Distributed Machine Learning

Kanan Pandit
Sudam Kumar Paul

April 26, 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.

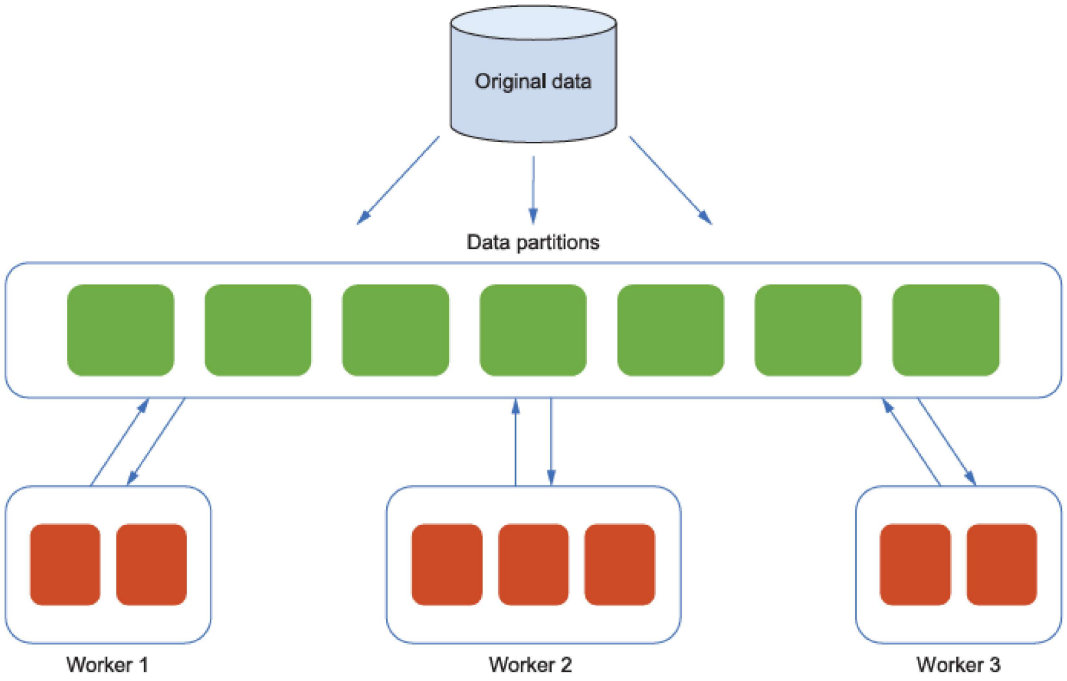


Figure: Working Procedure In DML

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 H2O

H2O is an open-source, distributed, in-memory machine learning platform designed for big data analytics.It enables fast and scalable model training while supporting a wide range of machine learning algorithms. **Key Features:**

1. **High Performance:** In-memory computing for faster processing.
2. **Scalability:** Works on local machines, cloud, and distributed environments.
3. **Ease of Use:** Supports Python, R, Java, and Web UI (Flow).
4. **AutoML:** Automates model selection and hyperparameter tuning.
5. **Integration:** Connects with HDFS, S3, SQL databases, and local storage.

