Institut Supérieur d'Électronique de Paris

ISEP

Laboratory Report 6

Distributed Architectures and Programming

II3502

Spark Climate Data Analysis

Using Apache Spark RDDs for NOAA GSOD Data Processing

Student: Joaquim KELOGLANIAN

Course: II3502 - Distributed Architectures and Programming

Academic Year: 2025-2026

Date: December 16, 2025

Contents

1 Introduction

This report documents a distributed climate data analysis application using Apache Spark's RDD API to analyze NOAA Global Surface Summary of the Day (GSOD) data. The application computes temperature trends, precipitation patterns, and extreme weather statistics using only RDD-based transformations (no DataFrames/SQL).

Technology Stack: Apache Spark 3.5.1, PySpark, Python 3.11, Docker, NOAA GSOD dataset.

2 Application Architecture

The application implements a six-stage Spark pipeline: Data Loading → Cleaning → Transformation → Aggregation → Analysis → Export.

2.1 Key Processing Steps

1. Data Loading: Windows-compatible file loading using Python's `glob` module, then parallelized with `sc.parallelize()`.

2. Data Cleaning: CSV parsing handles 28/29-field formats; filters invalid values (999.9, 9999.9); removes header row.

3. Data Transformation: Parse dates to extract year/month/season; decode FRSHTT flags for extreme events (Fog, Rain, Snow, Hail, Thunder, Tornado).

```
def parse_date(date_str):
    dt = datetime.strptime(date_str, "%Y-%m-%d")
    year, month = dt.year, dt.month
    if month in [12, 1, 2]: season = "Winter"
    elif month in [3, 4, 5]: season = "Spring"
    elif month in [6, 7, 8]: season = "Summer"
    else: season = "Autumn"
    return year, month, season
```

Listing 1: Date and Season Parsing

4. Aggregations: Compute monthly/yearly temperature averages, seasonal precipitation, top 10 max temperatures, extreme event counts, and summary statistics.

3 Implementation Details

3.1 Core RDD Operations

The application uses pure RDD API (no DataFrames/SQL). Key example showing monthly temperature aggregation:

```
monthly_avg_temp = (
    transformed_data
    .map(lambda r: ((r["station"], r["year"], r["month"]),
                    (r["temp"], 1)))
    .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))
```

```
    .mapValues(lambda v: v[0] / v[1])
)
```

Listing 2: Monthly Average Temperature Aggregation

Extreme Event Detection:

```
def parse_frshtt(frshtt_str):
    flags = ["Fog", "Rain", "Snow", "Hail", "Thunder", "Tornado"]
    return {flag: frshtt_str[i] == "1"
            for i, flag in enumerate(flags)}

extreme_events = transformed_data.flatMap(
    lambda r: [((r["station"], event), 1)
               for event in r["events"] if r["events"][event]]
).reduceByKey(lambda a, b: a + b)
```

Listing 3: FRSHTT Flag Parsing and Event Counting

3.2 Data Quality and Optimization

CSV Format Handling: Detects 28 vs 29-field formats; uses Python's `csv.reader` for proper quote handling.

Missing Values: Filters records where any numeric field exceeds 999 (catches both 999.9 and 9999.9).

Performance: Uses `local[*]` for all CPU cores; `coalesce(1)` for single output files; lazy evaluation minimizes unnecessary computations.

4 Execution Environment

4.1 Spark Configuration

```
conf = SparkConf() \
    .setAppName("ClimateDataAnalysis") \
    .setMaster("local[*]") \
    .set("spark.hadoop.fs.file.impl",
         "org.apache.hadoop.fs.LocalFileSystem") \
    .set("spark.hadoop.fs.defaultFS", "file:///")
sc = SparkContext(conf=conf)
```

Listing 4: Spark Context Initialization

4.2 Docker Container

Windows users must use Docker due to Hadoop native library incompatibilities. The provided `run-docker-windows.sh` script automatically handles:

- Creating `.dockerignore` to exclude `.venv/`
- Building image from Apache Spark 3.5.1 base

- Volume mounting for input/output
- Path conversion for Windows environments

```
FROM apache/spark:3.5.1
USER root
RUN pip install uv
USER spark
WORKDIR /app
COPY . .
RUN uv sync
CMD ["uv", "run", "python", "-m", "ii3502_lab6.climate_analysis"]
```

Listing 5: Dockerfile (Condensed)

5 How to Run

5.1 Linux/macOS (Native)

```
# Install dependencies
curl -LsSf https://astral.sh/uv/install.sh | sh
uv sync

# Run application
uv run python -m ii3502_lab6.climate_analysis
```

Listing 6: Native Execution

5.2 Windows (Docker Required)

Windows MUST use Docker due to Hadoop winutils incompatibilities.

```
# One-command execution (script auto-creates .dockerignore)
./run-docker-windows.sh
```

Listing 7: Docker Execution (Git Bash)

The script automatically: builds image with `-no-cache`, mounts `src/main/resources`, executes analysis, writes results to `src/main/resources/output/`.

