DATA ENGINEERING

Spark Performance Tuning: A Field Guide

Diagnosing shuffles, skew and memory pressure in production

Houssam Kodad

PDF · DATAFORGE BOOKS

CONTENTS

Table of Contents

CHAPTER 01

How Spark Actually Runs Your Job

A Spark job that flies on a sample can collapse on the full dataset, and the stack trace rarely tells you why. Tuning Spark is less about memorising configuration flags and more about understanding what the engine is actually doing when you call an action — and learning to read the evidence it leaves behind in the UI. This is a field guide: short on theory, long on the diagnostic moves that turn a six-hour job into a forty-minute one.

This first chapter builds the foundation the rest of the book stands on. We trace how Spark turns your code into a physical plan, what jobs, stages and tasks really are, why laziness means the slow line of code is rarely where the cost lives, and — most importantly — why the distinction between wide and narrow transformations is the single most useful thing to internalise before you tune anything.

Master this mental model and tuning stops being guesswork. You will be able to look at a transformation and predict whether it shuffles, look at the UI and find the real bottleneck, and look at a slow stage and know which of a handful of causes is to blame. That predictive ability, not a list of magic settings, is what makes someone good at Spark.

How Spark Runs Your Job

Spark organises work into a hierarchy worth committing to memory. An action — count, write, collect — triggers a job. Each job is split into stages, and the boundaries between stages are shuffles, the points where data must be redistributed across the cluster. Each stage is split into tasks, one per partition, and tasks are the actual unit of parallel execution on the cluster's cores.

This hierarchy is your map for every performance problem. When a job is slow, you find the slow stage; within that stage, you look for tasks that take far longer than their peers. Almost every Spark problem shows up in exactly one of three forms: a skewed task that runs long after the others finish, an excessive number of tiny tasks, or a stage that shuffles far more data than it should.

Lazy Evaluation and the DAG

Spark transformations are lazy. When you filter, join or select, nothing runs — you are only describing a plan. Spark builds the directed acyclic graph of that plan, optimises it, and executes only when you call an action. This is why a mistake in a transformation often surfaces seconds later, when the action fires, rather than at the line that wrote it.

Laziness is a feature: it lets the Catalyst optimiser see the whole computation and rewrite it — pushing filters down to the scan, pruning unused columns, and combining steps. But it also means the line in your stack trace is rarely where the real cost lives. Reading the query plan with explain, not staring at the code, is how you find the truth of what Spark will actually do.

```
df = (spark.read.parquet("events/")
        .filter(col("country") == "FR")   # narrow: no shuffle
        .groupBy("user_id").count())        # wide: forces a shuffle

df.explain("formatted")   # read the physical plan before you tune anything
```

Wide vs Narrow Transformations

A narrow transformation — filter, map, select, withColumn — needs only the data already present on each partition. It is cheap, fully parallel, and never moves data between machines. A wide transformation — groupBy, join, distinct, repartition — needs data from many partitions brought together, which forces a shuffle: serialising data, writing it to disk, sending it across the network, and reading it back on the other side.

Shuffles are where Spark jobs go to die, and most of this book is, in one way or another, about reducing them, sizing them sensibly, and surviving the skew they expose. Once you can look at a chain of transformations and predict exactly where the shuffle boundaries fall, you can predict where the job will struggle before you ever run it — and that single skill is the foundation of everything that follows.

Reading the Spark UI

The Spark UI is your instrument panel, and learning to read it quickly is the highest-leverage skill in this book. Start at the stages view and find the stage consuming the most time. Open it and look at the task summary: the difference between the median task duration and the maximum tells you instantly whether you have skew — a healthy stage has tasks of similar length, a sick one has a long tail.

From there, the per-task metrics tell the rest of the story. Large shuffle-read or shuffle-write numbers point to an expensive redistribution. Spill-to-disk metrics reveal memory pressure. A stage with thousands of tiny tasks or a handful of giant ones points to a partitioning problem. You are not guessing; the evidence for every diagnosis in this book is sitting in these views.

Data Skew: the Usual Suspect

Skew is the most common and most frustrating Spark problem. It happens when one key — a null, a default, a single huge customer — holds far more rows than the others, so after a shuffle one partition is enormous while the rest are tiny. One task then runs for an hour while the cluster sits idle waiting for it, and no amount of extra machines helps, because the work cannot be split.

The cure depends on the cause, and we devote a full chapter to it. Sometimes you filter or special-case the skewed key; sometimes you "salt" it by adding a random suffix to spread it across partitions and then re-aggregate; sometimes adaptive query execution can split the skewed partition for you. Recognising skew from the UI's long-tail task distribution is the first step, and it is one you will take constantly.

```
# Salting a skewed join key: spread the hot key across N buckets,
# join, then aggregate back. Removes the single oversized partition.
N = 16
big  = big.withColumn("salt",  (rand() * N).cast("int"))
small = small.crossJoin(spark.range(N).withColumnRenamed("id", "salt"))
joined = big.join(small, ["key", "salt"], "inner")
```

Partitions, Memory and Spills

Two settings cause a disproportionate share of Spark pain, and both are about sizing. Too few partitions and tasks are huge, spill to disk, and fail with out-of-memory errors; too many and the scheduler drowns in overhead launching tiny tasks. The right number depends on your data size and cluster, and a large part of tuning is simply finding partition sizes in the sweet spot of roughly a hundred to a few hundred megabytes each.

Memory deserves the same respect. Spark divides executor memory between execution (shuffles, joins, aggregations) and storage (cached data), and when execution memory runs short, Spark spills intermediate data to disk — correct, but dramatically slower. Spill metrics in the UI are a direct signal that a stage needs more memory, fewer rows per task, or a smarter algorithm. We will return to all three repeatedly.

A Repeatable Tuning Process

The rest of this book hangs on a simple, repeatable loop, and it is worth stating now. Run the job. Open the UI and find the single slowest stage. Read its task distribution to classify the problem — skew, shuffle, partitioning, or memory. Apply the one targeted fix for that class. Measure again. Resist the urge to change five settings at once, because then you learn nothing about which one mattered.

This discipline is what separates engineers who tune Spark from engineers who poke at it. Every chapter that follows takes one class of problem, shows you how to recognise it in the UI, and gives you the specific moves to fix it. Keep the loop in mind as you read, and you will find that most "mysterious" Spark slowness is, in fact, one of a small number of familiar shapes.

This is a free sample

You've reached the end of the sample chapter.

Get the complete book — every chapter, fully worked — at **dataforgebooks.com**.

FULL EDITION · 176 PAGES · PDF

Read the full title at **dataforgebooks.com**

Questions? support@dataforgebooks.com