
#UnlockThePower
Swipe to see the journey
SPARK
SPARK
DATA
AI
DATA
SWIPE FOR KNOWLEDE!

Prabhu Krishnammagari

Module 1: What is Big Data

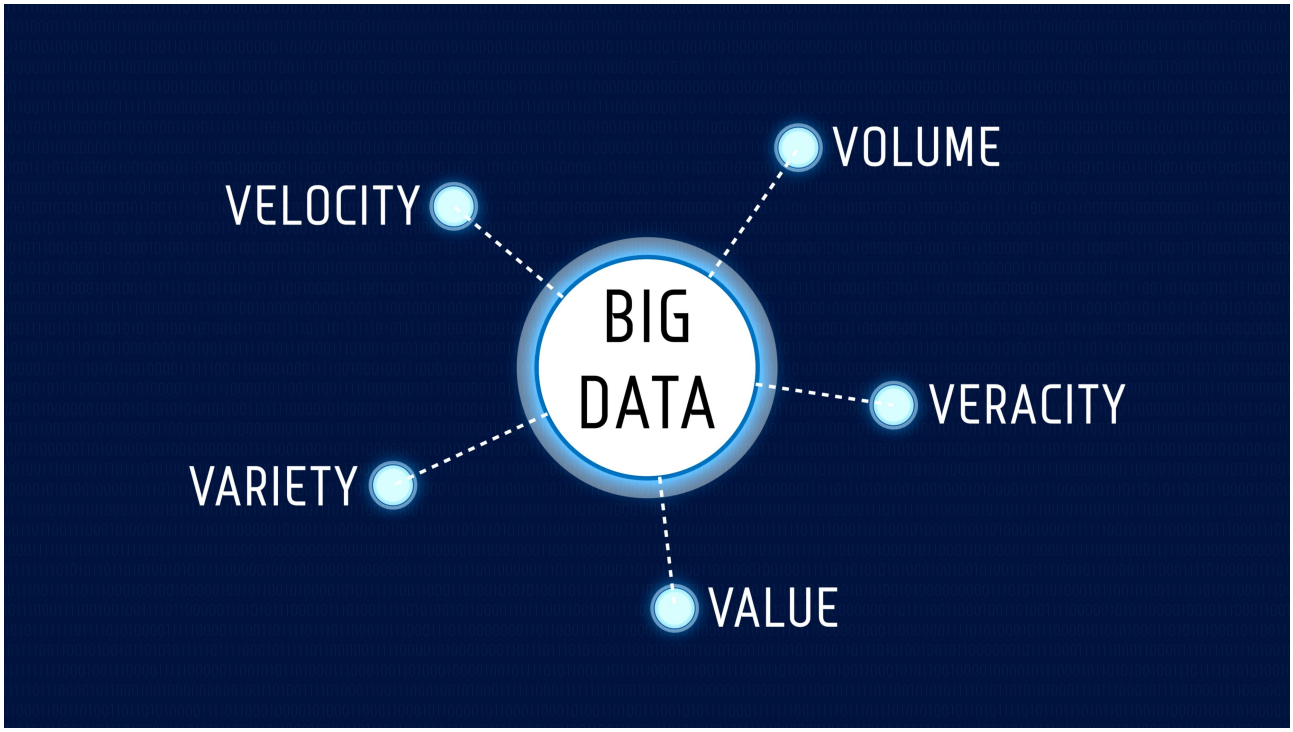
Our goal here is to establish a clear, standard definition we can use throughout this entire learning plan. As the roadmap suggests, we'll ground our definition in the **NIST (National Institute of Standards and Technology) Big Data Interoperability Framework**. This is a great approach because it moves beyond vague buzzwords and gives us a structured, engineering-focused perspective.

The NIST Definition of Big Data

NIST defines Big Data as data that **"exceeds the capacity or capability of current or conventional methods and systems."**

The key takeaway here is that "big" is a moving target. It's not about a specific number of gigabytes or terabytes. It's a **relative term** that describes a situation where your existing tools are no longer sufficient. The data's characteristics—its size, speed, or complexity—force you to adopt a new class of technologies (like Hadoop, Spark, etc.) to store, process, and analyze it effectively.

The NIST framework identifies key characteristics, often called the "V's," which help us diagnose a Big Data problem. While the original list had 3, it's commonly expanded to 5 or more. Let's start with the core 5.



Big Data: A Framework for Decision-Making

1. Domain-Mapped Definition (Grounded in NIST)

Big Data refers to data ecosystems whose characteristics of **Volume**, **Velocity**, **Variety**, **Veracity**, and **Value** render traditional data processing architectures inadequate. It necessitates a distributed systems approach where data is stored across a cluster of machines and processed in parallel to generate insights within a tolerable timeframe.

The key domains involved are:

- **Data Providers:** The sources generating the data (e.g., application logs, IoT sensors, social media feeds, transactional databases).
- **Big Data Framework Providers:** The technologies that enable distributed storage and processing (e.g., HDFS, Spark, Kafka).
- **Big Data Application Providers:** The tools and platforms that perform the actual analysis and deliver insights (e.g., custom Spark jobs, data warehouses, BI dashboards).
- **Data Consumers:** The end-users or systems that consume the insights (e.g., data scientists, business analysts, recommendation engines).

2. "When-to-Choose Big Data Tools" Rubric

Use this rubric to determine if a problem requires a Big Data solution. A "Yes" to one or more of these questions, especially Volume and Variety, strongly indicates the need for tools like Spark.

Characteristic	Question to Ask	Example & Why it Matters	When to Choose Big Data Tools?
Volume (Scale)	Is the dataset too large to fit on a single, powerful server (e.g., multi-terabyte or	A 10 TB log file from a web server. A single machine can't store it, let alone read and	**Yes**
Velocity (Speed)	Is data arriving faster than a traditional database can ingest, index, and serve it (e.g., thousands of events per	Real-time fraud detection on credit card swipes. You need to process the event in milliseconds, which a traditional DB write-	**Yes**
Variety (Complexity)	Does the data come in multiple, complex formats (e.g., structured tables, unstructured text, JSON,	Analyzing customer support tickets (text), call recordings (audio), and CRM data (structured) together to	**Yes**
Veracity (Quality)	Is the data messy, inconsistent, or untrustworthy, requiring significant cleaning and	Aggregating product data from multiple online vendors where SKUs, names, and prices are inconsistent and often missing.	**Maybe.** If the volume is small, traditional ETL tools might suffice. If large, Spark is ideal for
Value (Economics)	Is there a clear business case that justifies the cost and complexity of a distributed	Analyzing years of clickstream data to build a recommendation engine that increases sales by	**Crucial Prerequisite.** Without value, a Big Data project is just an

Module 2: Big Data Characteristics in Practice

In Module 1, we defined the 5Vs. Now, we'll connect them to real-world engineering decisions. Each "V" isn't just a descriptor; it's a force that dictates specific architectural choices and system controls. Understanding this link is key to designing effective Big Data systems.

Tying the 5Vs to System Architecture

📦 Volume (The Challenge of Scale)

When data volume grows, a single machine can no longer store or process it.

- **Architectural Response:**
  - **Distributed Storage:** Instead of one large hard drive, we use systems like **HDFS (Hadoop Distributed File System)** or cloud **Object Storage (like Amazon S3)** that spread data across a cluster of commodity machines.
  - **Parallel Processing:** We use frameworks like **Apache Spark** that can process chunks of this distributed data simultaneously on different machines.
- **System Controls:**
  - **Autoscaling:** In the cloud, we automatically add or remove compute nodes from our cluster based on the workload, ensuring we have enough power for large jobs without overpaying during quiet periods.
  - **Data Partitioning:** We strategically break data into smaller chunks (partitions) based on keys (like date or region) to ensure queries only read the data they need, dramatically speeding up processing.

⚡ Velocity (The Challenge of Speed)

When data arrives at high speed, the system must be able to ingest it without falling behind or losing information.

- **Architectural Response:**
  - **Decoupled Ingestion:** We use a messaging queue like **Apache Kafka** as a buffer. Data producers write to Kafka at their own pace, and our processing application (Spark) consumes from it at its own pace. This prevents the producer from being slowed down by the consumer.
  - **Stream Processing:** We use engines like **Spark Structured Streaming** that are designed to operate on endless streams of data, not just finite batches.

- **System Controls:**
    - **Backpressure:** A critical mechanism where the downstream consumer (Spark) can signal to the upstream source (Kafka) to slow down if it's becoming overwhelmed. This prevents system crashes due to data overload.
    - **Watermarking:** A technique in stream processing to handle late-arriving data, ensuring correctness in time-based calculations (e.g., "how many users logged in during the last 5 minutes?").

🎭 Variety (The Challenge of Complexity)

When data comes in different formats (structured, semi-structured, unstructured), a rigid, predefined database schema is no longer viable.

- **Architectural Response:**
    - **Schema-on-Read:** Unlike traditional databases (schema-on-write), in Big Data we often store the raw, varied data first and then apply a schema when we read or query it. This is the core idea of a **Data Lake**.
    - **Flexible File Formats:** We use formats like **Parquet** or **Avro** that are self-describing and can handle complex, nested data structures efficiently.
- **System Controls:**
    - **Schema Enforcement & Evolution:** Tools like **Delta Lake** or **Apache Iceberg** act as a layer on top of a data lake, providing the ability to enforce a schema (preventing bad data from being written) while also allowing that schema to evolve gracefully over time (e.g., adding a new column).

✅ Veracity (The Challenge of Trust)

When data is inconsistent, incomplete, or inaccurate, decisions based on it will be flawed. Garbage in, garbage out.

- **Architectural Response:**
    - **ETL/ELT Pipelines:** We build automated pipelines that clean, transform, and validate data as it moves from its raw state to a curated, analytics-ready state (e.g., the Medallion Architecture: Bronze → Silver → Gold).
- **System Controls:**
    - **Data Quality Testing:** We use frameworks like **Great Expectations** to define "expectations" about our data (e.g., `user_id` should never be null, `age` should be between 0 and 120). These tests run automatically within our data pipelines to catch issues early.
    - **Data Lineage:** Tools that track the entire journey of a piece of data from source to final report. If a report looks wrong, we can trace back to see exactly how the data was transformed.

