HADOOP AND SPARK DATA PROCESSING

Data Data Everywhere!

90% of world's data is
generated in just last 2 years

In early 2000, the amount of data being generated exploded exponentially!!

Due to the use of Social media, Organizations found themselves struggling with this massive amount of data

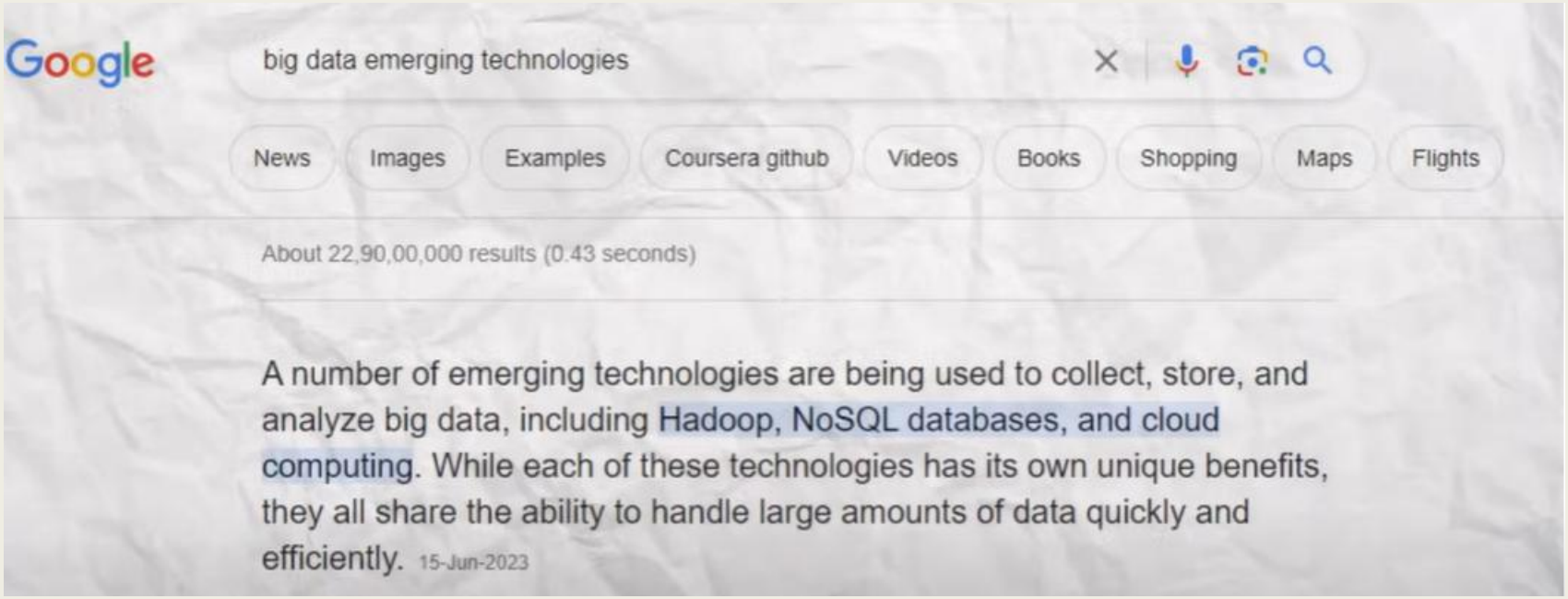
Which became very hard to process

Big Data

To address this the concept of big data emerged.

Big data refers to extremely huge amount of data that is very difficult to store and process

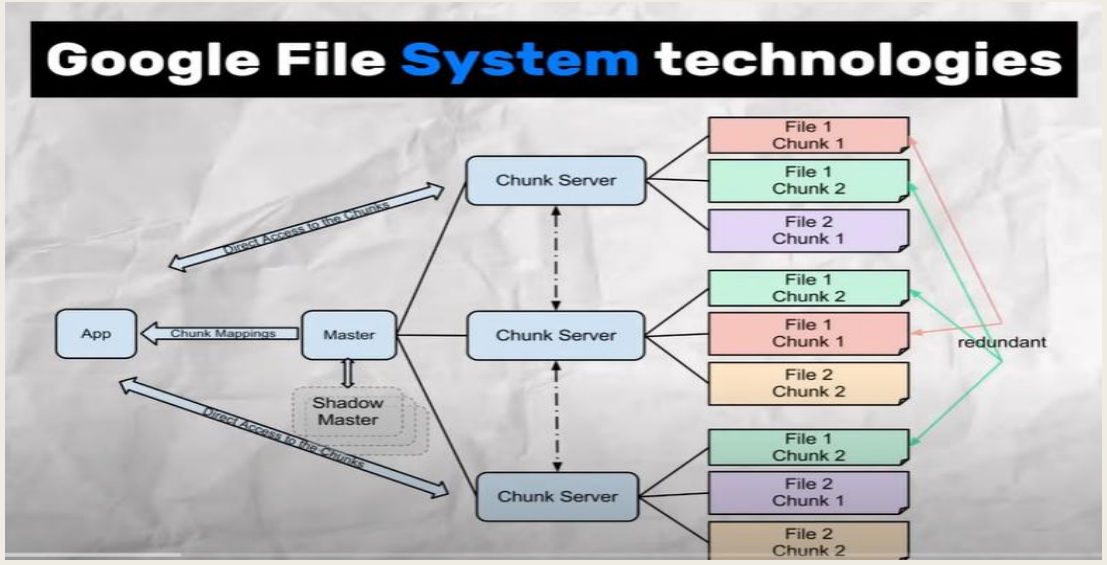
Organizations around the globe wanted to process this huge amount of data to get important insights from it.

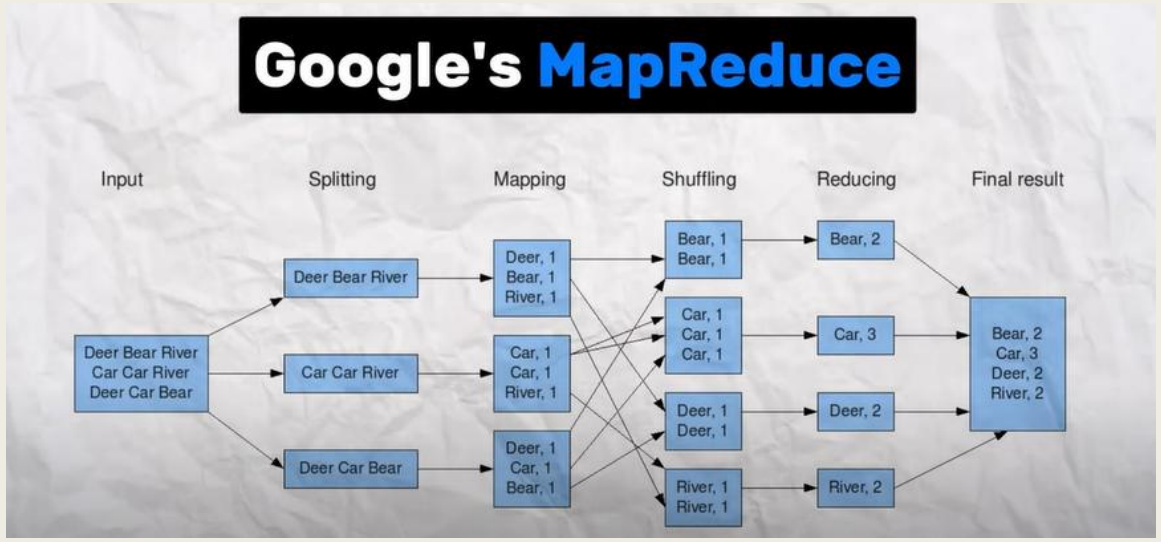


Hadoop

In 2006 a group of engineers from Yahoo developed a software Framework called Hadoop

Inspired by Googles' MapReduce and GFS, Hadoop introduced "Distributed Processing"





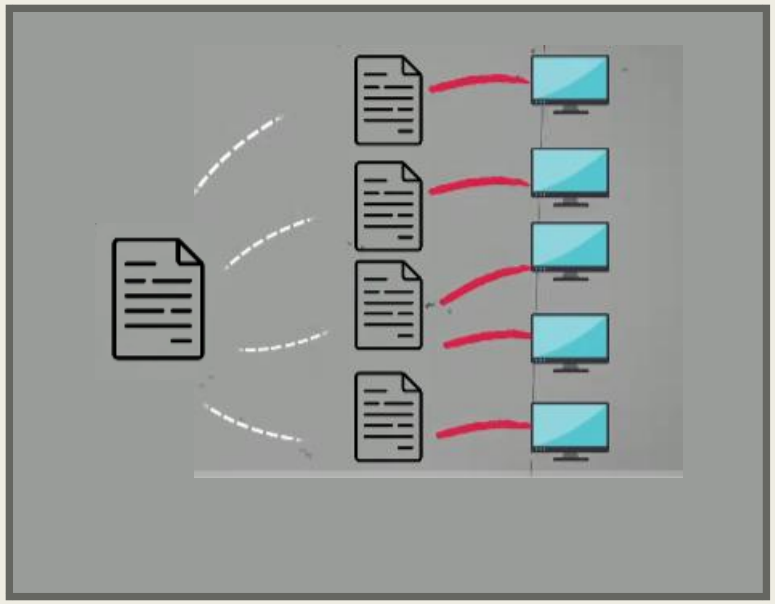
Distributed Processing

Instead of relying on a single machine it used multiple computers to get the results

