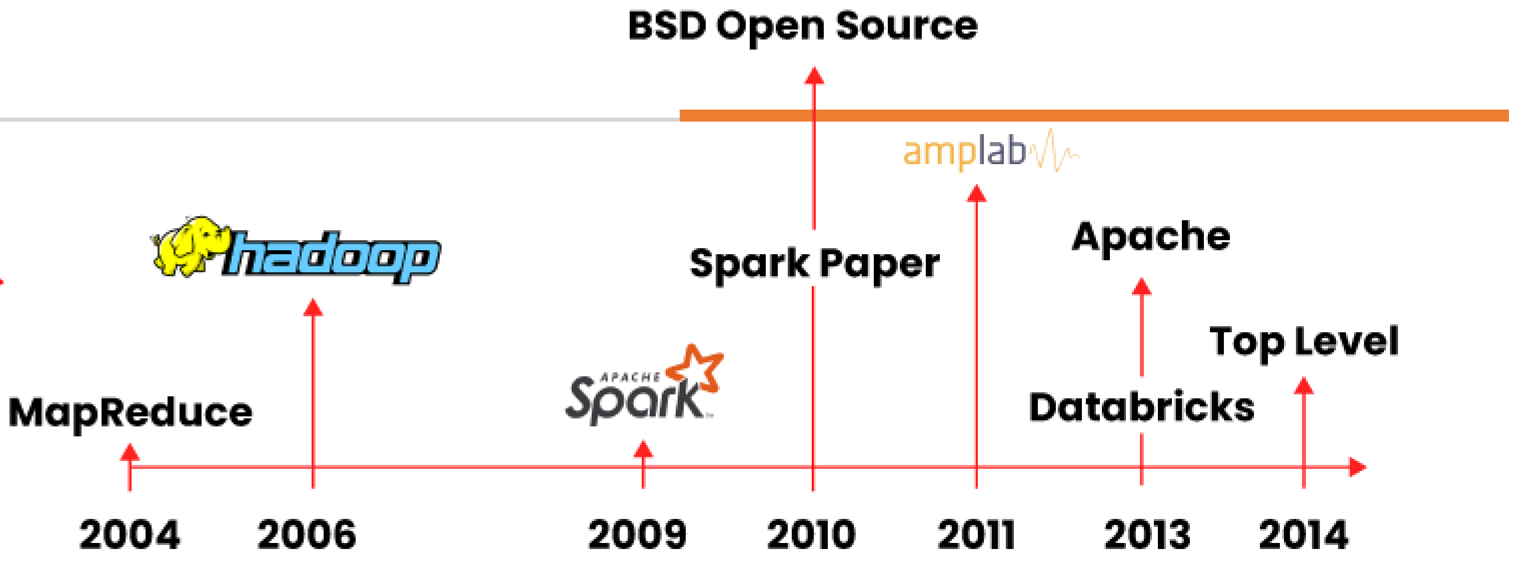
PySpark

History of Spark



PySpark Overview

What is PySpark?

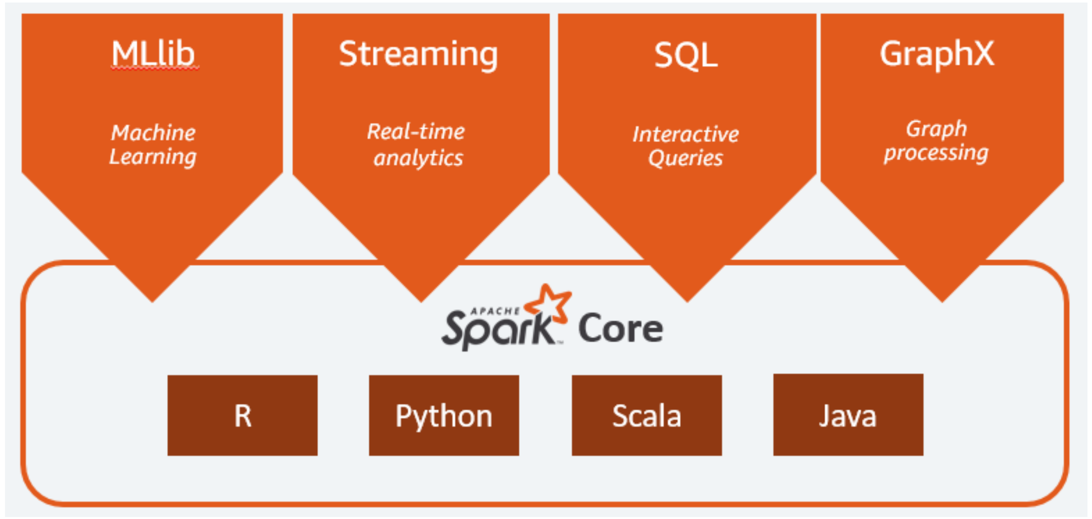
PySpark is the Python API for Apache Spark. It enables you to perform real-time, large-scale data processing in a distributed environment using Python. It also provides a PySpark shell for interactively analyzing your data.

What is PySpark?

PySpark combines Python’s learnability and ease of use with the power of Apache Spark to enable processing and analysis of data at any size for everyone familiar with Python.

What is PySpark?

PySpark supports all of Spark's features such as Spark SQL, DataFrames, Structured Streaming, Machine Learning (MLlib) and Spark Core.



Who uses it?

trivago

Walmart

amazon

PySpark is very well used in Data Science and Machine Learning community as there are many widely used data science libraries written in Python including NumPy, TensorFlow. PySpark brings robust and cost-effective ways to run machine learning applications on billions and trillions of data on distributed clusters 100 times faster than the traditional python applications.

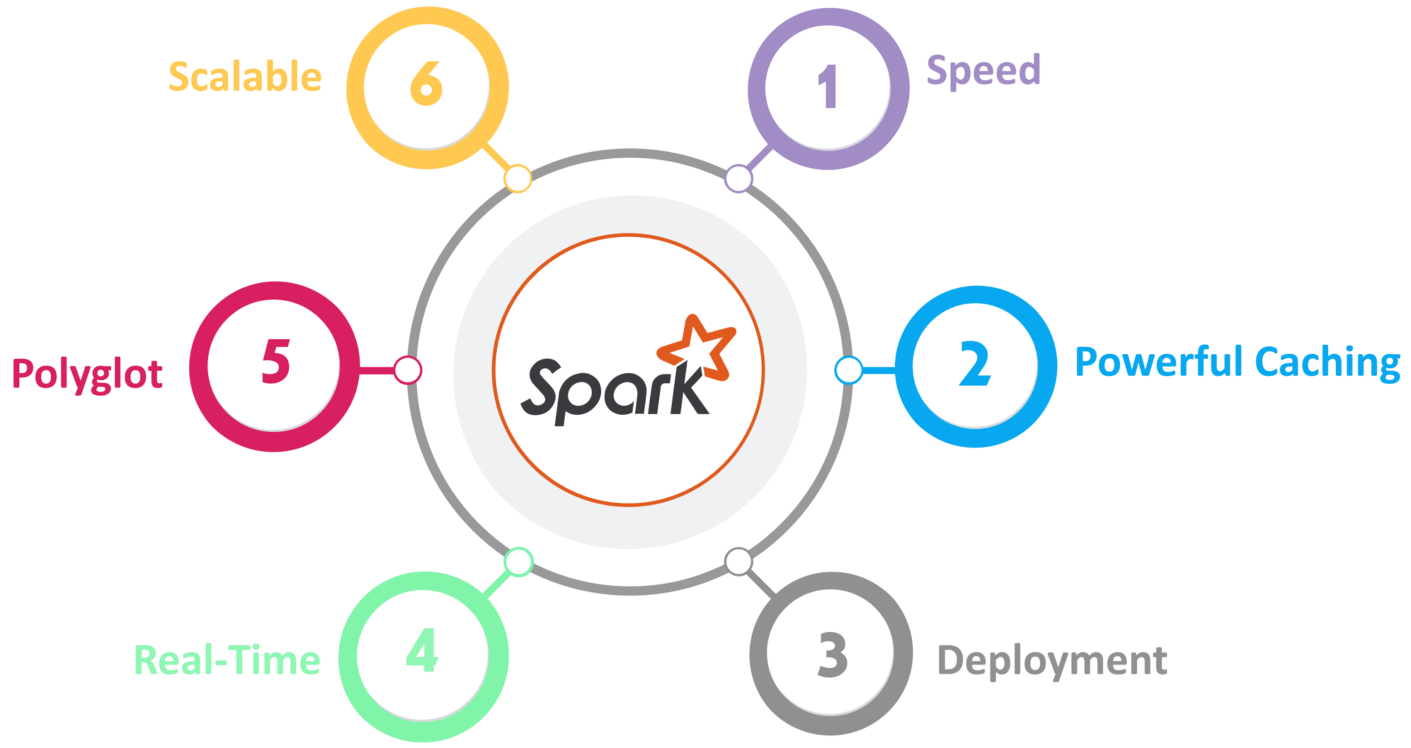
PySpark has been used by many organizations like Amazon, Walmart, Trivago, and many more. PySpark also used in different sectors.

Other Company using Spark

- **Yahoo!** uses Apache Spark for its Machine Learning capabilities to personalize its news and web pages and also for target advertising. They use Spark with Python to find out what kind of news users are interested in reading and categorizing the news stories to find out what kind of users would be interested in reading each category of news.
- **TripAdvisor** uses Apache Spark to provide advice to millions of travelers by comparing hundreds of websites to find the best hotel prices for its customers. The time taken to read and process the reviews of the hotels in a readable format is done with the help of Apache Spark.
- One of the world's largest e-commerce platforms, **Alibaba**, runs some of the largest Apache Spark jobs in the world in order to analyze hundreds of petabytes of data on its e-commerce platform.

Why using PySpark?

- Efficient processing
- Big data analysis
- Fault tolerance



Difference between PySpark and Python

	PySpark	**Python**
1.	PySpark is easy to write and also very easy to develop parallel programming.	Python is a cross-platform programming language, and one can easily handle it.
2.	One does not have proper and efficient tools for Scala implementation.	As python is a very productive language, one can easily handle data in an efficient way.
3.	It provides the algorithm which is already implemented so that one can easily integrate it.	As python language is flexible, one can easily do the analysis of data.
4.	It is a memory computation.	It uses internal memory and nonobjective memory as well.
5.	It only provides R-related and data science-related libraries.	It supports R programming-related libraries with data science, machine learning, etc libraries too.
6.	It allows distribution processing.	It allows to implementation a single thread.
7.	It can process the data in real-time.	It can also process data in real-time with huge amounts.
8.	Before implementation, one requires to have Spark and Python fundamental knowledge.	Before implementation, one must know the fundamentals of any programming language.

Pandas vs PySpark..!

- ***PySpark*** is a library for working with large datasets in a distributed computing environment, while ***pandas*** is a library for working with smaller, tabular datasets on a single machine.
- ***PySpark*** is built on top of the Apache Spark framework and uses the Resilient Distributed Datasets (RDD) data structure, while ***pandas*** uses the DataFrame data structure.
- ***PySpark*** is designed to handle data processing tasks that are not feasible with ***pandas*** due to memory constraints, such as iterative algorithms and machine learning on large datasets.
- ***PySpark*** allows for parallel processing of data, while ***pandas*** does not.
- ***PySpark*** can read data from a variety of sources, including Hadoop Distributed File System (HDFS), Amazon S3, and local file systems, while ***pandas*** is limited to reading data from local file systems.
- ***PySpark*** can be integrated with other big data tools like Hadoop and Hive, while ***pandas*** is not.
- ***PySpark*** is written in Scala, and runs on the Java Virtual Machine (JVM), while ***pandas*** is written in Python.
- ***PySpark*** has a steeper learning curve than pandas, due to the additional concepts and technologies involved (e.g. distributed computing, RDDs, Spark SQL, Spark Streaming, etc.).

How to decide which library to use PySpark vs Pandas

The decision of whether to use PySpark or pandas depends on the size and complexity of the dataset and the specific task you want to perform.

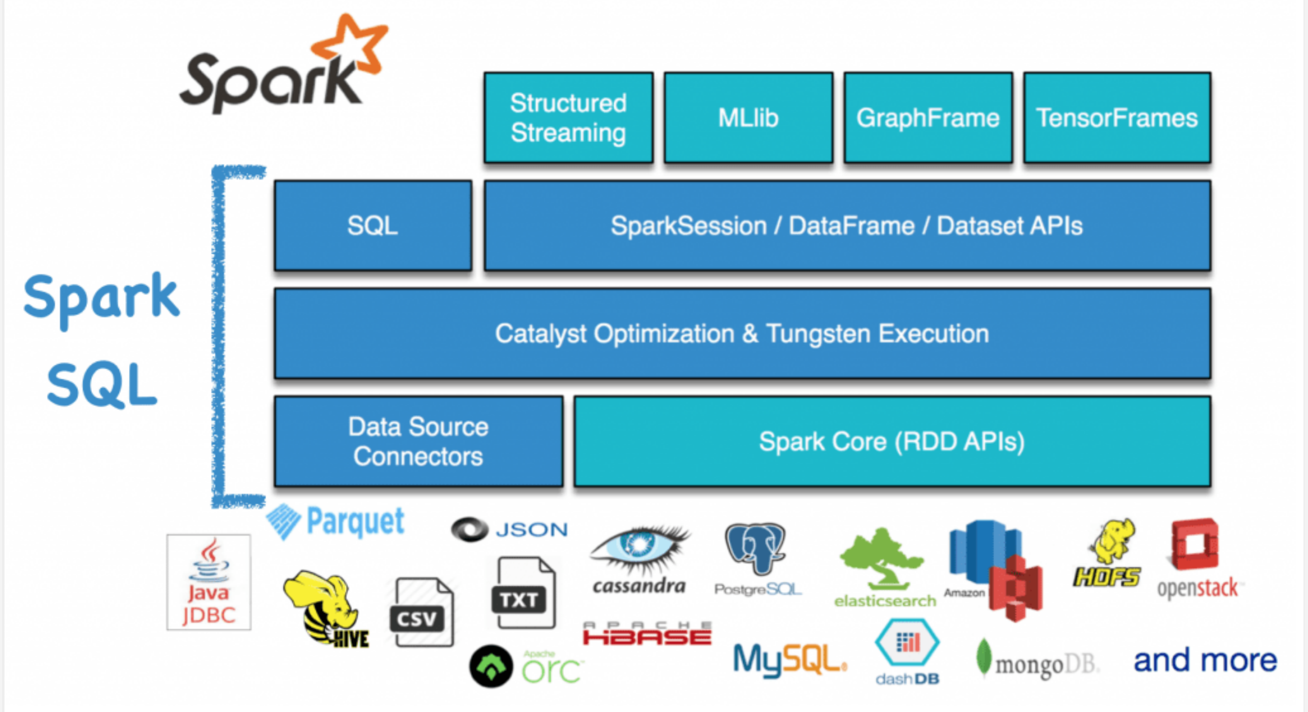
- **Size of the dataset:** *PySpark* is designed to handle large datasets that are not feasible to work with on a single machine using *pandas*. If you have a dataset that is too large to fit in memory, or if you need to perform iterative or distributed computations, *PySpark* is the better choice.
- **Complexity of the task:** *PySpark* is a powerful tool for big data processing and allows you to perform a wide range of data processing tasks, such as machine learning, graph processing, and stream processing. If you need to perform any of these tasks, *PySpark* is the better choice.
- **Learning Curve:** *PySpark* has a steeper learning curve than *pandas*, as it requires knowledge of distributed computing, RDDs, and Spark SQL. If you are new to big data processing and want to get started quickly, pandas may be the better choice.
- **Resources available:** *PySpark* requires a cluster or distributed system to run, so you will need access to the appropriate infrastructure and resources. If you do not have access to these resources, then *pandas* is a good choice.

In summary, use ***PySpark*** for large datasets and complex tasks that are not feasible with ***pandas***, and use ***pandas*** for small datasets and simple tasks that can be handled on a single machine.