💡 Value (The Challenge of ROI)

A Big Data system is complex and expensive. Its existence must be justified by the value it creates.

- **Architectural Response:**
    - **Layered Architectures (e.g., Lakehouse):** We design systems that can serve a wide range of use cases—from raw data access for data scientists to clean, aggregated tables for BI dashboards—maximizing the return on investment.
- **System Controls:**
    - **Data Governance & Catalogs:** We use tools like **Apache Ranger** for security and access control, and Data Catalogs to make data discoverable and understandable for users. This ensures the right people can access the right data securely to drive business value.
    - **Cost Monitoring & Optimization:** Actively tracking cloud spend and optimizing Spark jobs to run more efficiently.

Exercise: 5V-to-Architecture Matrix

Let's use a **Ride-Sharing Service (e.g., Uber/Ola)** as our reference dataset to build the matrix.

V	Ride-Sharing Example	Architectural Choice/Tradeoff
Volume	Billions of GPS location pings and ride events stored daily (Petabytes/year).	Use cloud object storage (S3) for infinite, cheap storage. Use Spark on a Kubernetes/YARN cluster for parallel processing of historical ride
Velocity	Thousands of real-time GPS pings per second from active drivers and riders.	Use Kafka to ingest the firehose of GPS data. A Spark Structured Streaming job consumes this data for real-time applications like surge pricing
Variety	GPS coordinates (structured), user reviews (unstructured text), driver profile photos (binary image files), payment records	Store all raw data in a Data Lake (S3). Use Spark SQL to process structured/semi-structured data and Spark ML libraries (NLP) to analyze
Veracit	GPS signals can be inaccurate in tunnels or dense urban areas ("noisy data"). User-	The data pipeline includes filtering/smoothing algorithms for GPS data. Fuzzy matching logic
Value	Identifying fraudulent trips, optimizing driver dispatch to reduce wait times, creating dynamic "surge" pricing to	The entire architecture is justified by its ability to directly increase revenue (surge pricing) and reduce costs (fraud detection, efficiency).

Tradeoff Focus: Batch vs. Streaming for Velocity

For our ride-sharing service, the **Velocity** dimension forces a critical tradeoff:

- **Batch Processing:** We could collect all ride data for an hour or a day and process it in a large batch job.

    - **Use Case:** Calculating a driver's total daily earnings, generating weekly reports on the most popular routes.
    - **Pros:** Simpler to manage, higher throughput, more cost-effective for non-urgent tasks.
    - **Cons:** High latency. A driver has to wait until the next day to see their earnings. Can't react to events in real-time.
- **Streaming Processing:** We process each event (e.g., a GPS ping, a ride request) as it occurs, within seconds or milliseconds.
    - **Use Case:** Matching a rider with the nearest available driver, updating the map with the car's live location, calculating surge pricing based on real-time demand.
    - **Pros:** Very low latency, enables real-time decision-making.
    - **Cons:** More complex to build and operate (requires managing state, handling failures, ensuring exactly-once processing), can be more expensive.

Conclusion: A modern ride-sharing service cannot function without a streaming architecture for its core operations. However, it still relies on a batch architecture for analytics, reporting, and machine learning model training. This hybrid approach is very common.

Module 3: Storage Infrastructure

This module is foundational. The way we store data dictates how we can process it. We'll explore the shift from traditional, single-machine storage to the distributed models that make Big Data possible.

Monolithic vs. Distributed Storage

Monolithic (Scale-Up)

Think of a traditional, high-end enterprise server or a Storage Area Network (SAN). To increase capacity, you "scale up": you add more disks, more memory, or a faster CPU to that **single box**.

- **Analogy:** A single, giant, industrial-strength filing cabinet. To store more, you have to buy a bigger, more expensive cabinet.
- **Limitations:**
    - **Single Point of Failure:** If the cabinet breaks, all access to information is lost.
    - **Finite Scale:** Eventually, you can't make the cabinet any bigger.
    - **High Cost:** That high-end, specialized hardware is very expensive.

Distributed (Scale-Out)

This is the core paradigm of Big Data. Instead of one giant machine, we use a cluster of many cheaper, commodity servers. We "scale out" by simply adding more servers to the cluster. Systems like **HDFS** pioneered this, and cloud **Object Storage** perfected it.

- **Analogy:** A massive warehouse filled with thousands of cheap, standard filing cabinets. To store more, you just add another row of cabinets.
- **Advantages:**
    - **Fault Tolerance:** If one cabinet catches fire, the documents are safe because you have copies in several other cabinets (**Replication**).
    - **Massive Scalability:** There's virtually no limit to how many cabinets you can add to the warehouse.
    - **Cost-Effective:** Standard cabinets are much cheaper than a single, custom-built industrial vault.

Key Concepts in Distributed Storage

Replication This is the cornerstone of reliability. To prevent data loss from hardware failure (which is expected in a large cluster), distributed systems automatically create multiple copies of each piece of data and store them on different machines (and ideally, in different server racks). A typical default is **3x replication**.

Durability This is a measure of the long-term safety of your data. Cloud object storage like **Amazon S3** provides incredible durability (often cited as "eleven 9s" or 99.999999999%). This means if you store 10 million objects, you can expect to lose only one every 10,000 years. This is achieved by replicating data across multiple, geographically distinct data centers.

Namespace Abstraction This is how a distributed system hides its complexity from the user. You don't see thousands of machines; you see a single, cohesive file system.

- **In HDFS:** A central server called the **NameNode** keeps the master directory—the metadata that knows which files are split into which blocks and where those blocks live across the cluster.
- **In Object Storage (e.g., S3):** This is even more abstract. You interact with a flat space of "buckets" and "objects." You give the system an object (your file) and a unique key (its name/path), and the system handles all the details of where and how it's stored. The path `s3://my-bucket/raw/events/2025/09/14/` isn't a series of folders; it's just part of the object's name. This simple, infinitely scalable model is why object storage has become the de facto standard for data lakes in the cloud.

Data Layout Draft

Let's draft the data layout for a modern data lake, often called a **Medallion Architecture**. This pattern organizes data into Bronze, Silver, and Gold layers.

Layer	Purpose & State	Layout Example (on S3)	Retention Policy	Encryption	Typical Access
Bronze (Raw)	Immutable, raw data ingested from source systems. The "single source of truth." Append-only.	`s3://lake/bronze/sap_erp/sales_orders/load_dt=2025-09-14/`	**Long-term/ Forever.** Store for years to meet compliance or for reproces	**Default Server-Side Encryption (SSE-S3).** Simple, manag	**Write-once** by ingestion jobs. **Rarely read**, except for backfil

Silver (Curated)	Data is cleaned, validated, de-duplicated, and conformed into a queryable structure (e.g., partitioned Parquet files or Delta Lake tables).	`s3://lake/silver/sales/dim_customer/` or `s3://lake/silver/web/page_views/`	**As needed for analytics.** Old partitions may be retired or archive	**Server-Side Encryption with managed keys (SSE-KMS).**	**Read/Write** by Spark ETL jobs. **Read-only** for data analyst
Gold (Serving)	Highly aggregated, business-level tables that are optimized for specific use cases like reporting and BI dashboards.	`s3://lake/gold/finance_mart/daily_revenue_summary/`	**Based on reporting needs.** May only keep the last 90 days of aggregates for a	**Server-Side Encryption with managed keys (SSE-KMS).**	**Infrequent writes** by final aggregation jobs. **Heavily read** by BI tools

This layered approach provides a clear separation of concerns, improves data quality, and allows different user groups to interact with the data at the appropriate level of refinement.

Module 4: Distributed File Systems (HDFS)

HDFS (Hadoop Distributed File System) is the original storage system that made large-scale data processing on commodity hardware a reality. Understanding its architecture is crucial because it established the patterns that later systems, including cloud storage, built upon or reacted to. The core principle of HDFS is **"move compute to the data, not the data to the compute."**

HDFS Architecture: The Master and the Workers

HDFS has a simple but powerful master/worker architecture.

🧠 NameNode (The Master)

The NameNode is the brain of the HDFS cluster. It's a single server that manages the entire file system's **metadata**. It does **not** store the actual data.

- **Analogy:** Think of it as a library's master card catalog or index. It knows the name of every book (file), where each chapter (block) of the book is located, and on which shelf (DataNode) you can find it.
- **Key Responsibilities:**
  - Maintains the directory tree (e.g., `/user/data/sales.csv`).
  - Tracks which blocks make up each file.
  - Knows the location of every replica of every block.
  - Manages file system access (permissions).

💪 DataNodes (The Workers)

The DataNodes are the workhorses of the cluster. They are responsible for actually storing the data on their local disks. A cluster will have many DataNodes, often thousands.

- **Analogy:** These are the actual shelves in the library holding the books (data blocks).
- **Key Responsibilities:**
  - Store and retrieve data blocks when told to by a client or the NameNode.
  - Handle the **replication** of data blocks to other DataNodes.
  - Constantly send "heartbeats" and "block reports" to the NameNode to report their health and the list of blocks they are storing.

Blocks

Files in HDFS are broken into large, fixed-size chunks called **blocks** (typically 128 MB or 256 MB). This large block size is optimized for the primary goal of HDFS: streaming large amounts of data sequentially with high throughput. It also reduces the amount of metadata the NameNode has to manage.

HDFS I/O Flows: Reading and Writing Data

Write Flow (Replication Pipeline)

This is how a file gets written and replicated across the cluster.

