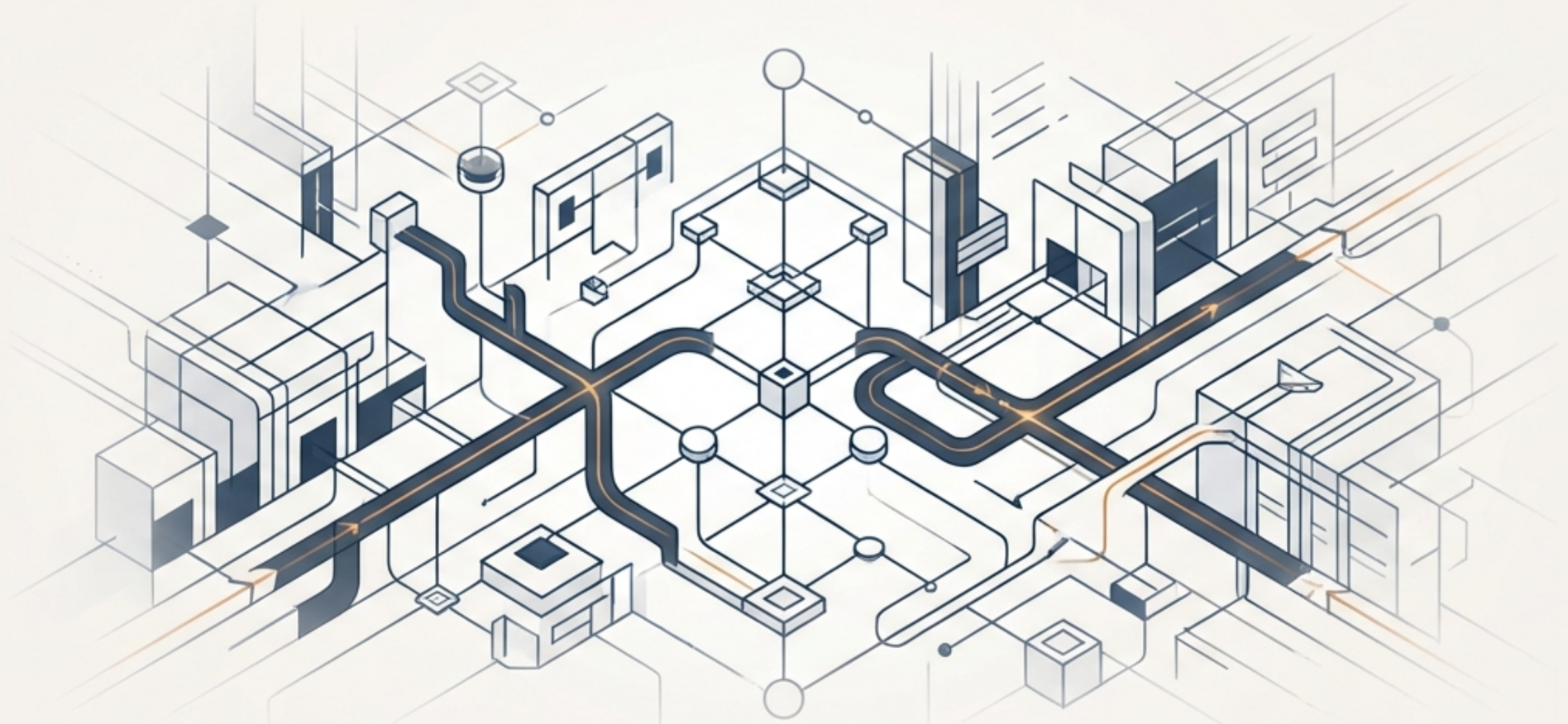
Apache Spark Architecture: Under the Hood

