SPARK

Maciej Marchwiany, PhD

https://www.linkedin.com/pulse/ai-powered-physics-how-ai-revolutionizing-scientific-discovery-cain-ddbwc

Plan

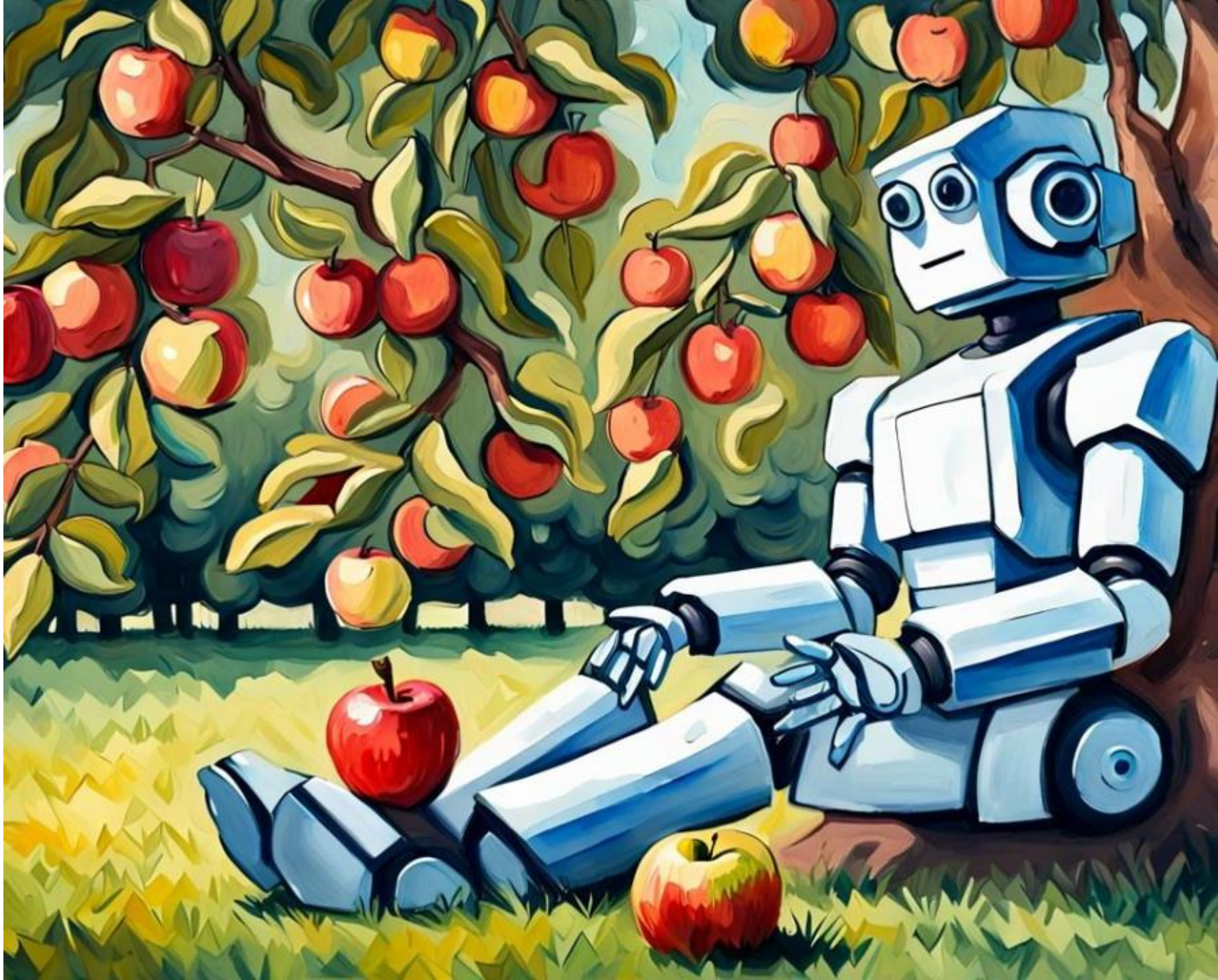
Spark

PySpark

MLlib

Running PySpark

Spark



SPARK

Apache Spark is an open-source unified **analytics engine** for **large-scale data** processing. Spark provides an interface for programming clusters with implicit **data parallelism** and **fault tolerance**. Originally developed at the University of California, Berkeley's AMPLab starting in 2009, in 2013, the Spark codebase was donated to the Apache Software Foundation, which has maintained it since.

http://en.wikipedia.org/wiki/Apache_Spark

Key Features

- **Distributed Computing Engine:**
  Spark operates across clusters of computers, enabling parallel data processing. It breaks down tasks into smaller units that run concurrently, thus managing massive volumes of data.
- **In-Memory Processing:**
  Unlike traditional disk-based systems, Spark leverages RAM to store intermediate results, which speeds up data processing by minimizing slow disk I/O operations.
- **Unified Analytics Platform:**
  Spark is not limited to one type of data processing. It offers a suite of libraries for different workloads, including:
  - **Spark SQL:** For querying structured data using SQL-like syntax.
  - **Spark Streaming:** For processing real-time data streams.
  - **MLlib:** A scalable machine learning library.
  - **GraphX:** For graph processing and analytics.
- **Multi-Language Support:**
  It supports several programming languages such as Python (through PySpark), Scala, Java, and R, making it accessible to a wide range of developers and data scientists.

