Transformations and Actions in Apache Spark

I Mercia Jeno Saeshwaran
Jitheesh V J

February 28, 2025

Overview

1. Prerequisite
   - What is RDD?
2. Why do we need Transformations and Actions
3. Transformations in Spark
4. Narrow Transformations
5. Wide Transformations
6. Actions in Spark
7. Actions
8. An Example
9. Conclusion

What is RDD?

Resilient Distributed Dataset (RDD) is a collection of data elements that are spread across a cluster of nodes. RDDs are the primary API that users interact with in Spark

Key Features of RDD

- Distributed: Data is partitioned and distributed across multiple nodes.
- Immutable: Cannot be changed after creation; transformations create new RDDs.
- In-Memory Processing: Data is stored in RAM, making it much faster than traditional disk-based systems.

Why do we need Transformations and Actions

In Apache Spark, transformations and actions are essential because they allow for efficient distributed data processing by defining data manipulations (transformations) without immediately executing them, and then triggering the actual computation only when needed through actions, which enables lazy evaluation and optimization of complex data pipelines across a cluster.

How it differs from Hadoop:

Hadoop primarily uses the MapReduce paradigm, where data is processed in batches through distinct "map" and "reduce" phases, typically writing intermediate results to disk between steps, which can be slower compared to Spark's in-memory processing

What are Transformations?

Definition

Transformations are operations on RDDs (Resilient Distributed Datasets) that return another RDD, creating a lineage of transformations.

- Transformations are operations that create a new RDD/DataFrame from an existing one.
- They are lazy, meaning they do not execute immediately but are computed when an action is triggered.
- Transformations are only executed when an action like collect() or count() is called. This improves efficiency by optimizing execution plans.

Types of Transformations

- **Narrow Transformations** - Data movement occurs within the same partition (e.g., `map`, `filter`).
- **Wide Transformations** - Data movement occurs across partitions (e.g., `reduceByKey`, `groupByKey`).

Example

Input Data

	Name	Salary	Dept
Partition 1	Tom	10000	IT
Partition 2	Mike	20000	IT
Partition 3	John	30000	ADMIN

Output Data 1

Name	Salary	Dept	
Tom	10000	IT	Partition 1
Mike	20000	IT	Partition 2

Filter Transformation(Narrow)
-> df.filter(col("salary") < 25000)

Output Data 2

	Dept	Salary
Partition 1 +2	IT	30000
Partition 3	ADMIN	30000

Group by Transformation (Wide)
-> df.groupBy(Dept).agg(sum("salary"))

Narrow Transformation

Definition: Narrow Transformation in Apache Spark is a transformation where each partition of the parent RDD is used by at most one partition of the child RDD. These transformations do not require data movement (shuffling) between partitions, making them more efficient.
Examples :

- `map()`: Applies a function to each element independently.
- `filter()`: Filters elements based on a condition.
- `flatMap()`: Maps each input to multiple outputs.

Wide Transformation

Definition: Wide Transformation in Apache Spark is a transformation where data from multiple partitions of the parent RDD is required by one or more partitions of the child RDD. This involves shuffling data across nodes, making it more expensive. Examples :

- `groupByKey()`: Groups values with the same key, requiring data movement.
- `reduceByKey()`: Aggregates values by key, leading to a shuffle.
- `join()`: Combines two datasets based on a key, requiring redistribution.

What are Actions?

Definition

Actions trigger execution of transformations and return results.

- Examples: `count()`, `collect()`, `reduce()`.
- Actions execute the lineage of transformations to compute results.

Key Characteristics:

- Unlike transformations (which are lazy), actions execute immediately.
- Actions force computation on an RDD or DataFrame.
- They return values or save results to disk.

Actions

Common Actions in Spark:

- `collect()`: Retrieves all elements of an RDD or DataFrame.
- `count()`: Returns the number of elements.
- `first()`: Retrieves the first element.
- `take(n)`: Returns the first `n` elements.
- `reduce(func)`: Aggregates elements using a function.
- `foreach(func)`: Applies a function to each element.
- `saveAsTextFile(path)`: Writes RDD data to a text file.