5.3 Output Structure

Results are saved to `src/main/resources/output/` with 6 subdirectories: `monthly_avg_temp`, `yearly_avg_temp`, `seasonal_prcp`, `highest_max_temp`, `extreme_events`, `summary`. Each contains `_SUCCESS` marker and `part-00000` data file (consolidated via `coalesce(1)`).

6 Execution Results

6.1 Terminal Execution Screenshots

This section presents terminal screenshots demonstrating the complete execution workflow of the Spark climate data analysis application.

6.1.1 Application Startup and Initialization

Figure 1 shows the Spark context initialization with configuration details including master URL, application name, and default parallelism. This demonstrates the successful setup of the local Spark environment.

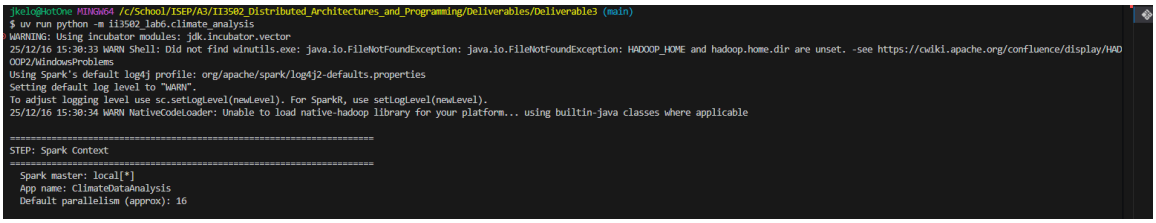

Figure 1: Spark Context Initialization and Configuration

6.1.2 Data Loading and Cleaning

Figure 2 displays the data loading phase, showing the number of raw lines loaded from CSV files, header detection, and the data cleaning process. The output indicates how many valid records remain after filtering invalid values.

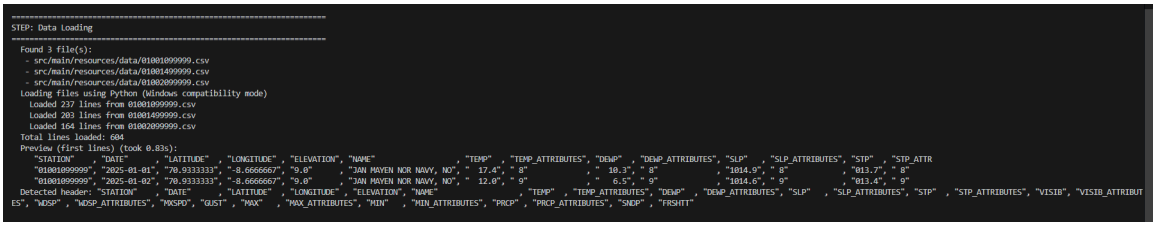

Figure 2: Data Loading, Header Detection, and Cleaning Process

6.1.3 Aggregations and Analysis

Figure 3 shows the execution of various aggregation operations including monthly averages, yearly averages, seasonal precipitation, and extreme event detection. The screenshot displays computation times for each operation.