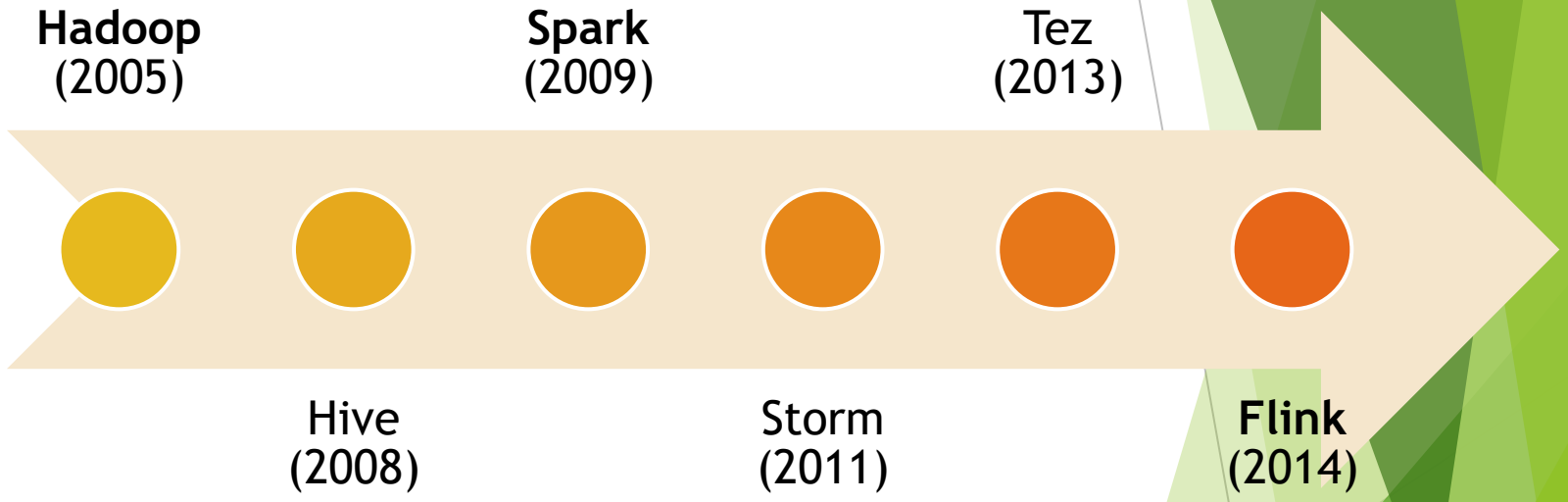
What Can You Use Apache Spark For?

- **Batch Processing:**
  - **Data Transformation:** Process large batches of data for Extract, Transform, Load operations.
  - **Data Aggregation and Summarization:** Quickly compute summaries over large datasets.
- **Real-Time Stream Processing:**
  - **Event Processing:** Analyze data in real time, which is useful for monitoring systems, detecting fraud, or managing live dashboards.
  - **Log Processing:** Continuously process and analyze log data from applications or websites.
- **Interactive Analytics and Data Exploration:**
  - **SQL Queries:** interactively query large datasets using familiar SQL syntax.
  - **Ad-hoc Analysis:** Explore and analyze data interactively, which is especially useful in data science workflows.
- **Machine Learning:**
  - **Building Predictive Models:** develop scalable machine learning models on massive datasets.
  - **Iterative Algorithms:** Efficiently run algorithms that require multiple passes over the data, such as clustering or classification.
- **Graph Processing:**
  - **Network Analysis:** analyzing social networks, recommendation systems, or any dataset that can be represented as a graph.

Why SPARK

- **Speed** – Spark processes data much faster than traditional frameworks like Hadoop MapReduce by using in-memory computation and optimized execution engines.
- **Ease of Use** – It supports multiple programming languages, including Python (PySpark), Java, Scala, and R.
- **Scalability** – Spark can run on clusters of thousands of nodes, making it suitable for large-scale data workloads.
- **Versatile Libraries** – It includes libraries for SQL (Spark SQL), machine learning (MLlib), streaming data (Spark Streaming), and graph processing (GraphX).
- **Compatibility** – Spark can run on various platforms, including Hadoop, Kubernetes, standalone clusters, and cloud services.

A bit of history



Hadoop

What it is:

- A framework for distributed storage (**HDFS - Hadoop Distributed File System**) and batch processing (**MapReduce**).
- Used for large-scale data processing in a **distributed cluster**.

Why it was used:

- Allowed companies to process **petabytes of data** across many machines.
- Open-source and scalable.

Limitations:

- **Slow processing** due to disk-based operations.
- **Not suitable for real-time analytics** (batch-only processing).
- **Complex programming** (writing MapReduce jobs in Java was cumbersome).

Hive

What it is:

- A **data warehousing and SQL-based query engine** built on top of **Apache Hadoop**.
- Used for **batch processing** of large-scale structured data stored in **HDFS**.
- Allows users to query big data using **HiveQL (SQL-like language)** instead of writing complex MapReduce jobs.

Why it was used:

- Enabled **SQL-like querying** on massive datasets without requiring expertise in **Java or MapReduce**.
- Scalable and open-source, making it suitable for **big data analytics**.
- Supported integration with **Hadoop ecosystem tools** like Apache Tez and Spark.

Limitations:

- **Slower performance** due to **disk-based execution**.
- **Batch-only processing** – not designed for **real-time** analytics.
- **High latency** – Queries take **seconds to minutes** to execute, unlike databases with **millisecond response times**.
- **Not optimized for small datasets** – Best suited for **large-scale** queries.

Flink

What it is:

- A **stream processing** engine optimized for real-time analytics.
- Unlike Spark, Flink is **natively streaming** (while Spark Streaming is micro-batch).

Why it was used:

- **Low-latency** processing (better for event-based applications).
- High-performance stateful processing.

Limitations:

- **Less popular than Spark** (fewer libraries, smaller community).
- **Not as good for batch processing** (although it supports it).

HDFS

Hadoop Distributed File System

What it is:

- A **distributed file system** designed to store **large datasets** across multiple machines.
- Uses a **master-slave architecture** with a **NameNode and multiple DataNodes**.
- **Fault-tolerant and scalable**, making it suitable for **big data storage and processing**.

Why it was used:

- Allowed companies to store and manage **petabytes of data** across **commodity hardware**.
- Provided **high availability** by replicating data across multiple nodes.
- **Integrated with Hadoop's processing frameworks** (MapReduce, Hive, Spark, etc.).

