International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 08 Issue: 07 | July - 2024 SJIF Rating: 8.448 ISSN: 2582-3930

# A Comparative Analysis of Apache Spark Dataframes over Resilient Distributed Datasets (RDDs)

**Author: Ashima Sahni**

**Synopsis**

Apache Spark is a widely used technology now a days for handling huge datasets in applications due to its flexibility, scalability, robustness, speed and integration with multiple programming languages like Java, Scala, Python. It provides multiple methodologies for implementation like dataframes, RDDs with these programming languages.


This paper provides a deep overview of Apache spark dataframes usage for performance enhancement over Apache Spark Resilient Distributed Datasets (RDDs) and SQL based data processing.

Spark dataframes are widely used in managing and processing large datasets which can be structured, non-structured or semi-structured. This paper describes the approach towards Spark dataframes for performance enhancement for large data processing in place of traditional usage of Spark RDDs with practical examples and use cases. It highlights the key points on why to use dataframes for a better performance achievement for any application where large dataset needs to be processed into a meaningful output.

**Overview of Apache Spark for large datasets processing**

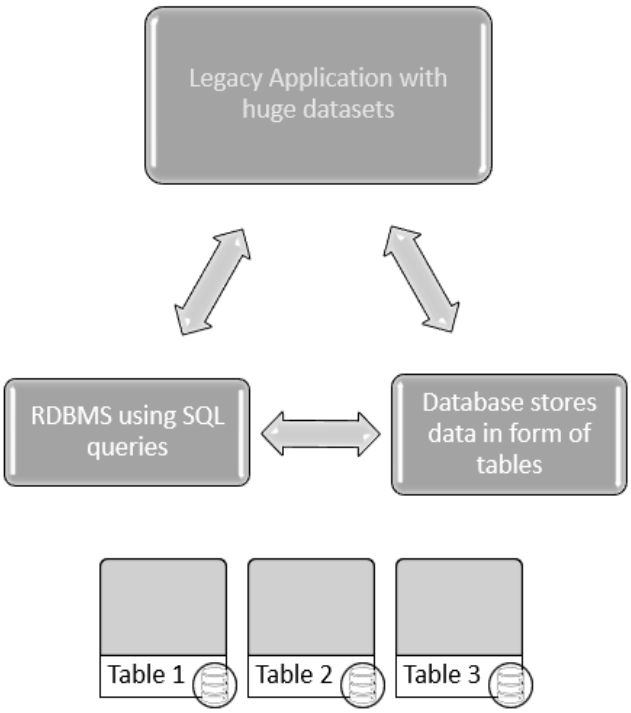
Before invention of Apache Spark technology for applications it was very difficult to handle large datasets effectively to generate desired outputs. The major difficulties for such application were the processing of data which was highly complex and time consuming.

Apache Spark came as a solution for huge amounts of data processing and performance management. With its capabilities like in memory data storage, integration with multiple technologies, robust nature to prevent any data loss, data processing over horizontal clusters, real time data processing with Machine learning and interoperability with technologies like Hadoop and Cassandra have made it highly popular.

Apache Spark provides two approaches of data structure management that are Resilient Distributed Datasets (RDDs) and Dataframes.

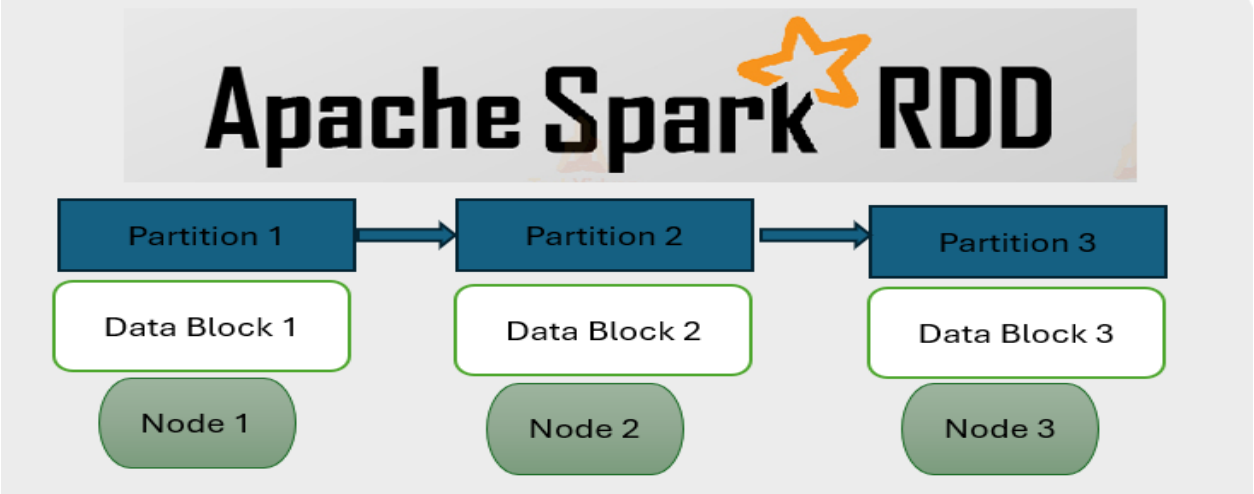
**Overview of Traditional approach of SQL data-based processing**

