

**Week I**

# **Spark / PySpark**

**Intermediate Data Engineer in  
Azure**  
Trainer: Balazs Balogh



# Apache Spark

A fast, unified Engine for Big Data Processing.

Apache Spark is an **open-source analytics processing engine** designed for large-scale, efficient distributed data processing and machine learning applications.

Originally developed at the University of California, Berkeley, Spark was later donated to the Apache Software Foundation.

In February 2014, Spark became a top-level Apache project and has since become one of Apache's most active open-source projects, with contributions from thousands of engineers.



# Apache Spark – Key advantages

- **Speed:**

Thanks to its distributed memory management and optimized execution engine, Spark can be up to 100 times faster than Hadoop MapReduce for in-memory processing.

- **Scalability:**

Easily scalable to clusters with hundreds or thousands of machines.

- **Versatility:**

Supports diverse data processing modes: **Batch** processing, Real-time data **streaming**, **Machine learning** (via MLlib), Interactive querying (via **Spark SQL**)

- **Ease of Development:**

Provides APIs for multiple popular programming languages: Python (PySpark), Scala, Java, R, and SQL.

Uses **Resilient Distributed Datasets (RDDs)** to simplify distributed data processing while offering high-level abstractions.

Although RDDs are core to Spark, **Spark SQL** and **DataFrame APIs** are now more commonly used due to their advanced abstractions.

- **Resilient Distributed Dataset (RDD):**

A fundamental data structure in Apache Spark, serving as the backbone for distributed data processing.

Allows parallel data processing across clusters.

While RDDs form Spark Core's foundation, DataFrame and Spark SQL APIs are often preferred for their advanced functionality.

DataFrames are built on top of RDDs but provide additional structure and optimizations.

- **Fault Tolerance:**

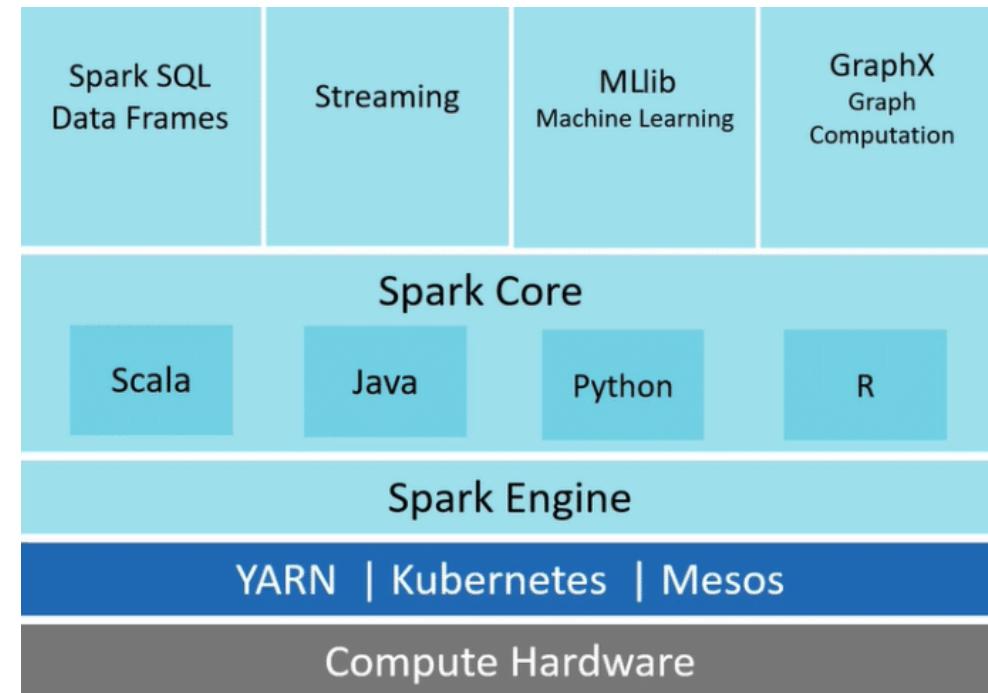
RDDs use a "lineage" mechanism for transformations, enabling automatic recomputation in case of failure, ensuring robustness.

- **Open Source:**

Free to use, with an active developer community continuously improving and extending the framework.

# Apache Spark – Spark ecosystem

- **Top layer:** Applications and libraries, such as: SparkSQL, Spark Streaming, Mllib, GraphX
- **Core layer:** The foundation of Spark, providing APIs in: Scala, Java, Python and R.
- **Spark engine:** Orchestrates and executes distributed tasks by breaking high-level computations into smaller tasks and optimizing their execution. It enhances performance with in-memory computation and integrates with cluster managers like YARN and Kubernetes for efficient resource utilization.
- **Cluster managers:** YARN, Kubernetes, Mesos manages resources for distributed computing.
- **Hardware layer:** The physical computing resources used for distributed processing.



# Apache Spark - Architecture

When you typically think of a "computer," you imagine a single machine on your desk at home or work. However, standalone machines lack the resources and power needed to compute vast amounts of data.

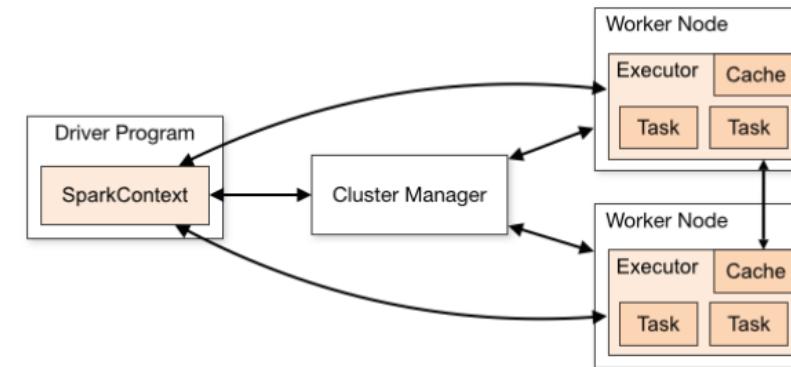
A cluster, which is a group of computers, combines the resources of multiple machines, allowing us to use their collective power as a single system.

