ASSIGNMENT – LAB 1.4 : Log Analysis & Spark UI Deep Dive

Learning Outcomes

By completing this assignment, you should be able to:

- Generate synthetic **web server logs** in a realistic text format.
- Parse semi-structured logs using **regex** into structured columns.
- Perform log-based **status/error**, **performance**, and **traffic** analysis with Spark.
- Persist cleaned logs and summaries as **Parquet/CSV**.
- Use the **Spark UI** (Jobs, Stages, Storage, Environment, Executors) to:
    - Read DAG visualizations and stages.
    - Understand shuffles, caching, and resource usage.
    - Relate Spark code to physical execution.

---

0. Context & Prerequisites

You are now instrumenting a simulated e-commerce website from an **operational analytics** viewpoint.

Prerequisites:

- `spark-data/ecommerce/` already exists (from previous labs).
- Spark environment working:
    - Either `master="local[*]"` or your cluster (`spark://localhost:7077`).
    - Spark Application UI accessible at `http://localhost:4040` while jobs are running.

Your goal: go from **raw log lines** → **structured analysis** → **UI interpretation**.

---

1. Part A – Generate Web Server Logs

Goal: Create a realistic text log file for a web server.

Task A.1 — Implement `generate_logs.py`

From your **project root** (e.g. `data-engineering-course`), create:

```
generate_logs.py
```

The script must:

1. Generate **10,000** synthetic log lines.
2. Use random combinations of:
   - IPv4 addresses,
   - HTTP methods (`GET`, `POST`, etc.),
   - URLs (`/home`, `/products`, `/cart`, etc.),
   - HTTP status codes (200, 304, 404, 500, 503),
   - response times (latency in ms),
   - user agents (desktop, mobile, etc.).
3. Emit log lines in an Apache-like format:

   ```
   IP - - [timestamp] "METHOD ENDPOINT HTTP/1.1" STATUS RESPONSETIME "-'
   ```

4. Write all lines to:

   ```
   spark-data/ecommerce/web_logs.txt
   ```

Then run:

```
python generate_logs.py
```

Check:

- File exists: `spark-data/ecommerce/web_logs.txt`.
- Script reports **10,000** generated log entries.

---

2. Part B — Parse and Analyze Logs with Spark

Goal: Parse the raw logs into structured columns and perform analytics.

Task B.1 — Create `lab4_log_analysis.py`

From your project root, create:

```
lab4_log_analysis.py
```

The script must:

1. **Create a SparkSession**
    - App name: `"Day1-LogAnalysis"`.
    - Master: `local[*]` (or your cluster master).
    - `spark.driver.memory = "2g"`.
2. **Load raw logs**
    - Read from `"spark-data/ecommerce/web_logs.txt"` using `spark.read.text(...)`.
    - Print:
        - Number of log lines (`raw_logs.count()`).
        - A small sample with `raw_logs.show(3, truncate=False)`.
3. **Parse the log format using regex**
    - Define a pattern similar to:

      ```
      log_pattern = r'(\S+) - - \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+)
      ```

    - Extract:
        - `ip`
        - `timestamp` (string)
        - `method`
        - `endpoint`
        - `protocol`
        - `status` (cast to `int`)
        - `response_time_ms` (cast to `int`)

using `regexp_extract` on column `"value"`.

- Build a `parsed_logs` DataFrame with these columns.
- Print its schema and the first 10 rows.

4. **Data quality check**
   - Compute:
     - `total_logs = parsed_logs.count()`.
     - `valid_logs = parsed_logs.filter(col("ip") != "").count()`.
     - `invalid_logs = total_logs - valid_logs`.
   - Print these three metrics.
   - Create `logs = parsed_logs.filter(col("ip") != "")` as the cleaned DataFrame.

5. **Basic analytics**
   - Status code distribution:

     ```
     logs.groupBy("status").count().orderBy(desc("count"))
     ```

   - HTTP method distribution:

     ```
     logs.groupBy("method").count().orderBy(desc("count"))
     ```

   - Top 10 most visited pages (by `endpoint`).

