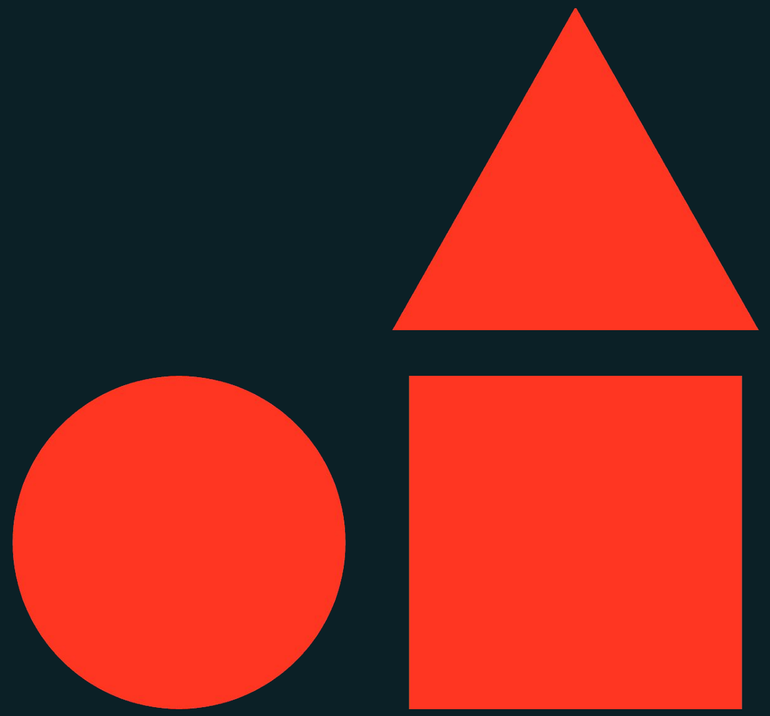
databricks

Machine Learning at Scale



Databricks Academy
May 2025

Agenda

01. Machine Learning Development with Spark	Lecture	Demo	Lab
A Brief Overview of Spark Architecture for Machine Learning	✓		
Introduction to Spark ML for Model Development	✓		
Model Tracking and Packaging with MLflow and Unity Catalog on Databricks	✓		
Model Development with Spark		✓	✓
02. Distributed Model Tuning on Databricks	**Lecture**	**Demo**	**Lab**
Overview of Hyperparameter Tuning	✓		
Scalable HPO Frameworks on Databricks	✓		
Hyperparameter Tuning with SparkML		✓	
HPO with Ray Tune		✓	

Agenda

03. Deploying Machine Learning Models with Spark	Lecture	Demo	Lab
Deployment with Spark	✓		
Inference with Spark	✓		
Model Deployment with Spark		✓	
Optimization Strategies with Spark and Delta Lake		✓	
Model Deployment with Spark			✓
04. Pandas on Spark	**Lecture**	**Demo**	**Lab**
Scaling with Pandas APIs	✓		
Pandas UDFs and Function APIs	✓		
Pandas APIs		✓	✓
Recap and CTA	✓		

Course Learning Objectives

- Understand Spark's Architecture and how it supports ML workloads within Databricks
- Know when it is appropriate to utilize Spark
- Explain and demonstrate how to use Spark ML for data preparation, model training, and model evaluation
- Explain and demonstrate how to use Hyperopt, Optuna and Ray for tuning hyperparameters on Databricks
- Explain and demonstrate how to Track and package models with MLflow and Unity Catalog with Spark
- Explain and demonstrate optimization strategies with Spark and Delta Lake
- Explain and demonstrate model deployment and inference scalability with Spark
- Explain and demonstrate how pandas APIs work on Spark

Prerequisites/Technical Considerations

Things to keep in mind before you work through this course

Prerequisites

1. Intermediate-level knowledge of traditional machine learning concepts
2. Intermediate-level experience with traditional machine learning development on Databricks
3. Intermediate-level knowledge of Python for machine learning projects

Technical Considerations

1. A cluster running on **DBR ML 16.3**
2. **Unity Catalog** and **Model Serving** enabled workspace

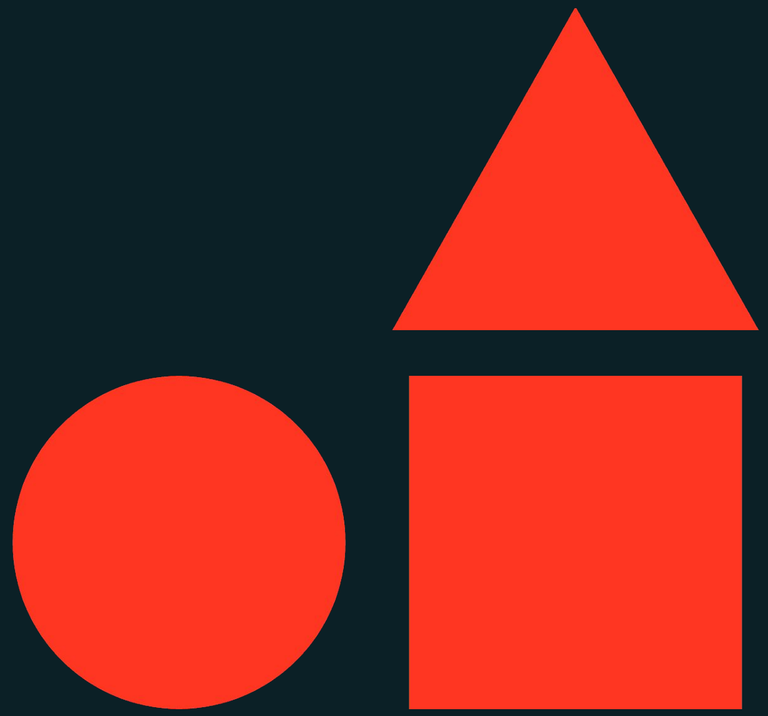
Lab Exercise Environment

Technical Details

- Your lab environment is provided by Vocareum.
- It will open in a new tab.
- It has been configured with the permissions and resources required to accomplish the tasks outlined in the lab exercise.
- Third party cookies must be enabled in your browser for Vocareum's user experience to work properly.
- Make sure to enable pop ups!

databricks

Machine Learning Development with Spark



Databricks Academy

databricks

Machine Learning Development with Spark

LECTURE

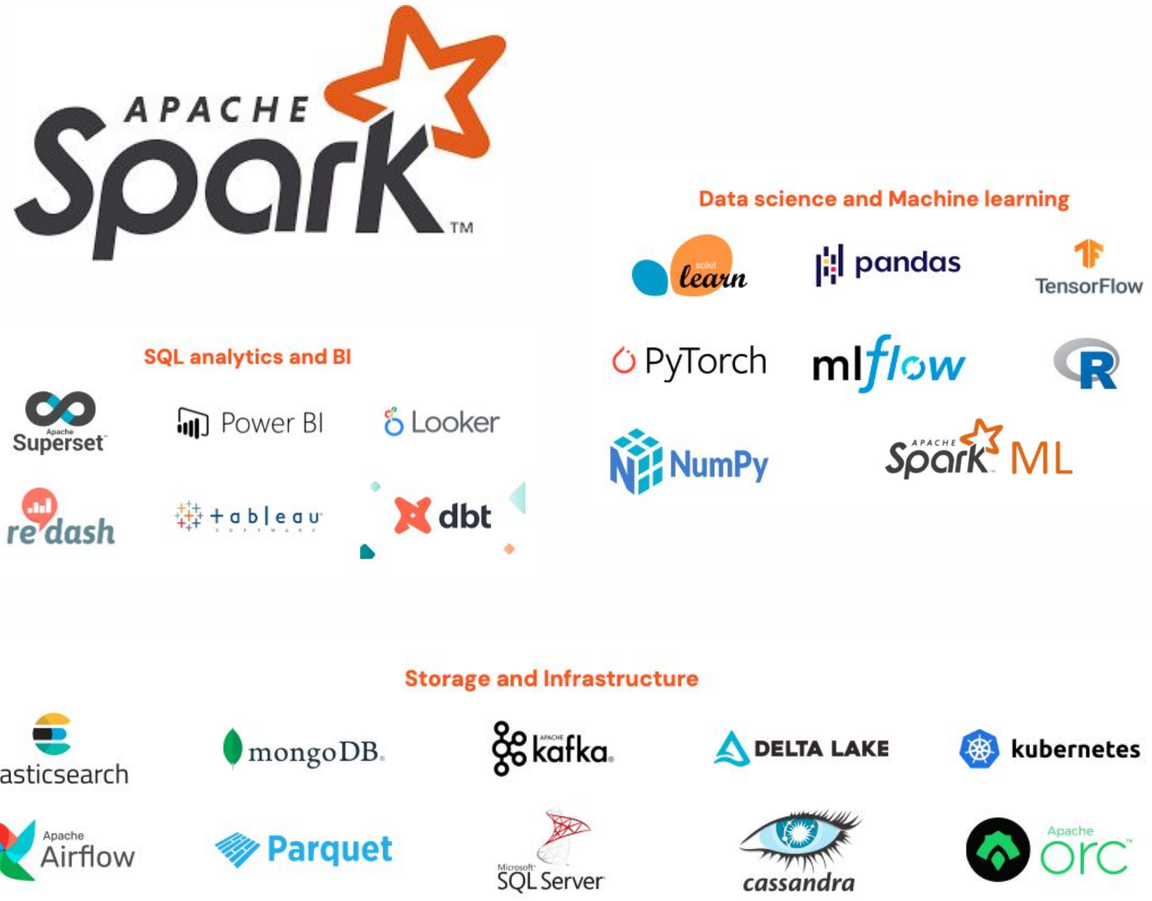
A Brief Overview of Spark Architecture for Machine Learning

Databricks Academy

Apache Spark

Unified open-source engine for fast, scalable data and machine learning analytics

- **Distributed SQL engine:** Fast, large-scale data processing with in-memory capabilities with
- **Multiple Components:** Spark ML, streaming, SQL DataFrames, and GraphX
- **Multi-language support:** Java, Scala, Python, and R.
- **Versatile workloads:** Batch, real-time, ML, and graph processing.
- **Ecosystem integration:** Connects with popular data science and BI tools.
- **Open-source community:** Driven by global collaboration and innovation.



Spark's Structured Data APIs

RDD
(2011 Release 1.0)

Resilient Distributed Dataset

- Fundamental Data Structure in Spark
- Low Level API
  - Offers Control & Flexibility
  - Code the "how-to"
- Compile-time Syntax & Analysis error detection
- No Schema
- Java, Scala, Python, R

DataFrame
(2013 Release 1.3)

- High Level API
  - Code the "what-to-do"
  - Ease-of use & Readability
- Schema: Structured Data
- Logical Plans and Optimizer
- SQL Support
- Compile-time Syntax error detection
- Run-time Analysis Error Detected
- Java, Scala, Python, R

- *Python Conversion e.g.*
  - → `df = rdd.toDF()`
  - ← `rdd = df.rdd`

Dataset
(2015 Release 1.6)

Best of RDD + Best of DataFrame

- High Level API
- Compile-time Syntax & Analysis error detection
- Schema: Structured Data
- Logical Plans and Optimizer
- More memory-efficient
- Java, Scala
- *Slower than*

Unifying DataFrames and Datasets
(2016 Release 2.0)

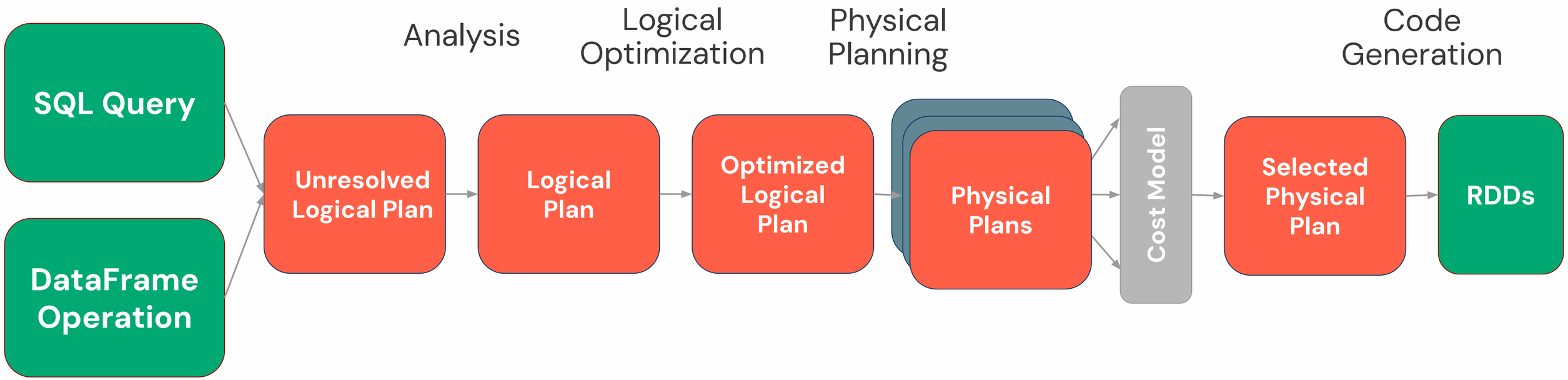
Dataset and DataFrame **merge into one unit** to reduce the complexity

Dataset API takes on two forms

Strongly-Typed API *(Java and Scala)*

Untyped API *(Python and R)*

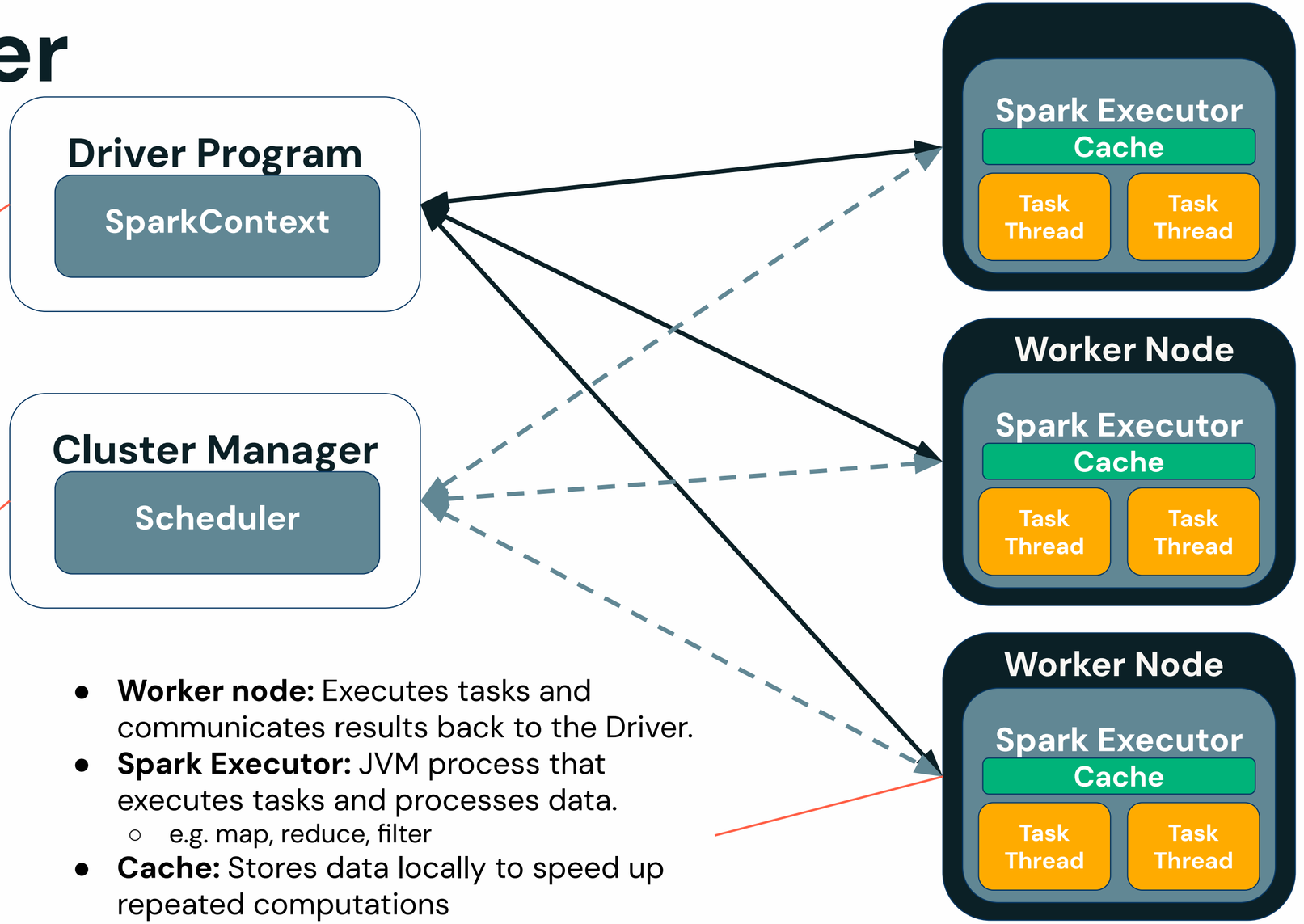
Under the Catalyst Optimizer's Hood



Spark Cluster

- Entry point for your Spark application.
- Submits jobs to the cluster

- Allocates resources across the Worker
- Launches Executors on these nodes
- Handles the distribution of tasks



- **Worker node:** Executes tasks and communicates results back to the Driver.
- **Spark Executor:** JVM process that executes tasks and processes data.
  - e.g. map, reduce, filter
- **Cache:** Stores data locally to speed up repeated computations
- **Task Thread:** Smaller unit of the overall assigned job, one thread per CPU core

When is Spark Recommended For ML?

Scaling Out or Speeding Up

Large-Scale Data Processing

- Excels at processing massive datasets needing distributed computation.
- Benefit: Handles data that can't fit on a single machine, *reduces or removes the need to sample down* your dataset.

Distributed Model Training

- Distributes model training and hyperparameter tuning across multiple nodes.
- Benefit: Significantly reduces training time using cluster computing.

Real-Time Data Processing and Streaming Analytics

- Integrates with Spark Streaming to apply ML models to real-time data streams.
- Benefit: Enables real-time predictions and analytics. e.g. fraud detection and monitoring.

End-to-End Machine Learning Pipelines

- Supports the full machine learning lifecycle.
- Benefit: Simplifies workflows with its Pipeline API, automating complex processes.

When is Spark Not Recommended for ML?

Small to Moderate-Sized Datasets

- Spark adds unnecessary complexity for small datasets that fit on a single machine.
- Alternative: Use scikit-learn or TensorFlow on one machine

Low-Latency, Real-Time Predictions

- Spark isn't optimized for microsecond or millisecond response times in real-time applications.
- Alternative: Use Databricks Model Serving or similar alternative.

Short-Lived Model Training or Batch Scoring

- The time to initialize a Spark session and distribute tasks can outweigh the benefits.
- Alternative: Use scikit-learn, XGBoost, or Docker for lightweight, focused execution.

Strict Data Privacy/Compliance

- Distributed processing complicates compliance with data privacy regulations.
- Alternative: Use on-premises tools or controlled environments for better data management.

Highly Custom Algorithms

- Offers wide variety of pre-built algorithms but, custom algorithm development in Spark can be complex and inefficient.
- Alternative: Use Python, scikit-learn, or R for more flexibility.

databricks

Machine Learning Development with Spark

LECTURE

Introduction to Spark ML for Model Development

Databricks Academy

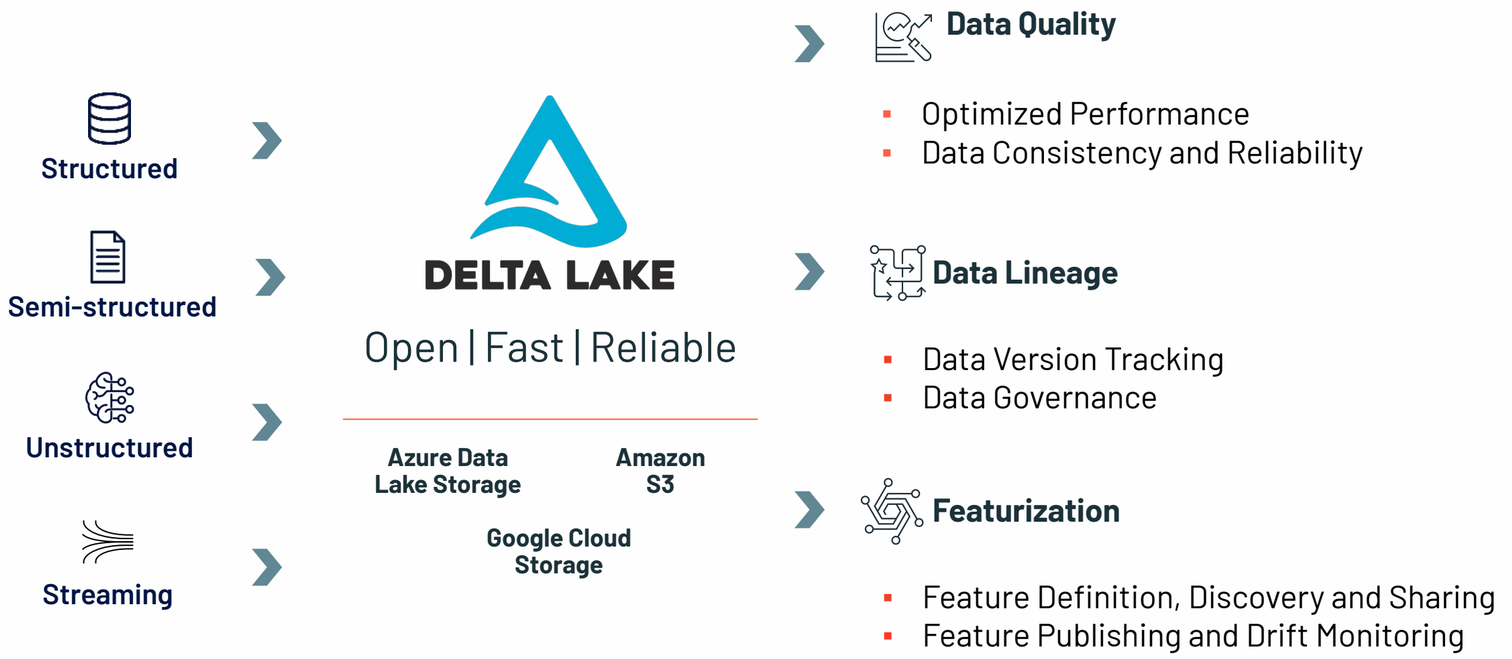
Reading, Writing, and Governance with Spark on Databricks

What is Delta Lake?

A **unified data management layer** that brings data reliability and fast analytics to cloud data lakes. It is the optimized storage layer that provides the foundation for storing data and tables in the Databricks DI Platform.

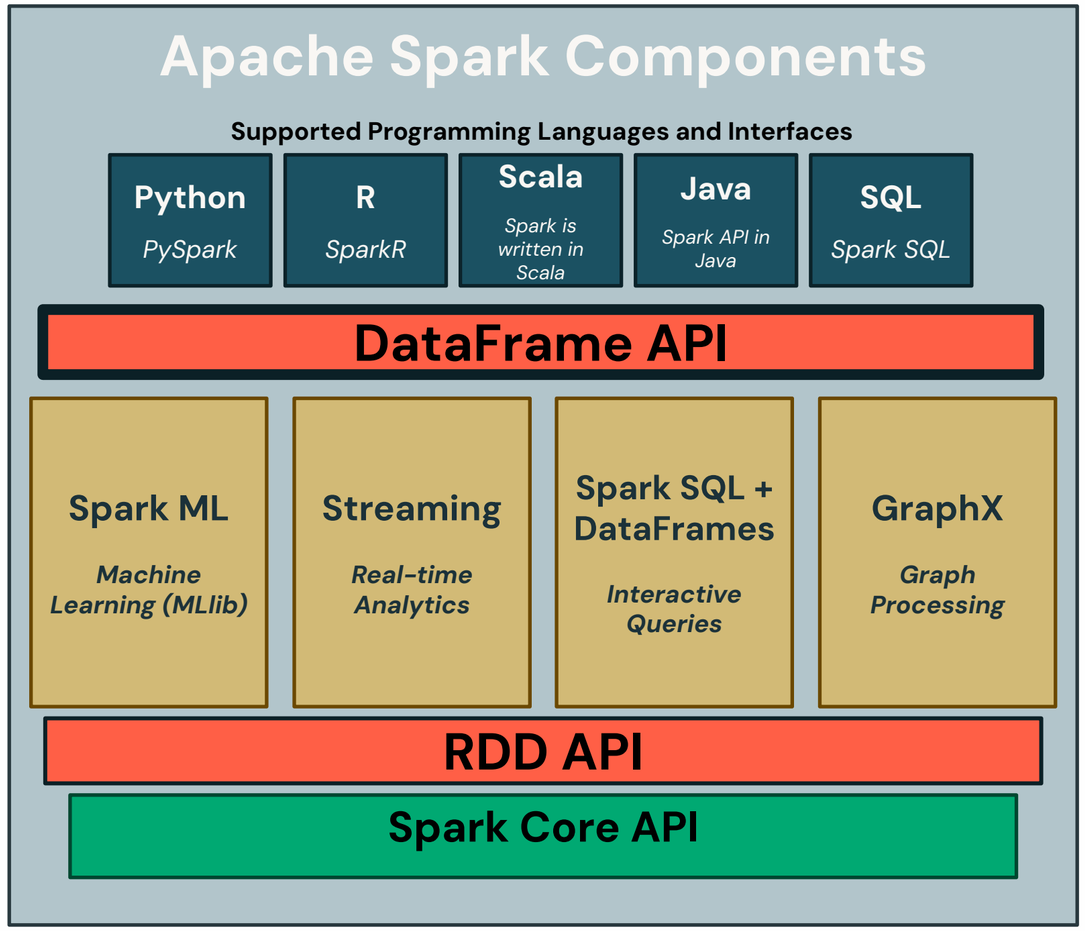
Key Features

- Unified **batch** and **streaming**
- **Schema Validation** with automatic checks
- **Upserts** with merge support.
- **Schema evolution** without data rewrite.
- **Change Data Feed**: Track row-level changes.
- **Time-travel**: Query historical versions.
- **Optimized Performance**: Data skipping & clustering.
- **Multi-language support**: Python, Scala, SQL.



Spark DataFrames

Use Spark DataFrames for scalable data manipulation.



- **High Level API** for **data manipulation** & **UDFs**
- Well-defined **schema** with names & data types.
- Query using **SQL syntax.**
- Optimized Data Processing
  - **Spark Catalyst optimizer** & **Tungsten engine**
- **Lazy Evaluation**
- **Distributed** across cluster for parallel processing
- Versatile with **Data Types** and **Sources**

Loading Data

Perform select, filter, join, groupBy, and aggregate operations efficiently.

```python
%python
# Read a Delta table using spark.read.table
df = spark.read.format("delta").table("your_table_name")

# Display the DataFrame
display(df)
```

Read a Delta Table into a Spark DataFrame

Perform data manipulations using the Spark DataFrame API.

Write to a Delta table from a Spark DataFrame.