PySpark concept

1.DATAFRAME
2.PANDAS API ON SPARK
3.Structure Steaming
4.Machine Learning
5.RDD

Spark SQL and DataFrames



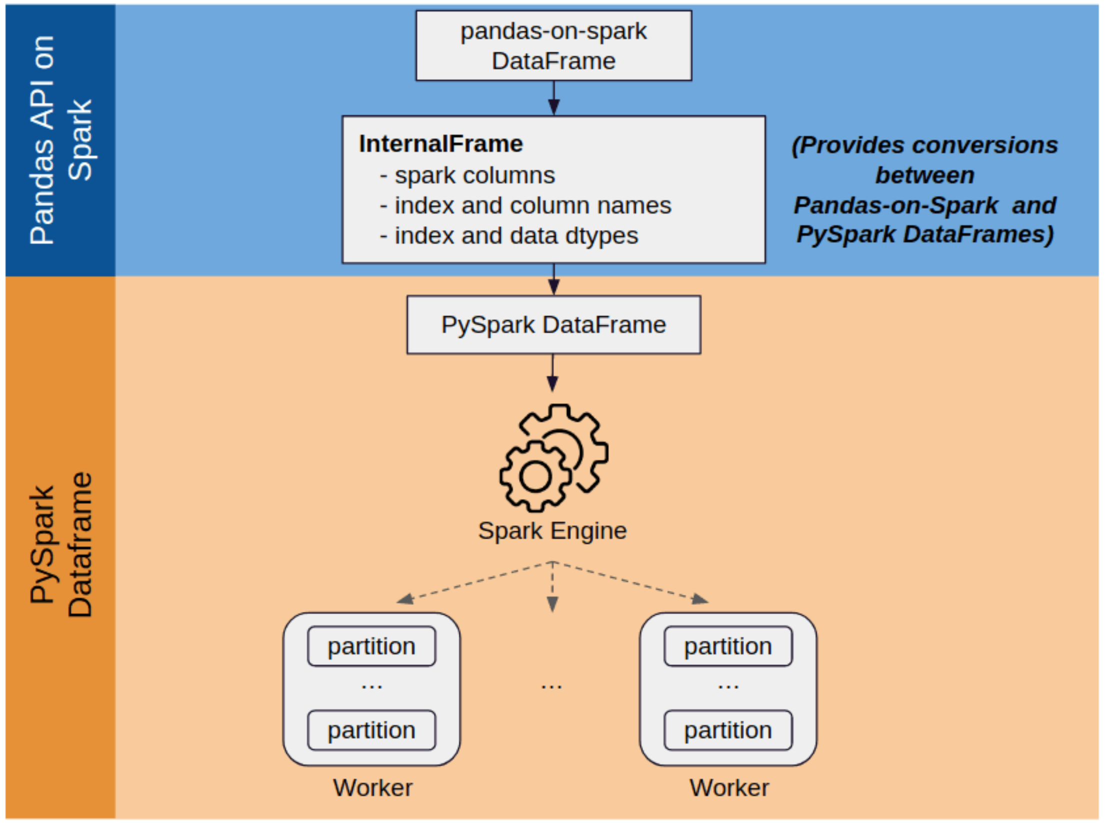
Spark SQL is Apache Spark's module for working with structured data. It allows you to seamlessly mix SQL queries with Spark programs. With PySpark DataFrames you can efficiently read, write, transform, and analyze data using Python and SQL. Whether you use Python or SQL, the same underlying execution engine is used so you will always leverage the full power of Spark.

https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/index.html

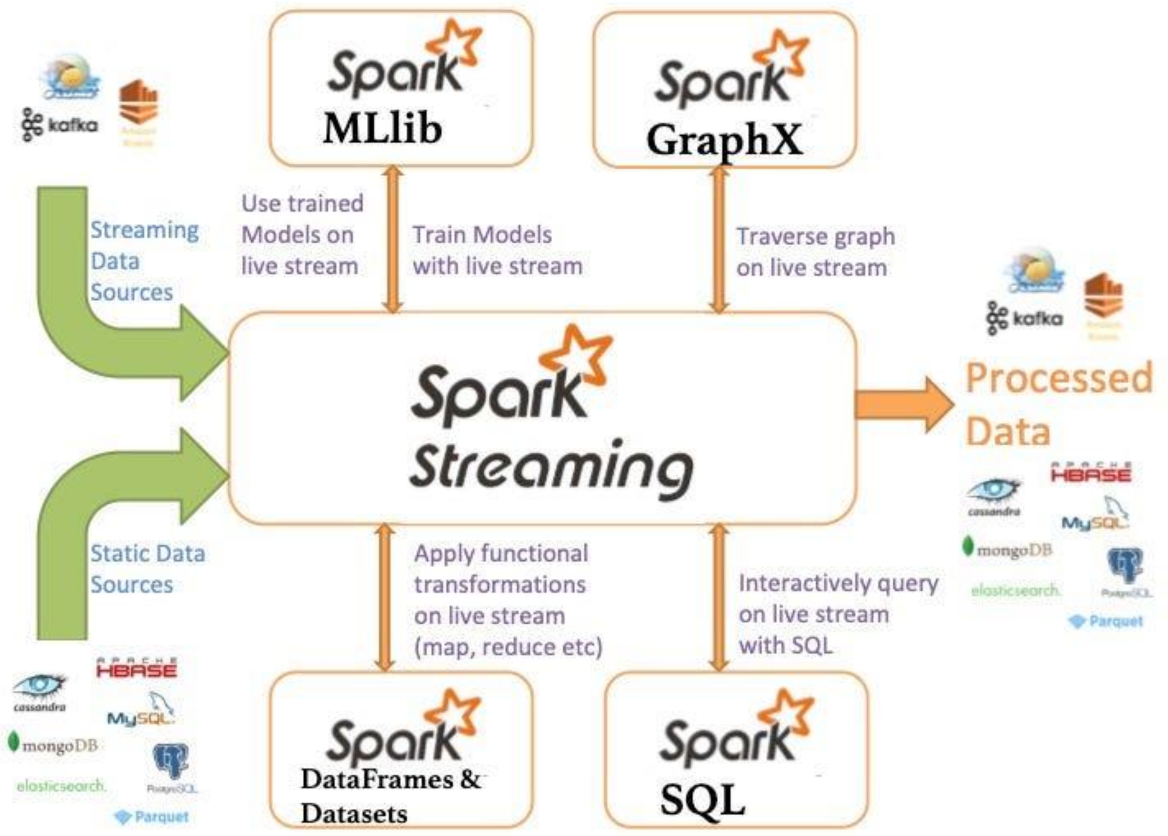
Pandas API on Spark

Pandas API on Spark allows you to scale your pandas workload to any size by running it distributed across multiple nodes. If you are already familiar with pandas and want to leverage Spark for big data, pandas API on Spark makes you immediately productive and lets you migrate your applications without modifying the code. You can have a single codebase that works both with pandas (tests, smaller datasets) and with Spark (production, distributed datasets) and you can switch between the pandas API and the Pandas API on Spark easily and without overhead.

Pandas API on Spark aims to make the transition from pandas to Spark easy but if you are new to Spark or deciding which API to use, we recommend using PySpark (see Spark SQL and DataFrames).



https://spark.apache.org/docs/latest/api/python/reference/pyspark.pandas/index.html



kafka
Spark MLlib
Spark GraphX
Streaming Data Sources
Use trained Models on live stream
Train Models with live stream
Traverse graph on live stream
Spark Streaming
kafka
Processed Data
Static Data Sources
Apply functional transformations on live stream (map, reduce etc)
Interactively query on live stream with SQL
HBASE
cassandra
MySQL
mongoDB
elasticsearch
PostgreSQL
Parquet
Spark DataFrames & Datasets
Spark SQL

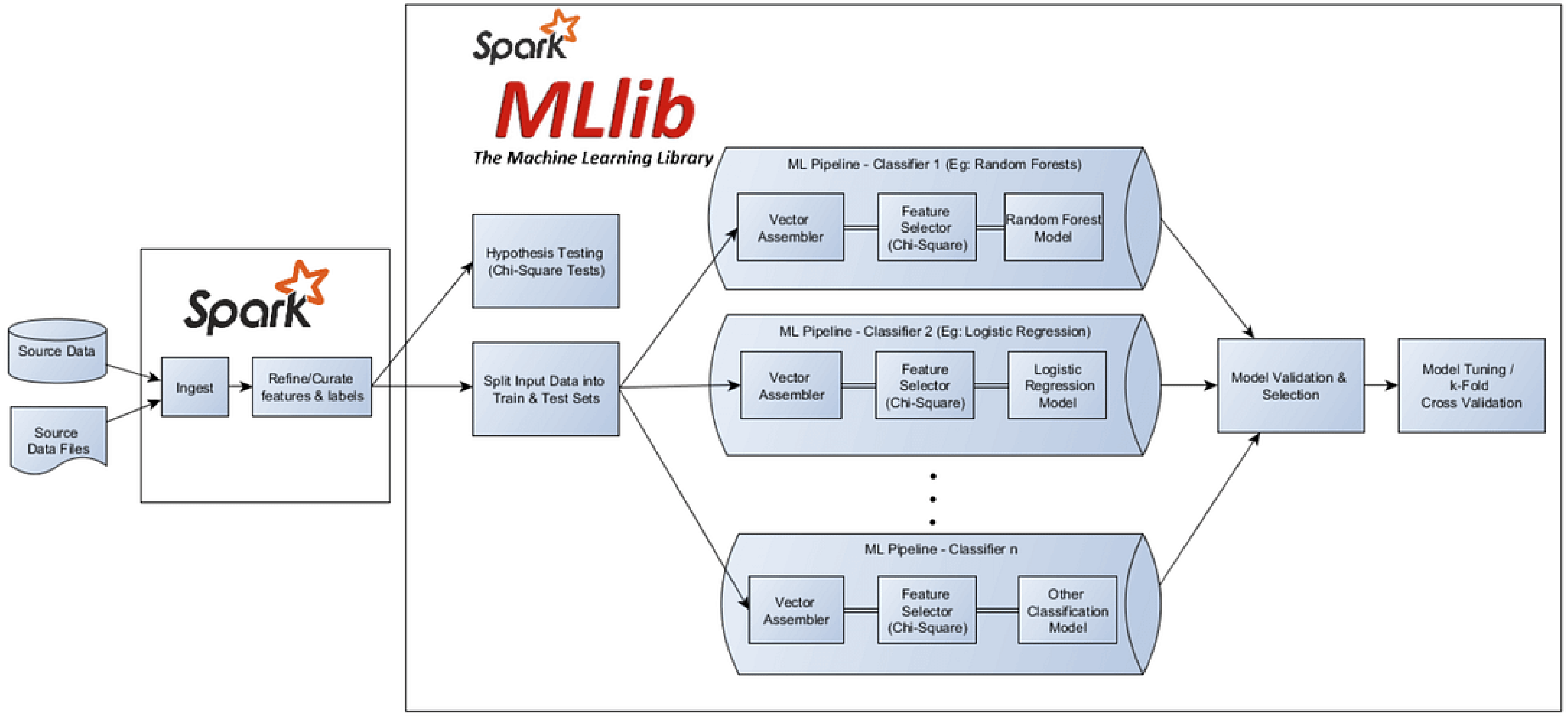
Spark Structured Streaming

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

https://spark.apache.org/docs/latest/api/python/reference/pyspark.ss/index.html

Machine Learning (MLlib)

Built on top of Spark, MLlib is a scalable machine learning library that provides a uniform set of high-level APIs that help users create and tune practical machine learning pipelines.

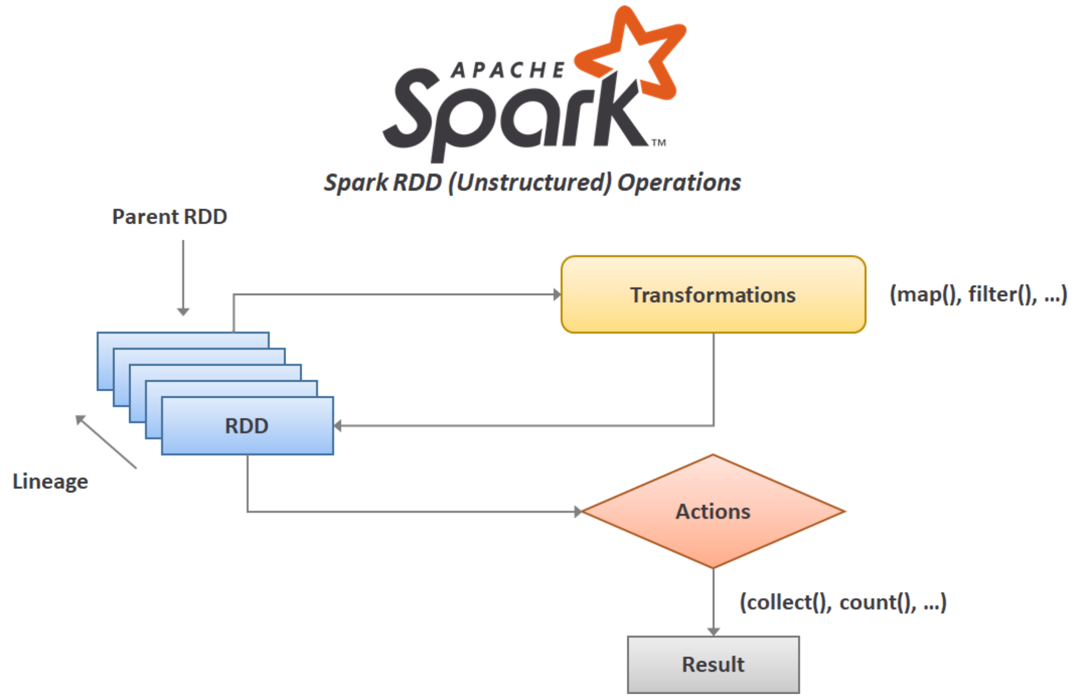

Spark
MLlib
The Machine Learning Library
Source Data
Source Data Files
Spark
Ingest
Refine/Curate features & labels
Hypothesis Testing (Chi-Square Tests)
Split Input Data into Train & Test Sets
ML Pipeline - Classifier 1 (Eg: Random Forests)
Vector Assembler
Feature Selector (Chi-Square)
Random Forest Model
ML Pipeline - Classifier 2 (Eg: Logistic Regression)
Vector Assembler
Feature Selector (Chi-Square)
Logistic Regression Model
ML Pipeline - Classifier n
Vector Assembler
Feature Selector (Chi-Square)
Other Classification Model
Model Validation & Selection
Model Tuning / k-Fold Cross Validation

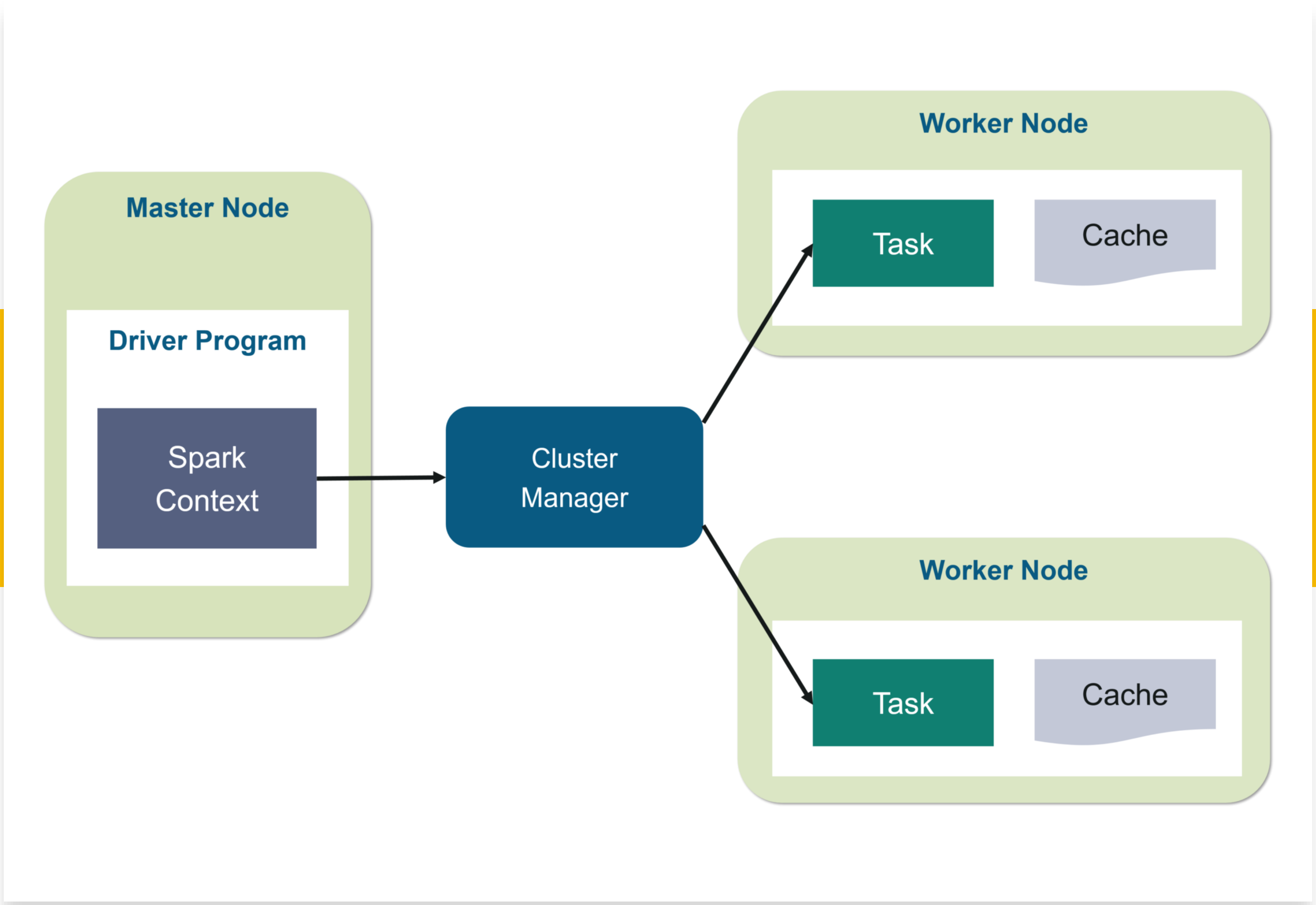
https://spark.apache.org/docs/latest/api/python/reference/pyspark.ml.html

Spark Core and RDDs

Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built on top of. It provides RDDs (Resilient Distributed Datasets) and in-memory computing capabilities.