Yet just having a group of machines isn't efficient enough, a framework is needed to coordinate the work among them. Spark does exactly that: it manages and orchestrates task execution on data within a cluster of computers.

A Spark application consists of **Driver** and **Executor** processes.

## How it works:

- The **Driver** starts the application and sends instructions to the Cluster Manager.
  - Responsibilities:
    - Keeping track of the Spark application's metadata.
    - Responding to user programs or inputs.
    - Analyzing, distributing, and scheduling tasks across the Executors.
- The **Cluster Manager** allocates Executors on the Worker Nodes.
- The **Executors** run the tasks and report results back to the Driver.



Source: <https://spark.apache.org/>

# Apache Spark - SparkSession

## SparkSession:

The central entry point for Spark applications, offering a unified API to manage and process data in various formats such as Parquet, JSON, and CSV.

Example of creating a SparkSession in Python:

```
from pyspark.sql import SparkSession

spark = (
    SparkSession
    .builder
    .appName("PySpark Tutorial")
    .master("local[*]")
    .getOrCreate()
)
```

# Apache Spark – RDD / DataSet / DataFrame

Spark provides three different data structures (**RDDs**, **Datasets** and **DataFrames**) each suited for different use cases:

- **DataFrames** are ideal for structured data and SQL-like operations.
- **RDDs** are more flexible and perform better in distributed computing environments.
- **Datasets** strike a balance between the two, supporting both structured data and distributed computing, making them versatile.

In most cases, you'll work with **DataFrames**, which we will also use in this course.

DataFrames are ideal for structured data and **SQL-like operations**. One significant advantage is that many people are already familiar with the syntax from **pandas**, making it easier to get accustomed to PySpark's syntax.

DataFrames are also performance-optimized due to the **Catalyst optimizer** and **Tungsten engine**, ensuring fast query execution. Compared to RDDs, they offer better abstraction, ease of use, and seamless SQL-like querying capabilities.

Manual creation of a Pyspark DataFrame:

```
data = [
    (1, "Alice", 20),
    (2, "Bob", 25),
    (3, "Charlie", 30),
    (4, "Robert", 33)
]
columns = ["id", "name", "age"]

df = spark.createDataFrame(data, columns)

df.show()
```

# Apache Spark - SparkSQL

For many Data Analysts, Data Scientists, or professionals working in business intelligence, **SQL** is the "native language" in the world of information technology.

**SparkSQL** is one of Spark's modules for data processing, designed for those who are more comfortable with SQL than with the other programming languages that Spark supports. With **SparkSQL**, you can take full advantage of all the benefits Spark has to offer.

Create a **temporary view** from the DataFrame first, then it can be queried:

```
taxi_df.createOrReplaceTempView("taxi_temp_view")
spark.sql("SELECT * FROM taxi_temp_view LIMIT 10").show()
```

As you are querying a **temporary view**, you cannot perform INSERT, UPDATE, or DELETE operations on it, it only supports analytical queries.

```
21] spark.read.csv("users_1.csv", header=True).createOrReplaceTempView("sql_view")

spark.sql("""
SELECT * FROM sql_view
""").show()
21] ✓ 0.4s
```

user_id	username	first_name	last_name	email	date_of_birth	street_address	city postal_code	country is_active	created_ts
25482b57	donna01	Carolyn	Ward	cmullins@example.com	1968-09-17	58241 Tim Spur	West Tylerchester	53663 Argentina	True 2024-07-26 11:59:...
714eb27a	xpatterson	Thomas	Roberts	ahoward@example.net	1992-07-22	265 Salinas Circl...	Obrienville	02635 Hong Kong	True 2023-09-22 19:09:...
0ac47a63	gcampbell	Zachary	Hernandez	reyeslance@example...	2007-06-17	090 Powell Passag...	Francismouth	70787 Slovenia	True 2023-05-17 07:00:...
14002fcd	jamesquinn	Shawn	Robinson	howellgregory@exa...	1982-07-20	79740 Mendoza Hei...	Powershaven	36983 Netherlands Antilles	True 2023-07-14 01:04:...
8007e684	brandondelacruz	John	Simmons	james07@example.com	1940-07-01	074 Devin Shoal S...	Brianhaven	03180 Congo	True 2024-05-18 23:38:...

# Apache Spark - Parquet

The Parquet file format is an **Apache** product, also Open Source like Spark. It is **columnar**, designed primarily for high-performance analytical processing. Compared to a CSV file, which is row-based, Parquet has significant performance benefits.

## Features:

- **Columnar Storage:** Parquet organizes data by columns, so each column's data type is uniform. This enables efficient encoding, compression, and optimization.
  - **Binary Format:** The data is stored in binary format, reducing the overhead of text-based representations. Parquet files are not readable in text editors.
  - **Data Compression:** Supported compression algorithms (Snappy, Gzip, LZ4) allow the file size to be reduced by up to 75% compared to other formats, such as CSV.
  - **Embedded Metadata:** It includes information about the location of columns, data types, minimum and maximum values, as well as the number of row groups. These help in efficiently processing the data.
  - **Parallel Processing Support:** Files can be split, allowing them to be processed in parallel in distributed systems like Apache Spark.

## Parquet vs. CSV:

- **Storage Efficiency:** Parquet creates smaller, compressed files due to its columnar storage. CSV is row-based and does not support compression. There can be up to a tenfold difference in favor of Parquet.
  - **Query Performance:** Parquet allows for fast access to specific columns. In CSV, the entire file must be read, which is slower.
  - **Schema Change Handling:** Parquet supports adding new columns without modifying the existing datasets. CSV does not support this.
  - **Compatibility:** CSV is widely supported by tools and languages, whereas Parquet requires additional libraries for use.



