Exercise 0 – Notebook & Data Setup

Create a **new Databricks notebook** with default language **Python**.

1. Attach the notebook to a running cluster.
2. Load a reasonably **large dataset** into a DataFrame `df` (at least a few million records if possible).
3. Perform some basic checks:
   - Show schema.
   - Show a few rows.
   - Count the records.
4. Create one or two **derived DataFrames** (e.g. filtered, aggregated) that you'll reuse later for caching experiments.

Deliverables for this exercise:

- A DataFrame `df` loaded from some data source.
- At least one derived DataFrame, e.g. `df_filtered`, `df_agg`.

Exercise 1 – Warm-up: Actions vs Transformations

1. For `df`, define a transformation pipeline, e.g.:
   - `filter` on some column.
   - `withColumn` to add or transform a column.
   - `select` to keep only a subset of columns.
2. **Do not** call any actions yet. Confirm there's no job triggered in the UI.
3. Call an **action** (e.g. `count()`) on the final DataFrame and observe:
   - A job runs for the first time.
4. Call the same action again on the same DataFrame **without caching**:
   - Observe that Spark recomputes the lineage (you will see another job).

Deliverables:

- Code for transformation pipeline.
- Evidence (from the UI or logs) that the job ran twice.

Exercise 2 – Basic `cache()` with Performance Comparison

Take a **CPU-heavy / shuffle-heavy** transformation (like a `groupBy` + `agg`).

1. Measure the **time** taken to run a particular action (e.g., `count()`, `collect()` or `show()`):
   - First **without caching**.
2. Now **cache** the resulting DataFrame.
3. Run the same action again and measure the time:

    - Compare first and second runs.
4. Check if the DataFrame is cached using Spark's catalog APIs.
5. Explore:
    - How many actions benefit from the cache.
    - What happens if you **re-cache** or **unpersist**.

Deliverables:

- Timings (even approximate) for uncached vs cached action.
- Code and comments summarizing your observations.

Exercise 3 – `persist()` with Different Storage Levels

1. Create a DataFrame `df_heavy` that's **expensive to compute** (e.g., multi-step pipeline with joins, aggregations, etc.).
2. Persist `df_heavy` with:
    - `MEMORY_ONLY`
    - `MEMORY_AND_DISK`
    - `DISK_ONLY`
3. For each storage level:
    - Persist the DataFrame.
    - Trigger an action to materialize it.
    - Measure the runtime of:
        - First action (materialization).
        - Second action (served from cache).
4. Check `df_heavy.storageLevel` after each persist.
5. If possible, observe memory usage / storage info in the Spark UI (Storage tab):
    - How much of the DataFrame fits in memory?
6. Unpersist the DataFrame and confirm it's removed from cache.

Deliverables:

- Code snippets showing different storage levels.
- Short notes on performance and storage level behavior.

Exercise 4 – Caching Temporary Views & SQL Queries

1. Register a **temporary view** from your DataFrame, e.g. `df.createOrReplaceTempView("my_table")`.
2. Run a **SQL query** using `%sql` on this view that is somewhat expensive (join, groupBy, etc.).
3. **Cache the table** using Spark SQL or PySpark:
    - e.g., `CACHE TABLE` or corresponding PySpark API.
4. Re-run the same SQL query:
    - Compare the runtime with and without cache.
5. Check:

    - Whether the table is listed as cached in Spark's catalog.
6. Drop or uncache the table and confirm it is removed.

Deliverables:

- SQL and Python code related to caching the table.
- Observations about performance benefits.

Exercise 5 – Cache Invalidation, Reuse, and Best Practices

Cache a DataFrame `df_cached`.

1. Perform multiple actions on `df_cached` and note performance.
2. Modify the **underlying data** (simulate this however is easiest in your environment):
    - Example: re-read the source file but with an extra filter, limit, or sample to mimic data changes.
3. Observe:
    - Does `df_cached` automatically reflect underlying data changes?
4. Unpersist `df_cached` and recreate it from the new data:
    - Cache again and rerun actions.
5. Think about and write down answers to:
    - When should you cache?
    - When should you avoid caching?
    - How do you avoid over-caching and memory pressure?

Deliverables:

- Code demonstrating cache reuse and unpersist.
-