6. **Error analysis**
   - Count total `4xx` and `5xx` errors.
   - Top 404 pages:

     ```
     logs.filter(col("status") == 404) \
          .groupBy("endpoint") \
          .count() \
          .orderBy(desc("count"))
     ```

   - Pages causing 5xx errors (group by `endpoint` and `status`).

7. **Performance analysis**

    - Global response time statistics:

        - count, min, max, avg for `response_time_ms` .

    - Slowest endpoints (by average latency), for endpoints with >10 requests.

    - Top 10 slowest individual requests (highest `response_time_ms`).

8. **Traffic patterns**

    - Add `parsed_timestamp` from string `timestamp` :

    ```
    logs_with_time = logs.withColumn(
        "parsed_timestamp",
        to_timestamp(col("timestamp"), "dd/MMM/yyyy:HH:mm:ss Z"),
    )
    ```

    - Compute **traffic by hour**:

        - Add `hour(parsed_timestamp)` as `hour` .
        - Group by `hour` , count, order by `hour` .

9. **User behavior**

    - Top 10 most active IPs (by request count).
    - Compute `requests_per_ip = logs.groupBy("ip").count()` and show `describe()` .
    - Potential bots: IPs with `count > 100` , ordered by count.

10. **Save processed data and summary**

    - Save `logs_with_time` as Parquet to:

    ```
    spark-data/ecommerce/processed_logs
    ```

    using `mode("overwrite")` .

    - Build a `summary` DataFrame with metrics such as:

        - Total Requests
        - Valid Requests
        - Unique IPs

        - Unique Pages
        - 4xx Errors
        - 5xx Errors

    - Show the summary, then write it as a single CSV (use `coalesce(1)`) to:

      ```
      spark-data/ecommerce/log_summary
      ```

11. **Stop Spark** with `spark.stop()`.

---

3. Part C — Spark UI Exploration

Goal: Learn to read the Spark UI and connect it to your code.

> Suggestion:
>
> 1. Open `http://localhost:4040` in your browser.
> 2. Then run:
>
>    ```
>    python lab4_log_analysis.py
>    ```
>
> 3. Refresh the UI while the job runs.

Task C.1 — Jobs Tab

- Go to: `http://localhost:4040/jobs/`.
- Count how many jobs were executed.
- Identify the **longest-running job** (by duration).
- For that job:
    - How many stages does it have?
    - Take a screenshot of the **DAG Visualization**.

Write short answers:

- Why does this job have multiple stages?
- Which operations in your code likely caused shuffles (and therefore extra stages)?

Task C.2 — Stage Details

- Click into one **stage** of the longest job.
- Observe:
  - Number of tasks.
  - Shuffle read/write size.
  - Task durations.

Answer:

- Are task durations similar or is there skew?
- How much data was shuffled in this stage?

Task C.3 — Storage Tab & Caching Demo

Create and run a small script: `lab4_caching_demo.py` that:

- Loads `web_logs.txt` into a DataFrame `logs`.
- **Without caching**:
  - Run `logs.count()` twice, measure and print both times.
- **With caching**:
  - Create `logs_cached = logs.cache()`.
  - Run `logs_cached.count()` twice, measure times.
  - Print a computed speedup.

Add an `input("Press Enter to continue...")` before stopping Spark so the UI stays visible.

While it's paused:

- Go to `http://localhost:4040/storage/` and observe:
  - Cached dataset name.
  - Number of cached partitions.
  - Storage level (e.g., MEMORY_ONLY).
  - Memory usage.

Answer:

- Is the second `count()` noticeably faster when cached?

- What does the Storage tab tell you about how Spark keeps data in memory?

Task C.4 — Environment Tab

Navigate to `http://localhost:4040/environment/` and inspect:

- `spark.master`
- `spark.driver.memory`
- Java version

Answer briefly:

- What master is your app connected to?
- How much memory is allocated to the driver?

Task C.5 — Executors Tab

Navigate to `http://localhost:4040/executors/` .

Observe:

- Number of executors (in local mode you typically see 1 executor + driver).
- Executor memory, active tasks, shuffle metrics, and GC time.