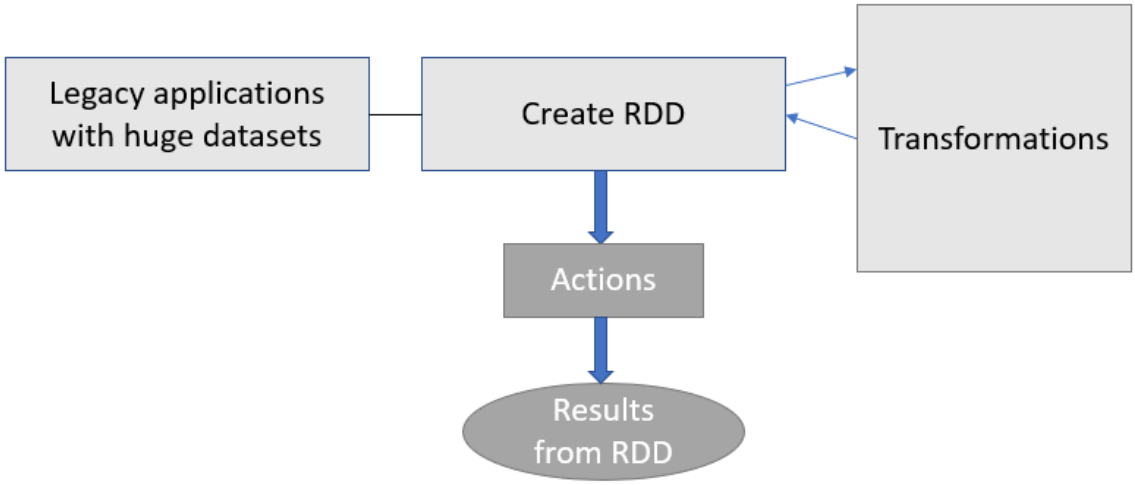
Earlier for managing huge datasets the concept of Relational Database Management System was widely used. Huge datasets were stored in table using relational schema. It had limitations related to scalability, performance and efficiency. Furthermore, data had to be stored at DB level which caused higher maintenance of the applications involving large datasets. Handling of unstructured data was a tedious task with this approach.