H2O: Addressing Machine Learning Challenges

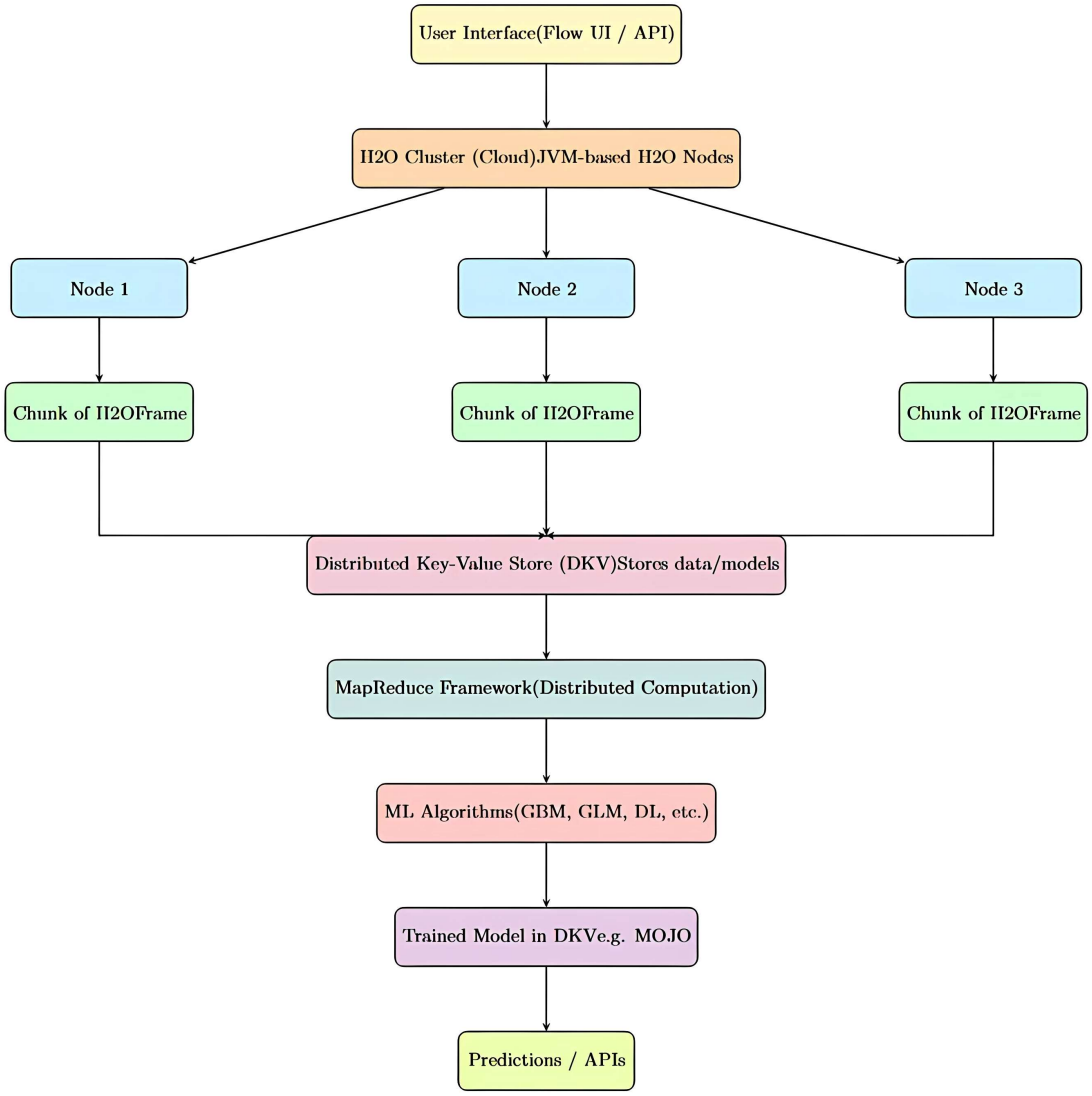
1. **Scalability Issues:** H2O distributes computations across multiple nodes for parallel processing, making it efficient for large datasets.
2. **Slow Model Training:** In-memory processing speeds up training and inference compared to traditional disk-based methods.
3. **Complex Model Selection:** H2O's AutoML automates algorithm selection and hyperparameter tuning, reducing manual effort.
4. **Big Data Handling:** It can effortlessly work with Hadoop, HDFS, AWS S3, and SQL databases for large-scale data processing.
5. **Lack of Interpretability:** H2O delivers explainable AI tools (such as SHAP and LIME) for enhanced model transparency.

H2O makes machine learning fast, scalable, and accessible, overcoming key ML challenges in industries like finance, healthcare, and predictive analytics.

H2O Architecture

H2O is a distributed, in-memory machine learning platform designed for fast scalable analytics. The architecture supports running on clusters of machines (multi-node) and can be integrated with tools like R, Python, Java, and Scala.

H2O General Architecture



Key Architectural Components of H2O

1. H2O Cluster (Cloud)

1. A group of H2O nodes working together.
2. Each node handles data storage, computation, and communication. All nodes are equal (peer-to-peer, no master-slave).
3. They communicate using flatfile or multicast (for discovery and coordination).
4. Data is automatically distributed across nodes.

2. H2O Node

1. Each node is a JVM process.
2. Nodes contain **Data storage**, **Execution engine**, **Communicationlayer**.
3. Nodes can read/write data, perform computations, and collaborate with others.

Key Architectural Components of H2O

3. H2O Core Components:

1. **Distributed Key-Value Store(DKV)**:
   - Stores data frames, models, and metadata.
   - Keys are used to access objects.
2. **MapReduce Framework**:
   - Executes distributed algorithms.
   - Built on top of DKV.
   - Supports parallel data processing.
3. **Distributed Data Frames(H2O Dataframe)**:
   - Analogous to R/Pandas DataFrames.
   - Partitioned across nodes.

Inference

1. Python, R, Java, REST are Language-specific interfaces to interact with H2O for interaction.
2. **Flow Web UI**: Visual interface for managing H2O workflows.