An Example

Example:

```
$ head /data/flight-data/csv/2015-summary.csv

DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count
United States,Romania,15
United States,Croatia,1
United States,Ireland,344
```

An Example

Taking 3 records from a CSV

```
# in Python
flightData2015 = spark\
  .read\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .csv("/data/flight-data/csv/2015-summary.csv")
flightData2015.take(3)
```

Output

```
Array([United States,Romania,15], [United States,Croatia...
```

An Example

Figure: Taking 3 records after reading flightData2015

Another Example

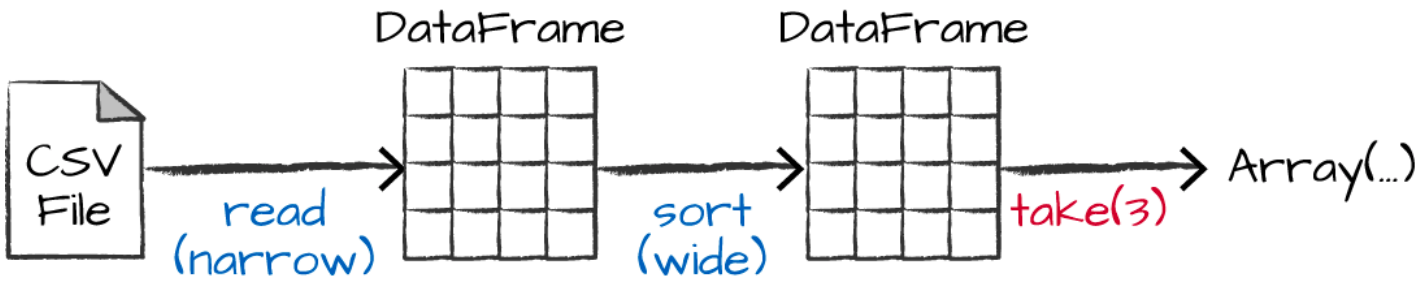

Figure: Reading, Sorting and Collecting a DataFrame

Another Example

Reading, Sorting and Collecting

```
flightData2015.sort("count").explain()
```

Output

```
== Physical Plan ==
*Sort [count#195 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#195 ASC NULLS FIRST, 200)
   +- *FileScan csv [DEST_COUNTRY_NAME#193,
        ORIGIN_COUNTRY_NAME#194,count#195] ...
```

Another Example

Reading, Sorting and Collecting

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
flightData2015.sort("count").take(2)
```

Output