Each machine in the cluster will get some part of data to process, they will work simultaneously on this data and at the end will combine the output to get the final results

Hadoop Components (1)

2 important components of Hadoop:

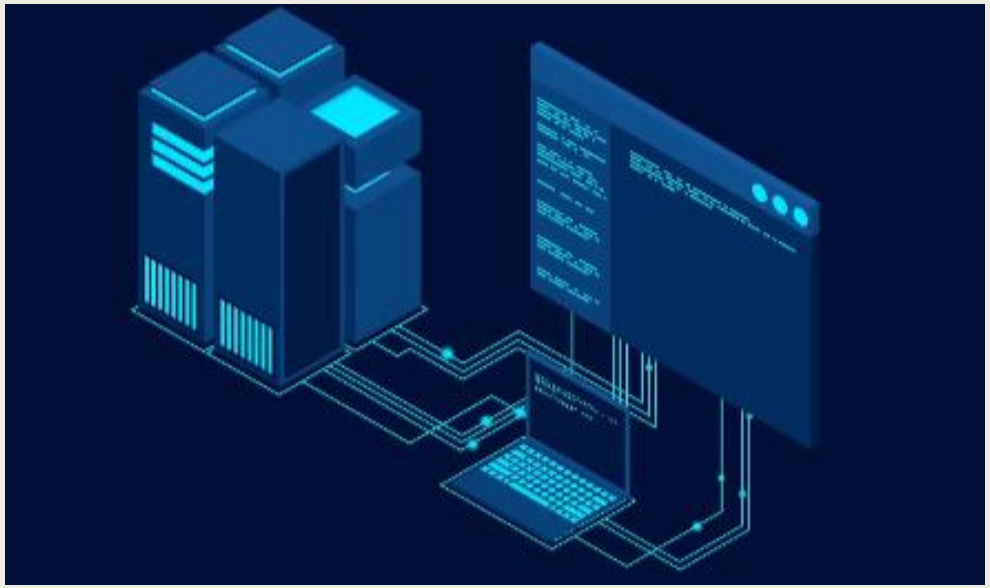


HDFS is a giant storage system for keeping our Big Data safe

It divides our data into multiple chunks and stores all of this data across different computers

Hadoop Components (2)

2 important components of Hadoop:



MapReduce is a super smart way to process all of this data together

It divides our data into multiple chunks and processes all of this data together in a parallel manner

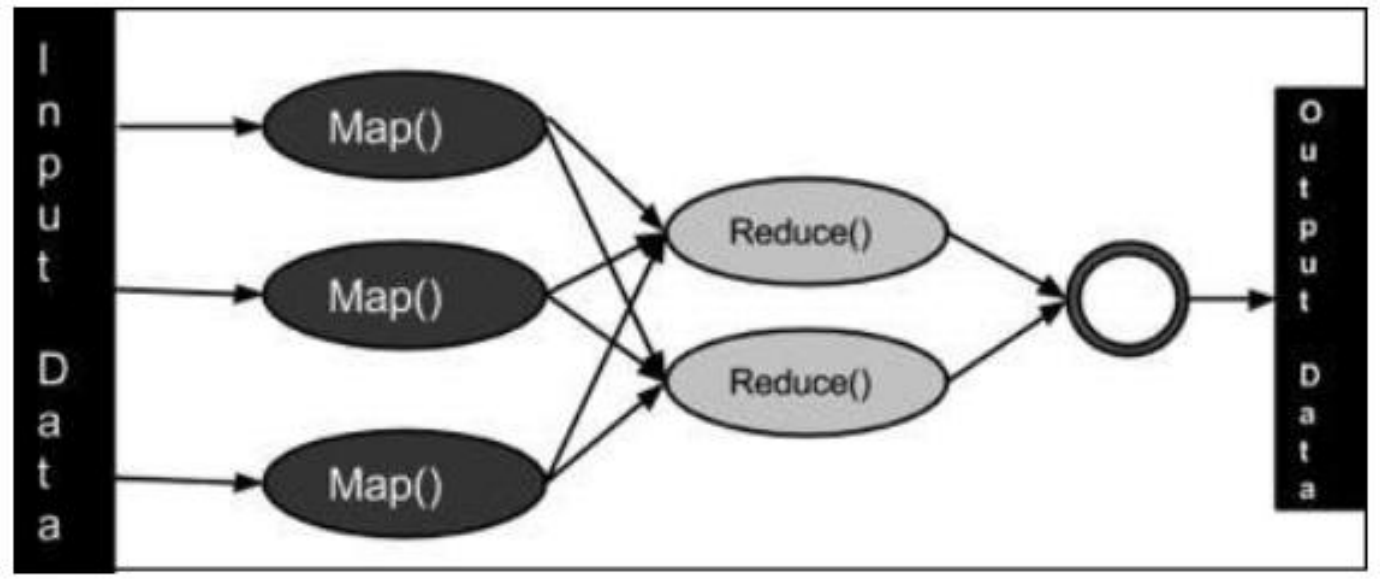
Map Reduce

MapReduce is a **framework**:

- Write applications to process huge amounts of data
- In parallel
- On large clusters of commodity hardware
- Reliable manner

MapReduce Tasks:

- Map
- Reduce

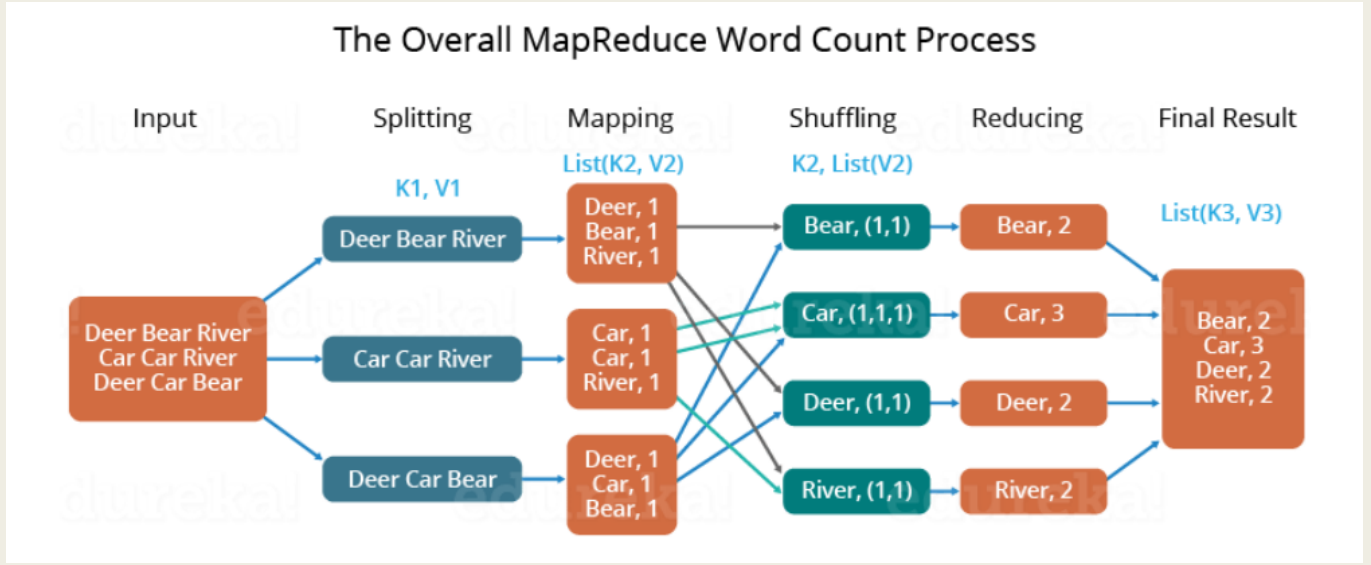


Map Reduce

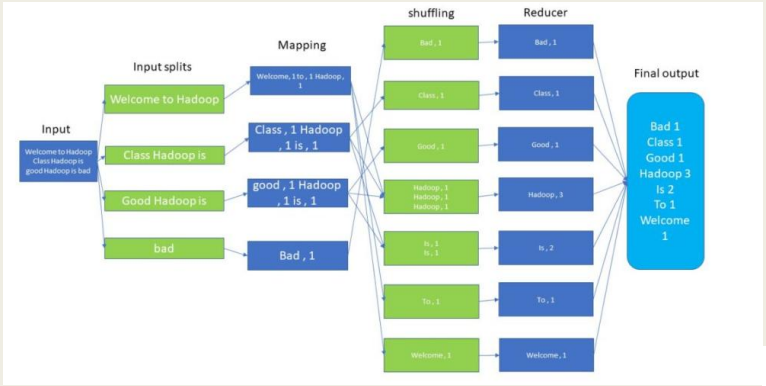
The Algorithm

- Generally MapReduce paradigm is based on sending the computer to where the data resides!
- MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
    - **Map stage** – The map or mapper's job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.
    - **Reduce stage** – This stage is the combination of the **Shuffle** stage and the **Reduce** stage. The Reducer's job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.
- During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.
- The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes.
- Most of the computing takes place on nodes with data on local disks that reduces the network traffic.
- After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.

