2.2 — Apache Spark & Query Execution Internals

Data Processing at Scale

2026-03-27

Data Processing at Scale — Day 2

Course website

https://vbergeron.github.io/data-processing-at-scale/

This presentation

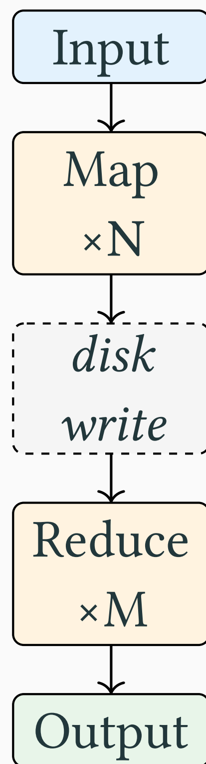
https://vbergeron.github.io/data-processing-at-scale/2.2-spark.pdf

Lazy Evaluation & DAG Execution

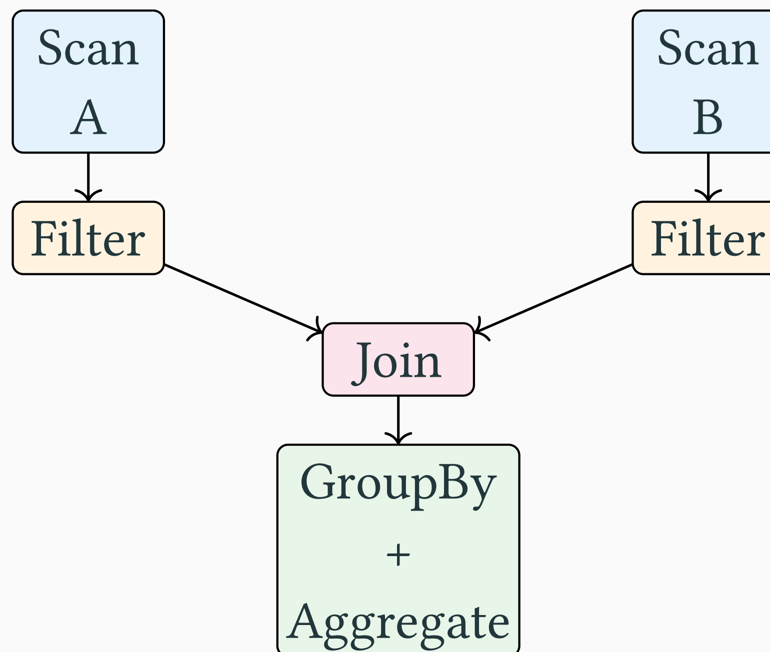
You built MapReduce.
Spark is what happens when you **generalize** it.

From MapReduce to Spark

MapReduce — fixed pipeline



Spark — arbitrary DAG, in-memory

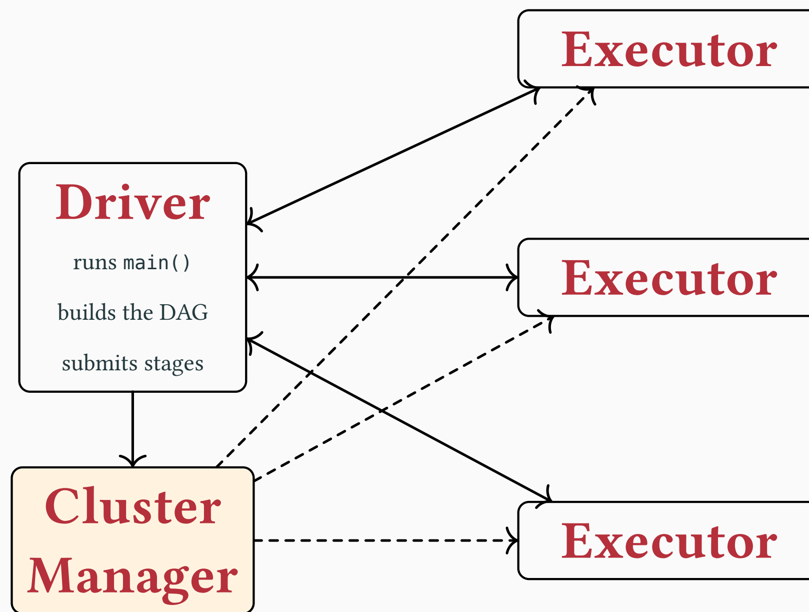


Resilient Distributed Datasets

The **RDD** (Resilient Distributed Dataset) is Spark's core abstraction, introduced in the 2012 NSDI paper.