```
... Array([United States,Singapore,1],
    [Moldova,United States,1])
```

An Example

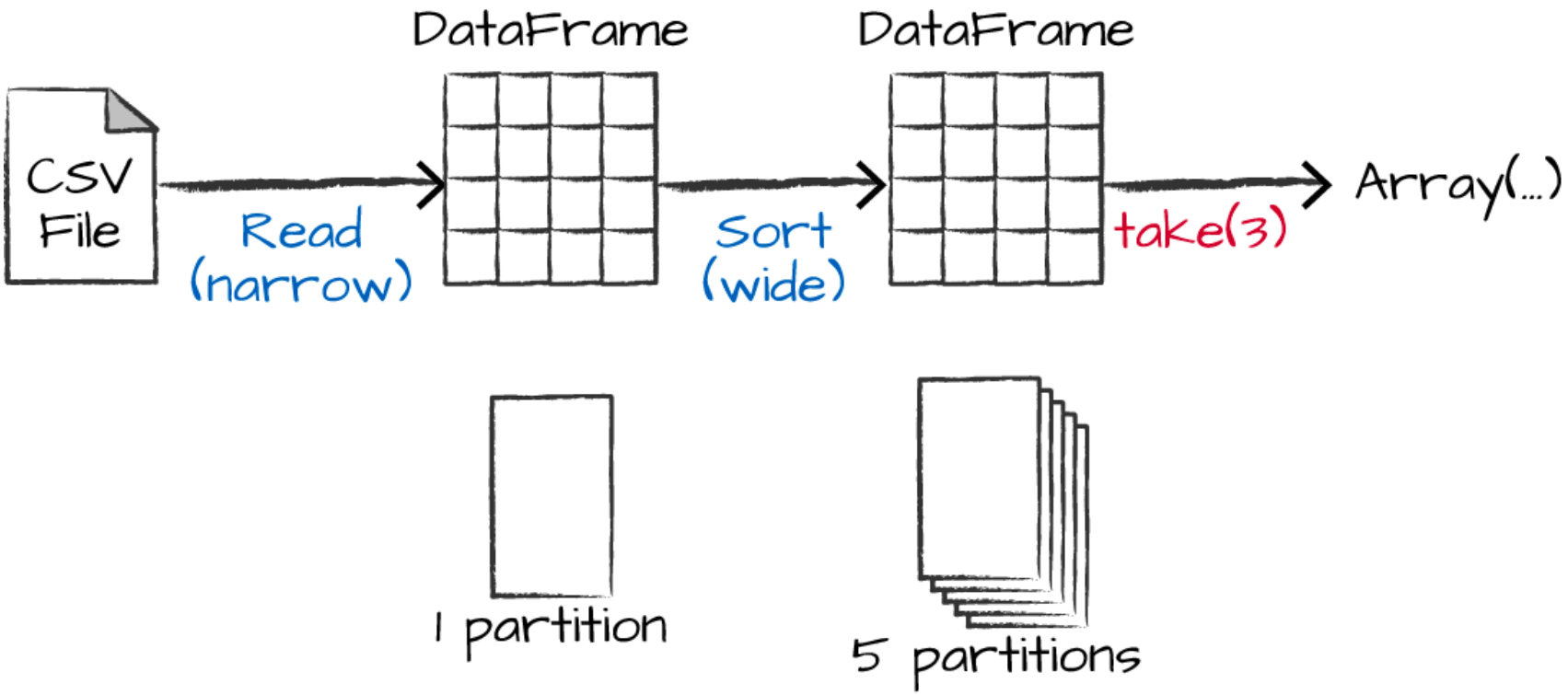

Figure: Logical and Physical Dataframe Manipulation

DataFrames and SQL

From DataFrame to table or view

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

Output

```
sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

dataFrameWay = flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .count()

sqlWay.explain()
dataFrameWay.explain()
```

DataFrames and SQL

Output

```
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182],
 functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
   +- *HashAggregate(keys=[DEST_COUNTRY_NAME#182],
       functions=[partial_count(1)])
        +- *FileScan csv [DEST_COUNTRY_NAME#182] ...
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182],
 functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
   +- *HashAggregate(keys=[DEST_COUNTRY_NAME#182],
       functions=[partial_count(1)])
        +- *FileScan csv [DEST_COUNTRY_NAME#182] ...
```

A More Complex Query

Top 5 destination countries in our data

```
# in Python
maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")

maxSql.show()
```

A More Complex Query

Top 5 destination countries in our data

DEST_COUNTRY_NAME	destination_total
United States	411352
Canada	8399
Mexico	7140
United Kingdom	2025
Japan	1548

A More Complex Query

Top 5 destination countries in our data – the DataFrame way

