Rockborne

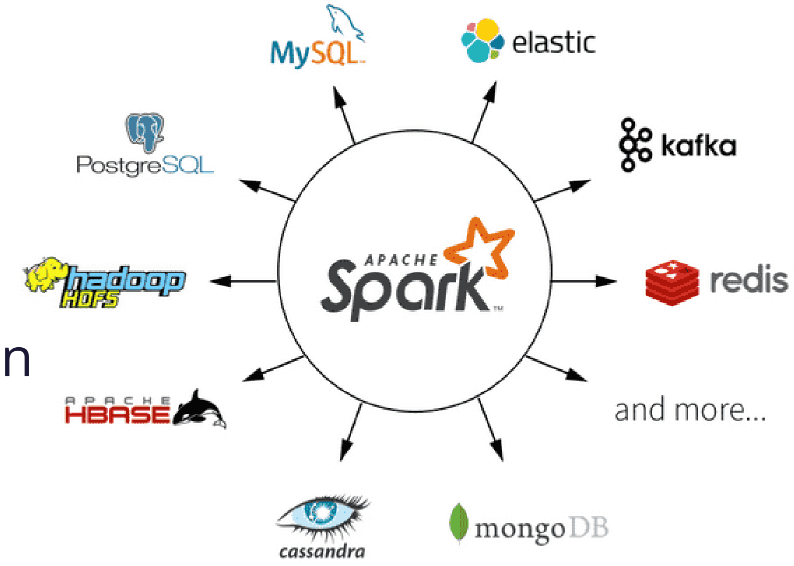
PySpark Fundamentals: Setup, Essentials, and Data Transformation

Workshop Modules

- PySpark Installation
- PySpark Essentials
    - Introduction to PySpark
    - Setting up the environment
    - Basic operations
    - RDDs and Data frames
    - Data frames and SQL
- Data Transformation with PySpark
    - Filtering
    - Manipulation
    - Aggregation
    - JOINs and UNION

Apache Spark

- Apache Spark is an open source, distributed computing system
- It processes large-scale data across multiple machines (clusters)
- Provides in-memory computation
  - Tasks are completed in place in memory, increasing energy efficiency
- Supports multiple programming languages



Big Data Analytics

Data Engineering/ETL

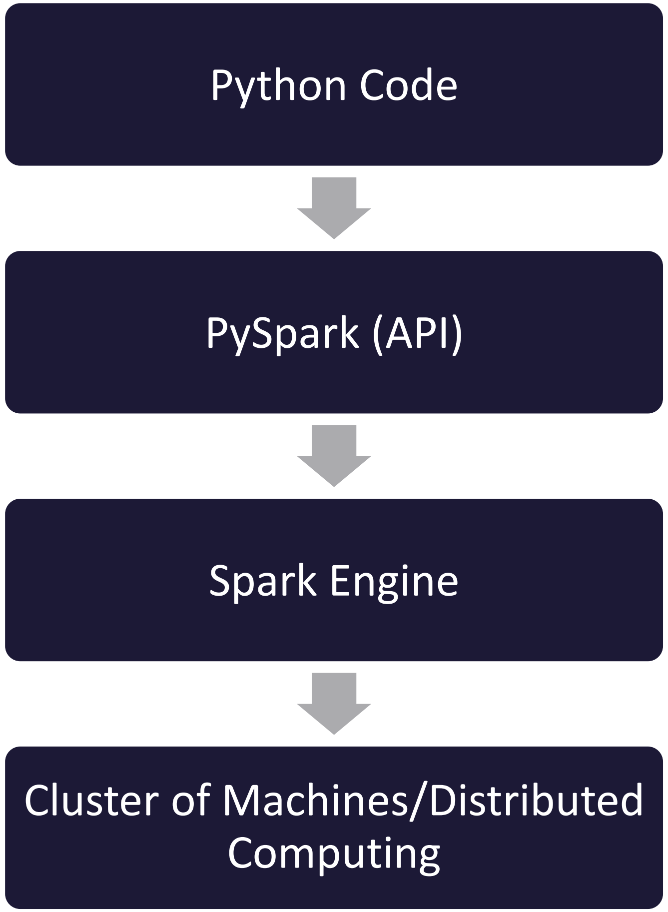
Real-Time Stream Processing

ML (MLlib)

Graph Processing

PySpark

- PySpark is the Python API for Apache Spark
- It allows you to write Spark applications using Python while still provides access to Spark's core features:
  - RDD API: low-level distributed data analysis
  - DataFrame API: high-level, optimised SQL-like tables
  - SQL API: query data with SWL syntax
- It integrates easily with the Python ecosystem (NumPy, Pandas, etc.)