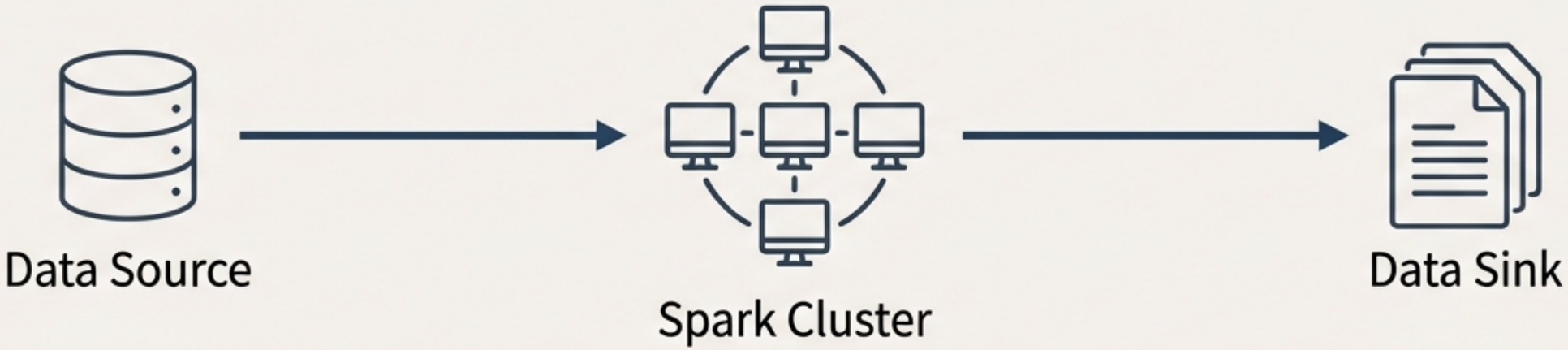
Tracing the life of a query from code to cluster.



Spark is a Unified Computing Engine, Not a Storage System.

Based on the official documentation, Spark's job is to:

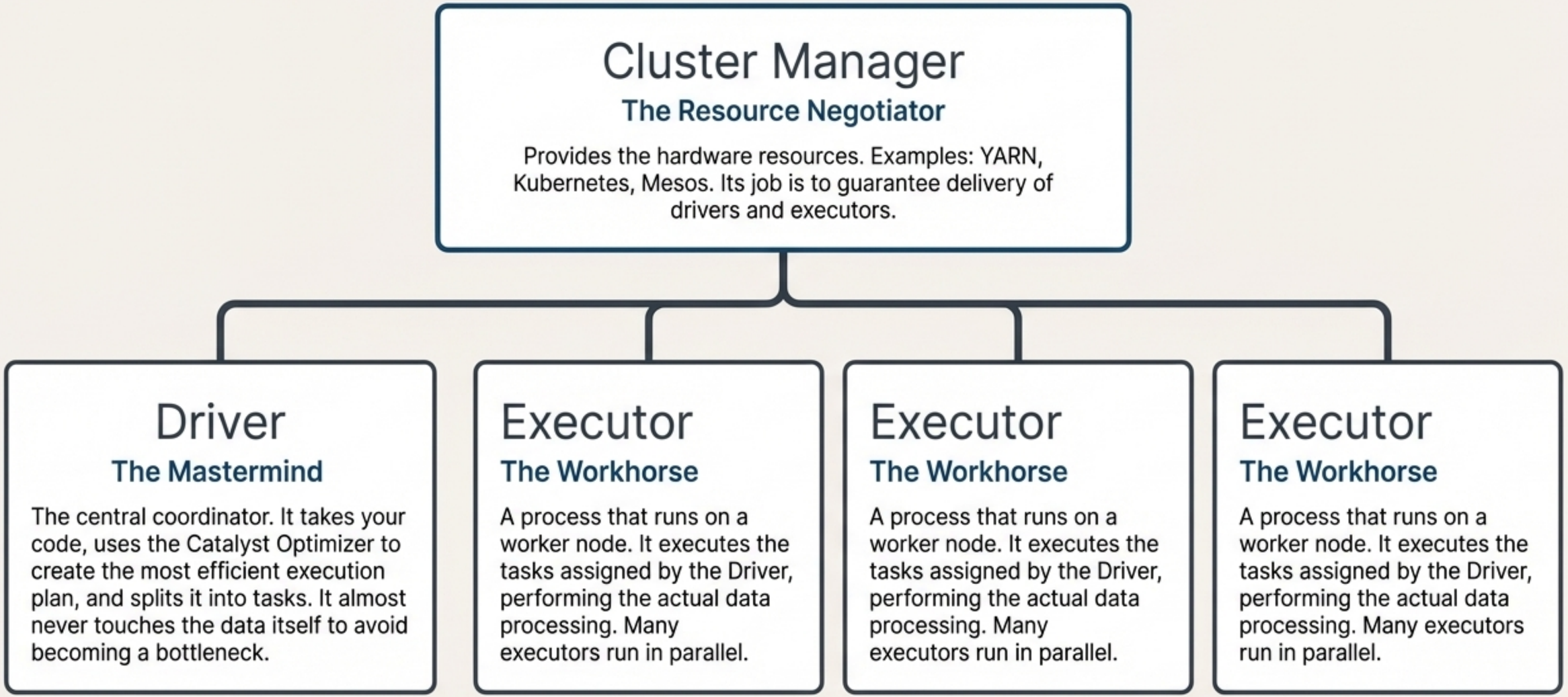
1. **Ingest** data from a source system.
2. **Process** it in parallel across a computer cluster.
3. **Output** the results to a destination.