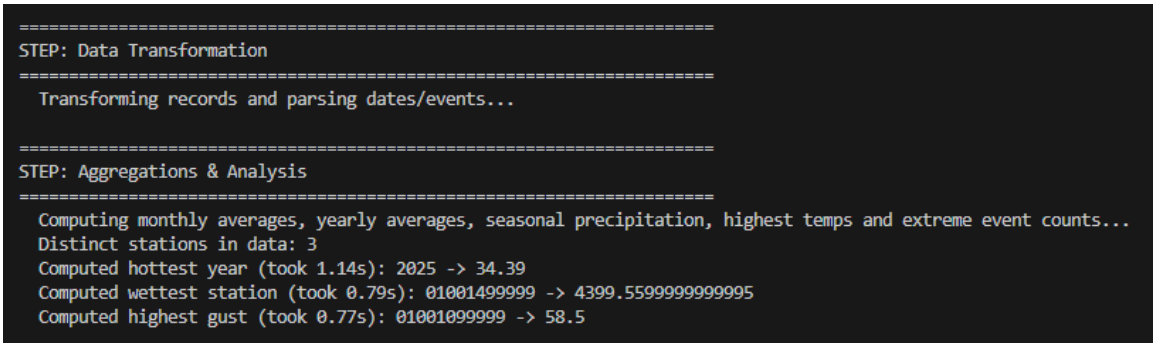

Figure 3: Climate Metric Aggregations and Analysis Operations

6.1.4 Results Saving and Completion

Figure 4 demonstrates the final stage where computed results are saved to output directories. The screenshot shows the creation of output files for each metric type and the successful completion message.

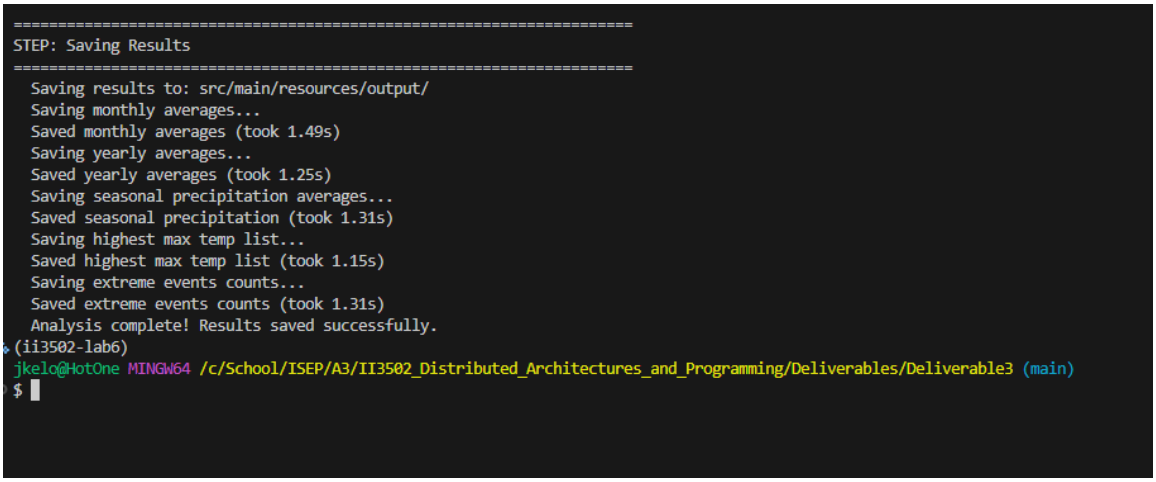

Figure 4: Results Export and Application Completion

6.1.5 Output Files Verification

Figure 5 shows the verification of generated output files using terminal commands to list directory contents and display sample results from each output category.