PySpark Installation - Software

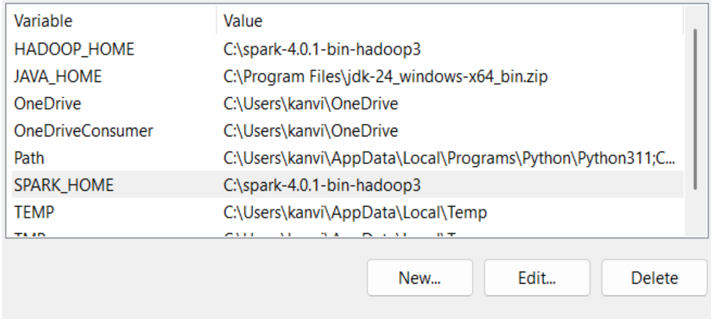
1. Download the Java Software Development Kit: Java SDK 8 | Oracle
    - Download the most compatible version (Kit 8)
    - You may have to create an account

2. Download Apache Spark: Downloads | Apache Spark
    - Download the latest version (4.0.1) and Package 2.4 and later
    - Place the folder in C:/

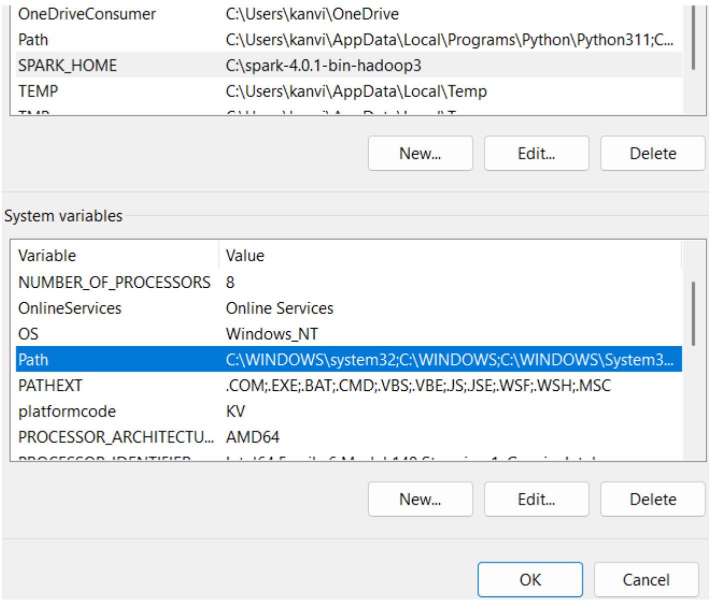
3. Download winutils: winutils | cdarlint
    - hadoop-2.7.7/bin
    - Copy the .exe file and place it into **C:\spark-4.0.1-bin-hadoop3\bin**

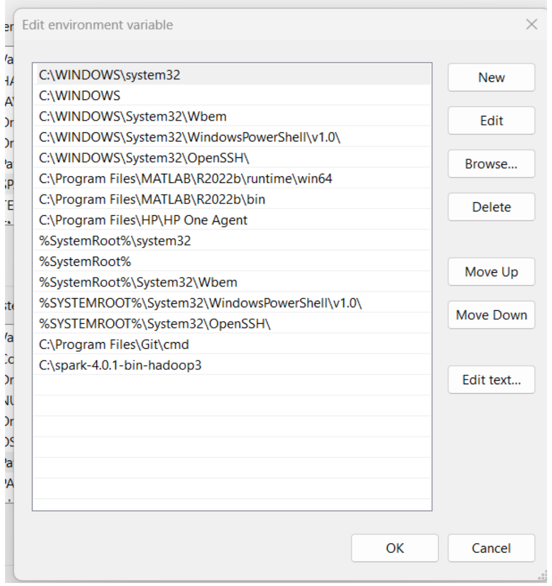
PySpark Installation – Environment Variables