Key Insight

Spark shines when data is too big for one machine. For small data which fits your personal computer, simpler tools like Pandas in Python are often better.

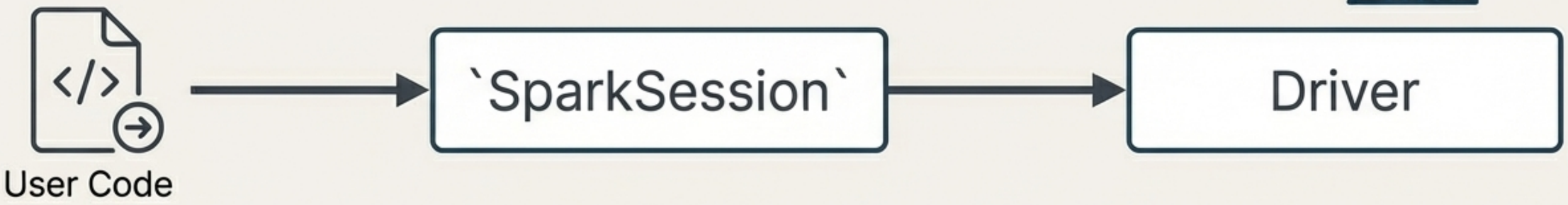
The Core Components: An Architectural Triad



```
df = spark.read.csv(...)
```

A Query is Born, But Execution Waits

- The journey starts when a user submits code through a `SparkSession`—the main entry point to Spark.
- The Driver receives the code but does not execute it immediately. This is **Lazy Evaluation**.



