1. Introduction to Big Data Processing

Hadoop vs Spark

Apache Hadoop and Apache Spark are both used for **Big Data processing**, but they differ in architecture, speed, and ease of use.

Feature	Hadoop (MapReduce)	Apache Spark
Processing Speed	Slow (disk-based processing)	Fast (in-memory processing)
Ease of Use	Complex, requires extensive coding	Simple, supports multiple languages
Processing Modes	Batch processing only	Batch, streaming, SQL, ML, and graph processing
Fault Tolerance	Replication-based	RDD lineage-based recovery
Resource Management	Uses YARN	Can use YARN, Kubernetes, or Mesos

HDFS (Hadoop Distributed File System)

- A distributed file system that stores data across multiple nodes.
- Stores large datasets efficiently and allows parallel processing.

YARN (Yet Another Resource Negotiator)

- Manages cluster resources efficiently.
- **Alternatives:** Docker and Kubernetes for containerised environments.

2. Limitations of MapReduce

MapReduce is **not suitable for real-time or interactive processing** due to several reasons:

1. High Latency

- Writes intermediate data to disk, increasing **I/O overhead**.
- **Slow processing** due to disk dependency.
- Inefficient for **real-time analytics**.

2. Complex & Boilerplate Code

- Requires **separate Mapper and Reducer classes**.
- Hard to maintain and debug.

3. Only Supports Batch Processing

- **Cannot handle streaming data.**
- Inefficient for **real-time fraud detection, log processing, or monitoring.**

4. Rigid Execution Flow

- Strict **Map → Reduce** flow.
- **Difficult to implement custom workflows.**

5. No Interactive Mode & Limited Job Monitoring

- Jobs must **fully execute** before results are visible.
- **Debugging & optimization are difficult.**

3. Apache Spark: Overview & Features

What is Apache Spark?

- **An open-source distributed computing system** for processing large-scale data efficiently.
- **Much faster than Hadoop** due to **in-memory processing**.

Processing Capabilities

- **Batch Processing** – Like MapReduce but much faster.
- **Real-time Streaming** – Processes real-time data.
- **SQL Queries** – Query structured data efficiently.

- **Machine Learning (MLlib)** – Built-in ML support.
- **Graph Processing (GraphX)** – For graph-based computations.

Characteristics of Apache Spark

Feature	Description
In-Memory Processing	Reduces disk I/O, increasing speed.
Ease of Use	Supports **Scala, Python, Java, R**.
Unified Framework	Handles **batch, streaming, ML, and graph processing**.
Storage Flexibility	Works with **HDFS, Amazon S3, NoSQL, SQL databases**.

5. Apache Spark: RDD & Execution Flow

What is an RDD (Resilient Distributed Dataset)?

- The **fundamental data structure in Spark**.
- Represents **distributed collections of data** across partitions.

RDD Execution Steps

1. **Data Loading** – Read data from HDFS, S3, or databases.
2. **Partitioning** – Data is split into **smaller chunks (partitions).**
3. **Execution** – Spark processes each partition **in parallel**.

RDD Properties

Property	Description
Immutable	Cannot be modified after creation.
Partitioned	Distributed across nodes for parallel processing.
Fault-Tolerant	Can recover data from lineage.
Lazily Evaluated	Execution is delayed until an action is called.

6. Lazy Evaluation in Spark

Why Lazy Evaluation?

- Optimizes execution by **building a DAG (Directed Acyclic Graph).**
- **Only executes transformations when an action is triggered**.

Transformations (Lazy) vs Actions (Triggers Execution)

Operation Type	Description	Examples
Transformations	Creates a new RDD	`map()`, `filter()`, `flatMap()`
Actions	Executes all transformations	`collect()`, `count()`, `reduce()`

7. Transformations & Actions in RDD

Narrow Transformations (No Shuffling)

- Each output partition **depends on a single input partition**.
- **Faster execution.**
- **Examples:** `map()`, `filter()`, `flatMap()`.

