ASSIGNMENT — LAB 1.3 : First Transformations & Actions

Learning Outcomes

By completing this assignment, you should be able to:

- Distinguish **transformations (lazy)** vs **actions (eager)** in Spark.
- Chain multiple DataFrame transformations safely.
- Use core actions: `count`, `show`, `collect`, `take`, `first/head`, `write`.
- Understand when **not** to use `collect()` and why it is dangerous on large data.
- Relate code to **jobs/stages** in the Spark UI.

---

0. Context & Prerequisites

You are continuing work on the **e-commerce dataset** from the previous labs.

Required:

- Lab 1.1 and Lab 1.2 completed.
- CSV files present under `spark-data/ecommerce/`:
  - `customers.csv`
  - `orders.csv`
- Spark can read them from your environment (local or `spark://localhost:7077`, depending on your setup).

---

1. Part A — Lazy Transformations vs Actions

Goal: Show, in a controlled way, what is "lazy" and what triggers execution.

Task A.1 — Create `lab3_transformations.py`

From your project root, create:

```
lab3_transformations.py
```

The script must:

1. **Create a SparkSession**
   - App name: `"Day1-Transformations"`
   - You can use either `local[*]` or your cluster master (depending on your environment for this lab).
     For pedagogy, `local[*]` is acceptable here.
2. **Load** `customers.csv` from `spark-data/ecommerce/customers.csv` with:
   - `header=true`
   - `inferSchema=true`
3. **Apply a sequence of transformations (only transformations, no actions at first):**
   - Filter only `"Enterprise"` customers.
   - Select a subset of columns, e.g.: `customerName`, `phone`, `city`, `country`.
   - Add a new column `customerNameUpper = upper(customerName)`.

   Each step should **print a message** like:
   - `"Transformation X: ... → Defined, but NOT executed yet"`

   This emphasizes laziness.
4. **Print the execution plan** of the final DataFrame using:

   ```
   enterprise_with_upper.explain()
   ```

   This triggers **planning**, but not full execution.
5. **Trigger at least three different actions:**
   - `count()` on the final DataFrame.
   - `show(5, truncate=False)` to display a sample.
   - `limit(3).collect()` and iterate over the small list in Python to print some rows.
6. **Print a short "Key Insights" summary** at the end, explicitly stating:

- What a transformation is.
- What an action is.
- Why lazy evaluation is useful (optimization, avoiding unnecessary work, etc.).

7. **Stop Spark** at the end with `spark.stop()`.

Task A.2 — Observe Spark UI (optional but recommended)

While `lab3_transformations.py` is running:

- Open `http://localhost:4040`.
- Observe:
  - How many **jobs** are triggered.
  - How they map to your actions (`count`, `show`, `collect`).

You don't need a screenshot here, just the understanding.

---

2. Part B — Common Transformations Practice

Goal: Practice the most common DataFrame transformations and see how they combine.

Task B.1 — Create `lab3_transformation_practice.py`

From your project root, create:

```
lab3_transformation_practice.py
```

The script must:

1. **Create a SparkSession**
   - App name: `"Day1-TransformationPractice"`
   - `master = "local[*]"` (fine for this lab).
2. **Load** `orders.csv` from `spark-data/ecommerce/orders.csv` with header + schema inference.
3. Implement the following **exercises** using DataFrame transformations + actions:

Exercise 1 – Filter

> Filter orders with `totalAmount > 5000`.

- Store the result as `high_value`.
- Print:
  - Number of such orders → `high_value.count()`
  - First 5 rows → `high_value.show(5)`

Exercise 2 – Select & Rename

> Build a simplified view of orders.

- Select and **rename**:
  - `orderNumber` → `id`
  - `orderDate` → `date`
  - `totalAmount` → `amount`
  - keep `status`
- Save as `order_summary` and display the first 5 rows.

Exercise 3 – Add a Computed Column

> Classify orders by size.

- Add a new column `orderSize` based on `totalAmount`:
  - `< 1000` → `"Small"`
  - `1000-4999` → `"Medium"`
  - `>= 5000` → `"Large"`
- Show a few rows with: `orderNumber`, `totalAmount`, `orderSize`.
- Compute and display the **distribution**:

```
orders_categorized.groupBy("orderSize").count().orderBy("orderSize")
```

Exercise 4 – Chain Multiple Transformations

> Build a small pipeline for "shipped orders".

- Starting from `orders`, chain:
  - Filter only `status == "Shipped"`.
  - Add `amountRounded = round(totalAmount, 0)`.
  - Add `priority`:
    - `"High"` if `totalAmount > 5000`
    - `"Medium"` if `totalAmount > 2000`
    - `"Low"` otherwise
  - Select: `orderNumber`, `orderDate`, `amountRounded`, `priority`, `status`.
- Store as `processed_orders`.
- Print how many shipped orders and show the first 10.

Exercise 5 — Drop Unnecessary Columns

> Reduce the schema.

- Drop at least `requiredDate` and `paymentMethod` from `orders`.
- Show:
  - Number of original columns.
  - Number of columns after dropping.
  - First 5 rows of the reduced DataFrame.

Exercise 6 — Distinct Values

> Explore categorical values.