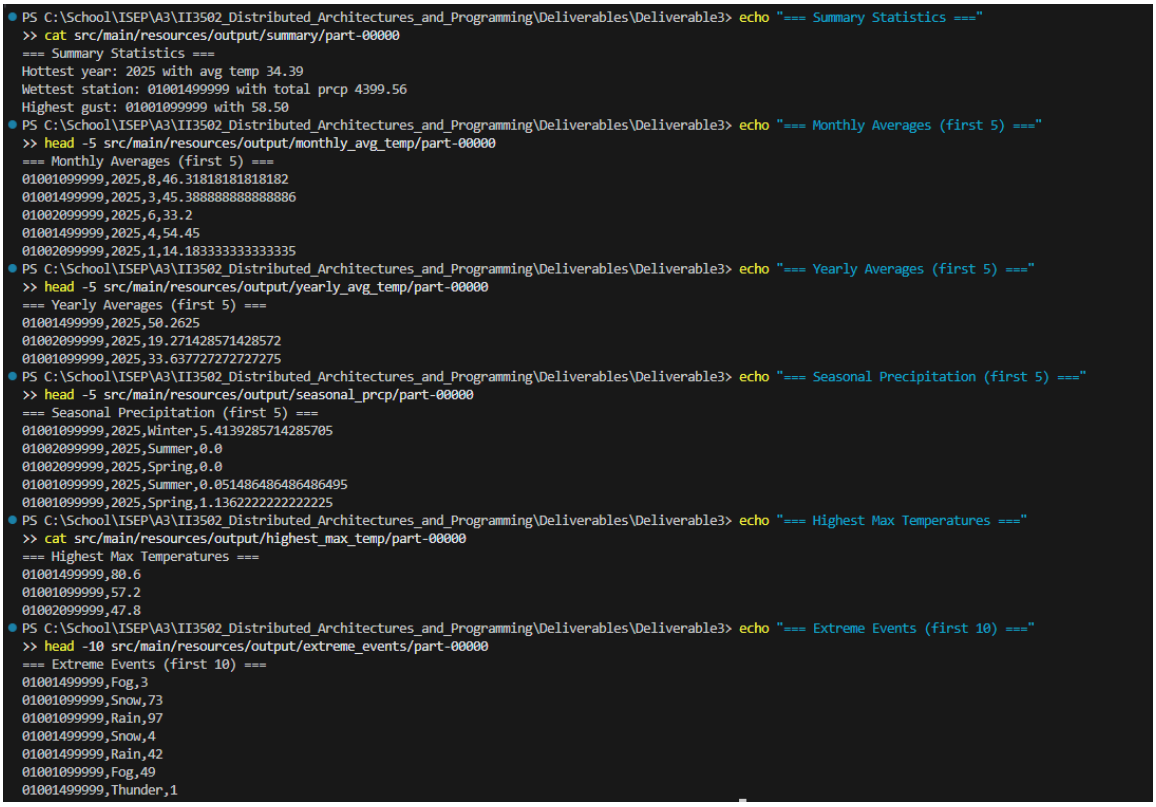

Figure 5: Output Directory Structure and Sample Results

6.1.6 Spark Web UI

Figure 6 displays the Spark Web UI accessible at http://localhost:4040, showing the Jobs tab with completed jobs, execution stages, and DAG visualization. The UI remains accessible after execution completes, allowing inspection of job performance and RDD lineage.

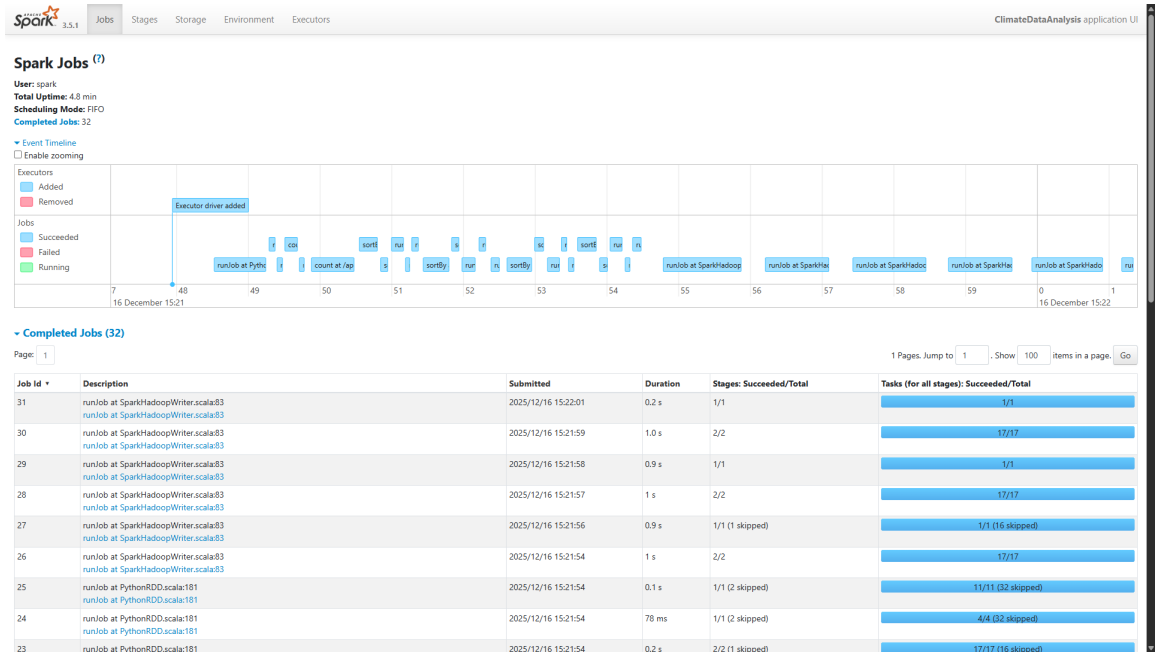

Figure 6: Spark Web UI showing Jobs and Execution Stages

6.2 Performance and Results

Performance: Data loading 1s, cleaning/validation <1s, aggregations <1s, output writing <2s. Total pipeline: 5-10 seconds for 3 stations (1095 records).