	Input	Output
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)

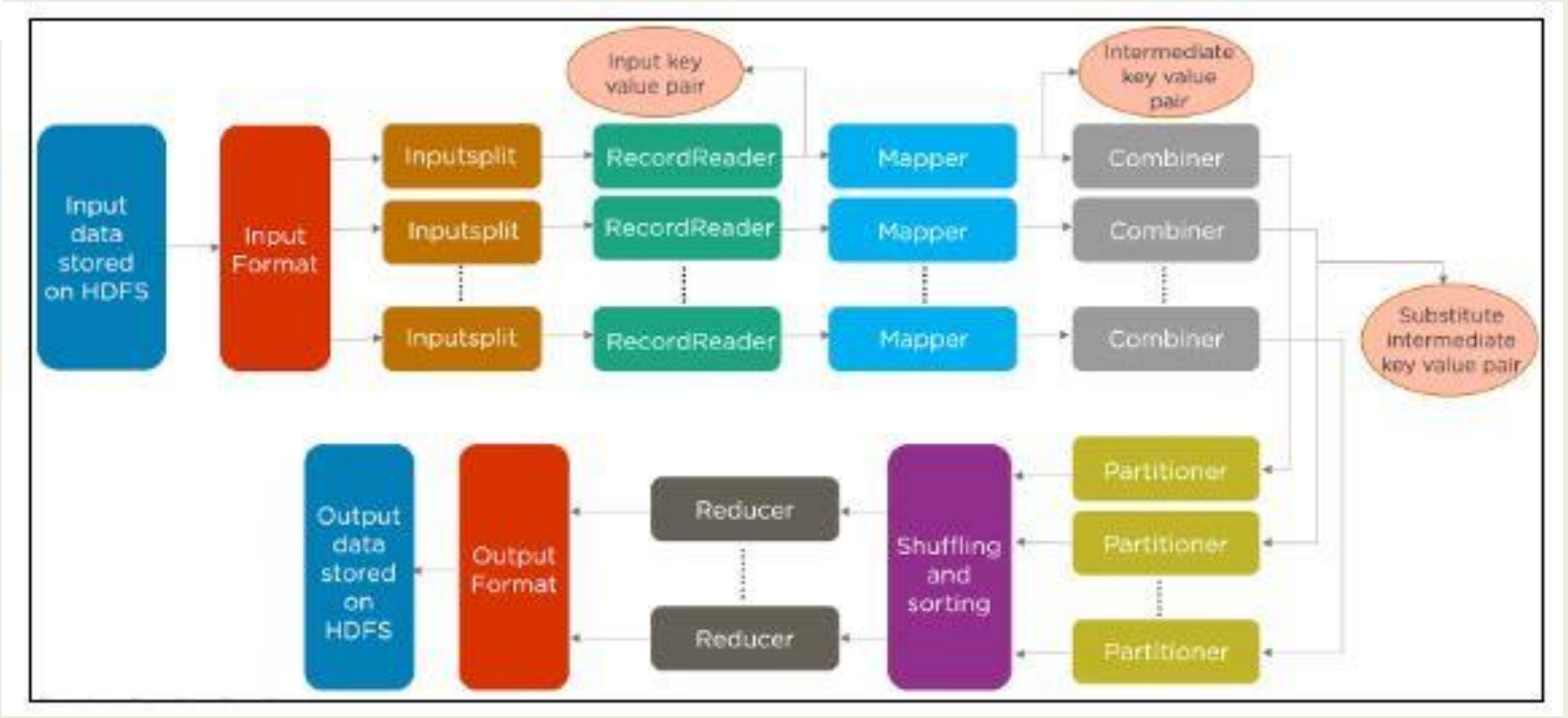


Map Reduce



Batch processing overview

Job Processing overview



Map Reduce

mapper.py

```python
#!/usr/bin/env python
"""mapper.py"""

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print('\n' % (word, 1))
```

reducer.py

```python
#!/usr/bin/env python
"""reducer.py"""

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print('\n' (current_word, current_count))
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print('%s\t%s' % (current_word, current_count))
```

```
Hassan.Arshad.Confiz@ubuntu:~$ ls -l /tmp/demo/
total 3604
-rw-r--r-- 1 Hassan.Arshad.Confiz hadoop  674566 May  17 10:17 pg20417.txt
-rw-r--r-- 1 Hassan.Arshad.Confiz hadoop 1573112 May  17 10:18 pg4300.txt
-rw-r--r-- 1 Hassan.Arshad.Confiz hadoop 1423801 May  17 10:18 pg5000.txt
Hassan.Arshad.Confiz@ubuntu:~$
```

```
>> Hassan.Arshad.Confiz@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -copyFromLocal /tmp/demo /user/Hassan.Arshad.Confiz/demo
>> Hassan.Arshad.Confiz@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -ls
>> Found 1 items
>> drwxr-xr-x   - Hassan.Arshad.Confiz supergroup          0 2024-05-17 17:40 /user/Hassan.Arshad.Confiz/demo
>> Hassan.Arshad.Confiz@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -ls /user/Hassan.Arshad.Confiz/demo
>> Found 3 items
>> -rw-r--r--   3 Hassan.Arshad.Confiz supergroup     674566 2024-05-17 11:38 /user/Hassan.Arshad.Confiz/demo/pg20417.txt
>> -rw-r--r--   3 Hassan.Arshad.Confiz supergroup    1573112 2024-05-17 11:38 /user/Hassan.Arshad.Confiz/demo/pg4300.txt
>> -rw-r--r--   3 Hassan.Arshad.Confiz supergroup    1423801 2024-05-17 11:38 /user/Hassan.Arshad.Confiz/demo/pg5000.txt
>> Hassan.Arshad.Confiz@ubuntu:/usr/local/hadoop$
```

```
Hassan.Arshad.Confiz@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/streaming/hadoop-*streaming*.jar \
-file /home/Hassan.Arshad.Confiz/mapper.py    -mapper /home/Hassan.Arshad.Confiz/mapper.py \
-file /home/Hassan.Arshad.Confiz/reducer.py   -reducer /home/Hassan.Arshad.Confiz/reducer.py \
-input /user/Hassan.Arshad.Confiz/demo/* -output /user/Hassan.Arshad.Confiz/demo-output
```

```
Hassan.Arshad.Confiz@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/streaming/hadoop-*streaming*.jar
-mapper /home/Hassan.Arshad.Confiz/mapper.py
-reducer /home/Hassan.Arshad.Confiz/reducer.py
-input /user/Hassan.Arshad.Confiz/demo/*
-output /user/Hassan.Arshad.Confiz/demo-output
 additionalConfSpec_:null
 null=@@@userJobConfProps_.get(stream.shipped.hadoopstreaming
 packageJobJar: [/app/hadoop/tmp/hadoop-unjar54543/]
 [] /tmp/streamjob54544.jar tmpDir=null
 [...] INFO mapred.FileInputFormat: Total input paths to process : 7
 [...] INFO streaming.StreamJob: getLocalDirs(): [/app/hadoop/tmp/mapred/local]
 [...] INFO streaming.StreamJob: Running job: job_200803031615_0021
 [...]
 [...] INFO streaming.StreamJob:  map 0%  reduce 0%
 [...] INFO streaming.StreamJob:  map 43%  reduce 0%
 [...] INFO streaming.StreamJob:  map 86%  reduce 0%
 [...] INFO streaming.StreamJob:  map 100%  reduce 0%
 [...] INFO streaming.StreamJob:  map 100%  reduce 33%
 [...] INFO streaming.StreamJob:  map 100%  reduce 70%
 [...] INFO streaming.StreamJob:  map 100%  reduce 77%
 [...] INFO streaming.StreamJob:  map 100%  reduce 100%
 [...] INFO streaming.StreamJob: Job complete: job_200803031615_0021
 [...] INFO streaming.StreamJob: Output: /user/Hassan.Arshad.Confiz/demo-output
Hassan.Arshad.Confiz@ubuntu:/usr/local/hadoop$
```

