# CS3.304 - Advanced Operating Systems: Assignment 4

Deadlock Detection using MapReduce (PySpark)

## Objective

The primary objective of this assignment is to implement a scalable algorithm for **deadlock detection** in a distributed system by analyzing resource allocation logs. You are required to leverage the **MapReduce programming model** using the **PySpark** library to construct the Resource Allocation Graph (RAG) and iteratively detect cycles.

## Dataset: hpc_rag.log

You will work with a log file (`hpc_rag.log`) (maximum 50,000 records) that records resource allocation and request events. The key fields are: `TIMESTAMP`, `LEVEL` (`WAIT/HOLD`), `PROCESS_ID`, and `RESOURCE_ID`.
Example entries:

```
2025-10-25 00:28:37     WAIT    P105    R024
2025-10-25 00:28:44     WAIT    P110    R009
2025-10-25 00:29:01     WAIT    P174    R024
```

### RAG Edge Construction Rules

The solution must transform the log events into unique directed edges $E$ of the Resource Allocation Graph (RAG):

- `WAIT` events create a **Request Edge**: Process → Resource (`PROCESS_ID` → `RESOURCE_ID`).
- `HOLD` events create an **Allocation Edge**: Resource → Process (`RESOURCE_ID` → `PROCESS_ID`).

**Note:**
1. Each resource has only one instance.
2. Each process and resource participates in at most one deadlock cycle.

## Questions

### Q1: RAG Construction and Graph Metrics

This question focuses on data parsing and the initial Map phase.

- **Q1.1 RAG Edge Generation:** Write a PySpark script to generate the RDD of unique directed edges $E$ (`Source Node, Destination Node`) based on `WAIT` and `HOLD` events.

- **Q1.2 Initial Graph Metrics:** Report the following statistics from the generated edges RDD $E$:

   a. Total number of unique **Process** nodes.

   b. Total number of unique **Resource** nodes.

   c. Total number of unique directed edges in the RAG.

## Q2: Cycle Detection (Deadlock Detection)

This is the core task: implementing the MapReduce pattern for path propagation (maximum cycle length = 25).

- **Q2.1 Cycle Detection Algorithm:** Implement the MapReduce algorithm to detect cycles in the RAG.
- **Q2.2 Deadlock Report:** Output the complete node sequence for all detected deadlock cycles (e.g., $P1 \rightarrow R1 \rightarrow P2 \rightarrow R2 \rightarrow P3$) in ascending order of length (shorter cycles first, lexicographically when lengths are equal). The final repeated cycle node should not be printed again.
- **Q2.3 Involved Processes:** Extract and list all unique **Process IDs** involved in the reported deadlock cycles.

## Q3: Performance and Time-Series Analysis

This question requires integrating performance analysis.

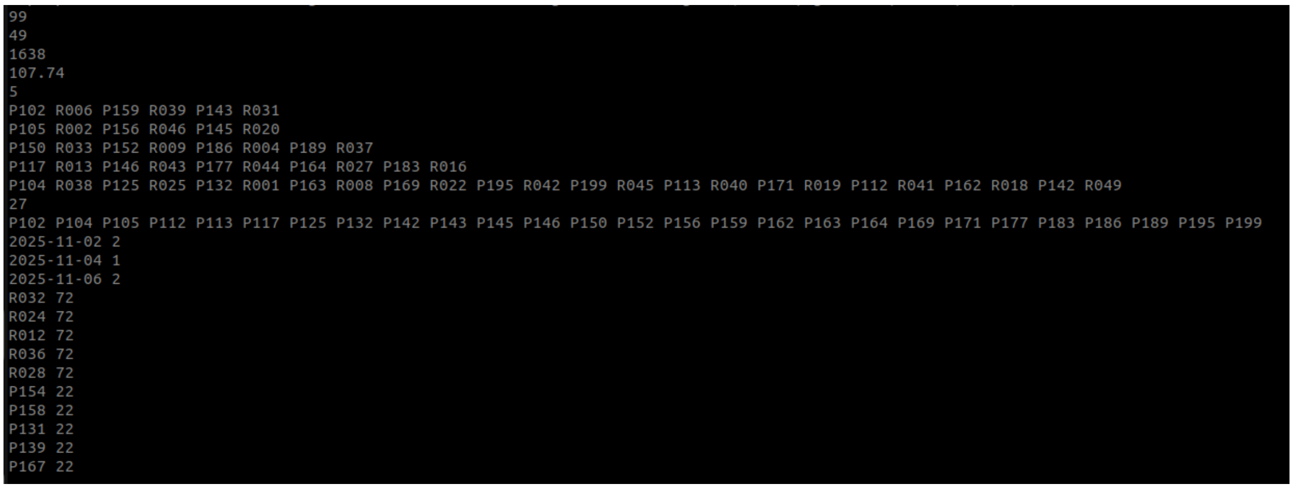
- **Q3.1 Deadlock Formation Timeline:** For each detected deadlock cycle, determine the date when the cycle was first fully formed (i.e., when the last edge of the cycle appeared in the log). Report the frequency of deadlock formations per day as (Date, Count) pairs.
- **Q3.2 Scalability Analysis and Report:** Measure and report the total execution time required to complete **Question 2** by varying the number of cores used for the PySpark job.
- **Q3.3 Observations:** Provide a detailed explanation in your report of the observed relationship between the number of cores used and the execution time (speedup).