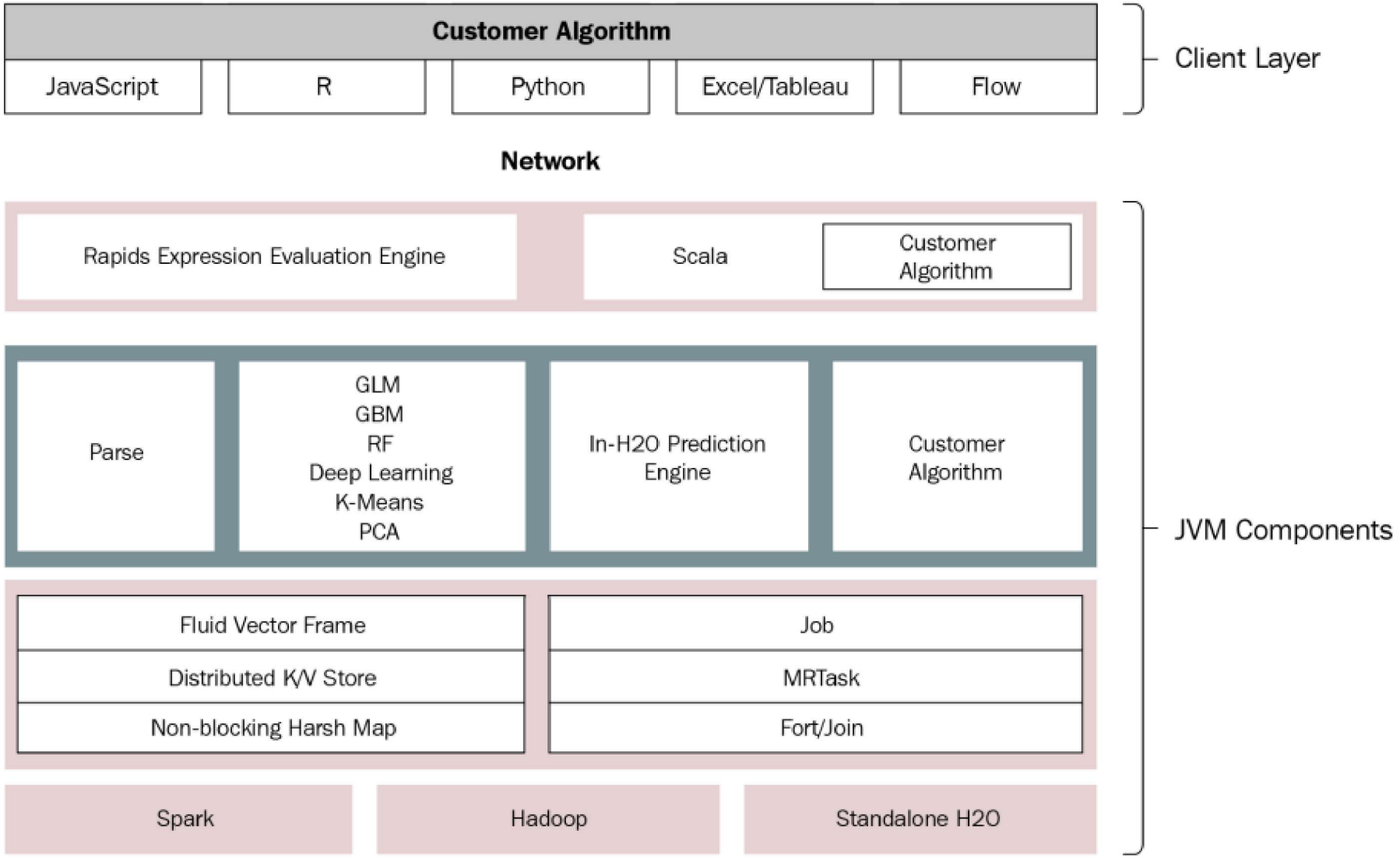
Execution Workflow

1. Load data $\rightarrow$ Stored in H2OFrame $\rightarrow$ Distributed across nodes internally.
2. Algorithms (e.g., GBM, XGBoost, GLM) run using MapReduce.
3. Model is trained and results stored back in DKV $\rightarrow$ Accessible via API or Flow.

Scalability and Performances

1. In-memory data representation boosts speed.
2. Data locality ensures minimal shuffling.
3. Fault-tolerant to some extent due to distributed design.

H2O Cluster Architecture



H2O Cluster Architecture

1. **Client Layer (User Interface & APIs)**: This is how users interact with H2O. **R / Python** programming languages used to run machine learning models on H2O and **Flow** is a web-based UI for building models without coding.
2. **Network**: It is nothing but the general internet, which requests are sent over.
3. **Java Virtual Machine (JVM) components**:It indicates the H2O server and all of its JVM components that are responsible for the different functionalities of H2O AI, including AutoML.