1. Set up Environment Variables
    - Press Win + R > type 'sysdm.cpl' > hit Enter > go to the Advanced tab > click Environment Variables
    - In the System Variables section, click New
    - HADOOP_HOME: C:\spark-4.0.1-bin-hadoop3
    - JAVA_HOME: C:\Program Files\Java\jdk-24
    - SPARK_HOME: C:\spark-4.0.1-bin-hadoop3



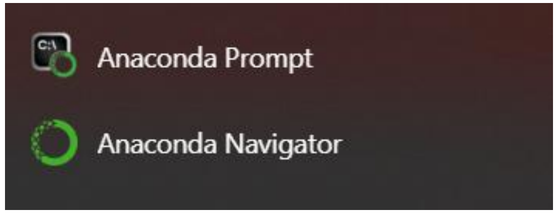
2. Add the PySpark file to PATH:
    - Select SPARK_HOME in User Variables, and double click on 'Path' in System Variables
    - Select 'New' and add your C:/Spark file

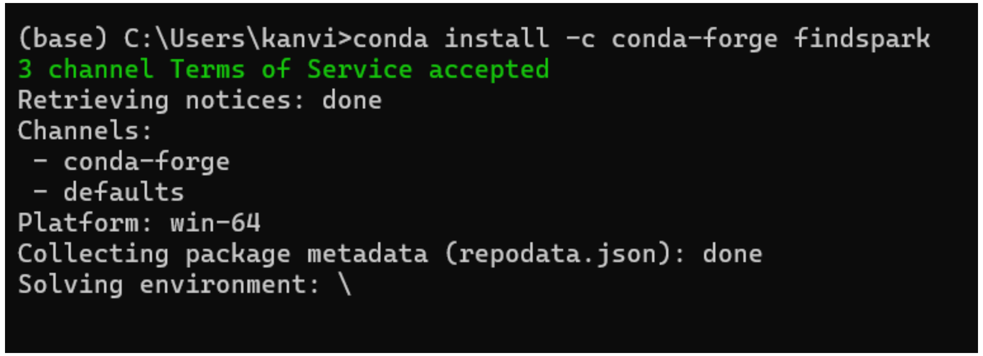




PySpark Installation - Validation

1. Ensure you have Anaconda Prompt installed onto your device, and enter the following:
    - conda install –c conda-forge findspark





2. Restart your computer

3. In Jupyter Notebook, validate the installation:
    - Import findspark
      findspark.init()
      findspark.find()
      >>> output should be your C:/Spark file
    - Import pyspark
    - If no errors are returned, the installation has been successful!

Dataset – superstore(in).csv

We will be using the **superstore(in).csv** dataset provided in the SharePoint file:

- Thousands of interactions – great for demos
- Columns for sales, profit, and discounts – numeric analysis
- Columns for customer, category, and region – categorical analysis
- Perfect for filtering, aggregations, and joins later on

Set up the environment by laoding the dataset:

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

PySpark Essentials:

- Basic Operations
- RDDs and DFs
- DFs and SQL

Why PySpark?

PySpark is the Python API for Apache Spark, allowing for distributed data processing:

- It handles big datasets across multiple machines
  - It splits large datasets into smaller chunks for parallel processing
  - Scales to terabytes of data across clusters
- Provides both low-level control (RDDs) and high-level abstraction (Dataframes)
  - RDDs: fine-grained control, custom transformations
  - Dataframes: SQL-like and user-friendly for most analytics
- Integrates easily with Python
  - Works in Jupyter Notebooks
  - Converts well between Spark and Pandas Dataframes
  - Enables end-to-end data analysis and ML pipelines

Basic Operations using PySpark

Loading and Inspecting data:

t

1. The first step is to always load your data into a Dataframe.
    - This allows for external datasets to be distributed and processed

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

2. Inspecting the data allows you to quickly explore and understand the data structure and quality
    - Ensures correct schema and makes catching data issues easy

```
df.show(5) ## Shows first 5 rows

df.printSchema() ## Print schema with column types

df.columns ##Column names
```