- **Steps:**

1. Mapper Function
2. Reducer Function
3. Downloaded files from internet
4. Uploaded those on HDFS
5. Ran MapReduce Job
6. Output of the job (Terminal)

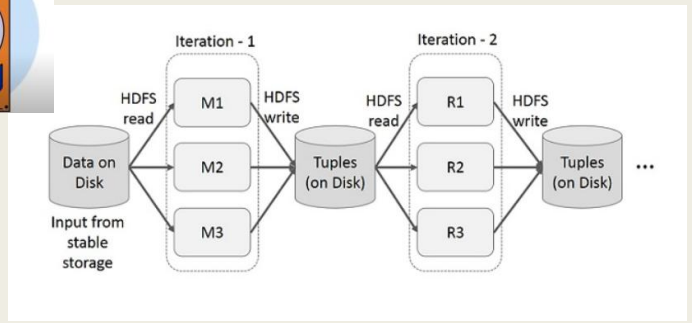
Advantages of Map Reduce

- **Fault tolerance**: It can handle failures without downtime.
- **Speed**: It splits, shuffles, and reduces the unstructured data in a short time.
- **Cost-effective**: Hadoop MapReduce has a scale-out feature that enables users to process or store the data in a cost-effective manner.
- **Scalability**: It provides a highly scalable framework. MapReduce allows users to run applications from many nodes.
- **Parallel Processing**: Here multiple job-parts of the same dataset can be processed in a parallel manner. This can reduce the task that can be taken to complete a task.

Hadoop Limitations (1)

Although Hadoop is good at storing and processing large amount of data, there were some limitations:

Relied on storing data on Disk

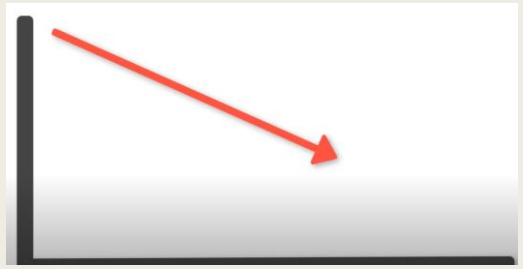


Every time we ran a job, it loads the data from disk, processes and stores the data back on disk.

Processes Data only in Batches

We have to wait for one batch run to execute before submitting a new job.

Which made things slower



There was the need to process data faster and in real-time manner

Apache Spark

Spark was developed as a research project by students at the California University in 2009

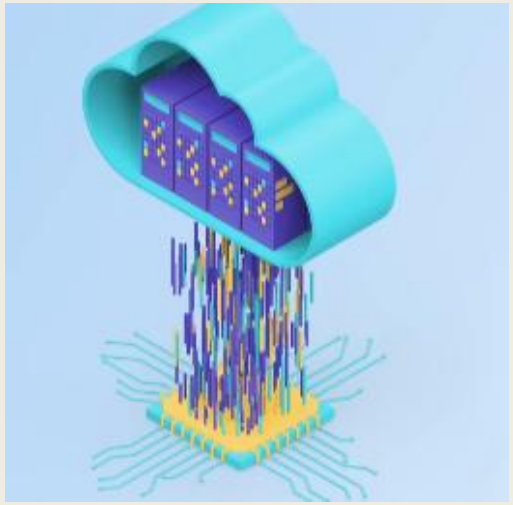
Spark was created to address the limitations of Hadoop

Apache Spark – RDDs

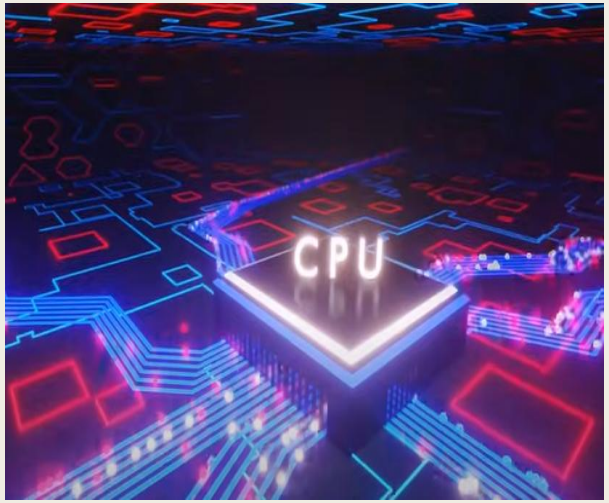
Works on the concept of RDDs
(Resilient Distributed Datasets)

RDDs are the backbone of Apache Spark

Allows Data to be stored In-Memory

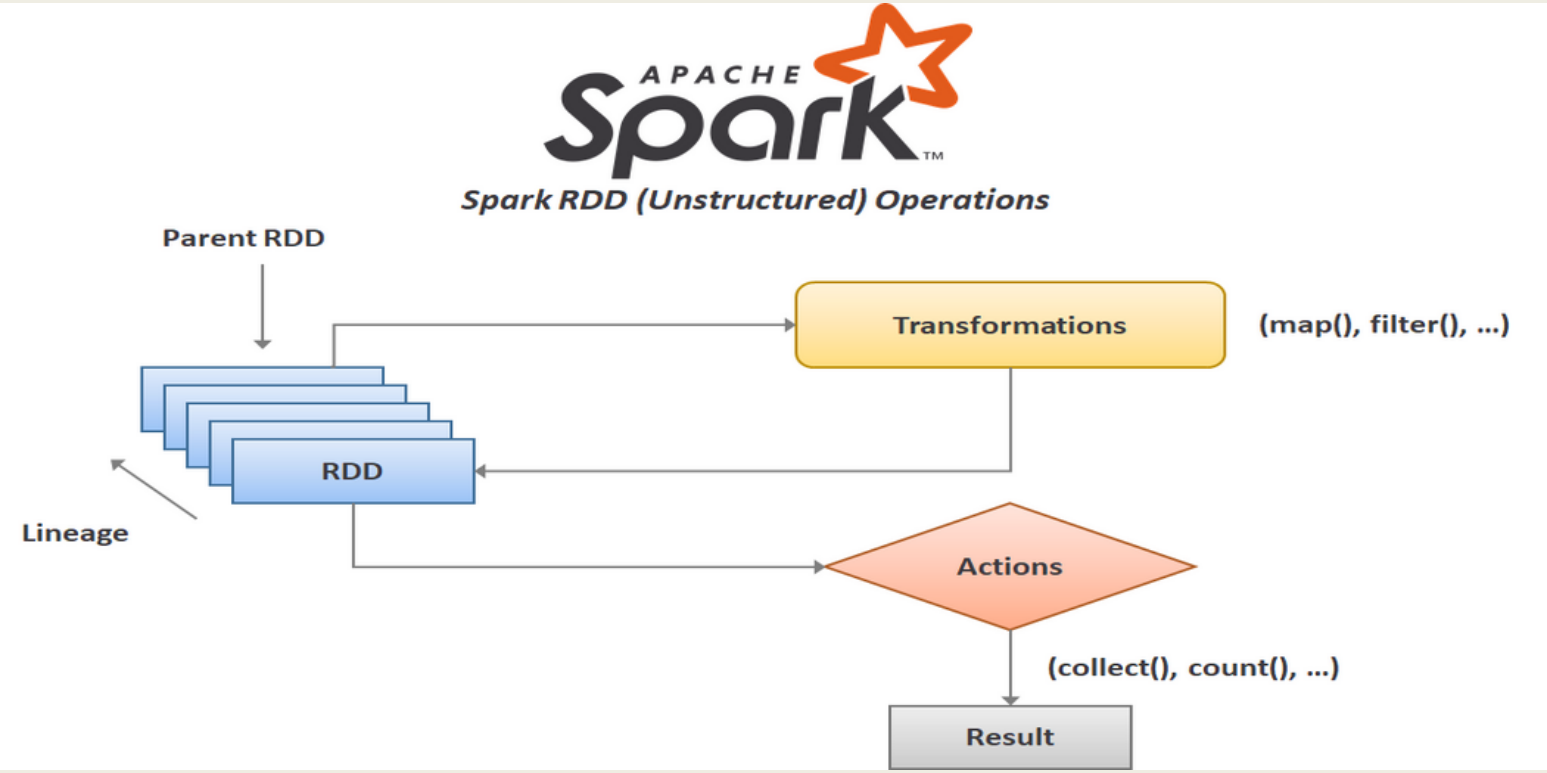


Enables faster data access



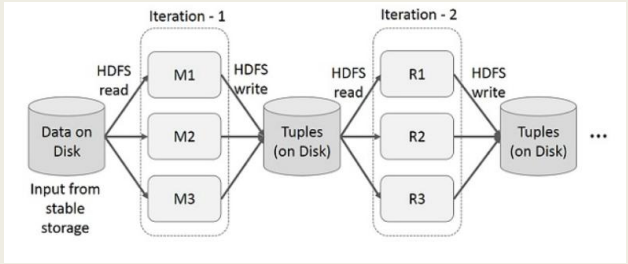
RDDs

RDDs (Resilient Distributed Datasets)