Interactions of various H2O clients with the same H2O server

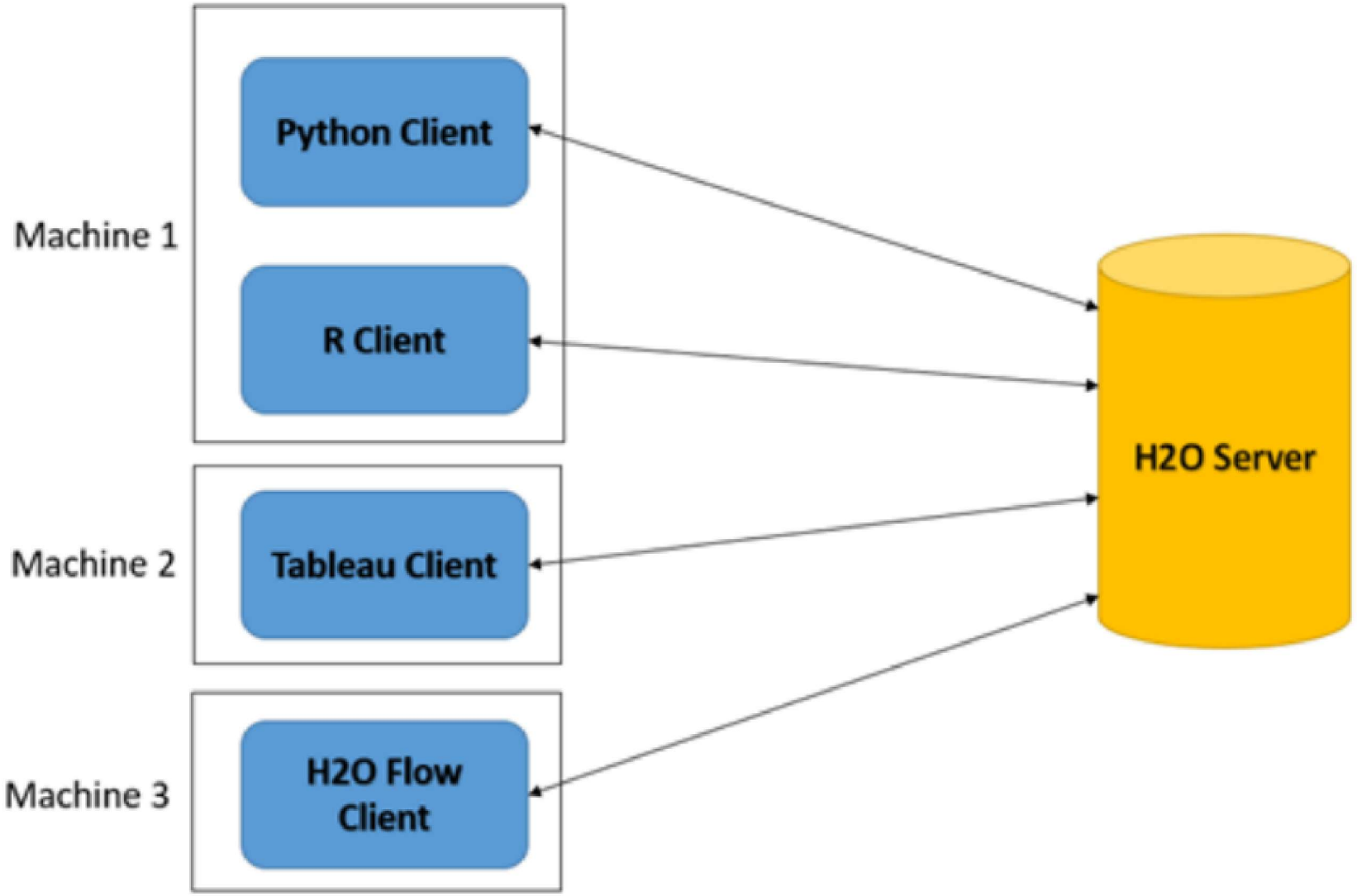


Figure 4.3 – Different clients communicating with the same H2O server

JVM Components

- The JVM is a runtime engine that runs Java programs in your system. The H2O cloud server runs on multiple JVM processes, also called JVM nodes. Each JVM node runs specific components of the H2O software stack.

