

Tutorial 3

Introduction to Apache Spark

Anis Barhoum

Recap of the last tutorial



Big Data introduced significant challenges in **storing**, **processing**, and **managing** huge datasets efficiently.

Traditional systems struggled to scale, leading to the need for distributed solutions like Hadoop.

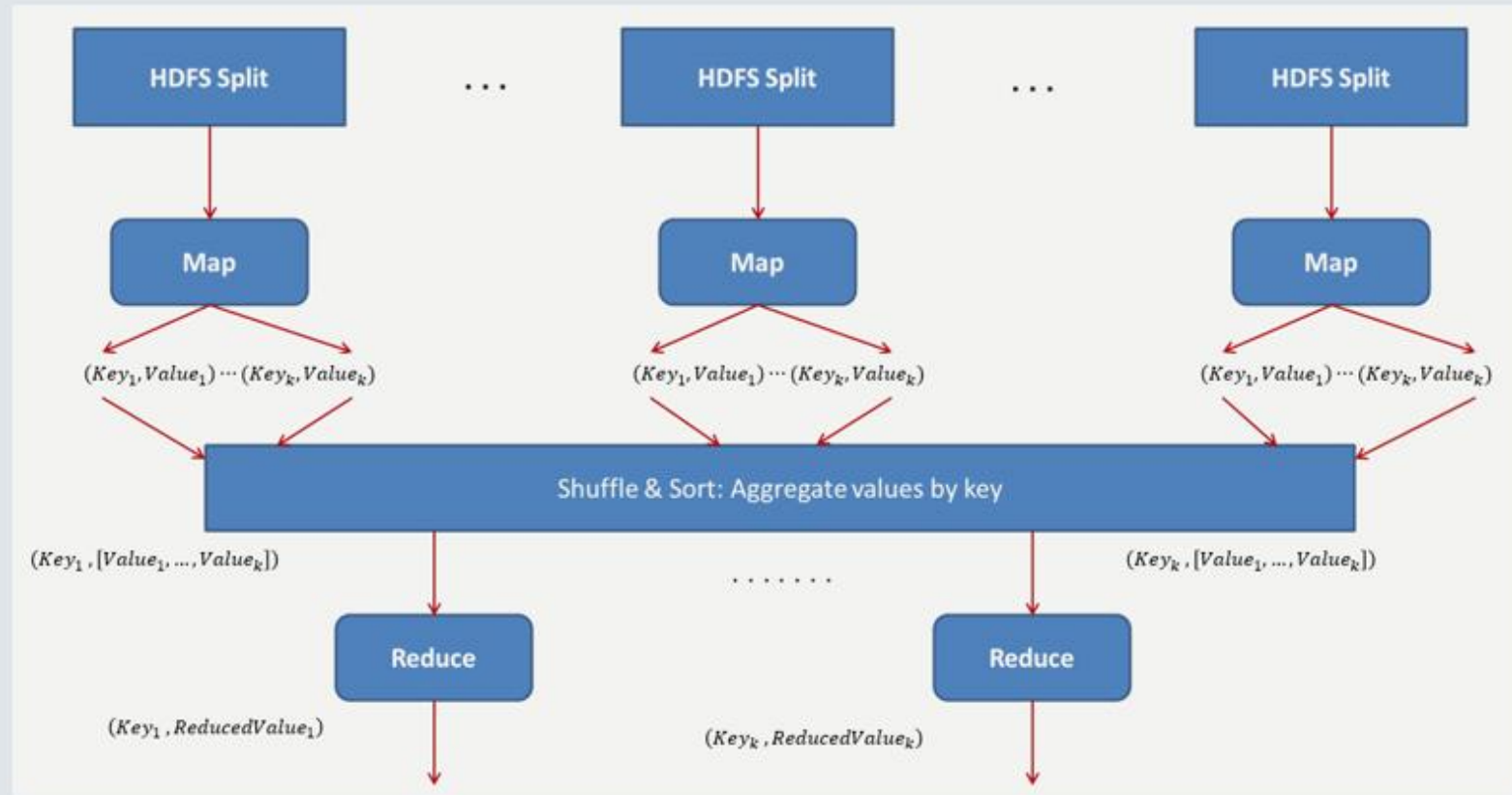


Figure 1: MapReduce Overview

However...

