# Distributed Machine Learning

Kanan Pandit
Sudam Kumar Paul

July 14, 2025

# Introduction

In the era of big data and deep learning, traditional machine learning methods face challenges in handling large-scale datasets and computationally expensive models.

- **Distributed Machine Learning (DML)** solves this problem by leveraging multiple computing resources to accelerate model training, improve efficiency, and enable large-scale AI applications.
- Modern frameworks like **H2O**, **Spark ML**, **TensorFlow Distributed**, and **PyTorch Distributed** provide powerful tools for implementing DML. However, deploying DML models comes with challenges such as synchronization overhead, communication bottlenecks, and fault tolerance.

This presentation explores DML architectures, types, benefits, challenges, and implementation strategies to help build scalable machine learning solutions.

# What is Distributed ML?

- It is a technique where ML tasks such as model training, evaluation, and data processing are distributed across multiple computing nodes instead of running on a single machine. This allows ML models to process massive datasets efficiently, reducing computational bottlenecks and accelerating training times.

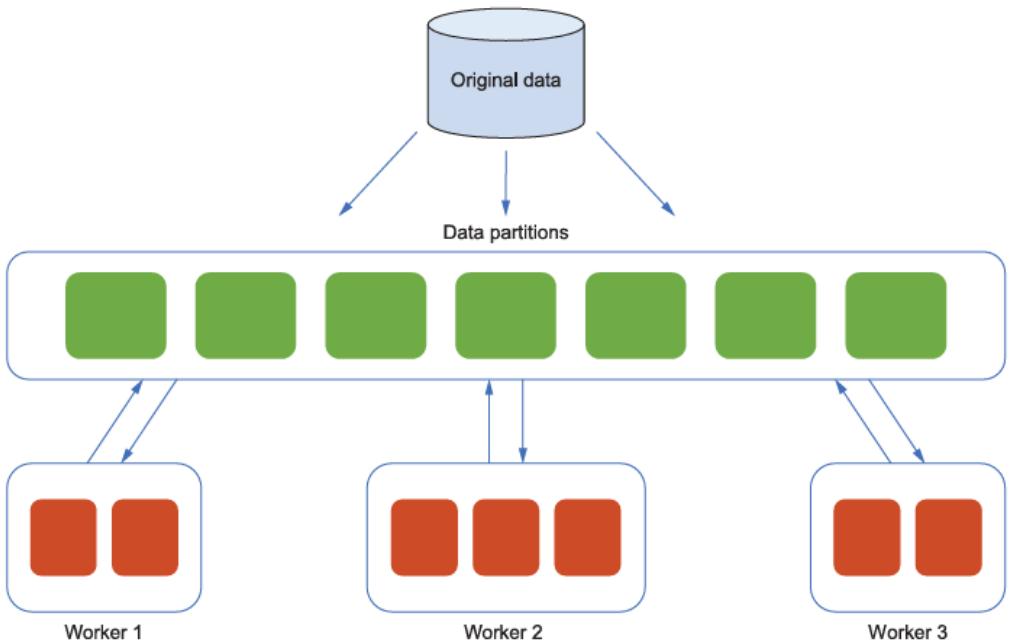


Figure: Working Procedure In DML

# Why Distributed ML is Needed?

Traditional machine learning runs on a single machine, limiting its ability to handle large-scale datasets and complex models. As data continues to grow exponentially, ML models require distributed computing to:

- **Handle Big Data:** Traditional machines struggle with high-dimensional and large-volume data.
- **Improve Speed & Efficiency:** Parallel processing speeds up model training and inference.
- **Utilize High-Performance Computing (HPC):** Leverages cloud computing and multi-GPU setups for deep learning.
- **Enhance Fault Tolerance:** Distributed systems can recover from individual node failures.