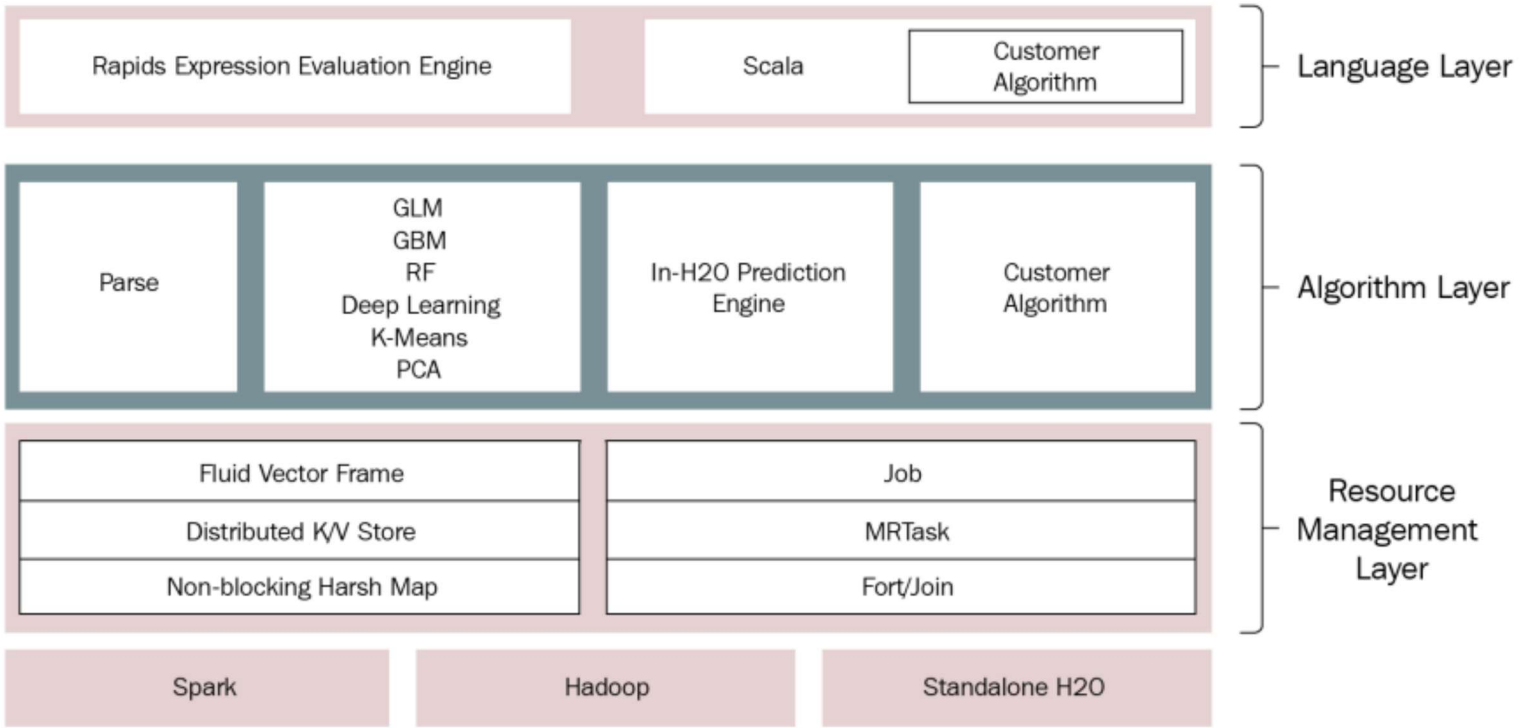


Figure 4.4 – H2O JVM component layer

JVM Components

1. **Language Layer**: This layer contains processes that are responsible for evaluating the different client language expressions that are sent to the H2O cloud server.
2. **Algorithm layer**: It handles Loading data, Training models using algorithms like GLM, GBM, Deep Learning, etc., Making predictions with the trained models, Running custom algorithms if added.
3. **Resource management layer**: It manages system resources like memory and CPU during machine learning tasks. It ensures everything runs smoothly by handling tasks, jobs, and how data is stored and shared across the system.

Resource Management layer

1. **Fluid Vector Frame:** A flexible, efficient, and mostly immutable data storage format (like a DataFrame) used in H2O.
2. **Distributed Key-Value Store:** A fast in-memory system to store and retrieve data using key-value pairs across the cluster.
3. **NonBlockingHashMap:** A scalable, lock-free map that allows multiple processes to access/update data without conflicts.
4. **Job:** Represents a single unit of work; H2O manages jobs to perform complex tasks efficiently.
5. **MRTask:** H2O's version of MapReduce for running data operations in parallel across the cluster.
6. **Fork/Join:** Splits tasks into smaller parts (forks) and combines results (joins) using Java's jsr166y library.

The entire JVM component layer lies on top of Spark and Hadoop data processing systems and H2O can also run independently on a single machine or a cluster. The components in the JVM layer leverage these data processing cluster management engines to support cluster computing.