Limitations of MapReduce

- Slow due to disk I/O – After mapping, the shuffling phase involves expensive operations like sorting, partitioning, and moving data between nodes, making MapReduce inefficient.
- Not suitable for complex tasks – Machine learning, graph processing, and interactive queries require more advanced frameworks, which MapReduce does not support.

You can think of more!

What now?



Introducing Spark, a framework that provides multiple APIs (Application Programming Interfaces – ways to interact with the system, we will mainly use the PySpark API (Python)). Spark is a:

- Fast, open-source, distributed computing framework.
- Built on top of Hadoop **MapReduce**.
- Utilizes In-memory computing (instead of writing to the disk) – up to ~100 times faster.

Key Concepts in Spark

- **Application** - A user program that runs on Spark using its APIs, consisting of a driver program and executors on a cluster.
- **SparkSession** - The entry point for interacting with the underlying spark functionality. In a Spark application, you must create a SparkSession object yourself.

```
spark = SparkSession.builder.appName('Tutorial 3').getOrCreate()
```

Key Concepts in Spark

- **Job** - A parallel computation triggered by a Spark action (e.g. `.save()`, `.collect()`, etc.)
- **Stage** - A subdivision of a job—tasks within the same stage depend on each other and execute serially.
- **Task** - A single unit of work sent to a Spark executor, executed in parallel across the cluster.

In Spark, an application runs a SparkSession, which manages the execution of jobs; each job consists of multiple stages, and each stage is broken down into parallel tasks.

Demo: 'Initialization' in the Colab Notebook

The RDD – Resilient Distributed Datasets

To overcome MapReduce's biggest limitation - constant disk I/O- Spark introduces RDDs, which allow data to be cached in memory for much faster processing.

- The core data structures in Spark.
- **Immutable** and **distributed** – split across nodes and processed parallelly.
- **Fault-tolerant** – automatically recovers from failures.
- **Partitioned** – RDDs are divided into smaller chunks across different machines

RDD Operations: Transformations & Actions

Transformations are operations that will not be completed at the time you write and execute the code in a cell – they only get executed once you call an action (next slide).

For example: `.select()`, `.distinct()`, `.groupBy()`, `.sum()`, `.orderBy()`, `.filter()`, `.limit()`, `.map()`, etc.

RDD Operations: Transformations & Actions

Actions are commands that are computed by Spark right at the time of their execution. They consist of running to the previous transformations to get back an actual result.

For example: `.show()`, `.count()`, `.collect()`, `.save()`, etc.