**Overview of Traditional approach of Apache Spark RDDs data-based processing**

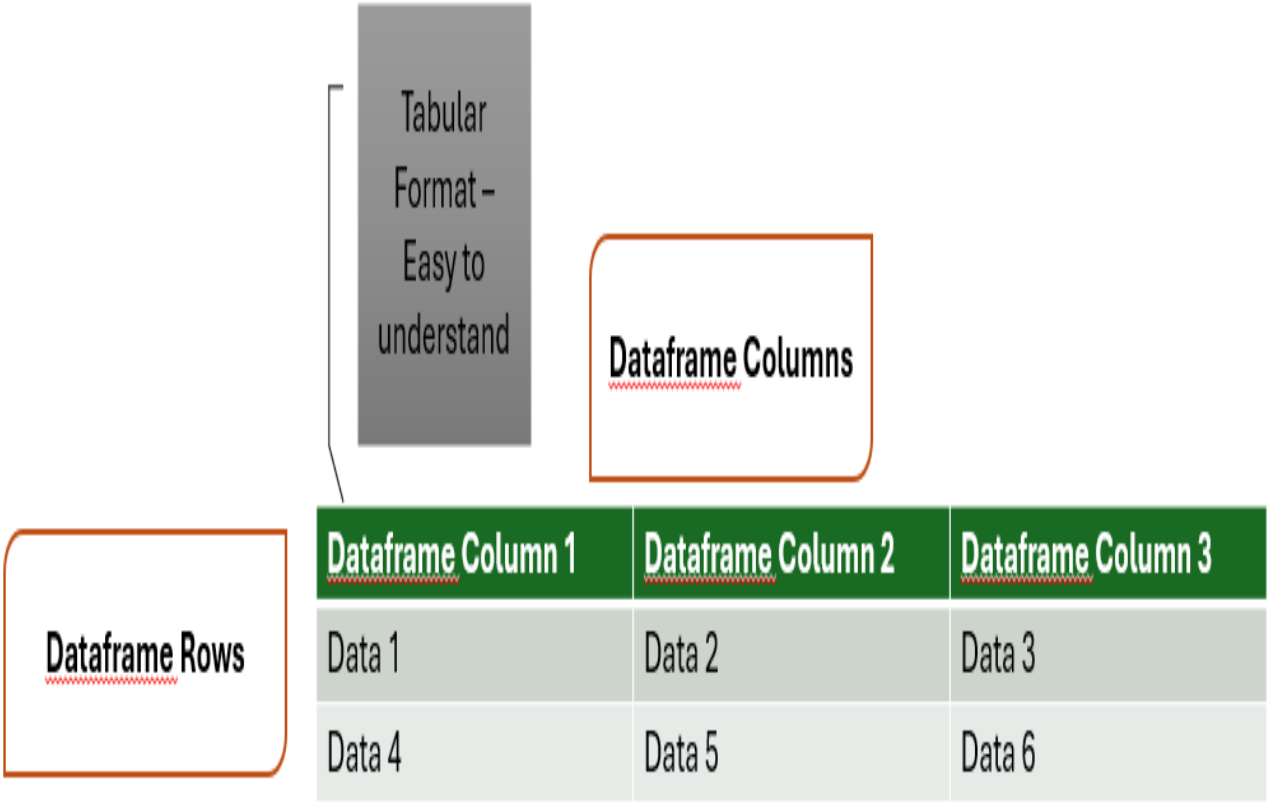
When Apache Spark came into being it was built on RDD Approach of data handling that involves usage of low-level APIs and programming frameworks for managing the processing of data. It came with features managed the data partitioning while allowing parallel processing, in memory storage, RDD transformations and recovery of data in case of any error or failure. However, there were many limitations related to this approach as lot of coding was required and the concepts related to transformations for getting control of the desired output involved a deeper understanding of RDD framework all together.