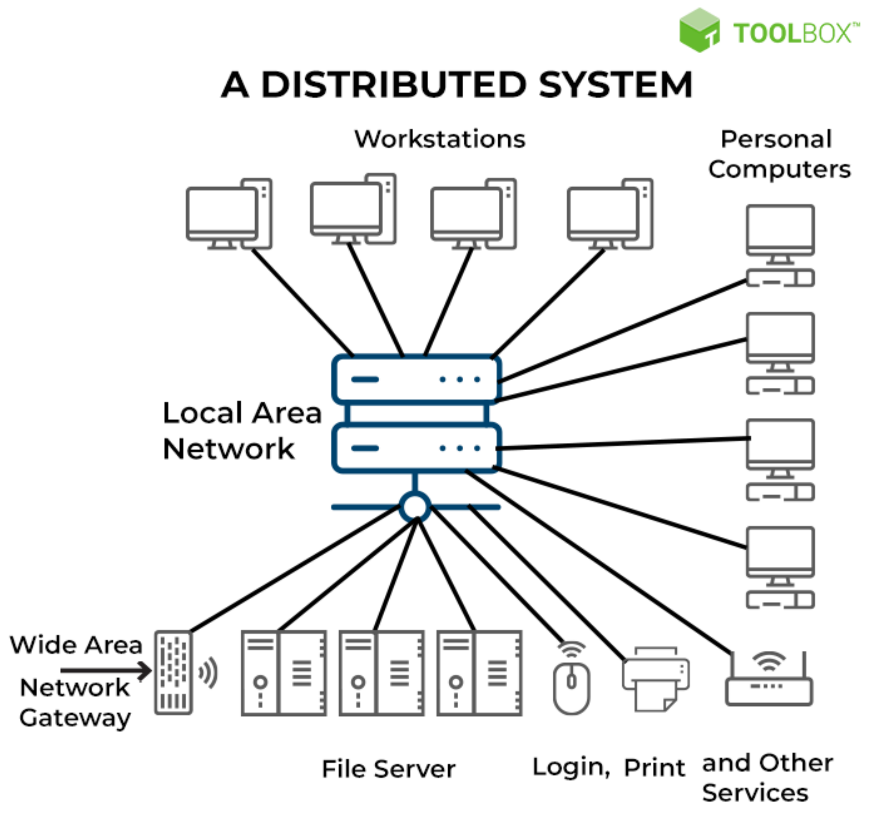
Note that the RDD API is a low-level API which can be difficult to use and you do not get the benefit of Spark's automatic query optimization capabilities. We recommend using DataFrames (see Spark SQL and DataFrames above) instead of RDDs as it allows you to express what you want more easily and lets Spark automatically construct the most efficient query for you.

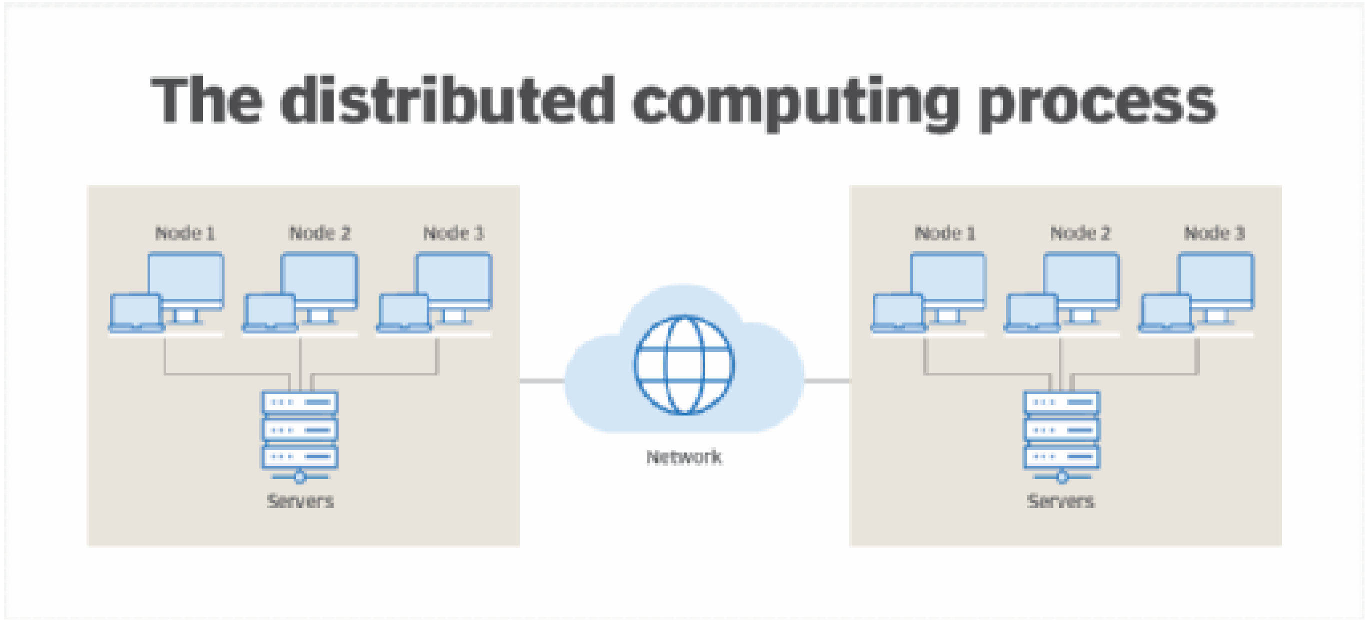




PySpark Architecture

Distributed Computing





What are the advantages of distributed computing?

- **Scalability:**
  Distributed systems can grow with your workload and requirements. You can add new nodes, that is, more computing devices, to the distributed computing network when they are needed.
- **Availability:**
  Your distributed computing system will not crash if one of the computers goes down. The design shows fault tolerance because it can continue to operate even if individual computers fail.
- **Consistency:**
  Computers in a distributed system share information and duplicate data between them, but the system automatically manages data consistency across all the different computers. Thus, you get the benefit of fault tolerance without compromising data consistency.
- **Transparency:**
  Distributed computing systems provide logical separation between the user and the physical devices. You can interact with the system as if it is a single computer without worrying about the setup and configuration of individual machines. You can have different hardware, middleware, software, and operating systems that work together to make your system function smoothly.
- **Efficiency:**
  Distributed systems offer faster performance with optimum resource use of the underlying hardware. As a result, you can manage any workload without worrying about system failure due to volume spikes or underuse of expensive hardware.

How does distributed computing work?

Loose coupling

Tight coupling

What are some distributed computing use cases?

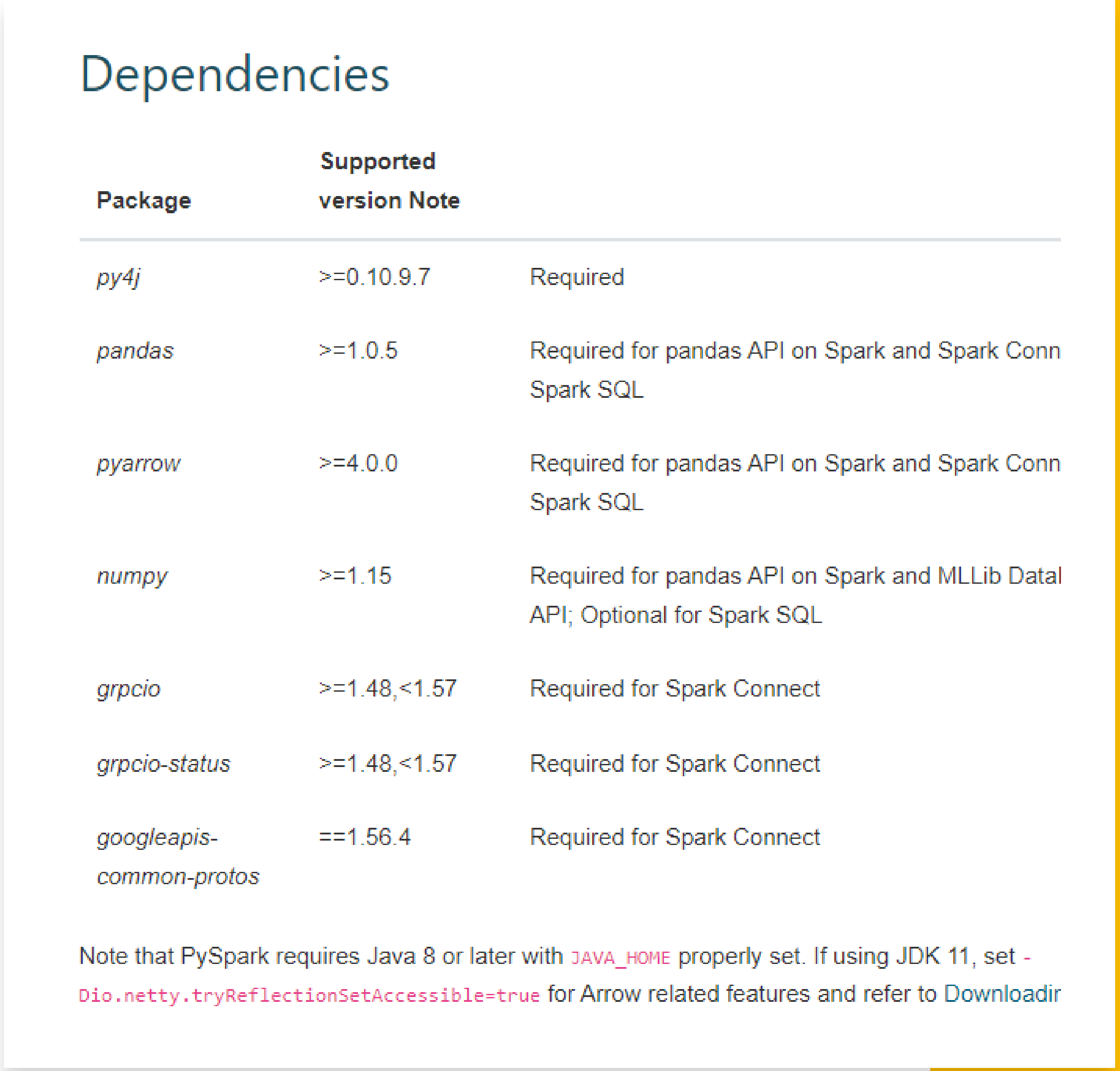
- Healthcare and life sciences
- Engineering research
- Energy and environment
- Financial services

PySpark install and setup

How to install PySpark

Python Versions Supported(Python 3.8 and above.)

1. Using PyPI
2. Using Conda
3. Manually Downloading
4. Installing from Source



Dependencies

Package	Supported version	Note
py4j	>=0.10.9.7	Required
pandas	>=1.0.5	Required for pandas API on Spark and Spark Conn Spark SQL
pyarrow	>=4.0.0	Required for pandas API on Spark and Spark Conn Spark SQL
numpy	>=1.15	Required for pandas API on Spark and MLLib Datal API; Optional for Spark SQL
grpcio	>=1.48,<1.57	Required for Spark Connect
grpcio-status	>=1.48,<1.57	Required for Spark Connect
googleapis-common-protos	==1.56.4	Required for Spark Connect

Note that PySpark requires Java 8 or later with `JAVA_HOME` properly set. If using JDK 11, set `-Dio.netty.tryReflectionSetAccessible=true` for Arrow related features and refer to Downloadir

https://spark.apache.org/docs/latest/api/python/getting_started/install.html

1. Using PyPI

```
pip install pyspark
```

```
# Spark SQL
pip install pyspark[sql]
# pandas API on Spark
pip install pyspark[pandas_on_spark] plotly # to plot your data, you can install plotly together.
# Spark Connect
pip install pyspark[connect]
```

```
PYSPARK_HADOOP_VERSION=3 pip install pyspark
```

```
PYSPARK_RELEASE_MIRROR=http://mirror.apache-kr.org PYSPARK_HADOOP_VERSION=3 pip install
```

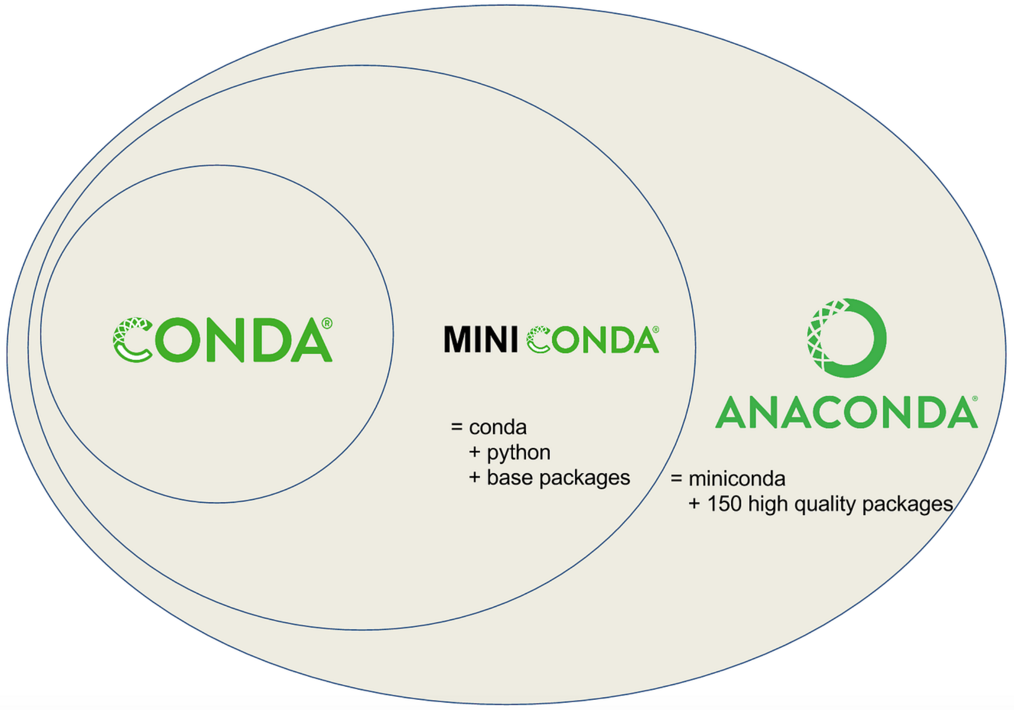
```
PYSPARK_HADOOP_VERSION=3 pip install pyspark -v
```

https://spark.apache.org/docs/latest/api/python/getting_started/install.html

2. Using Conda

```
conda create -n pyspark_env
conda activate pyspark_env
```

```
conda install -c conda-forge pyspark # can also add "python=3.8 some_package [etc.]" here
```

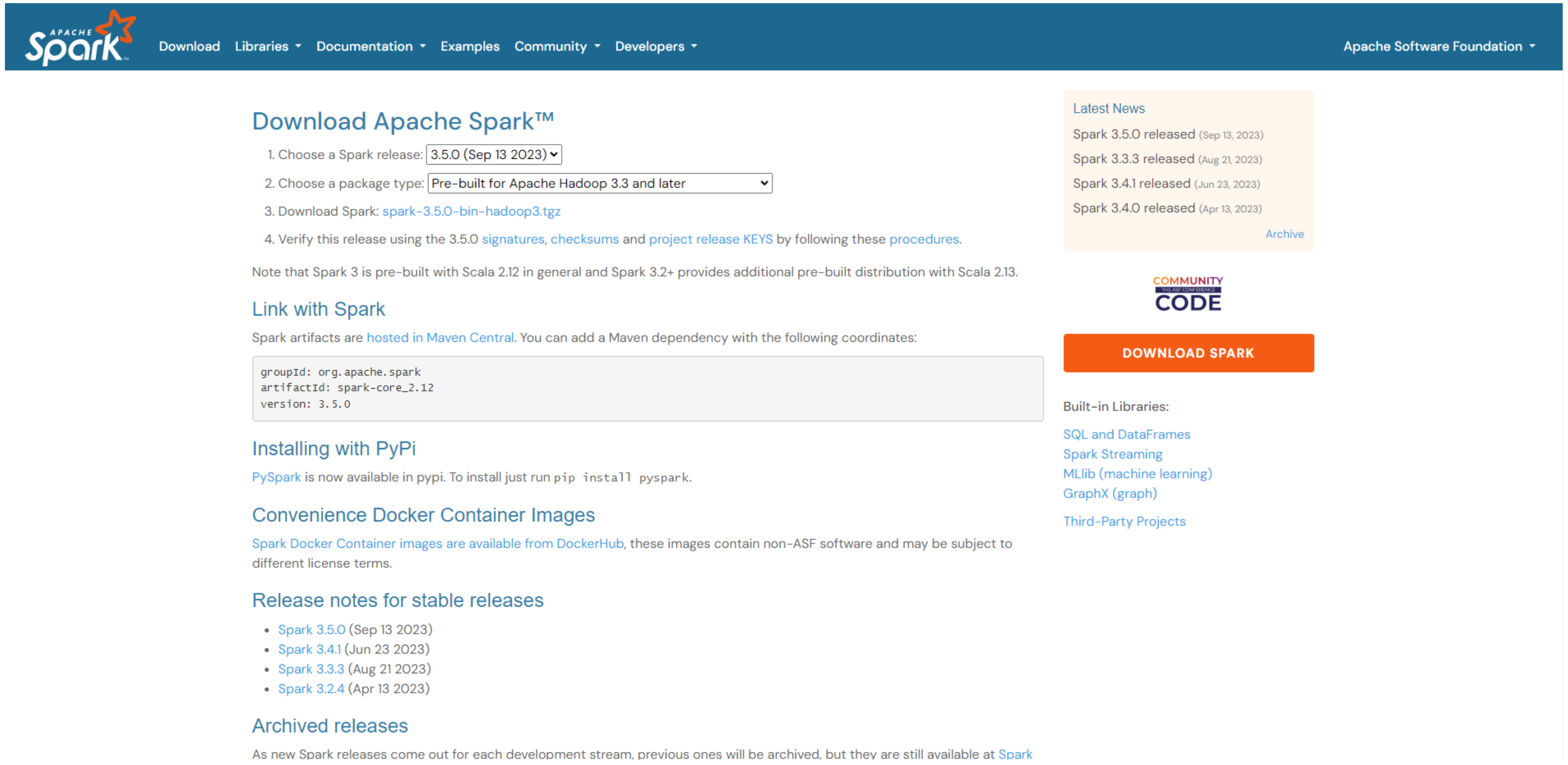


https://spark.apache.org/docs/latest/api/python/getting_started/install.html

3. Manually Downloading

```
tar xzvf spark-3.5.0-bin-hadoop3.tgz
```

```
cd spark-3.5.0-bin-hadoop3
export SPARK_HOME=`pwd`
export PYTHONPATH=$(ZIPS=("$SPARK_HOME"/python/lib/.zip); IFS=:; echo "${ZIPS[]}"):$PYTHONPATH
```



Download Apache Spark™

1. Choose a Spark release: 3.5.0 (Sep 13 2023)
2. Choose a package type: Pre-built for Apache Hadoop 3.3 and later
3. Download Spark: spark-3.5.0-bin-hadoop3.tgz
4. Verify this release using the 3.5.0 signatures, checksums and project release KEYS by following these procedures.

