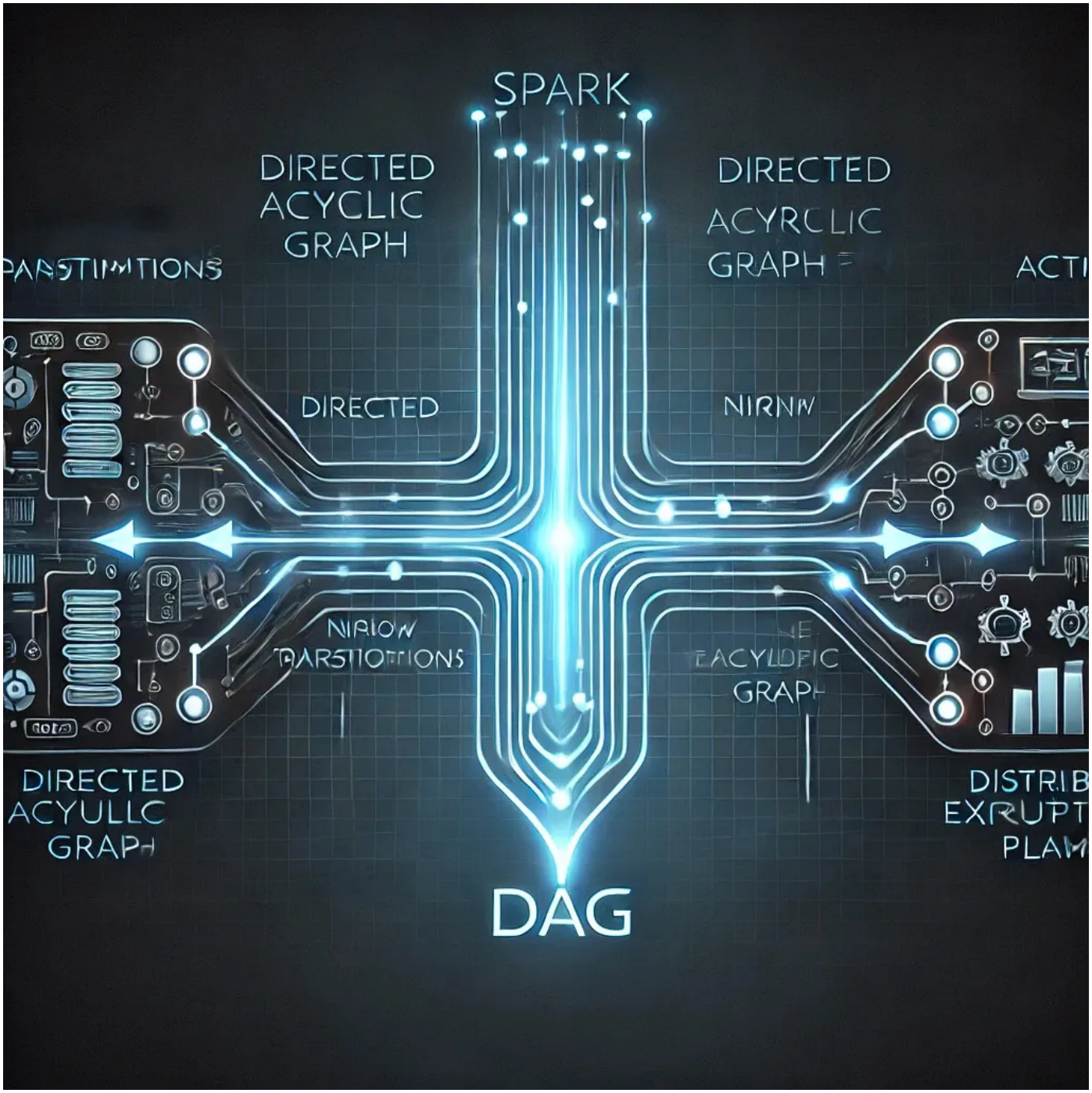
Dev Genius

Mastering Spark DAGs: The Ultimate Guide to Understanding Execution

Mohit Joshi Follow 11 min read · Feb 7, 2025



If you've ever worked with **Apache Spark**, you've probably heard about **DAGs (Directed Acyclic Graphs)** but do you really know how they supercharge Spark's performance? Whether you're a **beginner trying to understand Spark execution** or an **experienced developer optimizing your workflows**, mastering DAGs is key to writing efficient Spark applications.