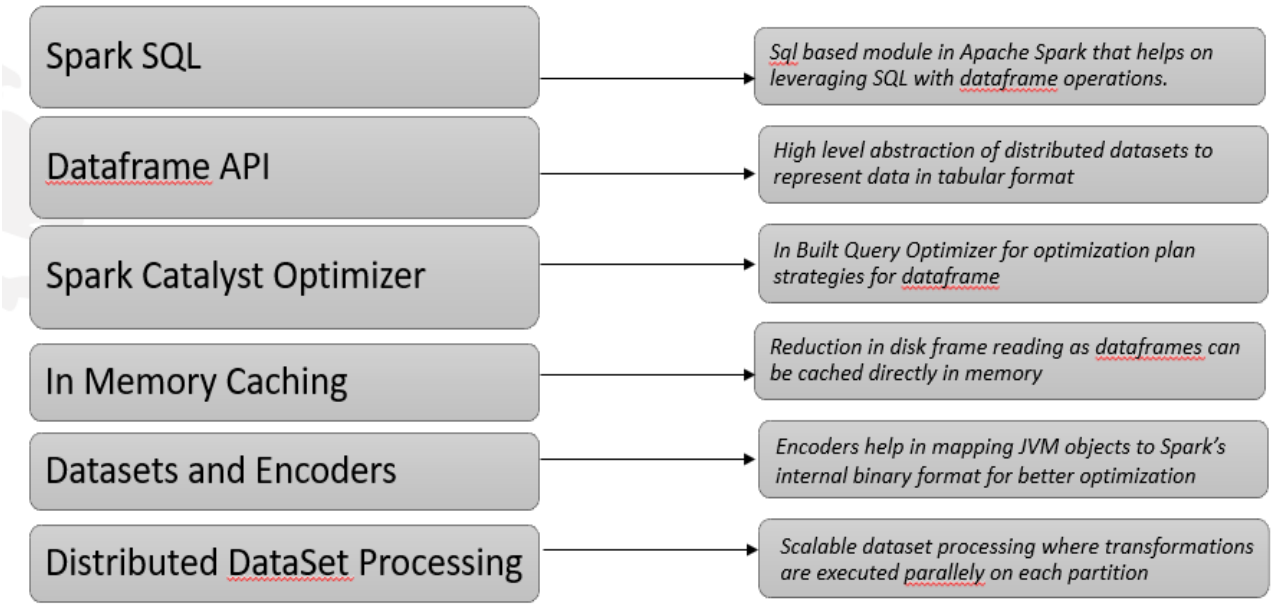

**Overview and Architecture of Spark DataFrames**

Due to above limitations related to Apache Spark RDD approach dataframes were introduced in Apache Spark. Spark dataframes are just the normal tabular representation of data which can provide a better and easier data processing unlike Apache Spark RDDs. They work on high level APIs which provide seamless integration with various technologies where the developers can majorly focus on the representation of data into the desired output without worrying about the data partitioning and memory management techniques. It is built under a more logical and optimized execution under Apache Spark that facilitates the usage of Spark dataframes for performance enhancement for heavy applications.


Tabular Format – Easy to understand
Dataframe Columns
Dataframe Rows
Dataframe Column 1
Dataframe Column 2
Dataframe Column 3
Data 1
Data 2
Data 3
Data 4
Data 5
Data 6


APACHE
Spark
Apache Spark
Dataframes

The **architecture** of spark dataframe involves following components:




**Comparison with traditional SQL-based data processing approach**

This portion describes the basic difference between the two approaches is the data processing techniques used by both methodologies. Spark Dataframes fits best when it comes to processing of huge datasets as compared to the traditional SQL based approach. Here are few basic differences:

| | **Spark Dataframes** | **SQL data processing** |
|---|---|---|
| **Data Processing Architecture** | Spark Dataframes represent in memory storage of datasets in a tabular format. | Data is stored using RDBMS (Relational Database Management System) in tables at a central database location |
| **Basic Syntax** | Spark Dataframes represent usage of high-level functional APIs and Spark SQL where transformations and actions can be applied easily on any kind of structured, unstructured and semi-structured data. | Traditional SQL data processing involves writing complex SQL queries on database tables which in turn makes it difficult for development as complex queries need to be generated on huge sets of data distributed in different set of tables. |
| **Performance Optimization** | Spark dataframes are already built on a cost-effective Spark catalyst optimizer providing in memory storage making it faster and reliable. | Optimization of queries in SQL needs to be checked mostly using database's engine query planner. This involves more focus on |

| | | |
|---|---|---|
| | | the query optimization planning which is very time consuming. |
| **Data Ingestion Flexibility** | Spark Dataframes can take any structured or non-structured format as an input like Parquets, CSVs, HDFS (Hadoop Distributed File System) and process them effectively by converting them into understandable tabular format. | SQL data-based processing can only work on structured data or on semi-structured data where normalizations on tables are performed using Relational Database Management System |

**Comparison with traditional Apache Spark RDD processing approach:**

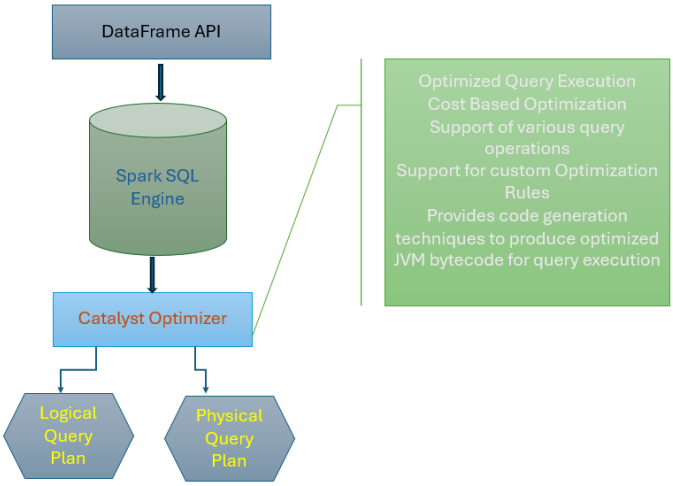
| | **Spark Dataframes** | **Spark RDDs** |
|---|---|---|
| **Data Structures** | Dataframes represent data in a tabular format which is easily understandable and well-structured with proper schema definition making them more suitable to use. | RDDs are more of distributed object collection providing low level APIs which are difficult and complex to read |
| **Ease of Use** | Dataframes provide a high-level API where less coding is involved and functional style of implementation on the tabular dataframes are easier to use and saves a lot of time. It gives privilege of writing SQL queries on dataframes which are easier for development | RDDs provide a low-level API base with a more boilerplate code to be written which is time consuming |
| **Partitioning and Memory Management** | Dataframes provide an inbuilt partitioning and effective use of memory optimization | Memory Management and Data Partitioning in RDDS have to be managed separately. |
| **Performance and Optimization** | Dataframes are more optimized and tend to give better performance as they are built on top of Spark Catalyst Optimizer | While working with RDDS developers must focus on the memory management and optimization manually as they lack in built optimization base. |