Limitations:

- **Not optimized for real-time data access** – Best for **batch processing** rather than low-latency queries.
- **High storage overhead** – Uses **data replication (default: 3 copies)** to ensure fault tolerance.
- **No support for random writes** – Designed for **write-once, read-many** workloads (append-only).

HDFS: Master-Slave Model

NameNode (Master)

- Manages **metadata** (file names, locations, permissions).
- Keeps track of which **DataNodes** store which data blocks.
- Does **not** store actual data.
- Single point of failure (unless High Availability mode is enabled).

DataNodes (Slaves)

- Store actual **data blocks** across multiple nodes.
- Send **heartbeat signals** to the NameNode to confirm they are running.
- Handle **read and write requests** from clients.

Default Replication Factor: Each file is divided into **blocks** (default: **128MB or 256MB**) and replicated **3 times** across different DataNodes for **fault tolerance**.

How Data is Stored in HDFS

1. A client uploads a file to HDFS.
2. File is split into blocks
   (e.g., a 1GB file with 128MB blocks → 8 blocks).
3. NameNode assigns DataNodes to store the blocks.
4. Blocks are replicated across multiple DataNodes
   (default: 3 copies).
5. DataNodes store the blocks and report back to the NameNode.

How Data is Read from HDFS

1. **Client requests a file** from HDFS.
2. **NameNode provides the list of DataNodes** that have the required blocks.
3. **Client directly reads blocks** from the nearest DataNodes (for efficiency).
4. **Data is reassembled** into the original file.

Fault Tolerance in HDFS

- **eplication:** If a DataNode fails, HDFS retrieves a copy from another node.
- **Heartbeat Check:** The NameNode continuously monitors DataNodes. If a node fails, blocks are **re-replicated** on another node.
- **Rack Awareness:** Ensures replicas are spread across different racks to improve reliability.

Problems with HDFS

- HDFS is Not Optimized for Low-Latency, High-Performance I/O
- HDFS Relies on Replication Instead of Parallel I/O
- HDFS is Designed for Commodity Hardware, Not HPC Supercomputers
- HDFS Does Not Support POSIX Compliance
- Different Workload Characteristics:

Feature	HPC Clusters	HDFS (Big Data Clusters)
Workload Type	Computationally intensive (scientific simulations, modeling)	Data-intensive (batch processing, analytics)
Data Access Pattern	Low-latency, high-speed I/O (random access)	High-throughput, sequential read/write
File Size	Small to large files	Very large files (TBs/PBs)
Processing Model	MPI-based (Message Passing Interface)	Distributed batch processing (MapReduce, Spark)

Unimaginably slow

MapReduce

MapReduce is a **distributed computing framework** used for processing **large-scale datasets** in parallel across multiple machines. It was originally developed by **Google** and later implemented in **Apache Hadoop**. It follows a **divide-and-conquer approach**, breaking tasks into two main phases: **Map and Reduce**.

Steps:

1. **Map Phase (Splitting & Processing)**
   - The input data is **split into chunks** (usually stored in **HDFS**).
   - The **Mapper** function processes each chunk in parallel and outputs **key-value pairs**.
2. **Shuffle & Sort (Data Grouping)**
   - The framework groups all **similar keys together** from different Mappers.
3. **Reduce Phase (Aggregation & Final Processing)**
   - The **Reducer** function processes the grouped data to **compute final results**.

MapReduce

INTPUT:
Dear Bear River
Car Car River
Deer Car Bear

MapReduce is a **distributed computing framework** used for processing **large-scale datasets** in parallel across multiple machines. It was originally developed by **Google** and later implemented in **Apache Hadoop**. It follows a **divide-and-conquer approach**, breaking tasks into two main phases: **Map and Reduce**.

Steps:

1. **Map Phase (Splitting & Processing)**
   - Input data is **split into chunks** (usually stored in **HDFS**).
   - **Mapper** function processes each chunk in parallel and outputs **key-value pairs**.

```
Mapper 1 -> <Dear, 1>, <Bear, 1>, <River, 1>
Mapper 2 -> <Car, 1>, <Car, 1>, <River, 1>
Mapper 3 -> <Deer, 1>, <Car, 1>, <Bear, 1>
```

2. **Shuffle & Sort (Data Grouping)**
   - The framework groups all **similar key**nt Mappers.

```
<Bear, [1, 1]>
<Car, [1, 1, 1]>
<Dear, [1]>
<Deer, [1]>
<River, [1, 1]>
```