At its core, Spark transforms your **high-level code** into an **optimized execution plan** using a **DAG**, ensuring that data flows smoothly without redundant computations. Behind the scenes, **Catalyst Optimizer** fine-tunes your queries, classifying operations into **narrow and wide transformations**, which are then split into **stages and tasks**. This intelligent orchestration is what makes Spark blazing fast compared to traditional processing engines.

But how exactly does all this happen? How does your PySpark code get converted into an optimized DAG? And why does Spark's DAG-based execution model make it **so much faster than MapReduce**? 🤔

In this article, we'll break it all down step by step, from **logical and physical plan generation** to **DAGScheduler's role in creating stages** and **TaskScheduler's role in executing tasks across the cluster**. By the end, you'll have a crystal-clear understanding of **how Spark executes your code under the hood** and how you can leverage this knowledge to write better, faster Spark applications.

Let's dive in! 🚀

What is a DAG in Spark?

Imagine you're working with a giant dataset, applying transformations step by step filtering, mapping, grouping. Wouldn't it be nice if Spark could remember what you did, optimize it, and only execute when absolutely necessary? That's exactly where a DAG (**Directed Acyclic Graph**) comes in.

> *A **Directed Acyclic Graph (DAG)** is a structure where each node represents a transformed dataset, and the edges between them represent the transformations applied. The "acyclic" part means there are no loops once you perform a transformation, you don't go back; you only move forward.*

Why Does Spark Need a DAG?

Spark follows lazy evaluation, meaning it doesn't execute transformations immediately. Instead, it records them in a DAG. This DAG is only triggered when an action (like `collect()` or `save()`) is encountered.

But why go through all this trouble? Because it's efficient! Instead of executing each transformation one by one and loading data multiple times, Spark optimizes the entire sequence, reducing unnecessary computation and improving performance.

DAG vs. MapReduce and Data Lineage

Before Spark, MapReduce relied on chaining jobs together manually, leading to inefficiencies and redundant data movement. Spark's DAG, on the other hand, builds a logical execution plan, optimizing dependencies and minimizing shuffling.

Another major advantage is data lineage. Since the DAG keeps track of every transformation, Spark can recover lost data if a node fails. Instead of recomputing everything from scratch, it simply traces back to the original dataset and reruns only the affected transformations. This makes Spark highly fault-tolerant compared to traditional MapReduce.

🚀 Inside Spark: How DAGs Are Built & Executed

Spark isn't just any data processing tool it's a distributed computing engine designed to handle massive datasets across multiple nodes. Unlike traditional single-machine execution, Spark breaks down computations into tasks that can run in parallel. But how does it decide what runs where? That's where DAGs come in.

In PySpark, DAGs are built by the driver node, which acts as the brain of the operation. The process involves several key components:

- **DAGScheduler:** Converts high-level transformations into a series of stages.
- **Catalyst Optimizer:** Optimizes queries to ensure efficient execution.
- **TaskScheduler:** Assigns tasks to worker nodes for execution.

The DAG isn't actually constructed when you write transformations like `filter()` or `map()`. Instead, Spark logs these transformations but waits for an action such as `.show()`, `.count()`, or `.write()` to kickstart execution. At this point, the `QueryExecution` component in the driver process takes over and constructs the final DAG, setting the stage for execution.

Below, I've explained step by step how Spark DAGs are built and executed, breaking down each stage of the process for better clarity.