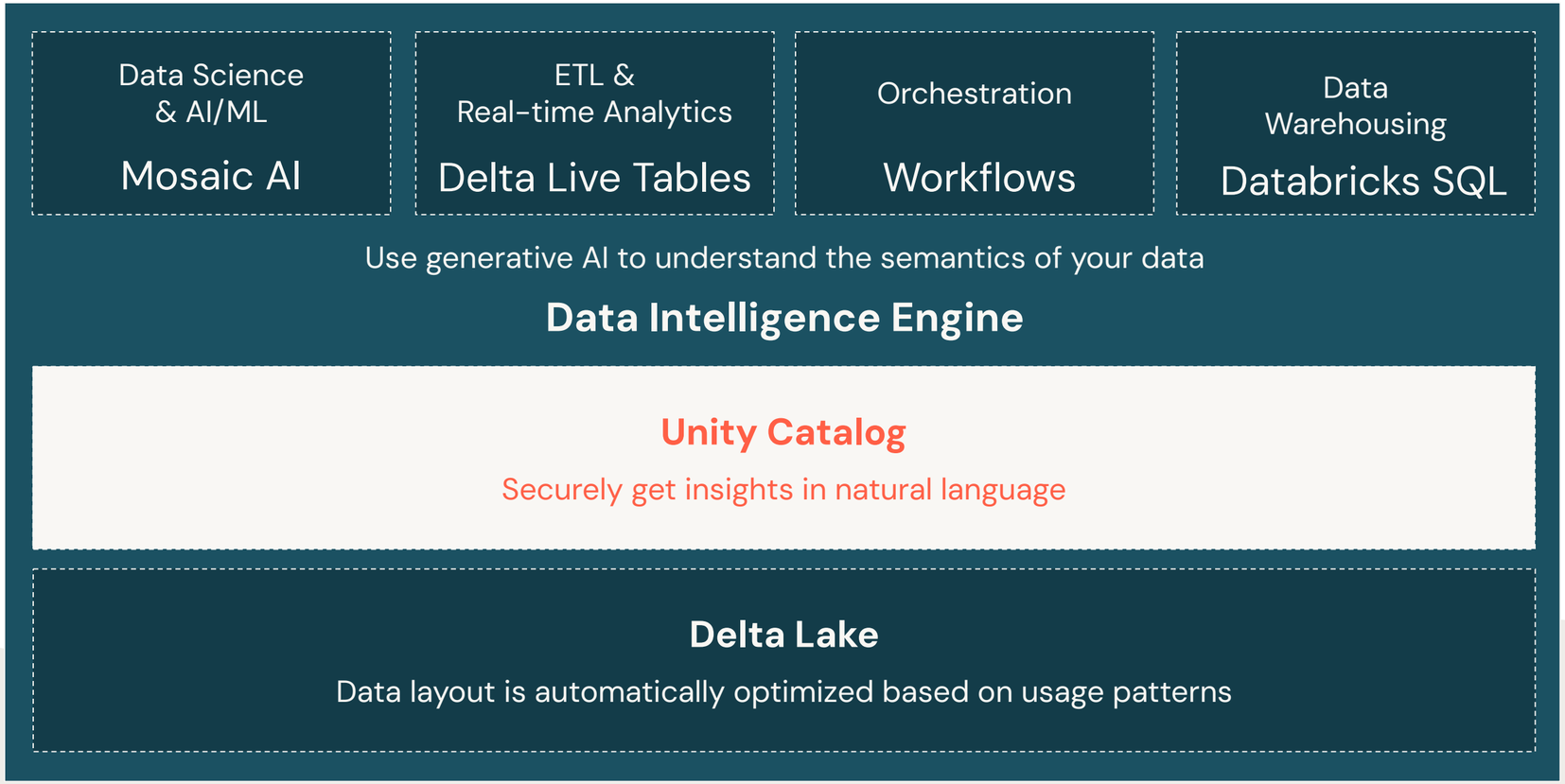

```python
# Write DataFrame to a Delta Table
df.write.mode("append").saveAsTable("new_table_name")

df.write.mode("overwrite").saveAsTable("new_table_name")

# Write DataFrame to a Feature Store Table
fe = FeatureEngineeringClient()
fe.create_table(
   name="feature_table_name",
    primary_keys="column_1",
    df = df_ml_prepared)
```

```python
# Select features and label
df_ml = df.select("feature1", "feature2", "label")

# Example: Filter out rows with missing values
df_ml_filtered = df_ml.na.drop()

# Example: Create a new feature (e.g., interaction term)
df_ml_prepared = df_ml_filtered.withColumn("feature_interaction", col("feature1") * col("feature2"))
```

Data Science & AI on Databricks

Unity Catalog

Plays a crucial role in managing, governing, and securing Delta Lake, enhancing Data Science, Data Processing, Workflows, and Databricks SQL capabilities.

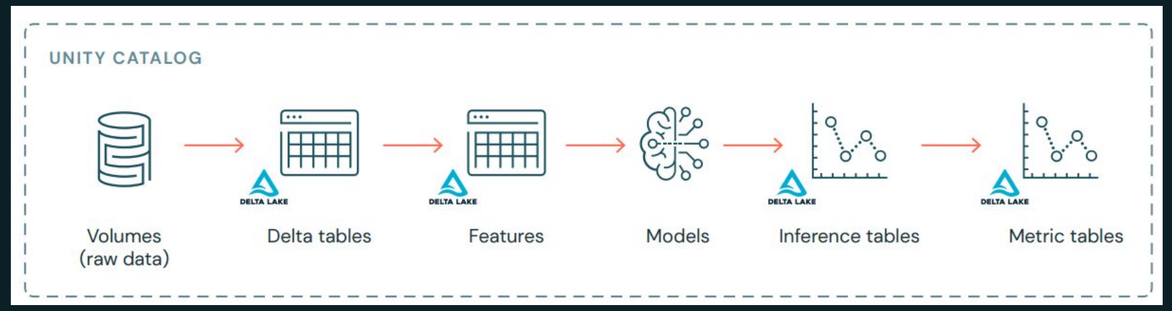


Open Data Lake

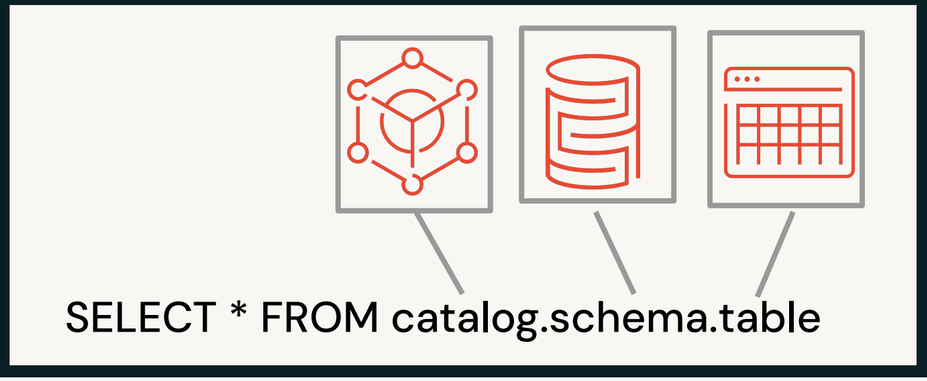
All Raw Data
(Logs, Texts, Audio, Video, Images)

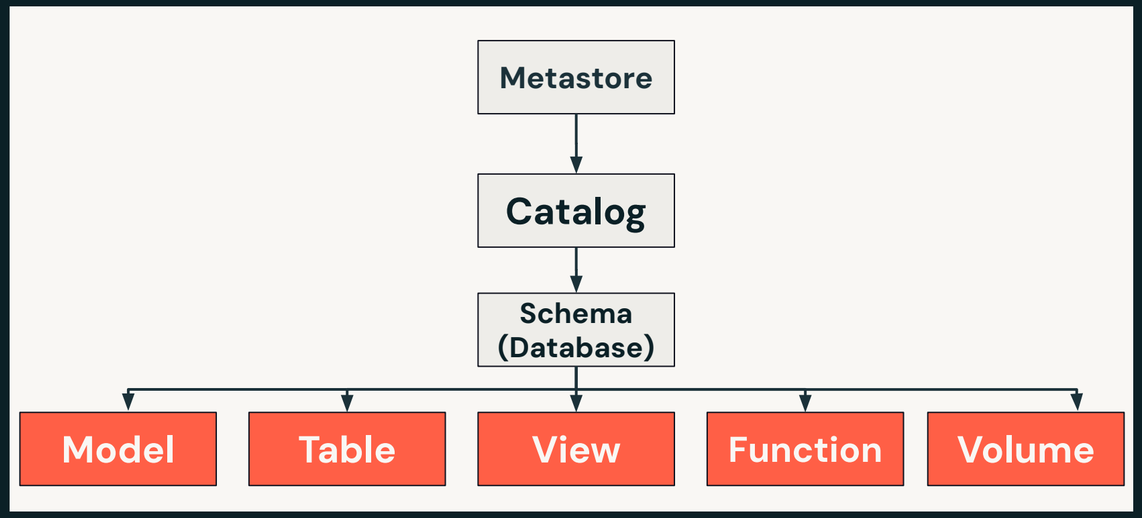
Unity Catalog

Key Capabilities



- Centralized governance, security, metadata and user management
- Data access auditing & Data lineage
- Data search and discovery
- Easy sharing of assets across Databricks workspaces
- Secure data sharing with Delta Sharing
- Feature Store
- A hosted version of the MLflow Model Registry

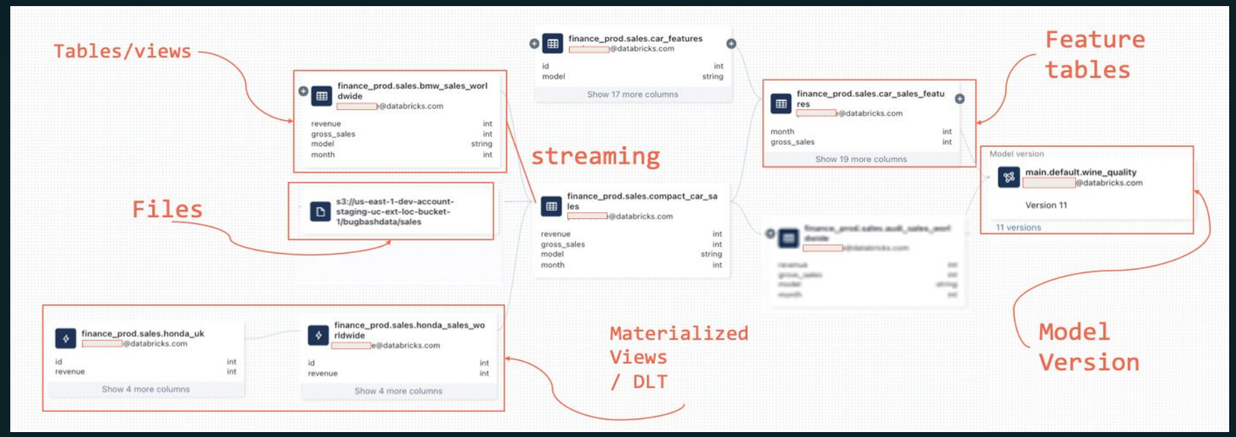


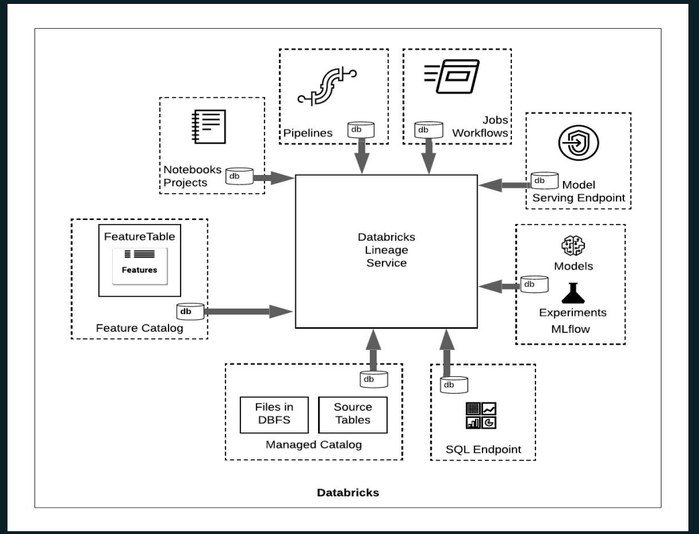


Automated lineage for all workloads

End-to-end visibility into how data flows and consumed in your organization

- Table and column lineage
- Notebook, workflow and dashboard table centric lineage
- Streaming lineage
- File, Path Lineage
- Feature Store Lineage
- DBSQL Query Lineage
- DLT Lineage
- Volume Lineage
- Mlflow model Lineage
- Lakeview Lineage

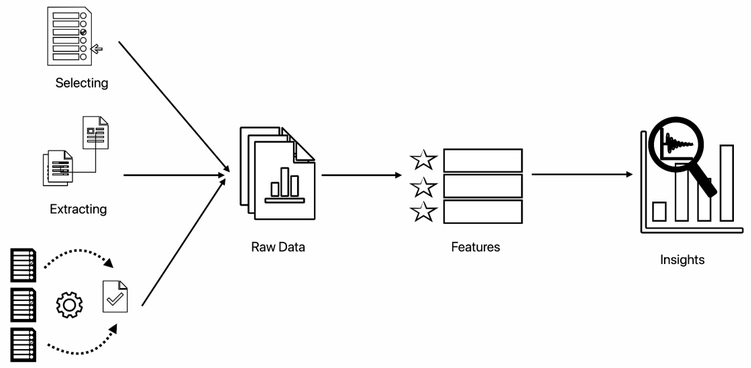




Feature Engineering

Transforming raw data into model-friendly features

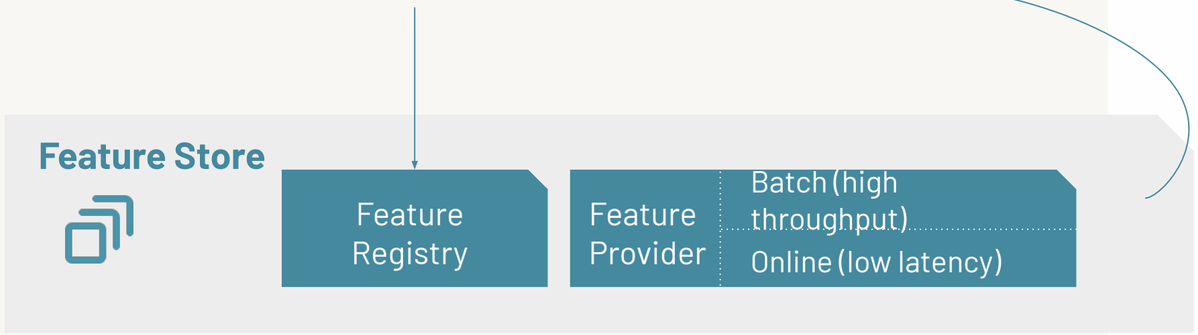
- It is a **critical step** in **preparing data** for machine learning models.
- It transforms raw data into a format that allows ML models **to extract meaningful patterns** and make accurate predictions.



Unity Catalog Feature Store

Features are organized as feature tables.

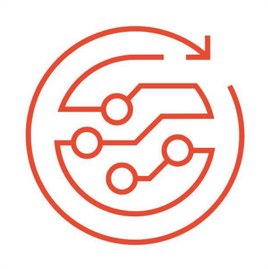
Features are stored in and provided from Unity Catalog as a Delta Table with the required primary key and additional metadata.



Overview of Spark ML

Machine Learning in Spark

Spark Machine Learning Libraries



Sparks built-in capabilities for distributed machine learning.

MLlib

Original ML API for Spark

RDD-based API

- `pyspark.mllib`
- `org.apache.spark.mllib`

Maintenance mode

Avoid use

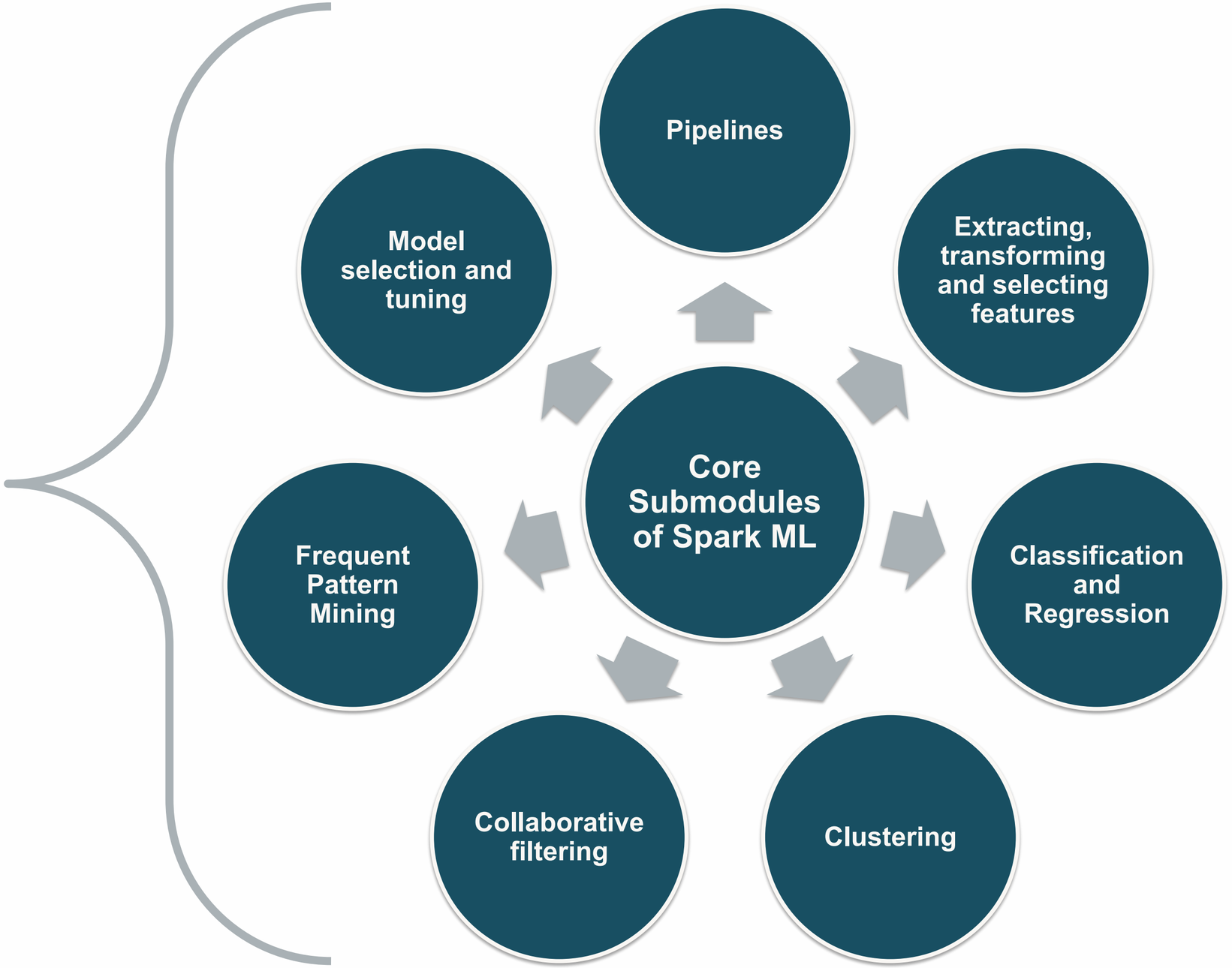
Spark ML

"Spark ML" is not an official name but refers to the **MLlib DataFrame-based API**

- `pyspark.ml`
- `org.apache.spark.ml`

Recommended Use

Spark ML



ML Pipelines

spark.ml import Pipeline Latest Documentation

Purpose: Streamlines the process of **building machine learning workflows** by chaining multiple stages, such as data preprocessing, feature extraction, and model training, into a single pipeline.

Pipeline

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Configure an ML pipeline, which consists of three stages: tokenizer,
hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

Extracting, Transforming and Selecting Features

spark.ml.feature Latest Documentation

Purpose: This submodule is dedicated to **transformation**, **feature extraction**, **dimensionality reduction**, **selection**, and **locality sensitive hashing**.

- Examples:
  - **StringIndexer:** Converts categorical labels into numerical ones
  - **Word2Vec:** An Estimator which takes sequences of words trains a Word2VecModel which maps each word to a unique fixed-size vector.
  - **PCA (Principal Component Analysis):** transforms high-dimensional data into a lower-dimensional form while preserving as much variance as possible
  - **ChiSqSelector**: Select the most relevant features from a dataset based on the chi-square (χ^2) statistical test measuring how strongly each feature is associated with the target variable.
  - **Approximate Nearest Neighbor Search**: where it hashes data points so that similar points are more likely to end up in the same bucket

Classification and regression

spark.ml.classification **Latest Documentation** **spark.ml.regression**

Purpose: Provide scalable, distributed algorithms for **classifying** data and predicting **continuous** outcomes.

- Examples:
  - **LogisticRegression (Classification):** A widely-used algorithm for binary classification, predicting categorical outcomes by modeling the probability that a given input belongs to a certain class.
  - **DecisionTreeClassifier (Classification):** A tree-based method that recursively splits the data into subsets based on feature values
  - **RandomForestRegressor (Regression):** An ensemble learning method that aggregates the predictions of multiple decision trees.
  - **LinearRegression (Regression):** A fundamental algorithm that models the relationship between a dependent variable and one or more independent variables using a linear equation.

Clustering

spark.ml.clustering Latest Documentation

Purpose: is used to **group similar data points together** without requiring labeled data. It helps in discovering natural groupings or patterns within datasets

- Examples:
  - **KMeans:** Partitions data into K distinct clusters based on feature similarity, widely used in applications like customer segmentation and pattern recognition.
  - **BisectingKMeans:** A hierarchical clustering algorithm that splits clusters to build a tree of clusters, offering a more interpretable approach.

KMeans

```
from pyspark.ml.clustering import KMeans

# Trains a k-means model.
kmeans = KMeans().setK(2).setSeed(1)
model = kmeans.fit(dataset)

# Shows the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

```
Cluster Centers:
[5.2, 3.0, 1.2, 0.2]
[6.7, 3.3, 5.7, 2.1]
```

Collaborative Filtering

spark.ml.recommendation Latest Documentation

Purpose: A technique used in recommendation systems to predict a user's preferences by learning from past interactions of similar users or items.

- spark.ml uses the **alternating least squares (ALS) algorithm** to learn these latent factors with parameters such as:
  - ***maxIter*** is the maximum number of iterations to run (defaults to 10).
  - **regParam** is the regularization parameter balancing between fitting the training data well and keeping the model simple. Higher value means simpler model. (defaults to 1.0).

```
from pyspark.ml.recommendation import ALS

# Build the recommendation model using ALS on the training data
als = ALS(maxIter=5, regParam=0.01, userCol="userId",
itemCol="movieId", ratingCol="rating")
model = als.fit(training)

# Generate top 10 movie recommendations for each user
userRecs = model.recommendForAllUsers(10)

# Generate top 10 user recommendations for a specified set of
movies
movies = ratings.select(als.getItemCol()).distinct().limit(3)
 movieSubSetRecs = model.recommendForItemSubset(movies, 10)
```

Frequent Pattern Mining

spark.ml.fpm [Latest Documentation]

Purpose: Used to identify recurring patterns, associations, or correlations in large datasets.

- **FP-Growth:** Compresses the input data into a prefix-tree (FP-tree) and then recursively mines the frequent patterns.
  - ***Scenario:*** Retail: to find frequent combinations of items that customers often purchase together
- **PrefixSpan:** A sequential pattern mining algorithm that identifies frequent sequences of items in transactional data.
  - ***Scenario:*** Analyzing customer purchasing patterns over time, where the order of items is important.

```
from pyspark.ml.fpm import FPGrowth

df = spark.createDataFrame([
    (0, [1, 2, 5]), (1, [1, 2, 3, 5]), (2, [1, 2])], ["id", "items"])

fpGrowth = FPGrowth(itemsCol="items", minSupport=0.5, minConfidence=0.6)
model = fpGrowth.fit(df)

# Display frequent itemsets.
model.freqItemsets.show()

# transform examines the input items against all the association rules and
summarize the consequents as prediction
 model.transform(df).show()
```

```
+---------+----+
|    items|freq|
+---------+----+
|      [1] |   3|
|      [2] |   3|
|      [5] |   2|
|   [1, 2] |   3|
|   [1, 5] |   2|
|   [2, 5] |   2|
|[1, 2, 5] |   2|
+---------+----+
```

```
+---+---------------+----------------+
| id|          items|    prediction  |
+---+---------------+----------------+
|  0|      [1, 2, 5]|              []|
|  1|   [1, 2, 3, 5]|              []|
|  2|          [1, 2]|             [5]|
+---+---------------+----------------+
```

prediction: The items that the model predicts could be added to the transaction based on the association rules.

ML Tuning: Model Selection and Hyperparameter Tuning

spark.ml.evaluation **Latest Documentation** **spark.ml.tuning**

Purpose: Facilitating **model selection** and hyperparameter optimization through techniques like **Cross-Validation** and **Train-Validation Split** to achieve the highest predictive performance.

Examples:

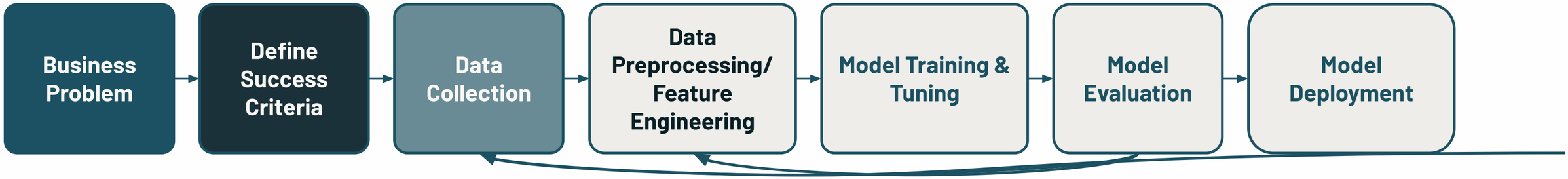
- **BinaryClassificationEvaluator:** An evaluator used specifically for binary classification problems, which calculates performance metrics such as area under the ROC curve (AUC) or area under the precision-recall curve (AUPRC).
- **CrossValidator:** A tool that performs k-fold cross-validation to evaluate the performance of different models or hyperparameter settings.
- **TrainValidationSplit:** An alternative to CrossValidator that splits the dataset into a training set and a validation set for hyperparameter tuning, offering a faster but less robust method of model selection.

Data Preparation & Feature Engineering with Spark

Model Development Scenario

- **6.3 million** animals **enter** U.S. shelters annually.
- **920,000** shelter animals are **euthanized** each year.
- **4.1 million** shelter animals are **adopted** annually.
  ASPCA

PawMatch: An animal shelter dedicated to finding forever homes for pets.



Business Problem: PawMatch aims to increase adoption rates by predicting which animals are at risk of longer shelter stays to allow for targeted interventions.

Example Success Criteria:

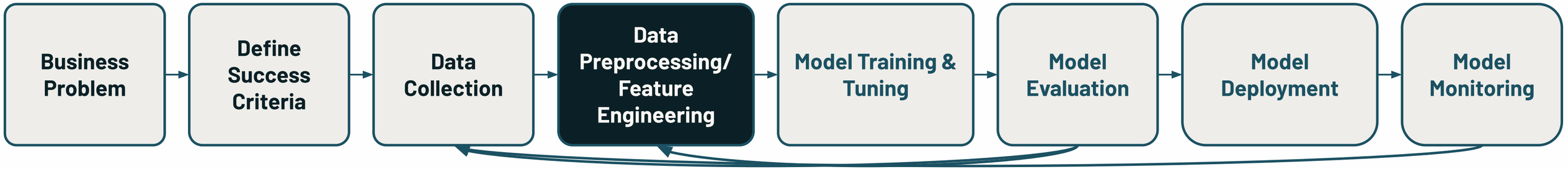
- **Increased Adoption Rates:** Increase in the number of animals adopted within a shorter timeframe
- **Shorter Shelter Stays:** Fewer average days in the shelter before adoption.

Example Data Collection:

- **Animal Demographics:** Age, breed, size, color, gender.
- **Behavioral Assessments:** Temperament evaluations.
- **Shelter History:** Length of stay, adoption attempts.
- **Health Information:** Medical history, spay/neuter status.

Data Preprocessing & Feature Engineering

PawMatch: An animal shelter dedicated to finding forever homes for pets.



- **Objective:** Identify the most effective data preprocessing and model training techniques for predicting animal adoption likelihood.
- **Action:** Select Transformers**:** Choose appropriate transformers for preprocessing the data, considering categorical and continuous features.

What Transformers Might be Useful?

Evaluating Preprocessing Methods

Compare and Contrast Spark ML Transformers and Estimators

StringIndexer

Converts categorical features such as breed, color, and gender into numerical indices.