An RDD is:

- **Distributed**: split into partitions spread across executor JVMs
- **Immutable**: operations never modify an RDD; they produce a new one
- **Typed**: `RDD[String]`, `RDD[(K, V)]`, etc. — checked at compile time
- **Resilient**: if a partition is lost, Spark recomputes it by replaying its **lineage** — the sequence of transformations that produced it



The Cluster Manager provisions executor JVMs; the Driver then dispatches tasks and collects results **directly** via RPC — the Cluster Manager is no longer in the loop. The driver **never touches data**.

Kubernetes

Each executor runs in its own **pod** — ephemeral, isolated, deleted when the task completes.

Dynamic allocation (`spark.dynamicAllocation.enabled = true`): the driver requests new executor pods as pending tasks grow, and releases idle pods. No fixed pool to pre-provision.

Driver placement:

- **Client mode**: driver runs on the submitting machine, executor pods connect back — good for interactive sessions, requires network access from the cluster
- **Cluster mode**: driver itself runs as a pod — fully in-cluster, preferred for production jobs submitted via CI or a scheduler (Argo, Airflow)

Transformations vs actions

Every Spark operation is either a **transformation** or an **action**.

Transformations — describe what to compute, produce a new RDD/DataFrame, execute nothing:

- `map`, `filter`, `groupBy`, `join`, `select`, `withColumn`

Actions — trigger execution, return a result or write output:

- `count`, `collect`, `show`, `write`, `save`

Nothing runs until an action is called. The driver builds the full plan first.

Why lazy evaluation?

Building the plan before running it allows the optimizer to:

- Eliminate columns never referenced downstream (**projection pruning**)
- Push filters as close to the source as possible (**predicate pushdown**)
- Reorder joins based on table sizes (**join reordering**)
- Fuse consecutive operations into a single pass (**pipelining**)

A row-at-a-time system executing eagerly cannot do any of this — each operation commits its output before the next one has a chance to react.

The DAG

Each transformation adds a node to the DAG. Nodes carry:

- The operation type and its parameters
- A reference to parent node(s)
- The **partitioning scheme** of the output

Spark walks this DAG during planning, identifies **shuffle boundaries**, and groups everything between two shuffles into a **stage**.

Within a stage, all operations are **pipelined** — no intermediate materialization.

Stages and tasks

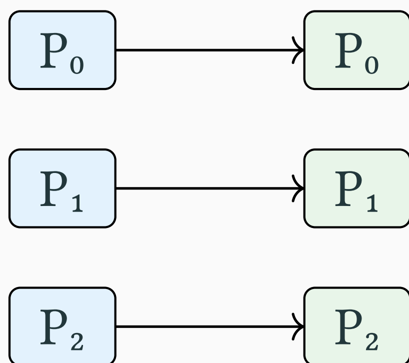
A **stage** is a maximal sequence of operations that can run without a shuffle.

A **task** is one stage applied to one partition. If a stage processes 200 partitions, it creates 200 tasks, each running on one executor core.

Stage boundaries are always caused by **wide dependencies** — operations where a row can go to any output partition (shuffle). Narrow dependencies (one input partition → one output partition) never create a stage boundary.

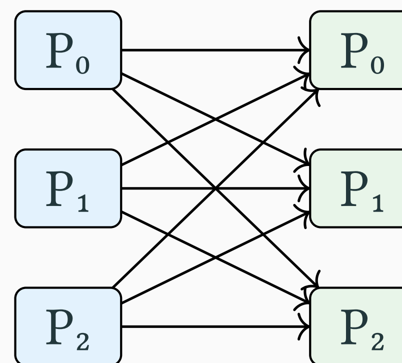
Narrow vs wide dependencies

Narrow — 1 input ⟶ 1 output partition
no shuffle



`map`, `filter`, `union`

Wide — 1 input ⟶ any output partition
shuffle



`groupBy`, `join`, `distinct`

What triggers a shuffle

Operations with wide dependencies:

- `groupBy` / `groupByKey`
- `join` (unless one side is broadcast)
- `repartition` / `coalesce` (increasing partition count)
- `distinct`
- Window functions with `PARTITION BY`

Operations with narrow dependencies (no shuffle):

- `map`, `filter`, `flatMap`
- `union`
- `coalesce` (reducing partition count, no network transfer)
- Broadcast joins

The DataFrame API

DataFrames and Datasets are built on top of RDDs — they add a schema and give Spark enough structure to optimize. The execution model underneath is the same.

```
val orders = spark.read.parquet("orders/")
// orders: DataFrame with schema (id: Long, region: String, amount: Double)

val result = orders
  .filter($"region" === "EU")
  .groupBy($"region")
  .agg(sum($"amount").as("total"))
```

Operations are expressed as **column expressions** (not opaque lambdas) — Spark can inspect, rewrite, and optimize the full plan.