Wide Transformations (Shuffling Required)

- Data is **redistributed across partitions**.
- **Slower due to network communication.**
- **Examples:** `reduceByKey()`, `groupByKey()`, `sortByKey()`.

Example

ReduceByKey vs GroupByKey in Spark

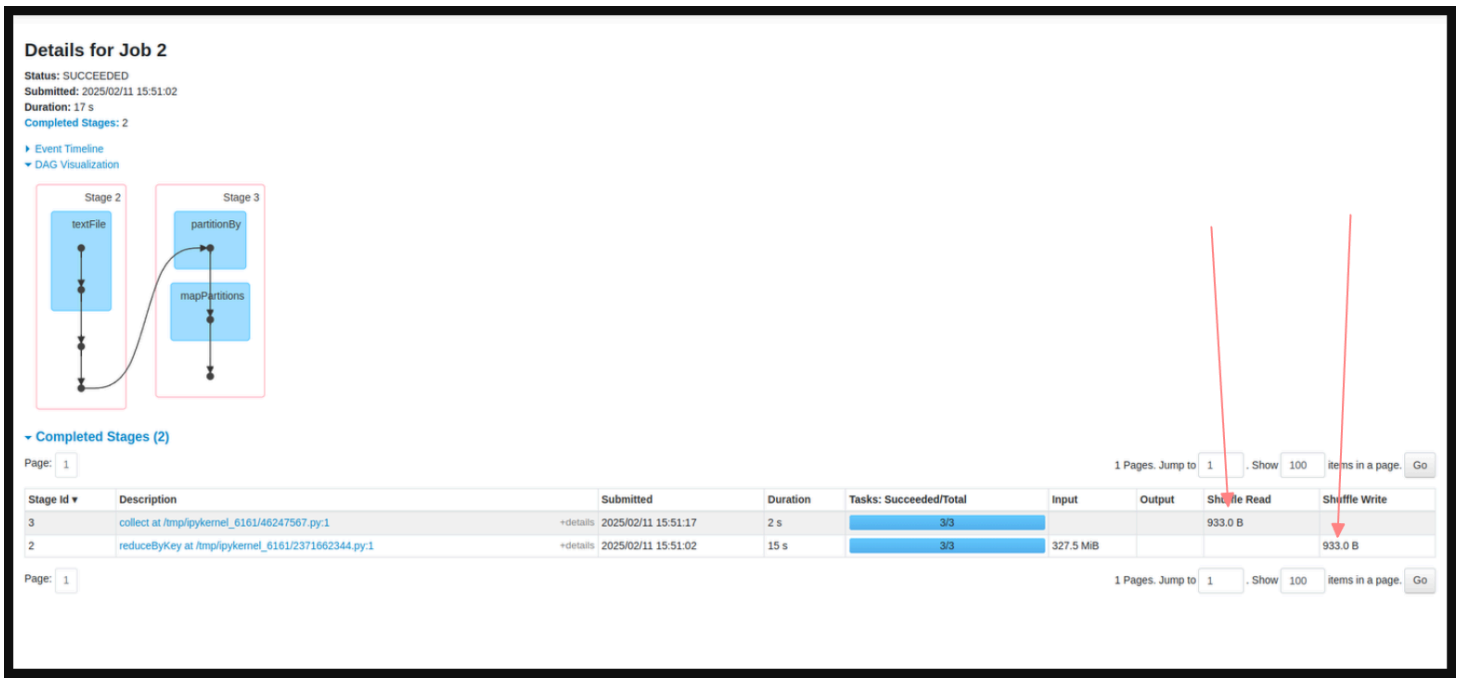
1. ReduceByKey

- **Type:** Wide Transformation

- **Definition:**
  - Aggregates values for each key using a **specified associative and commutative reduce function**.
  - **Combiner Optimization:** Combines the values **locally within each partition** before shuffling data across the network.
  - **Result:** Produces a single output value per key.
- **Example Function:**
  - `lambda x, y: x + y`
- **Key Points:**
  - Reduces the amount of data shuffled over the network.
  - More efficient for large datasets.
- **Output:**
  - RDD of `(key, aggregated_value)` pairs.
- **Example:**