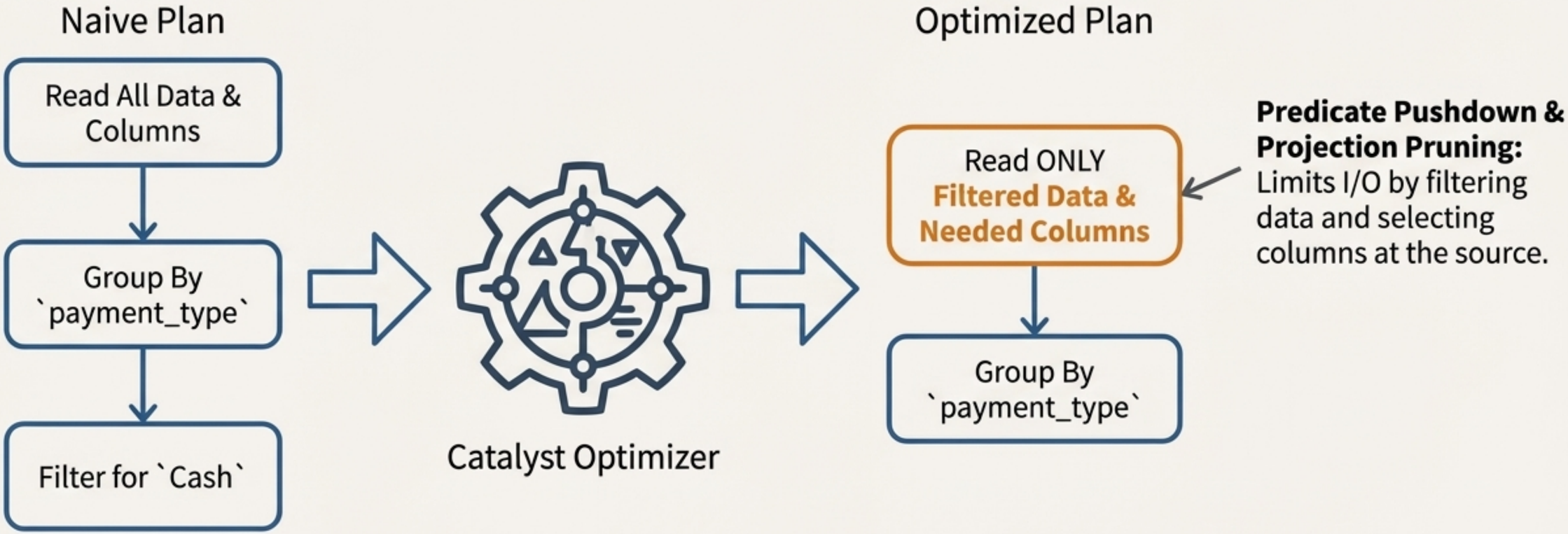
Key Insight

Spark doesn’t run anything until an ‘action’ is called. It waits to see the full plan first, allowing it to perform powerful optimizations.

```
... .filter(col("payment_type") == "Cash")
```

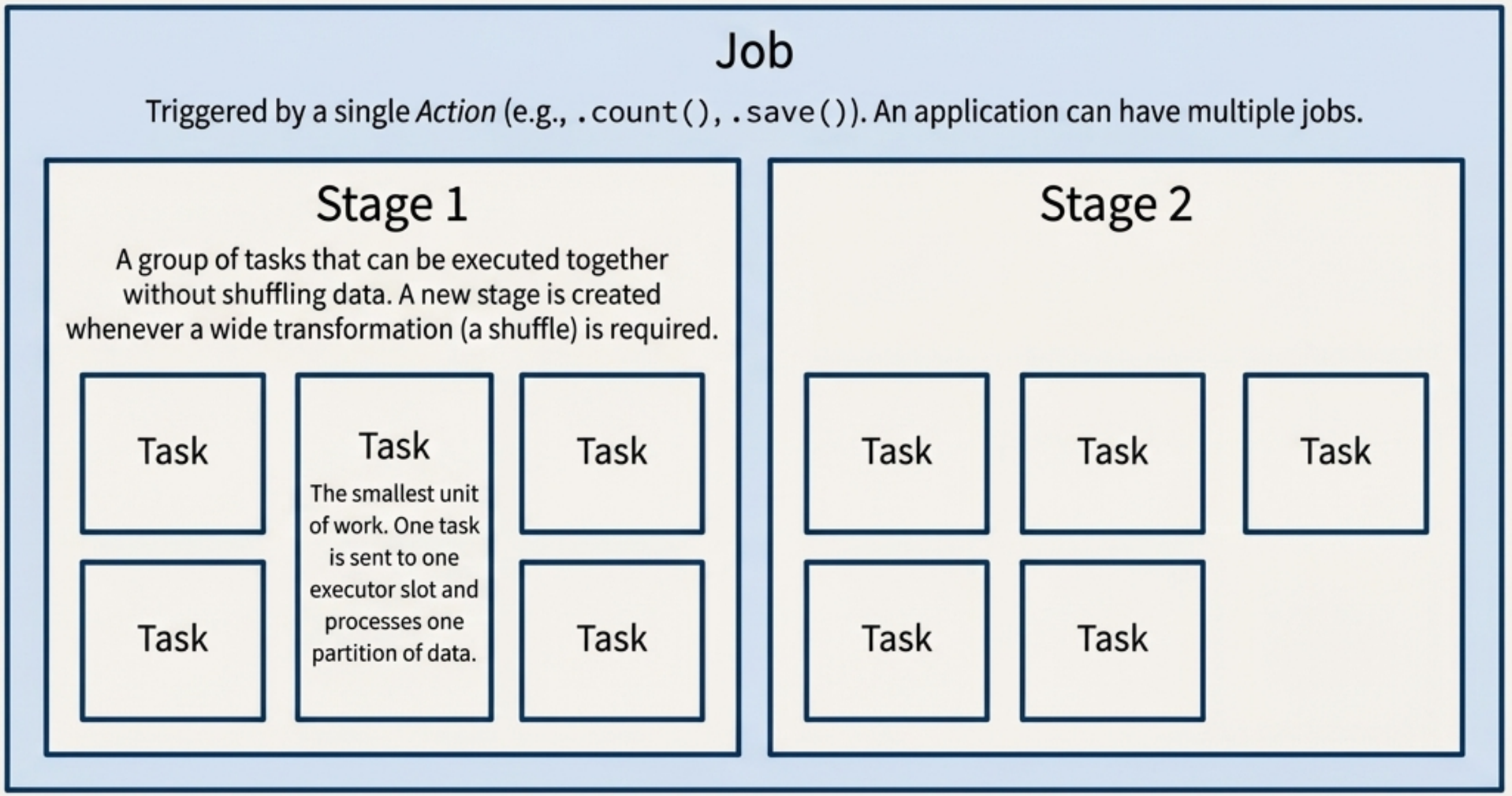
The Mastermind's Plan: The Catalyst Optimizer