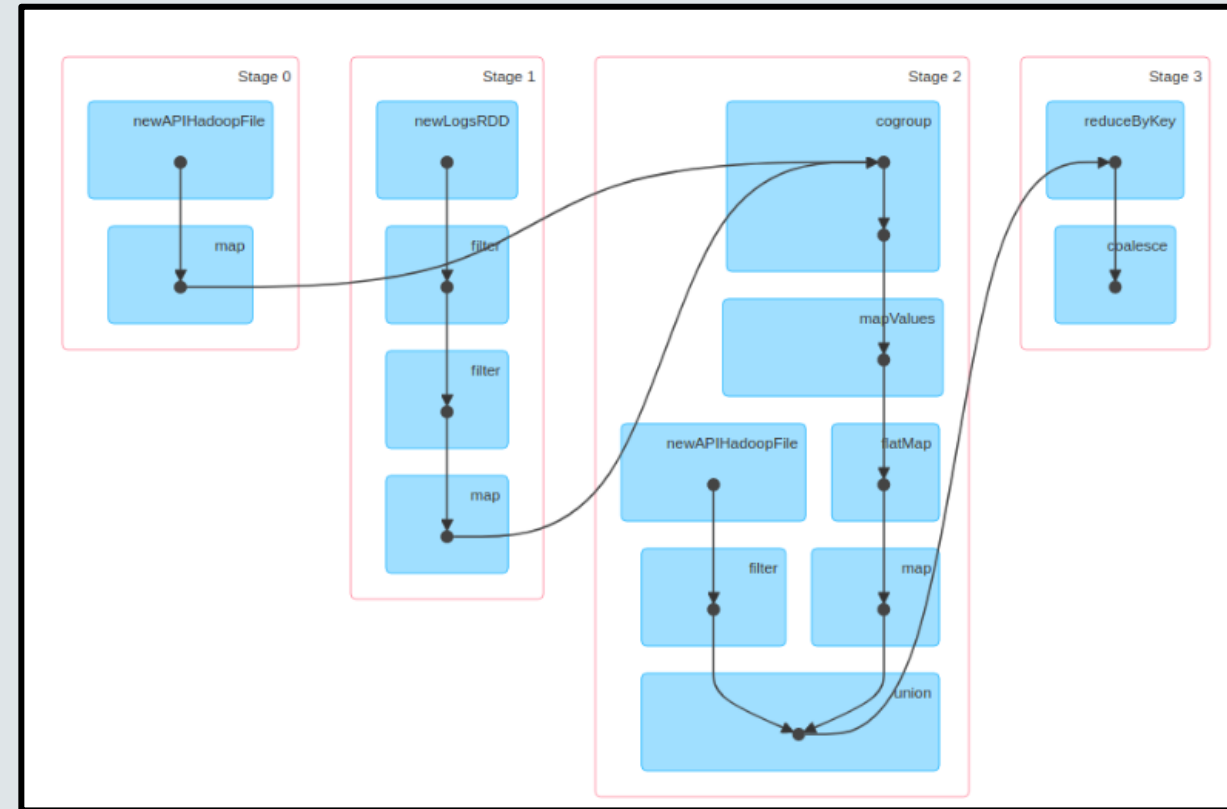
Spark 🤝 Lazy Evaluation

Spark is designed with lazy evaluation at its core, meaning:

- It **avoids unnecessary computations** – Only runs when needed.
- **Optimizes execution** – reduces redundant work and disk writes.
- **Improves fault tolerance** – tracks transformations efficiently.

Spark Execution: DAG in Action


- Vertices are RDD, edges are Transformations.
- Generalization of MapReduce
- Actions divide the DAG to stages.
- Spark decides which calculations should be recomputed/reused.



Demo: 'Working with RDDs' in the Colab Notebook

Spark Structured APIs

the **Structured APIs** refer to the high-level APIs that provide a structured way to process data using distributed computing. These APIs include:

- DataFrames API (what we're going to use )
- Datasets API
- SQL API

The APIs allow Spark to run its optimizations and an easier way for us to do what we want.

The DataFrame

- A DataFrame is the most common Structured API in Spark, representing a table of data with rows and columns.
- Like the RDD, the DataFrame is an immutable, in-memory, resilient, distributed collection of data.
- One main advantage is that it allows for better optimizations (memory and execution wise).