1. **Client → NameNode:** Your application (the client) tells the NameNode, "I want to create a new file named `/data/log.txt`."
2. **NameNode → Client:** The NameNode checks permissions. If everything is ok, it responds with the addresses of a set of DataNodes (e.g., DataNode A, DataNode B, DataNode C) to host the replicas for the *first block* of the file.
3. **Client → DataNodes (The Pipeline):**
    - The client sends the first block's data **only to the first DataNode** (DataNode A).
    - DataNode A receives the data, writes it to its local disk, and then **forwards the data to DataNode B**.
    - DataNode B receives it, writes it locally, and **forwards it to DataNode C**.
4. **Acknowledgements:** As each DataNode finishes writing, it sends an acknowledgement back up the pipeline. Once the client receives the final acknowledgement, it knows the block has been safely replicated and can proceed to the next block.
5. **Repeat:** The client requests a new list of DataNodes from the NameNode for the next block and repeats the process until the entire file is written.

Key Insight: The client streams the data only once. The replication is handled efficiently by the DataNodes in a pipeline, minimizing the client's network bandwidth usage.

Read Flow

This is how a file is read from the cluster.

1. **Client → NameNode:** The client tells the NameNode, "I want to read `/data/log.txt`."
2. **NameNode → Client:** The NameNode checks its metadata and returns a list of all the blocks for that file. For each block, it provides the addresses of **all the DataNodes** that hold a replica.

3. **Client → DataNode:** The client directly contacts the **closest DataNode** for each block to read the data. The data flows directly from the DataNode to the client. The NameNode is not involved in the data transfer.

4. **Fault Tolerance:** If a DataNode is unreachable or slow during a read, the client simply tries the next DataNode in the list for that block's replica.

Diagram and Comparison

Simplified Flow Diagrams

- **Write Flow:** `Client --(1. Req)--> NameNode Client <--(2. DN List)-- NameNode Client --(3. Data Block 1)--> DN-A --(Data)--> DN-B --(Data)--> DN-C Client <--(4. ACK)-- DN-A <--(ACK)-- DN-B <--(ACK)-- DN-C`
- **Read Flow:** `Client --(1. Req)--> NameNode Client <--(2. Block Locations)-- NameNode Client --(3. Read Block 1)--> Closest DataNode (e.g., DN-B) Client <--(4. Data)-- DN-B`

HDFS vs. Other DFS Approaches (Cloud Object Storage - S3)

Feature	HDFS	Cloud Object Storage (e.g., Amazon S3)
Architecture	**Coupled Compute & Storage.** Designed for data locality where compute nodes are also data nodes.	**Decoupled Compute & Storage.** Storage (S3) and compute (EC2) are separate services, connected by a high-speed network.
API	File System API (directories, files, paths).	Object API (buckets, keys, objects) accessed via HTTP.
Consistency	**Strongly Consistent.** Changes to the file system namespace are immediately visible to all clients.	**Eventually Consistent (historically).** Now offers strong read-after-write consistency for new objects, but eventual for overwrites/deletes.
Replication	**Managed by the User.** You configure the replication factor (e.g., 3x). The system replicates within a single data center/cluster.	**Managed by the Cloud Provider.** Abstracted away from the user. Data is automatically replicated across multiple physical data centers (Availability Zones) for extreme durability.
Use Case	The original on-premise Big Data file system.	The standard for data lakes in the cloud. Cheaper, more durable, and more flexible.

Module 5: MapReduce

MapReduce is the original processing paradigm that came with Hadoop. While largely superseded by Spark for most new development, its core concepts are fundamental to understanding how all distributed data processing works. It brilliantly simplified parallel programming, allowing developers to process petabytes of data without being distributed systems experts.

The MapReduce Model: A Census Analogy

Imagine you need to count the population of a large country. Doing it with one person would take forever. The MapReduce approach is to divide and conquer.

1. The Map Phase (Distribute & Count Locally) 🗺️

First, you send thousands of census takers (the **Mappers**) to every single town and city. Each mapper's job is simple: count the population in *their assigned town only*. They don't need to know about any other town. When they're done, they write their result on a small card, like `(Town_A, 5280)`.

- **In Code:** A mapper is a piece of code that takes a single input record (like a line in a file) and outputs one or more key-value pairs. For the classic "word count" example, if the input line is `hello world`, the mapper outputs two pairs: `(hello, 1)` and `(world, 1)`.

2. The Shuffle & Sort Phase (Collect & Group) shuffling cards

Next, a central coordinator collects all the cards from all the census takers. They then perform a massive sorting operation, grouping all the cards for the same town together. All the cards for `Town_A` go into one pile, all the cards for `Town_B` go into another, and so on.

- **In the Framework:** This phase is handled **automatically** by the Hadoop framework. It's the most complex and often most expensive part of the job, as it involves moving data across the network to group related keys together.

3. The Reduce Phase (Aggregate Final Results) ➕

Finally, you assign a single accountant (the **Reducer**) to each pile. The reducer for the `Town_A` pile takes all the cards for that town and sums them up to get the final, total population.

- **In Code:** A reducer is a piece of code that receives a key (e.g., `hello`) and a list of all its associated values (e.g., `[1, 1, 1, 1, 1]`). It then performs an aggregation on this list (e.g., summing them up) and outputs a single final result: `(hello, 5)`.

Why Was It Replaced by Spark?

MapReduce was revolutionary, but it had one major drawback: **it's extremely disk-heavy**. The output of the Map phase is always written to HDFS. The Reducers then read this data back from HDFS. This constant writing to and reading from disk makes it slow and unsuitable for:

- **Iterative Algorithms:** Like many machine learning algorithms that need to pass over the same data multiple times.
- **Interactive Analysis:** Where a data scientist wants to run a query and get a result in seconds, not minutes.

This is the exact problem Spark was designed to solve by using in-memory processing, which we'll see in the next module.

Module 6: Spark

Welcome to Spark! This is the engine that redefined big data processing. It's a unified, general-purpose computing engine that's significantly faster and more flexible than the classic MapReduce model.

The Core Idea: Speed Through Memory

The primary reason for Spark's performance advantage is its use of **in-memory processing**.

- **MapReduce Analogy:** Imagine a chef making a complex dish. After every single step—chopping an onion, searing the meat, mixing a sauce—the MapReduce chef puts all the ingredients back into the refrigerator (disk). This is slow and generates a lot of I/O.
- **Spark Analogy:** The Spark chef keeps all the necessary ingredients on the kitchen counter (RAM). They perform all the steps in a fluid workflow and only put the final, finished meal onto a plate (write the final result to disk).

This in-memory approach, combined with an advanced **DAG (Directed Acyclic Graph)** execution engine that plans the entire job upfront, can make Spark up to 100x faster than MapReduce.

Spark's Architecture: The Foreman and the Workers

Understanding Spark's architecture is key to understanding how it runs your code in parallel.

Analogy: Think of a construction project.

- **Cluster Manager (The General Contractor - e.g., YARN, Kubernetes):** This is the resource manager. It provides the worksite (the cluster nodes) and allocates workers for the job. Spark plugs into it.
- **Driver Program (The Site Foreman 👷):** This is the brain of your Spark application; it's where your `main()` function lives.
    - It creates the `SparkSession`, which is your entry point to all Spark functionality.
    - It analyzes your code, creates the optimal execution plan (the DAG), and orchestrates the entire job.
- **Executors (The Workers 🔨):** These are processes launched on the worker nodes of the cluster.
    - They have one job: execute the specific **tasks** given to them by the Driver.
    - They store data partitions in memory (cache) and on their local disks.
    - They report the status and results of their work back to the Driver.

The process is simple: you submit your code (`spark-submit`), it starts the **Driver**, the Driver requests resources (Executors) from the **Cluster Manager**, and then the Driver sends work (tasks) to the **Executors** to be processed in parallel.

The Spark UI: Your Job's Instrument Panel

The Spark UI is a web interface that gives you an incredible view into what your application is doing. It's the most important tool for debugging and tuning. When your application is running, you can access it (usually at `http://<driver-node>:4040`) to see:

- **DAG Visualization:** A graph of how Spark is executing your job.
- **Jobs:** The high-level actions in your code (e.g., `.count()`, `.save()`).
- **Stages:** Jobs are broken into stages whenever data needs to be shuffled across the network.
- **Tasks:** The individual units of work that run on the executors.

Hands-on: Local Install & Minimal ETL

Now it's time to get your hands dirty.

1. Install Spark Locally

For learning purposes, you don't need a full cluster. You can install PySpark, the Python API for Spark, right on your machine.

Bash

```
pip install pyspark
```

2. Complete the Spark Quick Start

The best way to start is with the official documentation. Work through the "Spark Quick Start" guide to get a feel for the interactive shell.

3. Build a Minimal ETL Job

Here is a sample PySpark script that fulfills the module's hands-on goal. It reads a CSV file, performs a simple transformation, and writes the output as partitioned Parquet files.

Python

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, year

# 1. Create a SparkSession (the entry point)
spark = SparkSession.builder \
    .appName("MinimalETL") \
    .master("local[*]") \
```

```
    .getOrCreate()

# Assume you have a CSV file named 'sales.csv' with columns:
'order_id', 'product', 'amount', 'order_date'

# 2. READ: Load the raw data from a CSV file
print("Reading raw sales data...")
df_raw = spark.read.csv("sales.csv", header=True,
inferSchema=True)

# 3. TRANSFORM: Clean and enrich the data
print("Transforming data...")
df_transformed = df_raw \
    .withColumn("sale_year", year(col("order_date"))) \
    .filter(col("amount") > 0)

# 4. WRITE: Save the transformed data as Parquet, partitioned
by year
print("Writing partitioned data...")
df_transformed.write \
    .mode("overwrite") \
    .partitionBy("sale_year") \
    .parquet("output/sales_transformed")

print("ETL job complete.")