Dataset[T]

`Dataset[T]` wraps a DataFrame with a compile-time type — the schema is enforced by the Scala type system, not just at runtime.

```
case class Order(id: Long, region: String, amount: Double)

val orders: Dataset[Order] = spark.read.parquet("orders/").as[Order]

val result: Dataset[(String, Double)] = orders
  .filter(_.region == "EU")       // typed lambda — loses optimizer visibility
  .groupByKey(_.region)
  .mapValues(_.amount)
  .reduceGroups(_ + _)
```

Typed lambdas (`_.region == "EU"`) are opaque to the optimizer — prefer column expressions (`$"region" === "EU"`) even on `Dataset[T]` to keep the optimizer in the loop.

Spark SQL

`spark.sql` accepts a plain SQL string and parses it into the same logical plan as the DataFrame API.

```
orders.createOrReplaceTempView("orders")

val result = spark.sql("""
  SELECT region, sum(amount) AS total
  FROM orders
  WHERE region = 'EU'
  GROUP BY region
""")
```

SQL strings are parsed into the same logical plan as the equivalent DataFrame chain — same optimizer, same physical plan, same generated bytecode.

One plan, three syntaxes

All three APIs compile to the same **unresolved logical plan**:

API	**Use when**
DataFrame	Programmatic composition, conditional logic, reusable plan fragments
Dataset[T]	Domain model already exists, type safety at boundaries matters
Spark SQL	Ad-hoc exploration, non-engineer authors, readability

Choosing between them is a matter of **ergonomics**, not performance. The same unresolved logical plan is produced regardless of which syntax you use — we will see how it gets optimized next.

Scala vs Python

Both languages have full Spark APIs. The trade-offs are real.

PySpark (Python):

- Larger data science ecosystem — pandas, scikit-learn, matplotlib
- Easier onboarding, Jupyter-native workflow
- Python UDFs: cross-process serialization via pickle — 10–100× slower than JVM code
- Pandas UDFs (Arrow-based): much faster, still cross-process
- No compile-time column checking — `$"reigon"` fails at runtime

Spark with Scala:

- JVM-native — no process boundary, no serialization overhead
- UDFs stay inside the JVM, code generation uninterrupted
- Compile-time type checking on the JVM side
- Same JVM as the executor: jar deployed, no language bridge

Scala vs Python

Rule of thumb: use Python for exploration and pipelines that stay in SQL/DataFrame; use Scala when you need custom UDFs, tight performance, or type safety at scale.

Query Execution Models

How fast can a CPU **actually** process data once it hits memory?

Pull-based execution

The consumer drives execution. Each operator exposes a `next()` method; the root calls `next()` on its child, which calls `next()` on its child, and so on until a row surfaces from the scan.

```
Aggregate.next()
  └─ Filter.next()
       └─ Scan.next()    ← produces one row
```

Control flows **top-down** (the request) and data flows **bottom-up** (the row). Every row crosses every operator boundary as a separate virtual function call — at 100M rows × 5 operators that is 500M dispatches just for routing.

This is the Volcano model (Graefe, 1994), used by most databases until the 2010s and by Spark's pre-2.0 RDD API.

Push-based execution

The producer drives execution. Each operator implements `produce()` / `consume(row)`. The scan calls `consume(row)` on its parent, which immediately calls `consume(row)` on its parent, and so on.

```
Scan.produce()
  → Filter.consume(row)
       → Aggregate.consume(row)    ← row processed end-to-end
```

Control and data both flow **bottom-up** in one pass. There are no `next()` calls crossing operator boundaries — the whole pipeline becomes a single nested call stack, which the JIT can inline into one tight loop.

Spark's Whole-Stage Code Generation uses push semantics internally, even though the external DataFrame API remains pull-style at the stage boundary.