Step 1: From Code to Plan — The Birth of a DAG

Let's start with a simple PySpark code:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("DAG_Example").getOrCreate()
df = spark.read.csv("data.csv", header=True, inferSchema=True)
result = (
    df.filter(df.age > 30)  # Transformation 1
    .groupBy("city")        # Transformation 2
    .agg({"salary": "avg"}) # Transformation 3
    .orderBy("avg(salary)", ascending=False)  # Transformation 4
)
result.show()  # Action
```

When the `.show()` action is triggered, Spark's Query Planner kicks in and constructs the initial logical plan. This plan is raw and unoptimized, meaning it directly reflects the query structure without any performance improvements.

At this stage, Spark simply parses the query and arranges operations in the order they appear. However, it doesn't yet consider aspects like redundant computations, predicate pushdown, or reordering for efficiency. The logical plan is essentially a blueprint of what needs to be done but is not necessarily the best way to execute it.

Here's how the **unoptimized logical plan** looks:

```
Project(name, city, age, salary)
   |
Filter(age > 30)
   |
```

```
Aggregate(city, avg(salary))
   |
Sort(avg(salary))
   |
FileScan (CSV)
```

Step 2: Optimizing the Plan — Catalyst to the Rescue

The raw logical plan we saw earlier isn't efficient it needs some smart tuning. That's where Catalyst Optimizer steps in.

What is Catalyst Optimizer?

Catalyst is Spark's query optimization framework, designed to improve execution efficiency before running the actual computation. It restructures the plan, eliminates inefficiencies, and applies rule-based and cost-based optimizations.

One of the key techniques it applies is **Predicate Pushdown.**

What is Predicate Pushdown?

Normally, when you apply a `filter()`, Spark could read all the data first and then filter it in memory but that would be inefficient. Instead, **Predicate Pushdown moves the filter closer to the data source** (like a CSV, Parquet, or database). This means Spark reads only the necessary data instead of scanning everything.

For example, in our query, `Filter(age > 30)` was initially applied after loading all columns from the CSV. Catalyst moves it directly to the FileScan stage, reducing the amount of data read.

Here's how the **optimized logical plan** looks after Catalyst's magic:

```
Sort(avg(salary))
   |
Aggregate(city, avg(salary))
   |
FileScan (CSV) [Only necessary columns, filtering applied]
```

Now, Spark avoids scanning unnecessary data and improves performance. But this is still a logical plan it describes what needs to be done, not how. That's where the physical plan comes in.

Step 3: From Plan to Execution — The Physical Blueprint

Even with an optimized logical plan, Spark still doesn't know how to execute it efficiently across the cluster. That's why it needs to generate a physical plan, which acts as the execution blueprint.

At this stage, Spark decides execution strategies, such as how to distribute computations and optimize resource usage. The **QueryPlanner**, the same component that created the initial logical plan, now generates a physical plan.

Here's how the physical plan looks:

```
*(3) Sort ['avg(salary)] DESC
   |
*(2) HashAggregate (groupBy city, compute avg(salary))
```