Note that Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13.

Link with Spark

Spark artifacts are hosted in Maven Central. You can add a Maven dependency with the following coordinates:

```
groupId: org.apache.spark
artifactId: spark-core_2.12
version: 3.5.0
```

Installing with PyPi

PySpark is now available in pypi. To install just run `pip install pyspark`.

Convenience Docker Container Images

Spark Docker Container images are available from DockerHub, these images contain non-ASF software and may be subject to different license terms.

Release notes for stable releases

- Spark 3.5.0 (Sep 13 2023)
- Spark 3.4.1 (Jun 23 2023)
- Spark 3.3.3 (Aug 21 2023)
- Spark 3.2.4 (Apr 13 2023)

Archived releases

As new Spark releases come out for each development stream, previous ones will be archived, but they are still available at Spark

Latest News

Spark 3.5.0 released (Sep 13, 2023)
Spark 3.3.3 released (Aug 21, 2023)
Spark 3.4.1 released (Jun 23, 2023)
Spark 3.4.0 released (Apr 13, 2023)
Archive

DOWNLOAD SPARK

Built-in Libraries:

SQL and DataFrames
Spark Streaming
MLlib (machine learning)
GraphX (graph)

Third-Party Projects

https://spark.apache.org/downloads.html

4. Installing from Source

```
./bin/pyspark --master "local[2]"
```

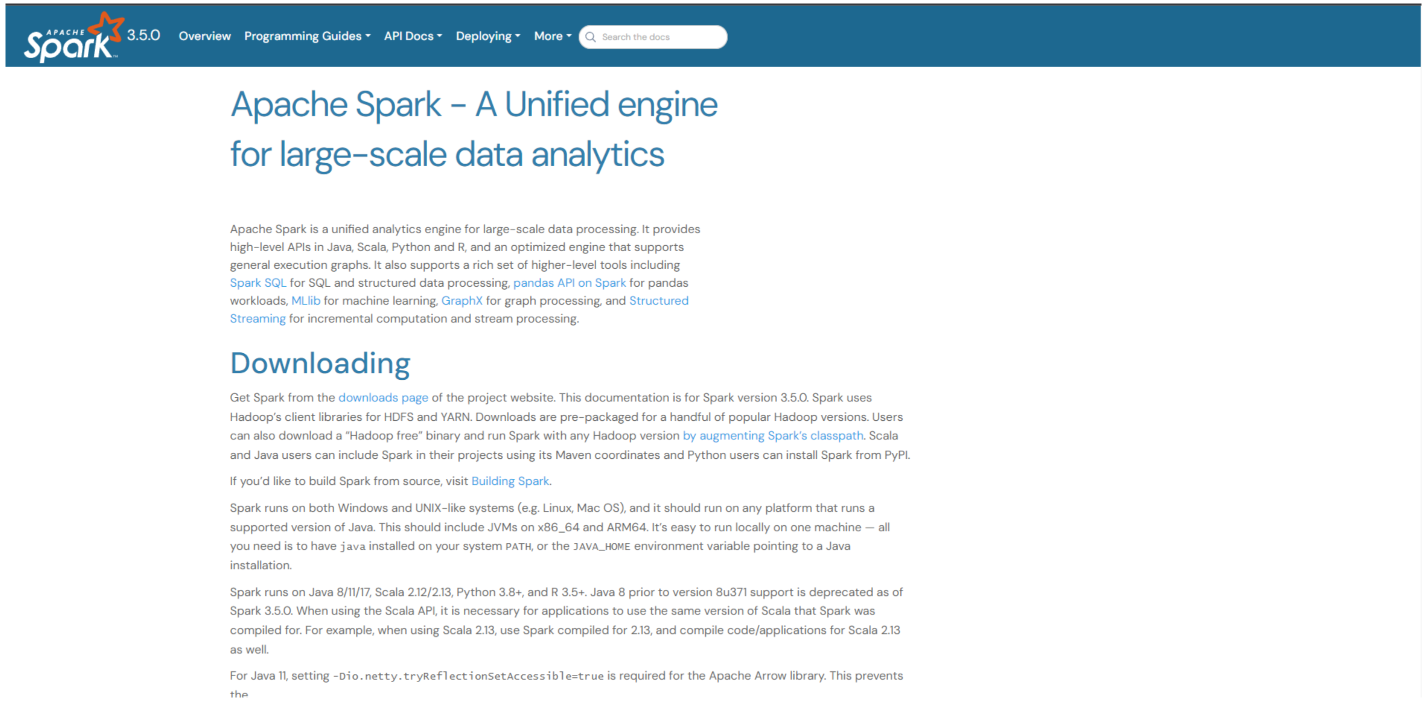
```
./bin/spark-submit examples/src/main/python/pi.py 10
```

```
./bin/run-example SparkPi 10
```

```
./bin/spark-shell --master "local[2]"
```

```
./bin/sparkR --master "local[2]"
```

```
./bin/spark-submit examples/src/main/r/dataframe.R
```



Apache Spark – A Unified engine for large-scale data analytics

Apache Spark is a unified analytics engine for large-scale data processing. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, pandas API on Spark for pandas workloads, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for incremental computation and stream processing.

Downloading

Get Spark from the downloads page of the project website. This documentation is for Spark version 3.5.0. Spark uses Hadoop's client libraries for HDFS and YARN. Downloads are pre-packaged for a handful of popular Hadoop versions. Users can also download a "Hadoop free" binary and run Spark with any Hadoop version by augmenting Spark's classpath. Scala and Java users can include Spark in their projects using its Maven coordinates and Python users can install Spark from PyPI.

If you'd like to build Spark from source, visit Building Spark.

Spark runs on both Windows and UNIX-like systems (e.g. Linux, Mac OS), and it should run on any platform that runs a supported version of Java. This should include JVMs on x86_64 and ARM64. It's easy to run locally on one machine — all you need is to have `java` installed on your system `PATH`, or the `JAVA_HOME` environment variable pointing to a Java installation.

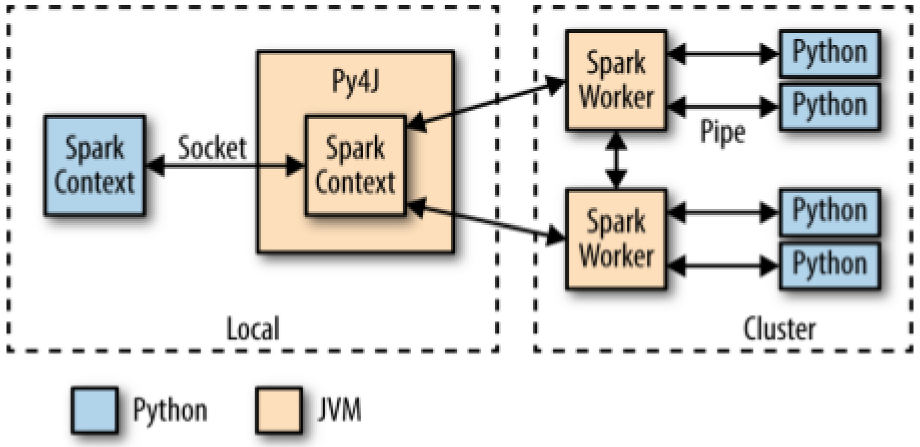
Spark runs on Java 8/11/17, Scala 2.12/2.13, Python 3.8+, and R 3.5+. Java 8 prior to version 8u371 support is deprecated as of Spark 3.5.0. When using the Scala API, it is necessary for applications to use the same version of Scala that Spark was compiled for. For example, when using Scala 2.13, use Spark compiled for 2.13, and compile code/applications for Scala 2.13 as well.

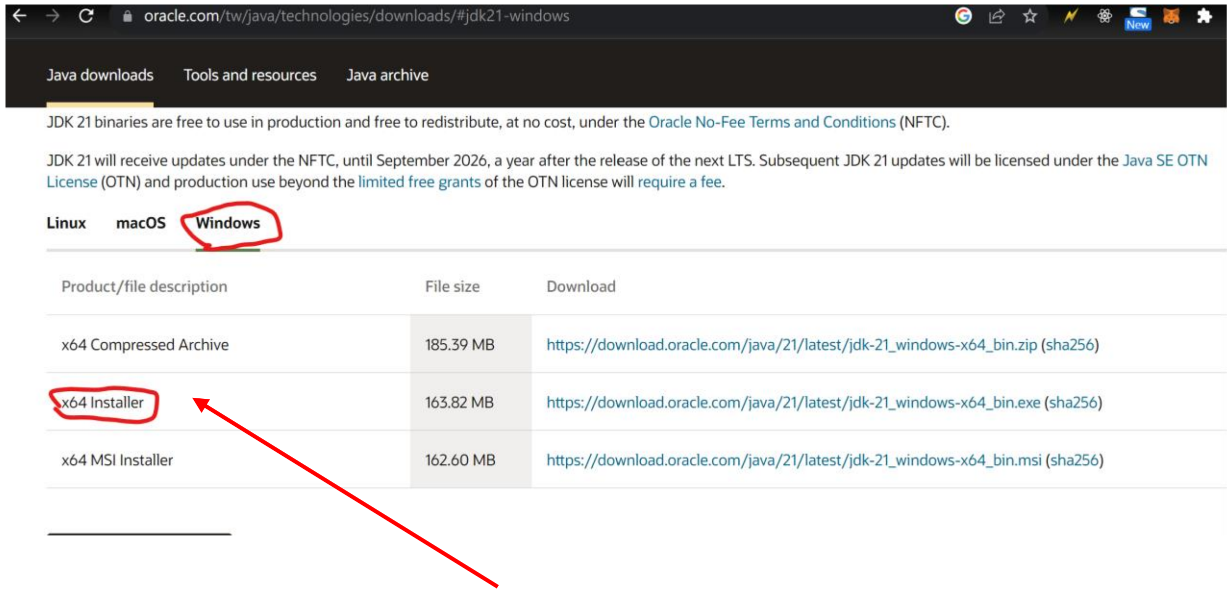
For Java 11, setting `-Dio.netty.tryReflectionSetAccessible=true` is required for the Apache Arrow library. This prevents the

https://spark.apache.org/docs/3.5.0/#downloading

Install The Java Development Kit(JDK)

- PySpark is running on Java Virtual Machine(JVM), So before we run the PySpark application we should install the JDK and set the environment variable.





JDK 21 binaries are free to use in production and free to redistribute, at no cost, under the Oracle No-Fee Terms and Conditions (NFTC).

JDK 21 will receive updates under the NFTC, until September 2026, a year after the release of the next LTS. Subsequent JDK 21 updates will be licensed under the Java SE OTN License (OTN) and production use beyond the limited free grants of the OTN license will require a fee.

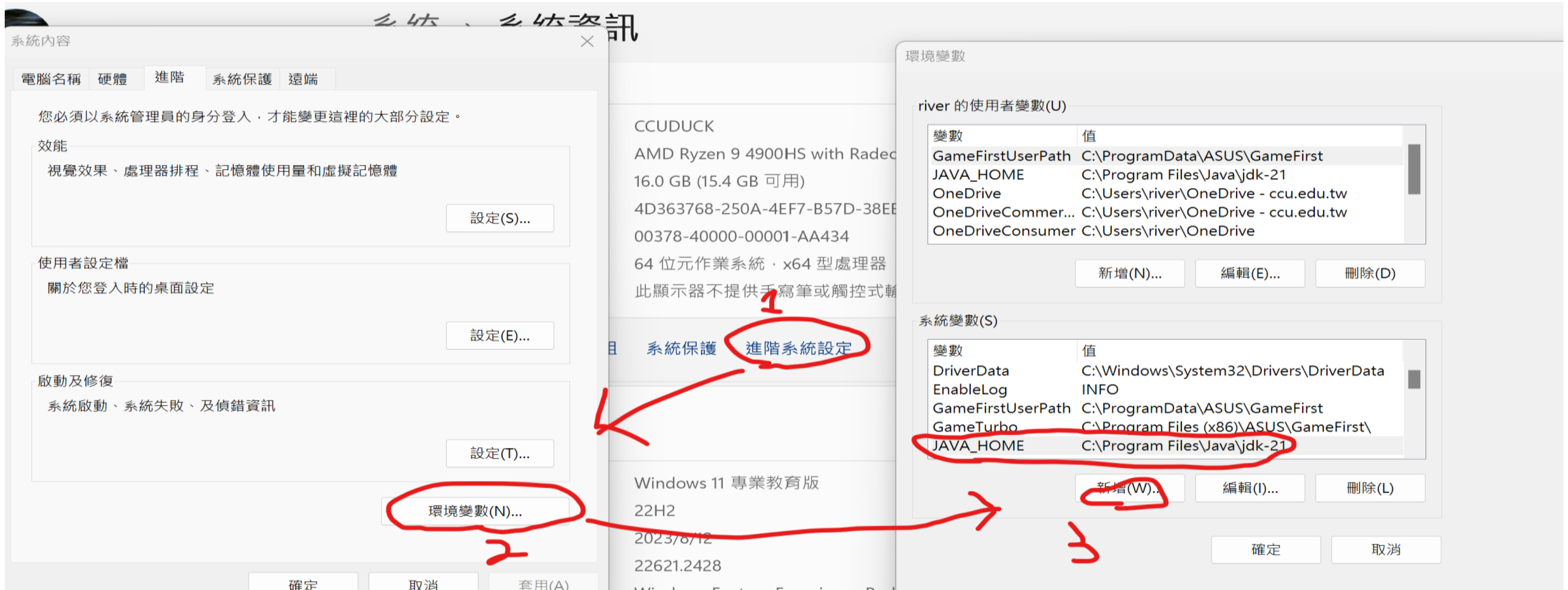
Product/file description	File size	Download
x64 Compressed Archive	185.39 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.zip (sha256)
x64 Installer	163.82 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.exe (sha256)
x64 MSI Installer	162.60 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.msi (sha256)

- Please click the following link and download the suitable JDK executable installer, then install it!

https://www.oracle.com/tw/java/technologies/downloads/

Set The Java Path to The Environment Variable

- Windows
  - In Search, search for and then select: System (Control Panel)
  - Click the **Advanced system settings** link
  - Click **Environment Variables**. In the section **System Variables**, click **New** to add a variable **JAVA_HOME** and **set its value to the path where you install the JDK**
  - **Restart your computer!**


系統內容

電腦名稱 硬體 進階 系統保護 遠端

您必須以系統管理員的身分登入，才能變更這裡的大部分設定。

效能
視覺效果、處理器排程、記憶體使用量和虛擬記憶體
設定(S)...

使用者設定檔
關於您登入時的桌面設定
設定(E)...

啟動及修復
系統啟動、系統失敗、及偵錯資訊
設定(T)...

環境變數(N)...

確定 取消 套用(A)

系統 › 系統資訊

CCUDUCK
AMD Ryzen 9 4900HS with Radec
16.0 GB (15.4 GB 可用)
4D363768-250A-4EF7-B57D-38EE
00378-40000-00001-AA434
64 位元作業系統，x64 型處理器
此顯示器不提供手寫筆或觸控式輸

系統保護 進階系統設定

Windows 11 專業教育版
22H2
2023/8/12
22621.2428

環境變數

river 的使用者變數(U)

變數	值
GameFirstUserPath	C:\ProgramData\ASUS\GameFirst
JAVA_HOME	C:\Program Files\Java\jdk-21
OneDrive	C:\Users\river\OneDrive - ccu.edu.tw
OneDriveCommer...	C:\Users\river\OneDrive - ccu.edu.tw
OneDriveConsumer	C:\Users\river\OneDrive

新增(N)... 編輯(E)... 刪除(D)

系統變數(S)

變數	值
DriverData	C:\Windows\System32\Drivers\DriverData
EnableLog	INFO
GameFirstUserPath	C:\ProgramData\ASUS\GameFirst
GameTurbo	C:\Program Files (x86)\ASUS\GameFirst\
JAVA_HOME	C:\Program Files\Java\jdk-21

新增(W)... 編輯(I)... 刪除(L)

確定 取消

Example 1

Example from the document

1. **DataFrame**
2. **Pandas API on Spark**
3. **Spark Connect**
4. **Testing PySpark**

DataFrame

A PySpark DataFrame can be created via pyspark.sql.SparkSession.createDataFrame typically by passing a list of lists, tuples, dictionaries and pyspark.sql.Rows, a pandas DataFrame and an RDD consisting of such a list. pyspark.sql.SparkSession.createDataFrame takes the schema argument to specify the schema of the DataFrame. When it is omitted, PySpark infers the corresponding schema by taking a sample from the data.

PySpark applications start with initializing SparkSession which is the entry point of PySpark as below.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

https://spark.apache.org/docs/latest/api/python/getting_started/quickstart_df.html

DataFrame Creation

Firstly, you can create a PySpark DataFrame from a list of rows.

```
from datetime import datetime, date
import pandas as pd
from pyspark.sql import Row
df = spark.createDataFrame([
           Row(a=1, b=2., c='string1', d=date(2000, 1, 1), e=datetime(2000, 1, 1, 12, 0)),
           Row(a=2, b=3., c='string2', d=date(2000, 2, 1), e=datetime(2000, 1, 2, 12, 0)),
           Row(a=4, b=5., c='string3', d=date(2000, 3, 1), e=datetime(2000, 1, 3, 12, 0)) ])
df
```

```
[2]: DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

Create a PySpark DataFrame with an explicit schema.

