Apache Spark fundamentals in Databricks

Spark Architecture:

The Spark architecture in Databricks involves two main layers: the **Databricks platform architecture** (control and compute planes) and the underlying **Apache Spark runtime architecture** (driver, cluster manager, and executors).

Databricks Platform Architecture

Databricks acts as a managed layer on top of cloud providers (like AWS or Azure), providing an optimized environment for Spark. It is divided into two primary planes:

- **Control Plane:** This is managed by Databricks and hosts backend services, the web application (notebooks, UI), and metadata storage. It handles coordination but does not process user data directly.
- **Compute Plane:** This runs within the customer's cloud subscription (or a Databricks-managed serverless plane) and is where the actual data processing occurs. It contains the Spark clusters (driver and worker nodes) that execute tasks on the data.

Jayarani Arunachalam

Apache Spark fundamentals in Databricks

Spark Architecture:

The Spark architecture in Databricks involves two main layers: the **Databricks platform architecture** (control and compute planes) and the underlying **Apache Spark runtime architecture** (driver, cluster manager, and executors).

Databricks Platform Architecture

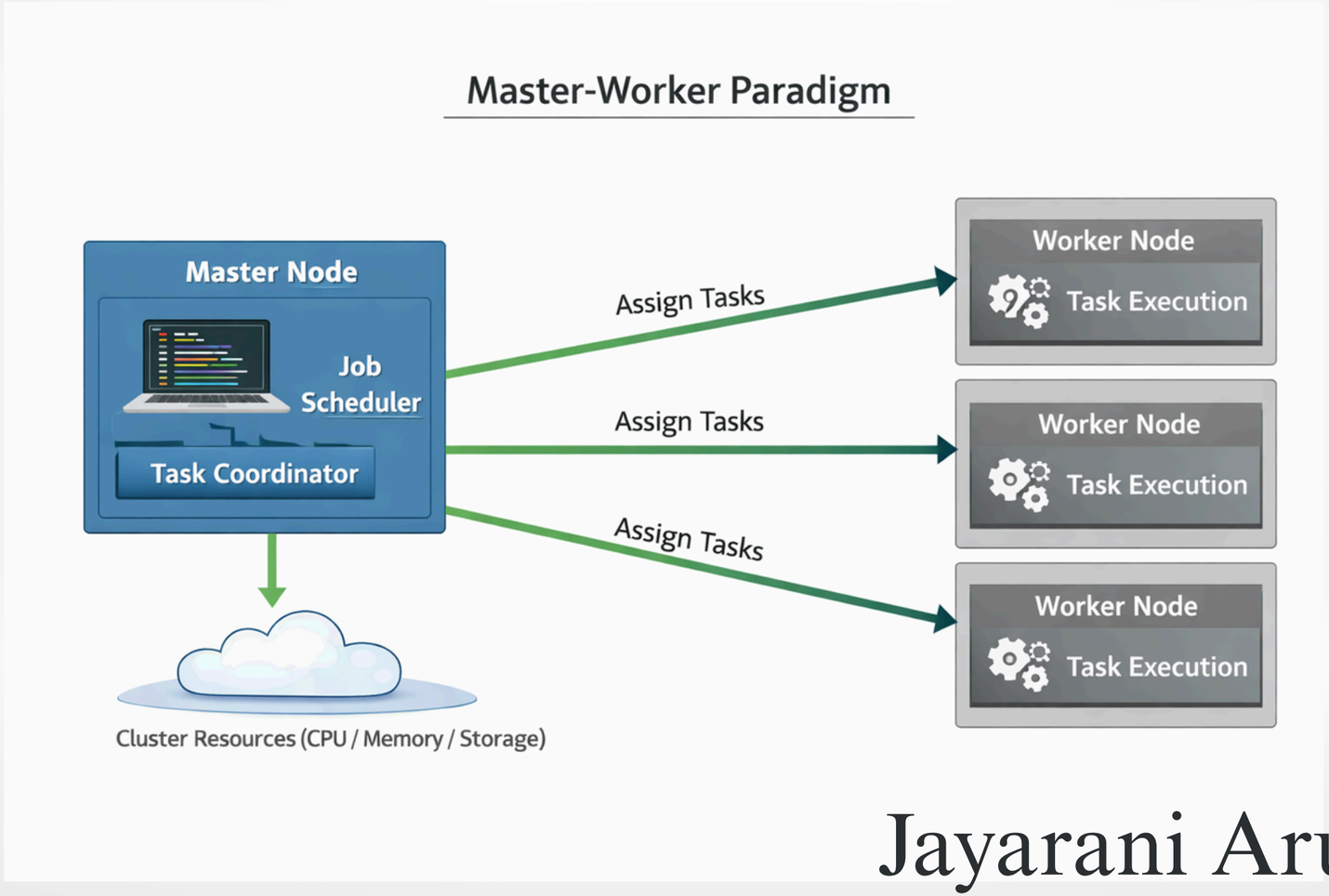
Databricks acts as a managed layer on top of cloud providers (like AWS or Azure), providing an optimized environment for Spark. It is divided into two primary planes:

- **Control Plane:** This is managed by Databricks and hosts backend services, the web application (notebooks, UI), and metadata storage. It handles coordination but does not process user data directly.
- **Compute Plane:** This runs within the customer's cloud subscription (or a Databricks-managed serverless plane) and is where the actual data processing occurs. It contains the Spark clusters (driver and worker nodes) that execute tasks on the data.