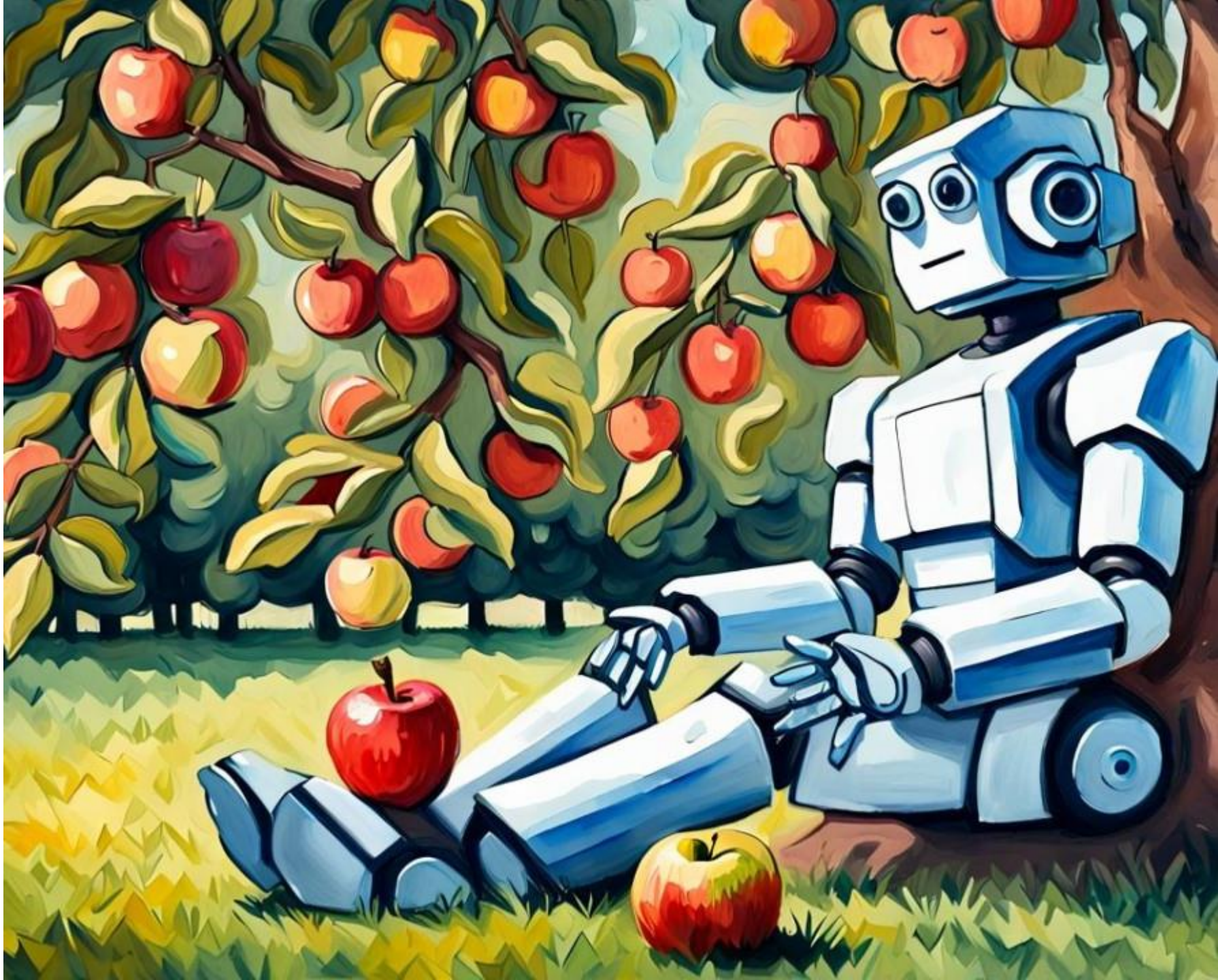
3. **Reduce Phase (Aggregation & Fin**
   - The **Reducer** function processes the grouped data to **compute final results**.

```
<Bear, 2>
<Car, 3>
<Dear, 1>
<Deer, 1>
<River, 2>
```

Why do not use Mapreduce

- **Slow Performance (Especially for Iterative Jobs)**
  MapReduce processes data in **two stages**: the **Map phase** and the **Reduce phase**. Between each stage, data must be written to disk, which can be **slow**. Many **iterative algorithms** (ML algorithms) require multiple passes over the same data.
- **No Built-In Caching**
  In MapReduce, every time a new job is executed, the data has to be read from disk, processed, and written back out.
- **Complex Programming Model**
  Writing MapReduce jobs, especially in **Java**, can be complex and tedious.
- **Lack of Real-Time Processing**
  MapReduce was designed primarily for **batch processing**.
- **Limited Data Operations**
  MapReduce is designed for relatively simple operations like counting, summing, or filtering.
- **Scaling Challenges**
  While **MapReduce** could scale well with the addition of more nodes to a cluster, its **disk-based operations** and the two-phase job structure often led to **inefficiencies** when handling very large datasets.

PySpark



PySpark

PySpark is the **Python API** for **Apache Spark**, allowing users to harness Spark's powerful data processing capabilities using Python. It provides an easy-to-use interface for **big data processing, machine learning, and real-time analytics** without needing to write code in Java or Scala.

How PySpark Works

When you write Python code in PySpark:

1. The PySpark API **translates** Python commands into **Spark jobs**.
2. These jobs run **in parallel** on a distributed cluster.
3. Spark's **execution engine** handles the scheduling and optimization.

Why Use PySpark?

1. Leverages Spark's Speed & Power
   - PySpark runs on Apache Spark, making it much faster than traditional Python-based data tools (like Pandas) for handling large datasets.
   - It supports parallel processing and in-memory computation, reducing disk read/write operations (unlike MapReduce).
2. Simplifies Big Data Processing in Python
   - Python developers can process petabytes of data without learning Java or Scala.
   - It integrates well with NumPy, Pandas, and ML frameworks like TensorFlow & Scikit-learn.
3. Works on Clusters & Cloud
   - PySpark can run on distributed clusters (e.g., Hadoop YARN, Kubernetes) or on the cloud (e.g., AWS EMR, Databricks).
4. Supports SQL, Machine Learning, and Streaming
   - Provides Spark SQL for querying big data using SQL.
   - Has built-in MLlib for scalable machine learning.
   - Supports real-time data streaming with Spark Streaming.

PySpark Components

- Resilient Distributed Datasets (RDDs)
  - The core data structure in PySpark.
  - Immutable & distributed across multiple machines.
  - Supports parallel operations like `map()`, `filter()`, `reduce()`.
- DataFrames
  - High-level abstraction over RDDs (like Pandas DataFrames).
  - Uses Spark SQL engine for fast, optimized queries.
  - Supports SQL-like operations (`select()`, `groupBy()`, `join()`).
- Spark SQL
  - Allows querying big data using SQL.
  - Works with Hive, Parquet, JSON, CSV, and other formats.
- MLlib (Machine Learning Library)
  - Provides scalable machine learning algorithms.
- Spark Streaming
  - Enables real-time data processing.
  - Works with Kafka, Flume, and live log data.

PySpark Architecture

Driver Program

- The **Driver Program** is the main entry point for a PySpark application.
- It contains the **SparkContext** (or **SparkSession**) and is responsible for:
  - Creating RDDs/DataFrames.
  - Defining transformations and actions.
  - Distributing tasks across worker nodes.
  - Collecting the final results.