```
df = spark.createDataFrame([
           (1, 2., 'string1', date(2000, 1, 1), datetime(2000, 1, 1, 12, 0)),
           (2, 3., 'string2', date(2000, 2, 1), datetime(2000, 1, 2, 12, 0)),
           (3, 4., 'string3', date(2000, 3, 1), datetime(2000, 1, 3, 12, 0))
           ], schema='a long, b double, c string, d date, e timestamp’)
df
```

```
[3]: DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

DataFrame Creation

Create a PySpark DataFrame from a pandas DataFrame.

```
pandas_df = pd.DataFrame({
           'a': [1, 2, 3],
           'b': [2., 3., 4.],
           'c': ['string1', 'string2', 'string3'],
           'd': [date(2000, 1, 1), date(2000, 2, 1), date(2000, 3, 1)],
           'e': [datetime(2000, 1, 1, 12, 0), datetime(2000, 1, 2, 12, 0), datetime(2000, 1, 3, 12, 0)]
})
df = spark.createDataFrame(pandas_df)
df
```

```
[4]: DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

The DataFrames created above all have the same results and schema.

```
# All DataFrames above result same.
df.show()
df.printSchema()
```

```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
|  2|3.0|string2|2000-02-01|2000-01-02 12:00:00|
|  3|4.0|string3|2000-03-01|2000-01-03 12:00:00|
+---+---+-------+----------+-------------------+

root
 |-- a: long (nullable = true)
 |-- b: double (nullable = true)
 |-- c: string (nullable = true)
 |-- d: date (nullable = true)
 |-- e: timestamp (nullable = true)
```

Viewing Data

The top rows of a DataFrame can be displayed using DataFrame.show().

```
df.show(1)
```

```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+-------+----------+-------------------+
only showing top 1 row
```

Alternatively, you can enable spark.sql.repl.eagerEval.enabled configuration for the eager evaluation of PySpark DataFrame in notebooks such as Jupyter. The number of rows to show can be controlled via spark.sql.repl.eagerEval.maxNumRows configuration.

```
spark.conf.set('spark.sql.repl.eagerEval.enabled', True)
df
```

[8]:

a	b	c	d	e
1	2.0	string1	2000-01-01	2000-01-01 12:00:00
2	3.0	string2	2000-02-01	2000-01-02 12:00:00
3	4.0	string3	2000-03-01	2000-01-03 12:00:00

Viewing Data

The rows can also be shown vertically. This is useful when rows are too long to show horizontally.

```
df.show(1, vertical=True)
```

```
-RECORD 0------------------
 a   | 1
 b   | 2.0
 c   | string1
 d   | 2000-01-01
 e   | 2000-01-01 12:00:00
only showing top 1 row
```

You can see the DataFrame's schema and column names as follows:

```
df.columns
```

```
[10]: ['a', 'b', 'c', 'd', 'e']
```

```
df.printSchema()
```

```
root
 |-- a: long (nullable = true)
 |-- b: double (nullable = true)
 |-- c: string (nullable = true)
 |-- d: date (nullable = true)
 |-- e: timestamp (nullable = true)
```

Viewing Data

Show the summary of the DataFrame

```
df.select("a", "b", "c").describe().show()
```

```
+-------+---+---+-------+
|summary|  a|  b|      c|
+-------+---+---+-------+
|  count|  3|  3|      3|
|   mean|2.0|3.0|   null|
| stddev|1.0|1.0|   null|
|    min|  1|2.0|string1|
|    max|  3|4.0|string3|
+-------+---+---+-------+
```

DataFrame.collect() collects the distributed data to the driver side as the local data in Python. Note that this can throw an out-of-memory error when the dataset is too large to fit in the driver side because it collects all the data from executors to the driver side.

```
df.collect()
```

```
[13]: [Row(a=1, b=2.0, c='string1', d=datetime.date(2000, 1, 1), e=datetime.datetime(2000, 1, 1, 12, 0))
       Row(a=2, b=3.0, c='string2', d=datetime.date(2000, 2, 1), e=datetime.datetime(2000, 1, 2, 12, 0))
       Row(a=3, b=4.0, c='string3', d=datetime.date(2000, 3, 1), e=datetime.datetime(2000, 1, 3, 12, 0))
```

Viewing Data

In order to avoid throwing an out-of-memory exception, use DataFrame.take() or DataFrame.tail().

```
df.take(1)
```

```
[14]: [Row(a=1, b=2.0, c='string1', d=datetime.date(2000, 1, 1), e=datetime.datetime(2000, 1, 1, 12, 0))]
```

PySpark DataFrame also provides the conversion back to a pandas DataFrame to leverage pandas API. Note that toPandas also collects all data into the driver side that can easily cause an out-of-memory-error when the data is too large to fit into the driver side.

```
df.toPandas()
```

[15]:

	a	b	c	d	e
0	1	2.0	string1	2000-01-01	2000-01-01 12:00:00
1	2	3.0	string2	2000-02-01	2000-01-02 12:00:00
2	3	4.0	string3	2000-03-01	2000-01-03 12:00:00

Selecting and Accessing Data

PySpark DataFrame is lazily evaluated and simply selecting a column does not trigger the computation but it returns a Column instance.

```
df.a
[16]: Column<b'a'>
```

In fact, most of column-wise operations return Columns.

```
from pyspark.sql import Column
from pyspark.sql.functions import upper
type(df.c) == type(upper(df.c)) == type(df.c.isNull())
[17]: True
```

Selecting and Accessing Data

These Columns can be used to select the columns from a DataFrame. For example, DataFrame.select() takes the Column instances that returns another DataFrame.

```
df.select(df.c).show()
+-------+
|      c|
+-------+
|string1|
|string2|
|string3|
+-------+
```

Assign new Column instance.

```
df.withColumn('upper_c', upper(df.c)).show()
```

```
+---+---+-------+----------+-------------------+-------+
|  a|  b|      c|         d|                  e|upper_c|
+---+---+-------+----------+-------------------+-------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|STRING1|
|  2|3.0|string2|2000-02-01|2000-01-02 12:00:00|STRING2|
|  3|4.0|string3|2000-03-01|2000-01-03 12:00:00|STRING3|
+---+---+-------+----------+-------------------+-------+
```

To select a subset of rows, use DataFrame.filter().

```
df.filter(df.a == 1).show()
```

```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+-------+----------+-------------------+
```

Applying a Function

PySpark supports various UDFs and APIs to allow users to execute Python native functions. See also the latest Pandas UDFs and Pandas Function APIs. For instance, the example below allows users to directly use the APIs in a pandas Series within Python native function.

```
import pandas as pd
from pyspark.sql.functions import pandas_udf
@pandas_udf('long')
def pandas_plus_one(series: pd.Series) -> pd.Series:
        # Simply plus one by using pandas Series.
        return series + 1
df.select(pandas_plus_one(df.a)).show()
```

```
+------------------+
|pandas_plus_one(a)|
+------------------+
|                 2|
|                 3|
|                 4|
+------------------+
```

Another example is DataFrame.mapInPandas which allows users directly use the APIs in a pandas DataFrame without any restrictions such as the result length.

```
def pandas_filter_func(iterator):
        for pandas_df in iterator:
                yield pandas_df[pandas_df.a == 1]
df.mapInPandas(pandas_filter_func, schema=df.schema).show()
```

```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+-------+----------+-------------------+
```

Grouping Data

PySpark DataFrame also provides a way of handling grouped data by using the common approach, split-apply-combine strategy. It groups the data by a certain condition applies a function to each group and then combines them back to the DataFrame.

```
df = spark.createDataFrame([
          ['red', 'banana', 1, 10], ['blue', 'banana', 2, 20], ['red', 'carrot', 3, 30],
          ['blue', 'grape', 4, 40], ['red', 'carrot', 5, 50], ['black', 'carrot', 6, 60],
          ['red', 'banana', 7, 70], ['red', 'grape', 8, 80]], schema=['color', 'fruit', 'v1', 'v2′])
df.show()
```

```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

Grouping and then applying the avg() function to the resulting groups.

```
df.groupby('color').avg().show()
```

```
+-----+-------+-------+
|color|avg(v1)|avg(v2)|
+-----+-------+-------+
|  red|    4.8|   48.0|
|black|    6.0|   60.0|
| blue|    3.0|   30.0|
+-----+-------+-------+
```

Grouping Data

You can also apply a Python native function against each group by using pandas API.

```
def plus_mean(pandas_df):
        return pandas_df.assign(v1=pandas_df.v1 - pandas_df.v1.mean())
df.groupby('color').applyInPandas(plus_mean, schema=df.schema).show()
```

```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana| -3| 10|
|  red|carrot| -1| 30|
|  red|carrot|  0| 50|
|  red|banana|  2| 70|
|  red| grape|  3| 80|
|black|carrot|  0| 60|
| blue|banana| -1| 20|
| blue| grape|  1| 40|
+-----+------+---+---+
```

Co-grouping and applying a function.

```
df1 = spark.createDataFrame(
        [(20000101, 1, 1.0), (20000101, 2, 2.0), (20000102, 1, 3.0), (20000102, 2, 4.0)],
        ('time', 'id', 'v1’))
df2 = spark.createDataFrame(
        [(20000101, 1, 'x'), (20000101, 2, 'y’)],
        ('time', 'id', 'v2’))
def merge_ordered(l, r):
        return pd.merge_ordered(l, r)
df1.groupby('id').cogroup(df2.groupby('id')).applyInPandas(
        merge_ordered, schema='time int, id int, v1 double, v2 string').show()
```

```
+--------+---+---+---+
|    time| id| v1| v2|
+--------+---+---+---+
|20000101|  1|1.0|  x|
|20000102|  1|3.0|  x|
|20000101|  2|2.0|  y|
|20000102|  2|4.0|  y|
+--------+---+---+---+
```

Getting Data In/Out

CSV is straightforward and easy to use. Parquet and ORC are efficient and compact file formats to read and write faster.

CSV

```
df.write.csv('foo.csv', header=True)
spark.read.csv('foo.csv', header=True).show()
```

```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

Parquet

```
df.write.parquet('bar.parquet’)
spark.read.parquet('bar.parquet').show()
```

```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

ORC

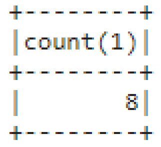
```
df.write.orc('zoo.orc’)
spark.read.orc('zoo.orc').show()
```

```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

Working with SQL

DataFrame and Spark SQL share the same execution engine so they can be interchangeably used seamlessly. For example, you can register the DataFrame as a table and run a SQL easily as below:

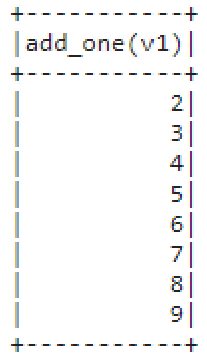
```
df.createOrReplaceTempView("tableA")
spark.sql("SELECT count(*) from tableA").show()
```



```
+--------+
|count(1)|
+--------+
|       8|
+--------+
```

In addition, UDFs can be registered and invoked in SQL out of the box:

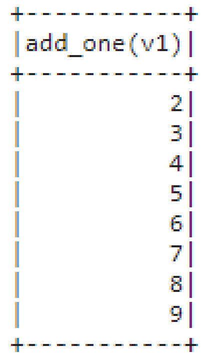
```
@pandas_udf("integer")
def add_one(s: pd.Series) -> pd.Series:
           return s + 1
spark.udf.register("add_one", add_one)
spark.sql("SELECT add_one(v1) FROM tableA").show()
```



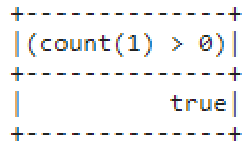
```
+-----------+
|add_one(v1)|
+-----------+
|          2|
|          3|
|          4|
|          5|
|          6|
|          7|
|          8|
|          9|
+-----------+
```

These SQL expressions can directly be mixed and used as PySpark columns.

```
from pyspark.sql.functions import expr
df.selectExpr('add_one(v1)').show()
df.select(expr('count(*)') > 0).show()
```



```
+-----------+
|add_one(v1)|
+-----------+
|          2|
|          3|
|          4|
|          5|
|          6|
|          7|
|          8|
|          9|
+-----------+
```



```
+--------------+
|(count(1) > 0)|
+--------------+
|          true|
+--------------+
```

Spark Connect

Spark Connect introduced a decoupled client-server architecture for Spark that allows remote connectivity to Spark clusters using the DataFrame API.

Launch Spark server with Spark Connect

To launch Spark with support for Spark Connect sessions, run the start-connect-server.sh script.

```
!$HOME/sbin/start-connect-server.sh --packages org.apache.spark:spark-connect_2.12:$SPARK_VERSION
```

Connect to Spark Connect server

Now that the Spark server is running, we can connect to it remotely using Spark Connect. We do this by creating a remote Spark session on the client where our application runs. Before we can do that, we need to make sure to stop the existing regular Spark session because it cannot coexist with the remote Spark Connect session we are about to create.

```
from pyspark.sql import SparkSession
SparkSession.builder.master("local[*]").getOrCreate().stop()
```

The command we used above to launch the server configured Spark to run as localhost:15002. So now we can create a remote Spark session on the client using the following command.

```
spark = SparkSession.builder.remote("sc://localhost:15002").getOrCreate()
```

https://spark.apache.org/docs/latest/api/python/getting_started/quickstart_connect.html

Create DataFrame

Once the remote Spark session is created successfully, it can be used the same way as a regular Spark session. Therefore, you can create a DataFrame with the following command.

```
from datetime import datetime, date
from pyspark.sql import Row
df = spark.createDataFrame([
              Row(a=1, b=2., c='string1', d=date(2000, 1, 1), e=datetime(2000, 1, 1, 12, 0)),
              Row(a=2, b=3., c='string2', d=date(2000, 2, 1), e=datetime(2000, 1, 2, 12, 0)),
              Row(a=4, b=5., c='string3', d=date(2000, 3, 1), e=datetime(2000, 1, 3, 12, 0)) ])
df.show()
```

```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
|  2|3.0|string2|2000-02-01|2000-01-02 12:00:00|
|  4|5.0|string3|2000-03-01|2000-01-03 12:00:00|
+---+---+-------+----------+-------------------+
```

Pandas API on Spark

This notebook shows you some key differences between pandas and pandas API on Spark.

Customarily, we import pandas API on Spark as follows:

```python
import pandas as pd
import numpy as np
import pyspark.pandas as ps
from pyspark.sql import SparkSession
```

https://spark.apache.org/docs/latest/api/python/getting_started/quickstart_ps.html

Object Creation

Creating a pandas-on-Spark Series by passing a list of values, letting pandas API on Spark create a default integer index:

```
s = ps.Series([1, 3, 5, np.nan, 6, 8])
s
```

```
[3]: 0    1.0
     1    3.0
     2    5.0
     3    NaN
     4    6.0
     5    8.0
     dtype: float64
```

Creating a pandas-on-Spark DataFrame by passing a dict of objects that can be converted to series-like.

```
psdf = ps.DataFrame(
          {'a': [1, 2, 3, 4, 5, 6],
          'b': [100, 200, 300, 400, 500, 600],
          'c': ["one", "two", "three", "four", "five", "six"]},
          index=[10, 20, 30, 40, 50, 60])
psdf
```

[5]:

	a	b	c
10	1	100	one
20	2	200	two
30	3	300	three
40	4	400	four
50	5	500	five
60	6	600	six

Object Creation

Creating a pandas DataFrame by passing a numpy array, with a datetime index and labeled columns:

```
dates = pd.date_range('20130101', periods=6)
```

```
dates
```

```
[7]: DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
                    '2013-01-05', '2013-01-06'],
                   dtype='datetime64[ns]', freq='D')
```

```
pdf = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=('ABCD'))
```

```
pdf
```

[9]:

	A	B	C	D
2013-01-01	0.912558	-0.795645	-0.289115	0.187606
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828
2013-01-03	0.332871	-1.262010	-0.434844	-0.579920
2013-01-04	0.924016	-1.022019	-0.405249	-1.036021
2013-01-05	-0.772209	-1.228099	0.068901	0.896679
2013-01-06	1.485582	-0.709306	-0.202637	-0.248766

Now, this pandas DataFrame can be converted to a pandas-on-Spark DataFrame

```
psdf = ps.from_pandas(pdf)
type(psdf)
```

```
[11]: pyspark.pandas.frame.DataFrame
```

Object Creation

It looks and behaves the same as a pandas DataFrame.

```
psdf
```

[12]:

	A	B	C	D
2013-01-01	0.912558	-0.795645	-0.289115	0.187606
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828
2013-01-03	0.332871	-1.262010	-0.434844	-0.579920
2013-01-04	0.924016	-1.022019	-0.405249	-1.036021
2013-01-05	-0.772209	-1.228099	0.068901	0.896679
2013-01-06	1.485582	-0.709306	-0.202637	-0.248766

Also, it is possible to create a pandas-on-Spark DataFrame from Spark DataFrame easily.
Creating a Spark DataFrame from pandas DataFrame

```
spark = SparkSession.builder.getOrCreate()
sdf = spark.createDataFrame(pdf)
sdf.show()
```

```
+--------------------+-------------------+--------------------+--------------------+
|                   A|                  B|                   C|                   D|
+--------------------+-------------------+--------------------+--------------------+
|    0.91255803205208|-0.7956452608556638|-0.28911463069772175| 0.18760566615081622|
|-0.05970271470242...| -1.233896949308984|  0.3166246451758431| -1.2268284000402265|
|  0.3328710694753661|-1.2620100816441786| -0.4348444277082644| -0.5799199651437185|
|  0.9240158461589916|-1.0220190956326003| -0.4052488880650239| -1.0360212104348547|
| -0.7722090016558953|-1.2280986385313222|  0.0689011451939635|  0.8966790729426755|
|  1.4855822995785612|-0.7093056426018517| -0.2026366848847041|-0.24876619876451092|
+--------------------+-------------------+--------------------+--------------------+
```

Object Creation

Creating pandas-on-Spark DataFrame from Spark DataFrame.