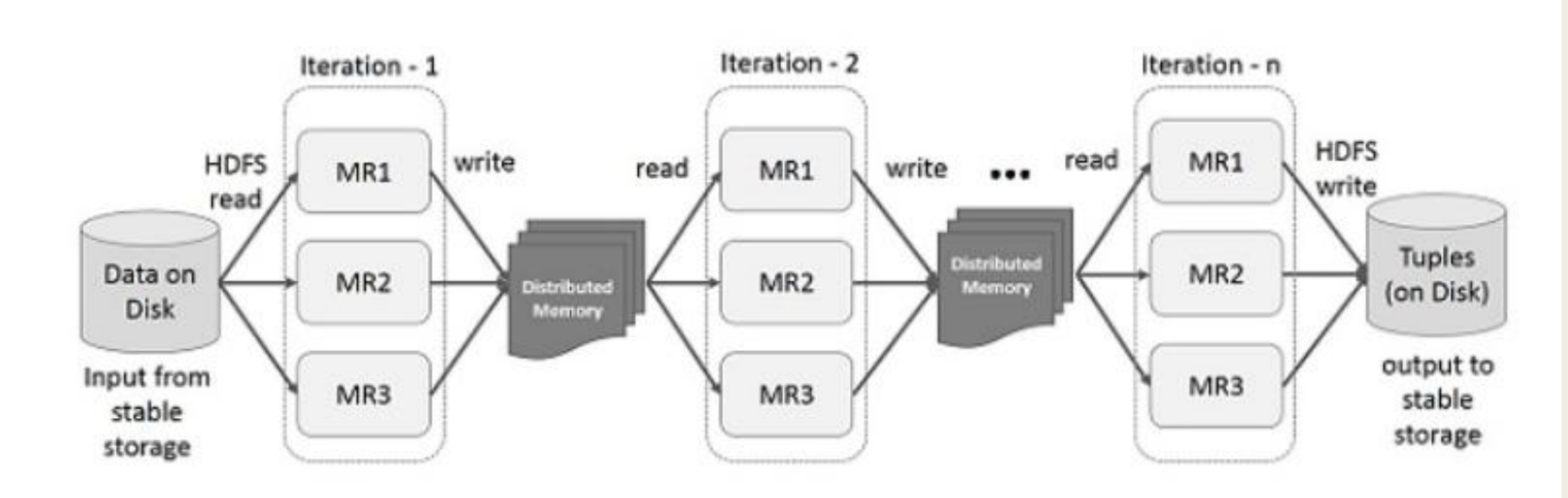


Iterative Operations – RDDs

RDDs (Resilient Distributed Datasets)

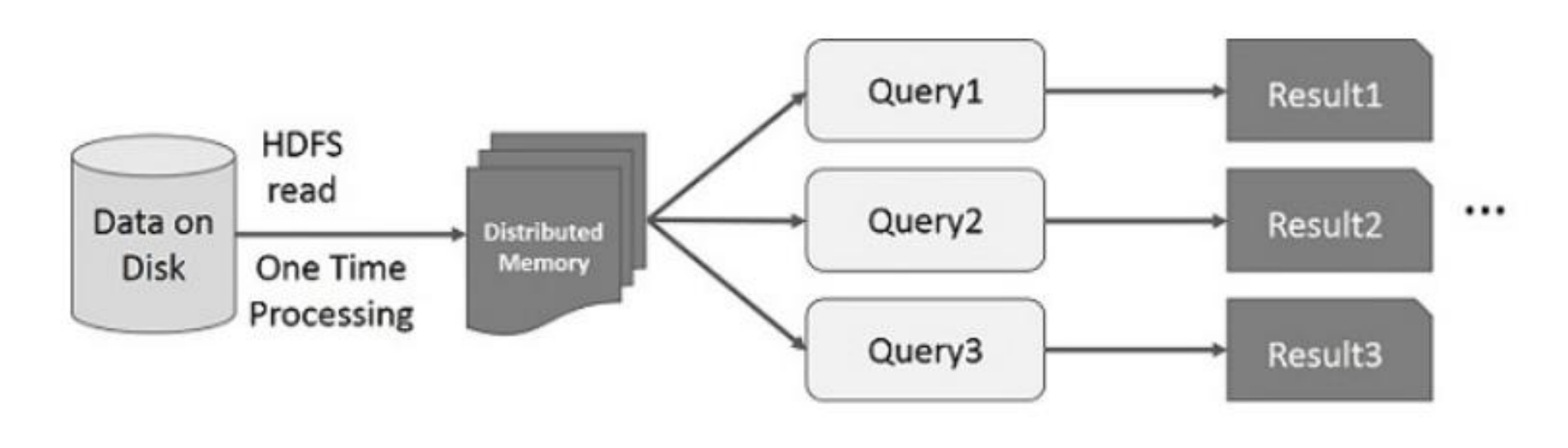


MapReduce

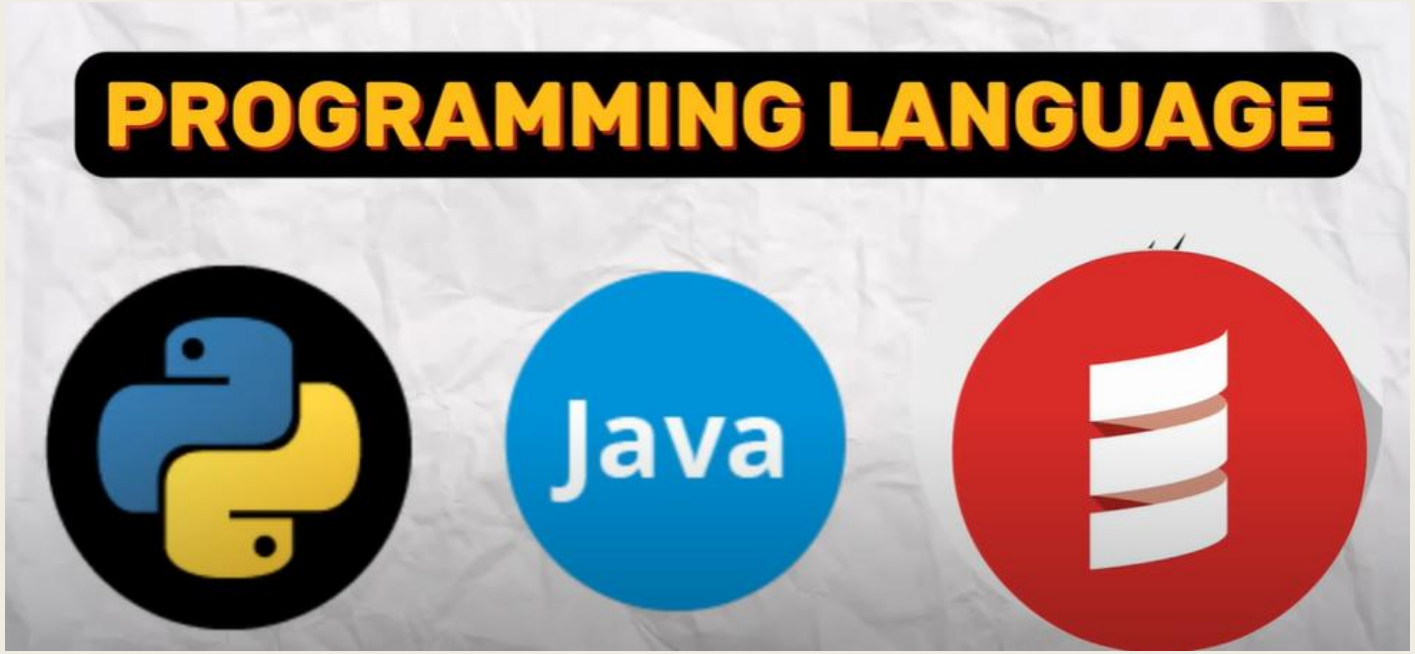


Interactive Operations – RDDs

RDDs (Resilient Distributed Datasets)