```
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer(inputCol="color", outputCol="colorIndex")
indexed = indexer.fit(df).transform(df)
```

AnimalID	color	colorIndex	Age	Weight	Adopted
1	Black	0.0	3	30	1
2	Brown	1.0	5	10	0
3	White	2.0	2	20	1

Imputer

Handles missing values by imputing them with mean, median, or mode values..

```
from pyspark.ml.feature import Imputer
# Apply Imputer to fill missing values in the "Age" column
imputer = Imputer(inputCols=["Age"],
outputCols=["imputedAge"]).setStrategy("mean")
imputed_df = imputer.fit(df).transform(df)
```

AnimalID	Age	imputedAge
1	3	3.0
2	None	2.5
3	2	2.0

StandardScaler

Normalizes continuous features like age, weight, and length of stay to have a mean of 0 and standard deviation of 1.

```
from pyspark.ml.feature import StandardScaler
scaler = StandardScaler(inputCol="age", outputCol="scaledAge",
withMean=True, withStd=True)
scaled = scaler.fit(df).transform(df)
```

AnimalID	Age	scaledAge
1	3	-0.5
2	5	1.0
3	2	-1.5

OneHotEncoder

Use OneHotEncoder for categorical features to create binary vectors, avoiding ordinality.

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder

# StringIndexer and OneHotEncoder
indexer = StringIndexer(inputCol="AnimalType",
    outputCol="AnimalTypeIndex")
encoder = OneHotEncoder(inputCol="AnimalTypeIndex",
    outputCol="AnimalTypeVec")

# Apply transformations
df_indexed = indexer.fit(df).transform(df)
df_encoded = encoder.fit(df_indexed)
    .transform(df_indexed)
```

AnimalID	AnimalType	AnimalTypeVec
1	Dog	(1,[],[])
2	Cat	(1,[0],[1.0])
3	Dog	(1,[],[])

VectorAssembler

Assembling Features into a Vector for Machine Learning Models

- **Objective:** Combine multiple features into a single vector column
- **Action:** Use VectorAssembler to create a feature vector by combining individual features (e.g., age, weight, breed) into a single vector column.
- **Reasoning:**
  - Most ***Spark ML algorithms require input features to be in vectorized form*** for efficient processing and scalability.
  - VectorAssembler allows features to be **efficiently represented**, enabling Spark to handle large datasets with optimized memory usage.
  - VectorAssembler chooses dense vs. sparse output format based on whichever one uses less memory

```
from pyspark.ml.feature import VectorAssembler

# Assuming df is your DataFrame with separate feature columns
assembler = VectorAssembler(inputCols=["Age", "Weight",
   "BreedIndex"], outputCol="features")
df_assembled = assembler.transform(df)

# Select and show results
df_assembled.select("AnimalID", "Age", "Weight",
   "BreedIndex", "features").show()
```

```
+---------+---+------+----------+-----------------+
| AnimalID|Age|Weight|BreedIndex|features         |
+---------+---+------+----------+-----------------+
|        1|  3|    30|       0.0| [3.0, 30.0, 0.0]|
|        2|  5|    10|       1.0| [5.0, 10.0, 1.0]|
|        3|  2|    20|       0.0| [2.0, 20.0, 0.0]|
+---------+---+------+----------+-----------------+
```

Reproducible Train-Test Split

Ensuring Consistency in Model Evaluation

- **Objective:**
  - To divide the dataset into training and testing sets in a way that ensures reproducibility, allowing consistent and reliable model evaluation.
- **Action:**
  - Use Spark ML's **randomSplit** function with a **fixed seed** to create reproducible train-test splits.
- **Reasoning:**
  - The randomSplit with a fixed seed ensures that every time the data is split, the training and testing datasets remain the same, which is crucial for comparing model performance over time.

```
# Perform train-validation-test split with a fixed seed
train_df, validation_df, test_df = df.randomSplit([0.6, 0.2, 0.2], seed=42)

# Show train, validation, and test datasets
print("Training Data:")
train_df.show()

print("Validation Data:")
validation_df.show()

print("Testing Data:")
test_df.show()
```

```
Training Data:
+---------+---+------+----------+
| AnimalID|Age|Weight|BreedIndex|
+---------+---+------+----------+
|        1|  3|    30|       0.0|
|        4|  4|    25|       1.0|
|        5|  6|    15|       0.0|
+---------+---+------+----------+
```

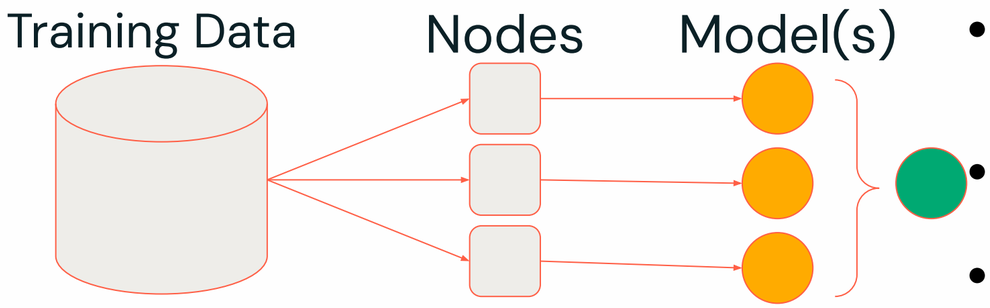
```
Validation Data:
+---------+---+------+----------+
| AnimalID|Age|Weight|BreedIndex|
+---------+---+------+----------+
|        3|  2|    20|       0.0|
+---------+---+------+----------+
```

```
Test Data:
+---------+---+------+----------+
| AnimalID|Age|Weight|BreedIndex|
+---------+---+------+----------+
|        2|  5|    10|       1.0|
+---------+---+------+----------+
```

ML Training and Tuning on Spark

Distributed ML Library

Data Parallel Training

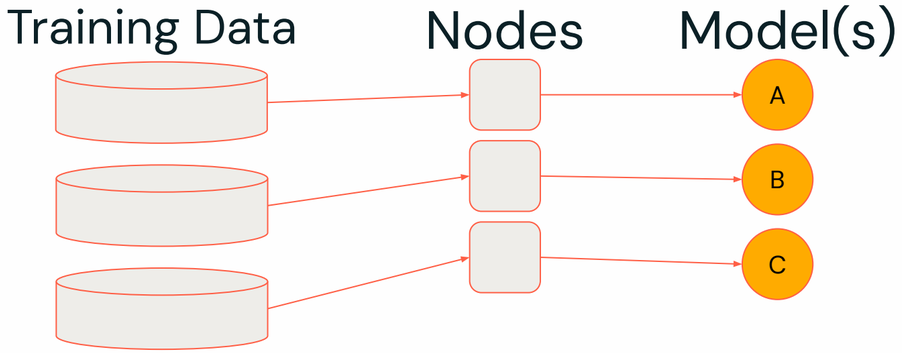


- Train one model across a distributed dataset.
  - Requires distributed ML library (e.g., Spark ML).
  - Produces a single, scalable model.
- Effective for large datasets; no downsampling needed.
- Limitation: Not all model algorithms scale this way efficiently. E.g. K-Nearest-Neighbors where the model is the entire dataset

Single Node ML Library

Parallel Model Training

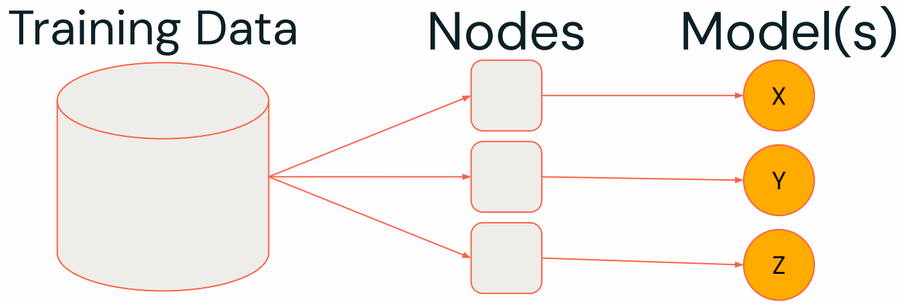
Discussed Later



- Training many unique single node models in parallel
  - Dataset must fit on a single worker node.
- Scales out model training and not restricted to a sequential training process.
- Compatible with any ML library.
- Ideal for scenarios like IoT devices training individual models.

Combination

Parallel Hyperparameter Optimization

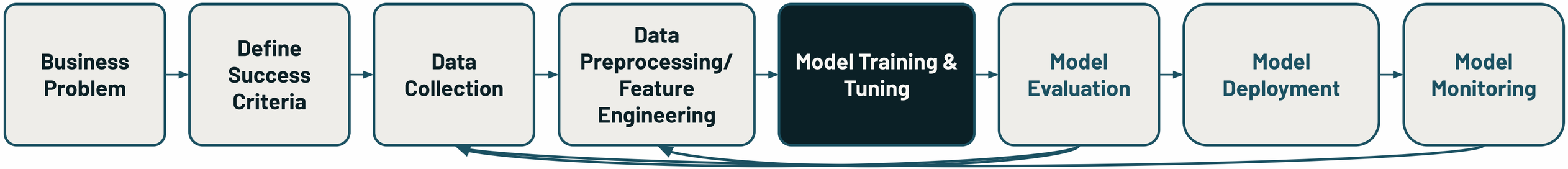


- Parallel fit for different hyperparameter configs
  - Applicable to single node and distributed training.
  - Reduces time for hyperparameter tuning.
  - Specifically powerful with large search space.
- On Databricks: Use Optuna with Joblib Apache Spark Backend or Spark ML CV
- Works with any ML Library

Model Training and Tuning with Spark

Model Training and Tuning

PawMatch: An animal shelter dedicated to finding forever homes for pets.



- **Objective:** Train a machine learning model on the training dataset using Spark ML to predict outcomes, such as animal adoption likelihood.
- **Action:** Select a suitable model algorithm or algorithms to fit a model on the training dataset.

What Model Algorithm(s) Might be Useful?

Model Training Scenario: LogisticRegression

Decision: Binary Classification to Long/Short Adoption Timeframe

- **LogisticRegression**: Model for binary classification tasks. It predicts the probability of the outcomes, providing insights into the likelihood of each class (e.g., long vs. short adoption timeframe).

```
from pyspark.sql import SparkSession
from pyspark.ml.classification import LogisticRegression

# Split the data into train and test sets
train_df, test_df = df_assembled.randomSplit([0.8, 0.2], seed=42)

# Initialize and fit Logistic Regression with specific hyperparameters
lr = LogisticRegression(featuresCol="features", labelCol="Adopted",
   maxIter=10, regParam=0.01, threshold=0.3)
lr_model = lr.fit(train_df)

# Explain the parameters of the model
print(lr_model.explainParams())

# Make predictions on the test data
predictions = lr_model.transform(test_df) predictions.select("AnimalID",
   "prediction", "probability").show()
```

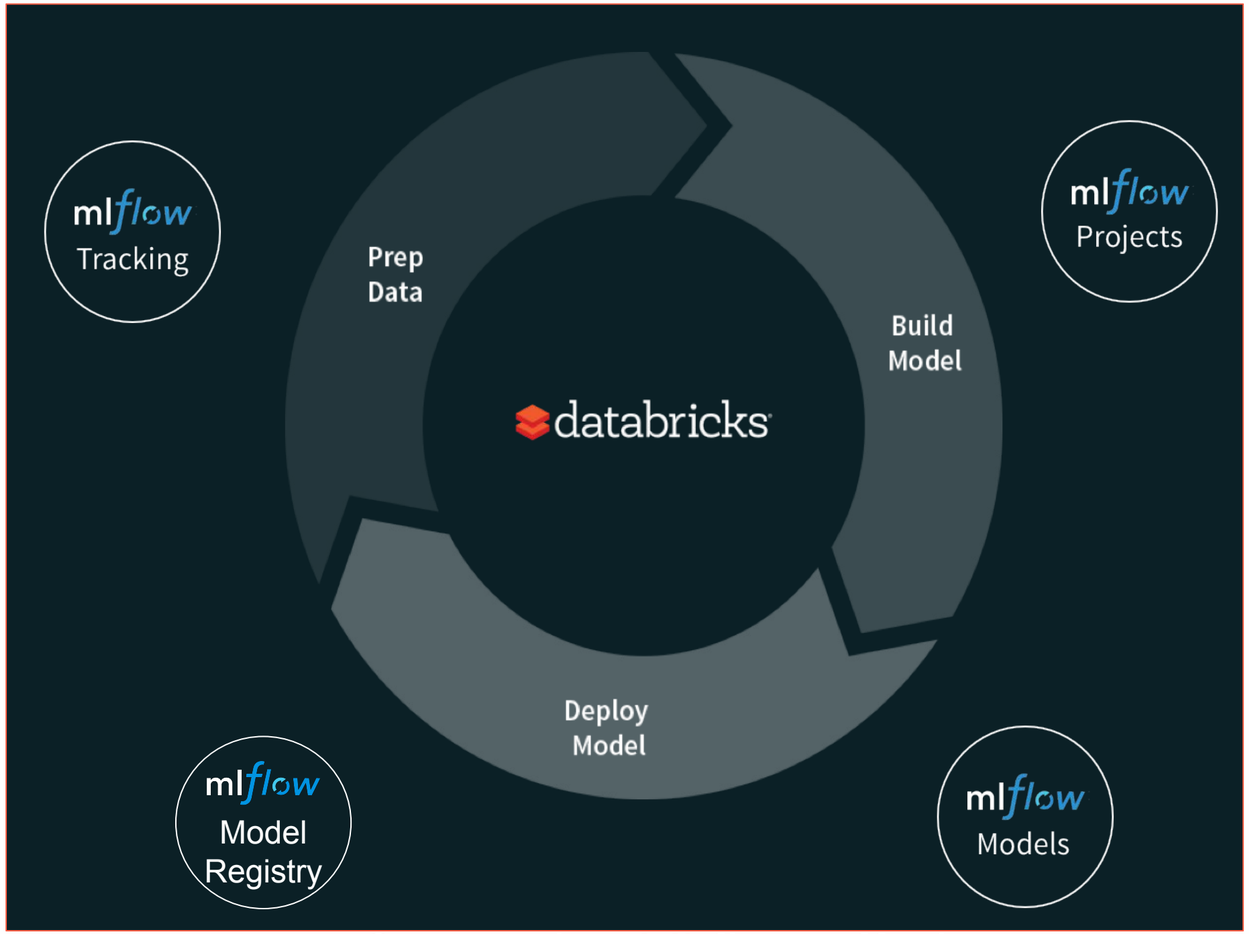
- **Method Examples:**
  - **fit:** Trains the Logistic Regression model on the input dataset.
  - **explainParams:** Returns the documentation of all params with their optionally default values and user-supplied values.
- **Hyperparameter Examples:**
  - **maxIter:** The maximum number of iterations (default: 100).
  - **threshold:** The threshold in binary classification to convert probability predictions into predicted labels (default: 0.5).

```
featuresCol: features (default: features)
labelCol: Adopted (default: label)
maxIter: max number of iterations (default: 10)
regParam: regularization parameter (default: 0.01)
...
+---------+----------+----------------------+
| AnimalID|prediction|probability           |
+---------+----------+----------------------+
|        2|       0.0|[0.8, 0.2]            |
|        4|       0.0|[0.7, 0.3]            |
+---------+----------+----------------------+
```

Databricks provides Managed MLflow

MLflow is an open-source platform designed to help manage the machine learning full lifecycle.

- **Key Advantages**:
  - **Maintenance-Free**: Eliminates overhead with a fully managed service.
  - **Pre-installed:** It comes pre-installed on Databricks Runtime for ML
  - **Built-in tracking server** logs all experiments and models, ensuring efficient management without the need for additional setup.
  - **Feature Store Integration**: Centralizes feature management, consistency across training and inference.
  - **Enhanced Security**: Includes model-level access control and secure transactions.
  - **Streamlined Operations**: Supports automated tracking of machine learning steps, making it easier to analyze and optimize processes.



Comparing MLflow offerings

MLflow Tracking

MLflow Tracking APIs provide a set of functions to track your runs.

- **mlflow.create_experiment()**
  - Creates a new experiment and returns the ID.
- **mlflow.set_experiment()**
  - New runs launch under this experiment if not specified in start_run().
  - If the experiment doesn't exist, it will be created and set as active.
- **mlflow.start_run()**
  - Starts a new run and returns an ActiveRun object.
  - Ends current run before starting a new one or use nested=True.
  - If run_name not set unique run name generated
- **mlflow.autolog() or Custom Logging**
  - With one line of code autolog() logs parameters, metrics, data lineage, model, and environment
  - Alternatively, you log MLflow metrics by adding log methods in your ML code.

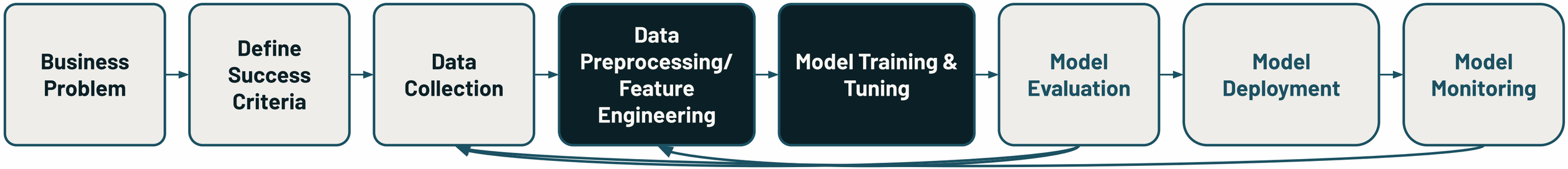
```
import mlflow

# Set an experiment name
experiment_name = "Wine Quality Model"
mlflow.set_experiment(experiment_name)

# Start a run under the set experiment
with mlflow.start_run():
    # Your ML code
    mlflow.spark.autolog() # Enable autologging
    ...
```

```
import mlflow

# Set an experiment name, which must be unique and
case-sensitive.
experiment_name = "Wine Quality Model"
experiment_id = mlflow.create_experiment(experiment_name)

# Start a run under the created experiment
with mlflow.start_run(experiment_id=experiment_id):
    # Your ML code
    ...
    mlflow.log_metric("rmse", rmse)
    mlflow.log_param("numTrees", 100)
    ...
```

Streamlining Model Development with Spark

Streamlining Model Development

PawMatch: An animal shelter dedicated to finding forever homes for pets.



- **Objective:** Optimize the entire model development process—spanning data preprocessing, feature engineering, model training, and tuning—by making it more efficient and repeatable.
- **Action:** Construct a Pipeline using Spark ML that integrates all stages of the machine learning workflow.

What will be part of the Pipeline?

Advantages of Using Spark ML Pipelines

Streamlining Machine Learning Workflows

Consistency:

Ensures that the same data transformations and model training steps are applied uniformly, reducing the risk of errors and inconsistencies.

Reusability:

Pipelines can be saved and reused across different datasets or projects, significantly reducing the time needed for model development.

Scalability:

Designed to efficiently handle large-scale datasets, Pipelines in Spark ML are optimized for distributed processing, making them ideal for big data applications.

Simplified Workflow:

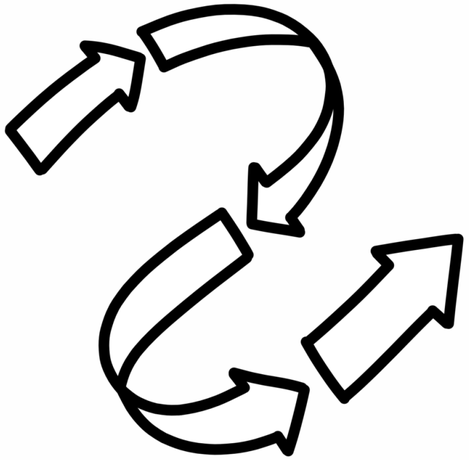
By chaining together multiple stages, Pipelines streamline the process, making complex workflows easier to manage and maintain.

Hyperparameter Tuning:

Integrates seamlessly with cross-validation and hyperparameter tuning methods, allowing for more efficient and automated model optimization.

APACHE Spark ML

Pipeline



Creating and Fitting a Pipeline in Spark ML

Ensuring Consistency in Model Evaluation

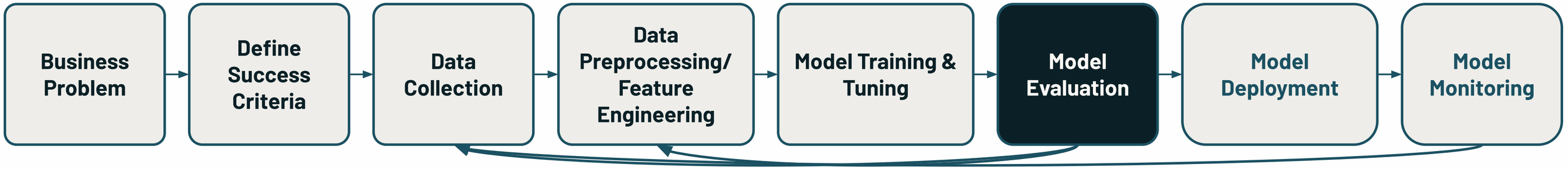
- **Objective:**
  - Build and apply a Pipeline in Spark ML for streamlined machine learning workflows.
- **Action:**
  - Define the Stages:
    - **Feature Engineering:** Use VectorAssembler to combine input features into a single vector.
    - **Scaling:** Apply StandardScaler to normalize the feature vector.
    - **Model Training:** Add a classifier like LogisticRegression as the final stage.
  - Build the Pipeline:
    - **Chain the defined stages** into a Pipeline object.
  - **Fit** the Pipeline
  - **Save and Load** the Pipeline

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StandardScaler
from pyspark.ml.classification import LogisticRegression

# Define stages
assembler = VectorAssembler(inputCols=["Age", "Weight", "BreedIndex"],
   outputCol="features")
scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures")
lr = LogisticRegression(featuresCol="scaledFeatures", labelCol="Adopted")

# Create a Pipeline
pipeline = Pipeline(stages=[assembler, scaler, lr])

# Fit the Pipeline on the training data
pipeline_model = pipeline.fit(train_df)

# Save and Load the trained Pipeline model
pipeline_model.save("/path/to/save/pipeline_model")
loaded_pipeline_model =
    PipelineModel.load("/path/to/save/pipeline_model")
```

Model Evaluation and Model Metadata with Spark

Model Evaluation

PawMatch: An animal shelter dedicated to finding forever homes for pets.



- **Objective:** To assess the performance and effectiveness of the trained machine learning model to predict the length of stay for pets, ensuring it meets the defined success criteria.
- **Action:** Use appropriate evaluation metrics (e.g., accuracy, precision, recall, AUC-ROC) and validation techniques (e.g., cross-validation) to thoroughly evaluate the model's predictive power and generalizability.

What should be used to evaluate the model?

Evaluating the Model Using Spark ML API

Selecting the Right Metrics and Evaluators for Model Assessment

Action to Evaluate:

- **Make Predictions:**
  - Apply the trained model to a test dataset to generate predictions.
- **Select an Evaluation Metric:**
  - Ex: accuracy, area under the ROC curve (AUC), precision, recall, F1 score, etc.
- **Use an Spark ML provided Evaluator:**
  - e.g. BinaryClassificationEvaluator, MulticlassClassificationEvaluator, and RegressionEvaluator

Key Points to Remember:

- **Choosing the Right Evaluation Metrics:**
  - **Area Under ROC (AUC):** Measures the model's ability to distinguish between classes, *Range: 0→1 (better)*
  - **Accuracy:** Measures the proportion of correct predictions among the total number of cases.
  - **Precision, Recall, F1 Score:** Evaluates trade-offs between different types of classification errors.
- **Choosing the Right Evaluator:**
  - **BinaryClassificationEvaluator:** Use for binary classification tasks.
  - **MulticlassClassificationEvaluator:** Use for multi-class classification problems.
  - **RegressionEvaluator:** Use for regression tasks.
- **Evaluate on a Test Dataset:**
  - Always evaluate your model on a separate test dataset that was not used during training to get an unbiased assessment of its performance.

Evaluating the Model Using Spark ML API

Assessing Model Performance with the BinaryClassificationEvaluator

- BinaryClassificationEvaluator:
  - **labelCol:**
    - The column name that contains the true labels (e.g., "Adopted").
  - **rawPredictionCol:**
    - The column that contains the raw predictions from the model (default: "rawPrediction").
  - **metricName:**
    - The evaluation metric to be used.
    - Supports::
      - **"areaUnderROC":** Area under the Receiver Operating Characteristic curve.
      - **"areaUnderPR"**: Area under the Precision-Recall curve.
  - **weightCol**
    - Adjust the influence of individual data points. If this is not set or empty, we treat all instance weights as 1.0.

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Assuming predictions have been made on the test dataset
predictions = lr_model.transform(test_df)

# Initialize the evaluator for binary classification
evaluator = BinaryClassificationEvaluator(labelCol="Adopted",
rawPredictionCol="rawPrediction", metricName="areaUnderROC")

# Evaluate the model
roc_auc = evaluator.evaluate(predictions)

# Display the evaluation result
print(f"Area Under ROC: {roc_auc:.2f}")
```

```
Area Under ROC: 0.85
```

