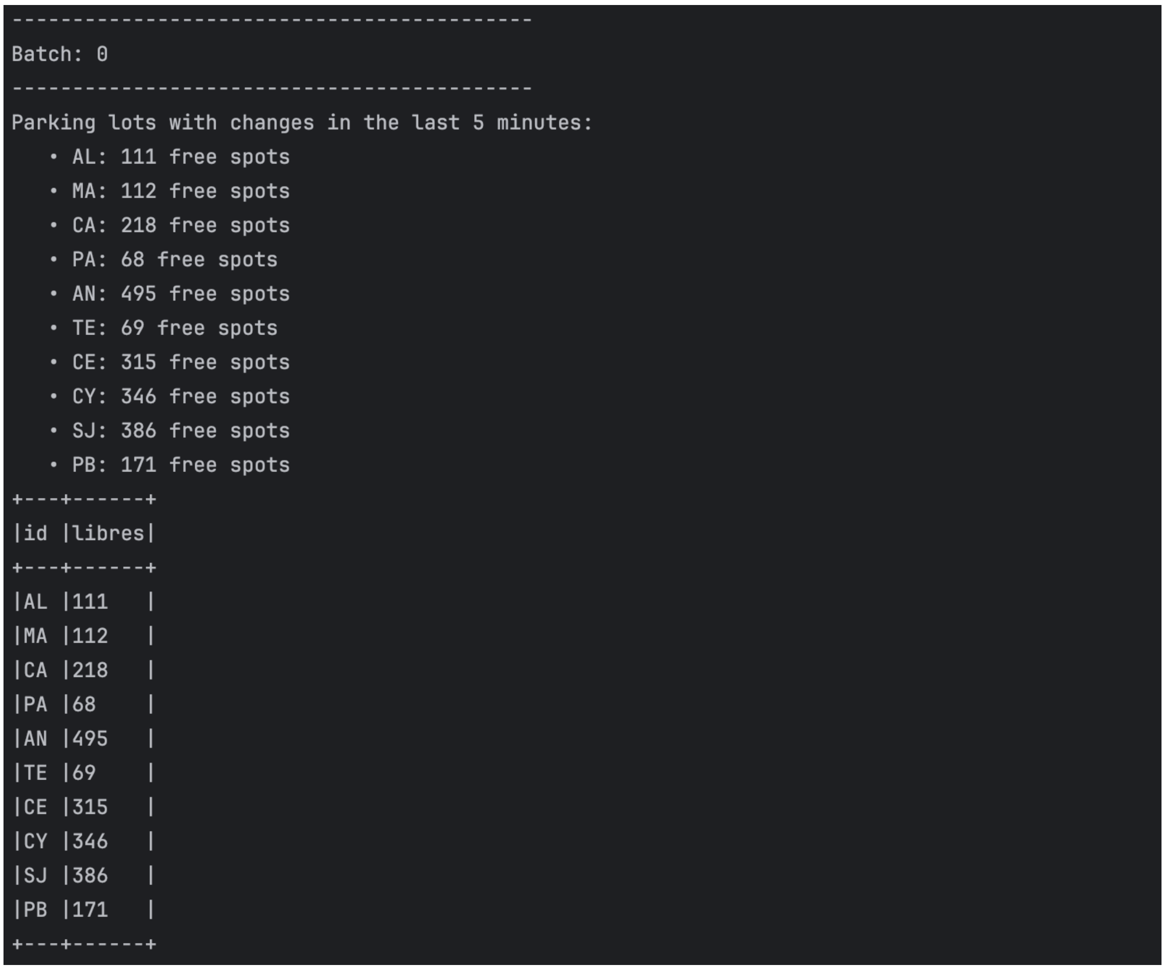
- **query_analysis**: This output stream uses `streamDF` as input and sends each micro-batch to a custom processing function. The configuration is as follows:
  - `foreachBatch(analyze_batch)`: Instead of writing to the console, this stream's format sends the entire batch DataFrame (and its batch ID) to the `analyze_batch` function.
  - `outputMode("append")`: New rows are processed.
  - `trigger(processingTime=f"PERIOD seconds")`: Process the data every 60 seconds, just like the console stream.

The `analyze_batch` function contains the two features for the optional task:

1. **Change Detection:** A global dictionary, `last_updates`, is used to store the last known "libres" value for each parking "id". For each row in the new batch, it compares the current value with the stored value. If the value is new or the "id" has not been seen before, it is added to a `changed` list and printed to the console.
2. **Evolution Graph:** A global list, `history`, is used to store all incoming data points as tuples containing the current timestamp, parking "id", and "libres" count.
   - At the beginning of each batch, new data is appended to this `history` list.
   - A `cutoff` variable is calculated by subtracting 5 minutes from the current time.
   - The `history` list is then filtered to remove any entries older than this 5-minute cutoff.
   - Finally, the `plot_history_plotty()` function is called.