# 5. VERIFY: Use Spark SQL to check the output
print("Verifying output with Spark SQL...")
df_final = spark.read.parquet("output/sales_transformed")
df_final.createOrReplaceTempView("sales")

spark.sql("SELECT sale_year, COUNT(*) as num_sales FROM sales
GROUP BY sale_year ORDER BY sale_year").show()

# Stop the SparkSession
spark.stop()
```

This simple script demonstrates the entire lifecycle.

Module 7: Spark's Core Abstraction - The RDD

Before we had the high-level, user-friendly DataFrames that we used in our ETL job, there was the **RDD**—the original, core data structure that made Spark so revolutionary. Understanding RDDs is like understanding how a car's engine works; you might not build one every day, but knowing the principles makes you a much better driver.

What is an RDD and Why Was It Created?

An **RDD** stands for **R**esilient **D**istributed **D**ataset. It's an immutable, partitioned collection of records that can be operated on in parallel across a cluster.

The key question is, *why* was it needed? It was designed to overcome the major limitations of Hadoop's MapReduce model:

- **Excessive Disk I/O:** MapReduce writes intermediate results to the HDFS disk after every single map and reduce stage. This is incredibly slow, especially for iterative algorithms (like in machine learning) that need to pass over the same data multiple times.
- **Lack of Abstraction:** Developers had to write complex, low-level Java code for even simple tasks.

The RDD solves this by allowing Spark to keep data **in-memory** between steps. Instead of writing to disk, Spark builds up a plan of all the steps you want to do.

Analogy: Imagine you're baking a cake. MapReduce is like a baker who, after cracking an egg, writes down "egg cracked," saves the note, and puts the egg back in the fridge. Then, to get the flour, they re-read the note, get the egg out again, add flour, write a new note "flour added," and put everything away. It's slow and cumbersome.

Spark, using RDDs, is like an experienced baker who reads the entire recipe first, gets all the ingredients out, and does all the steps in sequence without stopping. The final cake is only "served" (written to disk) at the very end.

The Core Properties of an RDD Explained

Let's break down the name, as it perfectly describes what it is:

- **Resilient:** This is the fault-tolerance mechanism. RDDs achieve this by tracking their **lineage**. Spark doesn't store the actual data at every step, but rather the *recipe* used to create it (this "recipe" is called a Directed Acyclic Graph or DAG). If a node holding a partition of data fails, Spark knows the exact steps to automatically recompute just that missing piece on another node.
- **Distributed:** The data in an RDD is split into smaller chunks called **partitions**. These partitions are distributed across the various executor nodes in the cluster, allowing Spark to process them in parallel.

- **Dataset:** It represents a collection of your data (e.g., lines from a log file, rows from a table). A critical feature is that RDDs are **immutable**. You cannot change an RDD. Instead, when you apply an operation to it, you create a *new* RDD.

RDD Operations: Transformations vs. Actions

This is the most crucial concept to grasp about RDDs. There are only two types of operations you can perform on them.

1. Transformations

Transformations create a *new* RDD from an existing one. Examples include `map()`, `filter()`, and `reduceByKey()`.

The key characteristic of transformations is that they are **lazy**. This means Spark doesn't execute them immediately. It just adds the operation to its DAG (the recipe). It's a way of saying, "Here's what I *want* you to do eventually."

2. Actions

Actions trigger the actual computation. When you call an action, Spark looks at the entire DAG of transformations you've built up, optimizes it, and then executes it on the cluster. Actions either return a final value to your driver program (your shell) or write data out to storage. Examples include `collect()`, `count()`, and `saveAsTextFile()`.

This "lazy evaluation" is a cornerstone of Spark's performance. It allows the engine to see the entire workflow from start to finish and find the most efficient way to execute it.

Let's Get Hands-On: The "Hello, World!" of Big Data

The canonical example for any distributed computing framework is the "word count." We'll do this right in your PySpark shell.

Step 1: Create a Sample Data File

We don't need a massive file for this. Open your terminal (make sure your `.venv` is activated) and run this command to create a simple text file.

Bash

```
echo "hello world spark is great hello spark" > words.txt
```

Step 2: Start the PySpark Shell

It's simple:

Bash

```
pyspark
```

You should see the Spark welcome screen. Notice the line `Spark context available as 'sc'`. This `sc` (SparkContext) is your entry point for creating and manipulating RDDs.

Step 3: Load the File into an RDD (Transformation)

Now, let's load `words.txt` into an RDD. The `textFile` method creates one RDD where each element is a line from the file.

Python

```
# In your pyspark shell
lines = sc.textFile("words.txt")
```

Nothing happens yet! `textFile` is a transformation that simply tells Spark where to find the data. No job has been launched.

Step 4: Perform the Word Count Logic (Transformations)

Here we'll chain together several transformations.

1. **`flatMap`**: We need to split each line into individual words. If we used `map(lambda line: line.split(' '))`, we'd get an RDD of lists `[['hello', 'world', ...]]`. `flatMap` is better; it "flattens" the output, giving us an RDD of words `['hello', 'world', ...]`.
   Python

   ```
   words = lines.flatMap(lambda line: line.split(" "))
   ```

2. **`map`:** We need to create a key-value pair for each word to count it, like `('hello', 1)`.
   Python

   ```
   word_tuples = words.map(lambda word: (word, 1))
   ```

3. **`reduceByKey`:** This is the magic. It groups all pairs with the same key (the word) and applies a function to their values (the 1s). Here, we add them up.
   Python

   ```
   word_counts = word_tuples.reduceByKey(lambda a, b: a + b)
```

Remember, still nothing has actually been computed on the cluster. We've just built a very detailed recipe (the DAG).

Step 5: View the Results (Action)

Now we'll use an action to trigger the computation and see the result. The `collect()` action brings all the data from the RDD back to the driver program. **Warning**: Be very careful with `collect()` on large datasets, as it can overwhelm your driver's memory! For our tiny file, it's perfect.

Python

```
results = word_counts.collect()

for word, count in results:
    print(f"{word}: {count}")
```

Now, Spark executes the full pipeline. You'll see some logs, and then your output should appear:

```
spark: 2
is: 1
great: 1
hello: 2
world: 1
```

Congratulations! You've just successfully implemented a distributed word count using Spark's core RDD API. We've defined a series of lazy transformations and then triggered the entire job with a single action.

This fundamental model of transformations and actions is the bedrock of everything else we will do in Spark.

Expanding Your RDD Toolkit 🧰

We've used `map`, `flatMap`, and `reduceByKey`. Here are a few more fundamental operations you'll frequently encounter. Let's create a new, simple RDD in your shell to play with.

Python

```
# Create an RDD from a simple Python list
numbers_rdd = sc.parallelize([1, 2, 3, 4, 5, 5, 6, 7, 8, 8])
```

Key Transformations

- `filter(func)`: Returns a **new RDD** containing only the elements that return `True` when passed to the function.
  Python
  `# Let's keep only the even numbers.`
  even_numbers_rdd = numbers_rdd.filter(lambda x: x % 2 == 0).
  # ACTION: Let's see the result
  even_numbers_rdd.collect()
  # Expected output: [2, 4, 6, 8, 8]

- `distinct()`: Returns a **new RDD** that contains only the unique elements from the source RDD.
  Python

```
# Let's get the unique numbers
unique_numbers_rdd = numbers_rdd.distinct()
# ACTION: Let's see the result
unique_numbers_rdd.collect()
# Expected output: [1, 2, 3, 4, 5, 6, 7, 8]
```

Key Actions

- `count()`: Returns the total number of elements in the RDD.
  Python

  ```
  numbers_rdd.count()
  # Expected output: 10
  ```

- `take(n)`: Returns a list containing the first `n` elements of the RDD. This is much safer than `collect()` on large datasets because it only fetches a small, defined subset.
  Python

  ```
  numbers_rdd.take(3)
  # Expected output: [1, 2, 3]
  ```

The Power of Persistence: Caching ⚡

This is a **critical concept** for performance.

The "Why": By default, Spark re-evaluates the entire lineage of an RDD every single time you call an action on it. In our word count example, if we called `word_counts.collect()` twice, Spark would read the file, split the lines, map the words, and reduce them **twice**. For complex, iterative jobs (like machine learning), this is incredibly wasteful.

The "What": You can tell Spark to **persist** or **cache** an RDD in memory after it's computed the first time. The next time you use that RDD, Spark will read it directly from the memory of the cluster nodes, skipping all the previous computation steps.

The easiest way to do this is with the `.cache()` method. It's a synonym for persisting with the default storage level (in-memory).

Analogy: Caching is like taking a picture of your finished cake. The first time, you have to do all the work (the recipe/lineage). But the next time someone wants to see the cake, you can just show them the picture (`cache`) instead of baking it all over again.

Example Usage

Python

```
# Let's go back to our word count RDD
word_counts = sc.textFile("words.txt") \
                .flatMap(lambda line: line.split(" ")) \
```

```
                .map(lambda word: (word, 1)) \
                .reduceByKey(lambda a, b: a + b)

# Tell Spark to cache this RDD in memory after it's computed
word_counts.cache()

# ACTION 1: The first time this runs, it computes everything
and stores the result in memory.
print(word_counts.collect())

# ACTION 2: This is now super fast! Spark reads from the
cache instead of re-running the whole job.
print(f"Total unique words: {word_counts.count()}")
```

This concludes our tour of the RDD API. You now understand:

- What an RDD is and **why** it was created.
- The difference between **lazy transformations** and **eager actions**.
- How to perform basic data manipulations like counting words.
- How to use **caching** to dramatically speed up your jobs.

While modern Spark code primarily uses DataFrames (which we'll cover next), they are built directly on top of RDDs. Everything you just learned about lineage, laziness, and distributed execution is still happening under the hood.

Module 8: The DataFrame API - Structured Data at Scale

What is a DataFrame?

A Spark DataFrame is a distributed collection of data organized into **named columns**.

The best way to think about it is as a table in a relational database or a `pandas` DataFrame, but with a crucial difference: it's **distributed** across many machines and **lazily evaluated**.

The "Why": From RDDs to DataFrames

If RDDs are the foundation, why did Spark introduce DataFrames? The answer lies in one word: **structure**.

1. **Schema Awareness:** Unlike an RDD, which just sees a collection of opaque Python objects, a DataFrame knows the name and data type of every column. This information is called the **schema**.
2. **The Catalyst Optimizer:** This schema is a superpower. It allows Spark to use a sophisticated engine called the **Catalyst Optimizer**. This engine can understand *what* you want to do (your "logical plan") and figure out the most efficient way to physically execute it on the cluster. It can reorder filters, optimize joins, and perform countless other tricks that are impossible with RDDs.
3. **Ease of Use:** The API is rich and expressive. Common data manipulation tasks that were complex with RDDs become single, clear commands with DataFrames. Plus, you can use **SQL**!

Analogy: An RDD is like giving someone a box of jumbled car parts and a vague goal. They can build the car, but they have to figure everything out from scratch. A DataFrame is like giving them the same parts but with a detailed blueprint (the **schema**). They can now build the car much faster and more efficiently. The **Catalyst Optimizer** is the master mechanic who can look at the blueprint and find clever shortcuts.

Hands-On: Your First DataFrame

Let's get practical. In the PySpark shell, the entry point for DataFrames is the `SparkSession`, which is already created for you and available as the variable `spark`.

Step 1: Create a Sample CSV File

Let's create a simple dataset about product sales. Open a regular terminal (not the PySpark shell) and create this file.

Bash