## Parquet in text editor:

PAR10 0-00000-5379f1ca-3056-45a7-9 x +

File Edit View

PAR10 0-00000-5379f1ca-3056-45a7-9

BRUNO -

XH 000

AfghanistanAngola

Builla

Dtigua and Barbuda Argentina

Sub Australia

Sia

zerbaijanBahrain

adosBelgiumBelize

rmudBolivia! Bonaire, Sint Eustatius, B!

SabaBrazilBritish Virgin IslandsBulgariaCanadaCape Verde CaymaChile

Uruguay Colombia

Brook(Costa RicaCubaCuracaoCyprusCzech RepublicDenmarkD(Dominica)

EnEcuadorEgyptEl SalvadorEquatorial GuineaEthiopiaFederated States of Micronesia

FijiFinFrance

?French

ana

PolyGeoTGermany

# Apache Spark - Parquet

In analytics, questions related to the data may arise such as:

- **How many T-Shirts did we sell?**
- On which day of the week do we have the most sales?
- Which product do we sell the most?
- Who buys the most?

To answer these questions with **row-based** data, we need to scan through all the rows from start to finish to extract the necessary columns and values, which can then be aggregated by the engine.

	<b>Product</b>	<b>Customer</b>	<b>Country</b>	<b>Date</b>	<b>Sales Amount</b>
Row 1	Ball	John Doe	USA	2023-01-01	100
Row 2	T-Shirt	John Doe	USA	2023-01-02	200
Row 3	Socks	Maria Adams	UK	2023-01-01	300
Row 4	Socks	Antonio Grant	USA	2023-01-03	100
Row 5	T-Shirt	Maria Adams	UK	2023-01-02	500
Row 6	Socks	John Doe	USA	2023-01-05	200

# Apache Spark - Parquet

In **columnar** data, each column is a separate entity. By asking the same questions, we can get answers much faster because we only need to scan the relevant columns, not all of them.

Column 1	Column 2	Column 3	Column 4	Column 5
Product	Customer	Country	Date	Sales Amount
Ball	John Doe	USA	2023-01-01	100
T-Shirt	John Doe	USA	2023-01-02	200
Socks	Maria Adams	UK	2023-01-01	300
Socks	Antonio Grant	USA	2023-01-03	100
T-Shirt	Maria Adams	UK	2023-01-02	500
Socks	John Doe	USA	2023-01-05	200

However, we are still not done with the advantages of Parquet. The format is not only column-based, but it also uses what are called "**row groups**". The task of row groups is to further reduce the data to be read.

In OLAP (Online **Analytical** Processing) use cases, we mostly use SELECT and WHERE clauses, meaning we define what we want to query—specific columns—and under which conditions.

If we are interested in **how many T-shirts were sold**, Row Group 2 will be skipped because it only contains socks, thus speeding up the processing.

Column 1	Column 2	Column 3	Column 4	Column 5
Product	Customer	Country	Date	Sales Amount
Row Group 1	Ball	John Doe	USA	2023-01-01
	T-Shirt	John Doe	USA	2023-01-02
Row Group 2	Socks	Maria Adams	UK	2023-01-01
	Socks	Antonio Grant	USA	2023-01-03
Row Group 3	T-Shirt	Maria Adams	UK	2023-01-02
	Socks	John Doe	USA	2023-01-05

# Apache Spark - Parquet

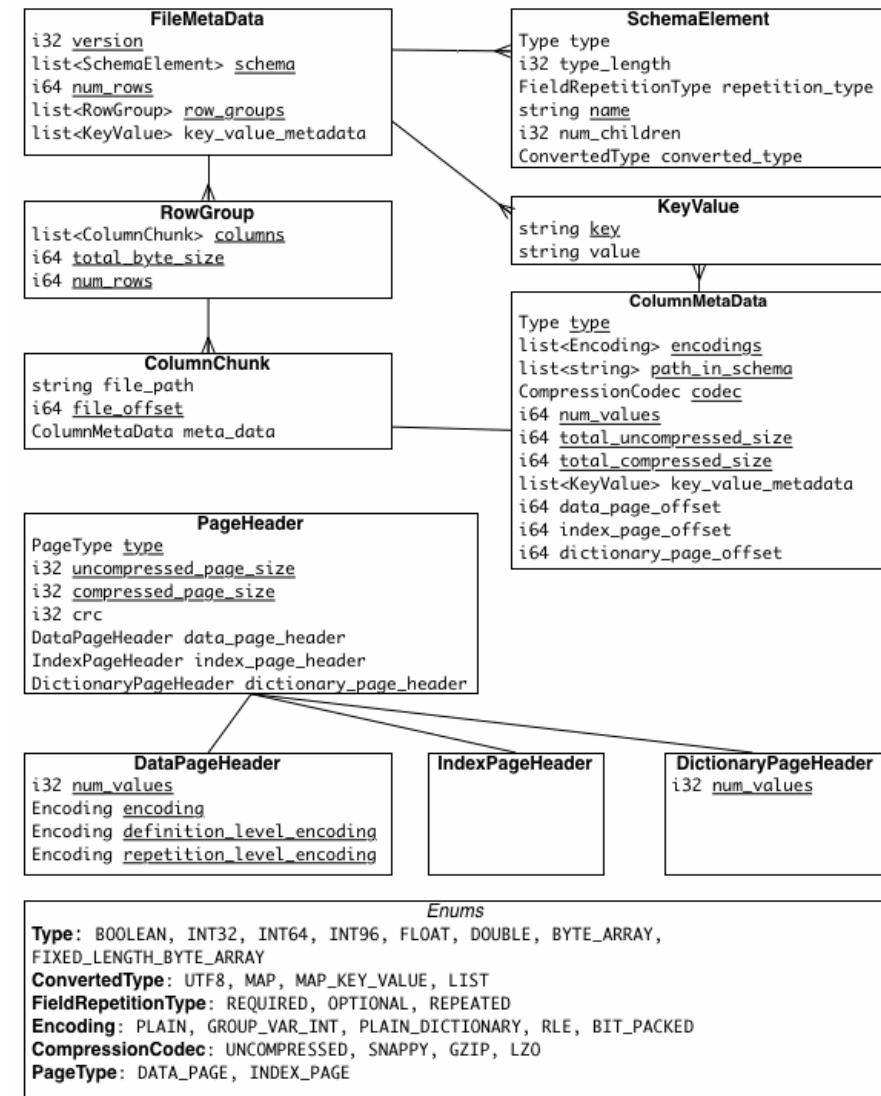
Let's get back to our example. To find out **how many T-shirts were sold**:

- With **row-based** data, we need to scan **6 rows and 5 columns**.
- With **column-based** data, we need to scan **6 rows and 2 columns (product / sales\_amount)**.
- With **columnar** data using **row groups**, we need to scan **4 rows and 2 columns**.