# Types of Distributed Machine Learning

DML can be categorized based on how data and computation are distributed:

| Type | Description | Example Frameworks |
|---|---|---|
| **Data Parallelism** | Data is split across multiple nodes, each training a model independently. | TensorFlow, PyTorch Distributed, H2O |
| **Model Parallelism** | Different parts of the model are trained on different nodes. | Megatron-LM (for large transformer models) |
| **Hybrid Parallelism** | A combination of data and model parallelism. | DeepSpeed, Horovod |
| **Federated Learning** | ML models are trained on decentralized devices without sharing raw data. | TensorFlow Federated, Flower |

- Now we will focus on Data Parallelism using **H2O** and **Spark**

# Introduction to SPARK

The Hadoop framework, with its HDFS and MapReduce data-processing engine, was the first that brought distributed computing to the masses. Hadoop solved the three main problems facing any distributed data-processing endeavor:

1. Parallelization
2. Distribution
3. Fault tolerance

While MapReduce enabled scalable data processing, it had key drawbacks:

- **High disk I/O**: Intermediate results were written to disk, slowing down performance.
- **Inefficient for iterative tasks**: Machine learning and graph algorithms suffered due to repeated disk reads/writes.
- **Rigid programming model**: Map and Reduce phases were too limiting for complex workflows.
- Poor support for real-time and interactive queries.

# Introduction to SPARK

Apache Spark is a fast, general-purpose distributed computing engine for big data. It was created to overcome Hadoop MapReduce's limitations by allowing in-memory processing, which makes it up to 100x faster for certain tasks.

**Key features:**

- Spark supports a wide range of workloads: batch, streaming, SQL, machine learning, and graph processing
- Works with Scala, Java, Python, and R.
- Keeping data in memory (instead of writing to disk repeatedly like MapReduce).
- Simplifying programming with high-level APIs
- Runs on local machines, Hadoop-YARN, Mesos, or Kubernetes.
- It integrates easily with HDFS, Amazon S3, Cassandra, Hive, Kafka, and many more.

# Key Components

Spark is made up of several core libraries:

1. **Spark Core**: Handles basic functions like task scheduling, memory management, fault recovery, and RDDs (Resilient Distributed Datasets).
2. **Spark SQL**: For working with structured data using SQL and DataFrames.
3. **Spark Streaming:** For real-time data processing using micro-batching.
4. **MLlib**: A scalable machine learning library with algorithms like regression, classification, clustering.
5. **GraphX**: For graph analytics and computations.