```
# in Python
from pyspark.sql.functions import desc

flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .sum("count")\
  .withColumnRenamed("sum(count)", "destination_total")\
  .sort(desc("destination_total"))\
  .limit(5)\
  .show()
```

Transformation Flow of our Complex Query

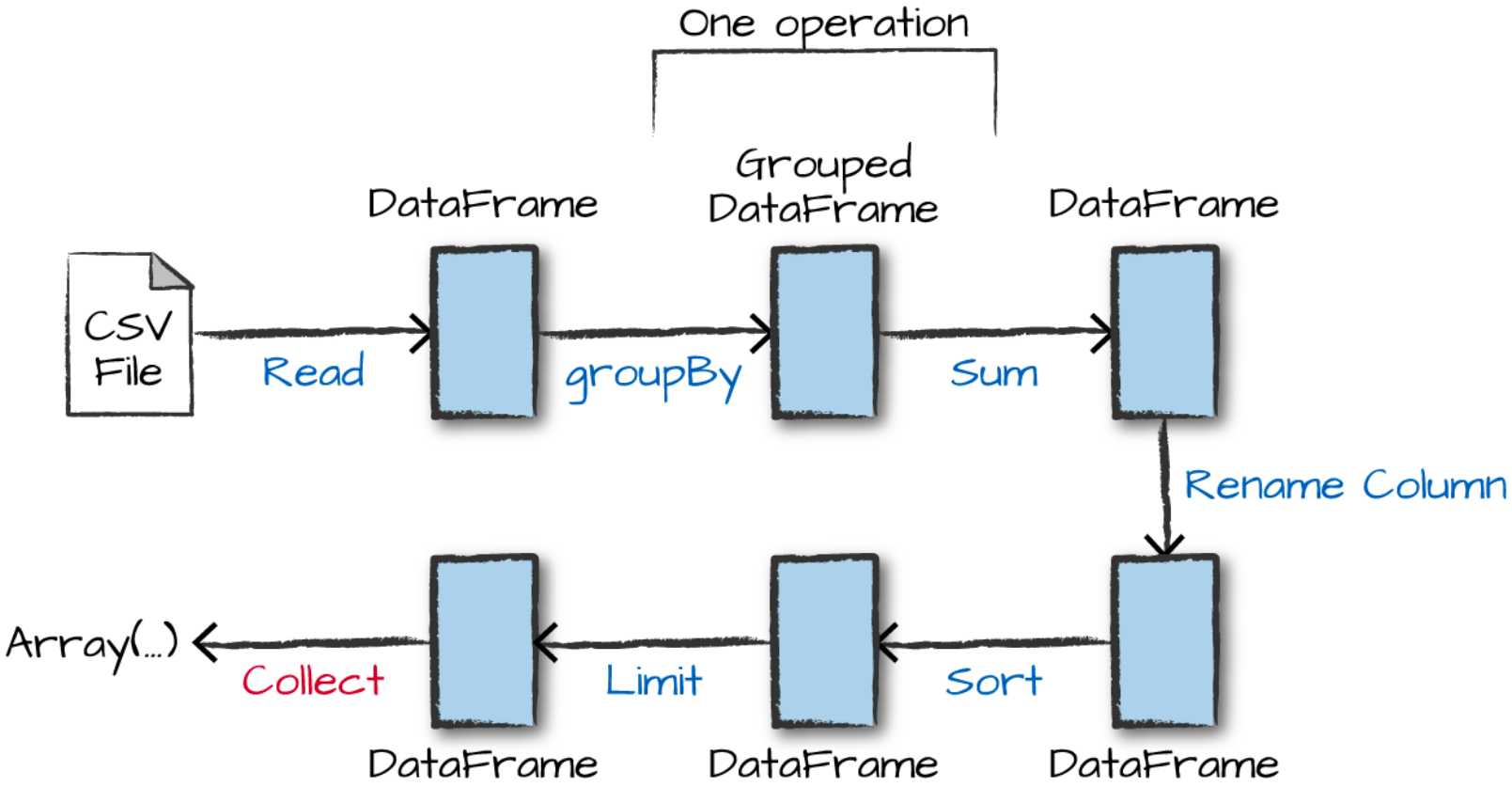

Figure: The entire DataFrame transformation flow in this example

Summary

Key Takeaways:

- Spark operates on distributed data using RDDs, DataFrames, and Datasets.
- Transformations create a new dataset from an existing one.
  - **Narrow Transformations:** Only require data from a single partition (e.g., `map`, `filter`).
  - **Wide Transformations:** Require data shuffling between partitions (e.g., `groupByKey`, `reduceByKey`).
- Actions trigger execution and return results to the driver.
  - Examples: `collect()`, `count()`, `reduce()`, `first()`, `take()`.
- Transformations in Spark are **lazy**, meaning they do not execute until an action is called.
- Spark uses DAG (Directed Acyclic Graph) to optimize job execution.
- Caching and persistence can enhance performance when reusing RDDs.
- Spark supports fault tolerance via lineage information.

Thank You!