Spark compatibility with Languages

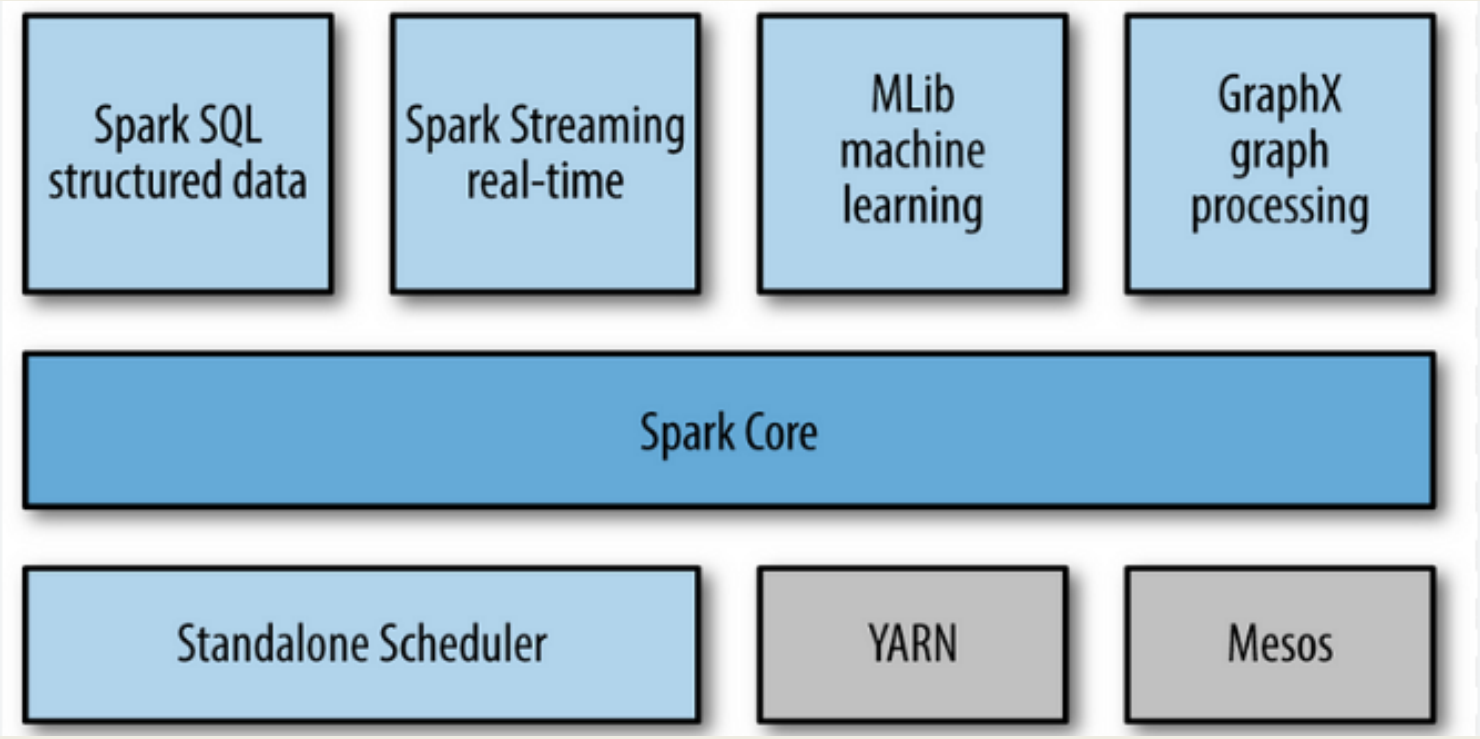


Spark & Hadoop

This RDD architecture and In-Memory processing makes

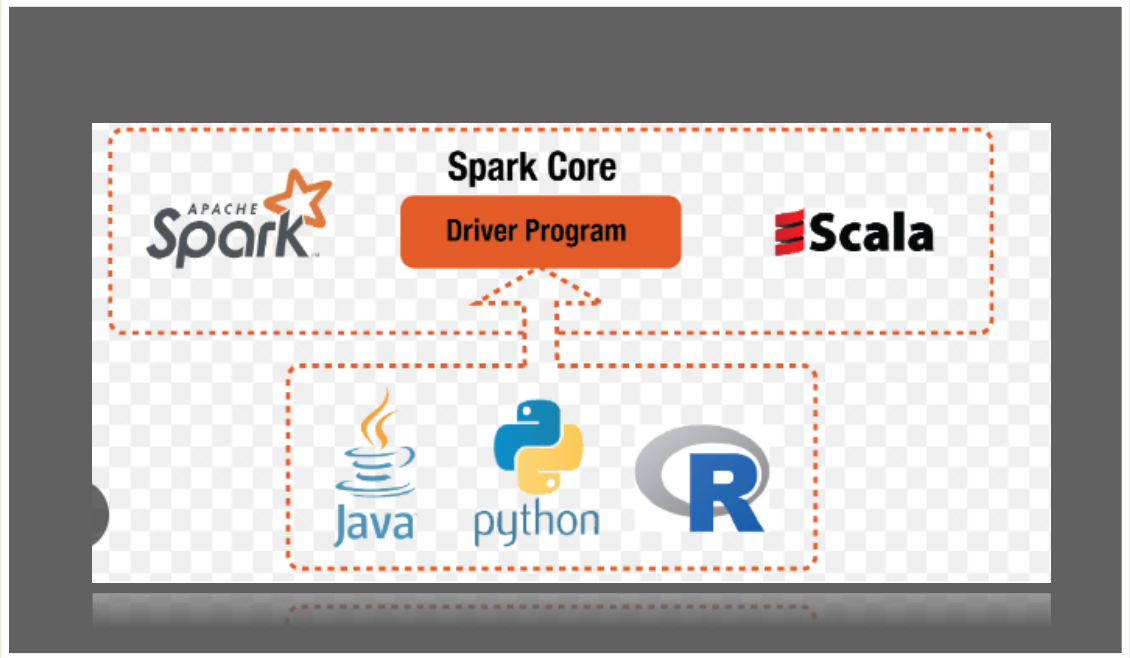
Spark 100x Faster than Hadoop

Components of Spark



Spark Core

Helps with processing data across multiple computers
It makes sure everything works efficiently and smoothly



Spark SQL

Allows writing SQL queries directly on your data through Spark

Spark Streaming

Helps you persist the real time data that you see in Google maps or Uber

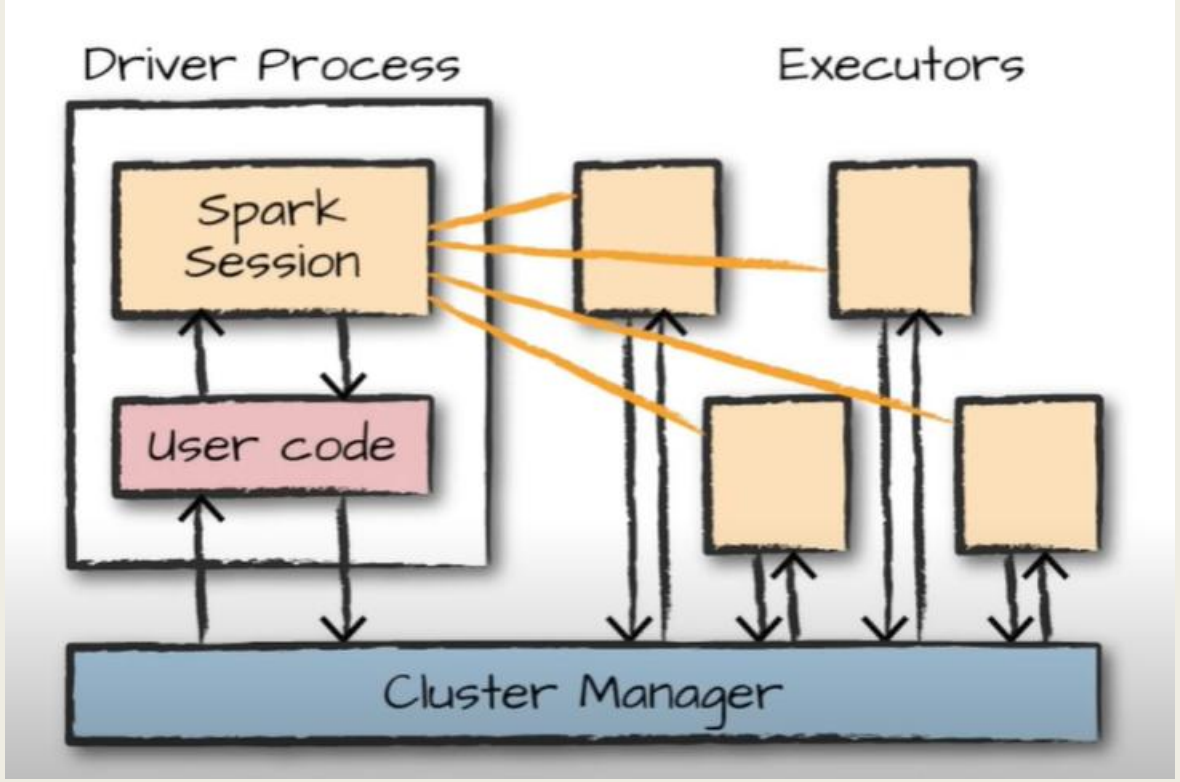
Spark MLlib

Helps you train the large scale Machine Learning models on Big Data

Spark Architecture

Since Spark distributes processing over different machines;
To coordinate work on all of the different machine we need a proper framework

