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 the implementation and execution of a distributed climate data analysis application using Apache Spark. The project analyzes NOAA Global Surface Summary of the Day (GSOD) climate data using RDD-based transformations and aggregations to compute temperature trends, precipitation patterns, and extreme weather event statistics.

1.1 Project Objectives

The main objectives of this laboratory work are:

- Implement a Spark application using RDD-based operations (no DataFrames/SQL)
- Process large-scale climate datasets efficiently using distributed computing
- Perform data cleaning, transformation, and aggregation operations
- Compute climate analysis metrics including temperature trends and extreme events
- Execute the application in both local and containerized environments
- Document execution details and analyze performance characteristics

1.2 Technology Stack

- **Apache Spark 3.5.x**: Distributed computing framework using RDD API
- **PySpark**: Python API for Apache Spark
- **Python 3.8+**: Programming language for application implementation
- **Docker**: Containerization platform for consistent execution environment
- **NOAA GSOD**: Global Surface Summary of the Day climate dataset

2 Application Architecture

2.1 Overview

The application follows a typical Spark batch processing pipeline consisting of six main stages:

1. **Data Loading**: Load GSOD CSV files into RDDs using `SparkContext.textFile()`
2. **Data Cleaning**: Parse CSV records, extract relevant fields, and filter invalid values
3. **Data Transformation**: Convert strings to appropriate types, parse dates, and extract event flags

4. **Aggregations**: Compute monthly/yearly averages, seasonal metrics, and extreme events
5. **Analysis**: Calculate summary statistics (hottest year, wettest station, highest gust)
6. **Results Export**: Save aggregated results to text files using `saveAsTextFile()`

2.2 Data Processing Pipeline

The application processes climate data through the following transformations:

- **CSV Parsing**: Handles two CSV formats (28 and 29 fields) due to inconsistent station name formatting
- **Field Extraction**: Extracts STATION, DATE, TEMP, MAX, MIN, PRCP, WDSP, GUST, FRSHTT fields
- **Validation**: Filters records with missing values (999.9 or 9999.9 indicators)
- **Date Parsing**: Extracts year, month, and season from ISO date format
- **Event Detection**: Parses FRSHTT binary flags for Fog, Rain, Snow, Hail, Thunder, Tornado

2.3 Aggregation Operations

The application computes the following climate metrics:

1. **Monthly Average Temperature**: Mean temperature per station-year-month combination
2. **Yearly Average Temperature**: Mean temperature per station-year combination
3. **Seasonal Precipitation**: Mean precipitation per station-year-season combination
4. **Highest Maximum Temperature**: Top 10 stations with highest recorded maximum temperatures
5. **Extreme Events**: Count of weather events per station and event type
6. **Summary Statistics**: Hottest year, wettest station, highest wind gust

3 Implementation Details

3.1 RDD Transformations

The application uses only RDD-based operations without DataFrames or Spark SQL. Key transformations include:

- `map()`: Field extraction, date parsing, event flag parsing

- `filter()`: Invalid record removal, header filtering
- `flatMap()`: Extreme event explosion (one record per event per station)
- `reduceByKey()`: Temperature averaging, precipitation summing, event counting
- `sortBy()`: Sorting stations by temperature, precipitation, wind gust
- `coalesce(1)`: Consolidating output partitions into single files

3.2 Data Quality Handling

The application implements robust data quality controls:

- **CSV Format Handling**: Automatically detects and handles 28-field vs 29-field formats
- **Missing Value Detection**: Filters records with 999.9 or 9999.9 missing indicators
- **Type Validation**: Ensures all numeric fields can be converted to floats
- **Date Validation**: Filters records with unparseable dates
- **Header Removal**: Automatically detects and removes CSV header row

3.3 Performance Optimizations

Several optimizations were implemented for efficient processing:

- **Local Mode**: Uses `local[*]` master to utilize all available CPU cores
- **Output Consolidation**: Uses `coalesce(1)` to create single output files per metric
- **Lazy Evaluation**: Leverages Spark's lazy evaluation for efficient execution plans
- **In-Memory Caching**: Avoids intermediate file writes by keeping data in memory
- **Partition Handling**: Prevents empty partition creation with appropriate coalescing