  Input: `[(a, 1), (b, 2), (a, 3), (b, 4)]`

  Output: `[(a, 4), (b, 6)]`



2. GroupByKey

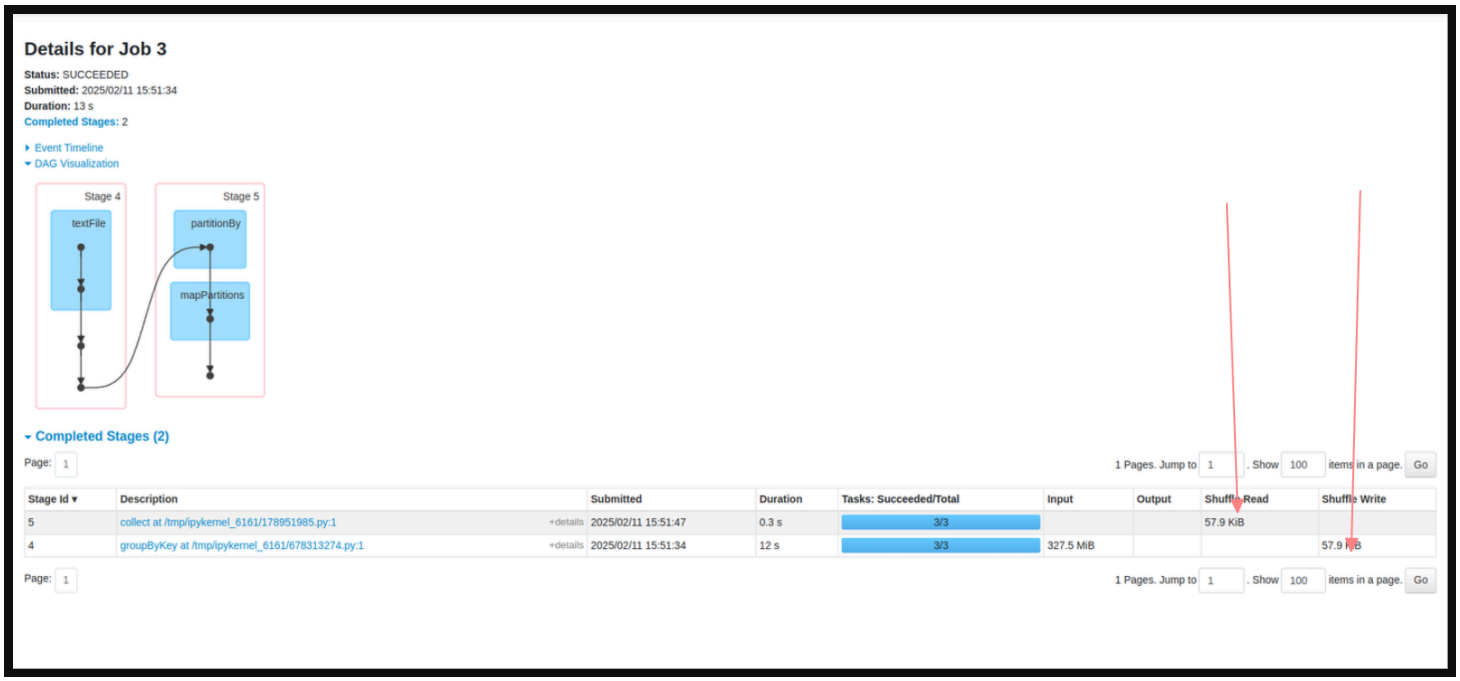
- **Type:** Wide Transformation
- **Definition:**
  - Groups all values associated with each key into a **single iterable collection**.
  - **No Aggregation:** Only groups data, does not perform aggregation.
- **Key Points:**
  - Shuffles all data across the network, which can be expensive for large datasets.