Pull = iterators, push = continuations

These are not new ideas — they are standard programming abstractions.

Pull maps directly to the **iterator** pattern:

```
trait Iterator[A] { def hasNext: Boolean; def next(): A }
```

The caller decides when to consume the next element. Scala's `Iterator`, Java's `Iterator`, Python's generators — all pull.

Push maps directly to **continuation-passing style** (CPS):

```
// instead of returning a value, call the next function with it
def process(row: Row, k: Row => Unit): Unit = k(transform(row))
```

`consume(row)` is a continuation — it tells the operator "here is your input, keep going." The pipeline is a chain of nested continuations, which is exactly what the JIT sees and inlines.

Pull = iterators, push = continuations

The duality is fundamental: any pull pipeline can be mechanically rewritten as a push pipeline by converting `next()` returns into `consume()` calls — which is precisely what Spark's code generator does at compile time.

Vectorized execution

Process a **batch** of rows at a time (typically 1024–8192) rather than one row.

Each operator receives a **column batch** — arrays of values for each column — and applies the operation to the whole array using tight loops.

Tight loops = **SIMD**: the CPU can apply the same operation to 4–16 values in a single instruction (AVX2, AVX-512).

Cache behavior improves dramatically: a 1024-row int column fits in 4 KB — one cache line.

ClickHouse and DuckDB are built on this model. Spark's Parquet reader uses vectorized decoding.

Compiled / JIT execution

Rather than interpreting a plan at runtime, **generate and compile native code** for the specific query.

The generated code is a single tight loop over the data, with no virtual calls, no boxing, no operator boundaries visible to the CPU.

Spark implements this as **Whole-Stage Code Generation** (Tungsten), introduced in Spark 2.0. The query plan is compiled to a single Java/JVM bytecode class at runtime.

Example: `SELECT sum(price * quantity) FROM orders WHERE region = 'EU'`

Generated code applies filter, multiply, and aggregate in one pass — no intermediate collections.

What this means in practice

Model	Unit	Virtual calls	SIMD	Best for
Pull / Volcano	1 row	Many	None	Simplicity
Vectorized	1K rows	Few	Yes	I/O bound, analytics
JIT / compiled	All rows	None	Partial	CPU bound, aggregations

Spark combines: **vectorized Parquet reading** (IO layer) + **compiled execution** (operator layer). The two complement each other.

The Catalyst Optimizer

The optimizer is why
Spark SQL is faster
than hand-written RDDs.

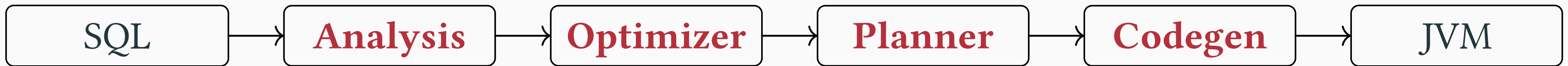
What Catalyst is

Catalyst is Spark's query optimizer. It takes a **logical plan** (what you asked for) and produces a **physical plan** (how to execute it).

It is implemented in Scala using **algebraic trees** and **pattern-matching rules**. Each optimization pass is a function `Tree → Tree`. New rules can be added by plugins.

Catalyst runs entirely on the driver, before any data moves.

The four-phase pipeline



- **Analysis**: resolve column names and types; catch schema errors before execution
- **Logical optimizer**: rule-based rewrites — predicate pushdown, constant folding, subquery elimination
- **Physical planner**: select operators and join strategies; cost-based when statistics are available
- **Code generation**: emit a single JVM class per pipeline stage (Tungsten)

Logical plan

The logical plan is a tree of **relational algebra** operators:

- `Filter(condition, child)`
- `Project(expressions, child)`
- `Join(left, right, condition, joinType)`
- `Aggregate(groupingKeys, aggregates, child)`
- `Scan(relation, filters, projections)`

The logical plan is **schema-aware** but **implementation-agnostic**. It does not know whether data is in Parquet or CSV, or how many partitions it has.

Logical optimization rules

Predicate pushdown: move `Filter` nodes as close to the `Scan` as possible.

Before: `Filter(age > 30, Join(users, orders, ...))` After: `Join(Filter(age > 30, users), orders, ...)`

Less data enters the join — this is typically the single biggest win.

Projection pruning: drop columns that are never referenced downstream. Avoids deserializing unused bytes from Parquet row groups.