But how does Parquet know which Row Group can be skipped?

Every Parquet file contains metadata, that is, data about the data, such as the minimum and maximum values of a given column in a specific Row Group, as well as information about the schema and columns.

Here is all the metadata from the Parquet documentation.



# Apache Spark - Encoding

Spark has multiple encodings, like **Dictionary Encoding**, or **Run Length Encoding (RLE)**.

Instead of handling string, which can be hundreds of characters long, **Dictionary Encoding** can automatically index them, substituting a string to an integer. An integer will always consume much less storage than a string, especially if the table has lots of repeating values.

RLE compresses data by storing the values and their counts when the same value appears consecutively in the data.

In this case, **Socks** will be stored in the dictionary as 2, every occurrence will be changed to 2. In the data there are two consecutive Socks, so it will look like under the hood like “2:2” as we have two times the second index.

Product	Index	Index	Product
Ball		0	Ball
T-Shirt		1	T-Shirt
Socks		2	Socks
Socks	2		
T-Shirt	1		
Socks	2		

# Apache Spark – Delta format

The Delta format is an open-source data **storage layer** designed to enhance the performance and reliability of big data workflows. Built on top of Apache **Parquet**, Delta introduces powerful features like ACID (Atomicity, Consistency, Isolation, Durability) transactions, ensuring data consistency even during concurrent read and write operations. This makes it **ideal for large-scale ETL** (Extract, Transform, Load) pipelines and streaming data.

## Key features:

- **Schema Evolution:** Supports updates to the dataset's structure without breaking existing workflows.
- **Transaction Log:** Maintains a log of all operations performed on the data, enabling features like rollbacks and time travel for historical queries.
- **Data Versioning:** Allows users to access and query previous snapshots of the data for audit or rollback purposes.
- **Schema Validation:** Enforces data schema consistency by rejecting writes with mismatched structures, preventing corruption.
- **Efficient Metadata Handling:** Provides rich metadata that improves query planning and execution.
- **Integration with Lakehouse Architecture:** Combines the flexibility of data lakes with the governance and performance of data warehouses.

Recap: With the Delta format, you can turn your Parquet-based data lake into a **Lakehouse** (more on this later), combining the scalability of data lakes with the performance and management features of a traditional data warehouse.



# Apache Spark – Data types

Data types used in Spark are similar what you are using Pandas, or in Python.

Spark DataType	Python Type	Pandas Type	Description
StringType	str	object (string)	Represents text data.
IntegerType	int	int64	32-bit signed integer.
LongType	int	int64	64-bit signed integer.
FloatType	float	float32	32-bit floating point number.
DoubleType	float	float64	64-bit floating point number (higher precision than FloatType).
BooleanType	bool	bool	Represents True/False values.
DateType	datetime.date	datetime64[ns]	Represents calendar dates.
TimestampType	datetime.datetime	datetime64[ns]	Represents timestamp values (date + time).
ArrayType	list or tuple	object (array)	Holds an array of elements with the same type.
StructType (with fields)	dict	DataFrame (object)	Represents a complex structure with multiple fields (like a schema).
BinaryType	bytes	object	Holds binary data (e.g., byte streams).
DecimalType	decimal.Decimal	float64 or object	Arbitrary precision decimal numbers (useful for financial calculations).
MapType	dict	object (dict)	Key-value pairs, where keys and values have fixed types.

## Key Notes:

- **Precision Differences:** Spark types like DoubleType and DecimalType are useful for high-precision calculations, whereas Python's float may lose precision.
- **Missing Values:** Spark uses `null`, while Pandas uses NaN or None for missing data.
- **Complex Types:** Spark supports **StructType**, **ArrayType**, and **MapType**, which allow for nested and hierarchical data structures. Pandas typically flattens such data into columns.

# Apache Spark – Partitioning

Partitioning involves **splitting data into smaller chunks**, increasing parallelism and improving data processing efficiency. This enhances performance when dealing with large datasets.

## How it works:

- **Methods:** Data is distributed using hash-based or range-based partitioning.
- **Numerical columns:** Partitioning based on numerical columns enables efficient querying.
- **CPU cores:** Each partition is assigned to a separate CPU core, maximizing parallel processing.
- **Shuffle operations:** Rearranging data can be expensive in terms of time and computational resources.

## Advantages:

- **Performance:** Reduces data movement and accelerates processing.
- **Resource management:** Better utilization with smaller chunks of data processed at a time.
- **Data organization:** Optimizes the file system and supports local processing.

## The three types of partitioning:

- Hash
- Range
- `partitionBy()`

# Apache Spark – Partitioning - Hash

```
# Create a sample DataFrame
df = spark.createDataFrame(
    [
        (1, "Gary", 20),
        (2, "Nora", 22),
        (3, "Joe", 26),
        (4, "James", 35),
        (5, "Marcus", 39),
        (6, "Frank", 56)
    ],
    schema=["id", "name", "age"])

# Create four partitions on the column "id"
df = df.repartition(4, "id")

print(df.rdd.glom().collect())

"""
Formatted output:
[
    [
        Row(id=2, name='Nora', age=22), Row(id=4, name='James', age=35),
        Row(id=5, name='Marcus', age=39)
    ],
    [
        Row(id=1, name='Gary', age=20),
        Row(id=6, name='Frank', age=56)
    ],
    [], # Third partiton is empty
    [
        Row(id=3, name='Joe', age=26)
    ]
]
```

- The default method assigns a unique hash value to each record based on a predetermined column (e.g., "id") and groups records into partitions accordingly.
- **Advantage:** Ensures even distribution, especially for evenly distributed keys.
- **Disadvantage:** Less effective if keys are not evenly distributed.