- Suitable when aggregation is not required and all values per key are needed.

- **Output:**
  - RDD of `(key, Iterable[values])` pairs.
- **Example:**

  Input: `[(a, 1), (b, 2), (a, 3), (b, 4)]`

  Output: `[(a, [1, 3]), (b, [2, 4])]`



Differences Between ReduceByKey and GroupByKey

Feature	ReduceByKey	GroupByKey
Operation Type	Aggregates values per key.	Groups values into an iterable per key.
Shuffling	Combines values locally before shuffling.	Shuffles all data directly.
Efficiency	More efficient for large datasets.	Less efficient due to higher shuffle cost.
Use Case	When aggregation is needed.	When all values for a key are required.
Output	`(key, aggregated_value)` pairs.	`(key, Iterable[values])` pairs.

Shuffle in Wide Transformations

- **Shuffle** involves **redistributing data** across the cluster.
- It is triggered when:
  - Data needs to be grouped or aggregated across partitions.
- Shuffling is **resource-intensive** and **time-consuming**, as it involves:
  - Writing intermediate data to disk.
  - Network communication between nodes.

Optimization Tip

- Try to minimize **wide transformations** (e.g., shuffles) to improve performance.
- Use **narrow transformations** whenever possible to avoid the overhead of data shuffling.

9. Higher-Level APIs: DataFrames & Spark SQL

Why Not Use RDDs?

- **Verbose, lacks schema support, and has no query optimization.**

What are Higher-Level APIs in Spark?

- Higher-level APIs include **DataFrame** and **SparkSQL**.
- These APIs provide schema-aware processing and are easier to use compared to RDDs.

Advantages of DataFrames & Spark SQL

1. **Schema Awareness** – Provides metadata for structured data.
2. **Performance Optimization** – Uses **Catalyst Optimizer & Tungsten Execution Engine**.
3. **Ease of Use** – Supports **SQL-like querying**.
4. **Integration** – Works with **JSON, Parquet, CSV, Avro**.

DataFrame

- A **distributed collection of data** organized into named columns.
- Similar to a **table** in a relational database or a **data frame** in Python's pandas or R.
- Supports SQL-like operations and is optimized for large-scale data processing.

Spark DataFrame Reading Process

Steps in `spark.read`

1. **Format** → Specifies the file format (e.g., CSV, JSON, Parquet).
2. **Header** → Determines whether the first row should be treated as column names.
3. **Infer Schema** → Automatically detects column data types (**Avoid This**).
4. **Load** → Reads data from a specified source (e.g., HDFS, S3, local storage).

Best Practices for Data Reading

- Avoid `inferSchema=True` (triggers extra jobs).
- **Manually define schema** for efficiency.

Behavior of `spark.read` in Different Scenarios

1. Without `inferSchema`, with `header=True`

- **Behavior:** Eager Evaluation
- **Explanation:**
  - Reads only the first line of the file to determine column names.
  - Triggers a lightweight job using `collect` with `limit 1`.

2. With `inferSchema=True`

- **Behavior:** Eager Evaluation
- **Explanation:**
  - Scans the entire dataset to infer the schema.
  - Triggers **two jobs**:
    a. Reads column names.
    b. Infers data types for each column.

3. With Explicit Schema Definition

- **Behavior:** Lazy Evaluation
- **Explanation:**
  - No upfront job is triggered.
  - Schema validation occurs only when an **action** is performed on the DataFrame.

Key Takeaways

- **Eager Evaluation:** Using `inferSchema=True` triggers jobs immediately.
- **Lazy Evaluation:** Defining the schema explicitly avoids upfront computation.
- **Performance Tip:**
  - For large datasets, explicitly defining the schema **improves performance** by preventing unnecessary scans.

SparkSQL

- Enables **querying structured data** using SQL.
- Provides **integration** between SQL and Python/Scala/Java APIs.