Answer:

- How many executors (excluding driver)?
- Is GC time significant compared to task time?

---

4. Part D — Challenge Problems (Advanced, for extra credit)

Using `logs_with_time` from `lab4_log_analysis.py` , implement the following at the bottom of the script (or in a separate `lab4_challenges.py`), and print the results.

Challenge 1 — Bounce Rate

> **Definition:** percentage of unique IPs that made **exactly one request**.

Steps:

- Group `logs` (or `logs_with_time`) by `ip` and count requests.
- Identify IPs with `count == 1`.
- Compute:

```
bounce_rate = bounced_ips / total_unique_ips * 100
```

Print bounce rate in percentage.

Challenge 2 — Conversion Funnel

We define a simple funnel:

`/home → /products → /product/* → /cart → /checkout`

Goal: estimate how many visitors (unique IPs) reach each step.

Hints:

- Define boolean flags or filtered DataFrames for each step:
  - `/home`
  - `/products`
  - endpoints starting with `/product/`
  - `/cart`
  - `/checkout`
- For each step, compute the set or count of **unique IPs** that visited at least one matching endpoint.
- Compare counts across steps to estimate drop-off.

Challenge 3 — Peak Traffic Hour

- Which hour of the day has the most requests?
- How does the traffic distribute across the 24 hours?

Steps:

- Use `hour(parsed_timestamp)` from `logs_with_time`.
- Group by hour and count.
- Order by hour and print table.

- compute percentages.

Challenge 4 — Suspicious Activity

Detect potentially abusive patterns, for example:

- IPs with **>50 requests per minute**.
- IPs with a **high proportion of 404s** (e.g. >50% of their requests).

Hints :

- Use `groupBy(ip, window(parsed_timestamp, "1 minute"))` to compute requests per minute.
- For 404 rate: group by `ip`, compute total requests and 404 requests, then compute ratio.

Challenge 5 — Performance Bottlenecks

- List endpoints with **average response_time_ms > 2000 ms**, for endpoints with more than a small number of hits (e.g. >10).

- For these endpoints, analyze relation with status code:

    - Group by `endpoint`, `status` and compute `avg(response_time_ms)`.

BONUS: Analyze whether some hours have systematically higher average response times.

---

5. Hints (High Level)

- For regex parsing: always inspect 3–5 sample lines before trusting the pattern.
- For time parsing: ensure timestamp format in `to_timestamp` is exactly `"dd/MMM/yyyy:HH:mm:ss Z"`.
- For "per IP" metrics: **groupBy("ip")** first, then aggregate.
- For "> X requests" conditions: use `filter(col("count") > X)` on an aggregated DataFrame.
- For the funnel: unique IP counts per step are enough (no need for full session tracking in this lab).

---

6. Common Pitfalls

- Wrong log path: `spark-data/ecommerce/web_logs.txt` must exist before running analysis.
- Regex not matching because of typos or spaces → leads to empty columns (check sample output).
- Forgetting to cast `status` and `response_time_ms` to `int` → numeric aggregations fail or behave strangely.
- Calling `count()` on `raw_logs` and `parsed_logs` without understanding extra jobs (normal; each action is a job).
- Trying to compute "per-minute" windows without converting to timestamp.

---

7. Deliverables

You should submit:

1. **Code:**
    - `generate_logs.py`
    - `lab4_log_analysis.py`
    - `lab4_caching_demo.py`
    - `lab4_challenges.py`
2. **Outputs / Evidence:**
    - Console output (or text export) from `lab4_log_analysis.py`.
    - Console output from `lab4_caching_demo.py` showing timing before/after caching and computed speedup.
3. **Screenshots of Spark UI:**
    - Jobs tab (with at least one DAG visualization).
    - A Stage detail page (for a chosen job).
    - Storage tab showing the cached dataset from `lab4_caching_demo.py`.
    - Executors tab.
4. **Short written answers:**
    - Answers to the UI questions in Part C (Jobs, Stages, Storage, Environment, Executors).
    - If challenges attempted:
        - Results + 2–3 lines explaining each (bounce rate, peak hour, suspicious IPs, etc.).