The Driver uses the Catalyst Optimizer to analyze the user's transformations and rewrite them for maximum efficiency.



Deconstructing the Plan into Jobs, Stages, and Tasks

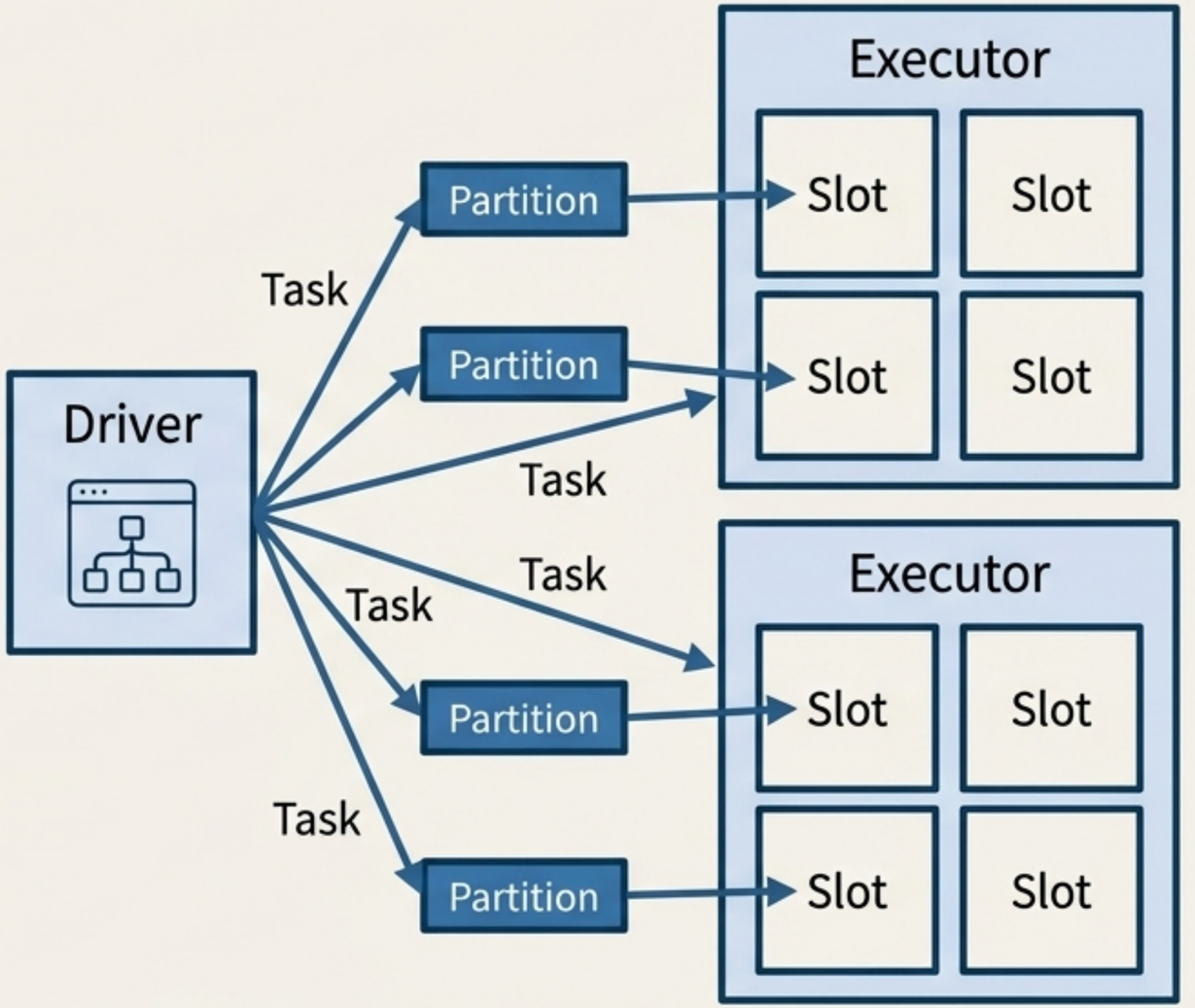
The **Driver** translates the optimized logical plan into a physical execution hierarchy:



`.show()`

The **Action** is Called: **Work is Distributed**

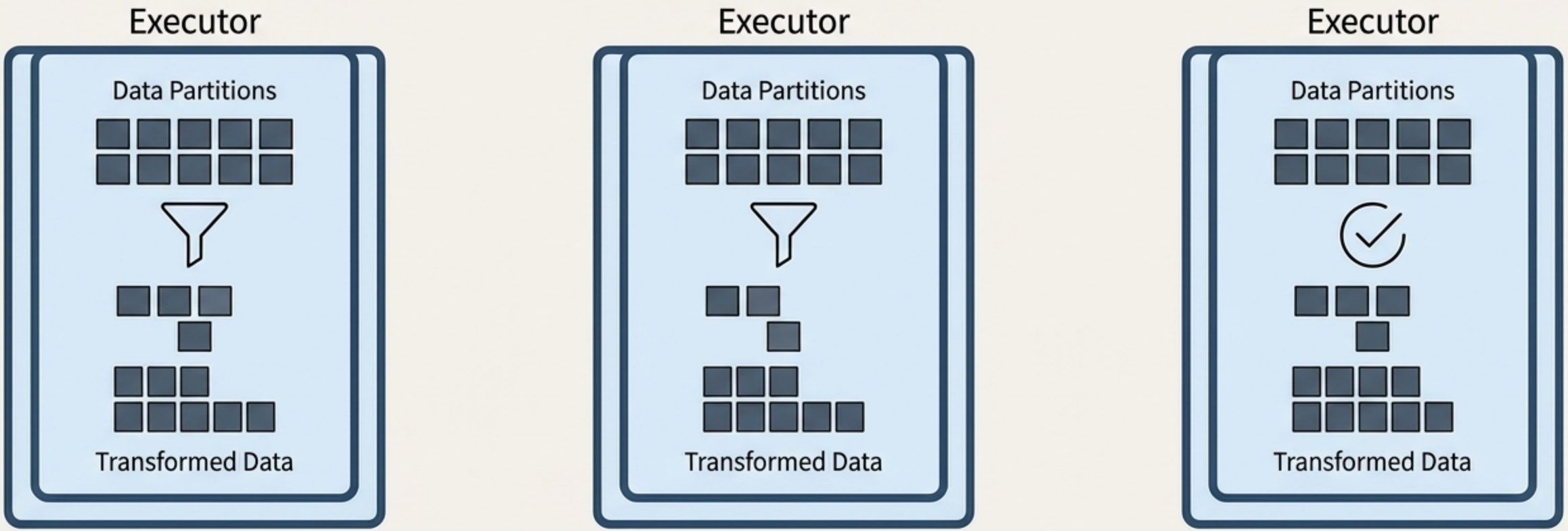
- The Driver sends tasks to the Executors.
- The source data is logically split into **Partitions**.
- Each **Task** operates on exactly one partition of data.
- Each Executor has multiple “slots” (equal to its CPU cores) and can run that many tasks in parallel. With 2 workers of 4 cores each, 8 tasks can run in parallel.