Computing Basic Predictions Using a Spark ML Model

Interpreting Model Outputs for Decision Making

- **prediction:**
  - The predicted class label (e.g., 0 or 1) based on the model's decision threshold (default: 0.5).
- **probability:**
  - The predicted probability of each class. The array represents the probability distribution over the classes (e.g., [P(Class 0), P(Class 1)]).
- ***Interpretation:***
  - Higher probability values for a class indicate greater confidence in the prediction.
    - **[0.7, 0.3]** means **70% confident in class 0** and 30% confident in class 1.

```
# Assuming you have a trained model (e.g., LogisticRegressionModel) and test data
predictions = lr_model.transform(test_df)

# Select and display the relevant columns: AnimalID, prediction, and probability
predictions.select("AnimalID", "prediction", "probability").show()
```

```
+--------+----------+----------------------+
| AnimalID|prediction|probability           |
+--------+----------+----------------------+
|        1|       0.0|[0.7, 0.3]            |
|        2|       1.0|[0.2, 0.8]            |
|        3|       0.0|[0.65, 0.35]          |
|        4|       1.0|[0.1, 0.9]            |
+--------+----------+----------------------+
```

Reviewing a Spark ML Model's Metadata

Gaining Clarity with Model Metadata

```python
from pyspark.ml import PipelineModel

# Load the saved PipelineModel
pipeline_model = PipelineModel.load("/path/to/pipeline_model")

# Inspect the stages of the pipeline
for stage in pipeline_model.stages:
    print(stage)

# Retrieve the Model as the last stage in the pipeline
lr_model = pipeline_model.stages[-1]

# Review the model parameters
print("Model Parameters:")
print(lr_model.explainParams())

# Accessing the training summary for LogisticRegression
training_summary = lr_model.summary

# Display key training statistics
print("Training Summary:")
print("Coefficients: ", training_summary.coefficientMatrix)
print("Intercept: ", training_summary.interceptVector)
print("Accuracy: ", training_summary.accuracy)
```

- **Objective:**
  - Be able to generate reproducible and comparable experiments leveraging the model's metadata to help.
- **What is Model Metadata?**
  - Information about the model's configuration, parameters, training statistics, and the data schema used during training.
- **Why Review Metadata?**
  - Provide a detailed record of the model's training process.
    - How model was trained, parameters used, data used, etc.

```
VectorAssembler_1d87ecb7c5f5
StandardScaler_4bcb8f96d0db
LogisticRegression_9b99d3c92e79
```

```
Model Parameters:
elasticNetParam: 0.0 (default: 0.0)
featuresCol: scaledFeatures (default: features)
labelCol: Adopted (default: label)
maxIter: 10 (default: 100)
...
```

```
Training Summary:
Coefficients:  [0.44, -0.15, 0.2]
Intercept:  0.1
Accuracy:  0.85
```

databricks

Machine Learning Development with Spark

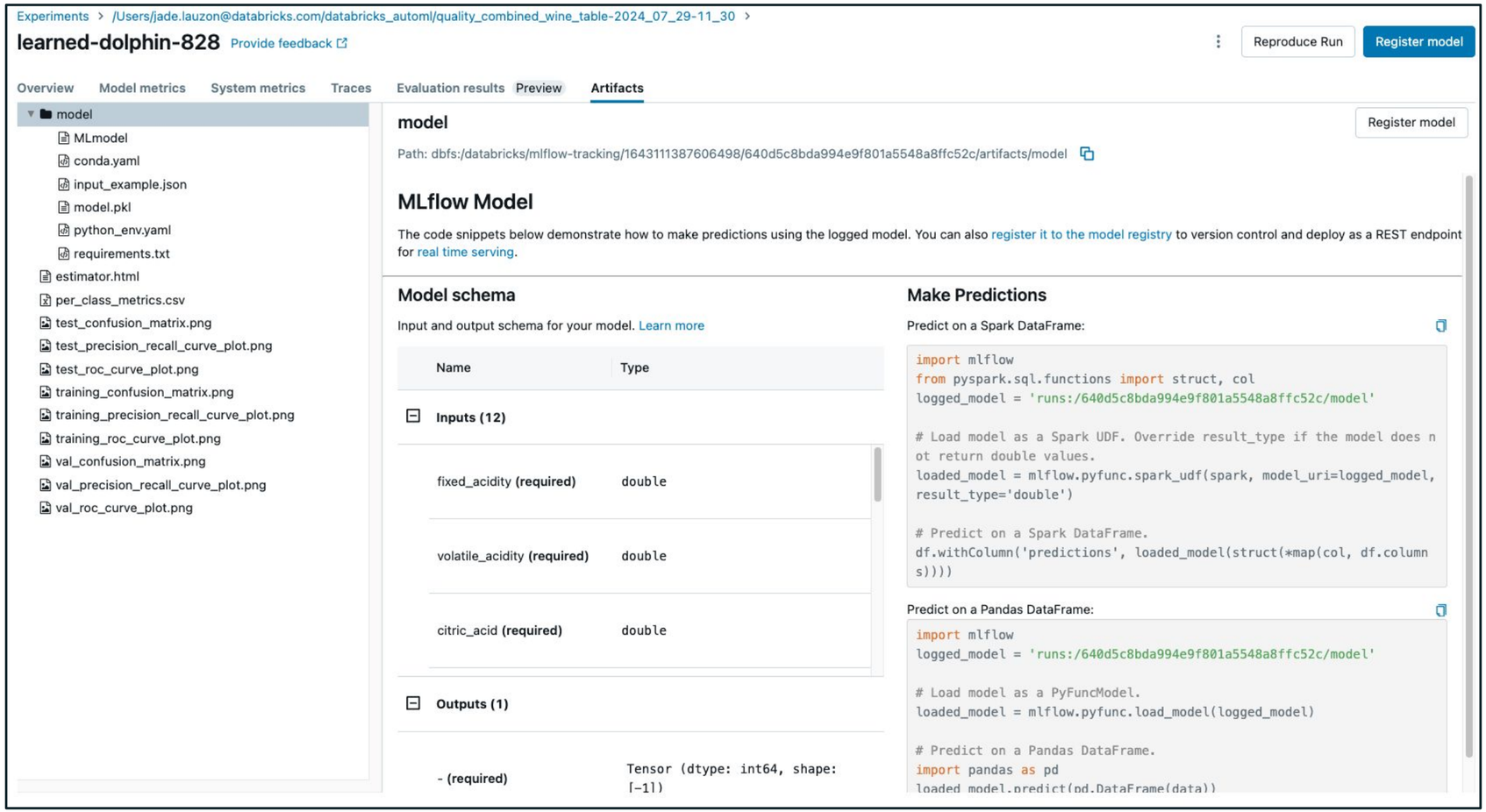
LECTURE

Model Tracking and Packaging with MLflow and Unity Catalog on Databricks

MLflow Models

MLflow Tracking Artifact includes the MLflow Model

Each MLflow Model is a directory containing arbitrary files, together with an MLmodel file in the root of the directory



Example MLflow Model

```
my_model/
├── sparkml/
├── MLmodel
├── conda.yaml
├── input_example.json
├── python_env.yaml
├── Requirements.txt
├── search_results.csv
└── estimator_info.json
    ...
```

```yaml
artifact_path: best_model
databricks_runtime: 15.2.x-cpu-ml-scala2.12
flavors:
  python_function:
    data: sparkml
    env:
      conda: conda.yaml
      virtualenv: python_env.yaml
    loader_module: mlflow.spark
    python_version: 3.11.0
  spark:
    code: null
    model_class: pyspark.ml.pipeline.PipelineModel
    model_data: sparkml
    pyspark_version: 3.5.0
mlflow_version: 2.11.3
model_uuid: 8e8d04a8730545d99c5fd70b0f43b3d8
run_id: 4575c73360594fc19a482318d573f43c
utc_time_created: '2024-08-26 14:41:18.521393'
```

Usable by any tool that can run Python

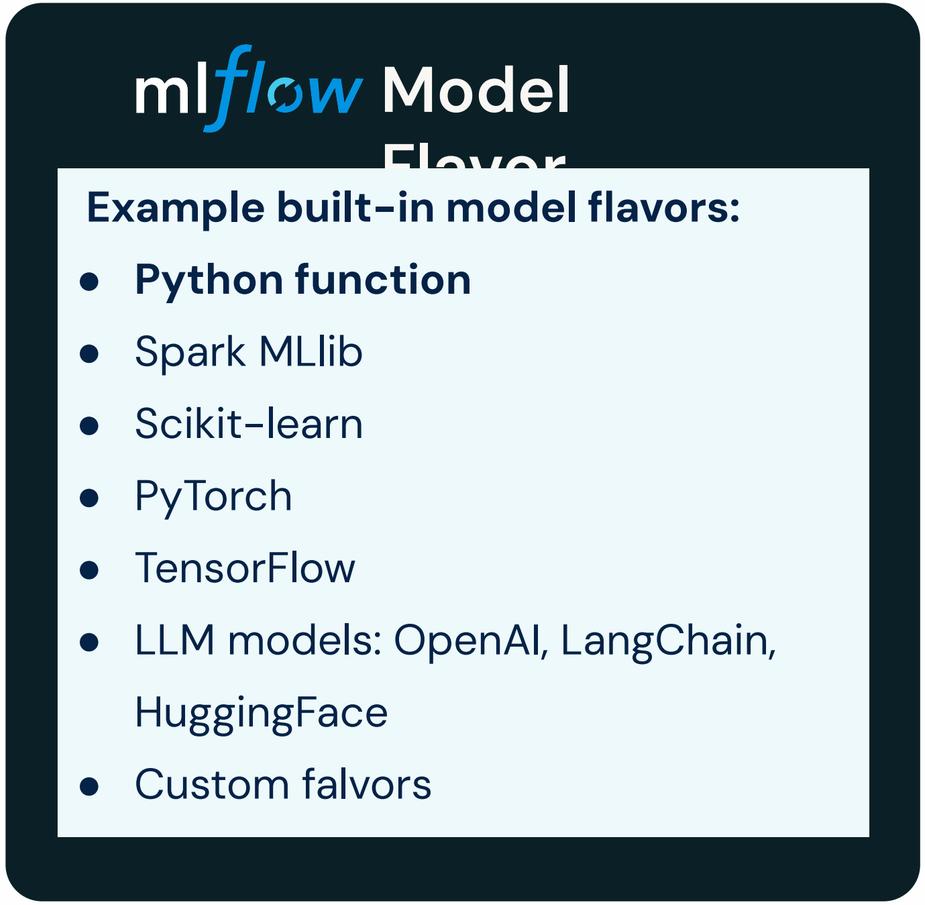
Usable by tools that understand Spark model format

MLflow Model ("Flavor")

Built-in model flavors

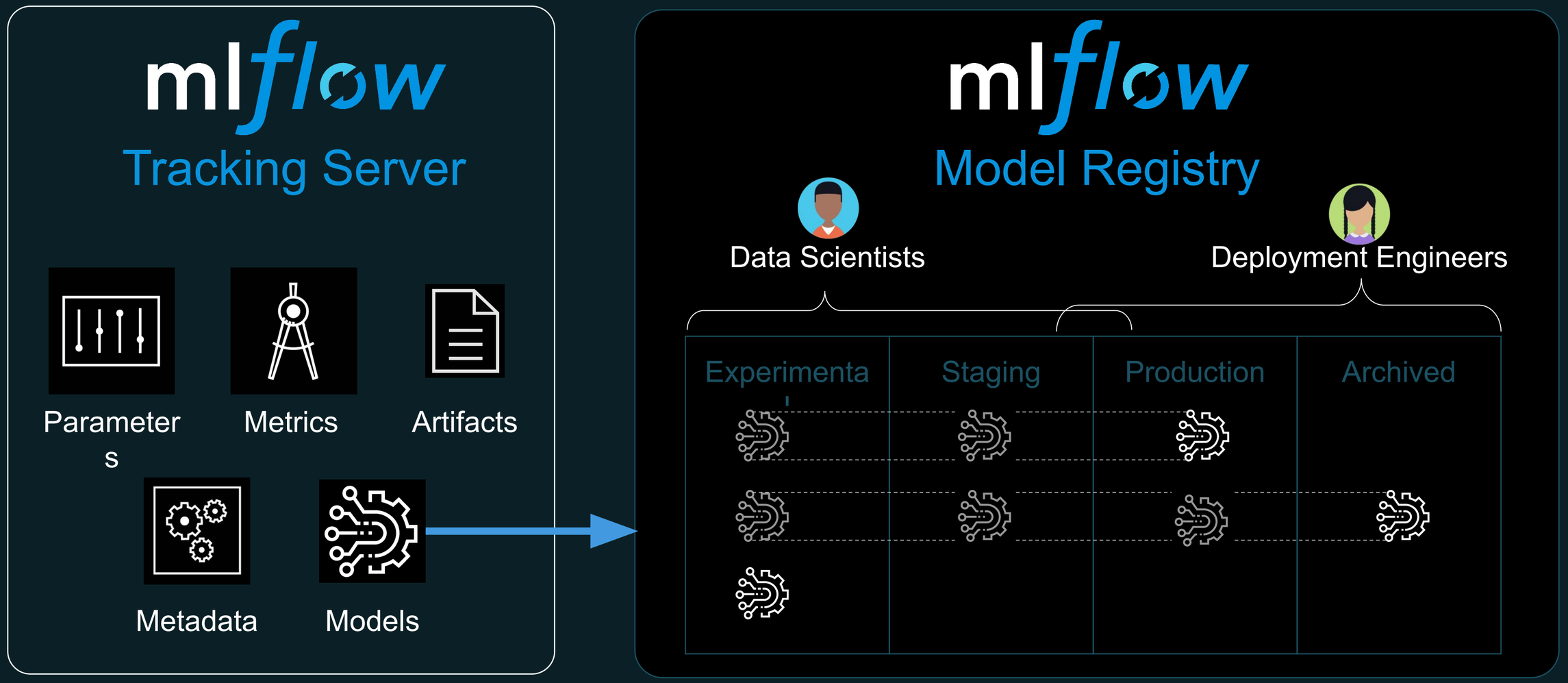
Python Function (`mlflow.pyfunc`):

- Serves as a **default model interface** for MLflow Python models.
- Any MLflow Python model is expected to be loadable as a python function.
- Allows you to deploy models as Python functions.
- It includes all the information necessary to **load and use a model**.
- Some functions: **`log_model`, `save_model`, `load_model`, `predict`**



mlflow Model Registry

VISION: Centralized and collaborative model lifecycle management



databricks

Register a Model to Unity Catalog

```
mlflow.set_registry_uri("databricks-uc")

X, y = datasets.wine_quality(return_X_y=True,
as_frame=True)
clf = RandomForestClassifier(max_depth=7)
clf.fit(X, y)

input_example = X.iloc[[0]]
print(input_example)

mlflow.spark.log_model(
   spark_model=clf,
   artifact_path="model",
   input_example=input_example,
registered_model_name="catalog.schema.model_name,
)
```

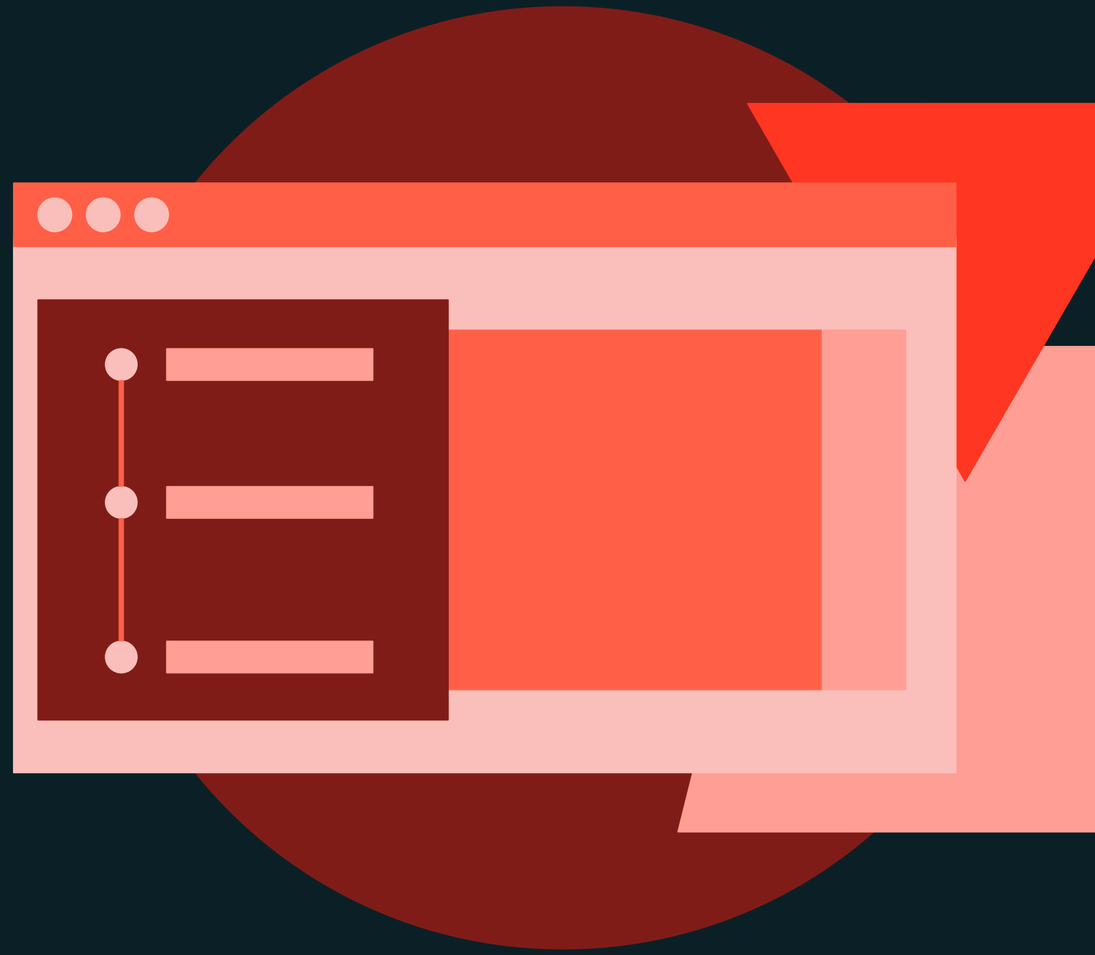
- Configure MLflow client to use the Unity Catalog Model Registry.
- MLflow uses first row of the training data to automatically infer the model's input signature (aka input schema)
- Handles the entire process of logging a model to the MLflow tracking server
- Optional. Create a model version in Unity Catalog under registered_model_name (creates new if doesn't exist).
  - Models can be registered independently using: mlflow.register_model(model_uri, name).

databricks

Machine Learning Development with Spark

DEMONSTRATION

Model Development with Spark



databricks

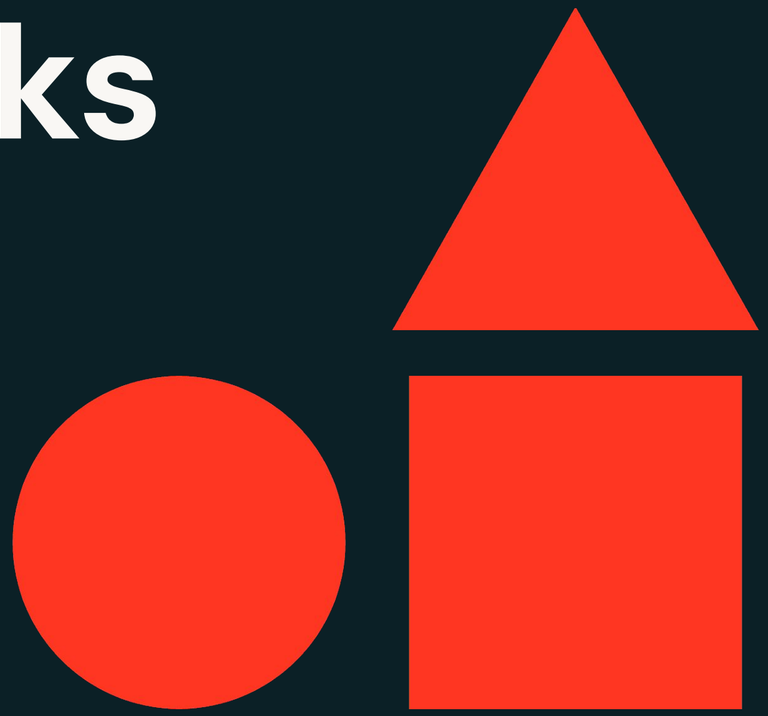
Machine Learning Development with Spark

LAB EXERCISE

Model Development with Spark

databricks

Distributed Model Tuning on Databricks



Machine Learning at Scale

databricks

Distributed Model Tuning on Databricks

LECTURE

Overview of Hyperparameter Tuning

Hyperparameter Tuning During Model Development

Use Spark....

- Hyperparameter Tuning involves training a lot of models with different hyperparameter values
- Model Tuning can be time consuming and is better not done manually.
- Testing more hyperparameter values → longer tuning process

Components of Hyperparameter Tuning

Key Elements for Efficient Model Optimization

Evaluation Metric

- The performance measure to improve (e.g., accuracy, F1-score, precision, recall, RMSE).
- Depends on the type of ML problem (regression, classification, forecasting) and criteria for success or failure.

Hyperparameters and Ranges

- Specific values for model parameters (e.g., learning rate, batch size).
- Depends on the chosen algorithm; documentation provides guidance.
- AutoML best run(s) provides insight for a starting point.

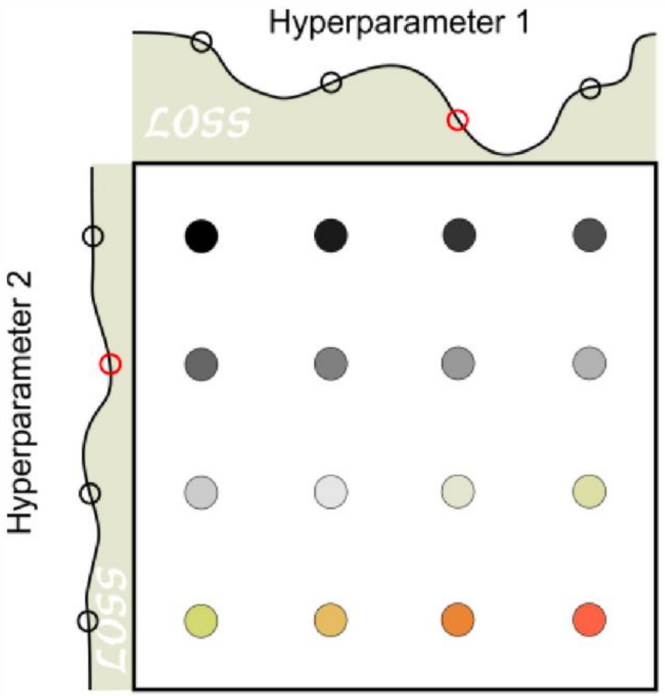
Job Configuration

- Define CPU/GPU usage, execution strategy (e.g., Bayesian, random search), parallelism, stopping criteria, and logging.
- Parallelism can increase efficiency.

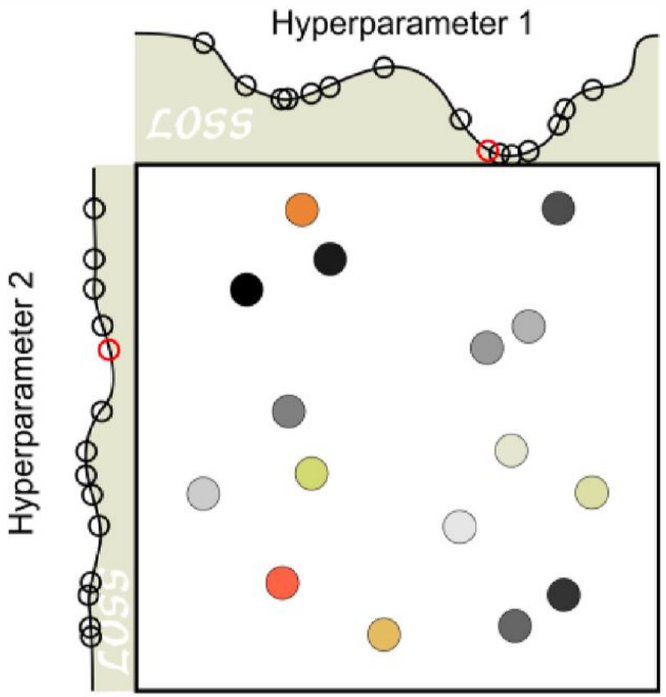
Optimizing Hyperparameter Values

Hyperparameter Search Methods

Grid Search



Random Search



Bayesian Optimization

The Tree of Parzen Estimators (TPE) algorithm is a popular Bayesian method

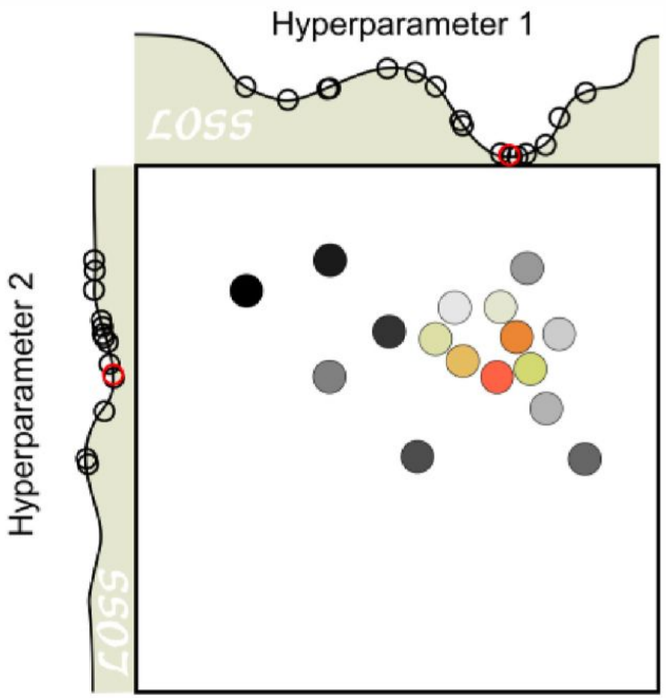


Image Source

databricks

Distributed Model Tuning on Databricks

LECTURE

Scalable HPO Frameworks on Databricks

Popular Tools for Hyperparameter Optimization

- **Spark ML:** Provides `CrossValidator` and `TrainValidationSplit` using distributed search on large datasets.

- **Optuna:** An open source hyperparameter optimization framework to automate hyperparameter search

- **Ray:** Ray Tune provides a scalable library for hyperparameter tuning, offering both grid and advanced search algorithms.

- **Hyperopt:** A popular library for performing hyperparameter optimization using Bayesian methods.

Note: The open-source version of Hyperopt is no longer being maintained. Hyperopt is no longer pre-installed on DRBML 17.0+. Databricks primarily recommends using Optuna or Ray.