```
# Make sure your venv is activated if you're in a project
folder
```

```
echo "product_id,category,amount" > sales.csv
echo "101,electronics,250" >> sales.csv
echo "102,books,45" >> sales.csv
echo "103,electronics,300" >> sales.csv
echo "104,books,60" >> sales.csv
echo "105,books,20" >> sales.csv
```

Step 2: Load the CSV into a DataFrame

If your PySpark shell isn't running, start it now (`pyspark`).

The standard way to create a DataFrame is by reading from a data source. `spark.read` is a powerful interface for this.

Python

```
# In your pyspark shell
sales_df = spark.read.csv("sales.csv", header=True,
inferSchema=True)
```

Let's break down those options:

- `"sales.csv"`: The path to our file.
- `header=True`: Tells Spark that the first line of the file is the column names.
- `inferSchema=True`: Tells Spark to make an extra pass over the data to figure out the data types of each column (e.g., `string`, `integer`). This is convenient for exploration but for production jobs, it's better to define the schema manually for performance.

Step 3: Basic DataFrame Operations

Just like with RDDs, these operations are divided into transformations and actions.

`printSchema()` (Utility)

This isn't a transformation or an action, but a quick utility to inspect the schema Spark figured out for us.

Python

```
sales_df.printSchema()
```

You should see something like:

```
root
 |-- product_id: integer (nullable = true)
 |-- category: string (nullable = true)
 |-- amount: integer (nullable = true)
```

`show()` (Action)

This is the most common action. It triggers the job and displays the first 20 rows of the DataFrame in a neat table.

Python

```
sales_df.show()
```

Output:

```
+----------+-----------+------+
|product_id|   category|amount|
+----------+-----------+------+
|       101|electronics|   250|
|       102|      books|    45|
|       103|electronics|   300|
|       104|      books|    60|
|       105|      books|    20|
+----------+-----------+------+
```

`select()` (Transformation)

Selects specific columns to create a new DataFrame.

Python

```
# Create a new DataFrame with just the category and amount
category_amount_df = sales_df.select("category", "amount")

# This is a transformation, so nothing has happened yet.
# We need an action to see the result.
category_amount_df.show()
```

`filter()` or `where()` (Transformation)

Filters rows based on a condition. They are aliases for each other.

Python

```
# Create a new DataFrame containing only electronics sales
electronics_df = sales_df.filter(sales_df.category ==
"electronics")

electronics_df.show()
```

Step 4: A Complete Analysis

Now, let's chain these together to answer a question: "**What is the total sales amount for each category?**"

This requires a `groupBy` transformation and an aggregation.

Python

```
# This is all one single transformation
category_totals_df = sales_df.groupBy("category") \
                             .sum("amount")

# Now, let's trigger the computation with an action
category_totals_df.show()
```

Output:

```
+-----------+-----------+
|   category|sum(amount)|
+-----------+-----------+
|electronics|        550|
|      books|        125|
+-----------+-----------+
```

Look how clean and readable that is compared to the RDD equivalent! We clearly stated our intent—group by category, then sum the amount—and the Catalyst Optimizer figured out the best way to execute it.

Now, let's see one of the most powerful features of the DataFrame API: the ability to seamlessly switch between the programmatic API and pure SQL.

We are going to perform the *exact same aggregation* (total sales by category), but this time using a SQL query.

Spark SQL: Your Data, Your Language

The core idea is simple: you can register any DataFrame as a temporary "view," which is essentially a table that you can query with SQL for the duration of your Spark session.

Step 1: Create a Temporary View

First, we need to give our `sales_df` a name that the SQL engine can recognize. We'll call it `sales_table`.

Python

```
# In your pyspark shell
sales_df.createOrReplaceTempView("sales_table")
```

This command doesn't move or copy any data. It simply creates a pointer, or an alias, telling Spark, "When you see the name `sales_table` in a SQL query, I'm referring to the data in `sales_df`."

Step 2: Execute a SQL Query

Now we can use the `spark.sql()` command to run any standard SQL query against our view.

Python

```
# Use spark.sql() to run a query. The query is just a string.
category_totals_sql_df = spark.sql("""
    SELECT
        category,
        SUM(amount) as total_sales
    FROM
        sales_table
    GROUP BY
        category
“"")
```

Crucial Point: The `spark.sql()` command does not immediately run the query. It is a **transformation** that returns a *new DataFrame*. All the rules of lazy evaluation still apply.

Step 3: Show the Result

To see the output, we call an action on our new DataFrame, just like before.

Python

```
category_totals_sql_df.show()
```

You will see the identical result:

```
+-----------+-----------+
|   category|total_sales|
+-----------+-----------+
|electronics|        550|
|      books|        125|
+-----------+-----------+
```

You can also run `.printSchema()` on `category_totals_sql_df` to see that it is a proper DataFrame with a new schema.

Why Does This Matter? The Takeaway

You have now seen two ways to achieve the exact same result:

1. **Programmatic API:** `sales_df.groupBy("category").sum("amount")`
2. **SQL API:** `spark.sql("SELECT ... FROM sales_table ...")`

This is incredibly powerful.

- **Flexibility:** Data engineers might prefer the programmatic API for building complex pipelines, while data analysts can use their existing SQL skills to query the exact same data.
- **No Performance Penalty:** Both approaches are converted into the same logical plan by Spark's engine. The Catalyst Optimizer will generate the **exact same highly-optimized physical execution plan** for both. You can choose the API that you are most comfortable with without worrying about performance.

This concludes our introduction to the DataFrame API. You've learned how to create, manipulate, and query structured data at scale using both a programmatic API and standard SQL. This is the foundation for virtually all modern Spark development.

Module 9: Schemas - The Blueprint of Your Data

In the last module, when we read our CSV file, we used the option `inferSchema=True`. While convenient for quick exploration, this is something you should **almost always avoid** in production code.

The "Why": The Importance of Explicit Schemas

Explicitly defining the structure of your data (the schema) before you read it has three major benefits:

1. **Performance:** `inferSchema=True` forces Spark to make an extra, full pass over your data just to guess the data types. For large datasets, this can be incredibly time-consuming and expensive. Providing an explicit schema allows Spark to read the data in a single pass.
2. **Correctness & Data Quality:** Spark's schema inference can be wrong. It might guess a column is an `Integer` when you need a `Long` to avoid overflow, or it might fail to recognize a specific date format. Defining a schema yourself ensures data is read into the correct types from the start. It acts as a contract; if the data doesn't conform, Spark will correctly flag it as null or throw an error, preventing silent failures down the line.
3. **Readability:** A defined schema at the top of your code serves as clear documentation for anyone (including your future self) who needs to understand the data your application expects.

Analogy: Using `inferSchema` is like asking a construction crew to build a house without a blueprint. They'll have to walk the site, guess the room sizes, and figure things out as they go. It's slow and error-prone. Providing an explicit **schema** is like handing them a detailed **blueprint**. They can build the house efficiently and correctly the first time.

How to Define a Schema in PySpark

You define a schema using two main components from PySpark's type library:

- `StructType`: Defines the structure of the entire DataFrame (a collection of fields).
- `StructField`: Defines a single column within the `StructType`. It takes three arguments:
    1. `name` (string): The name of the column.
    2. `dataType` (DataType): The data type, like `StringType()` or `IntegerType()`.
    3. `nullable` (boolean): Whether this column can contain null values.

Hands-On: Building and Applying a Schema

Let's create the schema for the **`sales.csv`** file we used in the previous module.

1. **Import the necessary types.** In your PySpark shell, run this import statement:
   Python
   ```
   from pyspark.sql.types import StructType, StructField, StringType, IntegerType
   ```

2. **Define the schema structure.** We will create a `StructType` that contains a list of `StructField` objects.
   Python
   ```
   sales_schema = StructType([StructField("product_id", IntegerType(), True), StructField("category", StringType(), True), StructField("amount", IntegerType(), True)])
   ```

   Here, we've explicitly defined each column's name, type, and nullability.

3. **Read the data using the defined schema.** Now, instead of `inferSchema`, we pass our custom schema object to the `.schema()` option.
   Python
   ```
   sales_df_with_schema = spark.read.csv("sales.csv", header=True, schema=sales_schema)
   ```

4. **Verify the result.** Check the schema of your new DataFrame.
   Python
   ```
   sales_df_with_schema.printSchema()
   ```
   The output will be identical to what `inferSchema` produced, but this time, the process was faster and more reliable because we told Spark exactly what to expect.

Common Data Types

Here are some of the most common data types you'll use when defining schemas:

Data Type	Description
`StringType()`	Represents character strings.
`IntegerType()`	Represents 4-byte signed integers.
`LongType()`	Represents 8-byte signed integers.
`DoubleType()`	Represents 8-byte double-precision floats.
`BooleanType()`	Represents boolean values (`True`, `False`).
`DateType()`	Represents calendar dates.
`TimestampType()`	Represents date and time with precision.

This practice of defining and applying schemas is a hallmark of professional Spark development. It's a fundamental step in building data pipelines that are not just functional, but also efficient and trustworthy.

Module 10: Interacting with Data Sources

Spark provides a powerful, unified API for reading from and writing to various data sources. The entry points are `spark.read` and `dataframe.write`.

The general syntax looks like this:

- **Reading:** `spark.read.format("...").option("key", "value").load("path")`
- **Writing:** `df.write.format("...").option("key", "value").save("path")`

Spark also provides convenient shortcuts for common formats, like the `spark.read.csv()` we've already used.

A Look at Common Data Formats

While Spark can handle many formats, you'll most often work with these three.

- **CSV:** Plain text, separated by commas.
    - **Pros:** Human-readable, universal.
    - **Cons:** No schema information is stored with the data, parsing can be slow, and it doesn't support complex data types well.
- **JSON:** A popular format for web APIs and semi-structured data.
    - **Pros:** Flexible schema, handles nested data structures well.
    - **Cons:** Very verbose (takes up more space), can be slow to parse.
- **Parquet:** The king of Spark formats. 👑
    - **Pros:** This is the **default and recommended format** for Spark. Here's why:
        1. **Columnar Storage:** Unlike CSV or JSON which are row-based, Parquet stores data in columns. This means all values for the `amount` column are stored together, and all values for the `category` column are stored together.
        2. **Amazing Compression:** Data with the same type (like a column of integers) compresses extremely well, saving significant disk space.
        3. **Performance via "Projection Pushdown":** Because it's columnar, if you run a query like `df.select("category")`, Spark will *only read the "category" column data from the disk*. This is a massive performance gain as

it drastically reduces I/O. For row-based formats, Spark would have to read every entire row just to pick out one field.

4. **Schema is Embedded:** The schema is stored within the Parquet files, making the data self-describing.

Write Modes: Preventing Data Disasters

When you write a DataFrame, you must specify a **save mode**. This is a critical safety feature that tells Spark what to do if data already exists at the target location.

- `SaveMode.Overwrite` or `"overwrite"`: **Deletes** whatever is in the directory and replaces it with the new data. Use with caution!
- `SaveMode.Append` or `"append"`: Adds the new data to the location, keeping the existing data.
- `SaveMode.Ignore` or `"ignore"`: If data already exists, do nothing and the save operation will not run.
- `SaveMode.ErrorIfExists` or `"error"` (**Default**): If data already exists, throw an exception and stop the application. This is the safest default.

Hands-On: Writing and Reading Parquet

Let's take the DataFrame from our last module and see this in action.

1. **Write the DataFrame to Parquet.** We'll use the `sales_df_with_schema` DataFrame we created with our explicit schema. We'll use `"overwrite"` mode for this exercise so we can run it multiple times without errors.
   Python
   ```
   # In your pyspark shell
   ```
   sales_df_with_schema.write.mode(“overwrite").parquet("sales_data_parquet")
   Spark will now execute a job to write the DataFrame's data into a new directory called `sales_data_parquet`.

2. **Read the Parquet Data Back.** Now, let's pretend we're starting a new Spark job and need to read this optimized data.
   Python
   ```
   parquet_df = spark.read.parquet("sales_data_parquet")
   ```

3. **Verify the Result.** Notice what we *didn't* have to do: we didn't provide a schema! Parquet took care of that for us.

The schema will be perfectly preserved, and the data will be identical. You've just performed a standard data engineering task: ingesting data from a raw format (CSV) and saving it in an optimized, queryable format (Parquet).

Module 11: DataFrame Operations - Manipulating Columns and Rows

In this module, we'll use the `parquet_df` DataFrame we created in the last lesson. If you've restarted your shell, you can recreate it quickly:

Python
```
# Recreate the DataFrame if needed
parquet_df = spark.read.parquet("sales_data_parquet")
```

Manipulating Columns

These are the bread-and-butter transformations for any data pipeline. A key thing to remember is that because DataFrames are immutable, each of these operations returns a **new DataFrame**.

Adding or Replacing a Column: `withColumn()`

This is the primary method for adding a new column or updating an existing one. It takes two arguments: the name of the new column and an expression defining its value.

To create these expressions, we often need to import functions from `pyspark.sql.functions`. The most basic is `col()`, which lets us refer to a column.

Python
```
from pyspark.sql.functions import col