Jayarani Arunachalam

Apache Spark Runtime Architecture

Apache Spark is a distributed data processing engine. Within the compute plane, a Spark cluster operates using a master-worker paradigm. Driver, Executors, and DAG are runtime components created when a Spark application runs.



Jayarani Arunachalam

Components of Apache SparkArchitecture

- **Driver Program:** It is a JVM process. This runs the main function of the application, creates the SparkSession (the entry point for Spark functionality), and coordinates the execution. It converts the user's code into a Directed Acyclic Graph (DAG) of operations, breaks it into stages and tasks, and manages task scheduling. It monitors executors and collects final results. In the Databricks environment, the notebook interface often serves as the driver program.

- **DAG** is Directed Acyclic Graph. It represents transformations and dependencies. Created due to lazy evaluation.Spark builds DAG but does not execute immediately. Execution starts only when an action is called: Ex: show(), count(), write()

- **Cluster Manager:** This component is responsible for acquiring and allocating resources (CPU, memory) on the cluster nodes. Databricks automatically manages the cluster, including auto-scaling and auto-termination features, so users don't have to manage complex resource negotiation.

- **Worker Nodes:** These are the physical or virtual machines that perform the actual data processing. Each worker node runs one or more executors.

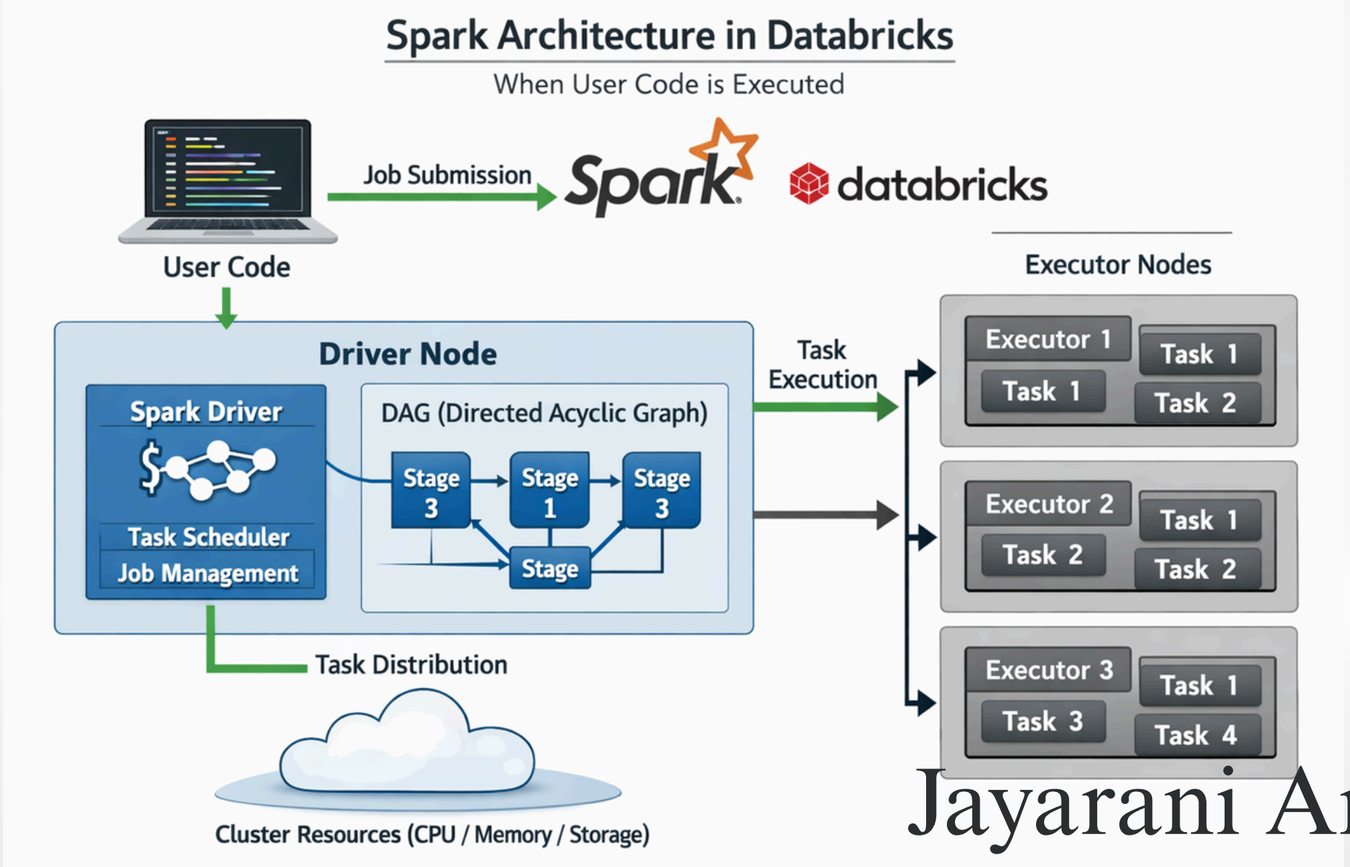
- **Executors:** These are processes that run on worker nodes, execute the tasks assigned by the driver, and store data in memory or on disk when caching is used. They return results to the driver program.