Hyperparameter Optimization with Spark ML

Scalable hyperparameter tuning for ML models in Spark

Key Features:

- **Scalable & Distributed** – Leverages Spark's parallelism for large-scale tuning.
- **Built-in Cross-Validation** – Uses CrossValidator and TrainValidationSplit for systematic search.
- **Native ML Pipelines** – Integrates seamlessly with SparkML's pipeline-based model training.
- **Supports Grid Search & Random Search** – Search strategies using ParamGridBuilder.
- **Integrates with MLflow** – Logs experiment results for tracking and reproducibility.

```
# Define model, hyperparameter grid, and evaluator
dt = DecisionTreeRegressor(featuresCol="f", labelCol="l")

paramGrid = ParamGridBuilder()
    .addGrid(dt.maxDepth, [3, 5, 10])
    .addGrid(dt.minInstancesPerNode, [1, 2, 4]).
    build()
evaluator = RegressionEvaluator(labelCol="label",

predictionCol="prediction", metricName="rmse")

# Cross-validation and model training
cvModel = CrossValidator(
    estimator=dt, estimatorParamMaps=paramGrid,
    evaluator=evaluator, numFolds=5)
    .fit(trainingData)

# Log best model with MLflow
mlflow.spark.log_model(cvModel.bestModel,
"best_sparkml_model")
```

Optuna

A hyperparameter optimization framework

- Key Features:
  - Lightweight, versatile, and platform agnostic architecture
  - Pythonic search spaces
  - Efficient optimization algorithms
  - Easy parallelization
  - Quick visualization
  - Integration with MLflow

databricks

mlflow™

Recommended Frameworks

A hyperparameter optimization framework

- Key Features:
  - Lightweight, versatile, and platfor
  - Pythonic search spaces
  - Efficient optimization algorithms
  - Easy parallelization
  - Quick visualization
  - Integration with MLflow

```
# Define objective function
def objective(trial):
   Write your machine learning logic here!
    # Pythonic search spaces
    max_depth = trial.suggest_int("max_depth",2,20)

# Run optimization with parallelization
study = optuna.create_study(
    sampler=optuna.samplers.TPESampler())
study.optimize(objective, n_trials=50, n_jobs=-1)

mlflow.log_params(study.best_params)
mlflow.log_metric("best_value", study.best_value)

optuna.visualization.plot_optimization_history(
    study).show()
```

Multi-Algorithm or Multi-Metric Optimization

Advanced Strategies for Model Selection and Evaluation

Multi Metric

```
def objective(trial)
    Write your machine
    learning logic here!
    return accuracy, f1_score

study = optuna.create_study(
    directions=["maximize",
    "maximize"])
study.optimize(objective,
n_trials=  # of trials
```

Multi Model

See Sample Notebook

```
def objective(trial)
    classifier=trial.suggest_categorical(
        "classifier",["SVC","RandomForest"])
    if classifier == "SVC":
        Search Space, Model Initialization, etc.
    else:
        Search Space, Model Initialization, etc.
    return evaluation_score
.....................................................................
print(f"Best trial metrics: {trial.values}")
print(f"Best params: {list(trial.params.items())}")
optuna.visualization.plot_pareto_front
optuna.visualization.plot_pareto_front(study).show()
```

Advanced Options in Optuna

Streamline Your Hyperparameter Tuning

Access Trials' History

Attributes: e.g. study.best_params	
best_params	Return parameters of the best trial in the study.
best_trial	Return the best trial in the study.
best_trials	Return trials located at the Pareto front in the study.
best_value	Return the best objective value in the study.
direction	Return the direction of the study.
directions	Return the directions of the study.
metric_names	Return metric names.
system_attrs	Return system attributes.
trials	Return all trials in the study.
user_attrs	Return user attributes.

Documentation

Not Satisfied … Continue to Optimize

- Optuna can resume the optimization after it is finished if not satisfied with the results.
- To continue searching, call optimize again.
- **load_if_exists=True** is default value for create_study.

```
study.optimize(objective,
  n_trials=100)
```

Resume Study

Objective Function with Multiple Params

- Recommended syntax is to wrap it inside another function. *Generally with a lambda function*

```
def objective(trial, X, y, cv, scoring):
    Machine Learning Logic

# Wrap the objective
kf = KFold(n_splits=5, random_state=18)
# Pass additional arguments inside a function
func = lambda trial: objective(trial, X, y,
  cv=kf, scoring="neg_mean_squared_log_error")

# Start optimizing with 100 trials
study.optimize(func, n_trials=100)
```

Kaggle Guide

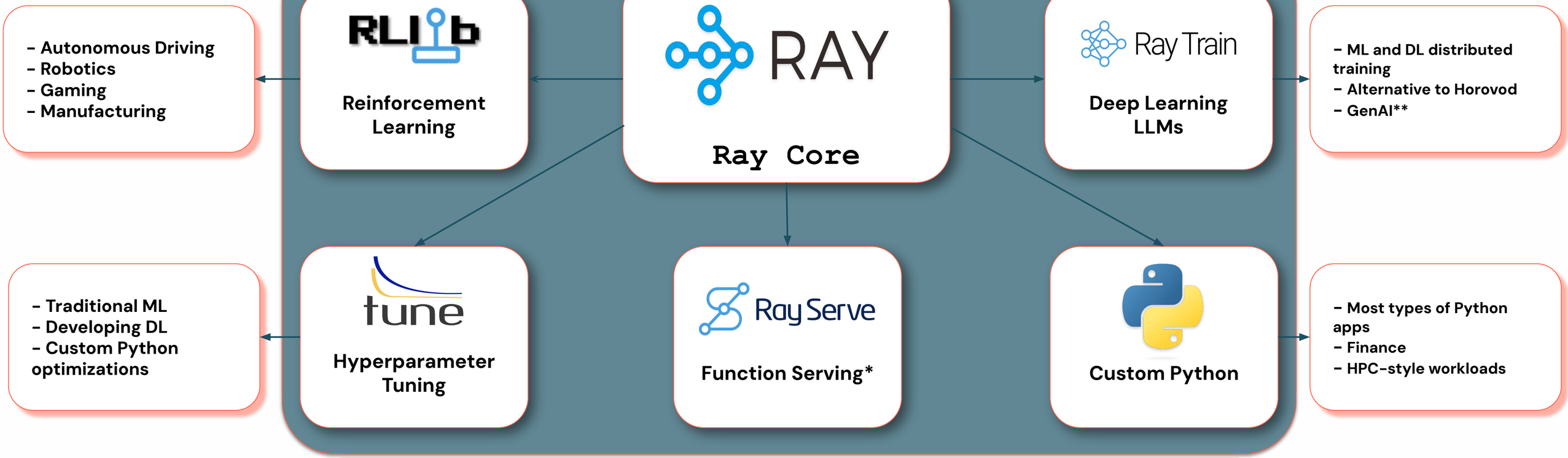
Ray Tune

Ray is a unified framework for scaling AI and Python applications.

- Key Features:
  - Ray Tune is a **scalable library** for **hyperparameter tuning**, built on top of the Ray distributed computing framework.
  - **Easily scale** from a single machine to large distributed clusters **without modifying code**.
  - Tune your favorite **machine learning framework** (PyTorch, XGBoost, TensorFlow and Keras, and more)
  - Ray Tune provides state-of-the-art tuning methods like **ASHA, BOHB, and Population-Based Training**.
  - **I**ntegrates with **TensorBoard & MLflow** for tracking results.

Ray Ecosystem

Integrates with many third-party OSS tools and libraries built on top of Ray***

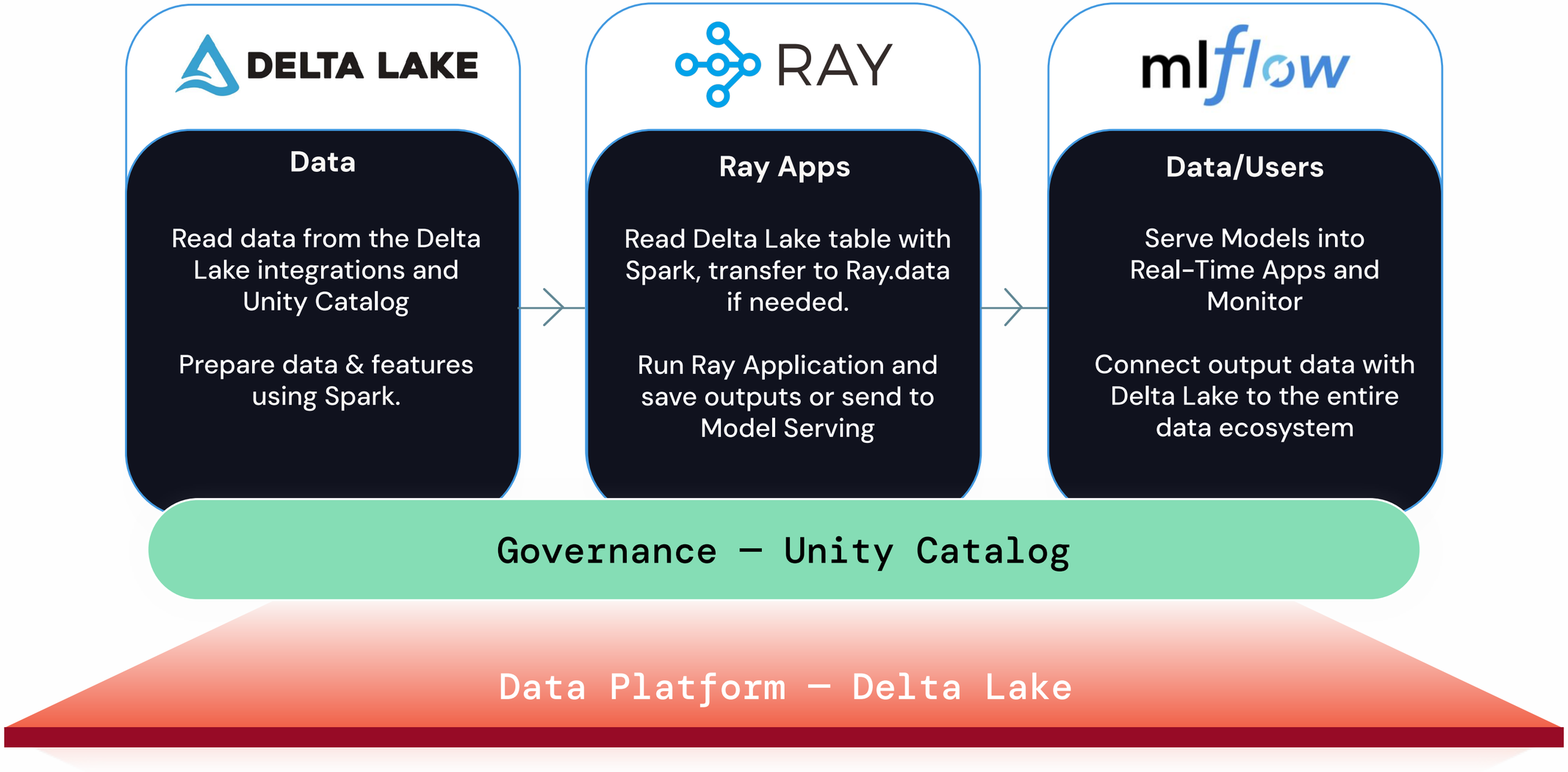


*Ray Serve applications are not recommended on Databricks, use Model Serving instead.

**For LLM use cases, Mosaic is recommended.

***Not all OSS tools built with Ray are supported on Databricks, such as tools built for Kubernetes

Ray on Databricks combines Spark and Ray



Key Concepts of Ray Tune

Six Crucial Components

- Trainables
- Search Space
- Search Algorithms
- Schedulers
- Trials
- Analyses

```
import ray
from ray import tune

# Define the objective function
def train(config):
    score = config["param"] ** 2  # Example computation
    tune.report(score=score)

# Define the search space
search_space = {"param": tune.uniform(-10, 10)}

# Run the tuning process
tuner = tune.Tuner(train, param_space=search_space,
    tune_config=tune.TuneConfig(
        search_alg=tune.suggest.basic_variant.BasicVariantGenerator(),
        scheduler=tune.schedulers.ASHAScheduler()))

results = tuner.fit()

# Analyze results
print("Best Config:", results.get_best_result(metric="score",
mode="min").config)
```

Defining Trainables in Ray Tune

An object (Function or Class) that you can pass into a Tune run

Two Ways to Define a Trainable

- **Function API (train.report())** – Simple & quick for most cases.
- **Class API (tune.Trainable)** – Provides more flexibility & control.

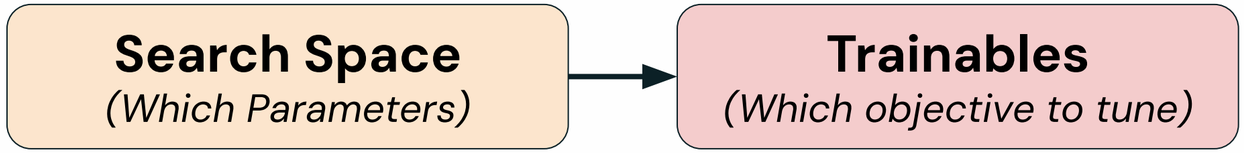
Key Features

- Pass **hyperparameters dynamically** through **config**.
- **Automatic checkpointing** for state recovery & fault tolerance
- **Parallel execution** with distributed Ray actors.
- Can use tune.stop() and define custom conditions for **early stopping.**

```
import xgboost as xgb

def train_function(config):
    context = get_context() # Access trial metadata

    # Define model parameters
    params = { "objective": "reg:squarederror", "eval_metric":
    "rmse", "max_depth": config["max_depth"] }

    # Train Model
    model = xgb.train(params, dtrain,
        num_boost_round=config["num_boost_round"])

    # Report RMSE and save checkpoint
    tune.report(
        rmse=float(model.eval(dtrain).split(":")[-1]))

    # Log experiment directory and trial ID
    print(f"Trial {context.trial_id} running in
        {context.experiment_dir}")
```

Search Space

Defines valid values and selection methods for hyperparameters.



Key Features

- Define search space using Tuner(param_space=...).
- **Avoid large objects** in param_space for better performance—use tune.with_parameters() instead.
- Use **tune.sample_from(func)** for dependent hyperparameters.
  - Only some algorithms (e.g., **HyperOpt**, **Optuna**) support conditional search spaces.
  - Default **Random Search** and **Grid Search** (BasicVariantGenerator) support all distributions.

```
tuner = tune.Tuner(
    trainable,
    param_space={
        "param1": tune.choice([True,
                  False]),
        "bar": tune.uniform(0, 10),
        "alpha": tune.sample_from
                 (lambda _:
                 np.random.uniform(100)
                  ** 2),
        "const": "hello"})# constant
```

Tune Search Space API

Random Distributions
tune.uniform
tune.quniform
tune.loguniform
tune.qloguniform
tune.randn
tune.qrandn
tune.randint
tune.qrandint
tune.lograndint
tune.qlograndint
tune.choice
Grid Search and Custom
tune.grid_search
tune.sample_from

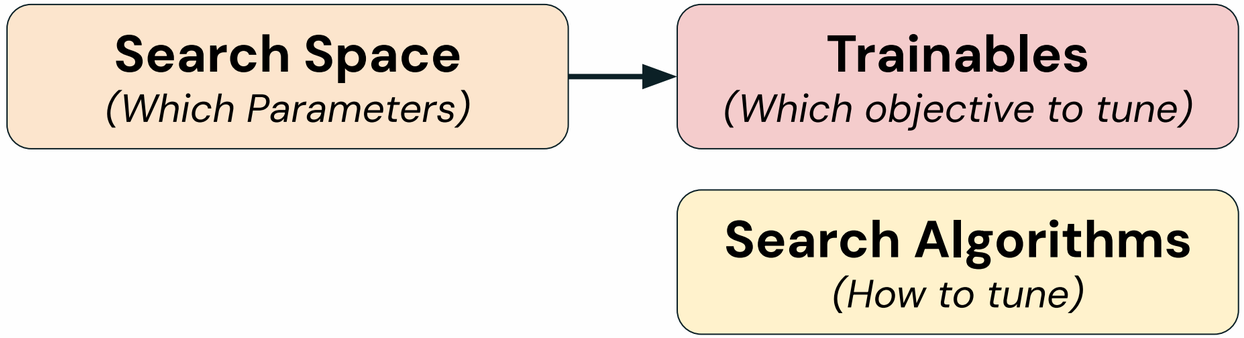
Search Algorithms

Efficient Strategies for Hyperparameter Optimization in Ray Tune

Key Points

- If no search algorithm is specified, Ray Tune **uses Random Search** *(default)*
- **Supports Popular Libraries for Flexible Search Strategies:**
  - **Bayesian Optimization:** Ax, BOHB, HyperOpt.
  - **Gradient-Free Methods:** Nevergrad, Random Search.
- Supports **parallel execution across CPUs/GPUs** for faster tuning.

```
search_alg = OptunaSearch()
```

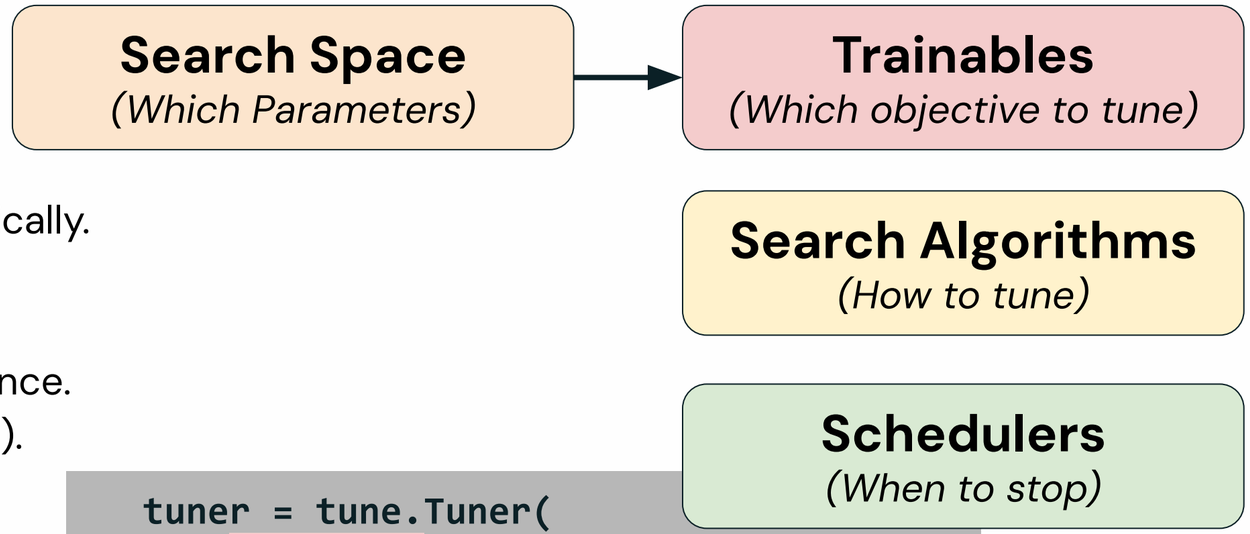


SearchAlgorithm	Summary	Website	Code Example
Random search/Grid search	Random search/grid search		tune_basic_example
AxSearch	Bayesian/Bandit Optimization	[Ax]	AX Example
HyperOptSearch	Tree-Parzen Estimators	[HyperOpt]	Running Tune experiments with HyperOpt
BayesOptSearch	Bayesian Optimization	[BayesianOptimization]	BayesOpt Example
TuneBOHB	Bayesian Opt/HyperBand	[BOHB]	BOHB Example
NevergradSearch	Gradient-free Optimization	[Nevergrad]	Nevergrad Example
OptunaSearch	Optuna search algorithms	[Optuna]	Running Tune experiments with Optuna
Repeated Evaluations	Support for running each sampled hyperparameter with multiple random seeds.		
ConcurrencyLimiter	Limits the amount of concurrent trials when running optimization.		
Shim Instantiation	Allows creation of the search algorithm object given a string.		

Tune Schedulers

Schedulers can Stop, Pause, or Tweak hyperparameters of running trials

- **Trial Schedulers** can:
  - **Stop** underperforming trials early.
  - **Pause & Resume** trials to optimize resources.
  - **Clone trials** and **tweak** hyperparameters dynamically.
- **How Schedulers Work:**
  - Use a **metric** (e.g., "loss") to decide trial performance.
  - Maximize or minimize the metric (e.g. mode="min").
- **Default Behavior:**
  - If no scheduler is specified, Ray Tune uses a **FIFO sch**
  - FIFO **does not perform early stopping**—it simply run




```
tuner = tune.Tuner(
    train_fn,
    tune_config=tune.TuneConfig(
        scheduler=ASHAScheduler(),
        metric="loss",
        mode="min",
        num_samples=10),
    param_space=config)
```

Tune Trial Schedulers

Trials

Schedulers can Stop, Pause, or Tweak hyperparameters of running trials

What is a Trial?

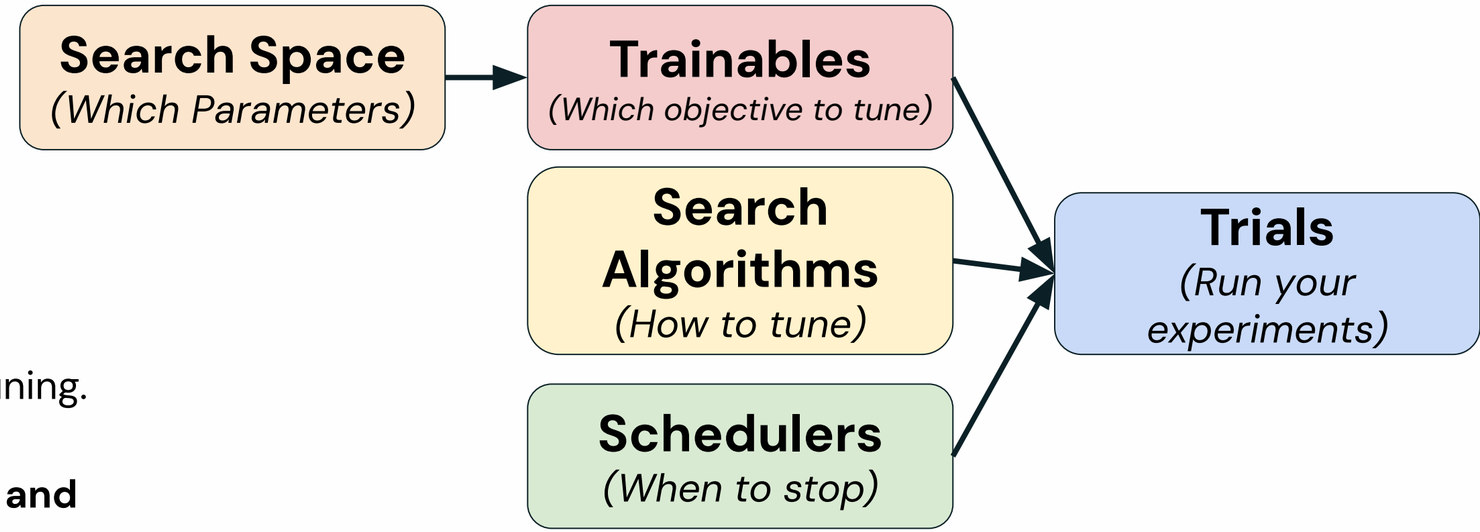
- A single experiment run within Ray Tune.
- Each trial corresponds to a unique set of hyperparameters.

How Trials are Managed:

- Tuner.fit() runs and manages hyperparameter tuning.
- Requires a **trainable** and a **param_space**
- Optionally accepts **tune, runtime, checkpoint, and failure** configurations.

Default Behavior:

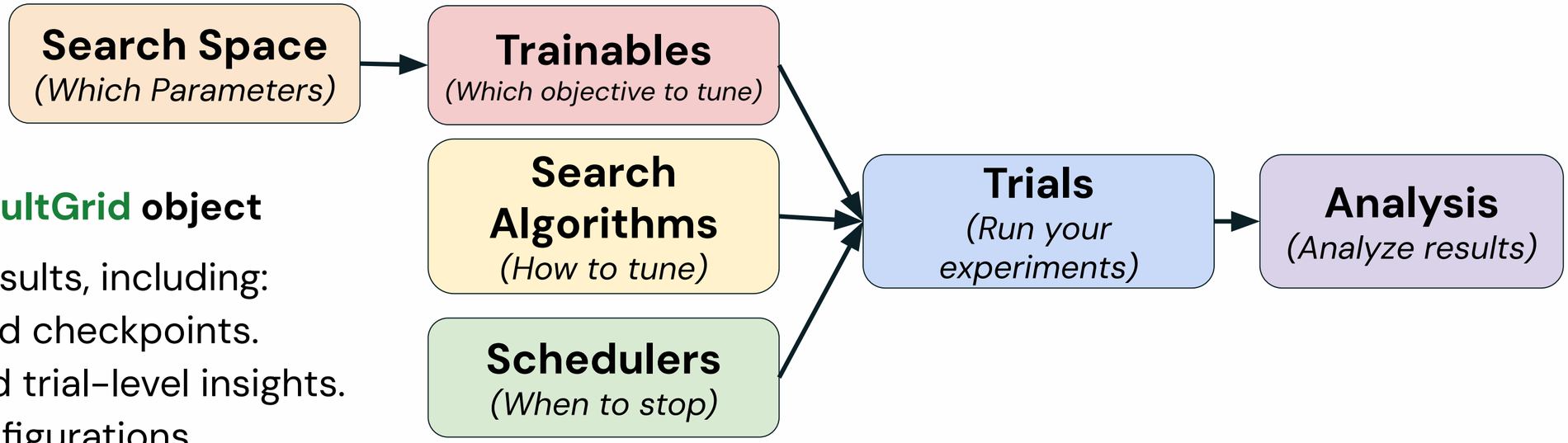
- Tuner.fit() **runs until all trials complete** or an error occurs.
- Returns a **ResultGrid**, allowing you to inspect and analyze results.




```
space = {"a": tune.uniform(0, 1), "b":
    tune.uniform(0, 1)}
tuner = tune.Tuner(
    trainable, param_space=space,
    tune_config=tune.TuneConfig(num_samples=10)
)
tuner.fit()
```

Tune Execution

Analysis



Tuner.fit() returns an ResultGrid object

- Stores all experiment results, including:
  - Trial logs, metrics, and checkpoints.
  - Experiment-level and trial-level insights.
  - Hyperparameter configurations.

```
results = tuner.fit()

best_result = results.get_best_result()  # Get best result object
best_config = best_result.config  # Get best trial's hyperparameters
best_logdir = best_result.path  # Get best trial's result directory
best_checkpoint = best_result.checkpoint  # Get best trial's best checkpoint
best_metrics = best_result.metrics  # Get best trial's last results
best_result_df = best_result.metrics_dataframe  # Get best result as pandas dataframe
df = results.get_dataframe(filter_metric="score", filter_mode="max") # Get a dataframe of results for a
    specific score or mode
best_result.metrics_dataframe.plot("training_iteration", "mean_accuracy") # Plot accuracy over training
    iterations for the best trial
```


Result Analysis User Guide

Optional Deep Dive

Optuna: Lightweight, Versatile, and Platform Agnostic

Handle a wide variety of tasks with a simple installation and minimal requirements.