Cluster Manager will grant resources to applications to complete the work



Spark Application – Components

Driver Process: Like a boss
Executor Processes: Like Workers

Driver keeps track of all the information about Apache Spark Applications
It responds to the commands and inputs from the user
Executors execute the tasks

Spark Application – Components

Driver divides the work into smaller tasks and assigns them to the executors and allocates the right resources based on the inputs we provide

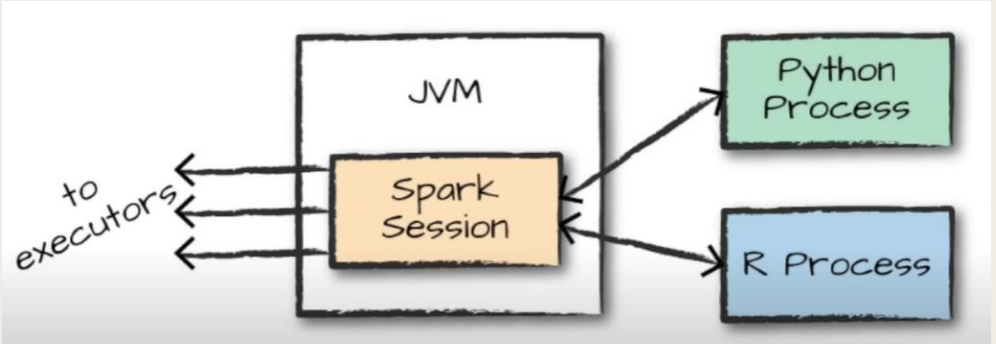
Executor Processes are the ones that actually does some work and executes the task assigned by driver process

Driver process is the heart of Apache Spark because it is responsible for getting the work done.

Reports back to the driver process, the progress and the results of the work done.

Spark Session

Creating Spark Session is the first thing that we do when working in Spark
Its like we are making the connection with the cluster manager



Scala

Spark is primarily written in Scala, making it Spark's "default" language. This book will include Scala code examples wherever relevant.

Java

Even though Spark is written in Scala, Spark's authors have been careful to ensure that you can write Spark code in Java. This book will focus primarily on Scala but will provide Java examples where relevant.

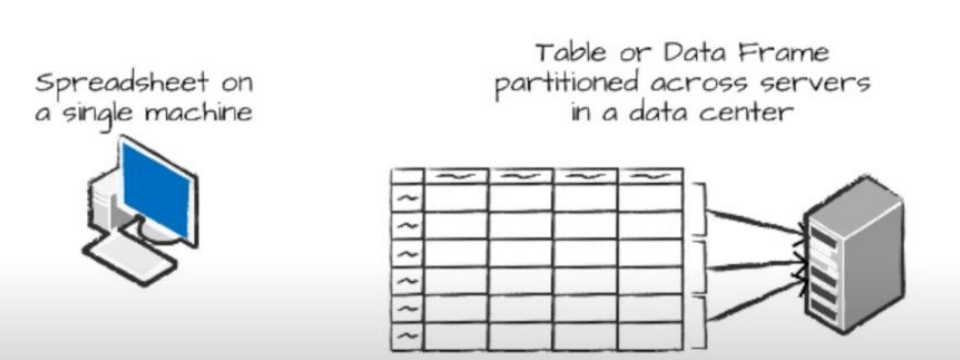
Python

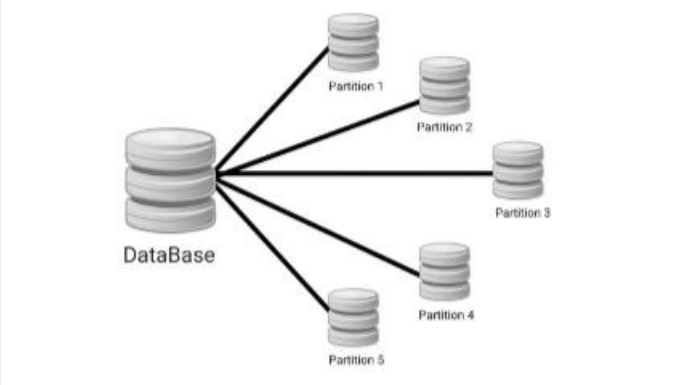
Python supports nearly all constructs that Scala supports. This book will include Python code examples whenever we include Scala code examples and a Python API exists.

We can create this session in any of the three languages

Spark DataFrame Distribution

Spark distributes the DataFrames among multiple computers





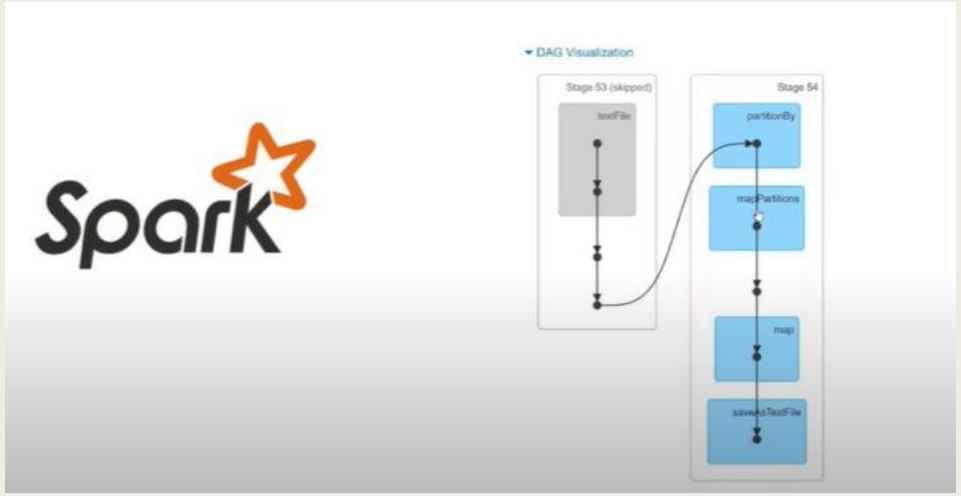
To make sure data is processed parallely we need to divide data into different chunks. This process is called partitioning.

All this is done using “Transformations”, which are instructions that tell Spark how to modify this data to get the desired results.

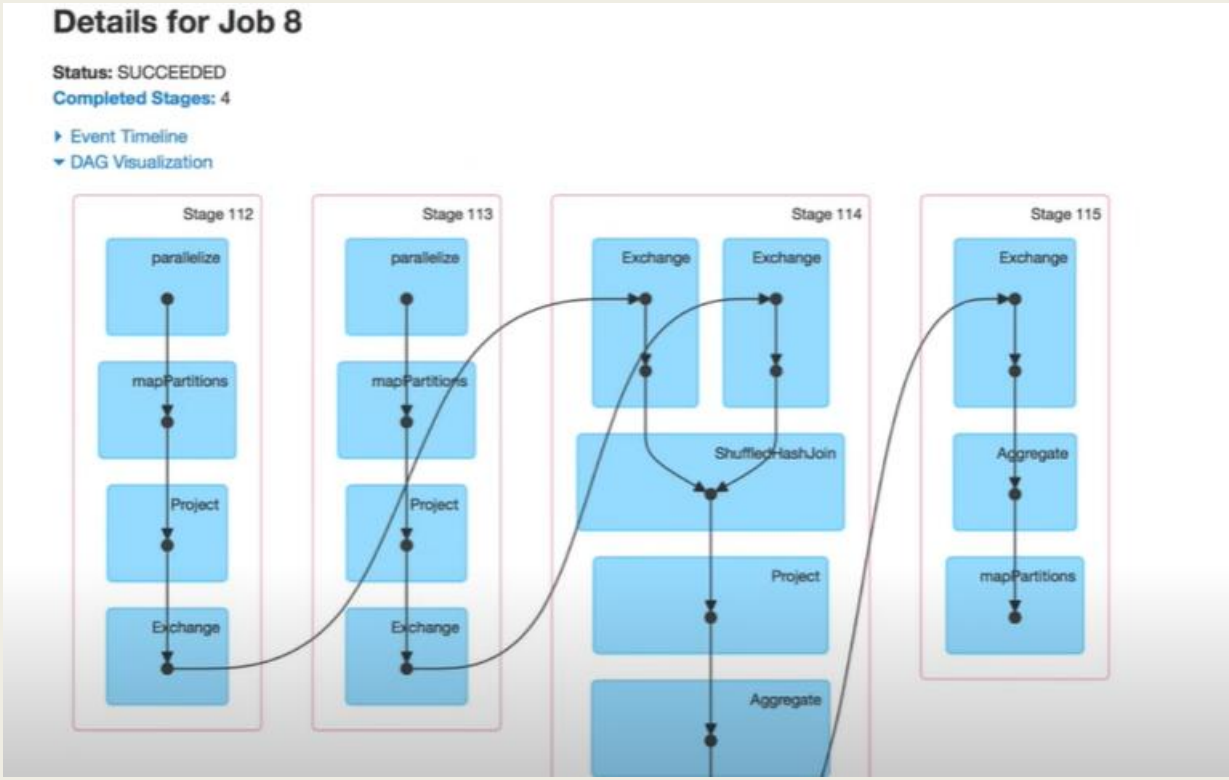
Spark Lazy Evaluations

Spark does not gives the results or processes the input as soon as you submit

Infact it will wait until you write your complete code and then it will generate plan to execute the code you have written

