Lecture 7 : Big Data Processing - HDFS, MapReduce, and Spark

Characteristics of Big Data

6 V’s of Big Data

V	Meaning	Explanation / Example
Volume	**Amount/size/quantity/(how much) of data**	Massive amounts of data — e.g., Facebook generates terabytes of user data daily.
Variety	**Different types/formats/(What kind) of data**	Structured (tables), semi-structured (JSON, XML), and unstructured (videos, images, text).
Velocity	**Speed/(how fast data moving) of data generation and processing**	Real-time streaming from IoT devices, financial trades, social media feeds, etc.
Value	**Usefulness/worth/(is data useful) of data for decision-making**	Data must provide insights, trends, or patterns that help businesses or organizations.
Variability	**Inconsistency or unpredictability in data flows (How data changes?)**	Data meaning changes over time — e.g., trending topics on Twitter vary hour by hour.
Veracity	**Accuracy/quality/reliability/(How trustworthy) of data**	Data can be noisy, incomplete, or misleading — requires cleaning and validation.

Hadoop MapReduce Function? [MeSSiR]

1. Map Phase

- Takes input data and processes it into intermediate **(key, value)** pairs.
- Each mapper works **independently and in parallel** on a chunk of data.

2. Shuffle and Sort Phase

- Groups and sorts the intermediate data by key.
- Ensures all values of the same key are sent to the same reducer.

3. Reduce Phase

- Takes grouped **(key, list of values)** and produces final output.
- Often used to **aggregate**, **summarize**, or **transform** data.

🛠️ Basic Flow Diagram

```
Input Data → Map → Shuffle & Sort → Reduce → Output
```

Hadoop MapReduce pseudo-code:

Example-1 : Word Count

Word Count Detailed Example:

- Goal: Count occurrences of each word in a document corpus.
- Map: For each word in a line, emit (word, 1).
- Shuffle & Sort: Group all (word, 1) pairs by word.
- Reduce: Sum counts for each word.

```
map(document):
    for word in document.split():
        emit(word, 1)
reduce(word, counts):
    emit(word, sum(counts))
```

Example-2 : Average Calculation

Average Calculation Example:

- Goal: Compute average value of numbers in a dataset.
- Map: Emit ("key", (number, 1))
- Reduce: Sum numbers and counts, then calculate average.

```
map(record):
    emit("key", (record.value, 1))
reduce("key", list_of_values):
    total,count = 0, 0
    for value, c in list_of_values:
        total += value
        count += c
    emit("key", total / count)
```

Example-3 : Compute average word length in a corpus. [No need for exam]

MapReduce math

understanding using numeric solution

flow chart

Numerical Example

We will be using **MovieLens Data.**

USER_ID	MOVIE_ID	RATING	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	[illegible]	[illegible]	[illegible]

244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

Solution

Step 1: First we have to map the values , it is happen in 1st phase of Map Reduce model.

196:242 ; 186:302 ; 196:377 ; 244:51 ; 166:346 ; 186:274 ; 186:265

Step 2*:* After Mapping we have to shuffle and sort the values.

166:346 ; 186:302,274,265 ; 196:242,377 ; 244:51

Step 3*:* After completion of step1 and step2 we have to reduce each key's values.

Now, put all values together

Mapper

↓

196:242 ; 186:302 ; 196:377 ; 244:51 ; 166:346 ; 186:274 ; 186:265

↓

Shuffle and Sort

↓

166:346 ; 186:302,274,265 ; 196:242,377 ; 244:51

↓

Reducer

↓

166:1 ; 186:3 ; 196:2 ; 244:1

MapReduce MATH 1

MapReduce Math - 1

apple box cat dog apple cat	Bangladesh cat dog box cat

Map:

apple: 1 box: 1
cat: 1 dog: 1
apple: 1 cat: 1

Bangladesh: 1
cat: 1 dog: 1
box: 1 cat: 1

Shuffle & Sort:

apple: [1, 1]
box: [1, 1]
cat: [1, 1, 1, 1]
dog: [1, 1]
Bangladesh: [1]

Reducer:

apple: 2
box: 2
cat: 4
dog: 2
Bangladesh: 1

MapReduce MATH 2

Math - 2

Map Reduce

10	20	30
10	20	30
40	50	60
40	50	60

Map:

10:1 , 20:1 , 30:1
10:1 , 20:1 , 30:1

40:1 , 50:1 , 60:1
40:1, 50:1 , 60:1

Shuffle & sort:

10 : [1, 1]
20: [1, 1]
30 : [1, 1]
40 : [1, 1]
50 : [1, 1]
60 : [1, 1]

Reducer:

10 : 2
20 : 2
30 : 2
40 : 2
50 : 2
60 : 2

Hadoop MapReduce vs Apache Spark

Feature	Hadoop MapReduce	Apache Spark
Processing	Disk-based after each phase	In-memory (RAM) with optional disk spill
Iterative Tasks	Slow due to repeated disk I/O	Fast using RDD/DataFrame caching
Programming	Low-level Java API	High-level APIs in Python, Scala, Java, R
Speed	Good for simple, one-pass batch jobs	Up to 100x faster for complex or iterative workloads
Use Case Fit	Ideal for large-scale ETL and linear batch jobs	Ideal for interactive, iterative jobs (e.g., machine learning, graph processing)

🧠 Summary:

- **Hadoop MapReduce** is disk-heavy, reliable, and best for **sequential batch jobs**.
- **Apache Spark** is memory-centric, faster, and better suited for **real-time, iterative, and ML tasks**.

🔍 1. Processing

✔️ Hadoop MapReduce:

- Processes data in **stages**, writing intermediate results to **disk** after every **Map** or **Reduce** phase.
- This ensures fault tolerance, but causes **slower performance** due to heavy **disk I/O**.

✔️ Apache Spark:

- Processes data **in memory**, meaning intermediate data is stored in **RAM**, not disk (unless necessary).
- This significantly **improves speed** for multi-stage or iterative computations.

🔄 2. Iterative Tasks

✔️ Hadoop MapReduce:

- Each job must read from and write to disk every time, even if the same data is reused.
- This makes it **inefficient** for iterative algorithms like machine learning or graph processing.

✔️ Apache Spark:

- Supports **in-memory caching** of datasets using **RDDs (Resilient Distributed Datasets)** or **DataFrames**.
- This makes it ideal for **reusing data across multiple operations**, resulting in **faster performance**.

💻 3. Programming

✔️ Hadoop MapReduce:

- Mostly uses **low-level Java APIs**.
- More **boilerplate code** is needed for writing and reading data, managing mappers/reducers, etc.

✔️ Apache Spark:

- Offers **high-level APIs** in **Scala, Python (PySpark), Java, and R**.
- Provides **simple functions** for map, filter, join, groupBy, etc., making development **faster and easier**.

⚡ 4. Speed

✔️ Hadoop MapReduce:

- Decent for **simple batch jobs** that need to scan large datasets once (e.g., logs processing).
- Slower for complex logic due to reliance on disk.

✔️ Apache Spark:

- **Up to 100x faster** than MapReduce for **complex, multi-step jobs**.
- Especially efficient for ML, streaming, and graph algorithms.

🎯 5. Use Case Fit

✔️ Hadoop MapReduce:

- Best for **large, one-pass data processing** tasks like ETL (Extract, Transform, Load), indexing, and archiving.

- Good when memory is limited, and reliability is key.

✔️ Apache Spark:

- Ideal for **interactive data analysis**, **real-time processing**, and **machine learning workflows**.
- Frequently used in modern data platforms due to its flexibility and speed.

🧠 In Short:

If your job is...	Choose...
Heavy, one-time processing on huge datasets	Hadoop MapReduce
Fast, repeated access to the same dataset	Apache Spark
Real-time or interactive	Apache Spark
Memory-constrained, disk-safe batch jobs	Hadoop MapReduce

Apache Spark – Key Characteristics

Characteristic	Explanation
Lazy Evaluation	Builds a **Directed Acyclic Graph (DAG)** of execution for better optimization.
In-Memory Computation	Stores intermediate results in RAM for faster performance than disk-based systems (like MapReduce).
Speed	Up to **100x faster** than Hadoop MapReduce for complex or iterative tasks.
Distributed Processing	Automatically distributes data and tasks across multiple nodes in a cluster.
Ease of Use	Supports high-level **APIs in Scala, Python (PySpark), Java, R**, and SQL.
Fault Tolerance	Uses **RDD lineage** to recover lost data without needing full data replication.