- Optuna is written entirely in Python with few dependencies.
- Intuitive API:
  - Define parameters within your code.
  - No need for framework-specific syntax.
- Platform and Framework Agnostic
  - e.g. PyTorch, TensorFlow, Keras, Scikit-Learn, XGBoost, LightGBM, etc.

```
import optuna


def objective(trial)


   Write your machine
   learning logic here!

   return evaluation_score

study =
optuna.create_study(direction=”maximize”)
study.optimize(objective, n_trials= # of trial )
```

Optuna: Pythonic Search Spaces

Define search spaces using familiar Python syntax including conditionals and loops.

- Define the search space during optimization using Python language
- Optuna can use any Python syntax to set up such space
- Reduces the amount of code and make optimization setup more intuitive

trial.suggest_categorical()	Suggest a value for the categorical parameter.
suggest_discrete_uniform(name, low, high, q)	Suggest a value for the discrete parameter.
suggest_float(name, low, high, *[, step, log])	Suggest a value for the floating point parameter.
suggest_int(name, low, high, *[, step, log])	Suggest a value for the integer parameter.
suggest_loguniform(name, low, high)	Suggest a value for the continuous parameter.
suggest_uniform(name, low, high)	Suggest a value for the continuous parameter.

```
def objective(trial)
   mxdepth = trial.suggest_int("mxdepth", 2, 32)
   classifier_obj = RandomForestClassifier(
     max_depth=mxdepth, n_estimators=10)
```

See suggest_int Sample

Optuna: Efficient Optimization Algorithms

2 Components of Optimization Algorithms: Sampling Strategy and Pruning Strategy

- Sampling Strategy determines how to decide the next hyperparameter

- Continuously refine the search space using past evaluation results to focus on promising areas.

GridSampler	Grid Search
RandomSampler	Random Search
TPESampler	Tree-structured Parzen Estimator algorithm (default)
CmaEsSampler	CMA-ES based algorithm
GPSampler	Gaussian process-based algorithm
PartialFixedSampler	Algorithm to enable partial fixed parameters
NSGAIISampler	Nondominated Sorting Genetic Algorithm II
QMCSampler	Quasi Monte Carlo sampling algorithm

```
study = optuna.create_study(sampler=optuna.samplers.GridSampler())
print(f"Sampler is {study.sampler.__class__.__name__}")
```

Optuna: Efficient Optimization Algorithms

2 Components of Optimization Algorithms: Sampling Strategy and Pruning Strategy

- Early stopping halts unpromising trials early based on intermediate results.
- Currently pruners are used only for single-objective optimization
- Call report() and should_prune() after each step to activate pruning.

optuna.pruners.BasePruner	Base class for pruners.
MedianPruner	Pruner using the median stopping rule.
NopPruner	Pruner which never prunes trials.
PatientPruner	Pruner which wraps another pruner with tolerance.
PercentilePruner	Pruner to keep the specified percentile of the trials.
SuccessiveHalvingPruner	Pruner using Asynchronous Successive Halving Algorithm.
HyperbandPruner	Pruner using Hyperband.
ThresholdPruner	Pruner to detect outlying metrics of the trials.
WilcoxonPruner	Pruner based on the Wilcoxon signed-rank test.

```
for step in range(100):
   clf.partial_fit(train_x, train_y, classes=classes)

   # Report intermediate objective value.
   intermediate_value = 1.0 - clf.score(valid_x, valid_y) trial.report(intermediate_value, step)

   # Handle pruning based on the intermediate value.
   if trial.should_prune():
      raise optuna.TrialPruned()
```

Easy Parallelization

Scale studies with little or no changes to the code.

Variations of parallelization:

- **Multi-threading (Single Node)**
  - Use **n_jobs** in study.optimize()
- **Multi-processing (Single Node)**
  - Also requires JournalFileBackend, SQLite3 or client/server RDB *(PostgreSQL, MySQL)*
- **Multi-processing (Multiple Nodes)**
  - Needs a shared client/server RDB

```
import optuna

def objective(trial)
    Write your machine
    learning logic here!
    return evaluation_score

study = optuna.create_study(
    study_name = ‘distributed-example’,
    storage = ‘mysql://root@localhost/example’,
    direction="maximize")
study.optimize(objective, n_trials=100, n_jobs=-1)
```

Optuna: Quick visualization

```
fig = optuna.visualization.plot_optimization_history(study)
fig.show()
```

The visualization module provides utility functions for plotting the optimization process using plotly and matplotlib.

optuna.visualization.matplotlib.plot_edf optuna.visualization.plot_edf	Plot the objective value EDF-empirical distribution function-of a study.
plot_contour	Plot the parameter relationship as contour plot in a study.
plot_hypervolume_history	Plot hypervolume history of all trials in a study.
plot_intermediate_values	Plot intermediate values of all trials in a study.
plot_optimization_history	Plot optimization history of all trials in a study.
plot_parallel_coordinate	Plot the high-dimensional parameter relationships in a study.
plot_param_importances	Plot hyperparameter importances.
plot_pareto_front	Plot the Pareto front of a study.
plot_rank	Plot parameter relations as scatter plots with colors indicating ranks of target value.
plot_slice	Plot the parameter relationship as slice plot in a study.
plot_terminator_improvement	Plot the potentials for future objective improvement.
plot_timeline	Plot the timeline of a study.
is_available	Returns whether visualization with plotly is available or not.

Optuna Integrates with MLflow

To track hyperparameters and metrics of all the Optuna trials, use the MLflowCallback

This callback adds relevant information that is tracked by Optuna to MLflow

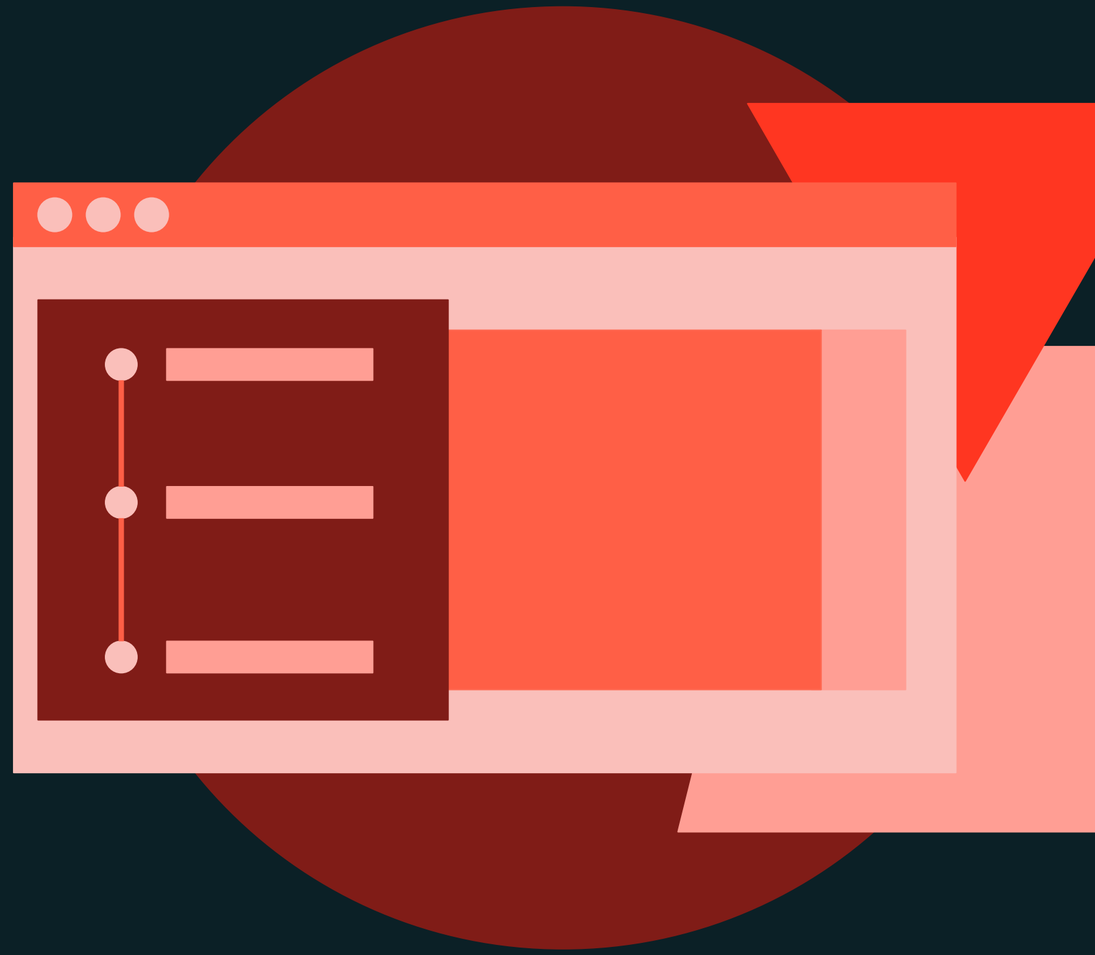
```
import mlflow
from optuna.integration.mlflow import MLflowCallback

mlflow_callback = MLflowCallback(
    tracking_uri="databricks",
    metric_name="accuracy",
    create_experiment=False,
    mlflow_kwargs={
        "experiment_id": experiment_id
    }
)

study.optimize(objective, n_trials=100, callbacks=[mlflow_callback])
```

databricks

Distributed Model Tuning on Databricks

DEMONSTRATION

Hyperparameter Tuning with SparkML

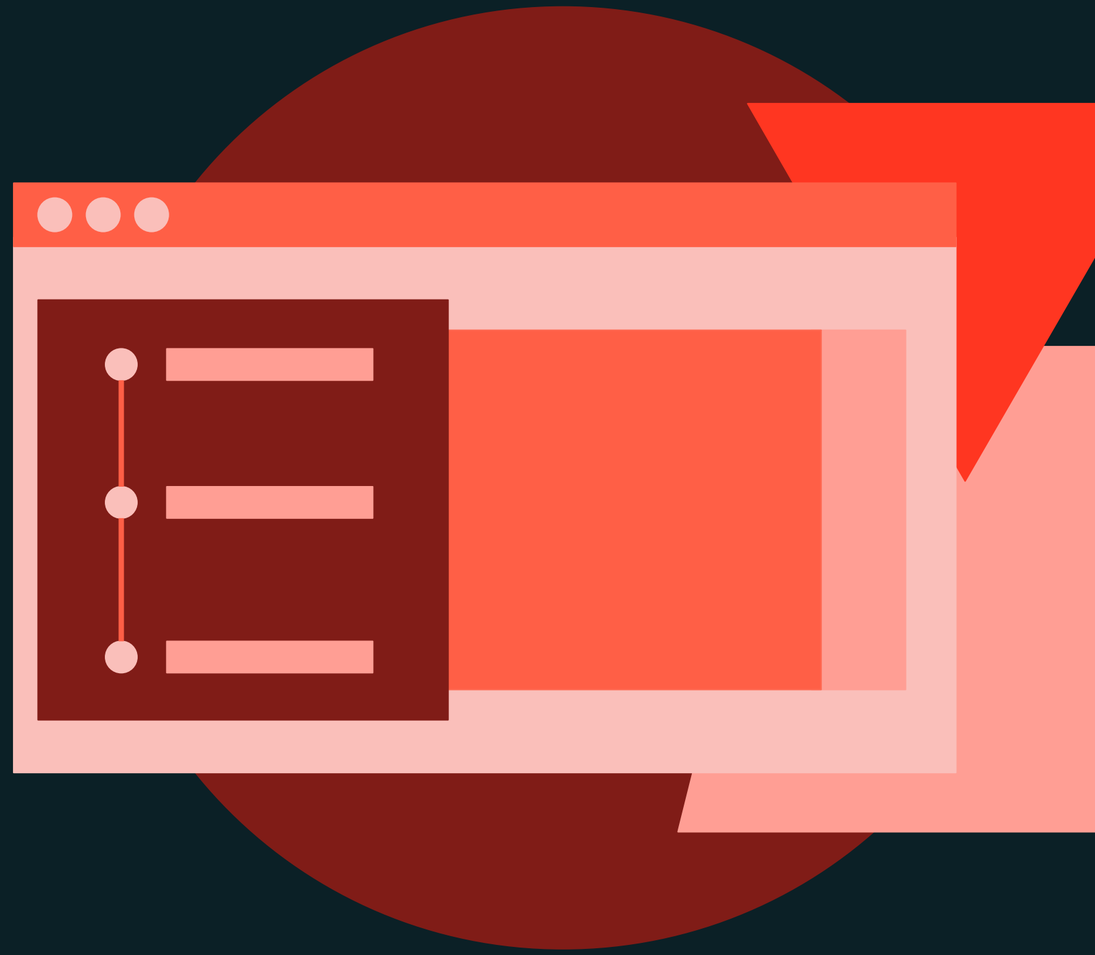


databricks

Distributed Model Tuning on Databricks

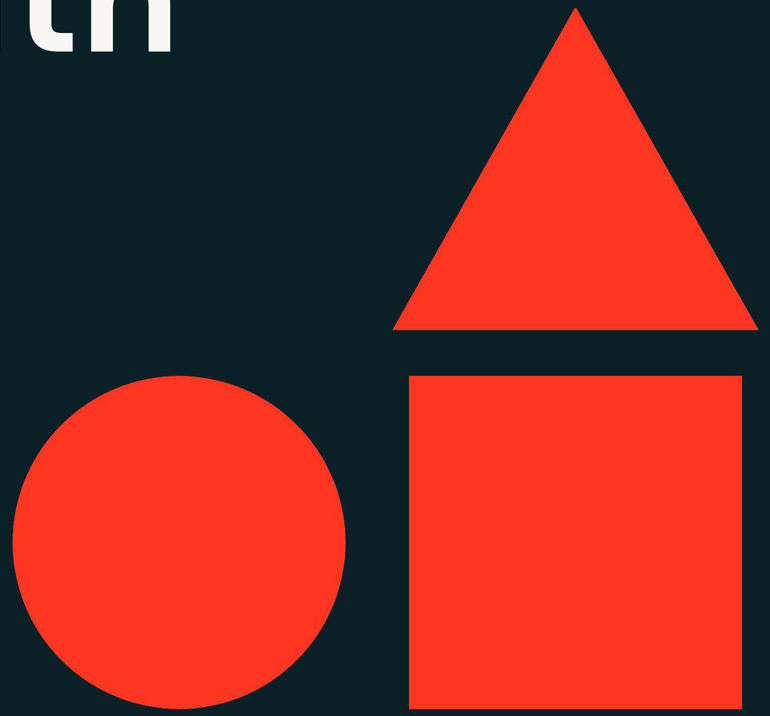
DEMONSTRATION

HPO with Ray Tune



databricks

Deploying Machine Learning Models with Spark



Machine Learning at Scale

databricks

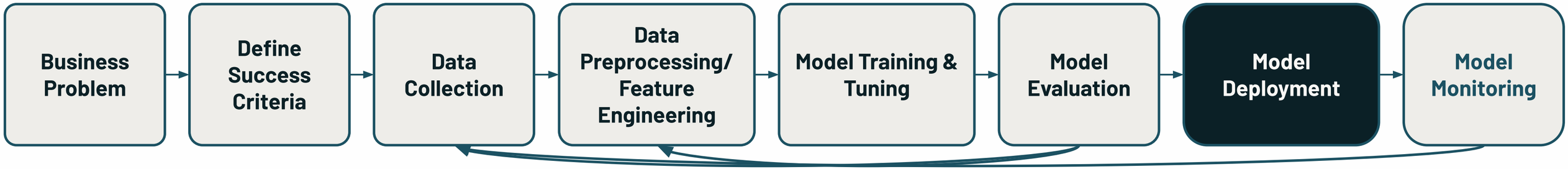
Deploying Machine Learning Models with Spark

LECTURE

Deployment with Spark

Model Deployment

Your Model is ready, then what?



The process of integrating a machine learning model into a **production environment**, making it accessible for end-users or other systems to **generate predictions** or insights.

(Deployment Strategies: batch, streaming, real-time, or embedded/edge)

Model Deployment Modes

Comparing methods

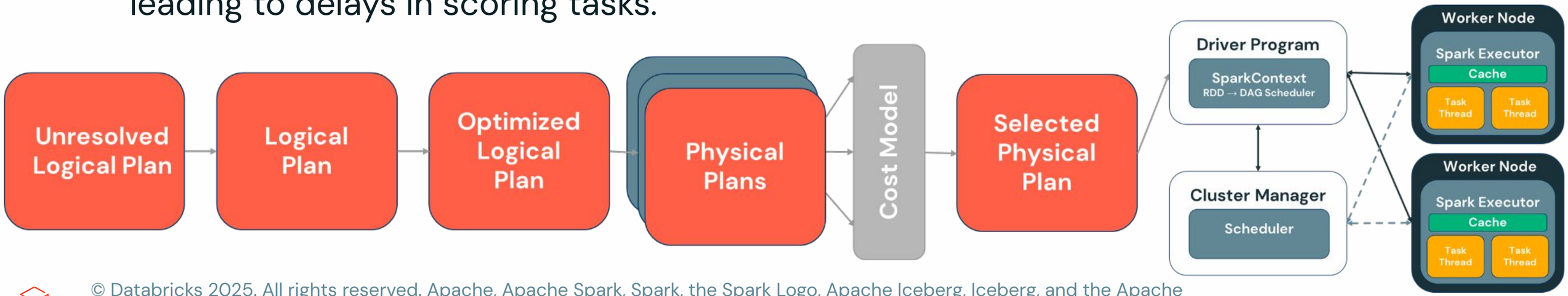
Deployment Method	Through-put	Latency	Best For	Example Application
Batch *80-90% of Deployments*	High	High (*hours to days*)	• Large-scale data processing when immediate results are not critical. • Prediction stored in persistent storage.	Periodic customer churn prediction
Streaming *10-15% of Deployments*	Moderate	Moderate (*seconds to minutes*)	• Data flows continuously and timely updates are crucial. • Processed in “micro-batches”	Dynamic pricing application
Real-time *5-10% of Deployments*	Low	Low (*milliseconds*)	• Scenarios requiring quick decision making and responsiveness to user interactions. • Usually using REST	Recommendation systems Chatbot
Edge/Embedded	Low	Low (Dependent on device processing power)	• Situations where data processing needs to be close to the data source • Models may need to be deployed in various hardware setups.	IoT Applications. Farm sensor for detecting humidity

***Note: The percentages are estimates and in flux with the increased focus Generative AI applications emphasis on direct user interactions.*

Why is Spark Slow in Online Scoring?

Spark is primarily designed for batch and streaming deployments.

- Spark generates complex **execution plans**, which are optimized for throughput rather than latency.
- Each job in Spark involves significant **overhead,** including the scheduling of tasks, serialization/deserialization of data, and coordination across nodes.
- **In-memory computation** can cause frequent garbage collection and disk spillage, leading to delays in scoring tasks.



Deploying a Single-Node model on a Spark DataFrame

Why would you use Single-Node model using Spark DataFrame API?

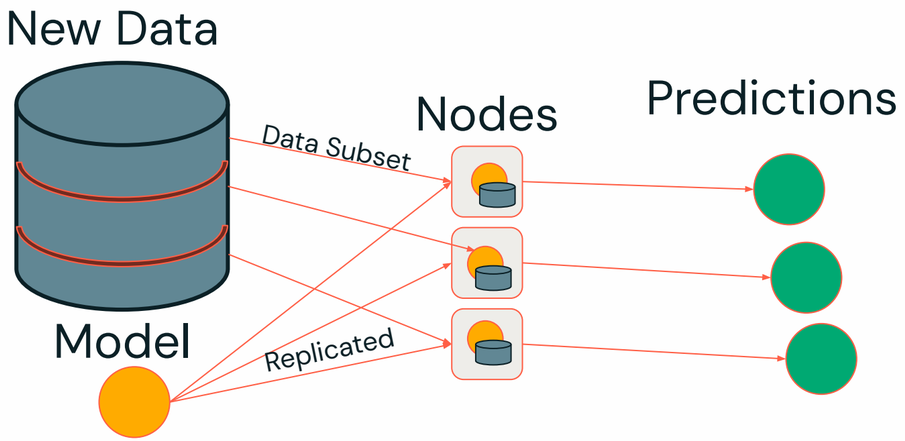
- **Ease of Use**: Simplifies distributed data manipulation with domain-specific language.
- **Scalability**: Supports larger datasets by distributing data across multiple nodes.
- **ML Integration**: Directly integrates with MLlib for advanced models and pipelines.
- **Cost Efficiency**: More cost-effective than distributed systems, with flexible resource management.
- **Flexibility**: Seamlessly switch between Pandas and Spark DataFrames.
- **Advanced Analytics**: Enables complex user functions for sophisticated analysis.
- *And more...*

Answer: Leverage Spark's distributed data processing capabilities while still using a model that is designed to run on a single machine

Deploying a Spark ML Model on a Spark DataFrame

Why would you use Spark ML Model using Spark DataFrame API?

- Distributed Training and Native Distributed Inference
- Native Integration with Spark's Distributed Computing Environment and DataFrame API
- Built to Handle Very Large Datasets
- Facilitates End-To-End ML pipelines within the Spark's Libraries and Toolsets.

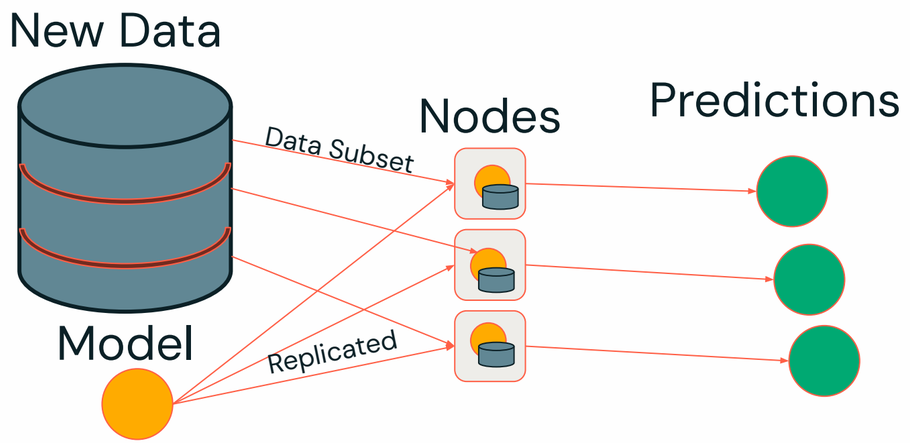


Answer: Harness the full power of distributed computing for both Data Processing, Model Training, and Inferencing.

Deploying a Spark ML Model on a Spark DataFrame

Why would you use Spark ML Model using Spark DataFrame API?

```
from pyspark.ml import PipelineModel

# Step 1: Load the DataFrame directly from Unity Catalog
df = spark.table("catalog_name.schema_name.table_name")
df.show()

# Step 2: Load a pre-trained Spark ML model
model = PipelineModel.load("/mnt/models/path_to_model")

# Step 3: Perform inference on the DataFrame
predictions = model.transform(df)

# Step 4: Show the predictions
predictions.select("id", "prediction").show()  # Adjust based on your data

# Step 5: Save the predictions back to a Delta table in Unity Catalog
predictions.write.format("delta").mode("overwrite")
   .saveAsTable("catalog_name.schema_name.predictions_table")
```



Answer: Harness the full power of distributed computing for both Data Processing, Model Training, and Inferencing.

databricks

Deploying Machine Learning Models with Spark

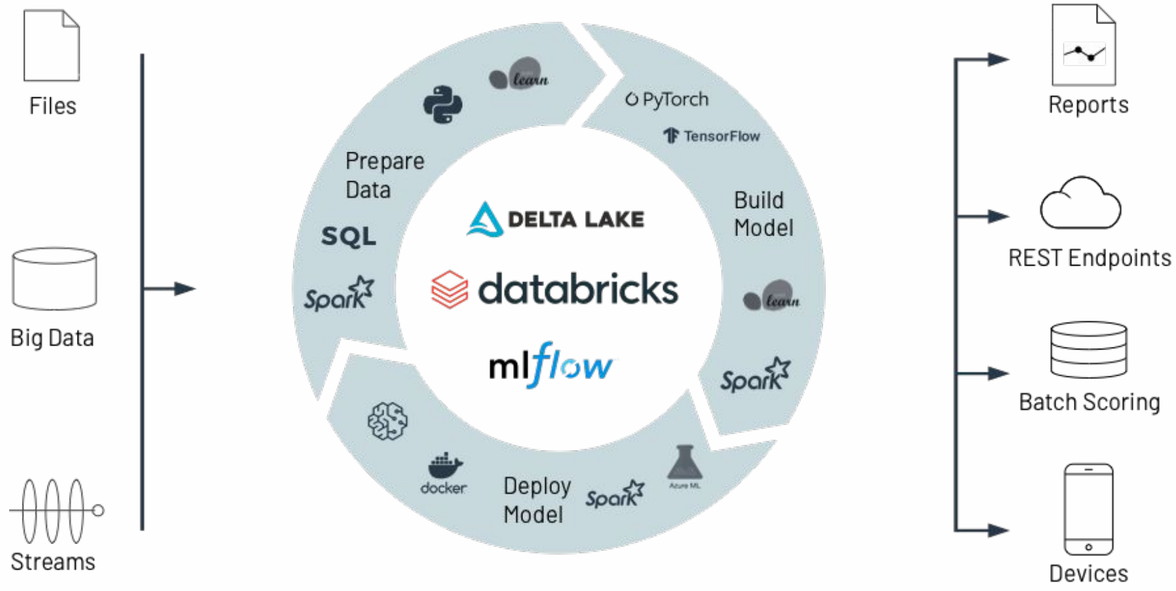
LECTURE

Inference with Spark

Approaches to Model Inferencing with Spark

Methods for Deploying and Running Inference

- **Batch Inferencing:** Use a Spark batch pipeline with mlflow.pyfunc.spark_udf for single-node models or Spark ML pipelines for Spark ML models.
- **Streaming Inferencing:** For near-real-time scoring, use Spark Structured Streaming or MLflow PySpark UDFs with Delta Live Tables for continuous inference on streaming data or incremental batch processing (e.g., Trigger.AvailableNow).
- **Real-Time Inferencing:** For low-latency needs, deploy models to a serving endpoint for real-time inference (not recommended for SparkML models)



Parallel Inference on Spark with MLflow

Leverage **mlflow.pyfunc.spark_udf** for Scalable Model Inferencing

Spark Approach:

- **Load the Model:** Load a Python function formatted model from MLflow Model Registry
- **Convert:** MLflow model is converted into a Spark UDF (User-Defined Function).
- **Apply in Parallel:** Distribute inference across Spark DataFrame partitions for parallel inference using withColumn..

Classic Approach: Load as a Python function

```
model = mlflow.pyfunc.load_model(model_path)
model.predict(model_input)
```

Spark Approach: Convert model to a Spark UDF

```
# load input data table as a Spark DataFrame
input_data = spark.table(input_table_name)
model_udf = mlflow.pyfunc.spark_udf(spark, model_path)
df = input_data.withColumn("prediction", model_udf("features"))
```

Note: The Spark approach optimizes resource usage by loading the model only once per Python process, ensuring efficient reuse during inference.