```
psdf = sdf.pandas_api()
```

```
psdf
```

[17]:

	A	B	C	D
0	0.912558	-0.795645	-0.289115	0.187606
1	-0.059703	-1.233897	0.316625	-1.226828
2	0.332871	-1.262010	-0.434844	-0.579920
3	0.924016	-1.022019	-0.405249	-1.036021
4	-0.772209	-1.228099	0.068901	0.896679
5	1.485582	-0.709306	-0.202637	-0.248766

Having specific dtypes . Types that are common to both Spark and pandas are currently supported.

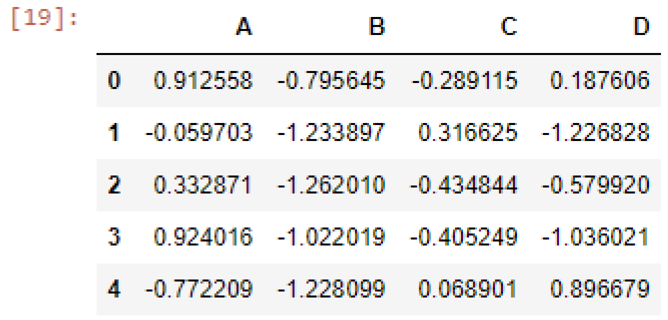
```
psdf.dtypes
```

```
[18]: A    float64
      B    float64
      C    float64
      D    float64
      dtype: object
```

Object Creation

Here is how to show top rows from the frame below.
Note that the data in a Spark dataframe does not preserve the natural order by default. The natural order can be preserved by setting compute.ordered_head option but it causes a performance overhead with sorting internally.

```
psdf.head()
```



[19]:

	A	B	C	D
0	0.912558	-0.795645	-0.289115	0.187606
1	-0.059703	-1.233897	0.316625	-1.226828
2	0.332871	-1.262010	-0.434844	-0.579920
3	0.924016	-1.022019	-0.405249	-1.036021
4	-0.772209	-1.228099	0.068901	0.896679

Displaying the index, columns, and the underlying numpy data.

```
psdf.index
```

```
[20]: Int64Index([0, 1, 2, 3, 4, 5], dtype='int64')
```

```
psdf.columns
```

```
[21]: Index(['A', 'B', 'C', 'D'], dtype='object')
```

```
psdf.to_numpy()
```

```
[22]: array([[ 0.91255803, -0.79564526, -0.28911463,  0.18760567],
             [-0.05970271, -1.23389695,  0.31662465, -1.2268284 ],
             [ 0.33287107, -1.26201008, -0.43484443, -0.57991997],
             [ 0.92401585, -1.0220191 , -0.40524889, -1.03602121],
             [-0.772209  , -1.22809864,  0.06890115,  0.89667907],
             [ 1.4855823 , -0.70930564, -0.20263668, -0.2487662 ]])
```

Object Creation

Showing a quick statistic summary of your data

```
psdf.describe()
```

[23]:

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.470519	-1.041829	-0.157720	-0.334542
std	0.809428	0.241511	0.294520	0.793014
min	-0.772209	-1.262010	-0.434844	-1.226828
25%	-0.059703	-1.233897	-0.405249	-1.036021
50%	0.332871	-1.228099	-0.289115	-0.579920
75%	0.924016	-0.795645	0.068901	0.187606
max	1.485582	-0.709306	0.316625	0.896679

Sorting by its index

```
psdf.sort_index(ascending=False)
```

[25]:

	A	B	C	D
5	1.485582	-0.709306	-0.202637	-0.248766
4	-0.772209	-1.228099	0.068901	0.896679
3	0.924016	-1.022019	-0.405249	-1.036021
2	0.332871	-1.262010	-0.434844	-0.579920
1	-0.059703	-1.233897	0.316625	-1.226828
0	0.912558	-0.795645	-0.289115	0.187606

Transposing your data

```
psdf.T
```

[24]:

	0	1	2	3	4	5
A	0.912558	-0.059703	0.332871	0.924016	-0.772209	1.485582
B	-0.795645	-1.233897	-1.262010	-1.022019	-1.228099	-0.709306
C	-0.289115	0.316625	-0.434844	-0.405249	0.068901	-0.202637
D	0.187606	-1.226828	-0.579920	-1.036021	0.896679	-0.248766

Sorting by value

```
psdf.sort_values(by='B')
```

[26]:

	A	B	C	D
2	0.332871	-1.262010	-0.434844	-0.579920
1	-0.059703	-1.233897	0.316625	-1.226828
4	-0.772209	-1.228099	0.068901	0.896679
3	0.924016	-1.022019	-0.405249	-1.036021
0	0.912558	-0.795645	-0.289115	0.187606
5	1.485582	-0.709306	-0.202637	-0.248766

Missing Data

Pandas API on Spark primarily uses the value np.nan to represent missing data. It is by default not included in computations.

```
pdf1 = pdf.reindex(index=dates[0:4], columns=(pdf.columns) + ['E'])
pdf1.loc[dates[0]:dates[1], 'E'] = 1
psdf1 = ps.from_pandas(pdf1)
psdf1
```

[30]:

	A	B	C	D	E
2013-01-01	0.912558	-0.795645	-0.289115	0.187606	1.0
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828	1.0
2013-01-03	0.332871	-1.262010	-0.434844	-0.579920	NaN
2013-01-04	0.924016	-1.022019	-0.405249	-1.036021	NaN

To drop any rows that have missing data.

```
psdf1.dropna(how='any')
```

[31]:

	A	B	C	D	E
2013-01-01	0.912558	-0.795645	-0.289115	0.187606	1.0
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828	1.0

Filling missing data.

```
psdf1.fillna(value=5)
```

[32]:

	A	B	C	D	E
2013-01-01	0.912558	-0.795645	-0.289115	0.187606	1.0
2013-01-02	-0.059703	-1.233897	0.316625	-1.226828	1.0
2013-01-03	0.332871	-1.262010	-0.434844	-0.579920	5.0
2013-01-04	0.924016	-1.022019	-0.405249	-1.036021	5.0

Operations

Stats

Performing a descriptive statistic:

```
psdf.mean()
```

```
[33]: A    0.470519
      B   -1.041829
      C   -0.157720
      D   -0.334542
      dtype: float64
```

Spark Configurations

Various configurations in PySpark could be applied internally in pandas API on Spark. For example, you can enable Arrow optimization to hugely speed up internal pandas conversion. See also PySpark Usage Guide for Pandas with Apache Arrow in PySpark documentation.

```
prev = spark.conf.get("spark.sql.execution.arrow.pyspark.enabled") # Keep its default value.
ps.set_option("compute.default_index_type", "distributed") # Use default index prevent overhead.
import warnings
warnings.filterwarnings("ignore") # Ignore warnings coming from Arrow optimizations.
```

```
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", True) %timeit ps.range(300000).to_pandas()
```

```
900 ms ± 186 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", False) %timeit ps.range(300000).to_pandas()
```

```
3.08 s ± 227 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
ps.reset_option("compute.default_index_type")
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", prev) # Set its default value back.
```

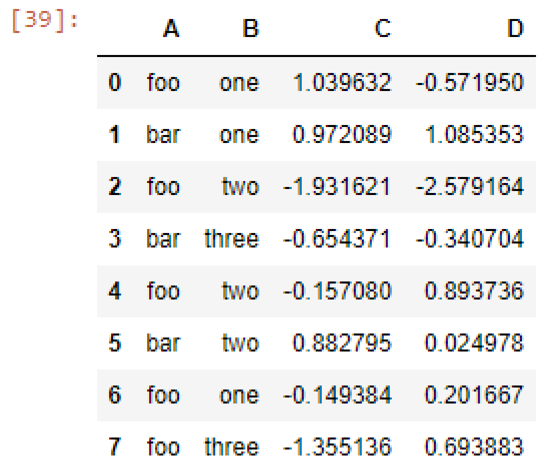
Grouping

By “group by” we are referring to a process involving one or more of the following steps:

- Splitting the data into groups based on some criteria
- Applying a function to each group independently
- Combining the results into a data structure

```
psdf = ps.DataFrame({'A': ['foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'foo’],
                     'B': ['one', 'one', 'two', 'three', 'two', 'two', 'one', 'three’],
                     'C': np.random.randn(8),
                     'D': np.random.randn(8)})
```

```
psdf
```



```
[39]:
       A      B          C          D
   0  foo    one   1.039632  -0.571950
   1  bar    one   0.972089   1.085353
   2  foo    two  -1.931621  -2.579164
   3  bar  three  -0.654371  -0.340704
   4  foo    two  -0.157080   0.893736
   5  bar    two   0.882795   0.024978
   6  foo    one  -0.149384   0.201667
   7  foo  three  -1.355136   0.693883
```

Grouping

Grouping and then applying the sum() function to the resulting groups.

```
psdf.groupby('A').sum()
```

[40]:

A	C	D
bar	1.200513	0.769627
foo	-2.553589	-1.361828

Grouping by multiple columns forms a hierarchical index, and again we can apply the sum function.

```
psdf.groupby(['A', 'B']).sum()
```

[41]:

A	B	C	D
foo	one	0.890248	-0.370283
	two	-2.088701	-1.685428
bar	three	-0.654371	-0.340704
foo	three	-1.355136	0.693883
bar	two	0.882795	0.024978
	one	0.972089	1.085353

Plotting

On a DataFrame, the plot() method is a convenience to plot all of the columns with labels:

```
pser = pd.Series(np.random.randn(1000),
           index=pd.date_range('1/1/2000', periods=1000))
```

```
psser = ps.Series(pser)
```

```
psser = psser.cummax()
```

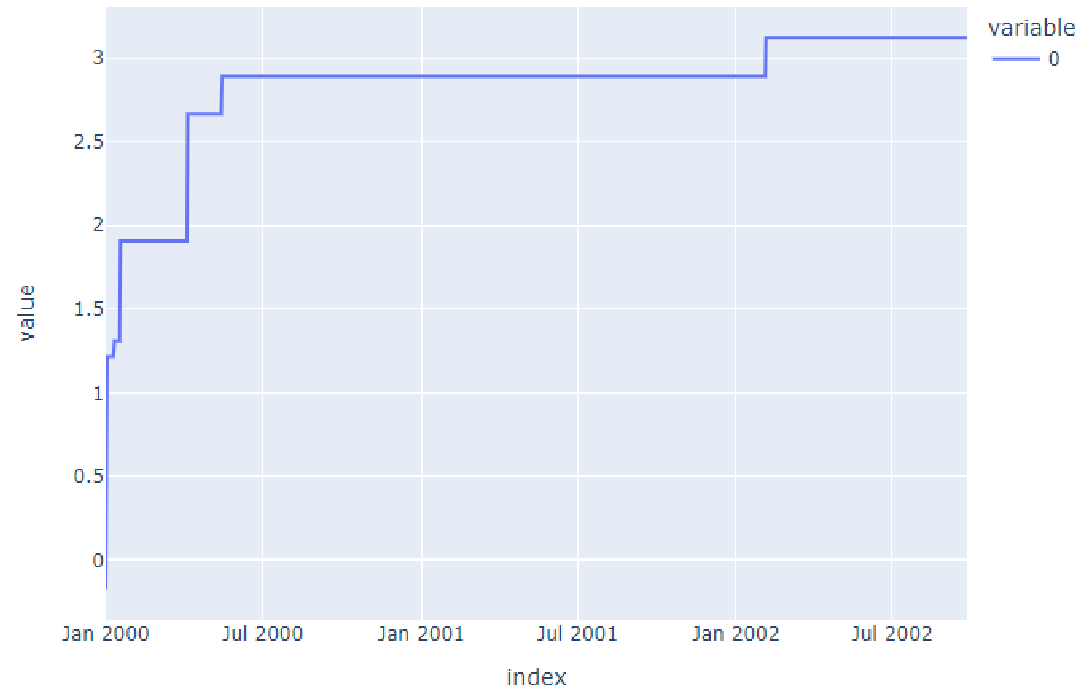
```
psser.plot()
```

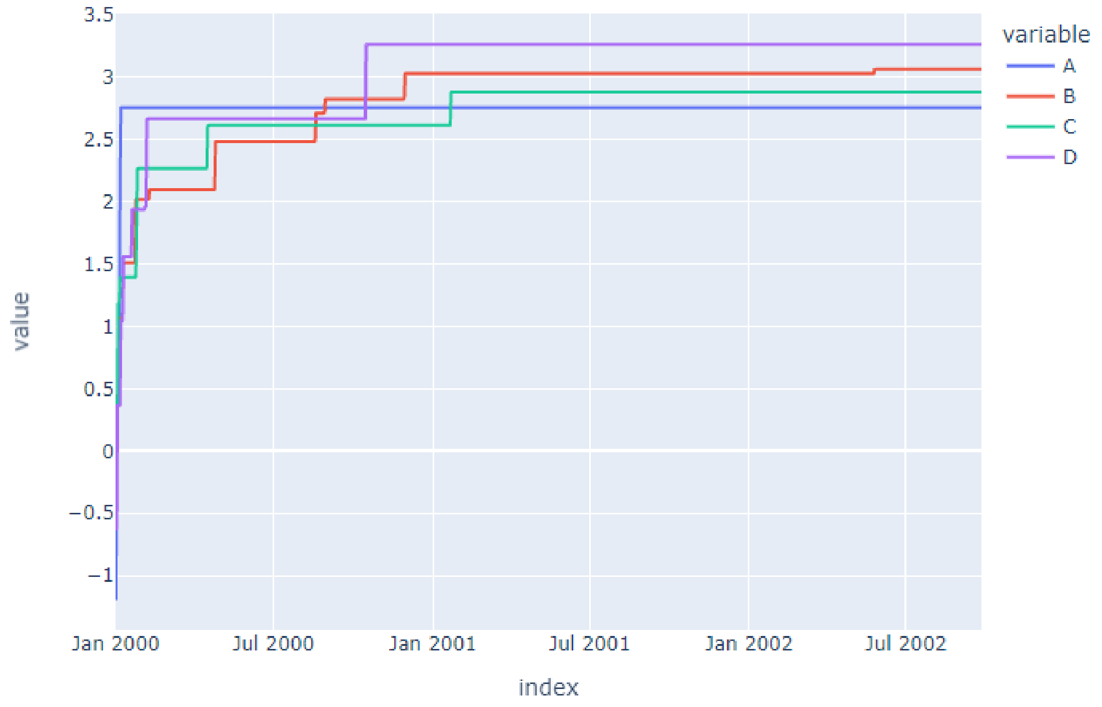
```
pdf = pd.DataFrame(np.random.randn(1000, 4), index=pser.index,
           columns=['A', 'B', 'C', 'D'])
```

```
psdf = ps.from_pandas(pdf)
```

```
psdf = psdf.cummax()
```

```
psdf.plot()
```





Getting data in/out

CSV is straightforward and easy to use.

```
psdf.to_csv('foo.csv') ps.read_csv('foo.csv').head(10)
```

[50]:

	A	B	C	D
0	-1.187097	-0.134645	0.377094	-0.627217
1	0.331741	0.166218	0.377094	-0.627217
2	0.331741	0.439450	0.377094	0.365970
3	0.621620	0.439450	1.190180	0.365970
4	0.621620	0.439450	1.190180	0.365970
5	2.169198	1.069183	1.395642	0.365970
6	2.755738	1.069183	1.395642	1.045868
7	2.755738	1.069183	1.395642	1.045868
8	2.755738	1.069183	1.395642	1.045868
9	2.755738	1.508732	1.395642	1.556933

Parquet is an efficient and compact file format to read and write faster.

```
psdf.to_parquet('bar.parquet') ps.read_parquet('bar.parquet').head(10)
```

[51]:

	A	B	C	D
0	-1.187097	-0.134645	0.377094	-0.627217
1	0.331741	0.166218	0.377094	-0.627217
2	0.331741	0.439450	0.377094	0.365970
3	0.621620	0.439450	1.190180	0.365970
4	0.621620	0.439450	1.190180	0.365970
5	2.169198	1.069183	1.395642	0.365970
6	2.755738	1.069183	1.395642	1.045868
7	2.755738	1.069183	1.395642	1.045868
8	2.755738	1.069183	1.395642	1.045868
9	2.755738	1.508732	1.395642	1.556933

In addition, pandas API on Spark fully supports Spark's various datasources such as ORC and an external datasource.

```
psdf.to_spark_io('zoo.orc', format="orc")
ps.read_spark_io('zoo.orc', format="orc").head(10)
```

[52]:

	A	B	C	D
0	-1.187097	-0.134645	0.377094	-0.627217
1	0.331741	0.166218	0.377094	-0.627217
2	0.331741	0.439450	0.377094	0.365970
3	0.621620	0.439450	1.190180	0.365970
4	0.621620	0.439450	1.190180	0.365970
5	2.169198	1.069183	1.395642	0.365970
6	2.755738	1.069183	1.395642	1.045868
7	2.755738	1.069183	1.395642	1.045868
8	2.755738	1.069183	1.395642	1.045868
9	2.755738	1.508732	1.395642	1.556933

Testing PySpark https://spark.apache.org/docs/latest/api/python/getting_started/testing_pyspark.html

Build a PySpark Application

Here is an example for how to start a PySpark application. Feel free to skip to the next section, “Testing your PySpark Application,” if you already have an application you’re ready to test.

First, start your Spark Session.

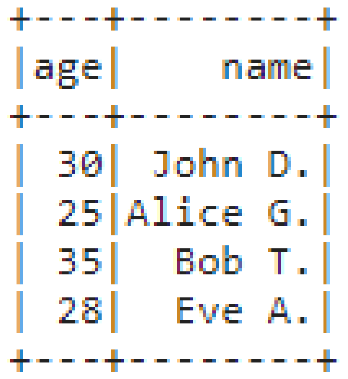
```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
# Create a SparkSession
spark = SparkSession.builder.appName("Testing PySpark Example").getOrCreate()
```

Next, create a DataFrame.

```python
sample_data = [{"name": "John D.", "age": 30},
          {"name": "Alice G.", "age": 25},
          {"name": "Bob T.", "age": 35},
          {"name": "Eve A.", "age": 28}]
df = spark.createDataFrame(sample_data)
```

Now, let’s define and apply a transformation function to our DataFrame.