In the example we asked for four partitions, but as there were six ids, and the distribution of rows are not even.

That's because repartition use **hash** partitioning where rows are assigned to partitions based on the hash of the column value modulo the number of partitions. This process **can result in uneven partition sizes** or empty partitions, especially with small datasets.

**coalesce()**, on the other hand, reduces the number of partitions by merging existing ones without a shuffle, which avoids empty partitions and helps create more balanced partition sizes.

## When to Use:

- Use **repartition** when you want to increase the number of partitions or enforce a specific shuffle to achieve uniform distribution, especially with large datasets.
- Use **coalesce** when reducing the number of partitions to optimize performance, such as before writing to a file or when dealing with small datasets.

# Apache Spark – Partitioning - Range

```
# Create a sample DataFrame
df = spark.createDataFrame(
    [
        (1, "Gary", 20),
        (2, "Nora", 22),
        (3, "Joe", 26),
        (4, "James", 35),
        (5, "Marcus", 39),
        (6, "Frank", 56)
    ],
    schema=["id", "name", "age"])

# Create three range partitions on the column "age"
df = df.repartitionByRange(3, "age")

print(df.rdd.glom().collect())

"""
Formatted output with three groups on "age":
[
    [Row(id=1, name='Gary', age=20), Row(id=2, name='Nora', age=22)],
    [Row(id=3, name='Joe', age=26), Row(id=4, name='James', age=35)],
    [Row(id=5, name='Marcus', age=39), Row(id=6, name='Frank', age=56)]
]
"""
```

- Data is partitioned into predefined ranges.
- Example: Numeric column values might be split into ranges such as [0-100], [101-200], etc.
- **Unlike hash partitioning**, range partitioning enables more predictable and query-friendly partitioning, which is ideal for queries that filter by ranges or sequential operations.
- A common drawback is the potential for skewed partitions if the data is unevenly distributed across the specified ranges.
- It is frequently used in data warehousing and partition pruning, where irrelevant partitions are excluded from processing, improving query performance.

# Apache Spark – Partitioning – partitionBy()

```
(  
    df  
    .write  
    .partitionBy("Age")  
    .mode("overwrite")  
    .parquet("output_file")  
)
```

- The partitionBy method in PySpark is used when **writing a DataFrame to storage**, allowing data to be partitioned based on the values of one or more specified columns.
- It **creates a directory structure** in the target location, where each partition's data is saved under a folder named after the column and its corresponding values (e.g., column=value).
- Partitioning **improves query performance** by enabling Spark to read only the relevant partitions when filtering data based on the partitioning column.
- However, **over-partitioning** (creating too many small files) or **under-partitioning** (grouping too much data into a few partitions) can negatively impact performance and storage efficiency.
- This method is **commonly used** in data lakes and large-scale ETL pipelines to organize data for efficient processing and analysis.

# Apache Spark – Lazy Evaluation

Due to Lazy Evaluation, data processing is **delayed**.

The idea is that operations (transformations) are **not executed immediately**, but are recorded as a **logical plan** (DAG - Directed Acyclic Graph). **Actions**, such as `collect()` or `show()`, **trigger the execution**.

This is efficient because it allows Spark to optimize the entire process.

The Spark **Catalyst query optimizer** is responsible for converting the logical plan into a physical plan. Catalyst examines possible execution strategies and selects the one that provides the best performance.

This includes query merging, distributed execution, and processing data queries in a compressed form.

As a result, it uses **less memory** and **reduces I/O costs**.

## Example of Lazy Evaluation:

- **Transformations (plan):** The courier receives a list of addresses at the start of the day but does not leave immediately. Instead, they create a plan: optimizing the route to visit the addresses in the least amount of time.
- **Action (execution):** Delivery only starts when the courier actually delivers the packages.

In Spark, transformations correspond to the route plan created by the courier. They are just part of a logical plan and are not executed immediately. The action, such as `collect()` or `show()`, is equivalent to the courier leaving, during which the optimized plan is executed.

In this example, Catalyst acts as the navigation system, calculating the best route between the addresses.

# Apache Spark – Lazy Evaluation

```
df_read_csv = spark.read.csv("../data/data_for_pyspark_tutorial/2010-summary.csv", header=True)
df_read_csv.createOrReplaceTempView("df_read_csv")
```

```
dest_country_count = spark.sql(
    """
        SELECT
            DEST_COUNTRY_NAME,
            count(*) as cnt
        FROM
            df_read_csv
        GROUP BY
            DEST_COUNTRY_NAME
    """
)
```

```
filtered_dest_country_count = dest_country_count.filter(sf.col("DEST_COUNTRY_NAME") == "United States")
filtered_dest_country_count.explain()
```

In the query plan for dest\_country\_count, you will see that the filtering is done **before** aggregation.

In our code, we specified a filter **after** the Group By. However, Spark knew that filtering could be done before the Group By and hence reordered the operation for performance.