Driving Predictive Insights with Delta Lake

Efficient Data Management for Scalable Querying and Dynamic Workloads

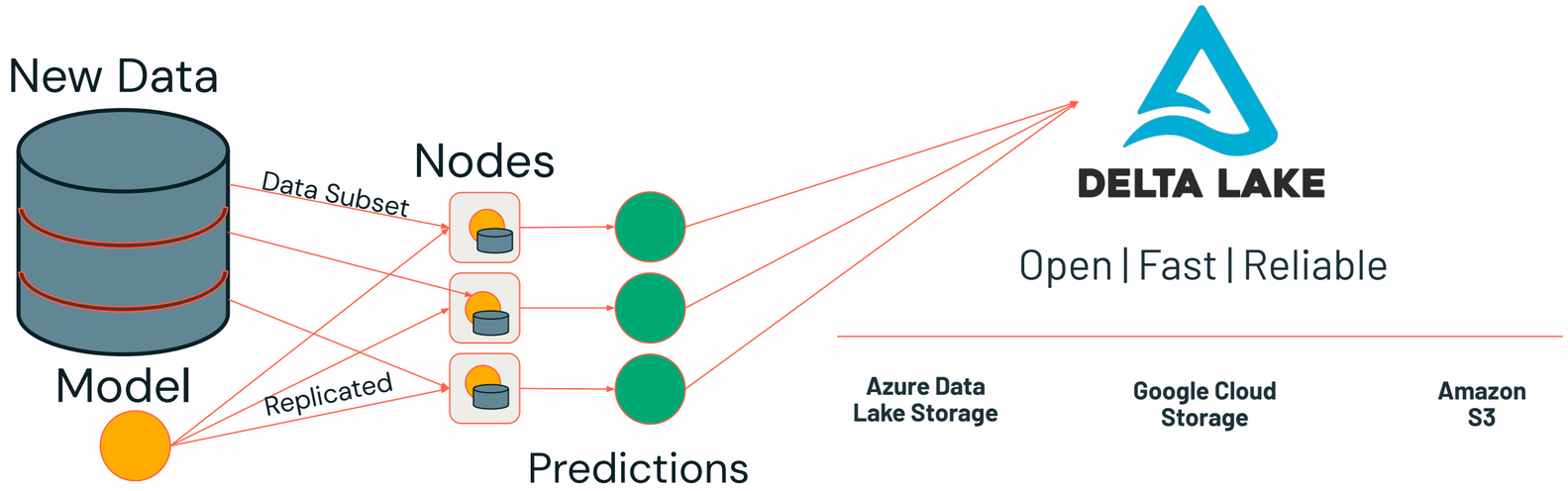
Delta Lake is the default format for tables created in Databricks

```
CREATE TABLE foo
USING DELTA
```

```
df.write
    .format("delta")
```

Delta Lake A **unified data management layer** that brings data reliability and fast analytics to cloud data lakes.

Support real-time and batch data with scalable metadata, performance optimization and more for optimal data usage.



BI and SQL Analytics

- Gain Insights
- Data Driven Decisions

Data Quality

- Optimized Performance
- Data Consistency and Reliability

Data Integrity

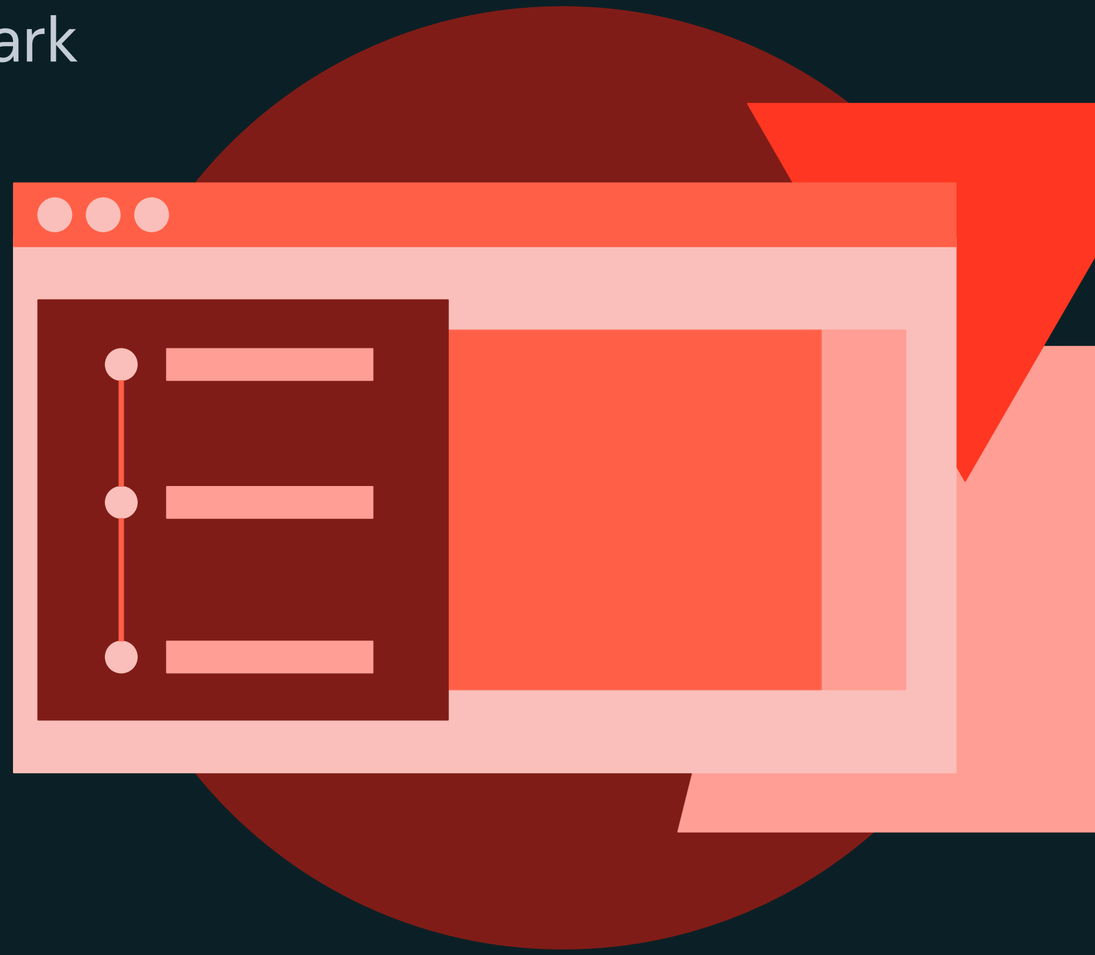
- ACID Compliance
- Data Version Tracking
- Data Governance
- Lineage

databricks

Deploying Machine Learning Models with Spark

DEMONSTRATION

Model Deployment with Spark

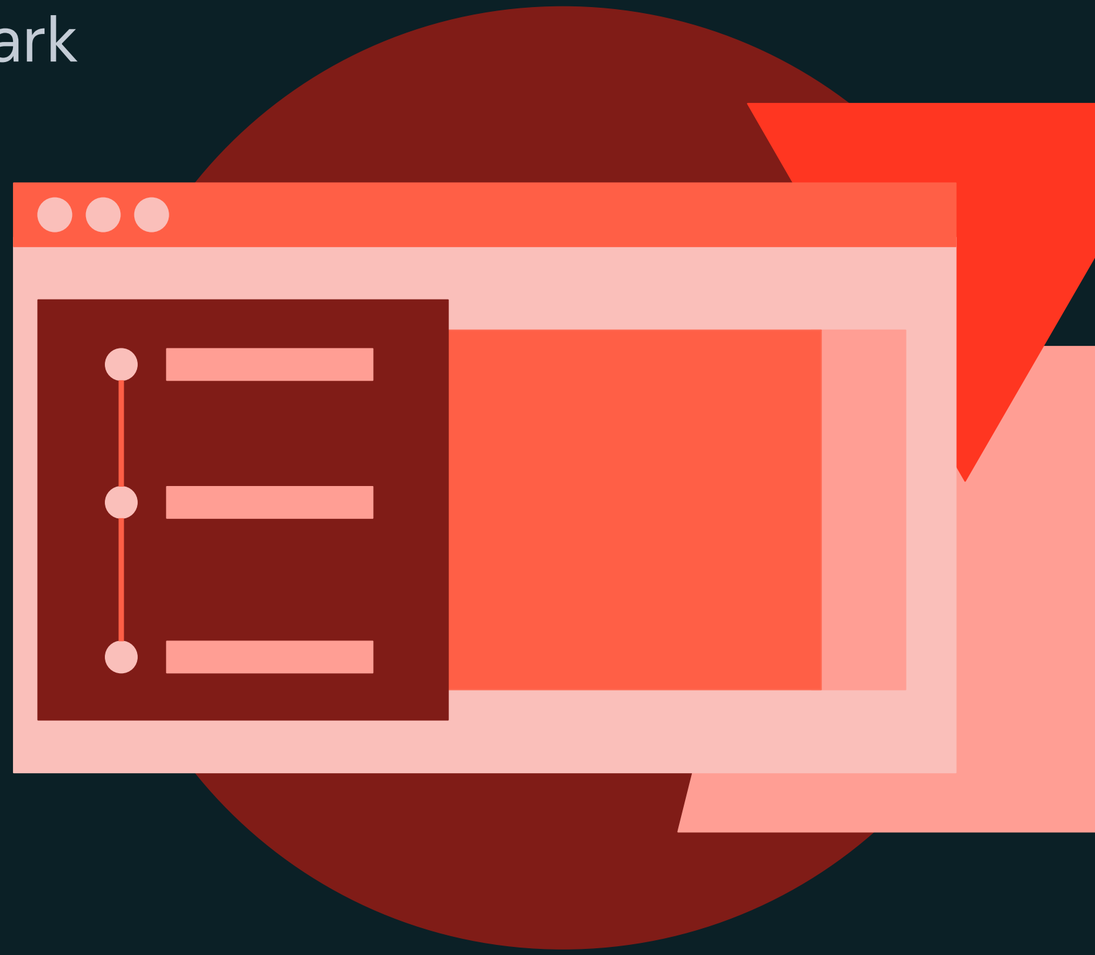


databricks

Deploying Machine Learning Models with Spark

DEMONSTRATION

Optimization Strategies with Spark and Delta Lake

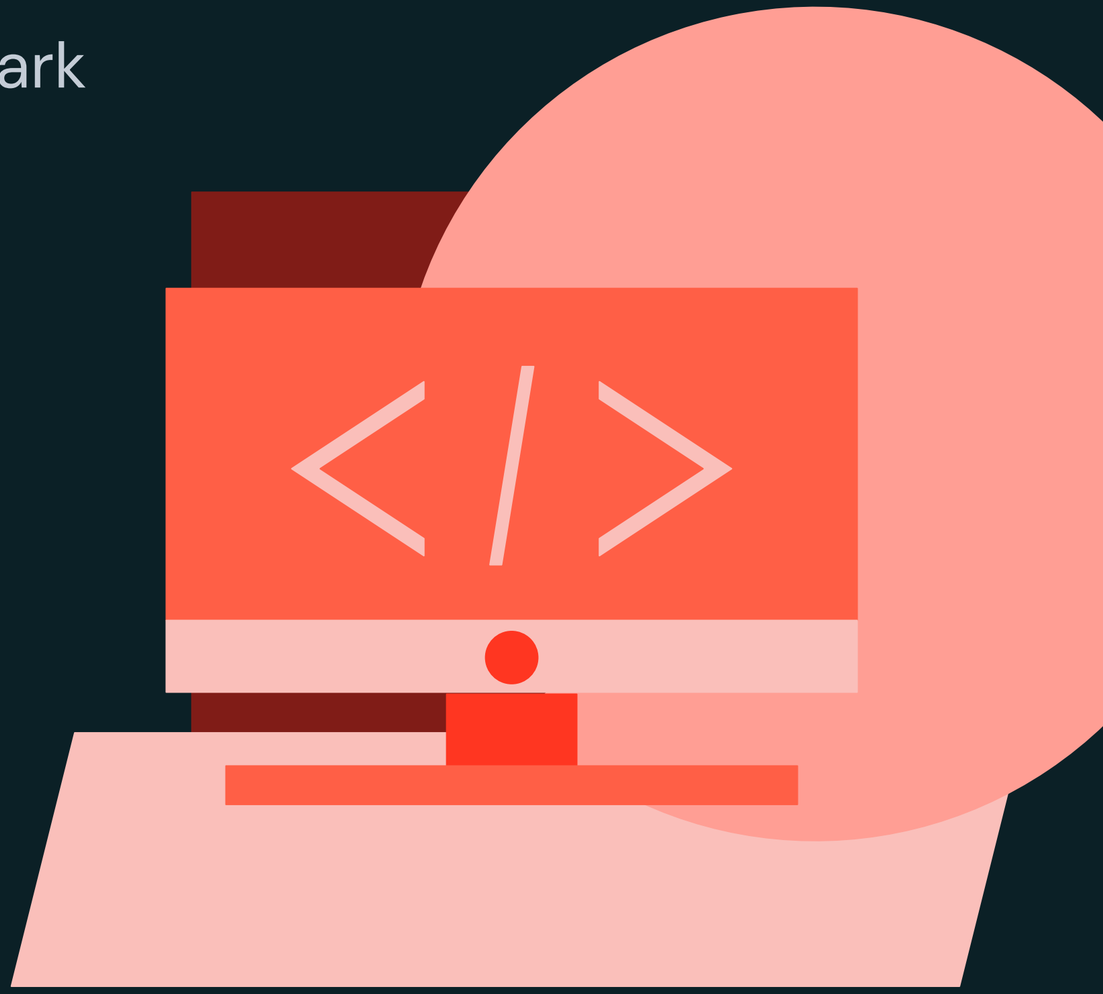


databricks

Deploying Machine Learning Models with Spark

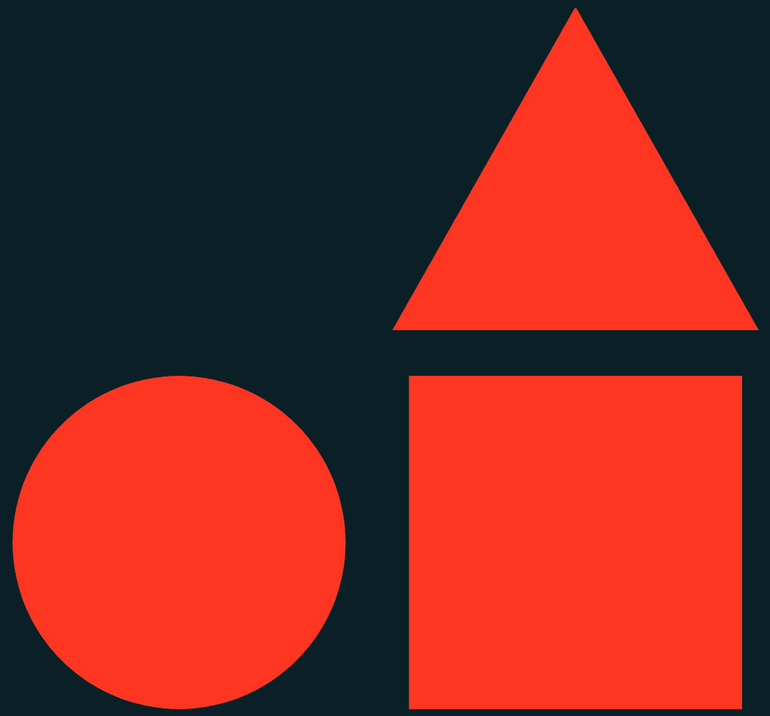
LAB EXERCISE

Model Deployment with Spark



databricks

Pandas on Spark



Machine Learning at Scale

databricks

Pandas on Spark

LECTURE

Scaling with Pandas APIs

Scaling Data with Pandas API on Spark (formerly Koalas Project)

Leverage the Power of Spark with Familiar Pandas API

pandas APACHE Spark™

Pandas

- Popular Data Analysis Library
- Single Machine Focus
- Very performant on small datasets
- Familiar and Powerful API
- Integration with Key Libraries
  - NumPy, Matplotlib, Seaborn, SciPy, Scikit-Learn

Apache Spark

- Distributed Computing Engine
- Scalable and Fast
- Supports various data sources and workloads
- Supports Multiple Languages

Pandas API on Spark *Fills the Gaps*

When your data gets big run existing Pandas code on Spark

- Avoids learning a new framework
- Be More productive
- Maintain single codebase
- Avoid time-consuming to rewrites & testing
- Avoid Confusion between pandas vs Spark API

```
~~from pandas import read_csv~~
from pyspark.pandas import read_csv
pdf = read_csv("data.csv")
```

Pandas DataFrame vs. PySpark DataFrame

	pandas DataFrame	PySpark DataFrame
Column	df['col']	df['col']
Mutability	Mutable	Immutable
Execution	Eagerly	Lazily
Add a column	df['c'] = df['a'] + df['b']	df = df.withColumn('c', df['a'] + df['b'])
Rename columns	df.columns = ['a','b']	df = df.select(df['c1'].alias('a'), df['c2'].alias('b')) df = df.toDF('a', 'b')
Value count	df['col'].value_counts()	df.groupBy(df['col']).count() .orderBy('count', ascending=False)

```python
# Pandas
import pandas as pd

# Perform basic operation
pandas_df['C'] = pandas_df['A'] + pandas_df['B']
print("Pandas DataFrame:\n", pandas_df)
```

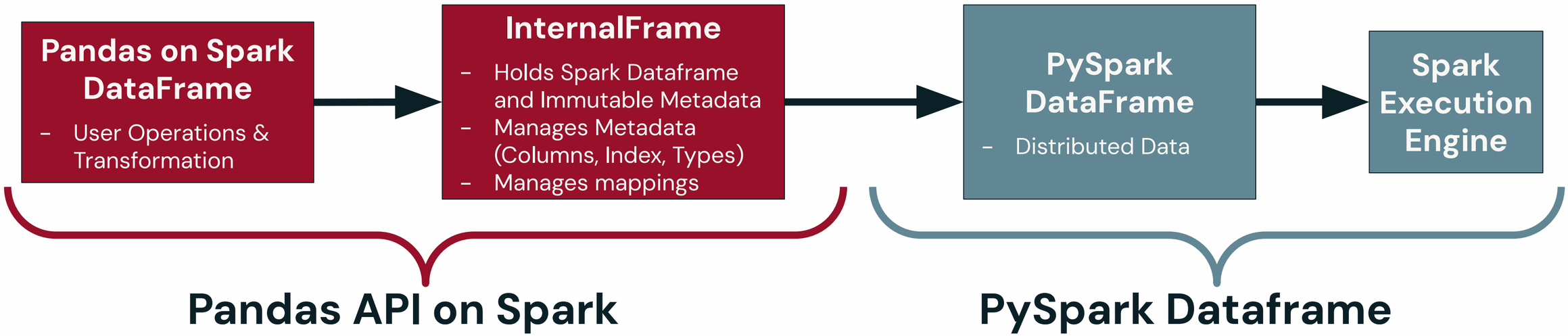
```python
# Apache Spark
from pyspark.sql.functions import col

# Perform same operation
spark_df = spark_df.withColumn('C', spark_df['A']
+ spark_df['B'])
print("Spark DataFrame:")
spark_df.show()
```

```python
# Pandas API on Spark
import pyspark.pandas as ps

# Perform same operation
ps_df['C'] = ps_df['A'] + ps_df['B']
print("Pandas API on Spark
DataFrame:")
print(ps_df)
```

Role of the InternalFrame in PySpark Pandas

Connecting Pandas API with Spark DataFrames

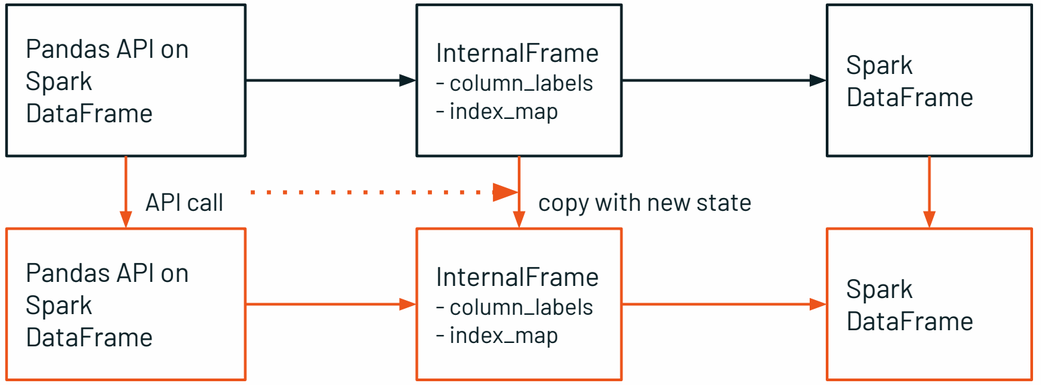
The “Internal Frame” converts between the PySpark Pandas (Pandas API on Spark) and the PySpark dataframes.



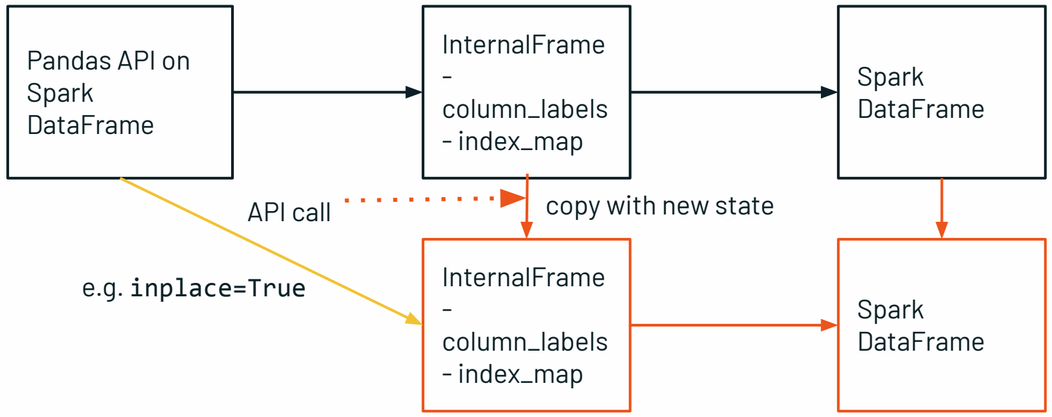
InternalFrame in Pandas API on Spark

If a user ….

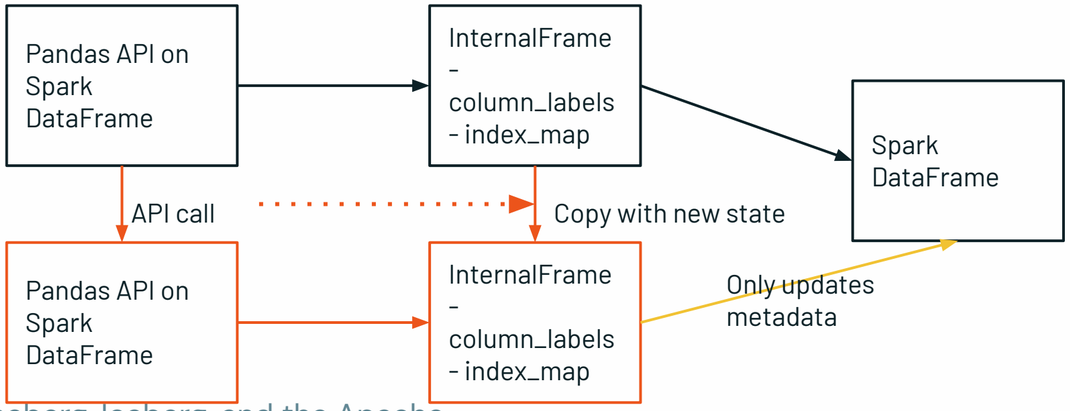
- **Calls an API:** The Pandas API on Spark updates the Spark DataFrame and modifies the metadata stored in the InternalFrame.
- The InternalFrame is either created or copied with the new states (e.g., new columns, indices).
- The operation results in a new DataFrames and InternalFrame.