```python
from pyspark.sql.functions import col, regexp_replace
# Remove additional spaces in name
def remove_extra_spaces(df, column_name):
          # Remove extra spaces from the specified column
          df_transformed = df.withColumn(column_name, regexp_replace(col(column_name), "\\s+", " "))
          return df_transformed
transformed_df = remove_extra_spaces(df, "name")
transformed_df.show()
```



```
+---+--------+
|age|    name|
+---+--------+
| 30| John D.|
| 25|Alice G.|
| 35|  Bob T.|
| 28|  Eve A.|
+---+--------+
```

Testing your PySpark Application

Now let’s test our PySpark transformation function.
One option is to simply eyeball the resulting DataFrame. However, this can be impractical for large DataFrame or input sizes.
A better way is to write tests. Here are some examples of how we can test our code. The examples below apply for Spark 3.5 and above versions.
Note that these examples are not exhaustive, as there are many other test framework alternatives which you can use instead of unittest or pytest. The built-in PySpark testing util functions are standalone, meaning they can be compatible with any test framework or CI test pipeline.

Option 1: Using Only PySpark Built-in Test Utility Functions

For simple ad-hoc validation cases, PySpark testing utils like assertDataFrameEqual and assertSchemaEqual can be used in a standalone context. You could easily test PySpark code in a notebook session. For example, say you want to assert equality between two DataFrames:

```python
import pyspark.testing
from pyspark.testing.utils import assertDataFrameEqual
# Example 1
df1 = spark.createDataFrame(data=[("1", 1000), ("2", 3000)], schema=["id", "amount"])
df2 = spark.createDataFrame(data=[("1", 1000), ("2", 3000)], schema=["id", "amount"])
assertDataFrameEqual(df1, df2) # pass, DataFrames are identical
```

```python
# Example 2
df1 = spark.createDataFrame(data=[("1", 0.1), ("2", 3.23)], schema=["id", "amount"])
df2 = spark.createDataFrame(data=[("1", 0.109), ("2", 3.23)], schema=["id", "amount"])
assertDataFrameEqual(df1, df2, rtol=1e-1) # pass, DataFrames are approx equal by rtol
```

You can also simply compare two DataFrame schemas:

```python
from pyspark.testing.utils import assertSchemaEqual
from pyspark.sql.types import StructType, StructField, ArrayType, DoubleType
s1 = StructType([StructField("names", ArrayType(DoubleType(), True), True)])
s2 = StructType([StructField("names", ArrayType(DoubleType(), True), True)])
assertSchemaEqual(s1, s2) # pass, schemas are identical
```

Option 2: Using Unit Test

For more complex testing scenarios, you may want to use a testing framework.

One of the most popular testing framework options is unit tests. Let's walk through how you can use the built-in Python unittest library to write PySpark tests. For more information about the unittest library, see here: https://docs.python.org/3/library/unittest.html.

First, you will need a Spark session. You can use the @classmethod decorator from the unittest package to take care of setting up and tearing down a Spark session.

```
import unittest
class PySparkTestCase(unittest.TestCase):
          @classmethod
          def setUpClass(cls):
                    cls.spark = SparkSession.builder.appName("Testing PySpark Example").getOrCreate()
          @classmethod
          def tearDownClass(cls):
                    cls.spark.stop()
```

Option 2: Using Unit Test

Now let’s write a unittest class.

```
from pyspark.testing.utils import assertDataFrameEqual
class TestTranformation(PySparkTestCase):
          def test_single_space(self):
                    sample_data = [{"name": "John D.", "age": 30},
                              {"name": "Alice G.", "age": 25},
                              {"name": "Bob T.", "age": 35},
                              {"name": "Eve A.", "age": 28}]
          # Create a Spark DataFrame
          original_df = spark.createDataFrame(sample_data)
          # Apply the transformation function from before
          transformed_df = remove_extra_spaces(original_df, "name")
          expected_data = [{"name": "John D.", "age": 30},
          {"name": "Alice G.", "age": 25},
          {"name": "Bob T.", "age": 35},
          {"name": "Eve A.", "age": 28}]
          expected_df = spark.createDataFrame(expected_data)
          assertDataFrameEqual(transformed_df, expected_df)
```

When run, unittest will pick up all functions with a name beginning with “test.”

Option 3: Using Pytest

We can also write our tests with pytest, which is one of the most popular Python testing frameworks.

Using a pytest fixture allows us to share a spark session across tests, tearing it down when the tests are complete.

```
import pytest
@pytest.fixture
def spark_fixture():
          spark = SparkSession.builder.appName("Testing PySpark Example").getOrCreate()
          yield spark
```

Option 3: Using Pytest

We can then define our tests like this:

```
import pytest
from pyspark.testing.utils import assertDataFrameEqual
def test_single_space(spark_fixture):
          sample_data = [{"name": "John D.", "age": 30},
                     {"name": "Alice G.", "age": 25},
                     {"name": "Bob T.", "age": 35},
                     {"name": "Eve A.", "age": 28}]
          # Create a Spark DataFrame
          original_df = spark.createDataFrame(sample_data)
          # Apply the transformation function from before
          transformed_df = remove_extra_spaces(original_df, "name")
          expected_data = [{"name": "John D.", "age": 30},
          {"name": "Alice G.", "age": 25},
          {"name": "Bob T.", "age": 35},
          {"name": "Eve A.", "age": 28}]
          expected_df = spark.createDataFrame(expected_data)
          assertDataFrameEqual(transformed_df, expected_df)
```

When you run your test file with the pytest command, it will pick up all functions that have their name beginning with "test."

Putting It All Together!

Let’s see all the steps together, in a Unit Test example.

```
# pkg/etl.py
import unittest
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from pyspark.sql.functions import regexp_replace
from pyspark.testing.utils import assertDataFrameEqual
# Create a SparkSession
spark = SparkSession.builder.appName("Sample PySpark ETL").getOrCreate()
sample_data = [{"name": "John D.", "age": 30},
           {"name": "Alice G.", "age": 25},
           {"name": "Bob T.", "age": 35},
           {"name": "Eve A.", "age": 28}]
df = spark.createDataFrame(sample_data)
# Define DataFrame transformation function
def remove_extra_spaces(df, column_name):
           # Remove extra spaces from the specified column using regexp_replace
           df_transformed = df.withColumn(column_name, regexp_replace(col(column_name), "\\s+", " "))
           return df_transformed
```

```python
# pkg/test_etl.py
import unittest from pyspark.sql import SparkSession
# Define unit test base class
class PySparkTestCase(unittest.TestCase):
        @classmethod
        def setUpClass(cls):
                cls.spark = SparkSession.builder.appName("Sample PySpark ETL").getOrCreate()
        @classmethod
        def tearDownClass(cls):
                cls.spark.stop()
# Define unit test
class TestTranformation(PySparkTestCase):
        def test_single_space(self):
                sample_data = [{"name": "John D.", "age": 30},
                        {"name": "Alice G.", "age": 25},
                        {"name": "Bob T.", "age": 35},
                        {"name": "Eve A.", "age": 28}]
                # Create a Spark DataFrame
                original_df = spark.createDataFrame(sample_data)
                # Apply the transformation function from before
                transformed_df = remove_extra_spaces(original_df, "name")
                expected_data = [{"name": "John D.", "age": 30},
                        {"name": "Alice G.", "age": 25},
                        {"name": "Bob T.", "age": 35},
                        {"name": "Eve A.", "age": 28}]
                expected_df = spark.createDataFrame(expected_data)
                assertDataFrameEqual(transformed_df, expected_df)
unittest.main(argv=[''], verbosity=0, exit=False)
```

```
Ran 1 test in 1.734s

OK
```

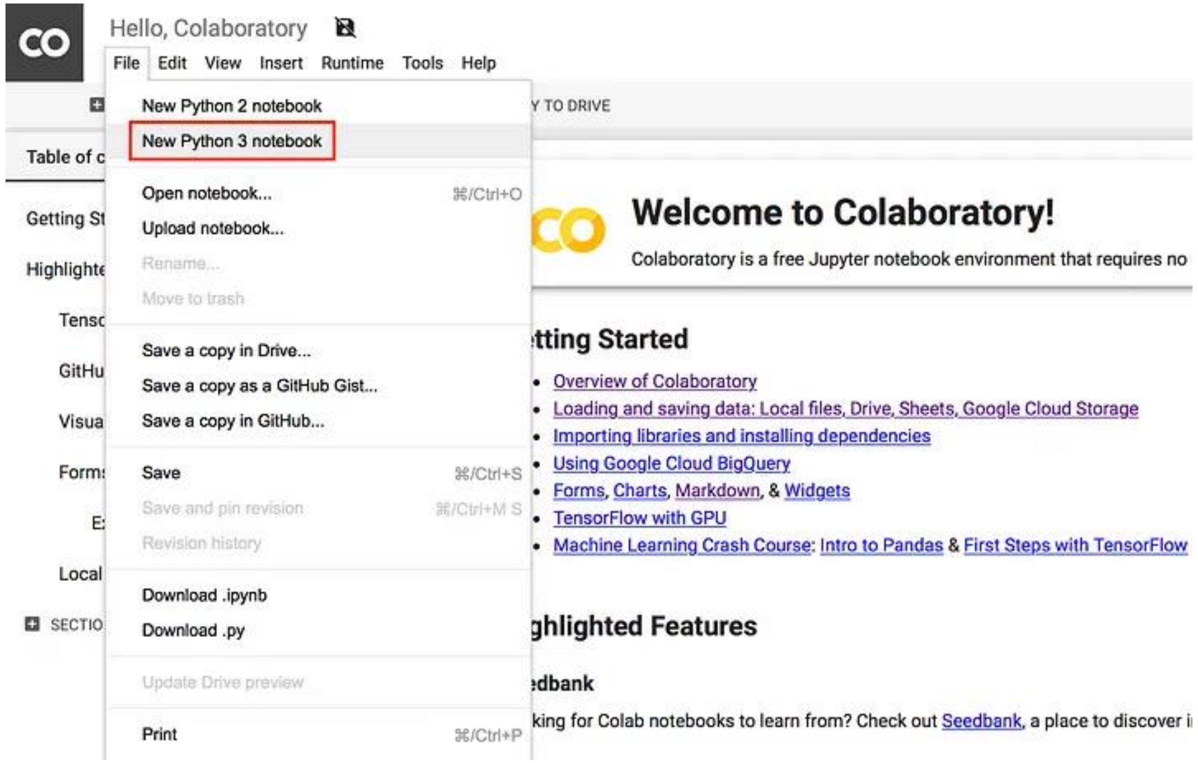
```
[27]: <unittest.main.TestProgram at 0x174539db0>
```

Example 2

1. Ruing PySpark on Google Colaboratory
2. Spark Walmart Data Analysis

Ruing PySpark on Google Colaboratory

Step 1：首先開啟 **Colaboratory** 建立一個新的 **Python 3 notebook**。



Welcome to Colaboratory!

Colaboratory is a free Jupyter notebook environment that requires no

tting Started

- Overview of Colaboratory
- Loading and saving data: Local files, Drive, Sheets, Google Cloud Storage
- Importing libraries and installing dependencies
- Using Google Cloud BigQuery
- Forms, Charts, Markdown, & Widgets
- TensorFlow with GPU
- Machine Learning Crash Course: Intro to Pandas & First Steps with TensorFlow

ghlighted Features

edbank

king for Colab notebooks to learn from? Check out Seedbank, a place to discover i

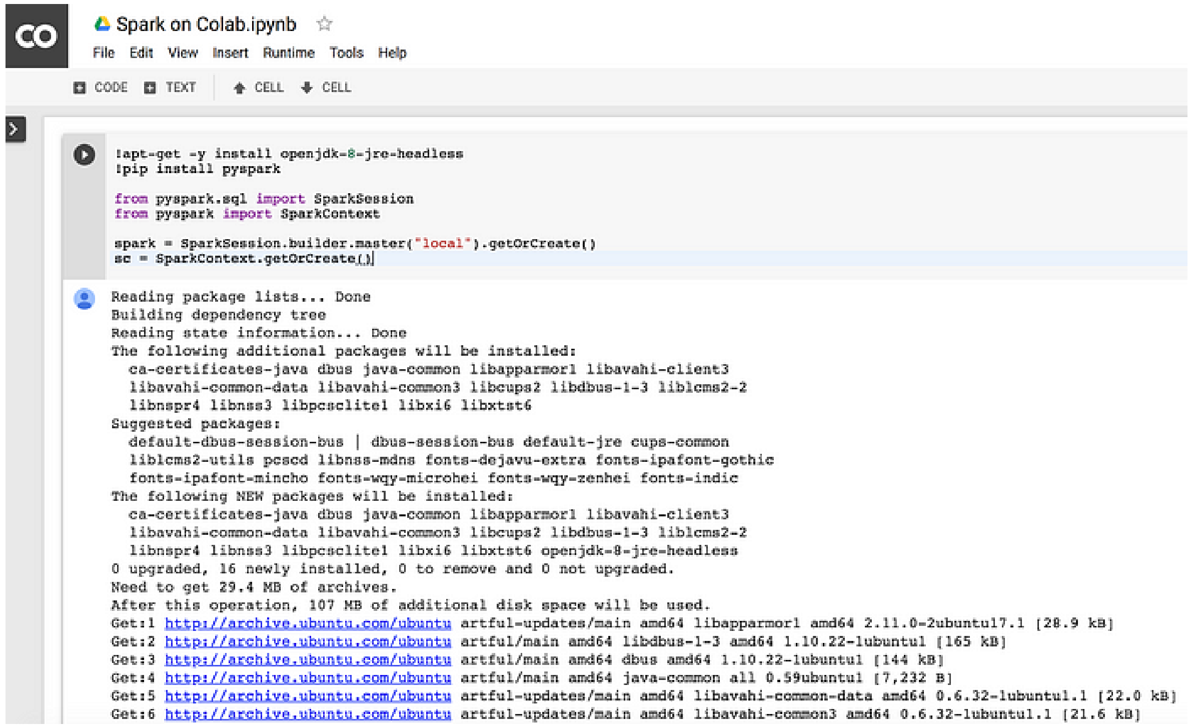
https://medium.com/@chiayinchen/%E4%BD%BF%E7%94%A8-google-colaboratory-%E8%B7%91-pyspark-625a07c75000

若還沒建立 Colaboratory 應用程式連結可以參考這篇文章:

https://medium.com/sparkamplab/%E6%B8%9B%E8%BC%95%E5%BB%BA%E7%AB%8B-python-%E9%81%8B%E8%A1%8C%E7%92%B0%E5%A2%83%E7%9A%84%E7%85%A9%E6%83%B1-824b150deaba

Step 2：安裝跑 PySpark 的環境。

```
!apt-get -y install openjdk-8-jre-headless
!pip install pyspark

from pyspark.sql import SparkSession
from pyspark import SparkContext

spark = SparkSession.builder.master("local").getOrCreate()
sc = SparkContext.getOrCreate()
```

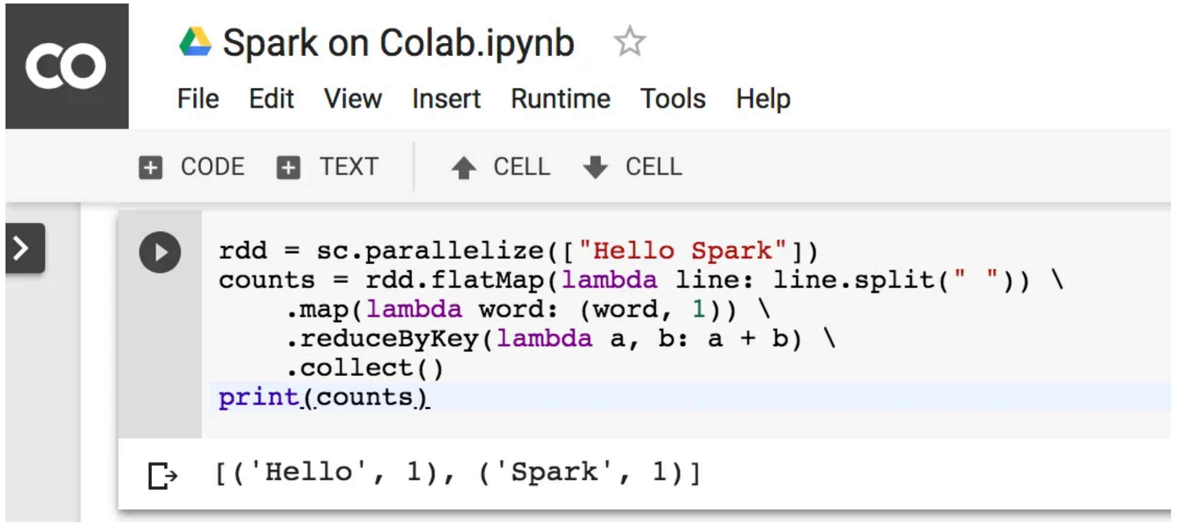