Schema Enforcement in Spark DataFrame

Challenges with `inferSchema=True`

1. **Incorrect Inference** → Spark might detect incorrect data types.
2. **Performance Issues** → Inferring schema requires scanning the dataset, increasing overhead.
3. **Not Suitable for Production** → Unreliable schema detection can lead to **inconsistencies** in data processing.

Issues with Headers & Schema Inference

- Headers are mapped to generic column names (`c1` , `c2` , etc.).
- Schema inference may produce **incorrect** or **inconsistent** results.

Schema Enforcement Techniques

To avoid the drawbacks of `inferSchema` , enforce schema using the following methods:

1. `StructType` (Recommended)

- **Why?**
  - Avoids incorrect type inference.
  - Ensures a consistent schema across datasets.
  - Enables **stricter validation**, preventing unexpected `NULL` values.

2. DDL String

- Schema can also be defined using a **Data Definition Language (DDL) string** for flexibility.

Creating Temporary, Global or Persistent View in Apache Spark

Persistence and Caching in Apache Spark

Why Caching is Needed in Spark?

Caching (or persistence) in Apache Spark is an optimization technique that helps store intermediate results in memory, reducing re-computation and improving performance. It is especially useful in the following cases:

1. Repeated Use of Data

- When a dataset is used multiple times across transformations or actions, recomputing it each time can be expensive.
- Caching helps store the dataset in memory, so subsequent operations can directly access it without recalculating.

2. Reducing Disk Reads

- Reading data from disk is significantly slower than reading from memory.
- If data is stored on disk, each access involves expensive I/O operations, which can degrade performance.

3.Lazy Evaluation Impact

1. Repeated Computation in Lazy Evaluation

- Spark follows **lazy evaluation**, meaning it builds a **DAG (Directed Acyclic Graph)** of transformations but doesn't execute them until an **action** is triggered.
- When an action is finally called, Spark computes the entire lineage to produce the results.

- If multiple actions are executed on the same dataset, Spark will **re-run the entire computation each time**, leading to redundant processing.

2. Why Caching is Required?

- **Memory is not unlimited:**
  - Spark processes everything in memory, but if the dataset is too large to fit into memory, Spark may need to recompute parts of it and spill data to disk.
  - With caching, intermediate results are **stored efficiently**, reducing unnecessary re-computation.
- **Lazy evaluation and re-computation:**
  - Without caching, Spark **recomputes everything** (entire lineage of transformations) whenever an action is triggered.
  - With caching, the dataset is **stored after the first computation**, eliminating redundant recalculations.

3.Reducing Disk & Network I/O Overhead

Even though Spark operates in memory, large datasets may require:

1. **Reading from disk (HDFS, S3, etc.)** when memory is insufficient.
2. **Fetching data from remote nodes** in a distributed environment, which adds network latency.

How and Where Caching Happens?

1. **Memory:**
   - Cached data is stored in a **deserialized format** for faster access.
   - If memory is insufficient, Spark **drops some cached data** to free up space.
2. **Disk:**
   - Data is **serialized and written (spill-over) to disk** when memory is full.
   - This helps when memory is limited but adds **disk I/O overhead**.

When to Use Caching in Spark?

1. **When the dataset is used multiple times**
   - Without caching, Spark **recomputes** the dataset every time an action is triggered.
   - Caching **avoids redundant computation**.
2. **When data is not small enough to fit entirely in memory**
   - If memory is limited, caching helps manage intermediate results efficiently.

3. **Avoid caching excessively in memory-constrained environments**
    - Caching too much data in memory can degrade **overall cluster performance**.
4. **When performance is critical**
    - If reducing execution time is a priority, caching can significantly improve efficiency.
5. **When data is expensive to compute**
    - If transformations involve complex operations, caching saves processing time.
6. **When data does not change frequently**
    - If the dataset remains **static**, caching prevents unnecessary re-computation.