- Show distinct values of `status`.
- Show distinct values of `paymentMethod`.

Task B.2 — Implement "Your Turn" Practice Questions

At the end of `lab3_transformation_practice.py`, add a section where **you implement** the following:

1. Filter orders from `'2024-06-01'` onwards.
2. Create a Boolean column `isLargeOrder` (true if `totalAmount > 3000`).
3. Select only orders with status `"Processing"` or `"Shipped"`.
4. Add a column `orderCode` formatted as `"ORDER-00001"` based on `orderNumber`.
    - Hint: use `concat`, `lit`, and either `lpad` or `format_string`.
5. Find the **top 10 most expensive orders** by `totalAmount`.

> These are **student work**; in the assignment, they must be implemented and kept in the final file.

---

3. Part C — Actions Deep Dive & Performance

Goal: Understand the semantics and performance implications of the main actions.

Task C.1 — Create `lab3_actions.py`

From your project root, create:

```
lab3_actions.py
```

The script must:

1. **Create a SparkSession**
    - App name: `"Day1-Actions"`
    - `master = "local[*]"`.
2. **Load** `customers.csv` as before.
3. Implement the following sections:

Action 1 — `count()`

- Print the total number of customers.
- Filter `Enterprise` customers and print their count.

Action 2 — `show()`

- Show default behavior: `customers.show()` (20 rows, truncated).
- Show 5 rows with `truncate=False`.
- Show 3 rows with `vertical=True`.

Comment briefly in prints what each variant is useful for.

Action 3 — `collect()`

- Take a **small sample**:

```
small_sample = customers.limit(3).collect()
```

- Print:
  - Type of `small_sample`.
  - Type of the first element.
  - A few values, e.g. `customerName`, `country`.

Add an on-screen **warning** message that `collect()` is only safe on small datasets.

Action 4 — `take(n)`

- Call `customers.take(5)` and iterate over the result.
- Print a short summary per row.

Action 5 — `first()` and `head()`

- Call `first()` and `head()`.
- Print `customerName` for each and note (in text) that they return the same row on a DataFrame.

Action 6 — `write`

- Filter customers from `"USA"` and write them as **single CSV**:
  - Use `.coalesce(1)`.

- Mode: `"overwrite"`.
- Header: `true`.
- Output path: `spark-data/ecommerce/usa_customers`.

- Write all customers as **Parquet** to:
  - `spark-data/ecommerce/customers_parquet`.

Action 7 – `foreachPartition()`

- Define a small function that prints the size of each partition (only high-level info).
- Apply it using `customers.foreachPartition(...)`.

Note: Output may be interleaved in console. That's fine.

Action Performance Comparison

- Use `time.time()` to measure:
  - `customers.count()`
  - `customers.show(5)`
  - `customers.limit(100).collect()`
- Print timing for each and interpret:
  - `count()` vs `show()` vs `collect()`.

Dangerous Patterns (Theory)

At the end of the script, print a small block that summarizes **dangerous patterns**, for example:

- `df.collect()` on large DataFrames.
- `df.toPandas()` on large DataFrames.
- Loops with `for row in df.collect(): ...`

And the safer alternatives:

- `limit().collect()`
- `take(n)`
- `show()`

- `foreachPartition`
- `write.parquet(...)`

---

4. Hints

- **Transformations vs Actions:**
  - `filter`, `select`, `withColumn`, `join`, `groupBy` = transformations (lazy).
  - `count`, `show`, `collect`, `take`, `first`, `write` = actions (eager).
- **Lazy evaluation**: printing the DataFrame object does not execute anything; only actions traverse the DAG.
- `collect()` vs `take()` vs `show()`:
  - `collect()` → returns **all rows** in memory on the driver. Dangerous on big data.
  - `take(n)` → returns **n rows** to the driver.
  - `show(n)` → prints **n rows** in console, but does not give you a Python list.
- When chaining transformations, **no work happens** until the first action is reached.

---

5. Common Pitfalls

- Using `collect()` on the full DataFrame with thousands / millions of rows → potential OOM.
- Confusing transformation output with "executed result" — just because you defined `enterprise = customers.filter(...)` doesn't mean Spark has filtered anything yet.
- Assuming `show()` modifies the DataFrame — it doesn't; it just displays.
- Forgetting to stop Spark, leaving sessions open.
- For time measurements, running multiple actions in a row without understanding each one triggers a separate job.

---

6. Deliverables

You should submit:

1. **Terminal outputs** (or screenshots / text logs) for:
   - `lab3_transformations.py`

- `lab3_transformation_practice.py`
- `lab3_actions.py`

2. **Final version of** `lab3_transformation_practice.py` with the **5 "Your Turn" exercises implemented**.

3. A short written answer (3–5 lines):

   - Difference between **transformation** and **action** in Spark.
   - Why Spark uses **lazy evaluation**.

4. A short comparison (2–3 lines each) explaining:

   - When to use `collect()`.
   - When to use `take()`.
   - When to use `show()`.

     Include **at least one danger case** for inappropriate `collect()` usage.

These answers can be in a separate markdown/text file (e.g. `lab3_3_answers.md`) or as comments at the bottom of one of the scripts.