```
!apt-get -y install openjdk-8-jre-headless
!pip install pyspark

from pyspark.sql import SparkSession
from pyspark import SparkContext

spark = SparkSession.builder.master("local").getOrCreate()
sc = SparkContext.getOrCreate()
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  ca-certificates-java dbus java-common libapparmor1 libavahi-client3
  libavahi-common-data libavahi-common3 libcups2 libdbus-1-3 liblcms2-2
  libnspr4 libnss3 libpcsclite1 libxi6 libxtst6
Suggested packages:
  default-dbus-session-bus | dbus-session-bus default-jre cups-common
  liblcms2-utils pcscd libnss-mdns fonts-dejavu-extra fonts-ipafont-gothic
  fonts-ipafont-mincho fonts-wqy-microhei fonts-wqy-zenhei fonts-indic
The following NEW packages will be installed:
  ca-certificates-java dbus java-common libapparmor1 libavahi-client3
  libavahi-common-data libavahi-common3 libcups2 libdbus-1-3 liblcms2-2
  libnspr4 libnss3 libpcsclite1 libxi6 libxtst6 openjdk-8-jre-headless
0 upgraded, 16 newly installed, 0 to remove and 0 not upgraded.
Need to get 29.4 MB of archives.
After this operation, 107 MB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu artful-updates/main amd64 libapparmor1 amd64 2.11.0-2ubuntu17.1 [28.9 kB]
Get:2 http://archive.ubuntu.com/ubuntu artful/main amd64 libdbus-1-3 amd64 1.10.22-1ubuntu1 [165 kB]
Get:3 http://archive.ubuntu.com/ubuntu artful/main amd64 dbus amd64 1.10.22-1ubuntu1 [144 kB]
Get:4 http://archive.ubuntu.com/ubuntu artful/main amd64 java-common all 0.59ubuntu1 [7,232 B]
Get:5 http://archive.ubuntu.com/ubuntu artful-updates/main amd64 libavahi-common-data amd64 0.6.32-1ubuntu1.1 [22.0 kB]
Get:6 http://archive.ubuntu.com/ubuntu artful-updates/main amd64 libavahi-common3 amd64 0.6.32-1ubuntu1.1 [21.6 kB]
```

Step 3：使用 Word Count 的範例程式，驗證 PySpark 環境是否可以

```
rdd = sc.parallelize(["Hello Spark"])
counts = rdd.flatMap(lambda line: line.split(" ")) \
            .map(lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a + b) \
            .collect()
print(counts)
```

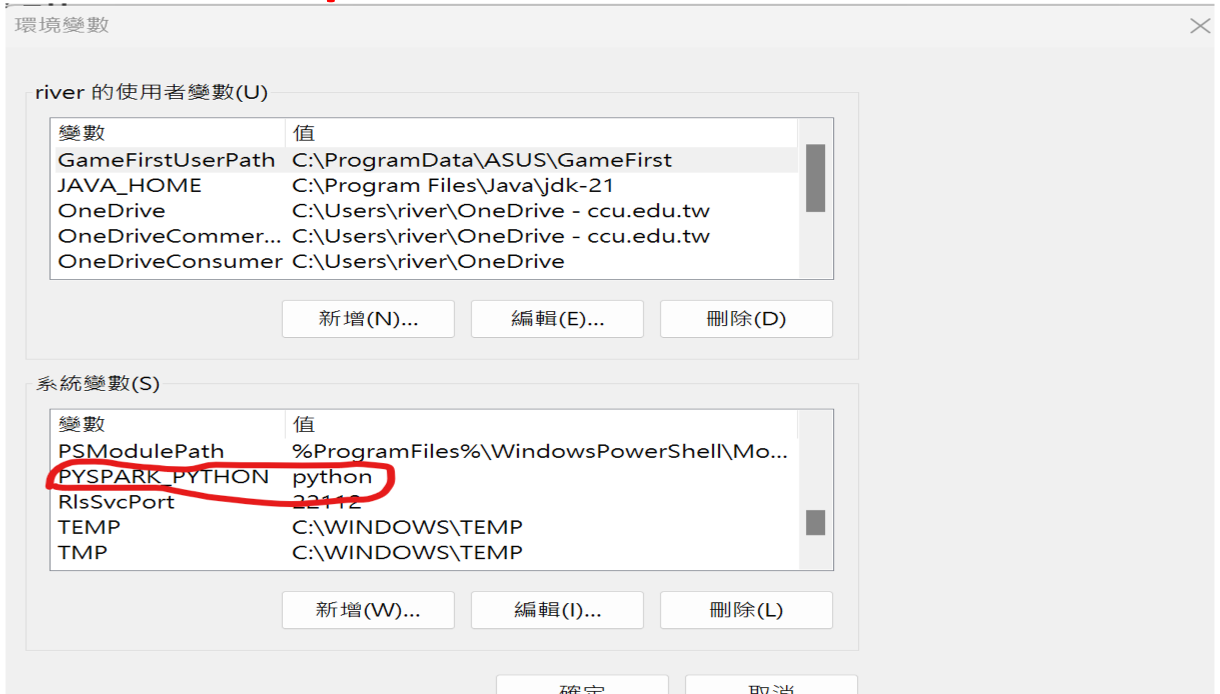


```
rdd = sc.parallelize(["Hello Spark"])
counts = rdd.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b) \
    .collect()
print(counts)
```

```
[('Hello', 1), ('Spark', 1)]
```

Set The PYSPARK_PYTHON Path to The Environment Variable

- Windows
  - In Search, search for and then select: System (Control Panel)
  - Click the **Advanced system settings** link
  - Click **Environment Variables**. In the section **System Variables**, click **New** to add a variable **PYSPARK_PYTHON** **and set its value as python**
  - **Restart your IDEs or computer!**



river 的使用者變數(U)

變數	值
GameFirstUserPath	C:\ProgramData\ASUS\GameFirst
JAVA_HOME	C:\Program Files\Java\jdk-21
OneDrive	C:\Users\river\OneDrive - ccu.edu.tw
OneDriveCommer...	C:\Users\river\OneDrive - ccu.edu.tw
OneDriveConsumer	C:\Users\river\OneDrive

系統變數(S)

變數	值
PSModulePath	%ProgramFiles%\WindowsPowerShell\Mo...
PYSPARK_PYTHON	python
RlsSvcPort	22112
TEMP	C:\WINDOWS\TEMP
TMP	C:\WINDOWS\TEMP

Spark Walmart Data Analysis Project Exercise

!!如果CoLab找不到findspark要下這個指令:

```
pip install findspark
```

1.Start a simple Spark Session

```
# import libraries
import findspark
findspark.init('/Users/pratikajitb/server/spark-2.4.0-bin-hadoop2.7')

from pyspark.sql import SparkSession

# creating the spark session
spark = SparkSession.builder.appName('walmart').getOrCreate()
```

2.Load the Walmart Stock CSV File, have Spark infer the data types.

```
df = spark.read.csv('walmart_stock.csv', inferSchema=True, header=True)
```

Csv檔連結: https://github.com/pratikbarjatya/spark-walmart-data-analysis-exercise/blob/master/walmart_stock.csv

github連結: https://github.com/pratikbarjatya/spark-walmart-data-analysis-exercise/blob/master/Spark%20Walmart%20Data%20Analysis%20Project.ipynb

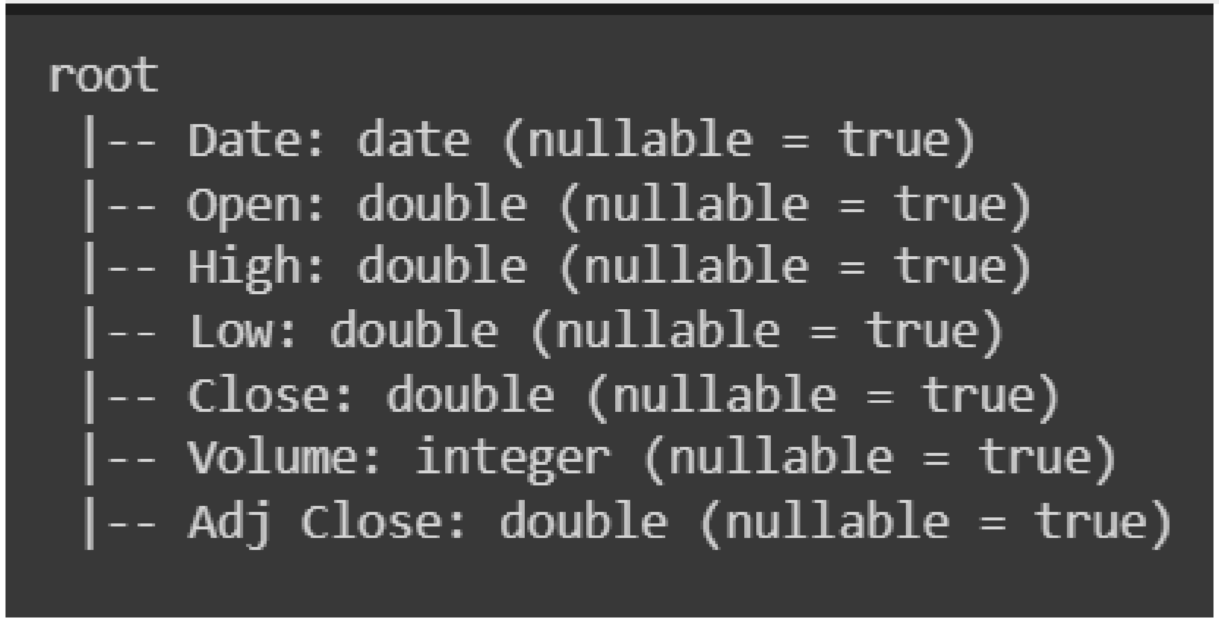
What are the column names?

```
df.columns
```

```
['Date', 'Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close']
```

What does the Schema look like?

```
df.printSchema()
```



```
root
 |-- Date: date (nullable = true)
 |-- Open: double (nullable = true)
 |-- High: double (nullable = true)
 |-- Low: double (nullable = true)
 |-- Close: double (nullable = true)
 |-- Volume: integer (nullable = true)
 |-- Adj Close: double (nullable = true)
```

Print out the first 5 columns.

```python
for line in df.head(5):
        print(line, '\n')
```

```
Row(Date=datetime.date(2012, 1, 3), Open=59.970001, High=61.060001, Low=59.869999, Close=60.330002, Volume=12668800, Adj Close=52.619234999999996)

Row(Date=datetime.date(2012, 1, 4), Open=60.209998999999996, High=60.349998, Low=59.470001, Close=59.709998999999996, Volume=9593300, Adj Close=52.078475)

Row(Date=datetime.date(2012, 1, 5), Open=59.349998, High=59.619999, Low=58.369999, Close=59.419998, Volume=12768200, Adj Close=51.825539)

Row(Date=datetime.date(2012, 1, 6), Open=59.419998, High=59.450001, Low=58.869999, Close=59.0, Volume=8069400, Adj Close=51.45922)

Row(Date=datetime.date(2012, 1, 9), Open=59.029999, High=59.549999, Low=58.919998, Close=59.18, Volume=6679300, Adj Close=51.616215000000004)
```

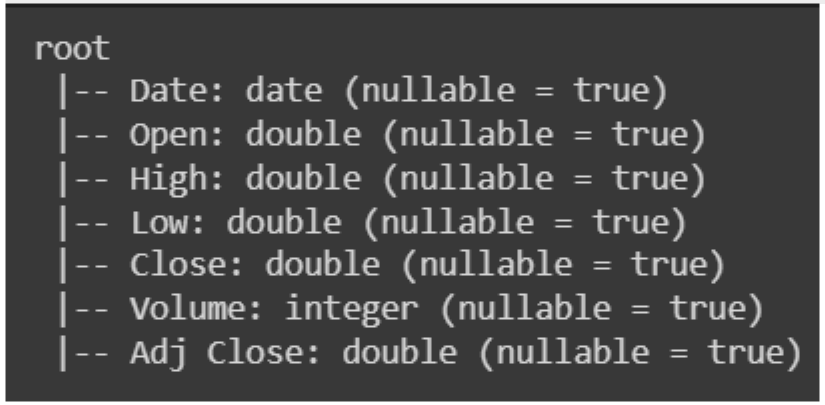
Use describe() to learn about the DataFrame.

```python
df.describe().show()
```

```
+-------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+
|summary|              Open|             High|              Low|            Close|           Volume|        Adj Close|
+-------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+
|  count|              1258|             1258|             1258|             1258|             1258|             1258|
|   mean| 72.35785375357709|72.83938807631165| 71.9186009594594|72.38844998012726|8222093.481717011|67.23883848728146|
| stddev|  6.76809024470826|6.768186808159218|6.744075756255496|6.756859163732991|  4519780.8431556|6.722609449996857|
|    min|56.389998999999996|        57.060001|        56.299999|        56.419998|          2094900|        50.363689|
|    max|         90.800003|        90.970001|            89.25|        90.470001|         80898100|84.91421600000001|
+-------+------------------+-----------------+-----------------+-----------------+-----------------+-----------------+
```

There are too many decimal places for mean and stddev in the describe() dataframe. Format the numbers to just show up to two decimal places.

```python
'''
from pyspark.sql.types import (StructField, StringType, IntegerType, StructType)

data_schema = [StructField('summary', StringType(), True), StructField('Open', StringType(), True),
           StructField('High', StringType(), True), StructField('Low', StringType(), True),
           StructField('Close', StringType(), True), StructField('Volume', StringType(), True),
           StructField('Adj Close', StringType(), True) ]
final_struc = StructType(fields=data_schema)
'''
df = spark.read.csv('walmart_stock.csv', inferSchema=True, header=True)
df.printSchema()
#The schema given below is wrong, as it is mostly from an older version.
#Spark is able to predict the schema correctly now
```



```
root
 |-- Date: date (nullable = true)
 |-- Open: double (nullable = true)
 |-- High: double (nullable = true)
 |-- Low: double (nullable = true)
 |-- Close: double (nullable = true)
 |-- Volume: integer (nullable = true)
 |-- Adj Close: double (nullable = true)
```

```python
from pyspark.sql.functions import format_number

summary = df.describe()
summary.select(summary['summary'],
           format_number(summary['Open'].cast('float'), 2).alias('Open'),
           format_number(summary['High'].cast('float'), 2).alias('High'),
           format_number(summary['Low'].cast('float'), 2).alias('Low'),
           format_number(summary['Close'].cast('float'), 2).alias('Close'),
           format_number(summary['Volume'].cast('int'), 0).alias('Volume')
           ).show()
```

```
+-------+--------+--------+--------+--------+----------+
|summary|    Open|    High|     Low|   Close|    Volume|
+-------+--------+--------+--------+--------+----------+
|  count|1,258.00|1,258.00|1,258.00|1,258.00|     1,258|
|   mean|   72.36|   72.84|   71.92|   72.39| 8,222,093|
| stddev|    6.77|    6.77|    6.74|    6.76| 4,519,780|
|    min|   56.39|   57.06|   56.30|   56.42| 2,094,900|
|    max|   90.80|   90.97|   89.25|   90.47|80,898,100|
+-------+--------+--------+--------+--------+----------+
```

Create a new dataframe with a column called HV Ratio that is the ratio of the High Price versus volume of stock traded for a day.

```
df_hv = df.withColumn('HV Ratio', df['High']/df['Volume']).select(['HV Ratio'])
df_hv.show()
```

```
+--------------------+
|            HV Ratio|
+--------------------+
|4.819714653321546E-6|
|6.290848613094555E-6|
|4.669412994783916E-6|
|7.367338463826307E-6|
|8.915604778943901E-6|
|8.644477436914568E-6|
|9.351828421515645E-6|
| 8.29141562102703E-6|
|7.712212102001476E-6|
|7.071764823529412E-6|
|1.015495466386981E-5|
|6.576354146362592...|
| 5.90145296180676E-6|
|8.547679455011844E-6|
|8.420709512685392E-6|
|1.041448341728929...|
|8.316075414862431E-6|
|9.721183814992126E-6|
|8.029436027707578E-6|
|6.307432259386365E-6|
+--------------------+
only showing top 20 rows
```

What day had the Peak High in Price?

```
df.orderBy(df['High'].desc()).select(['Date']).head(1)[0]['Date']
```

```
datetime.date(2015, 1, 13)
```

How many days was the Close lower than 60 dollars?

```
df.filter(df['Close'] < 60).count()
```

```
81
```

What is the mean of the Close column?

```
from pyspark.sql.functions import mean
df.select(mean('Close')).show()
```

```
+-----------------+
|       avg(Close)|
+-----------------+
|72.38844998012726|
+-----------------+
```

What is the max and min of the Volume column?

```
from pyspark.sql.functions import min, max
df.select(max('Volume'),min('Volume')).show()
```

```
+-----------+-----------+
|max(Volume)|min(Volume)|
+-----------+-----------+
|   80898100|    2094900|
+-----------+-----------+
```

What percentage of the time was the High greater than 80 dollars ?
In other words, (Number of Days High>80)/(Total Days in the dataset)

```
df.filter('High > 80').count() * 100/df.count()
```

```
9.14149443561208З
```

What is the Pearson correlation between High and Volume?

```
df.corr('High', 'Volume')
```

```
-0.3384326061737161
```

```
from pyspark.sql.functions import corr
df.select(corr(df['High'], df['Volume'])).show()
```

```
+-------------------+
| corr(High, Volume)|
+-------------------+
|-0.3384326061737161|
+-------------------+
```

What is the max High per year?

```python
from pyspark.sql.functions import (dayofmonth, hour,
                                    dayofyear, month,
                                    year, weekofyear,
                                    format_number, date_format)
year_df = df.withColumn('Year', year(df['Date']))
year_df.groupBy('Year').max()['Year', 'max(High)'].show()
```

```
+----+---------+
|Year|max(High)|
+----+---------+
|2015|90.970001|
|2013|81.370003|
|2014|88.089996|
|2012|77.599998|
|2016|75.190002|
+----+---------+
```

What is the average Close for each Calendar Month?

```python
#Create a new column Month from existing Date column
month_df = df.withColumn('Month', month(df['Date’ ]))

#Group by month and take average of all other columns
month_df = month_df.groupBy('Month').mean()

#Sort by month
month_df = month_df.orderBy('Month’ )

#Display only month and avg(Close), the desired columns
month_df['Month', 'avg(Close)'].show()
```

```
+-----+-----------------+
|Month|       avg(Close)|
+-----+-----------------+
|    1|71.44801958415842|
|    2|  71.306804443299|
|    3|71.77794377570092|
|    4|72.97361900952382|
|    5|72.30971688679247|
|    6| 72.4953774245283|
|    7|74.43971943925233|
|    8|73.02981855454546|
|    9|72.18411785294116|
|   10|71.57854545454543|
|   11| 72.1110893069307|
|   12|72.84792478301885|
+-----+-----------------+
```

ANY
QUESTIONS
?