Cluster Manager

- The **Cluster Manager** is responsible for **resource management** and **job scheduling**.
- It decides how to **distribute tasks across worker nodes**.
- PySpark can work with different types of cluster managers:

Executors (Worker Nodes)

- **Executors** are processes running on **worker nodes** that perform the actual computation.
- Each worker node runs one or more executors, which:

PySpark Architecture

- Distributing tasks across worker nodes.
- Collecting the final results.

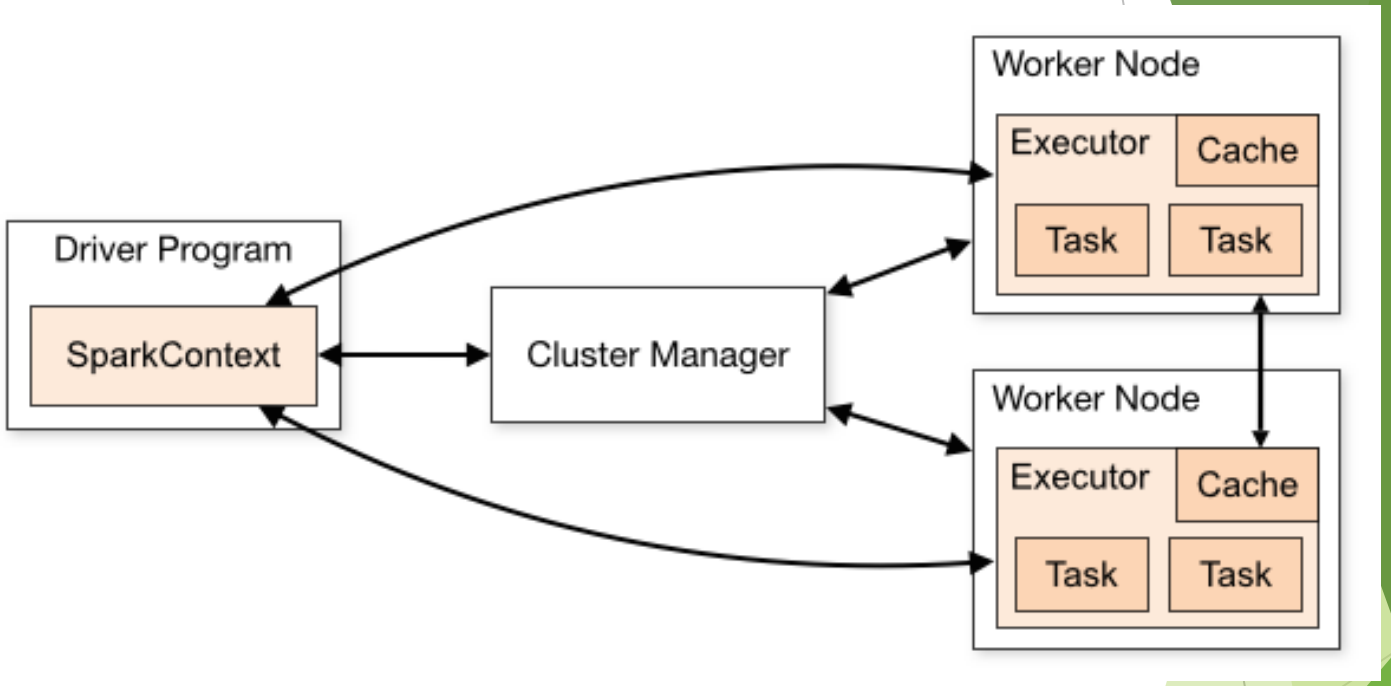
Cluster Manager

- The **Cluster Manager** is responsible for **resource management** and **job scheduling**.
- It decides how to **distribute tasks across worker nodes**.
- PySpark can work with different types of cluster managers:

Executors (Worker Nodes)

- **Executors** are processes running on **worker nodes** that perform the actual computation.
- Each worker node runs one or more executors, which:
  - Execute tasks assigned by the driver.
  - Store intermediate data in memory (to speed up performance).
  - Communicate with the driver.

PySpark Architecture



RDD

Resilient Distributed Datasets (RDDs)

- **RDDs** are the **core data structure** in PySpark.
- They allow for **fault-tolerant**, **parallel processing** of large datasets.
- RDDs are immutable and distributed across multiple worker nodes.

```
data = [1, 2, 3, 4, 5]
rdd = spark.sparkContext.parallelize(data)
```

DAG

DAG (Directed Acyclic Graph) Scheduler

- When a PySpark job is submitted, it creates a **Directed Acyclic Graph (DAG).**
- The DAG **divides the job into smaller tasks and stages.**
- Spark **optimizes the execution** by **minimizing data shuffling**

How PySpark Executes a Job

1. User submits a PySpark application
   - The Driver Program initializes a SparkSession and loads data.
2. DAG (Directed Acyclic Graph) is created
   - PySpark analyzes the transformations and creates a logical execution plan.
   - The DAG Scheduler breaks this into stages.
3. Task Scheduling & Resource Allocation
   - The Cluster Manager assigns resources (CPU, memory) to Executors.
   - The Task Scheduler distributes tasks to worker nodes.
4. Parallel Execution by Executors
   - Worker nodes execute their assigned tasks in parallel.
   - Intermediate results are stored in memory (to speed up processing).
   - Data shuffling happens when required (e.g., during groupBy, join operations).
5. Final Aggregation & Result Collection
   - The final computed results are sent back to the Driver Program.
   - The user gets the output (e.g., DataFrame results, saved file, etc.)

