Apache Spark vs Ray: a practical comparison for data engineers (with an Airflow tutorial)

If you work in data engineering, you've heard two names a lot:

- **Apache Spark** (the classic big-data workhorse)
- **Ray** (the newer "Python-first" distributed execution engine)

This article explains both **simply** (so anyone can follow) but also **deeply enough** to make real architecture decisions.

And because opinions are cheap and benchmarks are expensive, we include a mini project:

- an **Airflow DAG that runs a Spark job**
- an **Airflow DAG that runs a Ray job**
- the **same workload** in both
- a small **benchmark harness** to compare runtime

Table of Contents

The simplest mental model

Spark in one sentence

Spark is a distributed data processing engine built around a DataFrame/SQL execution model with strong optimizer support.

Ray in one sentence

Ray is a distributed execution engine built around tasks and actors (and higher-level libraries like Ray Data), designed to scale Python workloads.

Analogy

- **Spark** is like a large factory assembly line: you describe the plan, the factory optimizes the route, then runs it efficiently.
- **Ray** is like a flexible team of workers you can assign tasks to: you decide the structure, and it executes your Python tasks across machines.

When Spark usually wins

Spark tends to be the default choice when:

- you're doing **ETL at scale** (joins, aggregations, window functions)
- you want **SQL** + cost-based optimization
- you need a mature ecosystem (Spark SQL, structured streaming, Delta/Iceberg integration)
- you have a Hadoop/Spark-friendly platform already

Why: Spark's execution engine and query optimizer are extremely strong for relational-style workloads.

When Ray usually wins

Ray tends to shine when:

- the workload is **Python-native** and not purely SQL-shaped
- you have **ML pipelines** (training, batch inference, feature computation)

- you need **fine-grained parallelism** (many tasks, custom logic)
- you want to mix patterns: tasks + actors + data processing

Why: Ray makes it easy to distribute “normal Python” and scale libraries around it.

A deeper view: execution model (what actually happens)

Spark execution model

- You build a plan (DataFrame transformations).
- Spark constructs a **DAG**.
- Spark optimizes the plan (Catalyst optimizer).
- Spark executes stages, shuffling data when needed.

Key words:

- **Catalyst**: Spark SQL optimizer (logical → physical plan)
- **Shuffle**: redistribution of data across the cluster (often the expensive part)
- **Partition**: how data is split across workers

Ray execution model

- You submit **tasks** (functions) and/or **actors** (stateful workers).
- Ray schedules them across the cluster.
- Ray handles object passing using an object store.

Key words:

- **Task**: stateless remote function execution
- **Actor**: stateful worker (useful for caches, models)
- **Object store**: where distributed objects live (avoids copying too much)

Data engineering checklist: the decision criteria

1) Workload shape

- **Mostly SQL/joins/aggregations** → Spark is usually simpler and faster.
- **Custom Python logic** (heavy UDF-ish work) → Ray often feels more natural.

2) Team skills

- Strong SQL + Spark experience → Spark.
- Python-first + ML tooling → Ray.

3) Operational maturity

- Spark has battle-tested ops patterns in many enterprises.
- Ray is improving fast but may require more platform ownership depending on your environment.

4) Integration surface

- If your stack is already “Spark-native” (Databricks, EMR, on-prem Spark) → Spark is straightforward.
- If your stack is “ML platform first” (Ray Serve, distributed training, GPU scheduling) → Ray fits well.

The tutorial project you can run

This repo folder contains a mini project:

- `jobs/spark_job.py`: reads a dataset, runs transformations, writes output
- `jobs/ray_job.py`: does the same workload using Ray
- `dags/airflow_spark_dag.py`: runs Spark job from Airflow
- `dags/airflow_ray_dag.py`: runs Ray job from Airflow
- `bench/run_benchmark.py`: runs both and compares runtime
- `data/generate_data.py`: generates a synthetic dataset so you can run locally

The workload (kept identical)

We use a realistic “data engineering” style workload:

- generate event data (users, events, timestamps, amounts)
- filter by date
- group by user
- compute aggregates (count, sum)
- write results

Why this workload?

- it matches real pipelines
- it’s easy to understand
- it’s fair to compare

Performance comparison: how to benchmark fairly

If you want to write honestly about performance:

- Use the **same input data**
- Use the **same machine** (or same cluster shape)
- Warm up once (first run often includes JVM/Python startup)
- Compare:

- wall clock time

- memory usage (if possible)

- output row counts (to confirm correctness)

Your article should say:

- dataset size
- hardware specs

- Spark config and Ray config
- what exactly was measured

Practical conclusion

- Choose **Spark** when your world is mostly SQL/ETL, and you want mature, optimized distributed queries.
- Choose **Ray** when your world is Python-native, you mix ML + data processing, or you need flexible task/actor orchestration.

In the next articles, we can go deeper into:

- Spark shuffles and how to avoid them
- Ray actor patterns for feature computation and batch inference
- running both on Kubernetes
- cost-performance tradeoffs