# Key Components

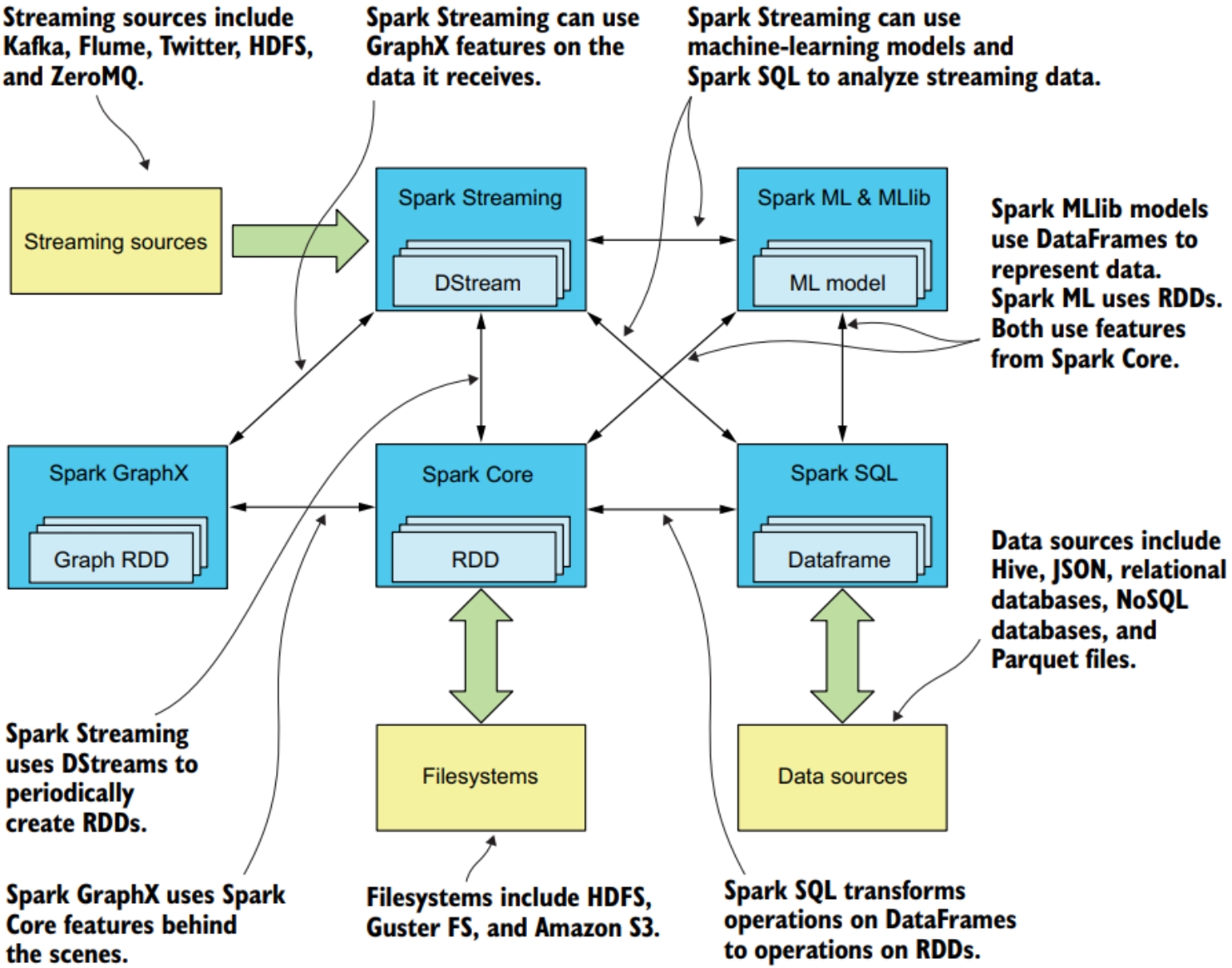


**Figure 1.2** Main Spark components and various runtime interactions and storage options

# Spark's Abstractions

Spark provides powerful abstractions to work with distributed data across a cluster. The main ones are:

1. RDD (Resilient Distributed Dataset)
2. DataFrame
3. Dataset
4. DStream (Discretized Stream)(for Spark Streaming)
5. Structured Streaming

# RDD

**RDD is the core data structure in Apache Spark**.
It just like a big list of data that is split and stored across many computers in a cluster. Even though the data is spread out, you can work with it as if it's just one normal list — like in Python or Scala.

- **Key Features:**
  1. Immutable: Once created, it can't be changed.
  2. Lazy Evaluation: Transformations are only executed when an action is called.
  3. Fault Tolerant: Spark can recover lost data using lineage (transformation history).
  4. In-memory: Can cache data in memory for faster computation.
- **Operations:**
  1. Transformations: Return a new RDD (e.g., map, filter, flatMap)
  2. Actions: Trigger execution (e.g., collect, count, reduce)
- **Use Cases:**
  1. Ideal for custom, fine-grained data processing
  2. Best when working with unstructured or semi-structured data

# DataFrame

A DataFrame is a table-like structure in Apache Spark with rows and columns, similar to an Excel sheet or a SQL table.

- **Key Features:**
  1. Stores structured data (with column names and data types).
  2. Allows SQL-like operations (e.g., select, filter, groupBy).
  3. Built on top of RDDs, but more efficient due to built-in optimizations.
  4. Supports reading from CSV, JSON, Parquet, databases, etc.
- **Use Cases:** Spark SQL, Data Analysis, and Machine Learning workflows.

# Dataset

A Dataset is a distributed, structured collection of data in Spark that combines the performance of DataFrames with the type-safety of RDDs.

- **Key Features:**
  1. Available in Scala and Java (not in Python).
  2. Supports compile-time type checking.
  3. Allows functional operations like map, filter, groupBy.
  4. Best of RDD (type safety) + DataFrame (speed and structure).
- **Use Cases:**
  1. Ideal for building structured pipelines where you load data from sources, transform it, and save the output.
  2. Handling Complex Data Types (data includes nested structures e.g., case classes, arrays, maps).
  3. Type-Safe Data Processing (compile-time error checking)

# SPARK Architecture

Spark's architecture is built around a master-slave model and follows a driver-executor pattern.

