# Lab 2.2 — Spark Internals: Plans, Caching, and RDDs

Session 2.2 — Apache Spark & Query Execution Internals
Individual hands-on lab
**Tools:** Scala 2.13, Spark 4.0.2, scala-cli, Java 17+

---

## 1. Objective

Observe Spark's execution engine on a real dataset — not in theory but through query plans, timing measurements, and the Spark UI. You will convert raw CSV data to Parquet in two lines, read and interpret physical plans, measure the concrete impact of caching, and implement iterative KMeans on RDDs to cluster global weather stations into climate zones.

## 2. Setup

### 2.1. Prerequisites

- Java 17 or later (`java -version`)
- `scala-cli` (scala-cli.virtuslab.org)

All exercises run in Spark local mode — no cluster required. The Spark UI is available at localhost:4040 while any exercise is running.

### 2.2. Dataset — NOAA Global Surface Summary of the Day

Daily weather observations from  10 000 stations worldwide. One CSV file per station,  365 rows each. Key columns: `STATION`, `DATE`, `LATITUDE`, `LONGITUDE`, `NAME`, `TEMP` (°F), `DEWP` (dew point °F), `SLP` (sea-level pressure hPa), `WDSP` (wind speed knots), `PRCP` (precipitation inches). Missing values are coded as `9999.9` / `999.9` / `99.99`.

Download one year ( 300 MB compressed,  4.7 M rows uncompressed):

```
mkdir -p data/noaa/raw
curl -L -o data/noaa/2023.tar.gz \
  "https://www.ncei.noaa.gov/data/global-summary-of-the-day/archive/2023.tar.gz"
tar -xzf data/noaa/2023.tar.gz -C data/noaa/raw/
```

Verify the download:

```
scala-cli run assets/setup.scala
```

This prints the Spark version and the row count. If you see  4–5 million rows, you are ready.

# 3. Exercise 1 — Two lines to Parquet

Open `assets/ex1.scala`. It reads the entire NOAA directory as CSV and writes it as Parquet — two lines of transformation code.

```
scala-cli run assets/ex1.scala
```

## 3.1. Tasks

1. Compare directory sizes:

   ```
   du -sh data/noaa/raw data/noaa/2023.parquet
   ```

   How large is the compression ratio? Why does Parquet outperform raw CSV here — think about both column encoding and the nature of weather data (many repeated values per station, slow-changing temperatures).

2. The file runs the same filter (`TEMP < 32`, i.e. freezing) on both the CSV and the Parquet and calls `explain("formatted")` on each. Locate `PushedFilters` in the Parquet plan. What does the engine skip at scan time that it cannot skip on CSV?

3. Change the filter to `TEMP.cast("double") + 0 < 32` on the Parquet path. Does `PushedFilters` still appear? Why not?

# 4. Exercise 2 — Reading a physical plan

Still in `ex1.scala`, inspect the plan for the join query: per-station annual mean temperature, joined to the station dimension extracted from the same dataset.

1. Which join strategy did Spark choose? Look for `BroadcastHashJoin` or `SortMergeJoin`. Why did it choose that strategy for the station dimension?

2. How many `Exchange` nodes appear in the default plan? Each one is a full network shuffle. How many stages does that imply?

3. Force a sort-merge join:

   ```
   spark.conf.set("spark.sql.autoBroadcastJoinThreshold", "-1")
   ```

   Run `explain("formatted")` again. How many `Exchange` nodes appear now? What is the concrete cost of losing the broadcast optimisation?

4. Which operators carry a `*(N)` prefix? What does that prefix mean, and which operators in the plan are **outside** a WholeStageCodeGen boundary?

The `readLine()` call at the end of the exercise keeps the Spark context alive. Open localhost:4040, navigate to **Jobs** and **Stages**, and inspect the DAG visualisation for the join query before pressing Enter.

# 5. Exercise 3 — Caching

Open `assets/ex2.scala`. It runs the same per-station aggregation five times with and without caching, printing elapsed time for each run.

```
scala-cli run assets/ex2.scala
```

### 5.1. Tasks

1. Record the timing output. Without caching, runs 2–5 should be roughly equal (each re-reads Parquet from disk). With caching, run 1 materialises the cache and subsequent runs read from JVM heap. What happens to run 1 with cache?
2. Open the Storage tab (`localhost:4040/storage`) while the cached run executes. What fraction of the DataFrame fits in memory? What happens when you call `df.unpersist()`?
3. The file also runs with `DISK_ONLY` persistence. Compare the timing with `MEMORY_AND_DISK`. Under what workload would `DISK_ONLY` be a better choice?
4. Describe a pipeline where calling `cache()` would make performance **worse**.

## 6. Exercise 4 (bonus) — Iterative KMeans: clustering climate zones

Open `assets/ex3.scala`. It provides:

- `case class StationFeatures(temp, dewp, wdsp, prcp)` — the feature type.
- `loadFeatures(spark, path)` — reads NOAA Parquet, aggregates per-station annual means, and returns an `RDD[StationFeatures]`.
- Helper functions `distance`, `nearest`, `addFeatures`, `scaleFeatures`.
- A `timed` utility to measure wall-clock execution of each run.
- The `kmeans` stub — **yours to implement**.

The `kmeans` function body contains the algorithm steps in plain English. Implement it using the RDD API before uncommenting and running the timing harness.

```
scala-cli run assets/ex3.scala
```

### 6.1. Tasks

1. Implement `kmeans` following the commented steps. Run it once to verify the centroids look reasonable before moving to the timing comparison.
2. Uncomment the body of `ex3()` and run both the uncached and cached variants. The uncached version rebuilds the full lineage from the Parquet source on every iteration — how does the uncached time grow across iterations?
3. With `rdd.cache()`, the first iteration materialises the partitions; subsequent ones read from memory. Does per-iteration time stabilise after the first cached run?
4. The script prints cluster centroids. Match them against known climate zones (tropical: high temp + high dewpoint; polar: low temp; arid: low dewpoint + low precip; temperate: moderate everything). Do the centroids correspond to recognisable zones?
5. Why is this algorithm difficult to express cleanly with the DataFrame API without an explicit `checkpoint()` or `cache()` between iterations?
6. **(Hard)** The assignment step shuffles all raw feature vectors across the network. Rewrite it with `aggregateByKey` to compute the cluster sum and count in a single combiner pass — sending only `(StationFeatures, Long)` per partition instead of all points. How does this reduce shuffle volume?

## 7. Key Takeaways

- Parquet reduces I/O by an order of magnitude for selective queries. Pushed filters skip entire row groups **before** any data value is decoded — but only when the filter column is not wrapped in a function.
- `BroadcastHashJoin` eliminates shuffles for small dimension tables. One configuration knob (`autoBroadcastJoinThreshold`) controls the threshold. Knowing when it fires — and when it silently falls back to sort-merge — is essential for tuning.
- Caching pays off only when the same DataFrame is consumed by multiple actions. Use the Storage UI and timing to verify, not intuition.
- Iterative algorithms that mutate shared state between rounds require the RDD API with explicit `cache()`. Without it, Spark rebuilds the full lineage from the source on every iteration — a quadratic disaster on large datasets.