RDD

- Creating from a List

```
data = [1, 2, 3, 4, 5]
rdd = spark.sparkContext.parallelize(data)
```

- Applying Transformations

```
# Squaring each elementsquared_rdd =
rdd.map(lambda x: x ** 2)
```

- Filtering

```
even_rdd = rdd.filter(lambda x: x % 2 == 0)
```

- Reducing

```
sum_rdd = rdd.reduce(lambda x, y: x + y)
```

DataFrames

- Creating from a List

```
from pyspark.sql import Row
# Creating Data
data = [Row(name="Alice", age=25), Row(name="Bob", age=30)]
# Creating DataFrame
df = spark.createDataFrame(data)
```

- Reading a CSV file

```
df= spark.read.csv("data.csv",header=True,inferSchema=True)
```

- Selecting and Filtering Data

```
# Select specific columns
df.select("name", "age").show()
# Filter rows where age > 30
df.filter(df.age > 30).show()
```

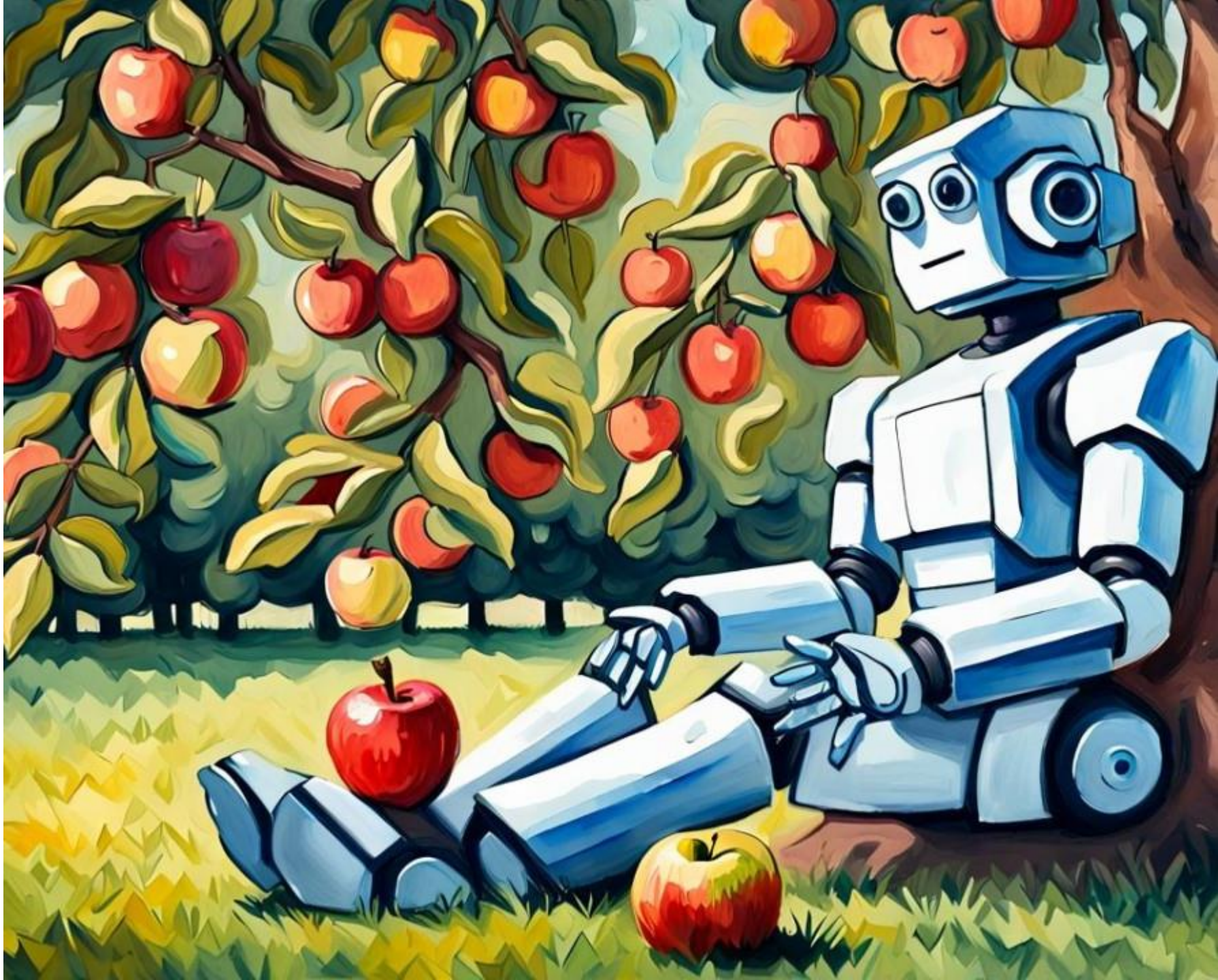
- Add and drop a column

```
# Add a new column from an existing
df.withColumn("NEW",col("age")* -1)
# Dropp a column
df.drop("NEW")
```

Spark SQL

```
# Create temporary table
df.createOrReplaceTempView("PERSON_DATA")

# Run SQL query
df2 = spark.sql("SELECT * from PERSON_DATA")
df2.printSchema()

# Using groupby
groupDF = spark.sql("SELECT gender, count(*)
                from PERSON_DATA group by gender")
```

MLlib



MLlib

MLlib (Machine Learning Library) is Apache Spark's built-in **distributed** machine learning framework. It provides scalable and high-performance ML algorithms for **classification, regression, clustering, recommendation, feature engineering, and more.**

Correlation

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import Correlation

data = [(Vectors.sparse(4, [(0, 1.0), (3, -2.0)]),),
        (Vectors.dense([4.0, 5.0, 0.0, 3.0]),),
        (Vectors.dense([6.0, 7.0, 0.0, 8.0]),),
        (Vectors.sparse(4, [(0, 9.0), (3, 1.0)]),)]
df = spark.createDataFrame(data, ["features"])

# Correlation matrix
r1 = Correlation.corr(df, "features")

# Correlation
r2 = Correlation.corr(df, "features", "spearman")
```

Summar

```
from pyspark.ml.stat import Summarizer
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors

df = sparc.parallelize(
       [Row(weight=1.0, features=Vectors.dense(1.0, 1.0, 1.0)),
        Row(weight=0.0, features=Vectors.dense(1.0, 2.0, 3.0))]).toDF()

# create summarizer for multiple metrics "mean" and "count"
summarizer = Summarizer.metrics("mean", "count")

# compute statistics for multiple metrics without weight
df.select(summarizer.summary(df.features)).show(truncate=False)

# compute statistics for single metric "mean" with weight
df.select(Summarizer.mean(df.features, df.weight)).show(truncate=False)