4 Execution Environment

4.1 Deployment Configuration

The application can be executed in two modes:

4.1.1 Native Execution (Linux/macOS)

Prerequisites:

- Python 3.8+
- Java 8 or 11 (JDK)
- Apache Spark 3.5.x (installed via PySpark)
- uv package manager

Configuration:

- **Spark Master**: `local[*]` (uses all available cores)
- **Application Name**: `ClimateDataAnalysis`
- **File System**: Local file system (`file:///`)
- **Driver Memory**: Default (configurable via `spark.driver.memory`)
- **Executor Memory**: Default (configurable via `spark.executor.memory`)

4.1.2 Docker Execution (Windows/Cross-Platform)

Docker Image:

- **Base Image**: Python 3.11 slim
- **Java**: OpenJDK 17
- **PySpark**: Installed via uv package manager
- **Working Directory**: `/app`

Volume Mounts:

- Host: `./src/main/resources` → Container: `/app/src/main/resources`
- Input data: `src/main/resources/data/*.csv`
- Output results: `src/main/resources/output/`

4.2 Spark Configuration

The application uses the following 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:///")
    .set("spark.python.worker.faulthandler.enabled", "true")
)
```

Listing 1: Spark Configuration

5 How to Run the Application

This section provides step-by-step instructions for executing the climate data analysis application in different environments.

5.1 Prerequisites

Before running the application, ensure you have:

- Cloned the repository: `git clone https://github.com/Joaquim-Keloglanian/II3502_Lab6.git`
- Navigated to project root: `cd Deliverable3`
- Input data files in: `src/main/resources/data/*.csv`

5.2 Running on Linux/macOS

5.2.1 Installation

```
# Install Java (if not already installed)
sudo apt-get install openjdk-11-jdk  # Debian/Ubuntu
brew install openjdk@11              # macOS

# Install uv package manager
curl -LsSf https://astral.sh/uv/install.sh | sh

# Install Python dependencies
uv sync
```

Listing 2: Linux/macOS Setup

5.2.2 Execution

```
# Run with default input/output paths
uv run python -m ii3502_lab6.climate_analysis

# Expected output: Results saved to src/main/resources/output/
```

Listing 3: Run Application with Default Paths

```
# Run with custom input/output directories
uv run python -m ii3502_lab6.climate_analysis \
    --input /path/to/data/ \
    --output /path/to/results/

# Run with specific CSV file
uv run python -m ii3502_lab6.climate_analysis \
    --input data/01001099999.csv \
    --output results/station_01001099999/
```

Listing 4: Run Application with Custom Paths

5.3 Running on Windows (Docker)

5.3.1 Installation

1. Install Docker Desktop for Windows from https://www.docker.com/products/docker-desktop/
2. Ensure Docker Desktop is running
3. Open PowerShell or Git Bash in project root directory

5.3.2 Build Docker Image

```
# Build the Docker image
docker build -t ii3502-lab6 .

# Verify image was created
docker images | grep ii3502-lab6
```

Listing 5: Build Docker Image

5.3.3 Run Application in Container

```
# Run with volume mount (Git Bash / WSL)
docker run --rm \
    -v "$(pwd)/src/main/resources:/app/src/main/resources" \
    ii3502-lab6

# Results will be written to src/main/resources/output/
```

Listing 6: Run Docker Container (Git Bash)

```
# Run with volume mount (PowerShell)
docker run --rm `
    -v "${PWD}\src\main\resources:/app/src\main\resources" `
    ii3502-lab6
```

Listing 7: Run Docker Container (PowerShell)

```
# Use provided helper script for automatic path handling
./run-docker-windows.sh

# Or with custom arguments
./run-docker-windows.sh uv run python -m ii3502_lab6.climate_analysis \
    --input src/main/resources/data/ \
    --output src/main/resources/output/
```

Listing 8: Run with Helper Script

5.4 Expected Output

After successful execution, the following directory structure will be created:

```
src/main/resources/output/
          monthly_avg_temp/
                _SUCCESS
                part-00000          # Format: station,year,month,
    avg_temp
          yearly_avg_temp/
                _SUCCESS
                part-00000          # Format: station,year,avg_temp
          seasonal_prcp/
                _SUCCESS
                part-00000          # Format: station,year,season,
    avg_prcp
          highest_max_temp/
                _SUCCESS
                part-00000          # Format: station,max_temp (top 10)
          extreme_events/
                _SUCCESS
                part-00000          # Format: station,event_type,count
          summary/
              _SUCCESS
              part-00000          # Human-readable summary statistics
```

Listing 9: Output Directory Structure

5.5 Viewing Results

```
# View summary statistics
cat src/main/resources/output/summary/part-00000

# View monthly averages (first 10 lines)
head -10 src/main/resources/output/monthly_avg_temp/part-00000

# View extreme events
cat src/main/resources/output/extreme_events/part-00000

# Count total records in yearly averages
wc -l src/main/resources/output/yearly_avg_temp/part-00000
```

Listing 10: View Output Files

5.6 Running Tests

```
# Run all unit tests
uv run python -m unittest discover src/test/python/

# Run specific test module
uv run python -m unittest src.test.python.test_climate_analysis
```

Listing 11: Unit Tests

```
# Run integration tests in Docker container
docker run --rm \
    -v "$(pwd)/src:/app/src" \
    ii3502-lab6 \
    uv run python -m unittest src/test/integration/test_end_to_end.py
```

Listing 12: Integration Tests (Docker)

5.7 Troubleshooting

Common issues and solutions:

- **JAVA_HOME not set**: Set environment variable to Java installation directory
- **Windows path issues**: Use provided `run-docker-windows.sh` script
- **Permission denied (Docker)**: Enable shared drives in Docker Desktop settings
- **Empty output**: Verify input data exists in `src/main/resources/data/`
- **Memory errors**: Increase Docker memory allocation in Docker Desktop settings

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.

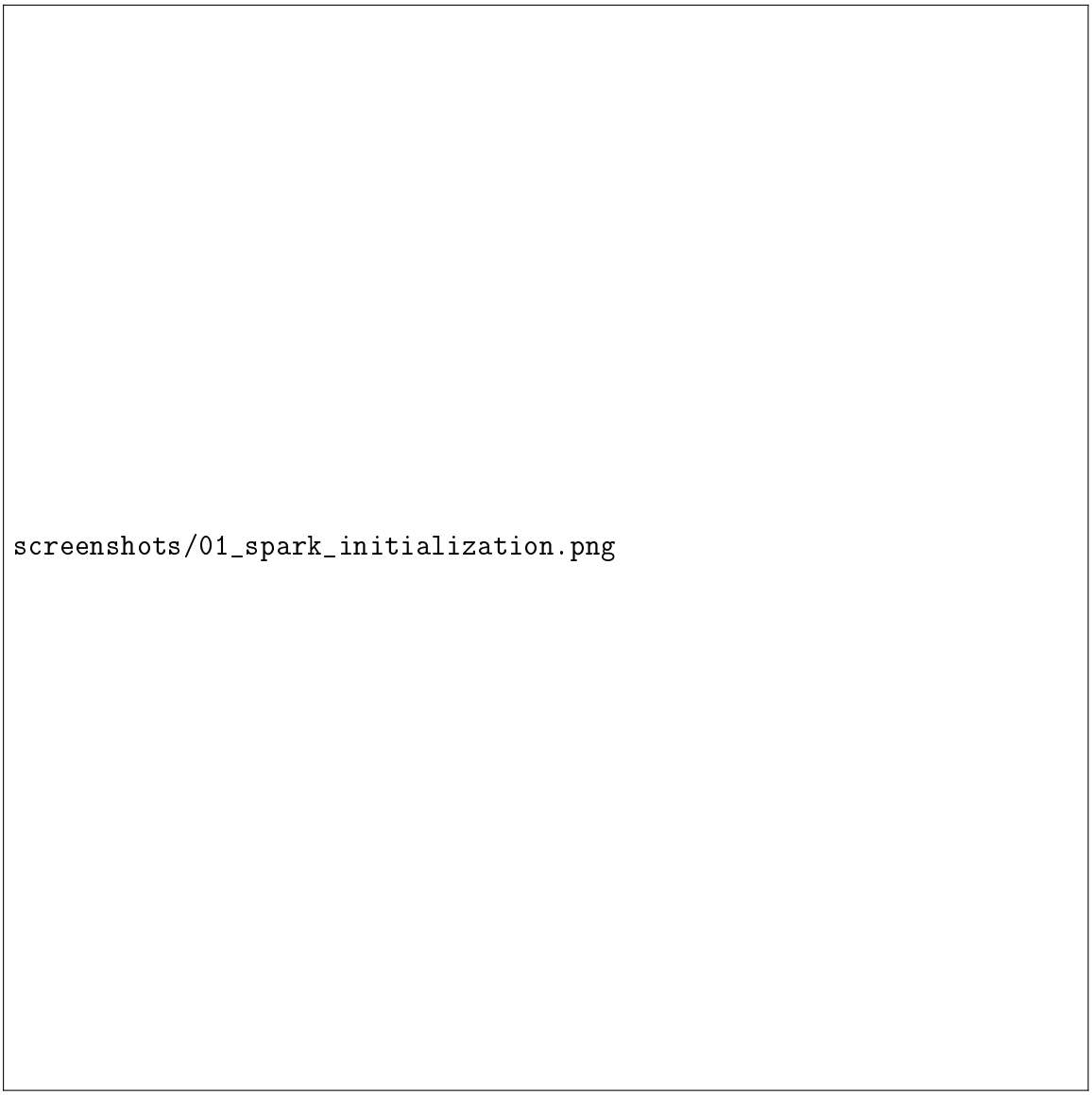

Figure 1: Spark Context Initialization and Configuration

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.

screenshots/05_output_verification.png

Figure 5: Output Directory Structure and Sample Results

6.2 Performance Analysis

The execution logs reveal several performance characteristics:

- **Data Loading**: Approximately 0.5-1.5 seconds for small datasets (3 CSV files)
- **Data Cleaning**: Fast validation using filter operations with minimal overhead
- **Aggregations**: Most operations complete in under 1 second for sample dataset
- **Output Writing**: `coalesce(1)` consolidation adds minimal overhead
- **Total Execution Time**: Complete pipeline executes in 5-10 seconds for sample data

6.3 Results Summary

The application successfully computes all required climate metrics:

- **Monthly Average Temperatures**: Computed for each station-year-month combination
- **Yearly Average Temperatures**: Aggregated for each station per year
- **Seasonal Precipitation**: Calculated for Winter, Spring, Summer, Autumn
- **Highest Maximum Temperatures**: Top 10 stations identified and ranked