## Q4: Graph Structure Analysis

**Resource Contention Hotspots:** Compute and report the **5 Resource Nodes (R-IDs)** with the highest **in-degree** (most processes requesting it) and the **5 Process Nodes (P-IDs)** with the highest **out-degree** (waiting for the most resources).

# Sample Output

The following sample output illustrates the exact format expected from the program. It demonstrates how the program prints various results in a structured format.
An example screenshot of the actual output(for a random log_file) is shown below.



```
99
49
1638
107.74
5
P102 R006 P159 R039 P143 R031
P105 R002 P156 R046 P145 R020
P150 R033 P152 R009 P186 R004 P189 R037
P117 R013 P146 R043 P177 R044 P164 R027 P183 R016
P104 R038 P125 R025 P132 R001 P163 R008 P169 R022 P195 R042 P199 R045 P113 R040 P171 R019 P112 R041 P162 R018 P142 R049
27
P102 P104 P105 P112 P113 P117 P125 P132 P142 P143 P145 P146 P150 P152 P156 P159 P162 P163 P164 P169 P171 P177 P183 P186 P189 P195 P199
2025-11-02 2
2025-11-04 1
2025-11-06 2
R032 72
R024 72
R012 72
R036 72
R028 72
P154 22
P158 22
P131 22
P139 22
P167 22
```

Figure 1: Example console output for Deadlock Detection using PySpark

Each line of the following sample output corresponds to a specific result or metric, as described alongside.

```
99
# Total number of unique Process nodes in the RAG (Q1.2a)

49
# Total number of unique Resource nodes in the RAG (Q1.2b)

1638
# Total number of unique directed edges (Process→Resource or Resource→Process) (Q1.2c)

107.74
# Total execution time (in seconds) for cycle detection using PySpark (Q2.1 & Q3.2)

5
# Total number of deadlock cycles detected in the system (Q2.2)

# Each of the following lines lists a detected deadlock cycle.
# Cycles are space-separated sequences of process and resource nodes,
# sorted first by cycle length, then lexicographically (Q2.2).

P102 R006 P159 R039 P143 R031
P105 R002 P156 R046 P145 R020
P150 R033 P152 R009 P186 R004 P189 R037
P117 R013 P146 R043 P177 R044 P164 R027 P183 R016
P104 R038 P125 R025 P132 R001 P163 R008 P169 R022 P195 R042 P199 R045 P113 R040 P171 R01

27
# Total number of unique processes involved across all detected deadlocks (Q2.3)

P102 P104 P105 P112 P113 P117 P125 P132 P142 P143 P145 P146 P150 P152 P156 P159 P162 P16
# List of all unique Process IDs involved in deadlocks (alphabetically sorted) (Q2.3)
# Deadlock formation timeline (date and number of deadlocks first formed on that day) (Q
2025-11-02 2
2025-11-04 1
2025-11-06 2

# Top 5 resources with the highest in-degree (most requested by processes) (Q4)
R032 72
R024 72
R012 72
R036 72
R028 72

# Top 5 processes with the highest out-degree (waiting for the most resources) (Q4)
P154 22
P158 22
P131 22
P139 22
P167 22
```

**Formatting Rules:**

- Each metric or count appears on a separate line.
- Detected cycles are sorted first by length, then lexicographically for deterministic output.
- The list of processes involved in deadlocks is alphabetically ordered.
- Dates are listed chronologically (earliest to latest).
- Top resources and processes are ranked by degree (descending); ties are resolved by ID.

## Submission Instructions

1. **PySpark Setup:** Install and configure **PySpark** inside a **virtual environment** in the cluster.
2. **Job Execution:** Run all Spark programs using **SLURM job submissions** under your roll number account (e.g., using `srun` or `sbatch`).
3. **Output Format:** Ensure your program's output strictly follows the sample format, as evaluation includes **automatic test verification**.
4. **Credentials:** Login credentials to `ada.iiit.ac.in` and execution details will be shared via email.
5. **Report:** Include `Report.pdf` with results, execution-time comparison for Q2, and observations (Q3.3).
6. **Submission Format:** Submit a folder named `<roll_number>_Assignment4` containing your code files, `Report.pdf`, and README.

## Mandatory Submission Structure (Strictly Enforced)

Strictly follow the directory structure shown below. Submissions that do not follow this structure will receive **zero marks**.

```
<rollnumber>_Assignment4.zip

 <rollnumber>_Assignment4/

     AOS_Assignment4.py         # main Python file (single file for Q1 to Q4)
     Report.pdf                 # performance graphs, analysis, and observations
     README.md                  # describes execution steps and dependencies
```

**Important:**

- Your code must accept the log file path and number of cores as **command-line arguments**. Do not hardcode any file paths in your code.
- All outputs for Q1–Q4 must be printed directly to the terminal (stdout) in the exact format shown in the sample output.
- The program will be run using the command:

  ```
  spark-submit AOS_Assignment4.py ../test_files/log_file_name   no_of_cores
  ```

  Ensure that your script supports this execution format and reads the file path argument correctly. Consider the log_files are present in a folder named test_files.
- For local testing (without Spark cluster mode), you run using python command:

  ```
  Example:
  python3 AOS_Assignment4.py ../test_files/hpc_rag_1.log  4
  ```

  but final evaluation will always be performed using `spark-submit`.
- **Do not** include any `requirements.txt` file in your submission. The evaluation environment (including all required Python and Spark installations) will be preconfigured. Your code should run as-is in this setup without requiring any additional manual installations. The versions of required installations provided are as below:
  Python: 3.10.12
  Apache Spark: 3.4.1
  Java: 11.0.27