Basic Operations using PySpark

Dimensions, and Statistics

1. Unlike Pandas, Spark separates rows and columns
    - This is important for large datasets

```
df.count() ##Row count
len(df.columns) ##Column count
```

- Spark also allows you to get a quick summary of numerical fields for initial exploration

```
df.describe().show() ##Summary statistics for all numerical columns
```

2. Spark allows you to narrow down to the columns you need, and focus on subsets of data
    - This improves performance and readability
    - Common in data cleaning and analysis

```
df.select("Sales").show(5) ##Single column
df.select("Category", "Sales").show(5) ##Multiple columns
```

```
df.filter(df["Category"] == "Furniture").show(5) ##Equality filter
df.filter(df["Sales"] > 500).show(5) ##Numeric condition
```

RDDs and DataFrames using PySpark

Resilient Distributed Datasets (RDDs) and Transformations

- They are low-level distributed collection of objects , and are the foundation of Spark
    - They allow for custom transformations and fine-grained control

```
rdd = df.rdd ##Convert df to RDD
rdd.take(5) ##Take sample rows
```

- They transform data only when an action is called, allowing for parallel and optimised execution.
    - Without these actions being called, Spark won't compute anything

```
sales_rdd = rdd.map(lambda row: row.Sales)
high_sales = sales_rdd.filter(lambda x: float(x) > 500)
```

```
sales_rdd.collect() ##Collect as list
sales_rdd.take(5) ##First 5
rdd.count() ##Count rows
```

RDDs and DataFrames using PySpark

RDD Actions

- Actions trigger execution and return results, and without these, Spark won't compute anything

```
sales_rdd.collect() ##Collect as list

sales_rdd.take(5) ##First 5

rdd.count() ##Count rows
```

RDDs to Dataframes:

- RDDs are flexible, but dataframes are easier to use
  - They give SQL-like power and optimisations

```
from pyspark.sql import Row

df2 = rdd.map(lambda row: Row(Sales=row.Sales)).toDF() ##Convert RDD back to DF

df2.show(5)
```

RDDs and DataFrames using PySpark

Dataframes

- They are a distributed table with named columns (similar to Pandas DF)
    - They are the most common Spark API, with easy syntax and optimised execution

```
df.show(5) ##df already created from csv
df.printSchema() ##Check schema
```

- You can add, rename, or transform columns – all of which are core steps

```
from pyspark.sql.functions import col
df = df.withColumn("Sales_x2", col("Sales") * 2) ##Add a new column
df = df.withColumnRenamed("Sales", "Total_Sales") ##Rename column
```

Rockborne

RDDs and DataFrames using PySpark

Grouping and Aggregation

- Grouping and aggregation allows you to summarise by categories, which is essential for analysis and business ins

```
from pyspark.sql.functions import col
df = df.withColumn("Sales_x2", col("Sales") * 2) ##Add a new column
df = df.withColumnRenamed("Sales", "Total_Sales") ##Rename column
```

Using SQL with Dataframes

- Registering the dataframe as a table allows analysts to reuse SQL skills in Spark

```
df.createOrReplaceTempView("orders") ## Register table

spark.sql("""
    SELECT Category, SUM(Total_Sales)
    FROM orders
    GROUP BY Category
""").show() ## Run SQL query
```

RDDs and DataFrames using PySpark

Filtering

- You should always only keep rows that meet your project's conditions
  - This removes noise and allows you to focus on relevant data

```
From pyspark.sql.functions import col

Df.filter(col("Sales") > 500).show() ##Filter rows where sales > 500

Spark.sql("SELECT * FROM orders WHERE Sales > 500").show() ##SQL equivalent
```

Data Transformation:

- Filtering and Manipulation
- Aggregation
- JOINs and UNION

Data Transformation using PySpark

Why do we need to transform data?

t

- Raw data is messy and hard to interpret, so transformations allow analysts to clean, filter, and restructure it.
- It prepares datasets for analysis, ML, or reporting

Some of the ways data can be transformed in PySpark include:

  - Filtering and manipulation
  - Aggregation
  - JOINs and UNION

Data Transformation using PySpark

Manipulation

- Select columns that are needed, simplifying the dataset and reduces computation time

```
df.select("Category", "Sales", "Profit").show()
```