Constant folding: evaluate `2 + 3` at plan time, not at row evaluation time.

Boolean simplification: `x AND TRUE → x`, `x OR FALSE → x`.

Physical planning — join strategies

For each logical `Join`, Catalyst must choose a physical strategy:

Broadcast hash join (BHJ): broadcast the smaller table to all executors, build a hash table in memory, probe with every row of the larger table. No shuffle. Requires the smaller side to fit in memory (`spark.sql.autoBroadcastJoinThreshold`, default 10 MB).

Sort-merge join (SMJ): shuffle both sides so matching keys land on the same partition, sort both sides, then merge. Scales to any data size. Requires two shuffles if inputs are not already sorted.

Shuffle hash join (SHJ): shuffle both sides like SMJ, but build a hash table instead of sorting. Works when one side fits in partition memory.

Physical planning — cost-based optimization

When statistics are available (via `ANALYZE TABLE` or collected during planning), Catalyst uses **cost-based optimization** (CBO) to:

- Estimate the size of each intermediate result
- Choose the probe side vs build side of a hash join
- Decide whether to broadcast based on estimated size (not just table-level stats)
- Reorder joins in a sequence to minimize intermediate data

CBO requires column-level statistics. Without them, Catalyst falls back to heuristics — which is why `ANALYZE TABLE` matters on large datasets.

Reading a query plan

`df.explain(mode = "formatted")` shows:

```
*(3) HashAggregate(keys=[region], functions=[sum(revenue)])
+- Exchange hashpartitioning(region, 200)          // shuffle boundary
   +- *(2) HashAggregate(...)                      // partial agg, same stage
      +- *(2) Filter (price > 0)
         +- FileScan parquet [region,price,qty]
            PushedFilters: [GreaterThan(price,0)] // sent to Parquet reader
```

Key things to identify:

- `*()` around an operator ⟶ inside a Whole-Stage CodeGen stage
- `Exchange` ⟶ shuffle boundary between stages
- `PushedFilters` ⟶ filters sent to the Parquet reader (skip row groups)
- `BroadcastExchange` ⟶ broadcast join, no shuffle for the smaller side

Reading a query plan — join identification

```
*(5) SortMergeJoin [user_id], [user_id], Inner
:- *(2) Sort [user_id ASC]
:  +- Exchange hashpartitioning(user_id, 200)
:     +- *(1) FileScan parquet orders
+- *(4) Sort [user_id ASC]
   +- Exchange hashpartitioning(user_id, 200)
      +- *(3) FileScan parquet users
```

Two `Exchange` nodes ⟶ two shuffles. Both sides are being repartitioned on `user_id` so matching keys co-locate. If `users` were small enough, Catalyst would replace this with a `BroadcastHashJoin` and eliminate both shuffles.

Tungsten

Spark was spending half its time
on **GC** and **boxing**.
Tungsten fixed both.

The problem Tungsten solves

Before Spark 1.5, Spark stored data as JVM objects:

- Every `Row` was a heap-allocated object with header overhead (16 bytes minimum)
- Column values were boxed: an `Integer` wraps an `int`, adding 16 bytes of overhead
- The garbage collector had to scan millions of short-lived objects between stages
- GC pauses of 10–30 seconds were common on large shuffles

Tungsten is Spark's answer: **manage memory manually**, store data in a compact binary format, generate native JVM bytecode instead of interpreting plans.

Off-heap memory management

Tungsten uses `sun.misc.Unsafe` to allocate memory **outside the JVM heap**:

- The GC never sees this memory — no GC pressure, no GC pauses
- Data is stored in a **compact binary row format** (UnsafeRow)
- Rows can be compared and sorted without deserialization
- Memory can be spilled to disk without serialization overhead

`UnsafeRow` layout: a fixed-length null bitmap + fixed-length values section + variable-length data appended at the end. The whole row is contiguous bytes.

Whole-Stage Code Generation

For each query, Tungsten generates a **single Java class** that implements the entire pipeline.

The generated class:

- Iterates over input rows in a tight loop
- Has all operators inlined (no virtual calls)
- Uses primitive types directly (no boxing)
- Is compiled by the JVM JIT into native machine code

The compilation happens at query planning time on the driver. The generated bytecode is sent to executors as part of the task.

You can see the generated code with `df.queryExecution.debug.codegen()`.