Learning about H2O client-server interactions during the ingestion of data

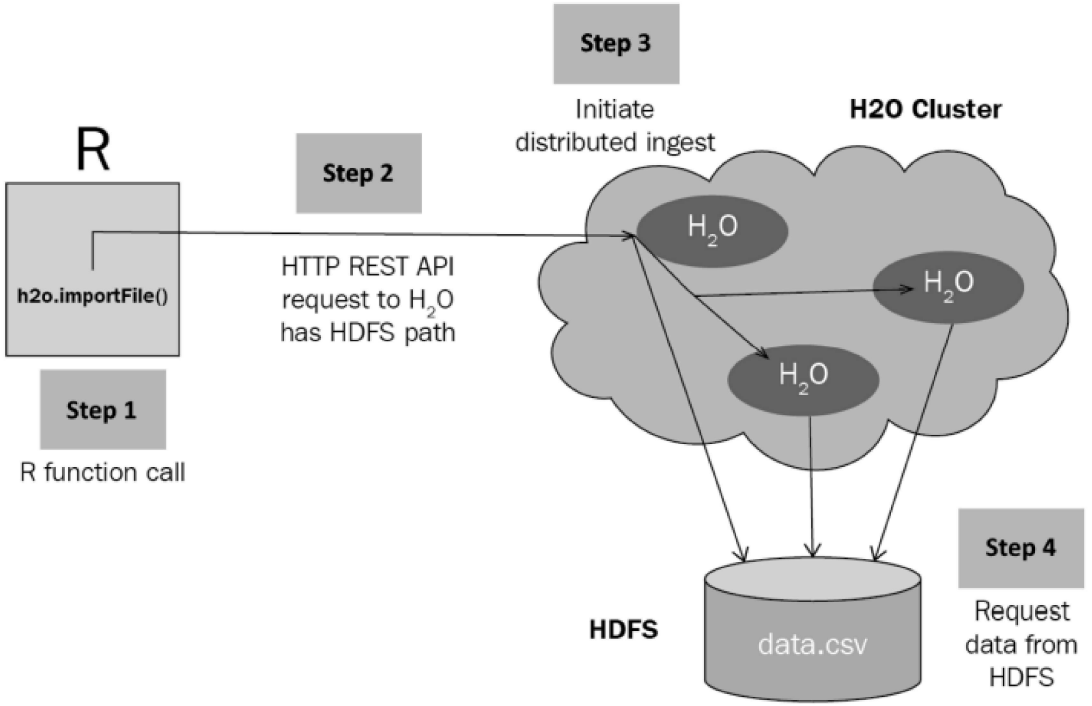


Figure 4.5 – H2O data ingestion request interaction flow

The following sequence of steps describes how a client request to the H2O cluster server to ingest data is serviced by H2O using HDFS.

Understand the flow of interaction once data is ingested and H2O returns a response:

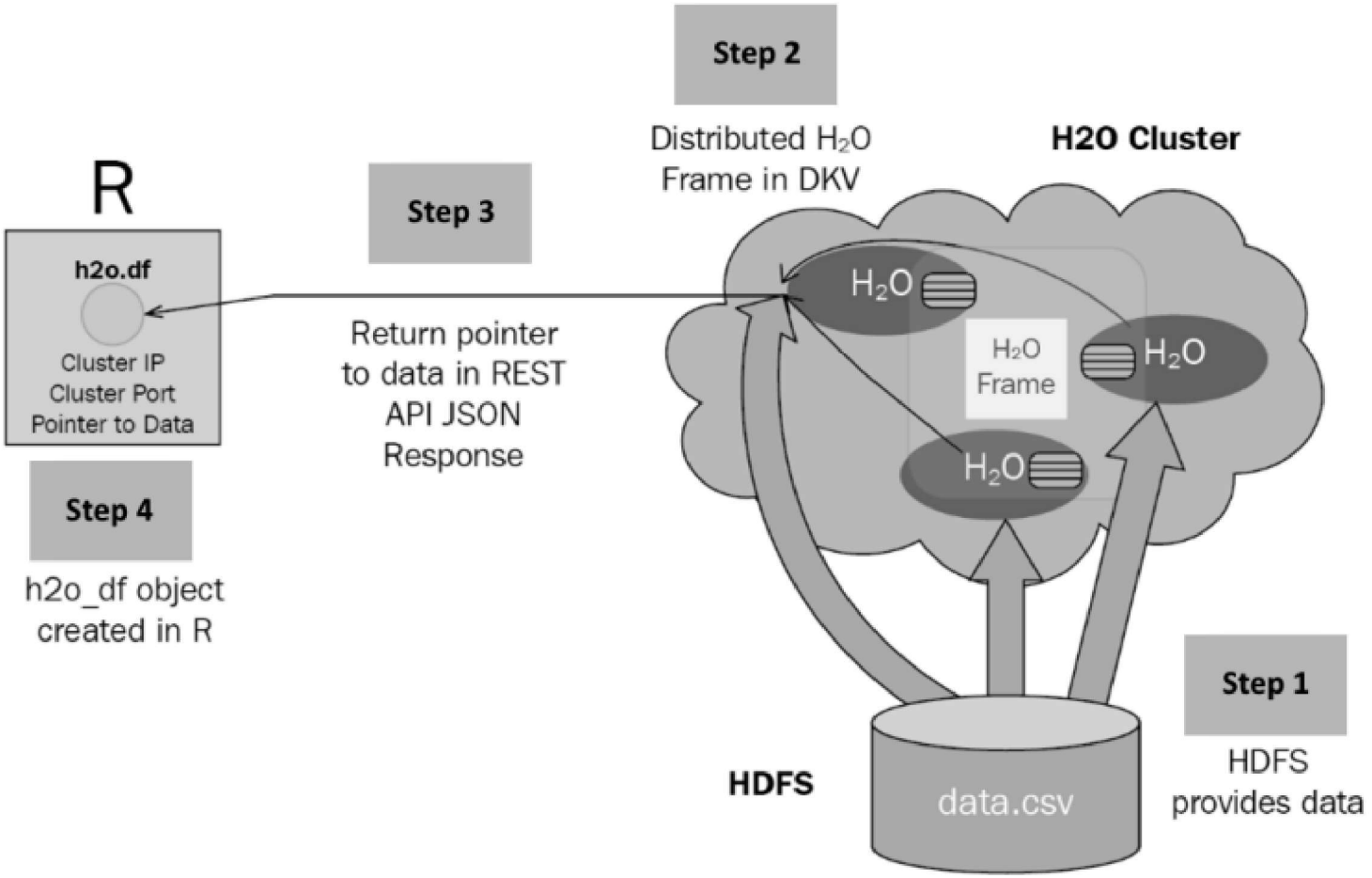


Figure 4.6 – H2O data ingestion response interaction flow

H2O Installation and Cluster Setup

Follow the instructions provided in the **H2O cluster setup.txt** file and run the given **sample prog.py** file.

Step-by-Step Workflow Using H2O Architecture With a Sample ML Task

1. Connecting to H2O Cluster:
`import h2o` (Acts as the interface between the user and the H2O cluster using a REST API.)
`from h2o.frame import H2OFrame` (H2O's equivalent of pandas DataFrame. An H2OFrame is stored in distributed memory across nodes in the H2O cluster.)
`from h2o.estimators import H2ORandomForestEstimator`
`h2o.init(ip="172.20.252.53", port=54323)` (Connects to the H2O Leader Node, which routes tasks to other worker nodes.)

Architecture Components:
User Interface (Flow UI / API) → H2O Cluster (Cloud)

- These lines connect your Python environment (client-side API) to the H2O cluster running at the specified IP and port.
- h2o.init() creates a link to the backend JVM-based H2O cluster.

Step-by-Step Workflow Using H2O Architecture With a Sample ML Task

2. Cluster Status Check:
`print(h2o.cluster_status())` (Prints the health and configuration of the connected H2O cluster.)

Architecture Components:

- Makes a REST API call to the leader node to fetch metadata about the cluster's health, node count, and memory usage.
- Ensures that multiple nodes (Node 1, Node 2, etc.) are active and communicating.

Visualization of Cluster Status

```
sysadm@bhaskara07:~/Videos/H2O_cluster$ source env/bin/activate
(env) sysadm@bhaskara07:~/Videos/H2O_cluster$ python3 sample_prog.py
Checking whether there is an H2O instance running at http://172.20.252.53:54323. connected.
Warning: Your H2O cluster version is (5 months and 14 days) old.  There may be a newer version available.
Please download and install the latest version from: https://h2o-release.s3.amazonaws.com/h2o/latest_stable.html
--------------------------  --------------------------
H2O_cluster_uptime:         1 min 29 secs
H2O_cluster_timezone:       Asia/Kolkata
H2O_data_parsing_timezone:  UTC
H2O_cluster_version:        3.46.0.6
H2O_cluster_version_age:    5 months and 14 days
H2O_cluster_name:           trial1
H2O_cluster_total_nodes:    2
H2O_cluster_free_memory:    4 Gb
H2O_cluster_total_cores:    8
H2O_cluster_allowed_cores:  8
H2O_cluster_status:         locked, healthy
H2O_connection_url:         http://172.20.252.53:54323
H2O_connection_proxy:       null
H2O_internal_security:      False
Python_version:             3.12.3 final
--------------------------  --------------------------
/home/sysadm/Videos/H2O_cluster/sample_prog.py:9: H2ODeprecationWarning: Deprecated, use ``h2o.cluster().show_status(True)``.
  print(h2o.cluster_status())
--------------------------  --------------------------
H2O_cluster_uptime:         1 min 30 secs
H2O_cluster_timezone:       Asia/Kolkata
H2O_data_parsing_timezone:  UTC
H2O_cluster_version:        3.46.0.6
H2O_cluster_version_age:    5 months and 14 days
H2O_cluster_name:           trial1
H2O_cluster_total_nodes:    2
H2O_cluster_free_memory:    4 Gb
H2O_cluster_total_cores:    8
H2O_cluster_allowed_cores:  8
H2O_cluster_status:         locked, healthy
H2O_connection_url:         http://172.20.252.53:54323
H2O_connection_proxy:       null
H2O_internal_security:      False
Python_version:             3.12.3 final
--------------------------  --------------------------
```