# compute statistics for single metric "mean" without weight
df.select(Summarizer.mean(df.features)).show(truncate=False)
```

Features

- TF-IDF
- Word2Vec
- Tokenizer
- StopWordsRemover
- CountVectorize
- n-gram
- PCA
- PolynomialExpansion
- OneHotEncoder
- Normalizer
- StandardScaler
- MinMaxScaler
- MaxAbsScaler
- …

https://spark.apache.org/docs/latest/ml-features.html

Machine Learning algorithms

Classification

- Logistic regression
- Decision tree classifier
- Random forest classifier
- Gradient-boosted tree classifier
- Multilayer perceptron classifier
- Linear Support Vector Machine
- Naive Bayes
- …

Regression

- Linear regression
- Generalized linear regression
- Decision tree regression
- Random forest regression
- Gradient-boosted tree regression
- …

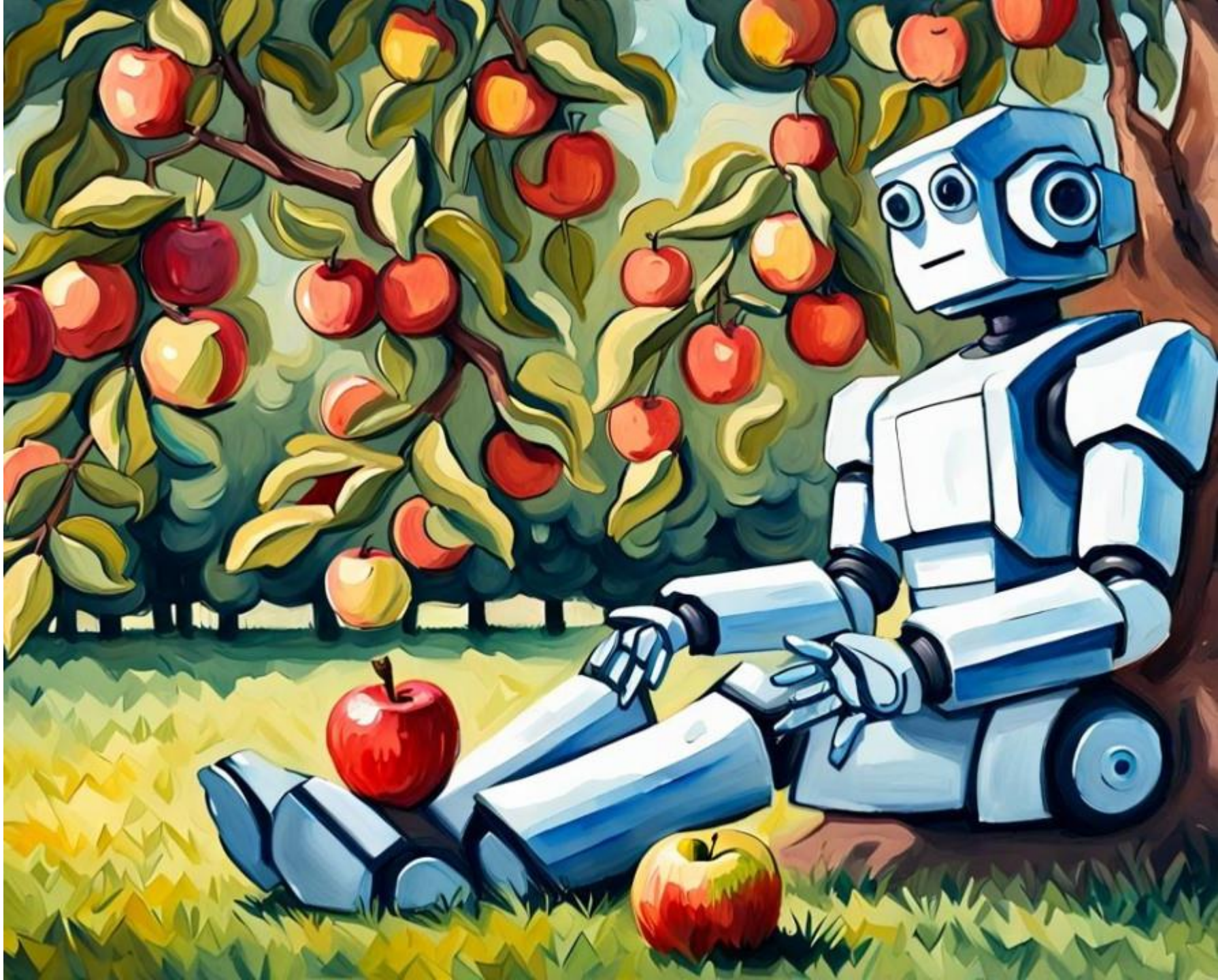
Clustering

- K-means
- Latent Dirichlet allocation (LDA)
- Gaussian Mixture Model (GMM)
- …

Linear Regression

```
from pyspark.ml.regression import LinearRegression

# Load training data
data = spark.read.csv("data.csv")

# Split the data into training and test sets
(trainingData, testData) = data.randomSplit([0.7, 0.3])

# Model definition
lr = LinearRegression()

# Fit the model
lrModel = lr.fit(trainingData)

# Summarize the model over the training set
trainingSummary = lrModel.summary
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)

# Make predictions
predictions = model.transform(testData)
```

Running PySpark



PySpark Session

A **PySpark Session** is the **entry point** to work with **Spark DataFrames, RDDs, SQL, and MLlib**. It **manages the Spark environment**, including:

- **Creating DataFrames**
- **Executing queries**
- **Managing configurations**
- **Interacting with Spark's cluster**Types of Code Profiling

Creating a PySpark Session