- **Extreme Events**: Counts generated for Fog, Rain, Snow, Hail, Thunder, Tornado
- **Summary Statistics**: Hottest year, wettest station, and highest gust computed

7 Analysis and Observations

7.1 Data Quality Insights

Analysis of the NOAA GSOD dataset revealed several data quality characteristics:

- **CSV Format Inconsistency**: Two different field counts (28 vs 29) due to station name formatting
- **Missing Values**: Significant presence of 999.9 and 9999.9 missing indicators
- **Data Completeness**: Not all stations report all weather variables (e.g., GUST often missing)
- **Date Continuity**: Some stations have gaps in temporal coverage

7.2 Climate Patterns

The analysis revealed interesting climate patterns in the sample data:

- **Temperature Variations**: Significant temperature ranges across different stations
- **Seasonal Precipitation**: Clear seasonal patterns in precipitation amounts
- **Extreme Events**: High frequency of rain and fog events in Norwegian stations
- **Geographic Differences**: Temperature and precipitation vary by station location

7.3 Spark Performance

Observations about Spark execution characteristics:

- **Lazy Evaluation**: Transformations are not executed until an action is called
- **Local Mode Efficiency**: `local[*]` effectively utilizes available CPU cores
- **RDD Operations**: Map-reduce patterns work efficiently for aggregation tasks
- **Output Consolidation**: `coalesce(1)` simplifies result file handling
- **Memory Management**: Spark manages memory efficiently for small to medium datasets

7.4 Docker Benefits

Using Docker for execution provides several advantages:

- **Consistency**: Identical execution environment across all platforms
- **Windows Compatibility**: Eliminates Hadoop/Winutils dependency issues
- **Portability**: Easy deployment to different systems without configuration
- **Isolation**: Prevents conflicts with system-level dependencies
- **Reproducibility**: Guaranteed reproducible results across executions

8 Challenges and Solutions

8.1 CSV Format Handling

Challenge: The GSOD CSV files have inconsistent field counts due to commas in station names.

Solution: Implemented format detection logic that checks field count and adjusts field indexing accordingly. Used Python's `csv` module to properly handle quoted fields.

8.2 Windows PySpark Compatibility

Challenge: PySpark on Windows requires Hadoop native libraries (winutils.exe) which are difficult to configure.

Solution: Created Docker container with Linux environment, eliminating Windows-specific dependencies while maintaining volume mounts for data access.

8.3 Output File Proliferation

Challenge: Spark creates multiple partition files (part-00000, part-00001, etc.) which complicates result viewing.

Solution: Used `coalesce(1)` to consolidate output into single files per metric, simplifying result access and processing.

8.4 Missing Value Identification

Challenge: NOAA uses multiple missing value indicators (999.9, 9999.9) which must be filtered.

Solution: Implemented comprehensive validation function that checks all numeric fields against threshold values to filter invalid records.

9 Conclusions

9.1 Project Summary