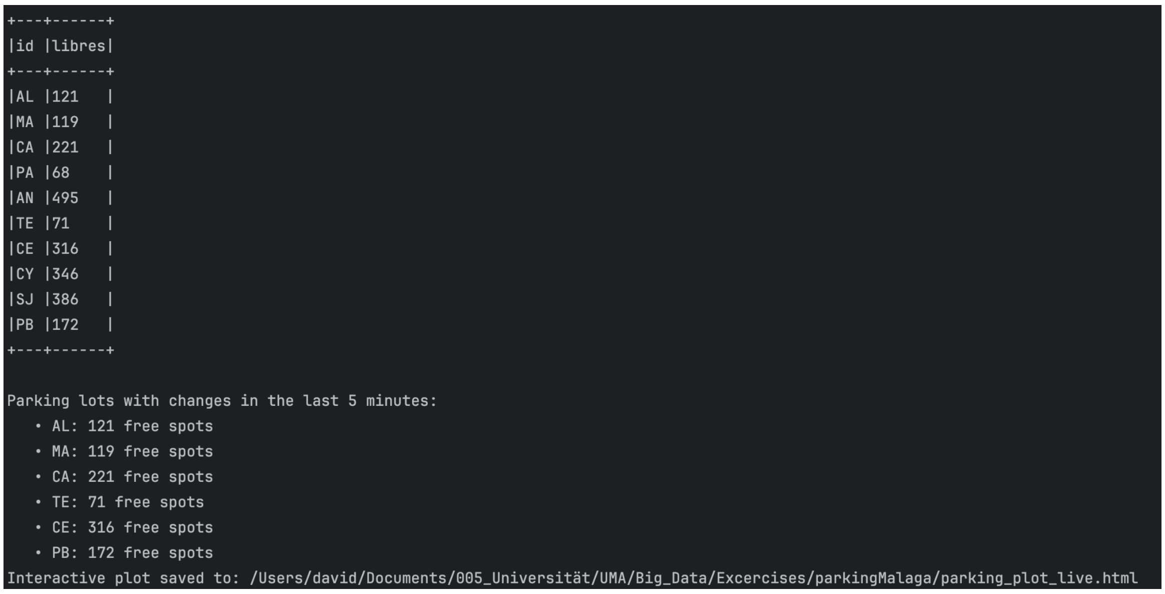
3 Execution Screenshots

To demonstrate the program, we ran the `StreamingParkingMalaga.py` script in a terminal. The following screenshots show the program in action.



```
-------------------------------------------
Batch: 0
-------------------------------------------
Parking lots with changes in the last 5 minutes:
   • AL: 111 free spots
   • MA: 112 free spots
   • CA: 218 free spots
   • PA: 68 free spots
   • AN: 495 free spots
   • TE: 69 free spots
   • CE: 315 free spots
   • CY: 346 free spots
   • SJ: 386 free spots
   • PB: 171 free spots
+---+------+
|id |libres|
+---+------+
|AL |111   |
|MA |112   |
|CA |218   |
|PA |68    |
|AN |495   |
|TE |69    |
|CE |315   |
|CY |346   |
|SJ |386   |
|PB |171   |
+---+------+
```

Figure 1: Live console output showing both streaming queries.



```
+---+------+
|id |libres|
+---+------+
|AL |121   |
|MA |119   |
|CA |221   |
|PA |68    |
|AN |495   |
|TE |71    |
|CE |316   |
|CY |346   |
|SJ |386   |
|PB |172   |
+---+------+

Parking lots with changes in the last 5 minutes:
   • AL: 121 free spots
   • MA: 119 free spots
   • CA: 221 free spots
   • TE: 71 free spots
   • CE: 316 free spots
   • PB: 172 free spots
Interactive plot saved to: /Users/david/Documents/005_Universität/UMA/Big_Data/Excercises/parkingMalaga/parking_plot_live.html
```

Figure 2: Console output showing parking facilities with changed free spaces in the last minute.

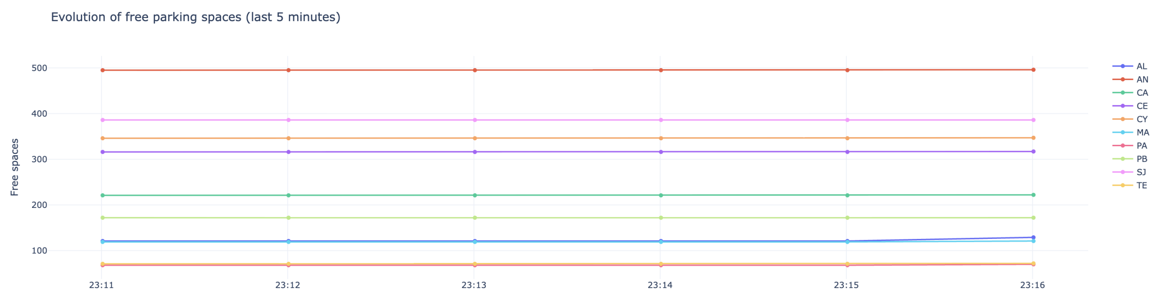


Figure 3: The generated `parking_plot_live.html` showing the 5-minute evolution of free spaces.