What gets code-generated (and what doesn't)

Operators inside a `*(N)` block in the plan are fused into one generated stage:

- Filter, Project, HashAggregate, BroadcastHashJoin, Sort

Operators that **break** code generation:

- `Exchange` (shuffle — data must leave the process)
- `HashAggregate` (the partial agg is code-gen'd; the merge is a separate stage)
- Python UDFs — they cross the JVM/Python boundary, breaking the generated loop

Python UDFs are the single biggest Tungsten performance killer. They force row-by-row serialization to a Python process and back, bypassing all code generation.

Impact

Benchmarks comparing Spark before and after Tungsten show 5–10× speedups on aggregation-heavy workloads and 2–3× on join-heavy ones.

The gains come from eliminating:

- Deserialization to `Row` objects (use `UnsafeRow` directly)
- GC pressure on intermediate results
- Virtual dispatch overhead in the Volcano iterator chain
- Redundant serialization across operators in the same stage

Partitioning, Shuffles & Tuning

Every shuffle is a network call
for every row.
Minimize them.

What a shuffle is

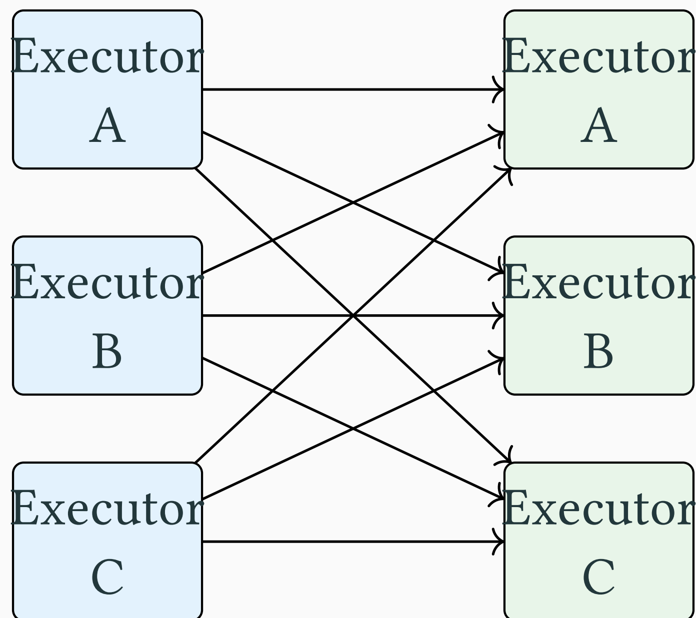
A shuffle is the process of redistributing data across partitions so that rows with the same key land on the same partition.

Mechanically:

1. Each task in stage N writes its output to local disk, partitioned by the target partition key (the **shuffle write**)
2. Each task in stage N+1 fetches the relevant partition files from all executors (the **shuffle read**)
3. The fetched data is sorted, merged, and processed

Shuffles are expensive because they involve **serialization**, **disk I/O**, and **network transfer** for every row.

What a shuffle is



Every executor writes to every other executor — N^2 data flows for N executors.

Partition count: the key knob

Default: `spark.sql.shuffle.partitions = 200`

Too few: each task processes too much data → spills to disk, OOM risk, long task time.

Too many: overhead of scheduling, launching, and tracking thousands of tiny tasks dominates computation time.

Rule of thumb: target **100 MB–1 GB of data per partition after the shuffle**. For a 100 GB dataset, `100–1000` partitions is reasonable.

With **Adaptive Query Execution** (AQE, Spark 3.0+), Spark can coalesce small post-shuffle partitions automatically — set `spark.sql.adaptive.enabled = true` and let it tune partition count.

Partition skew

Skew occurs when a small number of partitions hold a disproportionate share of the data — e.g., 90% of orders have `region = NULL`.

Symptoms: one long-running task while all others finish, OOM errors on specific executors, "straggler" stage in the UI.

Solutions:

Salting: add a random suffix to the skewed key to spread it across `N` partitions. Requires a corresponding re-aggregation step after the join/agg.

AQE skew join optimization: AQE detects skewed partitions after the shuffle and automatically splits them into smaller sub-tasks. Enabled with `spark.sql.adaptive.skewJoin.enabled = true`.

Filter and handle separately: isolate the dominant key, process it apart, union back.

Avoiding shuffles — broadcast joins

If one side of a join fits in memory on each executor, broadcast it:

```
import org.apache.spark.sql.functions.broadcast

orders.join(broadcast(countries), "country_code")
```

No shuffle on either side. The `countries` table is serialized on the driver and sent to every executor once.