# Let's add a "total_price" column that includes a 10% tax.
df_with_total = parquet_df.withColumn(
    "total_price",
    col("amount") * 1.10
)

df_with_total.show()
```

Renaming a Column: `withColumnRenamed()`

This is a straightforward transformation for renaming an existing column.

Python
```
# Let's rename 'amount' to 'base_price' for clarity
df_renamed = df_with_total.withColumnRenamed("amount",
"base_price")
```

```
df_renamed.show()
```

Dropping a Column: `drop()`

Used to remove one or more columns from a DataFrame.

Python

```
# Let's drop the original product_id column
df_final = df_renamed.drop("product_id")

df_final.show()
```

Manipulating Rows

Sorting Data: `orderBy()`

You can sort your DataFrame by one or more columns, in ascending or descending order.

Python

```
# Sort by the new total_price column in ascending order (the
default)
df_final.orderBy("total_price").show()

# Now sort by total_price in descending order
df_final.orderBy(col("total_price").desc()).show()
```

Handling Null Values: `.na` functions

Real-world data is messy and often contains nulls. The `.na` attribute on a DataFrame provides functions to handle them.

- **`na.drop()`**: Removes rows containing null values.
    - `df.na.drop(how='any')`: Drops a row if it has any nulls (default).
    - `df.na.drop(how='all')`: Drops a row only if all its values are null.
- **`na.fill()`**: Fills null values with a specified value.
    - `df.na.fill(0)`: Replaces nulls in all integer/double columns with 0.
    - `df.na.fill({'column_name': 'default_value'})`: Fills nulls in a specific column with a specific value.

Python

```
# Imagine our DataFrame had nulls. This would fill any nulls
# in the 'base_price' or 'total_price' columns with 0.
df_final.na.fill(0, subset=["base_price",
“total_price"]).show()
```

You now have a solid toolkit for the most common data manipulation tasks: adding, renaming, and removing columns; sorting your data; and handling missing values. These operations are the foundation of data cleaning and feature engineering in Spark.

Module 12: Joining DataFrames

A **join** is an operation that combines rows from two or more DataFrames based on a related column between them.

Analogy: Think of it like a VLOOKUP in Excel, but on a massive, distributed scale. You have a table of sales records with a `product_id`, and a separate table of product details, also with a `product_id`. A join lets you bring the product details (like its name and category) into your sales table.

The Main Types of Joins

Spark supports all standard SQL join types. Let's visualize the most common ones.

- **Inner Join (default):** This is the most common join. It returns only the rows where the key exists in **both** DataFrames.
- **Left Outer Join (`left`):** This returns **all** rows from the left DataFrame, and the matched rows from the right DataFrame. If there is no match for a key in the right DataFrame, the result is `null` for the right-side columns.
- **Full Outer Join (`outer`):** This returns **all** rows when there is a match in either the left or right DataFrame. It's essentially a combination of a left and right join. If a key from one table doesn't exist in the other, the missing side's columns will be `null`.

Hands-On: Enriching Sales Data

Let's see this in action. First, we need two DataFrames to work with.

1. **Create the DataFrames.** We'll create a `sales` DataFrame and a `products` DataFrame that can be joined on `product_id`.
   Python

   ```
   # In your pyspark shell
   sales_data = [(101, 250), (102, 45), (103, 300), (106, 80)]
   sales_df = spark.createDataFrame(sales_data, ["product_id", “amount"])
   products_data = [(101, "Laptop", "electronics"), (102, "Book", "books"), (103, "Desktop", "electronics"), (104, "Pen", “stationery")]
   products_df = spark.createDataFrame(products_data, ["product_id", "product_name", “category"])
   print("Sales Data:”)
   sales_df.show()
   print("Products Data:”)
   products_df.show()
   ```

   Notice that the sales data includes a sale for product `106` (which isn't in our products table) and the products table includes product `104` (which has no sales). This will help illustrate the different join types.

2. **Perform an Inner Join.** Let's combine them to see only the sales for which we have product information. The syntax is `df1.join(df2, on="key_column", how="join_type")`.
Python

```
# The 'on' argument specifies the key to join on
# 'how' specifies the type. "inner" is the default, so it's optional here.
inner_join_df = sales_df.join(products_df, on="product_id", how="inner")
print("Result of Inner Join:")
inner_join_df.show()
```

Result Analysis: Notice that product `106` (from sales) and product `104` (from products) are both missing. The inner join only keeps records where the `product_id` is present in **both** DataFrames.

3. **Perform a Left Join.** Now, let's say we want to keep all our sales records, even if we don't have product details for some of them.
Python

```
left_join_df = sales_df.join(products_df, on="product_id", how="left")
print("Result of Left Join:")
left_join_df.show()
```

Result Analysis: Look at the row for product `106`. Since we kept everything from the `sales_df` (the "left" DataFrame), this record is included. But since `106` doesn't exist in `products_df`, the `product_name` and `category` columns are filled with `null`. Product `104` is still gone because it wasn't in the left table.

Mastering joins is fundamental to building useful datasets. You can now combine data from virtually any number of sources to create a single, enriched view for your analysis.

Module 13: User-Defined Functions (UDFs)

What is a UDF?

A **U**ser-**D**efined **F**unction (UDF) is a feature that lets you register your own custom Python function and use it as a Spark SQL function within DataFrame operations.

The "Why": Sometimes, the logic you need to apply to a column is too complex or specific to be expressed using Spark's extensive library of built-in functions. You might need to use a third-party Python library or implement custom business logic. UDFs are the bridge that allows your Python code to operate on Spark's distributed data.

The Critical Performance Warning ⚠️

While UDFs offer great flexibility, they should be your **last resort**.

Analogy: Using Spark's built-in functions is like giving the Catalyst Optimizer a transparent blueprint. It can see everything, understand it, and optimize the entire construction plan. Using a **UDF** is like giving it a sealed, **black box**. The optimizer knows it has to send data into the box and get data out, but it has no idea what happens inside.

This "black box" behavior has two major performance costs:

1. **Serialization Overhead:** For every single row, Spark must pause its highly-optimized JVM process, convert the data into a format Python can understand (serialize), send it to a separate Python process, run your code, wait for the result, and then convert it back (deserialize). This back-and-forth is extremely slow compared to native operations.

2. **Optimization is Lost:** The Catalyst Optimizer cannot reason about or optimize your Python code. All its powerful abilities to reorder operations or simplify expressions are nullified.

Rule of Thumb: Always try to solve your problem using functions from `pyspark.sql.functions` first. Only use a UDF if it's impossible or impractical to do so.

Hands-On: Creating and Using a UDF

Let's use the `inner_join_df` from our previous lesson. First, recreate it if you've restarted your shell.

Python