| | | |
|---|---|---|
| **Robustness** | Dataframes are less error prone since schema is always imposed on dataframes<br>so it becomes easy to work around data | RDDs are more error prone since<br>there is no strict schema on any random type of data format. |

**Performance and Memory Optimization Techniques used by Spark Dataframes**:

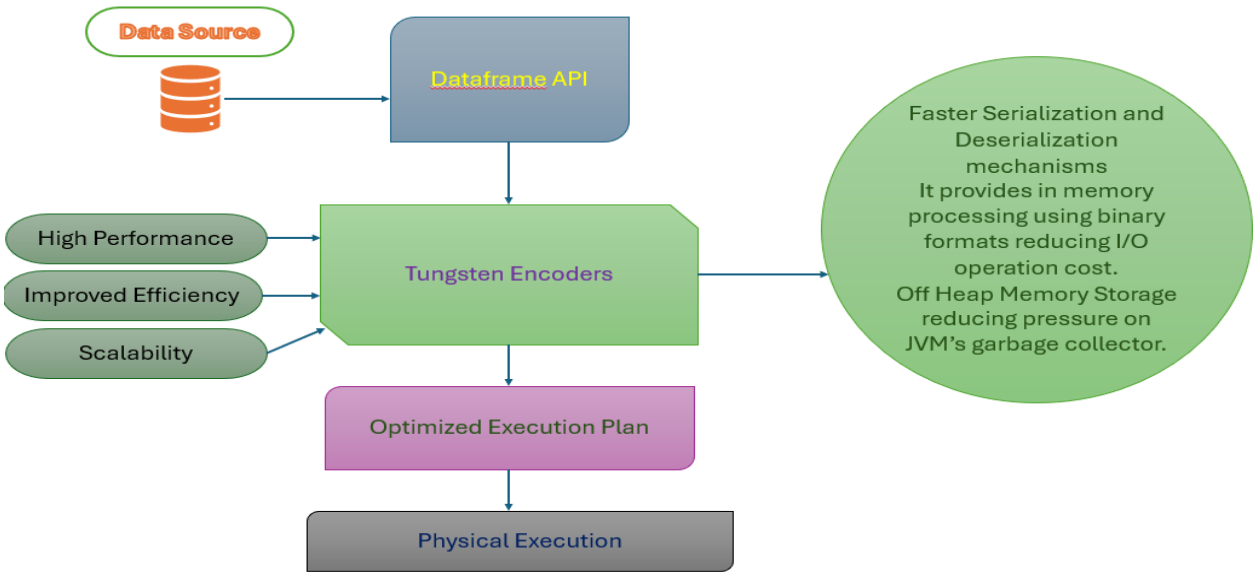
1.**Use of catalyst optimizer for Performance:**

Apache Spark RDDs provides full control on transformations and actions but they do not provide any optimization techniques for better performance. Manual intervention for understanding the best optimization query becomes tedious when working with RDDs. While on the other hand Apache Spark Dataframes are built on top of catalyst optimizer which automatically manages query optimization giving high performance benefits.




2.**Use of Tungsten Encoders for memory efficiency and performance:**

While Apache Spark RDDs tend to rely on Java serialization for memory management Dataframes are built on top of Tungsten Engine which uses Tungsten encoders for an efficient serialization mechanism making it highly beneficial for performance and resource consumption.




**Code complexity Apache Spark RDD vs Dataframes:**

**Using RDD code logic:**

It uses Spark's functional programming model with custom function definitions which involves multiple transformations and actions which is hard to understand and further requires manual optimization.