```
from pyspark.sql import SparkSession

# Create a Spark Session
spark = SparkSession.builder.appName("MyApp").getOrCreate()
```

Configuring a Spark Session

```
spark = SparkSession.builder \
    .appName("ConfigExample") \
    .master("local[2]") \
    .config("spark.executor.memory", "2g") \
    .getOrCreate()
```

Starting a Spark cluster

1. Set Spark environment variables: *SPARK_LOG_DIR* and SPARK_WORKER_DIR
2. Start Spark-master proces
3. Set Spark masters url
4. Prepare script for starting Spark-worker
5. Start Spark-workers on each node

Now you can Spark.

Starting a Spark cluster

1. Set Spark environment variables:
   SPARK_LOG_DIR and SPARK_WORKER_DIR

```
export SPARK_IDENT_STRING=$SLURM_JOBID
export SPARK_LOG_DIR="/tmp/$USER/$SPARK_IDENT_STRING/logs"
export SPARK_WORKER_DIR="/tmp/$USER//$SPARK_IDENT_STRING/work"
```

2. Start Spark-master proces
3. Set Spark masters url
4. Prepare script for starting Spark-worker
5. Start Spark-workers on each node

Now you can Spark.

Starting a Spark cluster

1. Set Spark environment variables: *SPARK_LOG_DIR* and SPARK_WORKER_DIR
2. Start Spark-master proces

```
start-master.sh
```

3. Set Spark masters url
4. Prepare script for starting Spark-worker
5. Start Spark-workers on each node

Now you can Spark.

Starting a Spark cluster

1. Set Spark environment variables: *SPARK_LOG_DIR* and SPARK_WORKER_DIR
2. Start Spark-master proces
3. Set Spark masters url

```
sleep 10
MASTER_URL=$(grep "Starting Spark master at" ${SPARK_LOG_DIR}/* | rev | cut -d " " -f 1 |rev)
```

4. Prepare script for starting Spark-worker
5. Start Spark-workers on each node

Now you can Spark.

Starting a Spark cluste

1. Set Spark environment variables: *SPARK_LOG_DIR* and SPARK_WORKER_DIR
2. Start Spark-master proces
3. Set Spark masters url
4. Prepare script for starting Spark-worker

```
cat <<EOF > $SPARK_WORKER_DIR/script_$SPARK_IDENT_STRING
   export SPARK_IDENT_STRING=$SPARK_IDENT_STRING
   export SPARK_LOG_DIR=$SPARK_LOG_DIR
   export SPARK_WORKER_DIR=$SPARK_WORKER_DIR
   export MASTER_URL=$MASTER_URL
   start-worker.sh \$MASTER_URL
EOF
```

5. Start Spark-workers on each node

Now you can Spark.

Starting a Spark cluster

1. Set Spark environment variables: *SPARK_LOG_DIR* and SPARK_WORKER_DIR
2. Start Spark-master proces
3. Set Spark masters url
4. Prepare script for starting Spark-worker
5. Start Spark-workers on each node

```
nodes=$(scontrol show hostnames $SLURM_JOB_NODELIST)
for node in $nodes; do
      ssh $node source $SPARK_WORKER_DIR/script_$SPARK_IDENT_STRING
 done
```

Now you can Spark.

Starting a Spark cluster

1. Set Spark environment variables: *SPARK_LOG_DIR* and SPARK_WORKER_DIR
2. Start Spark-master proces
3. Set Spark masters url
4. Prepare script for starting Spark-worker
5. Start Spark-workers on each node

Now you can Spark.

```
spark-submit --master ${MASTER_URL} $SPARK_HOME/examples/src/main/python/pi.py 100
```

```
#!/bin/bash -l
#SBATCH -J test
#SBATCH -o test.output
#SBATCH -e test.output
#SBATCH -t 1:00:00
#SBATCH -N 2
#SBATCH -n 2
#SBATCH --mem=2g

#Set Spark environment variables
export SPARK_IDENT_STRING=$SLURM_JOBID
export SPARK_LOG_DIR="/tmp/$USER/$SPARK_IDENT_STRING/logs„
export SPARK_WORKER_DIR="/tmp/$USER/$SPARK_IDENT_STRING/work„

#Create Sparks tmp folders
mkdir -p $SPARK_LOG_DIR
mkdir -p $SPARK_WORKER_DIR

#Start Spark master process:
start-master.sh

#set Spark masters url
#Wait for writing log
sleep 10
MASTER_URL=$(grep "Starting Spark master at" ${SPARK_LOG_DIR}/* | rev | cut -d " " -f 1 |rev)
```

```
#Set Spark masters url
#Wait for writing log
sleep 10
MASTER_URL=$(grep "Starting Spark master at" ${SPARK_LOG_DIR}/* | rev | cut -d " " -f 1 |rev)

#Prepare script for starting Spark-worker:
cat <<EOF > $SPARK_WORKER_DIR/script_$SPARK_IDENT_STRING
   export SPARK_IDENT_STRING=$SPARK_IDENT_STRING
   export SPARK_LOG_DIR=$SPARK_LOG_DIR
   export SPARK_WORKER_DIR=$SPARK_WORKER_DIR
   export MASTER_URL=$MASTER_URL
   start-worker.sh \$MASTER_URL
EOF

#Start Spark-workers on each node
#Get list of all available nodes
nodes=$(scontrol show hostnames $SLURM_JOB_NODELIST)

#Login (ssh) to each node and start Spark worker
for node in $nodes; do
      ssh $node source $SPARK_WORKER_DIR/script_$SPARK_IDENT_STRING
done

#Wait for synchronization
sleep 1

#Run calculation
spark-submit --master ${MASTER_URL} $SPARK_HOME/examples/src/main/python/pi.py 100
```

Spark in Jupyter

- Set JupyterLab for Spark

```
export PYSPARK_DRIVER_PYTHON_OPTS='notebook --no-browser --port=8889 --ip=0.0.0.0'
export PYSPARK_DRIVER_PYTHON='jupyter'
```

- Start pyspark

```
pyspark 2>&1
```

Warnings

- You need to have Spark installed before installing PySpark.
- You need to have the correct Java version.
- You need to be able to ssh to nodes

- `start-master.sh` and `start-worker.sh` are included with **Apache Spark**