```
sales_data = [(101, 250), (102, 45), (103, 300)]
sales_df = spark.createDataFrame(sales_data, ["product_id",
"amount"])
products_data = [(101, "Laptop", "electronics"), (102,
"Book", "books"), (103, "Desktop", "electronics")]
```

```
products_df = spark.createDataFrame(products_data,
["product_id", "product_name", "category"])
inner_join_df = sales_df.join(products_df, on="product_id",
how="inner")
```

1. **Write a standard Python function.** Let's create a function to label a sale as "High Value" or "Standard Value".
   Python

   ```
   def get_value_label(amount):

        if amount >= 200:
             return "High Value"
        else:
             return "Standard Value"
   ```

2. **Register the function as a UDF.** We need to import the `udf` function and the return data type.
   Python

   ```
   from pyspark.sql.functions import udf, col

   from pyspark.sql.types import StringType

   # Register the Python function as a UDF, specifying its
   return type.
   get_value_label_udf = udf(get_value_label, StringType())
   ```

3. **Apply the UDF in a `withColumn` transformation.** You can now use your registered UDF just like a built-in function.
   Python

   ```
   df_with_label = inner_join_df.withColumn(

  "value_label",
  get_value_label_udf(col("amount"))
   )

   df_with_label.show()
   ```

The result will have a new column with the correct labels.

The Better, Faster, Native Way

To prove the point about performance, here is how you should **always** try to implement the same logic using Spark's native functions. In this case, we can use the `when` function.

Python

```
from pyspark.sql.functions import when

# This achieves the exact same result as the UDF, but will be
MUCH faster.
df_native = inner_join_df.withColumn(
    "value_label",
    when(col("amount") >= 200, "High
Value").otherwise("Standard Value")
)

df_native.show()
```

The output is identical, but this second version allows the Catalyst Optimizer to fully understand and optimize the entire query plan.

You now know how to create a UDF for those rare cases when you truly need one, and more importantly, you know why you should prefer the native, built-in functions whenever possible.

Module 14: The Spark Execution Model & UI

When you submit your code, Spark translates it into a plan that can be executed in a distributed manner. Understanding this plan is crucial.

The Execution Hierarchy 🏗️

Your Spark code is broken down into a hierarchy of components.

1. **Application:** The highest level. This is your entire program. Your `pyspark` shell session is one Spark Application.
2. **Job:** A **Job** is created for every **action** you call. When you run `df.show()` or `df.write.parquet()`, Spark creates a new job to get that result. Transformations, being lazy, do not create jobs.
3. **Stage:** A Job is broken down into one or more **Stages**. A new stage is created whenever a **shuffle** is required. A shuffle is the expensive process of redistributing data across the nodes in your cluster. Operations like `groupBy`, `reduceByKey`, and `join` typically trigger a shuffle.
4. **Task:** A Stage is composed of many **Tasks**. A task is the smallest unit of work in Spark. It represents a unit of computation that is sent to a single executor core to run on a single **partition** of your data. If your DataFrame has 8 partitions, a stage will have at least 8 tasks.

The flow is: **Application > Job(s) > Stage(s) > Task(s)**

The Spark UI: Your Cockpit 🚀

The Spark UI is a web interface that gives you a detailed look inside your running Spark Application. It is your single most important tool for monitoring and debugging.

How to Access It: While your `pyspark` shell is running, the Spark UI is active. By default, you can access it by opening a web browser and navigating to: http://localhost:4040

Hands-On: A Guided Tour of the UI

Let's use the UI to see the execution model in action.

Step 1: Open the Spark UI Go ahead and open `http://localhost:4040` in your browser. You should see the main page for your PySpark application.

Step 2: Run a Simple Job (No Shuffle) In your `pyspark` shell, run the following code. This job simply creates a DataFrame and writes it out.

Python

```
# Make sure you have your inner_join_df from the last module
inner_join_df.write.mode(“overwrite").parquet("ui_test_data")
```

Now, look at the **Jobs** tab in the UI. You'll see a new job has appeared and completed. Click on its description. You'll see a **DAG Visualization** showing the plan. Notice that it's a simple, linear set of steps. This job likely only has one stage.

Step 3: Run a Complex Job (With a Shuffle) Now, let's run a job that requires a `groupBy`, which will trigger a shuffle.

Python

```
# This action will trigger a new job with a shuffle
spark.read.parquet(“ui_test_data").groupBy("category").count(
).show()
```

Step 4: Explore the Shuffle Job in the UI

1. Go back to the **Jobs** tab. You'll see a new job at the top of the list. Click on it.
2. Look at the **DAG Visualization**. You will now see that the plan is broken into **two boxes**, representing **two distinct stages**. The boundary between them is the shuffle caused by the `groupBy`.
3. Now, click on the **Stages** tab at the top of the page. You will see a list of all stages from all jobs. You can see how one job was broken down into two stages. You can also see details like how much data was read and written during the shuffle.
4. Finally, click on the **SQL / DataFrame** tab. This tab is amazing. It lists the queries you've run. Find the one for our `groupBy` and click on it. You can see the different plans: the initial plan, the "Optimized Logical Plan" created by Catalyst, and the final "Physical Plan" that Spark actually executed.

This UI provides complete transparency into your application's execution. By learning to read it, you can spot bottlenecks, understand why a job is slow, and see exactly how Spark is optimizing your code.

Module 15: Partitioning and Performance Tuning

What is a Partition? 🧩

A **partition** is a logical chunk of your DataFrame that is processed by a single task on a single executor core. The number of partitions determines the **degree of parallelism** in your Spark job.

Analogy: Imagine you have to sort a massive pile of 1 million letters. If you're one person, it will take a very long time. If you split the pile into 100 smaller piles (partitions) and give each pile to a different person (a core), the work gets done 100 times faster. Partitions are how Spark "divides and conquers" a big data problem.

Having the wrong number of partitions is a common cause of poor performance:

- **Too Few Partitions:** You don't use all the available cores on your cluster. If you have 100 cores but only 4 partitions, you're wasting 96 cores! This can also lead to OutOfMemory errors, as each large partition might not fit into an executor's memory.
- **Too Many Partitions:** Creates excessive overhead. Spark has to manage thousands of tiny tasks, and the scheduling overhead can sometimes outweigh the benefits of parallelism.

Controlling the Number of Partitions

You can explicitly control your DataFrame's partitioning with two key transformations.

`repartition(numPartitions)`

This transformation creates a new DataFrame with exactly `numPartitions`.

- **How it works:** It performs a **full shuffle**, meaning all the data is redistributed across the network.
- **When to use it:** Use `repartition` to either increase or decrease the number of partitions. Since it's a "heavy" operation involving a network shuffle, use it when the benefits of restructuring the data are high, such as before a series of expensive operations.

`coalesce(numPartitions)`

This is a more optimized method specifically for **decreasing** the number of partitions.

- **How it works:** It avoids a full shuffle. Instead, it merges existing partitions on the same worker node, minimizing network I/O.
- **When to use it:** Use `coalesce` when you have too many partitions and want to reduce them efficiently (e.g., after a filter operation that removes a lot of data). **You cannot use `coalesce` to increase the number of partitions.**

Hands-On: Managing Partitions

Let's use the `inner_join_df` from previous modules.

1. **Check the Current Number of Partitions.** You can see a DataFrame's partition count by accessing its underlying RDD.
   Python

```
# Recreate the DataFrame if needed

sales_data = [(101, 250), (102, 45), (103, 300)]
sales_df = spark.createDataFrame(sales_data,
    ["product_id", "amount"])
products_data = [(101, "Laptop", "electronics"), (102,
    "Book", "books"), (103, "Desktop", "electronics")]
products_df = spark.createDataFrame(products_data,
    ["product_id", "product_name", "category"])
inner_join_df = sales_df.join(products_df,
    on="product_id", how="inner")

# Check the partitions. The number will depend on your
local setup (e.g., 2, 4, 8)
print(f"Initial partitions:
{inner_join_df.rdd.getNumPartitions()}")
```

2. **Repartition the Data.** Let's increase the number of partitions to 5.
   Python

```
repartitioned_df = inner_join_df.repartition(5)

print(f"Partitions after repartition:
{repartitioned_df.rdd.getNumPartitions()}")
```

3. **Coalesce the Data.** Now let's efficiently reduce the partitions down to 2.
   Python

```
coalesced_df = repartitioned_df.coalesce(2)

print(f"Partitions after coalesce:
    {coalesced_df.rdd.getNumPartitions()}")
```

Strategic Partitioning: `partitionBy()`

The most advanced optimization is to partition your data on disk by a specific column. When you write a DataFrame using `partitionBy("column_name")`, Spark creates a subdirectory for each unique value in that column.

Python

```
# This writes the data to disk in a partitioned folder
structure
inner_join_df.write.partitionBy("category") \
           .mode("overwrite") \
           .parquet("sales_partitioned")
```

This creates a directory structure like:

```
sales_partitioned/
├── category=books/
│   └── ...part-0000.parquet
└── category=electronics/
    └── ...part-0000.parquet
```

The Benefit: When you later read this data and filter on the partition key (e.g., `spark.read.parquet("sales_partitioned").where(col("category") == "books")`), Spark is intelligent enough to **only read the `category=books` directory**. This is called **partition pruning** and it can make queries exponentially faster by avoiding massive amounts of I/O.

Module 16: Spark SQL - Window Functions.

What are Window Functions?

A **window function** performs a calculation across a set of rows that are somehow related to the current row. Unlike a standard `groupBy` aggregation which collapses many rows into a single output row, a window function returns a value **for every single row**.

Analogy: Imagine a `groupBy('department').avg('salary')` operation. It will tell you the average salary for the "Sales" department and return just one row for that department. A window function, however, can show you each employee in the Sales department, and on *every one of those rows*, it can also display the average salary for the entire Sales department. It lets you see the detail and the aggregate at the same time.

The `WindowSpec`: Defining Your "Window"

To use a window function, you first define the "window" or group of rows using the `Window` specification. This has two main parts:

1. **`PARTITION BY`**: This divides the rows into partitions, or groups. The window function is applied independently to each partition. This is the rough equivalent of `GROUP BY`.
2. **`ORDER BY`**: This orders the rows within each partition. This is essential for functions that depend on order, like ranking or accessing the previous/next row.

Hands-On: Ranking Employees by Salary

A classic use case for window functions is to rank items within a category. Let's find the top-earning employees in each department.

Step 1: Create the DataFrame First, let's create a sample DataFrame of employees.

Python

```
# In your pyspark shell
data = [("Michael", "Sales", 4600), ("Robert", "Sales",
4100),
        ("Maria", "Finance", 3000), ("Scott", "Finance",
3300),
        ("Jen", "Finance", 3900), ("Jeff", "Marketing",
3000),
        ("Kumar", "Marketing", 2000)]
```

```
df = spark.createDataFrame(data, ["name", "department",
"salary"])
df.show()
```

Step 2: Define the Window Specification We need to import the `Window` class and some functions. Our window will be partitioned by department and ordered by salary.

Python

```
from pyspark.sql.window import Window
from pyspark.sql.functions import col, rank

# Our window is partitioned by department
# and ordered by salary in descending order within each
partition.
windowSpec =
Window.partitionBy("department").orderBy(col("salary").desc()
)
```

Step 3: Apply the `rank()` function Now we can use this specification with a window function like `rank()` in a `withColumn` call. The `.over(windowSpec)` tells Spark to apply the function using our defined window.

Python

```
ranked_df = df.withColumn("rank_in_dept",
rank().over(windowSpec))

ranked_df.show()
```

Result Analysis: Your output will look like this. Notice how the rank resets for each department. Michael is #1 in Sales, but Jen is #1 in Finance. The window function operated independently on each partition.

```
+-------+----------+------+------------+
|   name|department|salary|rank_in_dept|
+-------+----------+------+------------+
|    Jen|   Finance|  3900|           1|
|  Scott|   Finance|  3300|           2|
|  Maria|   Finance|  3000|           3|
|  Kumar| Marketing|  2000|           1|
|   Jeff| Marketing|  3000|           2|
|Michael|     Sales|  4600|           1|
| Robert|     Sales|  4100|           2|
+-------+----------+------+------------+
```

Other Common Window Functions

- **Aggregate Functions (`sum`, `avg`, `max`):** You can use standard aggregate functions to show a group-level aggregate on every row. For example, you could show the average department salary next to each employee's individual salary.
- **`lag()` & `lead()`:** These are powerful functions that allow you to access data from a previous (`lag`) or subsequent (`lead`) row within your ordered partition. This is great for calculating differences between time periods.

Window functions are a key tool for sophisticated data analysis, allowing you to create rankings, running totals, and perform period-over-period comparisons with ease.

Module 17: Introduction to Structured Streaming 🌊

Structured Streaming is Spark's high-level API for processing "data in motion." It's designed for use cases where data is continuously arriving and needs to be processed with low latency, such as IoT sensor data, web clickstreams, or financial market data.

The Core Concept: A Stream as an Unbounded Table

The genius of Structured Streaming is its core model: it treats a live data stream as a **continuously growing, unbounded table**.

This is a revolutionary idea because it means you can use the **exact same DataFrame API** you've already mastered to query a live data stream. You can `select`, `filter`, `groupBy`, and even `join` streams. Spark takes on the massive complexity of managing the continuous query, fault tolerance, and state management under the hood.

Analogy: Batch processing is like developing a roll of film. You wait until you have all 24 pictures, then process them all at once. Streaming is like a live video feed. You process each frame as it arrives, continuously updating the view.

Hands-On: A Live Word Count

The "Hello, World!" of streaming is a live word count. We'll set up a simple server on our machine and have Spark process the words we type in real-time.

Step 1: Start a Data Server (`netcat`) For this, you will need **two separate terminal windows**.

1. **Terminal 1:** This will be your PySpark shell (if it's not running, start it).
2. **Terminal 2:** This will be our simple data server. In this new terminal, run the following command:
3. Bash

   ```
   nc -lk 9999
   ```

4. This command starts `netcat` (`nc`), which will listen (`-l`) on port `9999`. Whatever you type into this terminal will be sent over the port once you hit Enter.

Step 2: Define the Stream Source (in PySpark) Now, go back to your PySpark terminal (Terminal 1). We'll create a DataFrame that connects to our `netcat` server.

Python

```
# In your pyspark shell
lines = spark.readStream \
```

```
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()
```
Notice we use `spark.readStream`. The `lines` DataFrame now represents the unbounded table of text data arriving from the socket.

Step 3: Apply Transformations We can apply the same word count logic we've used before, but with the DataFrame API.

Python

```
from pyspark.sql.functions import explode, split

# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

# Generate running word count
word_counts = words.groupBy("word").count()
```
So far, this is all lazy, just like in batch processing. We've defined the plan, but nothing is running yet.

Step 4: Define the Sink and Start the Query Now, we define where to send the results (the "sink") and start the query. We'll use the console as our sink for easy viewing.

Python

```
# Start the query that prints the running counts to the
console
query = word_counts.writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()
```
- `writeStream`: Kicks off the streaming write.
- `outputMode("complete")`: Says that the full, updated result table will be written to the sink each time there's new data.
- `.start()`: Actually starts the query running in the background.

Step 5: See it in Action!

1. Go to your **Terminal 2** (the `netcat` window).
2. Type `hello spark` and press Enter.

3. Type `hello world` and press Enter.

4. Look at your **Terminal 1** (PySpark). You will see a table of word counts printing to the screen, updating with each new batch of data! It will look something like this after a few updates:

```
# After first batch
+-----+-----+
| word|count|
+-----+-----+
|spark|    1|
|hello|    1|
+-----+-----+

# After second batch
+-----+-----+
| word|count|
+-----+-----+
|spark|    1|
|world|    1|
|hello|    2|
+-----+-----+
```

Step 6: Stop the Query To stop the background streaming query, run this in your PySpark shell:

Python

```
query.stop()
```

You've just built your first real-time data processing application. This powerful model is the foundation for building sophisticated, scalable, and fault-tolerant streaming pipelines.

Module 18: Packaging and Submitting Spark Applications 🚀

The interactive `pyspark` shell is a fantastic tool for development, testing, and exploration. However, when you want to automate a task or run a complex job on a schedule, you need a way to run it as a standalone application. The standard tool for this is `spark-submit`.

From Interactive Shell to Standalone Script

There's one key difference when writing a Python script that will be run by `spark-submit`: you must create and manage the `SparkSession` yourself. In the shell, the `spark` variable is created for you. In a script, you have to initialize it.

The standard boilerplate code for this looks like this:

Python

```
from pyspark.sql import SparkSession

# Standard Python entry point
if __name__ == "__main__":
    # Create a SparkSession
    spark = SparkSession.builder \
        .appName("MyFirstApp") \
        .getOrCreate()

    # YOUR SPARK CODE GOES HERE
    # e.g., df = spark.read.csv(...)

    # Stop the session to release resources
    spark.stop()
```

Hands-On: Your First `spark-submit` Job

We are going to create a Python script that performs a word count, then use `spark-submit` to run it.

Step 1: Create the Python Application Open any text editor and create a new file named `app.py`. Copy and paste the following code into it.

Python

```
import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split
```

```
if __name__ == "__main__":
    # Expecting two command-line arguments: input file and
output directory
    if len(sys.argv) != 3:
        print("Usage: app.py <input_file>
<output_directory>", file=sys.stderr)
        sys.exit(-1)

    input_path = sys.argv[1]
    output_path = sys.argv[2]

    spark =
SparkSession.builder.appName("WordCountApp").getOrCreate()

    # Read the input file into a DataFrame
    lines_df = spark.read.text(input_path)

    # Perform the word count transformation
    word_counts_df = lines_df.select(
        explode(split(lines_df.value, " ")).alias("word")
    ).groupBy("word").count()

    print("--- Word Count Results ---")
    word_counts_df.show()

    # Write the result to a Parquet file

word_counts_df.write.mode("overwrite").parquet(output_path)
    print(f"--- Results saved to {output_path} ---")

    spark.stop()
```

This script reads an input file path and an output directory path from the command line, performs a word count, shows the result, and saves it.

Step 2: Create a Sample Input File In your terminal, in the same directory where you saved `app.py`, create a simple text file.

Bash

```
echo "spark is great and spark is fast" > input.txt
```

Step 3: Run the Application with `spark-submit` Now, we'll use the `spark-submit` command, which is available in your terminal because you have Spark installed. This command tells Spark to run your application.

Bash

```
spark-submit app.py input.txt output_data
```

Let's break this command down:

- `spark-submit`: The command to launch a Spark application.
- `app.py`: The Python script you want to run.
- `input.txt`: The first command-line argument, which our script will use as the input path (`sys.argv[1]`).
- `output_data`: The second command-line argument, which will be our output directory (`sys.argv[2]`).

You will see Spark's log messages, then the output from our `show()` command, and finally a confirmation that the data was saved. You can check that a new directory named `output_data` has been created.

🎉 Congratulations! 🎉

You have officially completed the entire Big Data and Spark curriculum.

You started with the fundamental concepts of Big Data, understood distributed storage with HDFS, and then dove deep into Spark. You mastered the low-level RDD API, became proficient with the modern DataFrame and SQL APIs, and learned to handle real-time data with Structured Streaming. Finally, you've learned how to package and submit a real Spark application.

You now have a robust, end-to-end understanding of the Spark ecosystem.