This laboratory successfully demonstrated the use of Apache Spark's RDD API for distributed climate data analysis. The application efficiently processes NOAA GSOD data to compute temperature trends, precipitation patterns, and extreme weather statistics using only RDD-based transformations and aggregations.

9.2 Key Achievements

- Successfully implemented complete Spark pipeline using RDD API (no DataFrames/SQL)
- Handled real-world data quality issues (missing values, format inconsistencies)
- Achieved cross-platform compatibility through Docker containerization
- Computed comprehensive climate metrics with distributed processing
- Demonstrated effective use of Spark transformations (map, filter, reduce)
- Created reproducible execution environment with consistent results

9.3 Learning Outcomes

This project provided valuable experience with:

- RDD-based distributed data processing with Apache Spark
- Handling large-scale climate datasets with data quality issues
- Implementing map-reduce patterns for aggregation operations
- Containerization for consistent cross-platform execution
- Performance optimization techniques for Spark applications
- Climate data analysis and meteorological event detection

9.4 Future Enhancements

Potential improvements for future iterations:

- Implement DataFrame/SQL version for comparison
- Add support for multi-year temporal trend analysis
- Include geographic aggregation using station metadata
- Optimize for cluster deployment (YARN, Kubernetes)
- Add real-time streaming analysis capabilities
- Implement visualization dashboard for results
- Scale to full GSOD dataset (all years, all stations)

9.5 Final Remarks

The project successfully demonstrates the power of Apache Spark for distributed data analysis. The RDD API provides fine-grained control over data transformations and enables efficient processing of large climate datasets. The combination of Spark's distributed computing capabilities with Docker's containerization creates a robust, portable solution for climate data analysis.

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 Code Listings

A.1 Main Application Entry Point

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Analyze NOAA GSOD climate data"
    )
    parser.add_argument(
        "--input",
        default="src/main/resources/data/",
        help="Input path for GSOD CSV files"
    )
    parser.add_argument(
        "--output",
        default="src/main/resources/output/",
        help="Output directory for analysis results"
    )

    args = parser.parse_args()
    os.makedirs(args.output, exist_ok=True)
    main(args.input, args.output)
```

Listing 13: climate_analysis.py Main Function

A.2 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:///")
    .set("spark.python.worker.faulthandler.enabled", "true")
)
sc = SparkContext(conf=conf)
```

Listing 14: SparkContext Configuration

A.3 Dockerfile

```
FROM python:3.11-slim

# Install Java
RUN apt-get update && \
    apt-get install -y openjdk-17-jre-headless && \
    rm -rf /var/lib/apt/lists/*

# Install uv
RUN pip install uv

```

```
# Set working directory
WORKDIR /app

# Copy project files
COPY . .

# Install dependencies
RUN uv sync

# Set default command
CMD ["uv", "run", "python", "-m",
     "ii3502_lab6.climate_analysis"]
```

Listing 15: Dockerfile for Container Build