- **Updates internal state** of Pandas API on Spark DataFrame
- *e.g. psdf.dropna(…., inplace = true)*



- **Updates Metadata Only** *e.g. psdf.set_index(…)*
- Doesn't copy complete Dataframe, just points new InternalFrame at the old Spark DataFrame



Unlocking Data Capabilities

Scalability, Easy Conversion, Integration with SQL

Scalability Beyond a Single Machine

```
1  from pandas import read_csv
2  read_csv("data.csv")
```

⊞ConnectException: Connection re

ConnectException error: This is often
Check your query's memory usage.
Spark tip settings

ConnectException error: This is often caused by an **OOM error** that causes the connection to the Python REPL to be closed. **Check your query memory usage**.

```
1  from pyspark.pandas import read_csv
2  read_csv("data.csv")
```

▸ (3) Spark Jobs

Out[3]:

	Emp ID	Name Prefix	First Name	Middle Initial	Last Nar
0	742048	Drs	Lizeth	R	Mcc

Easy Conversion

Between Pandas and Spark Dataframes

Integration with Pandas Libraries like Matplotlib

```python
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import pyspark.sql.functions as F

# Load a Spark DataFrame from Unity Catalog
sdf = spark.table("catalog.schema.table_name")

# Convert Spark DataFrame to Pandas DataFrame
pdf = sdf.toPandas()

# Plotting with Matplotlib (Spark can't do)
pdf.plot(kind="bar", x="Name", y="Salary")
plt.show()

# Perform a distributed group by aggregation on sdf
result =
sdf.groupBy("Name").agg(F.avg("Salary").alias("AvgSal"))

# Display the result of the aggregation
result.show()

# Convert Pandas DataFrame back to Spark DataFrame
sdf_converted = spark.createDataFrame(pdf)
```

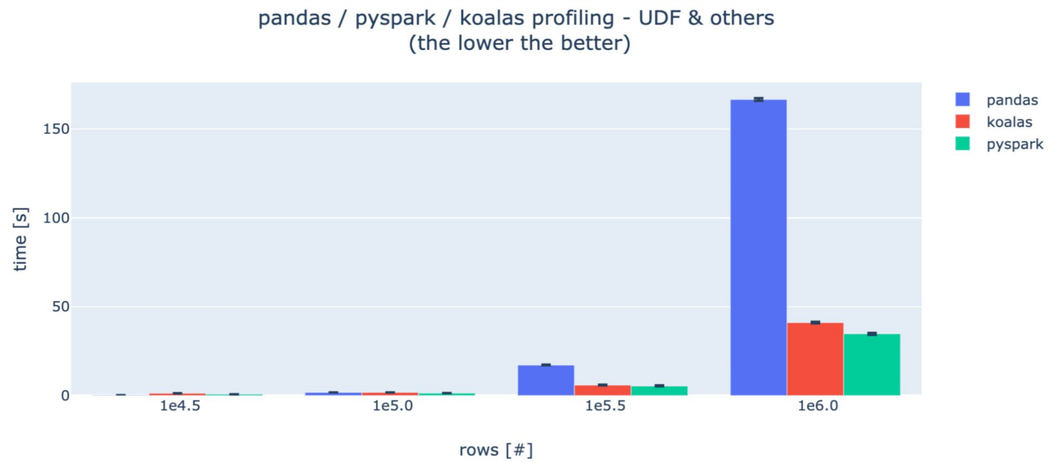
Integration with SQL

```python
# Import necessary library
import pyspark.pandas as ps

# Load a DataFrame from Unity Catalog
psdf = ps.read_table("catalog.schema.table_name")


# Method 1: Direct SQL Query using ps.sql()
direct_result = ps.sql("SELECT Name, Salary FROM {psdf}
WHERE Salary > 6000")

direct_result.show()


# Method 2: SQL Query after registering temporary view
psdf.spark.cache().createOrReplaceTempView("employees")

view_result = spark.sql("SELECT Name, Salary FROM
employees WHERE Salary > 6000")

view_result.show()
```

Performance of Pandas, Spark, and PySpark Pandas

pandas Koalas APACHE Spark™

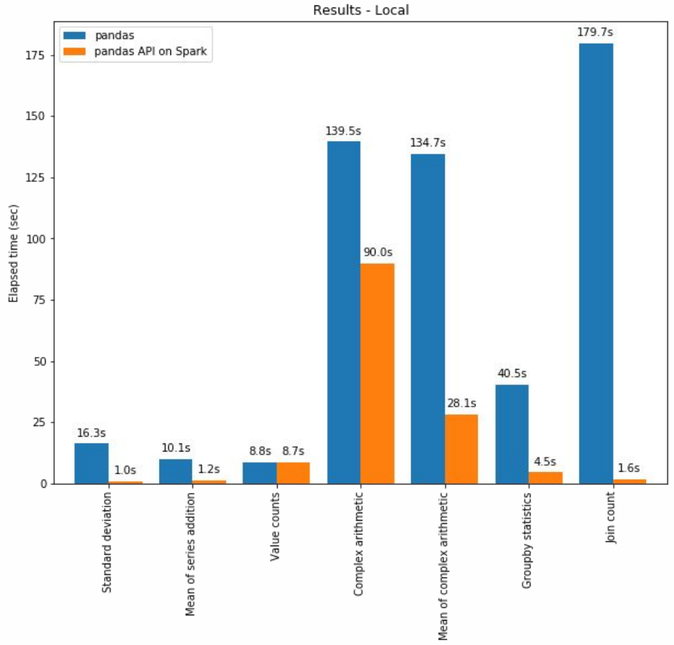
Virgin Hyperloop One reduced processing time from hours to minutes with Koalas (PySpark Pandas)



More than 10X faster with less than 1% code changes

Blog post with Hyperloop One

The pandas API on Spark often outperforms pandas even on a single machine thanks to the optimizations in the Spark engine.



- Both multi-threading and Spark SQL Catalyst Optimizer contribute to the optimized performance.
- E.g. Group by statistics is ~9 times faster

Blog Post: Pandas API on Upcoming Apache Spark™ 3.2

databricks

Pandas on Spark

LECTURE

Pandas UDFs and Function APIs

User-defined Functions (UDFs)

- A UDF is function defined by a user, allowing custom logic to be reused in the user environment
    - Called on one row of Spark DataFrame at a time
- **Performance Considerations:**
    - **Built-in Functions:** Fastest due to Databricks optimizations.
    - **JVM Execution (Scala, Java, Hive):** Faster than Python UDFs.
    - **Pandas UDFs:** Utilize Arrow for efficient data exchange, reducing serialization overhead.
    - **Python UDFs:** Useful for procedural logic, but less efficient for large-scale ETL workloads.
- **UDF Benefit:** Increases ***flexibility*** by allowing custom logic in ***familiar*** languages, reducing the human cost of refactoring code.

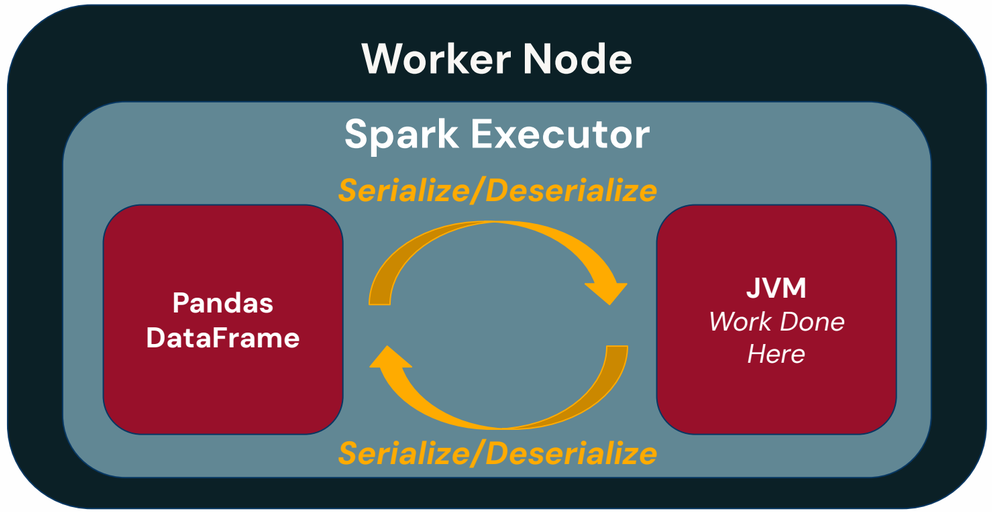
Type	Optimized	Execution environment
Hive UDF	No	JVM
Python UDF	No	Python
Pandas UDF	No	Python (Arrow)
Scala UDF	No	JVM
Spark SQL	Yes	JVM (Photon)
Spark DataFrame	Yes	JVM (Photon)

Uses Arrow to reduce serialization costs

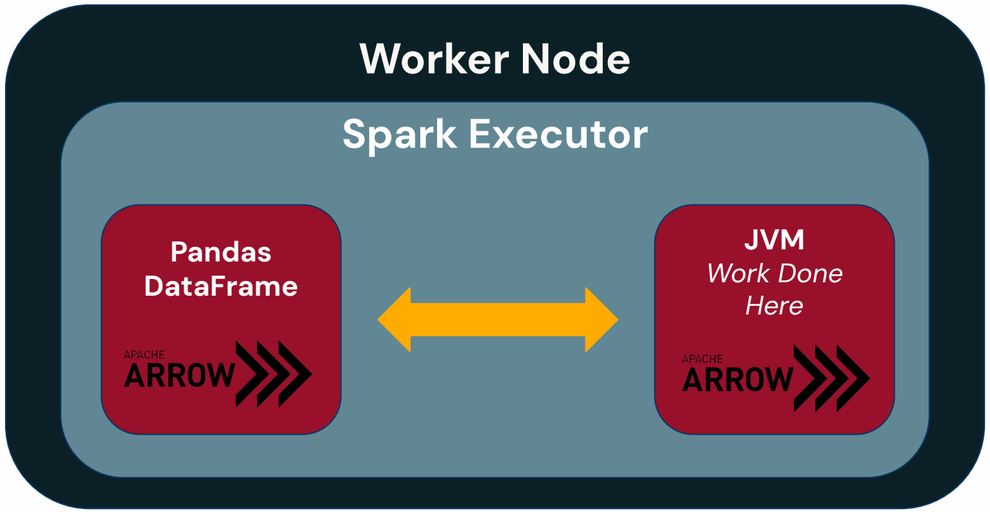
Defining custom logic without serialization penalties

Apache Arrow

Improving Performance in When Working With Pandas UDFs



- Pandas DataFrame is **serialized/deserialized** when passing data **to/from** the **JVM.**
- **Performance Cost:** Serialization overhead due to conversions between Pandas and JVM data formats.

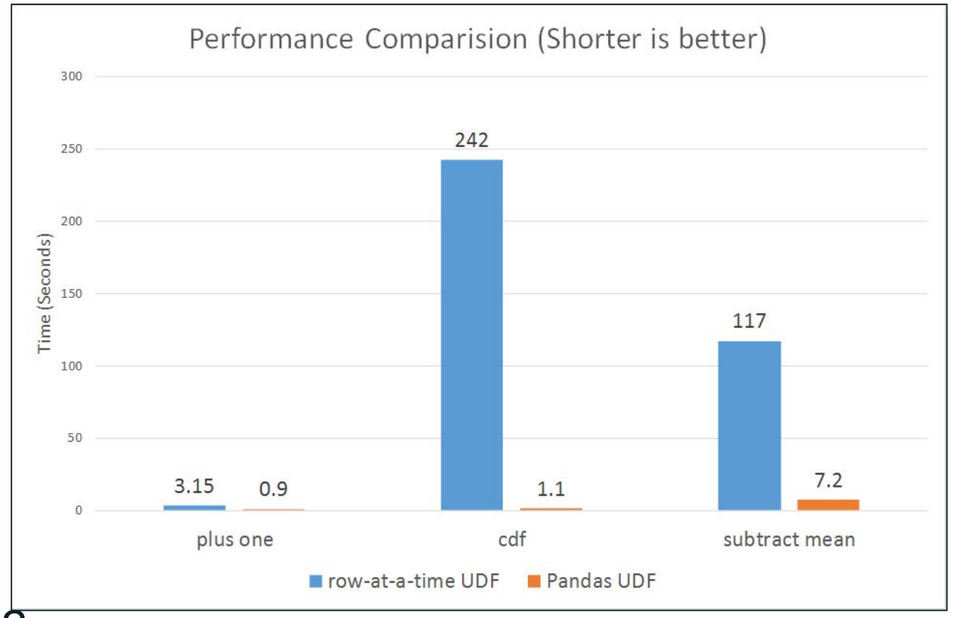


- **Apache Arrow** provides a **language-independent columnar memory format** for Pandas and JVM, eliminating the need for serialization.
- **Performance Boost:** Reduces serialization overhead, speeding up data processing.
- *Libraries are available for C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby, and Rust*

See Enabling Apache Arrow

Pandas UDFs

Scale Out with Vectorized Operations



Achieve up to 100x speedup over traditional row-based Python UDFs with vectorized operations

- **Pandas UDFs:**
  - Leverage Pandas within the function for native Pandas operations
  - Use Apache Arrow for efficient, near-zero overhead data exchange between JVM and Python environments
  - Executed in a vectorized manner

- **Writing a Pandas UDF**
  - Use pandas_udf as a decorator
  - Python type hints required
    - Indicate desired input & output
  - 4 Types of Pandas UDFs

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf( 'long' )
def pandas_plus_one (s: pd.Series) -> pd.Series:
    return s + 1

spark.range(10).select(pandas_plus_one("id")).show()
```

Pandas UDFs

4 Types of Pandas UDFs

Series to Series

- Takes one or more pandas.Series as input, outputs one pandas.Series.
- Input and output have the same number of rows.

Iterator of Series to Iterator of Series

- Takes an iterator of batches instead of a single input batch, Returns an iterator of output batches.
- Same number of rows in the input and output
- Useful for state initialization during execution, e.g., loading model for inferencing.

Iterator of multiple Series to Iterator of Series

- Takes an iterator of multiple pandas.Series inputs, outputs a single pandas.Series.
- Same number of rows in the input and output
- Ideal for stateful operations and data prefetching.

Series to scalar

- Converts one or more pandas.Series into a single scalar value.
- Use with APIs such as select, withColumn, groupBy.agg, and pyspark.sql.Window.

Series: Dataframe Column & Scalar = Single discrete value

4 Types of Pandas UDFs

Examples of 4 Types of Pandas UDFs

Series to Series

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf('double')
def standardize(s: pd.Series) -> pd.Series:
    return (s - s.mean()) / s.std()

df.select(standardize("feature_column")).show()
```

Iterator of Series to Iterator of Series

```
@pandas_udf("double")
def model_inference(iterator: Iterator[pd.Series]) -> Iterator[pd.Series]:
    # Load the pre-trained model
    model = load_model()
    # Apply the model to each batch in the iterator
    for batch in iterator:
        yield pd.Series(model.predict(batch))

# Apply the UDF to a DataFrame column
df.select(model_inference("features")).show()
```

Iterator of multiple Series to Iterator of Series

```
@pandas_udf("double")
def model_inference(iterator: Iterator[Tuple[pd.Series, pd.Series]]) ->
Iterator[pd.Series]:
    model = load_model()
    for feature1, feature2 in iterator:
        combined_features = np.column_stack((feature1, feature2))
        yield pd.Series(model.predict(combined_features))

df.select(model_inference("feature1", "feature2")).show()
```

Series to scalar

```
@pandas_udf("double")
def calculate_accuracy(predictions: pd.Series, labels: pd.Series) ->
float:
    return (predictions == labels).mean()

df.select(calculate_accuracy("predicted", "actual")).show()
```

Series: Dataframe Column & Scalar = Single discrete value

Pandas Function APIs in Spark

Optimizing Table-Level Operations with Pandas

- **Pandas Function APIs:**
  - Leverage Pandas within the function for **native Pandas DataFrame operations**
    - *E.g. Join, Sort, Filter …*
  - **Similar to Pandas UDFs**: also use Apache Arrow and executed in a vectorized manner
  - Input/Output are **DataFrames**, ***not individual Series***.
  - Flexible output length
  - Python type **hints are optional**
  - **Three Types:**
    - Map
    - Grouped map
    - Cogrouped map

```
from typing import Iterator
import pandas as pd

df = spark.createDataFrame([(1, 21), (2, 30)], ("id", "age"))

def pandas_filter(iterator: Iterator[pd.DataFrame]) -> Iterator[pd.DataFrame]:
    for pdf in iterator:
        yield pdf[pdf.id == 1]

df.mapInPandas(pandas_filter, schema=df.schema).show()
```

Pandas Function APIs

Three Types: Grouped Map, Map, Cogrouped Map

Map

- **Operation:** Applies a function to each partition (batch) of the DataFrame.
- **Input:** Whole DataFrame split into partitions.
- **Output:** Transformed DataFrame, typically of the same schema.

Grouped Map

- **Operation:** Applies a function to each group of a single DataFrame, defined by a group key.
- **Input:** DataFrame grouped by one or more columns.
- **Output:** Transformed DataFrame with the same schema as the input.

Cogrouped Map

- **Operation:** Group two DataFrames by a common key and applying the Python function to each group.
- **Input:** Two DataFrames grouped by a shared column.
- **Output:** Combined or transformed DataFrame.

Pandas Function APIs

Examples of Three Types: Grouped Map, Map, Cogrouped Map

Map

```
def normalize_partition(pdf):
    return (pdf - pdf.mean()) /
    pdf.std()

df.mapInPandas(normalize_partition,
schema=df.schema)
```

Normalize features within each partition of a DataFrame for batch model training.

Grouped Map

```
def calculate_group_mean(pdf):
    return pdf.assign(mean_value=
    pdf['feature'].mean())

df.groupby('group_key')
  .applyInPandas
  (calculate_group_mean,
   schema=df.schema)
```

Calculate the mean value of a feature within each group (e.g., customer segments) to use in feature engineering.

Cogrouped Map

```
def align_features(pdf1, pdf2):
    return pdf1.merge(pdf2,
    on='key')


df1.cogroup(df2)
  .applyInPandas(align_features,
  schema="...")
```

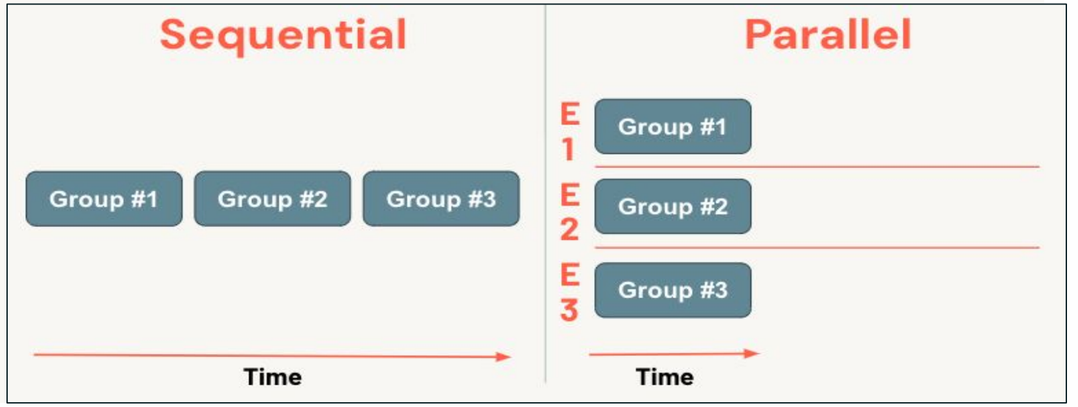
Perform a feature alignment between two datasets grouped by a common key, such as aligning training and validation sets

Parallelizing Group-Specific Model Training

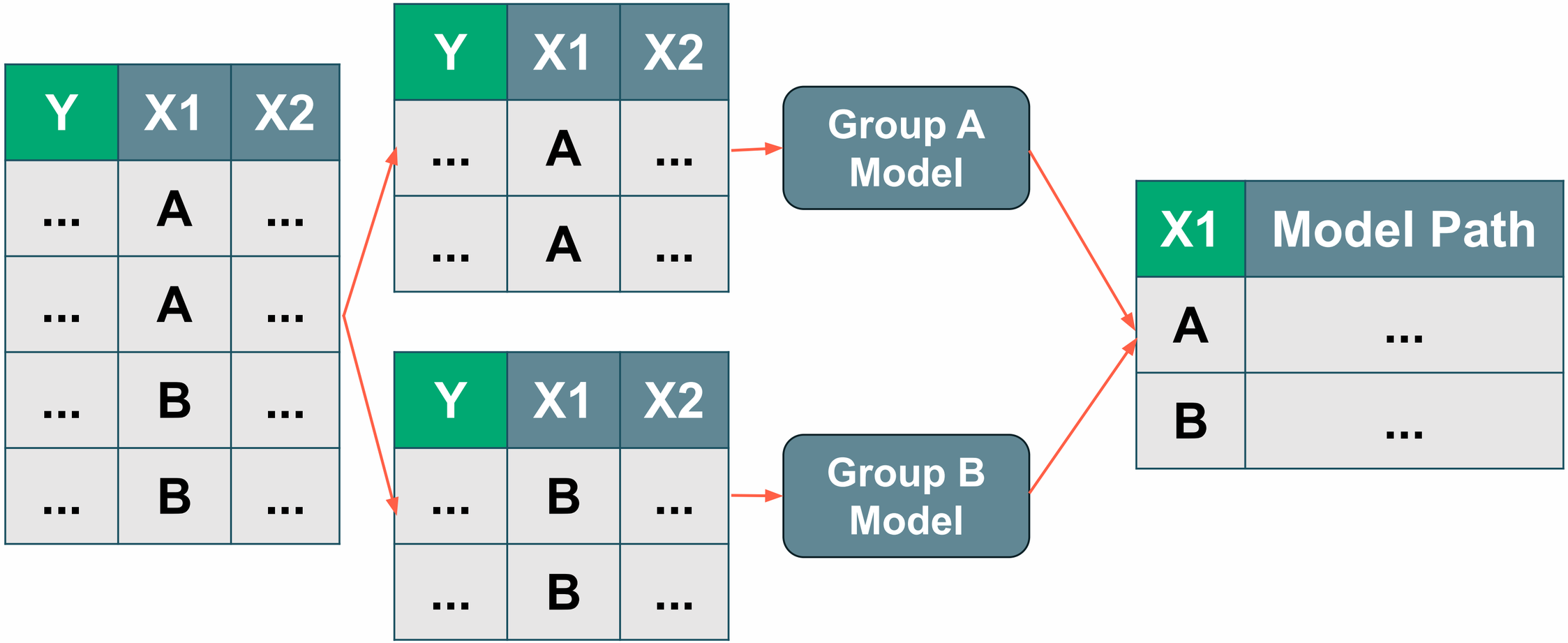
- **Train one model for each group of rows**
- Groups are defined by the values in a given column
- **Examples**
  - **Retail/CPG:** Model for each store/product/customer
  - **Finance:** Model for each stock symbol
  - **Healthcare:** Model for metric in a health-tracking app
  - **IOT Devices:** Model for monitoring or maintaining multiple devices.

Grouped map

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from pyspark.sql.functions import pandas_udf

def train_model(pdf: pd.DataFrame) -> pd.DataFrame:
    model = LinearRegression()
    X = pdf[['feature1', 'feature2']]
    y = pdf['target']
    model.fit(X, y)
    pdf['predictions'] = model.predict(X)
    return pdf

# Group by 'store_id' and apply the model training function
df.groupby('store_id').applyInPandas(train_model,
schema=df.schema)
```



Parallelizing Group-Specific Model Inferencing

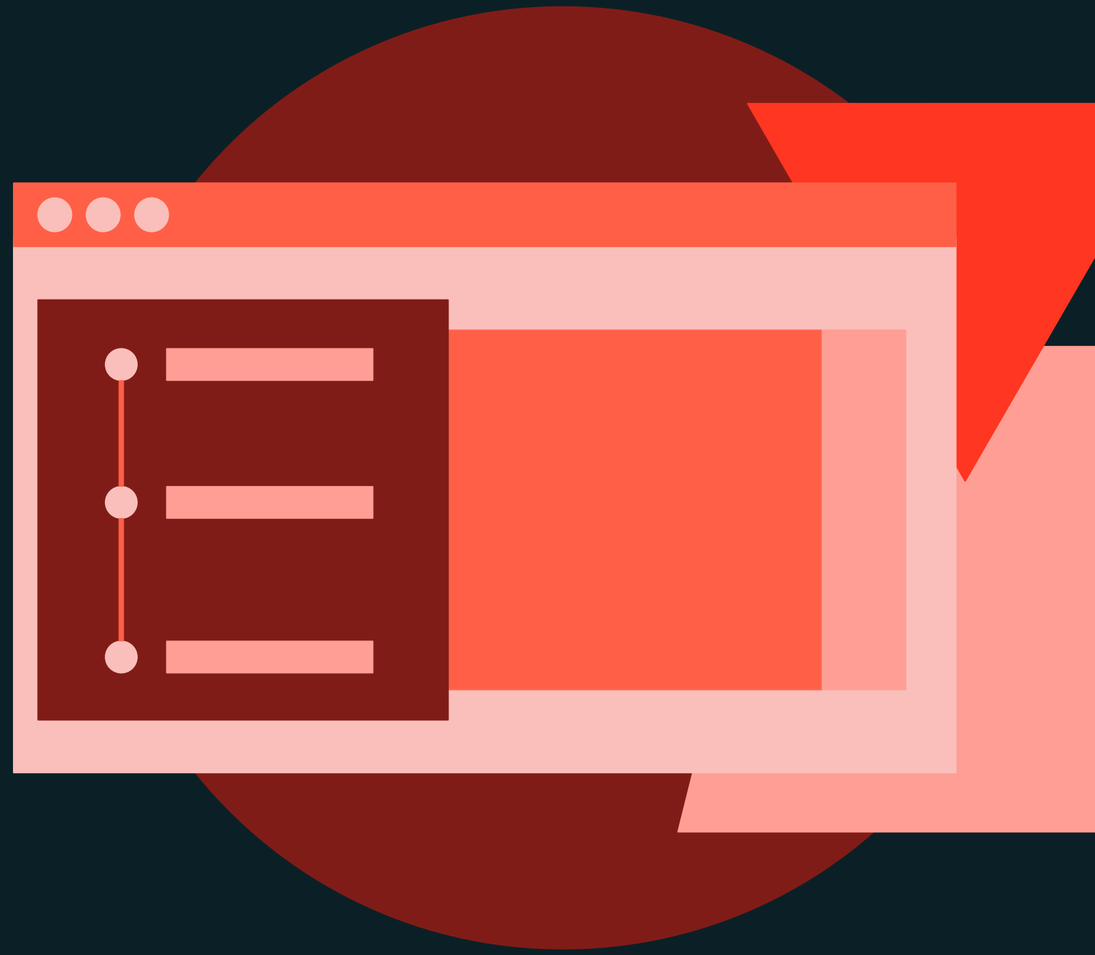


databricks

Pandas on Spark

DEMONSTRATION

Pandas APIs



databricks

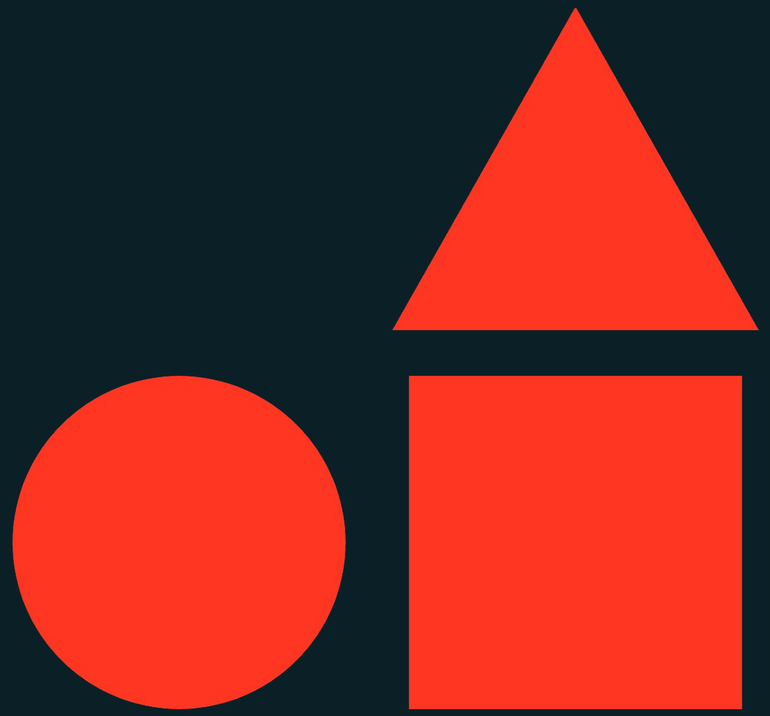
Pandas on Spark

LAB EXERCISE

Pandas APIs

databricks

Recap and CTA



Machine Learning at Scale

Course Learning Objectives Recap

- Understand Spark's Architecture and how it supports ML workloads within Databricks
- Know when it is appropriate to utilize Spark
- Explain and demonstrate how to use Spark ML for data preparation, model training, and model evaluation
- Explain and demonstrate how to use Optuna and Spark for tuning hyperparameters on Databricks
- Explain and demonstrate how to Track and package models with MLflow and Unity Catalog with Spark
- Explain and demonstrate optimization strategies with Spark and Delta Lake
- Explain and demonstrate model deployment and inference scalability with Spark
- Explain and demonstrate how pandas APIs work on Spark

Next Steps

Additional resources for continuing the learning journey

Machine Learning with Databricks

- Validate your data and AI skills on Databricks by earning a Databricks credential
  - Exam information and related training
    - Valid for a period of 2 years
    - Online Proctored
    - It's recommended that the user has at least 1 year of hands-on experience in performing machine learning tasks
    - Exam registration

databricks