Jayarani Arunachalam

Execution Flow:

When a user runs a command in a Databricks notebook, the process is as follows:

1. The **Driver Program** (notebook interface) translates the code into an optimized execution plan represented by a DAG.
2. The **Cluster Manager** allocates resources (worker nodes and executors) for the job.
3. The **Driver Program** sends tasks to the Executors on the Worker Nodes.
4. The **Executors** perform the tasks in parallel and store intermediate data in memory for speed.
5. The results are returned to the **Driver Program** (and displayed in the notebook or saved to storage).



DataFrames vs RDDs

In Databricks, DataFrames and RDDs (Resilient Distributed Datasets) are both immutable, distributed collections of data, but they differ fundamentally in how they are managed and optimized by the Spark engine.

DataFrames (The Modern Standard)

DataFrames are the primary way to interact with data in Databricks today. They organize data into named columns, much like a table in a relational database or a spreadsheet.

- **Performance:** Significantly faster than RDDs due to two core engines:
  - **Catalyst Optimizer:** Automatically optimizes query execution plans.
  - **Project Tungsten:** Manages memory at the JVM level for highly efficient binary data processing.
- **Ease of Use:** Provides a high-level, declarative API (SQL-like) that allows you to specify *what* to do, leaving Spark to figure out *how* to do it efficiently.
- **Schema Awareness:** DataFrames are "schema-aware," meaning Spark knows the data types of your columns and can optimize storage and access accordingly.

RDDs (The Low-Level Core)

RDDs are the original "building blocks" of Spark. While DataFrames actually run on top of RDDs "under the hood," you rarely need to interact with them directly in 2026.

- **Performance**: Generally slower because they lack the Catalyst and Tungsten optimizations. RDDs also suffer from higher Garbage Collection (GC) overhead because they store data as raw Java/Python objects.
- **Control:** Offers fine-grained, object-oriented control. You must tell Spark exactly how to process the data step-by-step.
- **Unstructured Data:** Best for data that doesn't fit a tabular format, such as raw text logs, media files, or binary packet streams.

Jayarani Arunachalam

Lazy Evaluation:

In Databricks, Lazy Evaluation is a core design principle of Apache Spark. It means that Spark does not execute your data transformations immediately when you write the code. Instead, it records the operations as a plan and waits until you explicitly ask for a result.

Transformations vs. Actions

Transformations (Lazy): These are instructions that describe how to change the data. When you run these, Spark just adds them to a "logical plan."

- Examples: filter(), select(), groupBy(), join(), withColumn().
- Result: It returns a new DataFrame immediately, but no actual data processing has happened yet.

Actions (Eager): These are commands that trigger the actual computation because they require a result to be shown or saved.

- Examples: show(), count(), collect(), save(), display().
- Result: Spark looks at the recorded transformations, optimizes them, and sends them to the executors for processing.

Jayarani Arunachalam

Lazy Evaluation Cntd...

Why does Spark do Lazy Evaluation? (The Benefits)

By waiting until an Action is called, Spark can perform Query Optimization through its Catalyst Optimizer.