Auto-broadcast threshold: `spark.sql.autoBroadcastJoinThreshold` (default: 10 MB). Increase for large in-memory clusters:

```
spark.sql.autoBroadcastJoinThreshold = 100MB
```

Force-disable broadcasting (e.g., when the table is large but Catalyst underestimates): `broadcast(df)` explicitly, or set threshold to `-1`.

Caching

`df.cache()` / `df.persist()` stores the DataFrame's data in memory across actions.

Use when: the same DataFrame is consumed by multiple downstream actions (e.g., a filter result used in two different joins).

Do not use when:

- The DataFrame is used only once — caching consumes memory and adds a write step
- The DataFrame does not fit in memory — spilling to disk can be slower than recomputing from Parquet with predicate pushdown

Cache is **lazy**: `df.cache()` registers intent; the data is actually cached on the first action that touches it.

Unpersist explicitly with `df.unpersist()`. Spark's LRU eviction policy is not always predictable in complex pipelines.

Adaptive Query Execution (AQE)

AQE re-optimizes the query **at runtime**, after each shuffle materializes:

- **Partition coalescing**: merge empty or tiny post-shuffle partitions into fewer tasks
- **Skew join handling**: split large skewed partitions, replicate the matching side
- **Join strategy switching**: if runtime stats reveal one side is smaller than estimated, switch from SMJ to BHJ mid-execution

Enable with:

```
spark.sql.adaptive.enabled = true
spark.sql.adaptive.coalescePartitions.enabled = true
spark.sql.adaptive.skewJoin.enabled = true
```

AQE does not replace understanding partitioning — it reduces the cost of getting it wrong.

Anatomy of a slow job — checklist

Symptom	Likely cause
One task takes 10× longer than others	Partition skew — salt the key or enable AQE skew join
OOM on executors	Partition too large — increase shuffle partitions or broadcast smaller side
Hundreds of tiny tasks	Too many shuffle partitions — reduce or enable AQE coalescing
Full-table scans on Parquet	Missing predicate pushdown — check `PushedFilters` in the plan
Unexpected SMJ instead of BHJ	Table stats not collected — run `ANALYZE TABLE`
Python UDF stage takes 10× longer	Cross-process serialization — replace with built-in functions

Wrap-Up

A slow Spark job is almost always
a **shuffle you didn’t need**
or a **filter that arrived too late**.

Key vocabulary

Term	Definition
Transformation	Lazy operation on a DataFrame; builds the plan, executes nothing
Action	Triggers execution; returns a result or writes output
Stage	Maximal sequence of operations that can run without a shuffle
Task	One stage applied to one partition; the unit of parallelism
Shuffle	Redistribution of data across partitions; always crosses the network
Volcano model	Iterator-based execution: one tuple at a time via `next()`
Vectorized execution	Batch-at-a-time using column arrays; enables SIMD
Whole-Stage CodeGen	Tungsten's JIT: entire pipeline compiled into one Java class
Catalyst	Spark's optimizer: logical plan $\longrightarrow$ physical plan via rule-based rewrites
BHJ	Broadcast hash join: no shuffle; smaller side sent to all executors
SMJ	Sort-merge join: both sides shuffled and sorted; works at any scale
Predicate pushdown	Filter evaluated at scan time, skipping row groups in Parquet
Projection pruning	Drop unused columns before deserialization

Key vocabulary

AQE	Adaptive Query Execution: re-optimizes at runtime using shuffle statistics
Skew	A few partitions hold most of the data; causes straggler tasks

One sentence to remember

The Catalyst optimizer rewrites your query for free — but only if you give it **structured expressions** it can inspect. Push filters early, broadcast small tables, and keep Python out of the hot path.

Further reading

- **Spark: The Definitive Guide** — Chambers & Zaharia (O'Reilly, 2018) — Chapters 15–19 on query execution internals
- Project Tungsten: Bringing Spark Closer to Bare Metal — Databricks blog, 2015
- Spark SQL: Relational Data Processing in Spark — Armbrust et al., SIGMOD 2015
- Cost-Based Optimizer in Apache Spark 2.2 — Databricks blog, 2017
- Apache Spark Adaptive Query Execution — VLDB 2020

Lab

https://vbergeron.github.io/data-processing-at-scale/lab-2.2-spark-query-plans.pdf