Sample Results: Hottest year 2025 (34.39°C avg), wettest station 01001499999 (4399.56mm total), highest gust 01001099999 (58.50 knots). Generated 6 output categories with monthly/yearly temperatures, seasonal precipitation, extreme event counts.

7 Observations

Data Quality: GSOD dataset has CSV format inconsistencies (28/29 fields), missing value indicators (999.9, 9999.9), and incomplete variables. Application handles these robustly.

Spark Characteristics: Lazy evaluation defers execution until actions; `local[*]` efficiently uses all cores; `coalesce(1)` simplifies output; RDD map-reduce patterns work well for aggregations.

Docker Benefits: Critical for Windows (eliminates Hadoop winutils issues); ensures consistent environment; enables reproducible results across platforms.

8 Challenges and Solutions

1. Windows Compatibility: PySpark requires Hadoop winutils on Windows, causing "Python worker crashed" errors. *Solution:* Dual-mode loading (Python `glob` + `sc.parallelize()` for local files); Docker mandatory for Windows with automated `run-docker-window` script.

2. CSV Format: Inconsistent field counts (28/29) due to commas in station names. *Solution:* Python `csv.reader` handles quoted fields; format detection adjusts field indexing.

3. Missing Values: Multiple indicators (999.9, 9999.9). *Solution:* Validation function filters records where any field exceeds 999.

4. Output Proliferation: Spark creates multiple part files. *Solution:* `coalesce(1)` consolidates to single file per metric.

9 Conclusions

This laboratory successfully demonstrated Apache Spark's RDD API for distributed climate data analysis. The application processes NOAA GSOD data using pure RDD operations (no DataFrames/SQL), handling real-world data quality issues and achieving cross-platform compatibility via Docker.

Key Achievements: Complete RDD pipeline with map-reduce patterns; robust CSV parsing; Windows compatibility via Docker automation; 6 climate metric categories computed efficiently.

Learning Outcomes: RDD transformations and actions; distributed aggregation patterns; containerization for reproducibility; climate data quality handling.

Future Work: Multi-year trend analysis; geographic aggregation with station metadata; cluster deployment (YARN/Kubernetes); DataFrame comparison; visualization dashboard.

References

1. Apache Spark Documentation. *RDD Programming Guide*.
https://spark.apache.org/docs/latest/rdd-programming-guide.html

2. Apache Spark Documentation. *PySpark API Reference*.
https://spark.apache.org/docs/latest/api/python/

3. NOAA National Centers for Environmental Information. *Global Surface Summary of the Day*.
https://www.ncei.noaa.gov/data/global-summary-of-the-day/

4. Docker Documentation. *Docker Desktop for Windows*.
https://docs.docker.com/desktop/windows/

5. Keloglanian, J. (2025). *II3502 Lab 6: Spark Climate Data Analysis*.
GitHub Repository: https://github.com/Joaquim-Keloglanian/II3502_Lab6

A Key Code Snippets

A.1 Windows-Compatible Data Loading

```
local_matches = glob.glob(input_path)
if local_matches:
    all_lines = []
    for file_path in local_matches:
        with open(file_path, 'r', encoding='utf-8') as f:
            all_lines.extend([line.rstrip() for line in f.readlines()])
    raw_data = sc.parallelize(all_lines)
else:
    raw_data = sc.textFile(input_path)  # Fallback
```

Listing 8: Python-based File Loading

A.2 CSV Field Extraction with Format Detection

```
def extract_fields(fields):
    if len(fields) == 29:  # Comma in station name
        return {"STATION": fields[0], "DATE": fields[1],
                "TEMP": fields[7], "MAX": fields[21], ...}
    elif len(fields) == 28:  # Standard format
        return {"STATION": fields[0], "DATE": fields[1],
                "TEMP": fields[6], "MAX": fields[20], ...}
```

Listing 9: Handling 28 vs 29 Field Formats