Data Types: <https://spark.apache.org/docs/latest/sql-ref-datatypes.html>

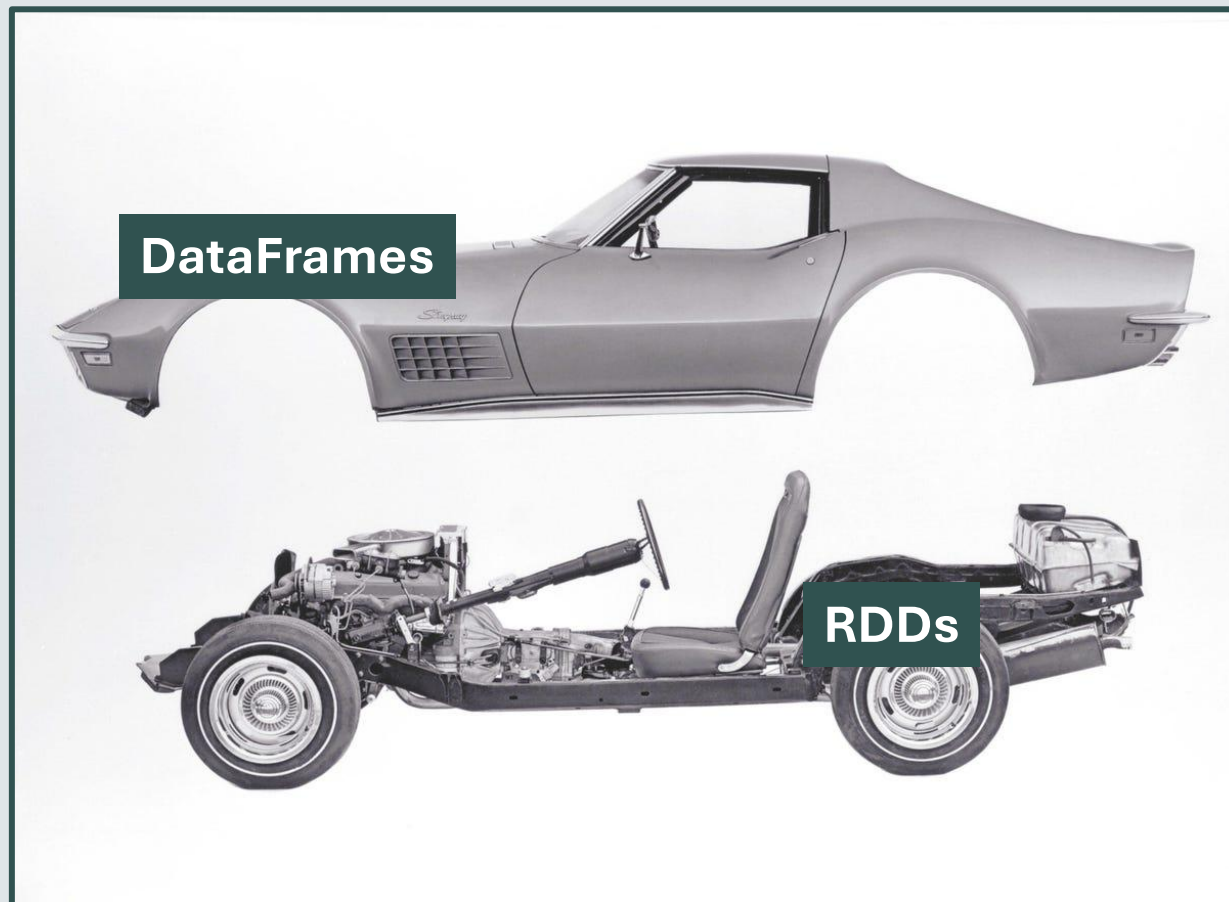
DataFrames vs RDDs

We usually prefer to use DataFrames!

However, consider using RDDs if:

- You have unstructured data (text, media).
- You need specific execution control.
- You need to perform data manipulation with functional programming concepts.

A little secret...



Working with DataFrames: The Schema

A DataFrame always has a **schema**.

- **Schema** – a StructType made up of several fields, StructFields, that have a name, type, and a Boolean flag which specifies if they can contain null.
- When reading data from a file, the schema can be inferred automatically at a cost of reading the data more than once.

Working with DataFrames: The Row

In Spark, each **row** in a DataFrame is a single record.

- Spark represents this record as an object of type Row.
- Spark manipulates Row objects using column expressions.
- Internally, rows are stored as byte arrays for efficiency, however, this is hidden from users – we can only use column expressions to process the data.

Demo: 'Working with DataFrames' and the rest of the Colab notebook