**Key Components:**

**1.Driver Program**

- Acts as the master.
- Runs the main() function of your Spark program.
- Responsible for:
  1. Converting code into tasks
  2. Scheduling tasks across workers
  3. Coordinating the whole application
- Maintains SparkContext — the entry point to Spark functionality.
  1. It acts like the connection between your application and the Spark Cluster.
  2. It sets up a connection to the cluster, allocates resources, and manages the execution.
  3. It is the starting point for any Spark application.
  4. Manages configurations and environment

# SPARK Architecture

**2.Cluster Manager**

- Allocates resources (CPU, memory) for Spark applications.
- Spark can work with:
  - Standalone Scheduler (built-in)
  - YARN (Hadoop)
  - Kubernetes (in later versions)

**3.Executors**

- Run on worker nodes.
- Executing tasks assigned by the driver
- Storing data in memory or disk (caching, shuffling)
- Reporting back the status/results to the driver

# SPARK Architecture

**4. Tasks**

- A unit of work sent by the Driver to the Executor.
- Spark splits jobs into stages (A group of tasks that can be executed in parallel), and each stage is broken into tasks.

**Execution Flow:**

- Submit Application to the Driver Programme (via spark-submit or shell) / User Code (e.g., in Scala, Python, Java) triggers Spark actions.
- The Driver is launched and creates a SparkContext (or SparkSession). It builds a DAG (Directed Acyclic Graph) of transformations.
- The Driver asks the Cluster Manager (Standalone, YARN, Mesos, or Kubernetes) to allocate resources.
- Cluster Manager launches Executor processes on worker nodes.
- Driver Sends Tasks to Executors.
- Tasks Are Executed in Executors.
- Optionally, data is cached in memory for faster reuse.
- Result Returned to Driver.

# SPARK Architecture Picture

https://spark.apache.org/docs/latest/cluster-overview.html

# Spark_Cluster Setup

Follow the instructions provided in the **Spark_cluster setup.txt** file and run the given **sample_prog.py** file.

# Spark Workflow With Sample ML Tasks

1.
*from pyspark.sql import SparkSession*
*from pyspark.ml.feature import VectorAssembler, StandardScaler*
*from pyspark.ml.clustering import KMeans*
*from pyspark.ml.evaluation import ClusteringEvaluator*

**Architecture:** The Driver loads required Spark ML and SQL libraries. These will be used to create the pipeline and transformations.
**In short:** Your entire Python script is the Driver program. When you run this file, your machine (or the machine running the script) becomes the Driver.

# Spark Workflow With Sample ML Tasks

2.

*spark = SparkSession.builder*
*.appName("KMeansClustering")*
*.getOrCreate()*

**Architecture:** Driver + SparkContext

- The Driver program is launched.
- It creates a SparkSession, which internally initializes a SparkContext.
- The SparkContext contacts the Cluster Manager to request Executors (worker processes on worker nodes). Since you have 2 worker nodes, each worker node will likely start one or more Executors based on available CPU/memory.

# Spark Workflow With Sample ML Tasks

3.
*df = spark.read.csv('iris_dataset.csv', header=True, inferSchema=True)*

**Architecture:** Driver + Executors

- Spark splits the iris_dataset.csv into multiple partitions.
- Spark creates a DataFrame and each partition are distributed across both worker nodes.
- Each worker node reads a portion of the file in parallel.
- Schema inference happens using distributed tasks (You are asking Spark to automatically figure out the data types (like Integer, Double, String) for each column based on the contents of the CSV file. This is called schema inference.)

# Spark Workflow With Sample ML Tasks

4.
*print(f"Number of rows: df.count(), Number of columns: len(df.columns)")*
*df.printSchema()*

**Architecture:** Action (Triggers Execution)

- count() is an action, so the Driver creates a DAG, breaks it into stages, and sends tasks to the executors to compute row counts.
- Executors perform computation and return the result to the Driver.
- Print the structure of your DataFrame.

# Spark Workflow With Sample ML Tasks

5.
*input_cols = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']*

**Architecture:** Driver Only

- Defines the columns to be used for clustering.
- No Spark action/transformation yet.

# Spark Workflow With Sample ML Tasks

6.

*vec_assembler = VectorAssembler(inputCols=input_cols,*
*outputCol='features')*
*df = vec_assembler.transform(df)*

**Architecture:** Transformation (lazy)

- The DataFrame transformation is lazy — Spark builds a new logical plan but doesn't execute yet.
- No real computation happens until an action is triggered.