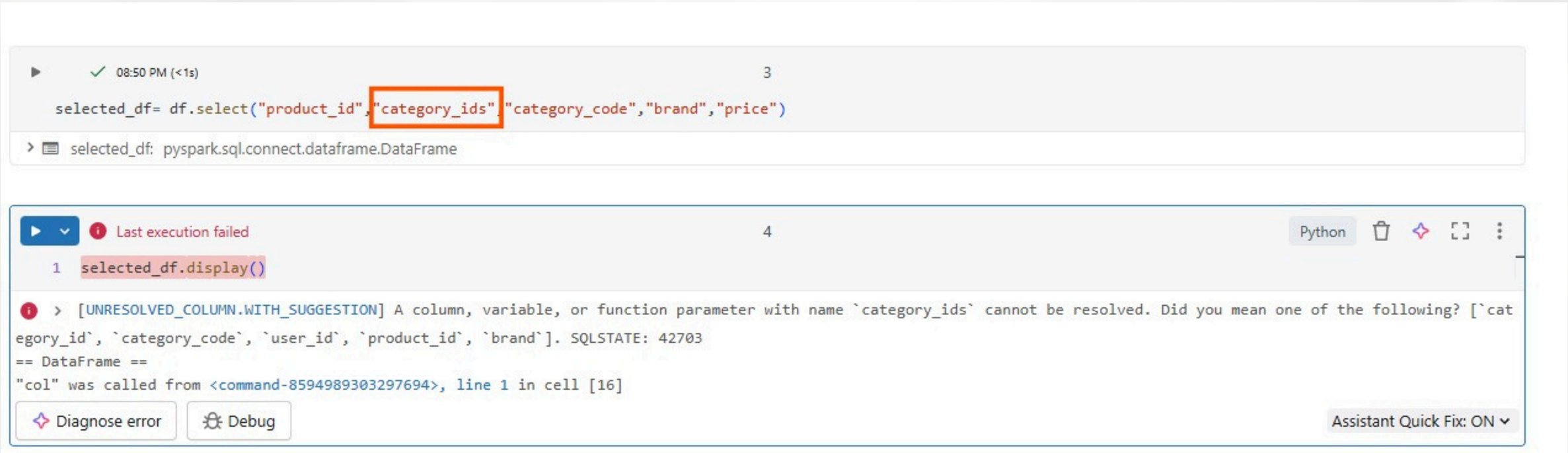
- **Predicate Pushdown:** If you load a 10TB table and then filter for just one city, Spark won't load the whole 10TB. It "pushes" the filter down to the data source and only reads the relevant rows.
- **Column Pruning**: If you only select 2 columns out of 100, Spark will only read those 2 columns from the disk.
- **Combined Steps:** Spark can merge multiple steps into one. For example, if you add a column and then filter the data, Spark can do both in a single pass over the data instead of two.

The Execution Flow

When an Action is finally triggered, Spark follows these steps:

1. DAG Creation: It builds a Directed Acyclic Graph (the "map" of your transformations).
2. Logical Plan: It creates a plan of what needs to be done.
3. Physical Plan: It decides how to do it (e.g., which join algorithm to use).
4. Execution: It breaks the plan into tasks and sends them to the worker nodes.

Lazy Evaluation Cntd...



In the above example, when df.select() is executed, Spark does not perform any computation. Instead, it only creates a logical execution plan. The actual execution happens only when an action such as display() is called. At this point, Spark evaluates the plan, and the error is thrown because the column category_ids does not exist. This behavior clearly demonstrates lazy evaluation in Databricks Spark.

Potential Pitfall

Because computations are delayed, "hidden" errors may be encountered. For example, if the initial data read points to a missing file, the code won't fail when the read transformation is defined; it will only crash much later when an Action like count() is called.

Notebook Magic Commands

In Databricks, magic commands are special instructions at the start of a notebook cell that allow you to switch languages, manage files, and control the environment.

1. Language Magic Commands

These commands override the notebook's default language for a single cell. This allows data scientists to use Python for modeling while data engineers use SQL for transformations in the same notebook.

- %sql: Executes Spark SQL queries directly against tables and views.
- %python: Runs Python code, even if the notebook's default language is Scala or SQL.
- %scala and %r: Switches the cell context to Scala or R respectively.
    - Note: Variables are not shared directly across different language REPLs (e.g., a Python variable is not accessible in a %sql cell without using temporary views).

2. File System Magic (%fs)

The %fs command provides a shorthand for Databricks Utilities (dbutils.fs) to interact with the Databricks File System (DBFS) and cloud object storage.

- %fs ls /path/: Lists files and directories in a specific path.
- %fs cp source destination: Copies files across the file system.
- %fs rm -r /path/: Recursively removes a directory.
- %fs head /path/file.txt: Displays the first few lines/bytes of a file.

Jayarani Arunachalam

Utility & Workflow Magics

- %md: Renders Markdown for documentation, including text, images, and mathematical formulas.
- %run <notebook_path>: Includes and executes another notebook inline. All variables and functions from the called notebook become available in the current one.
- %sh: Executes shell commands (Bash) on the cluster's driver node. Useful for installing OS-level dependencies or checking local disk space.
- %pip: Standard for installing notebook-scoped Python libraries. These libraries persist only for the current session and do not affect other users on the cluster.
- %skip (New in late 2025): A productivity feature that allows you to skip specific cells during "Run All" execution without commenting out the code.

Best Practice:

Always place %pip and environment configuration commands in the first cells of the notebook. Call dbutils.library.restartPython() if packages don't appear immediately.