`.filter(col("passenger_count") > 1)` or `.select("payment_type")`

Efficient & Independent: The Power of Narrow Transformations

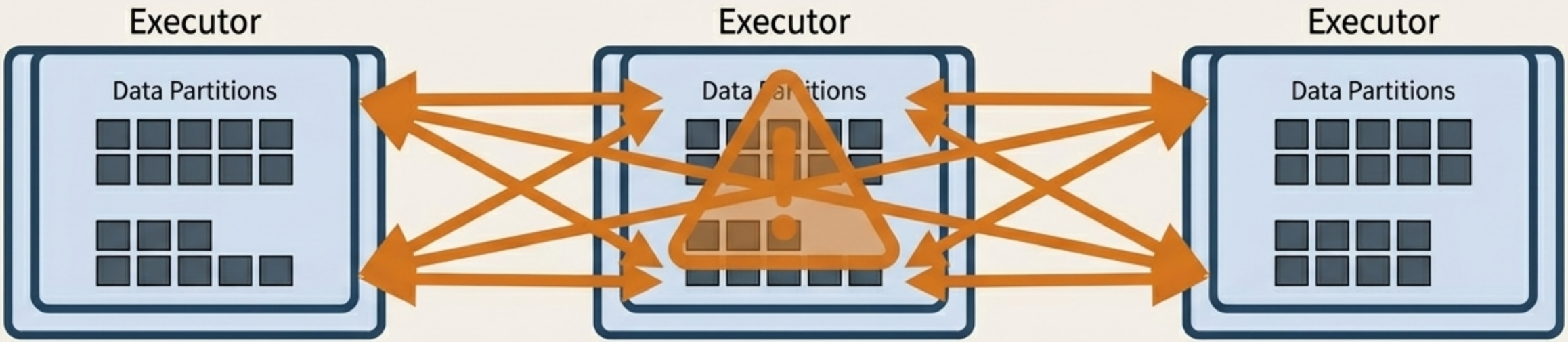
These are operations where each partition can be transformed without any knowledge of other partitions. Executors can work on their data independently, without needing to communicate with each other. They are fast, efficient, and do not trigger a shuffle.



```
.groupBy("payment_type") or .orderBy("trip_distance")
```

The **Bottleneck**: When Executors Must Communicate via a Shuffle

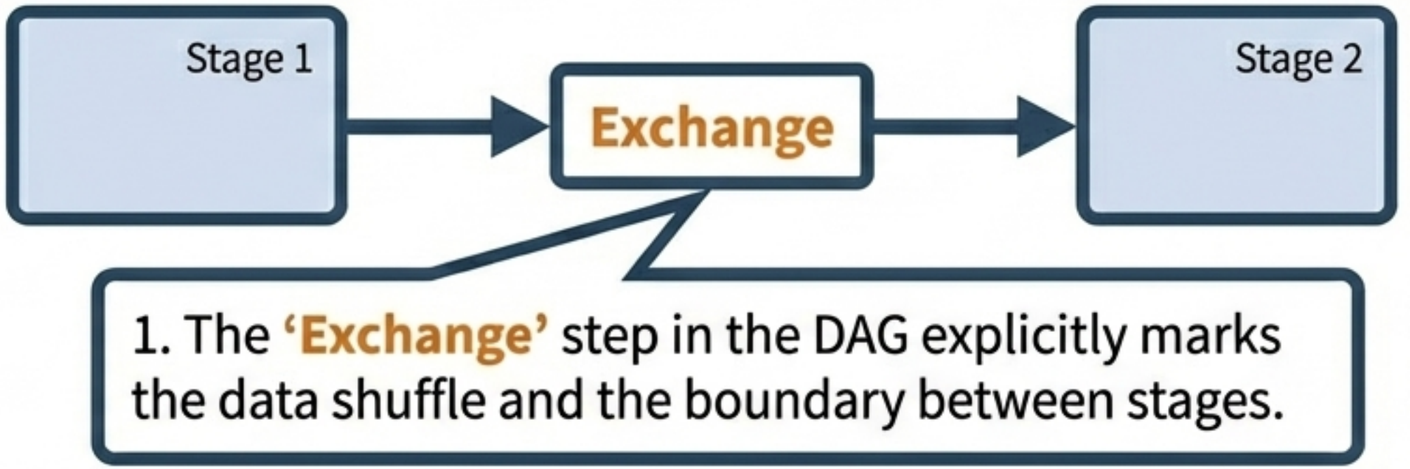
Wide transformations require data from other partitions. To perform a `groupBy`, an executor needs all records for a given key (e.g., all `payment_type='Cash'`). This requires a massive, expensive exchange of data between all executors, known as the **Shuffle**.



Key Insight
Shuffling requires writing data from memory to disk, sending it over the network, and reading it back into memory on another machine. It is **the most common performance killer in Spark**.