# Spark Workflow With Sample ML Tasks

7.
*scaler = StandardScaler(inputCol='features', outputCol='scaled_features', withStd=True, withMean=False)*
*scaler_model = scaler.fit(df)*
*df = scaler_model.transform(df)*

**Architecture:** fit() triggers action

- fit() causes Spark to compute summary statistics across partitions (Executors do this).
- A Transformer model (scaler_model) is created and then applied (transform()), generating a new DataFrame.

# Spark Workflow With Sample ML Tasks

8.

```
for k in range(2, 10):
kmeans = KMeans(k=k, featuresCol='scaled_features',
predictionCol='prediction') model = kmeans.fit(df)
predictions = model.transform(df)
evaluator = ClusteringEvaluator()
silhouette = evaluator.evaluate(predictions)
silhouette_scores.append(silhouette)
print(f"Silhouette score for k=k: silhouette")
```

**Architecture:** Model Training and Evaluation

- For each k, a new KMeans model is trained.
- fit() = action, so spark builds a DAG of stages/tasks and sends tasks to executors to compute centroids and assign clusters.
- transform() assigns cluster labels.
- evaluate() computes Silhouette score using executors, then result returns to the driver.

# Spark Workflow With Sample ML Tasks

9.
*plt.plot(range(2, 10), silhouette_scores)*

**Architecture:** Driver Only

- Done using matplotlib on the driver machine after collecting scores.

# Spark Workflow With Sample ML Tasks

10.
*kmeans = KMeans(k=3, featuresCol='scaled_features', predictionCol='prediction')*
*model = kmeans.fit(df) predictions = model.transform(df)*

**Architecture:** Same as earlier

- fit() = action → DAG → stages → tasks sent to executors.
- Executors cluster data.
- transform() adds predictions.

# Spark Workflow With Sample ML Tasks

11.
*predictions.groupBy('prediction').count().show()*

**Architecture:** Action

- groupBy().count() is a wide transformation (data shuffle).
- Spark breaks this into stages, runs tasks on executors, and returns the grouped result to the Driver.

# Spark Workflow With Sample ML Tasks

12.
*pandas_df = predictions.select(...).toPandas()*

**Architecture:** Action + Collect

- Triggers execution.
- Results are collected from all executors to the driver as a Pandas DataFrame.

# Spark Workflow With Sample ML Tasks

13.
*plt.figure()*
*ax = fig.add_subplot(...)*
*ax.scatter(...)*

**Architecture:** Driver Only

- Local plotting happens on the Driver machine using matplotlib.

# Spark Workflow With Sample ML Tasks

14.
*spark.stop()*

**Architecture:** Driver Only

- Gracefully shuts down the SparkContext and stops communication with the Cluster Manager.
- Executors are terminated.

# Comparison Between Apache Spark and H2O

| Aspect | Apache Spark | H2O |
|---|---|---|
| **Overview** | Distributed computing for big data processing. | Machine learning platform optimized for large-scale models. |
| **Core Strengths** | Versatile, supports ETL, batch, streaming, and ML. | Optimized for machine learning with AutoML and deep learning support. |
| **Machine Learning** | MLlib (fewer algorithms), supports pipelines. | Specialized in ML algorithms (e.g., GLM, GBM, deep learning). |
| **Big Data Integration** | Strong integration with Hadoop and other big data tools. | Integrates with Spark through Sparkling Water for ML tasks. |
| **Performance** | In-memory processing and distributed computation. | In-memory, GPU-accelerated ML, highly optimized for model training. |
| **Ease of Use** | Requires programming (Scala, Python, Java). | User-friendly web interface (H2O Flow), AutoML. |
| **Deployment & Scalability** | Highly scalable on clusters, cloud-compatible. | Scalable with cluster support and easy model deployment via REST API. |
| **Community** | Large, active community with extensive resources. | Growing community, particularly in data science. |

# Conclusion:

- **Apache Spark** is ideal for general-purpose big data processing, supporting diverse workloads like ETL (Extract, Transform, Load), real-time streaming, and batch processing. Its flexibility and integration with big data ecosystems make it a powerful tool for distributed computing. Works great when handling large-scale data pipelines and integrating with tools like Hadoop, Kafka, etc.
- **H2O**, on the other hand, excels in specialized machine learning tasks, providing optimized algorithms for model training and evaluation. Its AutoML capabilities and support for deep learning make it particularly suited for data science applications where ease of use and speed are crucial( simple and beginner-friendly a tool is, requiring minimal effort to operate and get results).