```
   |
Exchange (Shuffling: hash partitioning on city)
   |
*(1) FileScan csv (Filtered age > 30)
```

Key Differences Between Logical and Physical Plans

> **Logical Plans** describe what needs to be computed.
>
> **Physical Plans** define how it will be computed.

In some cases, **multiple physical plans** can be generated, and **Spark's Cost Model** selects the most efficient one. However, this doesn't always happen it depends on factors like data size, available resources, and query complexity.

When Does Spark Generate Multiple Physical Plans?

1. **Multiple Join Strategies Available** → Spark may generate multiple plans using Broadcast Join, Sort-Merge Join, or Shuffle Hash Join and choose the most cost-efficient one.
2. **Predicate Pushdown & Column Pruning** → Different pushdown strategies can lead to alternative plans, depending on the data source.
3. **Exchange and Partitioning Strategies** → Spark may consider different ways to shuffle or repartition data before execution.
4. **Data Skew Handling** → If Spark detects skew, it might generate multiple strategies to optimize execution.

If none of these conditions apply, Spark may proceed with just one Physical Plan. The Cost Model evaluates plans based on estimated execution time, I/O costs, and shuffle overhead before selecting the best one.

At this point, the DAG is almost ready for execution! The next step? Breaking it down into stages and tasks that can run in parallel across the cluster. 🚀

Step 4: Breaking It Down — Enter the DAGScheduler

Now that Spark has a physical plan, it's time to break it down into manageable execution units. This is where the DAGScheduler steps in.

What is DAGScheduler?

DAGScheduler is a Spark Driver component responsible for:

- Creating **Jobs** when an action (like `count()`, `show()`, or `collect()`) is triggered.
- Splitting the job execution into **Stages**, ensuring efficient execution across the cluster.
- Managing dependencies between different stages and optimizing the execution plan.
- Handling **task failures** and re-executing them when necessary.

Jobs in Spark

Each Spark **Job** corresponds to an **action** in the program. A single action

usually triggers **one Job**, but in some cases, multiple Jobs are created depending on the dependencies between transformations and the need for recomputation.

For example:

- If multiple actions are performed on the same DataFrame (e.g., `df.count()` followed by `df.show()`), Spark may create separate Jobs for each action.
- When an **RDD or DataFrame is persisted** (`persist()` or `cache()`), Spark avoids recomputation. However, if it is not persisted and the same dataset is used in multiple actions, Spark recomputes the lineage, triggering multiple Jobs.
- In cases where **multiple shuffles occur** (e.g., performing `groupBy()` followed by a `join()`), Spark may need to execute separate Jobs to handle the shuffle dependencies efficiently.

Understanding how Spark creates Jobs is crucial for optimizing performance and reducing redundant computations.

Narrow vs. Wide Transformations

The way Spark splits execution into stages depends on transformation types, which directly impact how **Jobs** are structured:

- 🤏**Narrow Transformations** (e.g., `map()`, `filter()`, `flatMap()`) - Data does not require shuffling across nodes, so all operations can be grouped into a **single stage** within a **single Job**.

- 🤝**Wide Transformations** (e.g., `groupBy()`, `reduceByKey()`, `join()`) - Data needs to be shuffled across partitions, creating **stage boundaries** and potentially leading to multiple Jobs if the DAG dependencies require recomputation.

A **Spark Job** is broken down into **Stages**, where narrow transformations are grouped into the same stage, and wide transformations introduce stage boundaries. A **Job** in Spark may contain one or more **Stages**, depending on the number of wide transformations present.

Here's how our query gets divided into stages:

Transformation	Narrow/Wide	Stage
FileScan + Filter	Narrow	Stage 1
Aggregation (groupBy city, avg)	Wide (Shuffle required)	Stage 2
Sort on avg(salary)	Narrow	Stage 3

Tasks: The Smallest Execution Unit

Once the job is split into stages, **DAGScheduler** further breaks each **stage** into **tasks.**

- Each Task represents a **unit of execution** on a single data **partition.**
- All tasks within a stage execute in **parallel** across the cluster.

Here's what the final **DAG** looks like:

```
Stage 1: [Task1, Task2, Task3, Task4] → Run in parallel (Filter)
    |
Stage 2: [Task5, Task6, Task7, Task8] → Run in parallel (Shuffle + Aggregate
    |
Stage 3: [Task9, Task10, Task11, Task12] → Run in parallel (Sort + Show)
```

At this point, Spark knows what needs to be executed, how it should be executed, and where the tasks will run. The next step? Actually scheduling and executing these tasks across worker nodes.

Step 5: Executing Tasks — The Role of TaskScheduler

Once the **DAGScheduler** has split the job into stages and tasks, it hands them over to the **TaskScheduler**, which takes care of executing them efficiently.

What is TaskScheduler?

TaskScheduler is a Spark Driver component responsible for:

✅ Communicating with the Cluster Manager to allocate resources

✅ Assigning tasks to Executors for execution

✅ Managing task execution within each stage

✅ Retrying failed tasks if necessary

At a high level, the task execution process looks like this:

```
TaskScheduler → Cluster Manager → Executors (Task Execution) → Task Completi
```

But how does this actually work? Let's break it down step by step.

🚀 Task Execution Flow

1️⃣ TaskScheduler Requests Resources from the Cluster Manager

- **TaskScheduler** sends task execution requests to the **Cluster Manager** via RPC (**Remote Procedure Call**).
- The Cluster Manager allocates **CPU** and **memory** resources and informs the TaskScheduler about available Executors.

2️⃣ TaskScheduler Assigns Tasks to Executors

- Once resources are allocated, TaskScheduler sends tasks to the assigned **Executors** via **RPC.**
- Each Executor process receives the task, **deserializes** it, and runs it inside a **TaskRunner thread.**

3️⃣ Execution of Tasks

- The Executor processes the task, which involves, reading the partition from the source.
- And applying transformations on **partitions.** If it's a **narrow transformation**, the output stays in memory. If it's a **wide**

transformation, intermediate results are stored for shuffling.

4️⃣ Heartbeat Monitoring & Failure Handling

- Executors send **heartbeat signals** to the Driver process at regular intervals. If the Driver does not receive a heartbeat, it assumes the Executor has **failed**.
- The **TaskScheduler** reassigns the failed tasks to a different active Executor.
- Meanwhile, the **Cluster Manager** may launch new Executors to replace the lost ones.

5️⃣ Task Completion & Final Results

- When an Executor completes a task, If the task is part of an action like `collect()` or `show()`, the Executor sends the results to the **Driver**.
- Otherwise, it stores intermediate results and notifies **TaskScheduler** about the completion.
- Once all tasks in a stage are completed, **TaskScheduler** informs **DAGScheduler** which either, Schedules the next stage if more stages remain or else sends the final result to the Driver if the execution is complete.

🛠️ Retry Mechanism — Handling Executor Failures

Spark automatically retries failed tasks to ensure reliability.

- **Heartbeats & Failure Detection:** Executors send heartbeat signals to the Driver. If the heartbeat stops, Spark assumes failure.
- **Task Rescheduling:** The TaskScheduler assigns the failed task to a new Executor.
- **Replacing Lost Executors:** The Cluster Manager launches new Executors when needed.

This mechanism ensures fault tolerance and prevents job failures due to node crashes.

🔍 Conclusion: Spark DAGs in Action: What You've Learned

Understanding Spark DAGs is more than just theory it's the key to writing high-performance PySpark applications. Let's quickly recap the most important takeaways:

✅ Lazy Evaluation & DAG Execution

Spark doesn't execute transformations immediately. Instead, it builds a DAG (Directed Acyclic Graph) and optimizes execution when an action is triggered.

✅ Catalyst Optimizer for Smart Query Planning

The Catalyst Optimizer refines your queries, applies techniques like predicate pushdown, and ensures efficient query execution.

✅ Narrow vs. Wide Transformations

- **Narrow transformations** (e.g., `map`, `filter`) don't require shuffling and run efficiently within the same partition.
- **Wide transformations** (e.g., `groupBy`, `join`) involve data shuffling and create execution bottlenecks if not optimized.

✅ DAGScheduler: Jobs, Stages & Tasks Breakdown

- The DAGScheduler divides the execution plan into Jobs, Stages, and Tasks based on the transformations applied. A Job corresponds to an action in the program, and is typically broken down into multiple Stages when wide transformations are present.
- Narrow transformations, which don't require shuffling, remain within the same Stage, while wide transformations trigger new Stages to handle the shuffling and dependencies effectively.

✅ TaskScheduler & Parallel Execution

- TaskScheduler assigns tasks to available executors across the cluster.
- Spark ensures fault tolerance by retrying failed tasks and reallocating resources when needed.

Optimize Your Spark Code for Performance:
By understanding how Spark's DAG execution works, you can:
✅ Avoid unnecessary computations and optimize queries.
✅ Reduce expensive data shuffling by carefully structuring transformations.
✅ Fine-tune resource allocation for better cluster performance.

So, the next time you run a Spark job, think about its DAG execution you

might spot opportunities to make it even more efficient!

📖 Want to Learn More?

If you're interested in understanding how PySpark code is executed, including how the driver, executors, and cluster manager interact, check out my in-depth article:

👉 **How PySpark Code is Executed: A Beginner's Guide to JVM, Py4J, and Spark Architecture**

This will help you connect the dots between DAGs, task execution, and Spark's internal processes.

Also, stay tuned for my next article on Data Lineage in Apache Spark, where I'll explore how Spark tracks data transformations, ensures reproducibility, and helps with debugging complex workflows. 🚀

📝References

- Pandas vs. Spark: Understanding Spark DAG
- DAG in Apache Spark — DataFlair
- Short Note on DAG and Catalyst Optimizer — Medium
- DAG Scheduler vs. Catalyst Optimizer — Stack Overflow
- Catalyst Optimizer: The Power of Spark SQL — Medium

Spark | Data Engineering | Pyspark | Directed Acyclic Graph | Distributed Systems