Visualization of Cluster Status

```
Nodes info:     Node 1                          Node 2
-------------   -----------------------------   -------------------------------
h2o             bhaskara07/172.20.252.53:54323  172.20.252.58/172.20.252.58:54323
healthy         True                            True
last_ping       1744775795914                   1744775795677
num_cpus        4                               4
sys_load        0.40283203                      0.14746094
mem_value_size  0                               0
free_mem        2147483648                      2147483648
pojo_mem        0                               0
swap_mem        0                               0
free_disk       75451334656                     79105622016
max_disk        97828995072                     97828995072
pid             23283                           170491
num_keys        0                               0
tcps_active     0                               0
open_fds        30                              30
rpcs_active     0                               0
None
Parse progress: |                                                                (done)| 100%
drf Model Build progress: |                                                      (done)| 100%
ModelMetricsMultinomial: drf
** Reported on test data. **

MSE: 0.037848213706600464
RMSE: 0.19454617371359548
LogLoss: 0.10490168432378205
Mean Per-Class Error: 0.14444444444444446
AUC table was not computed: it is either disabled (model parameter 'auc_type' was set to AUTO or NONE) or the domain size exceeds the limit (maximum is 50 domains).
AUCPR table was not computed: it is either disabled (model parameter 'auc_type' was set to AUTO or NONE) or the domain size exceeds the limit (maximum is 50 domains).

Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
setosa    versicolor    virginica    Error      Rate
--------  ------------  -----------  ---------  ------
14        0             0            0          0 / 14
0         9             1            0.1        1 / 10
0         1             2            0.333333   1 / 3
14        10            3            0.0740741  2 / 27
```

Step-by-Step Workflow Using H2O Architecture With a Sample ML Task

3. Load Dataset:

df = h2o.import_file("https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv")

Architecture Components:

Chunks of H2OFrame in Nodes

- H2O Core Engine running on the JVM cluster (both leader and worker nodes) do that process.
- The CSV is splited and stored as an H2OFrame, which is distributed across the nodes (Node 1, Node 2).
- Each chunk is processed in parallel by different worker nodes.

Step-by-Step Workflow Using H2O Architecture With a Sample ML Task

4. Convert Target to Categorical Variable for Classification

df["species"] = df["species"].asfactor()

Architecture Components:

- It happens inside the JVM-based H2O backend, specifically in the Distributed K/V Store layer + Frame metadata layer, and triggered via the Rapids Expression Engine.
- The species column's metadata is updated in-place, across the cluster — this is a distributed in-memory transformation.

Step-by-Step Workflow Using H2O Architecture With a Sample ML Task

5. Split the Data

train, test = df.split_frame(ratios=[0.8], seed=1234)

Architecture Components:

H2O Cluster / MapReduce Framework

- Uses MapReduce logic to split the H2OFrame into training and testing datasets, distributed across nodes.
- Again, stored in DKV.

6. Define the Model

rf_model = H2ORandomForestEstimator(ntrees=50, max_depth=10, nfolds=5)

Architecture Components:

ML Algorithms (e.g., Random Forest)

- We are defining a Random Forest model here with cross-validation (nfolds=5) to be trained across the cluster using distributed computation.

Step-by-Step Workflow Using H2O Architecture With a Sample ML Task

7. Train the model

rf_model.train(x=df.columns[:-1], y="species",training_frame=train)

Architecture Components:

Algorithm Layer and Resource Management Layer inside the distributed JVM-based nodes of the H2O cluster.

- The Algorithm Layer contains the core ML logic (e.g., Random Forest).
- Parallelism: Trees are trained in parallel across worker nodes using data-parallel processing. The Resource Management Layer handles data chunking, task distribution, and parallel execution using MRTask, Fork/Join, and the Distributed Key-Value Store.

Step-by-Step Workflow Using H2O Architecture With a Sample ML Task

8. Evaluate the Model

```
performance = rf_model.model_performance(test_data=test)
print(performance)
```

Architecture Component:

Trained Model in DKV → Predictions / APIs

- Algorithm Layer – for scoring logic.
- Resource Management Layer – for parallel execution on chunks: Split the test data across nodes, Run scoring in parallel and Collect results.
- Distributed Key/Value Store (DKV) – where test data and model are stored. During model_performance(), these are fetched directly from memory.

Step-by-Step Workflow Using H2O Architecture With a Sample ML Task

9. Shut Down the Cluster
h2o.cluster().shutdown()

Architecture Component:

User Interface → H2O Cluster

- Shuts down the H2O instance to free resources.