Reading the Signs: How to Spot a Shuffle in the Spark UI

The shuffle is what defines the boundary between stages. stages. You can diagnosed it in the Spark UI by looking for:



1. The **‘Exchange’** step in the DAG explicitly marks the data shuffle and the boundary between stages.

Stage ID	Description	Duration	Tasks	Shuffle Write Size	Shuffle Read Size
Stage 1		00:07:39	1/2	50.0 GB	50.0 GB
Stage 2		00:07:24	3/2	50.0 GB	50.0 GB

2. Key metrics like **‘Shuffle Write Size’** and **‘Shuffle Read Size’** quantify the cost. Large values here indicate an expensive shuffle.

The Shuffle’s True Cost is Determined by Data Cardinality

The volume of shuffled data depends not just on the transformation, but on the data itself.

Low Cardinality `groupBy`

```
.groupBy("payment_type")
```

Grouping by a column with few unique values (`'Cash'`, `'Credit'`) results in a small, fast shuffle.

125 KB

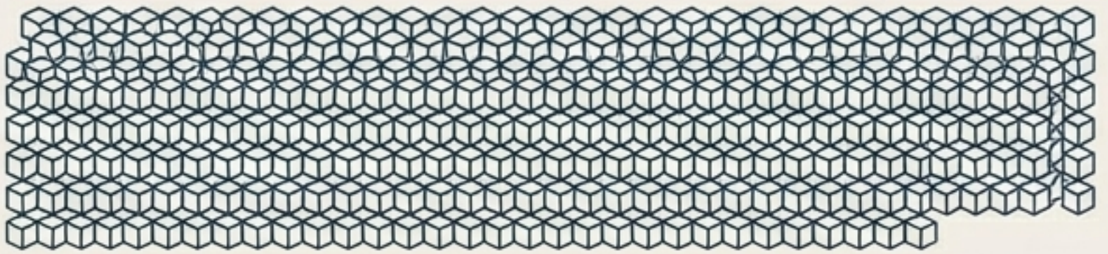
High Cardinality `groupBy`

```
.groupBy("pickup_date_time")
```

Grouping by a column with many unique values (like a timestamp) forces a massive data exchange.

673 MB

Over 5,000x larger



Spark’s Built-in Intelligence: Tungsten and AQE

Beyond Catalyst, other engines work to make Spark fast:

Project Tungsten

A “bare-metal” optimization engine. It generates optimized bytecode on the fly **(Whole-Stage Code Generation)** and manages memory directly, bypassing the JVM for significant speed gains.

Adaptive Query Execution (AQE)

Provides “mid-flight corrections.” Spark re-optimizes the plan *during* execution based on real data statistics, for example, by dynamically collapsing shuffle partitions from the default of 200 to a more optimal number.

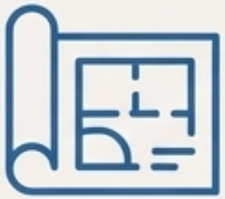
API Matters: Low-Level Control vs. High-Level Optimization

	RDD (Resilient Distributed Dataset)	DataFrame / Dataset
API Style	Functional. Low-level control over "how" to execute.	Declarative/SQL. High-level definition of "what" to compute.
Optimization	Manual. Considered a "black box" to Spark. **You forgo all benefits of the Catalyst Optimizer.**	Automatic. **Fully optimized by Catalyst.** Spark understands the query's intent and can rewrite it.
Use Case	Legacy code or niche use cases requiring fine-grained control over physical data distribution.	The modern standard for virtually all data processing tasks.

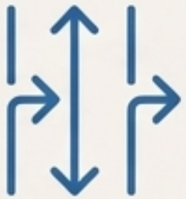
Key Insight

Always prefer the DataFrame/Dataset API. You get immense performance benefits from Catalyst for free. Using RDDs means you are taking on the role of the optimizer yourself.

The Architect’s View: Four Core Principles



Plan First, Execute Later: Lazy evaluation and the Catalyst Optimizer are the foundation of Spark’s performance. Trust them.



Parallelism is Everything: Work is executed via Tasks on Partitions across many Executors. Understand this distribution model.

Avoid the Shuffle (When Possible): Wide transformations are the primary performance bottleneck. Understand when and why they happen by checking the Spark UI.

Use High-Level APIs: The DataFrame API unlocks Spark’s full optimization potential. The Spark UI unlocks your ability to see it in action.

The Journey Continues in Your Own Code

The best way to learn is to build. Open the Spark UI on your next job and try to trace the journey of your query. Identify the jobs, find the stage boundaries, and measure the shuffle. This is how understanding becomes expertise.