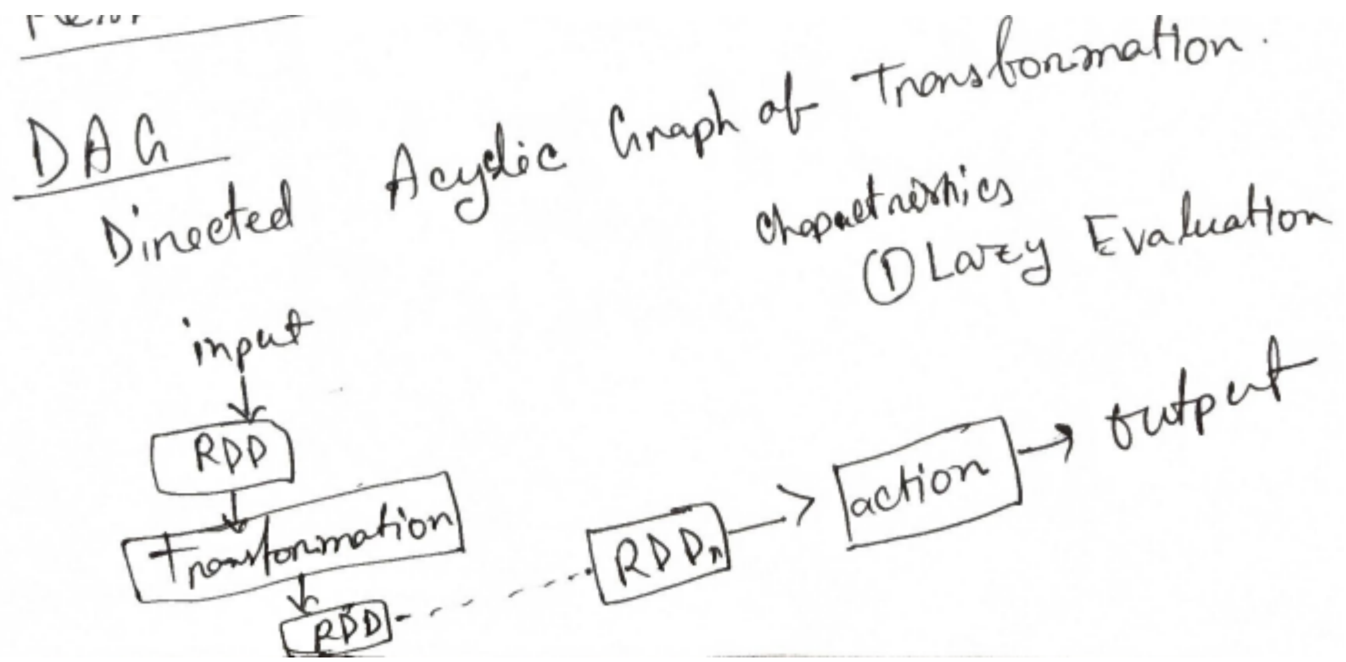
Characteristic	Explanation
Unified Engine	Handles **batch processing, streaming, machine learning, and graph processing**.
Rich Libraries	Includes **Spark SQL**, **Spark MLlib**, **Spark Streaming**, **GraphX**, etc.
Scalability	Scales from a laptop to thousands of nodes — suitable for both small and big data.
Integration Support	Integrates with **Hadoop (HDFS), Hive, HBase, Cassandra, Kafka, S3**, etc.

🧠 Summary

Apache Spark is:

- **Fast** (in-memory & parallel)
- **Flexible** (multiple languages & workloads)
- **Unified** (one engine for many tasks)
- **Scalable** (from GBs to petabytes)
- **Extensible** (via libraries and external data sources)

DAG



🔹 What is RDD in Apache Spark?

RDD stands for **Resilient Distributed Dataset**

✔️ Key Features of RDD

Feature	Description
Lazy Evaluation	Transformations are only executed when an action is called
Resilient	Fault-tolerant — automatically recovers lost data using **lineage (history)**
Distributed	Data is **automatically partitioned** across nodes in a cluster
Immutable	Once created, you **cannot modify** an RDD — every transformation creates a new RDD
In-Memory	Stored in RAM by default (fast), but can spill to disk if needed

RDD Operations

🔹 Transformations (return new RDDs)

🔸 Actions (trigger execution)