DEST_COUNTRY_NAME	cnt
Anguilla	1
Russia	1
Paraguay	1
Senegal	1
Sweden	1
Kiribati	1
Guyana	1

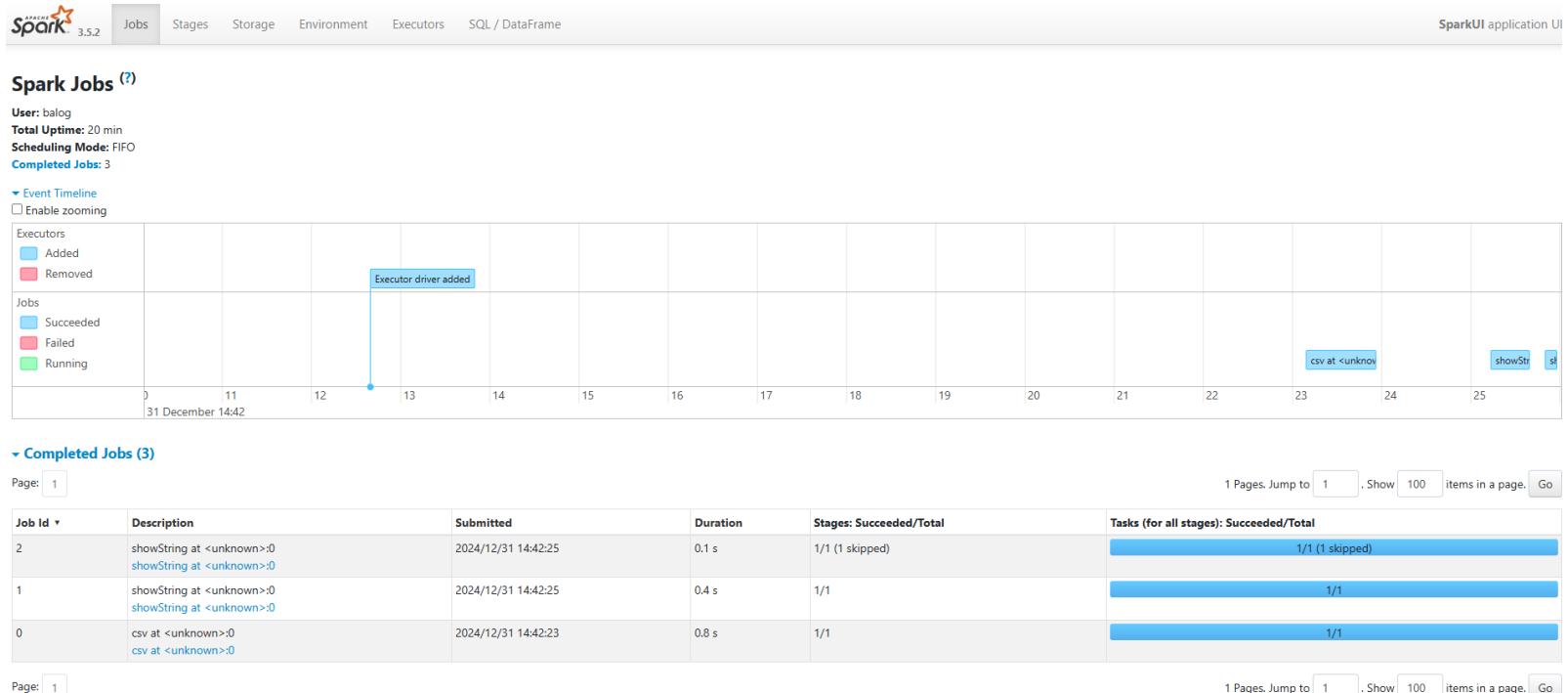
```
-- Physical Plan --
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[DEST_COUNTRY_NAME#63], functions=[count(1)])
  +- Exchange hashpartitioning(DEST_COUNTRY_NAME#63, 200), ENSURE_REQUIREMENTS, [plan_id=104]
    +- HashAggregate(keys=[DEST_COUNTRY_NAME#63], functions=[partial_count(1)])
      +- Filter (isnotnull(DEST_COUNTRY_NAME#63) AND (DEST_COUNTRY_NAME#63 = United States))
        +- FileScan csv [DEST_COUNTRY_NAME#63] Batched: false, DataFilters: [isnotnull(DEST_COUNTRY_NAME#63), (DEST_COUNTRY_NAME#63 = United States)]
```

# Apache Spark – SparkUI

The previous example is used, but the “explain()” was changed to “show()” to trigger computation.

`filtered_dest_country_count.show()`

After the code ran, this can be seen on localhost:4040:



The **SparkUI** helps track the execution of Spark applications.

Using it, you can see the **performance of individual jobs and stages**, the details of data processing, the utilization of executors, and the stored data. It aids in diagnosing issues, such as when the **application is slow or detects errors**, and provides insight into how Spark executes tasks, optimizing them for the best performance.

You can access it on your own machine at <http://localhost:4040/> once the **SparkSession** has started.

# Apache Spark – SparkUI

**Main sections of SparkUI:**

**1. Jobs tab:**

- Displays all executed jobs.
- A job is usually associated with the execution of an action (such as `show()`, `collect()`, etc.).

**2. Stages tab:**

- Shows the execution steps of individual jobs, broken down into logical stages (e.g., map, shuffle, reduce).
- The DAG Visualization helps visualize the data processing flow.

**3. Storage tab:**

- Displays the state and size of RDDs or DataFrames stored in the cache.

**4. Environment tab:**

- Lists the current Spark environment configuration (e.g., JVM settings, Spark parameters)

**5. Executors tab:**

- Contains the status, memory utilization, and processing statistics of the executors.

**6. SQL / DataFrame tab:**

- Displays the execution plan (logical and physical plan) of SQL queries generated from DataFrames.

# Apache Spark – Narrow vs. Wide transformations

Spark cluster consists of multiple worker nodes, each with one or more executors.

When processing data, Spark retrieves it from disk or external storage and performs transformations.

These transformations fall into two categories:

## 1. Narrow transformations:

These do not require data movement between executors. They work on individual rows of data.

## 1. Wide transformations:

These involve moving data between executors and require data from multiple rows. Common wide transformations include:

- **GroupBy:** Requires data from multiple rows to group by keys, causing data exchange between executors.
- **Join:** Moves data between executors to match rows based on join keys.
- **Window functions:** Require data from multiple rows in the same executor.

Wide transformations are more expensive because moving data between executors is slower than processing data in memory.