Published in Dev Genius

27K followers · Last published Jun 10, 2025

Follow

Coding, Tutorials, News, UX, UI and much more related to development

Written by Mohit Joshi

17 followers · 3 following

Follow

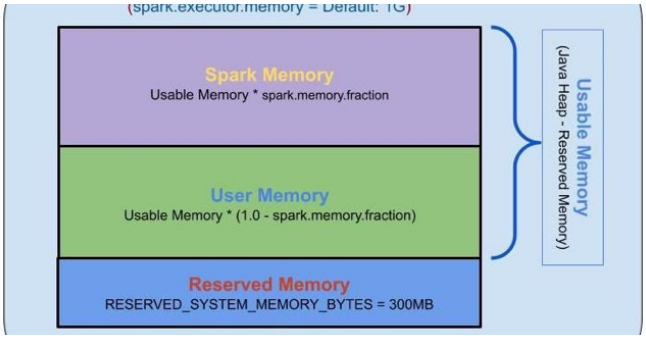
I'm Data + Software Engineer. Passionate reader & learner of new technologies & enthusiastic to share simplified tech knowledge.

No responses yet

Aaditya Adhikari

What are your thoughts?

More from Mohit Joshi and Dev Genius



In Dev Genius by Mohit Joshi

From Out-of-Memory to Optimized: Handling Java Heap...

In large-scale data processing using Apache Spark, memory-related issues like...

Sep 26, 2024 33 1

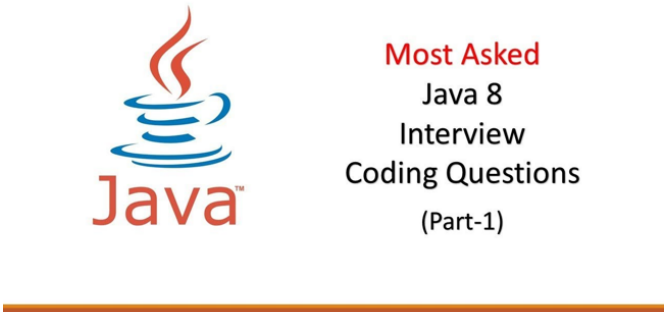
	Traditional MVP	MCVP (AI-native)
	2–3 months	<3 weeks with AI agents
	Forms, dashboards	Natural language UI
	Manual rules/config	Adaptive agents with memory
	Backend logic	Prompt tuning, skill enhancement

In Dev Genius by Mohammad Shoeb

This AI Agent Pattern Will Change .NET Forever (And...

“The future of software isn’t APIs. It’s agents that think, act, and collaborate.”

May 24 205 6

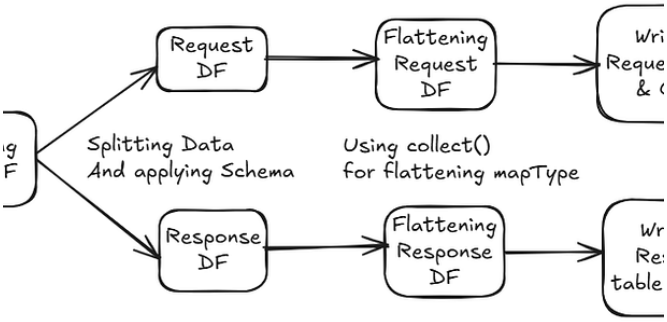


In Dev Genius by Anusha SP

Java 8 Coding and Programming Interview Questions and Answers

It has been 8 years since Java 8 was released. I have already shared the Java 8...

Jan 31, 2023 1.5K 24



Mohit Joshi

How We Optimized Spark Job: Great Journey of Learning

A real-world tale of logs, lag, and late-night Spark UI staring contests—with memes...

May 20 41

See all from Mohit Joshi See all from Dev Genius

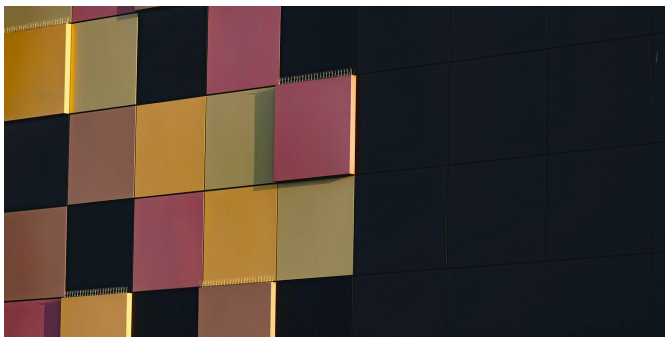
Recommended from Medium

Mayurkumar Surani

🚀 Mastering PySpark: Your Complete Guide to 46 Essential...

Collection of PySpark Functions

Jun 3 34



Vijay Gadhave

How Apache Spark Uses CPU, Memory, and Storage

Note: If you're not a medium member, CLICK HERE

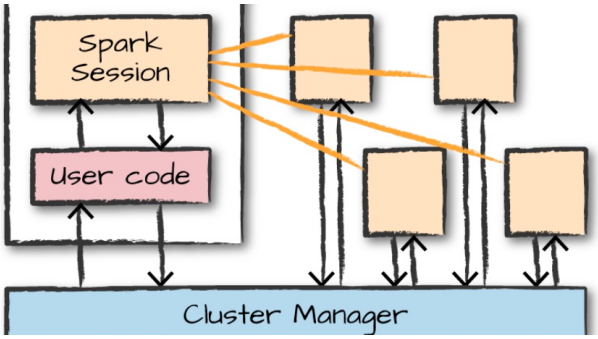
Jan 6 3

The Data Engineering Digest

🚀 Apache Spark 4.0—A Leap Forward in Big Data Processing

"From batch to blazing, Spark 4.0 isn't just a version bump—it's a smarter, scalable...

Jun 8 3



In Towards Dev by Prem Vishnoi(cloudvala)

Apache Spark Architecture :A Deep Dive into Big Data...

Agenda

Feb 6 98 1

In Level Up Coding by Yousry Mohamed

Resurrecting Scala in Spark : Another tool in your toolbox wh...

Spark Dataset API is still useful to handle some edge cases that require extra...

Jan 3 46

In Data Engineer Things by Santosh Joshi

Broadcast Tables vs Broadcast Variables vs Accumulators in...

How to Optimize Spark Jobs by Using These Powerful Features

Feb 23 62

See more recommendations