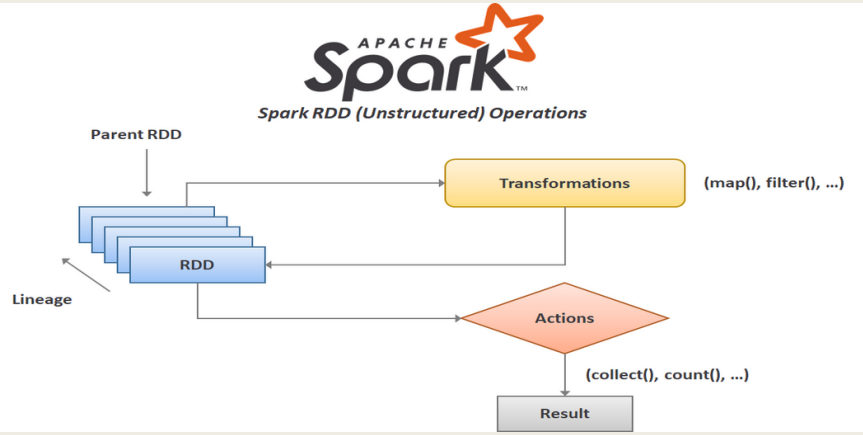


This allows spark to create a flow to process the code Efficiently



Spark Actions

Spark provides a list of Actions to execute our transformation blocks



Transformations	Actions
map (func)	reduce(func)
flatMap(func)	collect()
filter(func)	count()
groupByKey()	first()
reduceByKey(func)	take(n)
mapValues(func)	saveAsTextFile(path)
sample(...)	countByKey()
union(other)	foreach(func)
distinct()	...
sortByKey()	
...	

Spark Working

```python
import pandas as pd
import time

# Sample large dataset
data = ["word1 word2 word3", "word2 word3 word4", "word1 word3 word5"] * 100000000

# Start timing
start_time = time.time()

# Create DataFrame
df = pd.DataFrame(data, columns=['text'])

# Split words and explode
df = df['text'].str.split(expand=True).stack().reset_index(drop=True).to_frame('word')

# Group by word and count occurrences
word_counts = df['word'].value_counts()

# End timing
end_time = time.time()

print("Pandas processing time: {:.2f} seconds".format(end_time - start_time))
print(word_counts)
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_19000\364078481.py in <module>
     12
     13 # Split words and explode
---> 14 df = df['text'].str.split(expand=True).stack().reset_index(drop=True).to_frame('word')
```

Pandas Vs. Spark

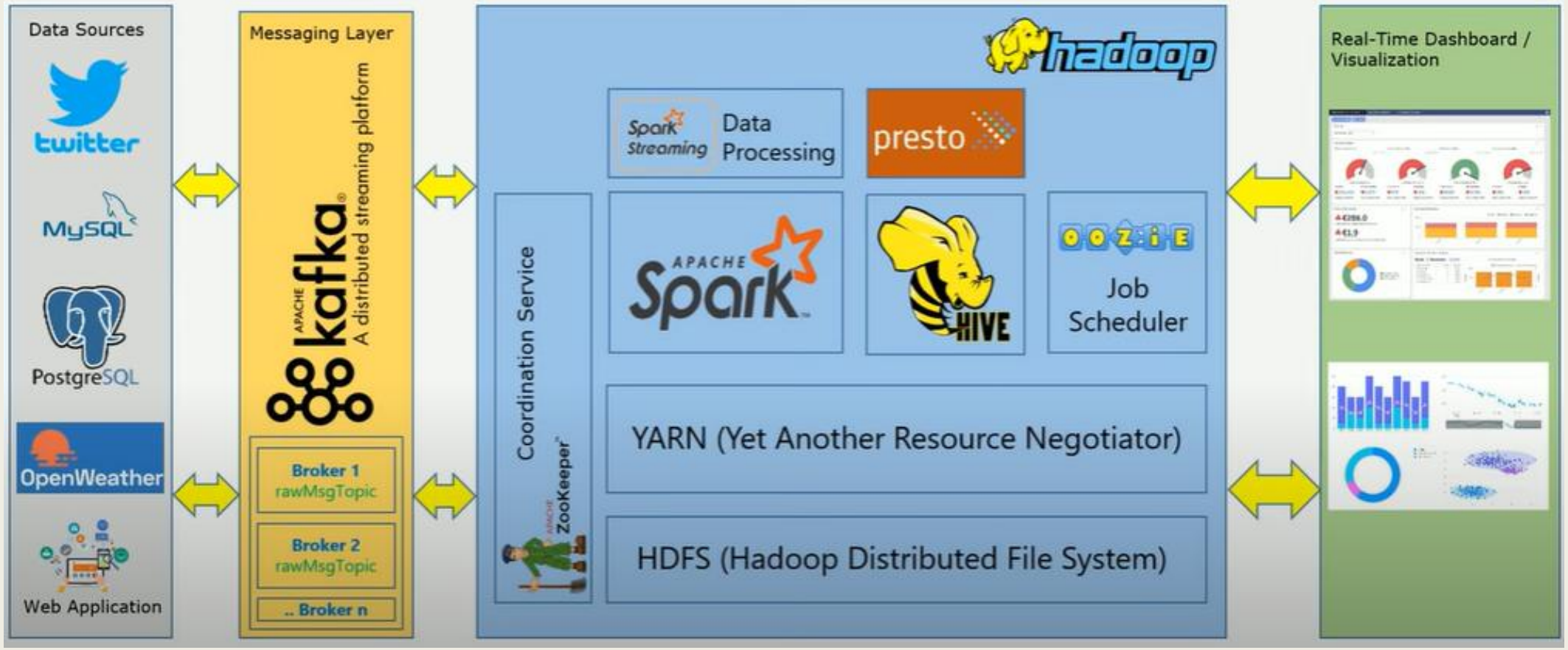
```python
from pyspark.sql import SparkSession
import time

# Initialize Spark session with all available cores
spark = SparkSession.builder \
    .appName("WordCount") \
    .master("local[*]") \
    .getOrCreate()

# Sample large dataset
data = ["word1 word2 word3", "word2 word3 word4", "word1 word3 word5"] * 100000000

# Start timing
start_time = time.time()

# Convert to RDD
rdd = spark.sparkContext.parallelize(data)

# Split lines into words, flatten, and map word counts
word_counts = rdd.flatMap(lambda line: line.split()) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(lambda a, b: a + b)

# Collect results
result = word_counts.collect()

# End timing
end_time = time.time()

print("Spark processing time: {:.2f} seconds".format(end_time - start_time))
print(result)

# Stop the Spark session
spark.stop()
```

```
Spark processing time: 26.35 seconds
[('word3', 3000000), ('word1', 2000000), ('word2', 2000000), ('word4', 1000000), ('word5', 1000000)]
```

Hadoop vs. Spark

- Key differences:
  - Batch processing vs. real-time processing.
- Complementary roles:
  - Hadoop for storage and batch processing
  - Spark for real-time processing.
- Performance comparison.

Hadoop + Spark
Real Time Architecture



High-Level Workflow: End User Perspective

- User workflow for data processing with Hadoop and Spark.
- Data ingestion, storage, processing, and analysis.
- Simplified steps from data input to output

How Hadoop Works Under the Hood

- HDFS: Data storage and replication.
- MapReduce: Parallel data processing.
- YARN: Resource management.

How Spark Works Under the Hood

- Spark Core: RDDs
- Spark execution model: In-memory computation.
- Role of the Spark Driver and Executors.

Time and Space Complexities

- Analysis of time complexity in MapReduce jobs.
- Space complexity considerations in HDFS.
- Spark's in-memory processing and impact on time/space complexity.

Benefits in Processing Large Data Sets

- Scalability and efficiency in data processing.
- Real-time analytics with Spark.
- Cost-effectiveness and flexibility.

Challenges and Issues

- Data skew and load balancing.
- Complexity in managing and tuning.
- Resource contention and overheads.

Case Studies and Real-World Applications

- Case studies of companies using Hadoop and Spark.
- Specific applications in various industries.
- Results and benefits achieved.

Thankyou!