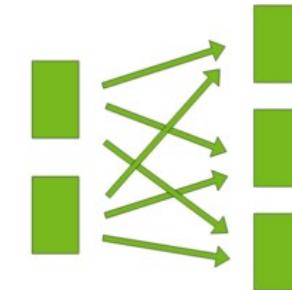
The operation cost hierarchy is (highest to lowest):

1. **Moving** data between executors.
2. **Reading** data from storage.
3. **Processing** data in memory.

To write efficient Spark code, reducing data movement between executors is very important.



Narrow Transformations



Wide Transformations (shuffles)

## Best practices for handling wide transformations:

- **Minimize shuffling:** Reduce the need for wide transformations by optimizing joins and groupings.
- **Broadcast joins:** Use when one table is small enough to fit in memory, avoiding a shuffle for the larger table.
- **Repartition carefully:** Repartition or coalesce data based on your transformations to minimize unnecessary shuffling.
- **Avoid unnecessary aggregations:** Try to pre-aggregate or filter data before applying transformations like groupBy.
- **Cache intermediate results:** Cache intermediate results to reduce redundant processing when performing wide transformations multiple times.

# Apache Spark – AQE

**Adaptive Query Execution (AQE)** is a feature in Apache Spark that allows for dynamic modification of the query execution plan during runtime based on the data being processed.

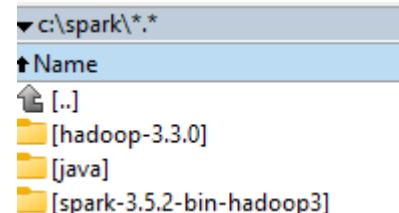
Introduced in Spark 3.0, this feature improves query performance by optimizing based on runtime statistics. AQE helps Spark better adapt to different data sizes and data skew issues, thereby increasing query efficiency and reducing the chances of errors during processing.

AQE is enabled by default.

# Apache Spark – Installing PySpark on Windows machines

1. Create a folder called “spark” on your “C:\” root (“C:\spark”).
2. **Install Java8. Important:** If you already have java (open cmd, and type “java –version”, if you get back “java version “1.8.xxxx”, then you have it), and it is installed in the “Program Files (x86)” folder, delete it via “Add or Remove Programs” and reinstall it under “C:\spark\java” (otherwise the whitespaces in “Program Files (x86)” will mess with the Spark installation).
3. **Download Spark.** Notice the Hadoop version (here it’s **3.3 and later**).
4. **Download Hadoop:** In order to make Spark work under Windows, we need “winutils” from [this](#) page. Click on “Code” and “Download ZIP” to get the repository. You’ll get all the Hadoop versions, but you’ll use only the one folder you need from Step 3. (“3.3 and later” in the example).
5. **Copy** the Spark and Hadoop folder to “C:\spark”. You will have three folders (maybe with different versions).
  - In the “**hadoop-3.3.0**” folder there will be one folder called “bin”.
  - In the “**spark-3.5.2-bin-hadoop3**” folder you’ll have multiple folders like “bin”, “conf”, “data”.

## Step 5.



## Step 3.

### Download Apache Spark™

1. Choose a Spark release: 3.5.4 (Dec 20 2024) ▾
2. Choose a package type: Pre-built for Apache Hadoop 3.3 and later ▾
3. Download Spark: spark-3.5.4-bin-hadoop3.tgz

## Step 4.: “Code” -> “Download ZIP”

# Apache Spark – Installing PySpark on Windows

6. Create System Environment Variables (Start / Edit the system environment variables): Add four new System variables (the bottom part): JAVA\_HOME, HADOOP\_HOME, SPARK\_HOME, PYSPARK\_PYTHON. The “Variable value” will be their path for the first three, and for PYSPARK\_PYTHON it’s “python”.

Edit System Variable

Variable name:	JAVA_HOME
Variable value:	c:\spark\java
Browse Directory... Browse File... OK Cancel	

Edit System Variable

Variable name:	HADOOP_HOME
Variable value:	c:\spark\hadoop-3.3.0
Browse Directory... Browse File... OK Cancel	

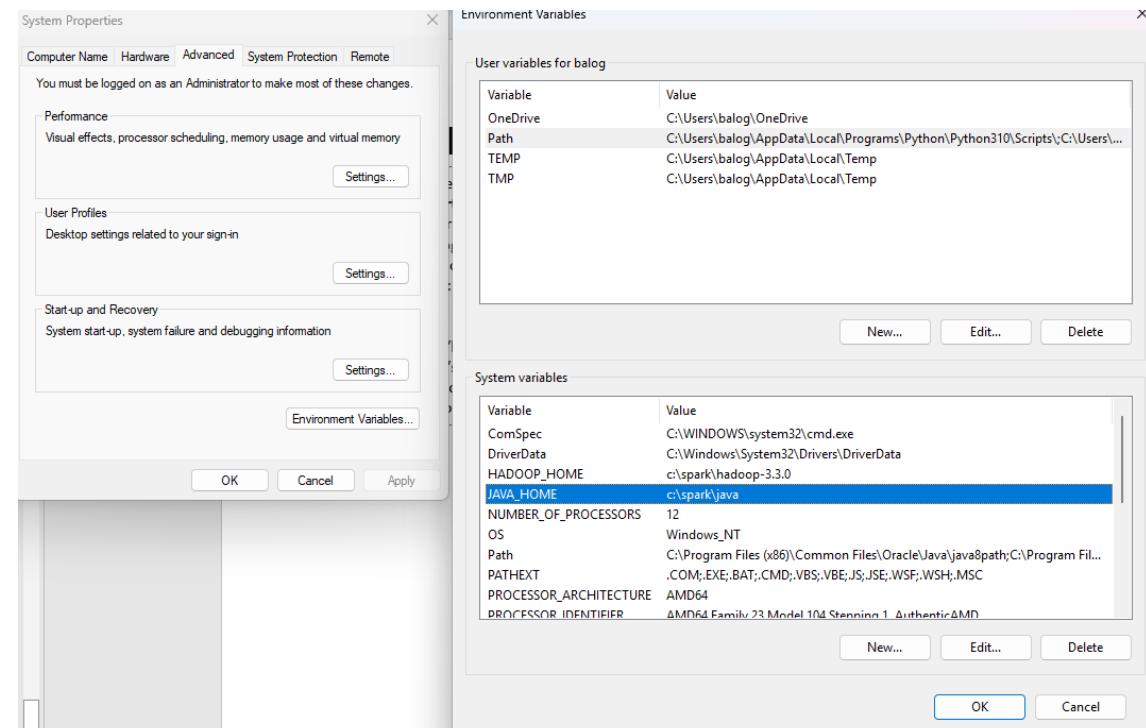
Edit System Variable

Variable name:	SPARK_HOME
Variable value:	c:\spark\spark-3.5.2-bin-hadoop3
Browse Directory... Browse File... OK Cancel	

Edit System Variable

Variable name:	PYSPARK_PYTHON
Variable value:	python
Browse Directory... Browse File... OK Cancel	

Step 6.



# Apache Spark – Installing PySpark on Windows

7. Then go to "Path" still under System Variables and add these three rows:

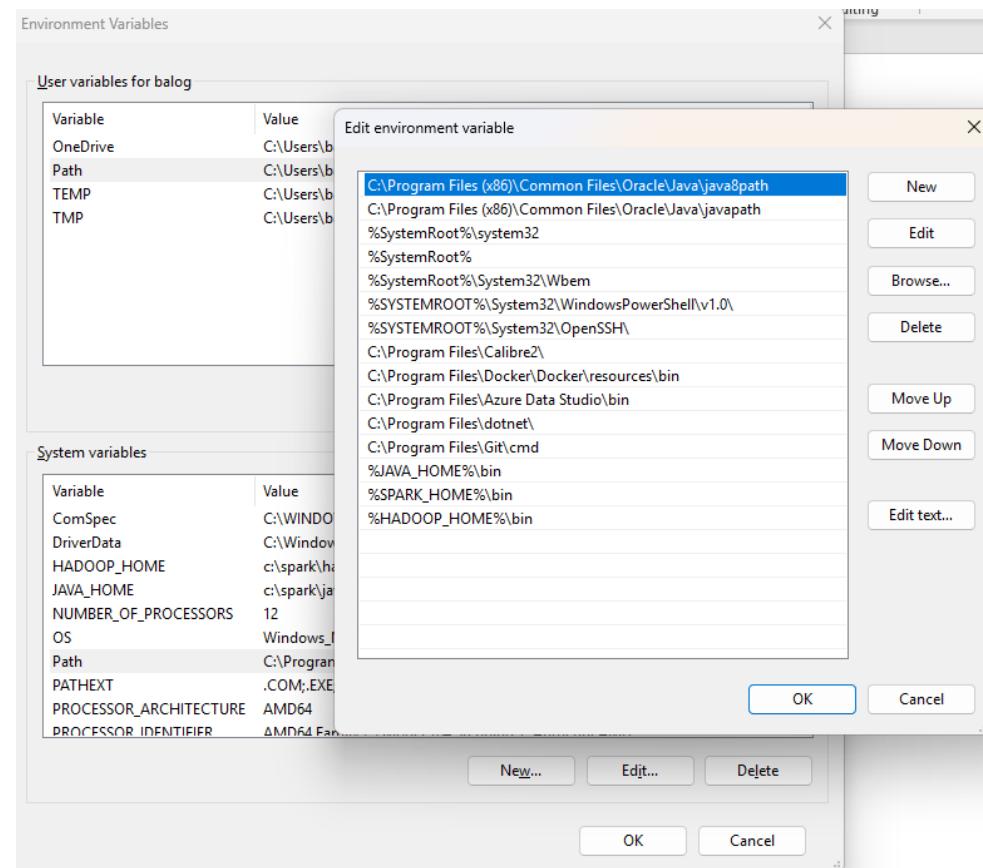
```
%JAVA_HOME%\bin  
%SPARK_HOME%\bin  
%HADOOP_HOME%\bin
```

Click on “OK” to close all windows for the Environment Variables.

8. Open a cmd, and type “spark-shell”, if you see similar things, then you’re good to go. If you get an error “**spark-shell is not a recognizable command**” then maybe you haven’t saved the Environment Variables, check them, and add them again if needed.

#### **Step 8. Test the installation**

## **Step 7. (add the last three rows)**



# Apache Spark – Using Google Colab

If you are unable to use PySpark locally, you can use [Google Colab](#).

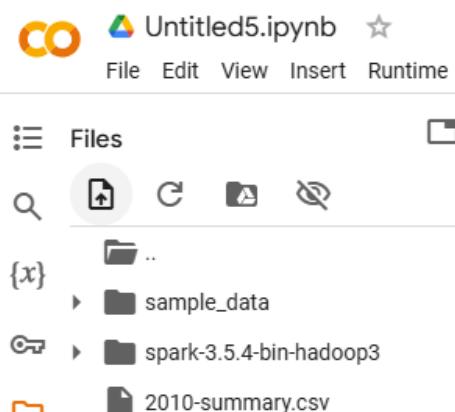
## What is Colab:

Colab is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs.

Run this code as the first cell of your notebook, and PySpark will be installed.

Upload the sample files under the “paper with the up arrow” icon, than use them:

```
df = spark.read.csv("/content/2010-summary.csv",  
header=True)
```



```
!sudo apt update  
!apt-get install openjdk-8-jdk-headless -qq > /dev/null  
!wget -q https://dlcdn.apache.org/spark/spark-3.5.4/spark-3.5.4-bin-hadoop3.tgz  
!tar xf spark-3.5.4-bin-hadoop3.tgz  
!pip install -q findspark  
!pip install pyspark  
!pip install py4j  
  
import os  
import sys  
  
import findspark  
findspark.init()  
findspark.find()  
  
import pyspark  
  
from pyspark.sql import DataFrame, SparkSession  
import pyspark.sql.types as st  
import pyspark.sql.functions as sf  
  
spark= SparkSession \  
    .builder \  
    .appName("Spark Tutorial") \  
    .getOrCreate()  
  
spark
```