**Language Support**-Java, Python and Scala

```
JavaRDD<String> input = sc.textFile("path/to/input.txt");

JavaRDD<String> words = input.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public Iterator<String> call(String s) {
        return Arrays.asList(s.split(" ")).iterator();
    }
});

JavaPairRDD<String, Integer> wordCounts = words.mapToPair(new PairFunction<String, String, Integer>() {
    @Override
    public Tuple2<String, Integer> call(String s) {
        return new Tuple2<>(s, 1);
    }
}).reduceByKey(new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer a, Integer b) {
        return a + b;
    }
});

wordCounts.saveAsTextFile("path/to/output");
```

**Using Dataframes:**

Apache Spark Dataframes provides more concise and readable implementations. It is based on SQL programming syntax which easy to understand offering an added benefit of writing code in lesser lines with single query and no manual optimization needs to be looked at while working with dataframes.

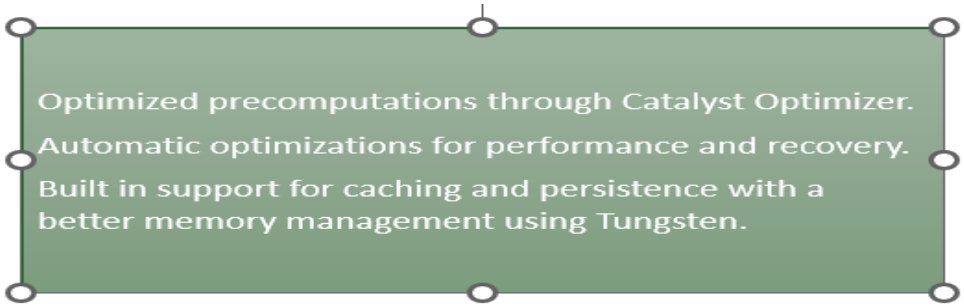
**Language Support**-Available in Java, Scala, Python, R

```
Dataset<String> input = spark.read().textFile("path/to/input.txt");

Dataset<Row> words = input
    .select(functions.explode(functions.split(input.col("value"), " ")).alias("word"));

Dataset<Row> wordCounts = words.groupBy("word").count();

wordCounts.write().format("text").save("path/to/output");

sc.close();
```

**Fault tolerance comparison:**

RDDs use Lineage Graph with deterministic operations with Lazy evaluation and checkpointing for fault tolerance and recovery whereas Dataframes use Logical plan and Catalyst Optimizer with Lazy evaluation and Caching/Persistence mechanisms.

**Advantages of Dataframes for fault tolerance mechanism:**




**Case Study Apache Spark RDD vs Dataframes:**

This case study represents the performance comparison analysis of Apache Spark RDDs and Apache Spark Dataframes.

There is a csv which represents the data of a **sales application** in a particular format.

| OrderID | ProductID | Quantity | UnitPrice |
|---|---|---|---|
| 1 | P1 | 10 | 100 |
| 2 | P2 | 15 | 150 |
| 3 | P1 | 8 | 95 |
| 4 | P3 | 12 | 150 |
| 5 | P2 | 20 | 110 |
| 6 | P1 | 5 | 120 |
| 7 | P3 | 10 | 180 |
| 8 | P2 | 18 | 105 |
| 9 | P1 | 15 | 110 |
| 10 | P3 | 22 | 160 |

**Use Case 1:**

We want to retrieve the total sales amount representing the Unit Price column for each product. For this use case we are performing following operations using spark:

**Aggregation** on Product Id column and **sum** to calculate the total amount of sales/Unit price per Product.

**RDD Logic and performance time:**

```
// Read CSV into RDD
JavaRDD<String> salesRDD = jsc.textFile("C://CSV//sales_data.csv");

// Skip header
String header = salesRDD.first();
JavaRDD<String> dataRDD = salesRDD.filter(row -> !row.equals(header));

// Map each line to (ProductID, TotalSales)
long startTimeRDD = System.currentTimeMillis();

JavaPairRDD<String, Double> productSalesRDD = dataRDD.mapToPair(line -> {
    String[] parts = line.split(",");
    String productID = parts[2];
    int quantity = Integer.parseInt(parts[3]);
    double unitPrice = Double.parseDouble(parts[4]);
    double totalSales = quantity * unitPrice;
    return new Tuple2<>(productID, totalSales);
});

// Reduce by key to calculate total sales amount per product
JavaPairRDD<String, Double> totalSalesRDD = productSalesRDD.reduceByKey((a, b) -> a + b);

// Collect results
List<Tuple2<String, Double>> resultsRDD = totalSalesRDD.collect();

// Print results
for (Tuple2<String, Double> result : resultsRDD) {
    System.out.println(result._1() + ": " + result._2());
}

long endTimeRDD = System.currentTimeMillis();
long elapsedTimeRDD = endTimeRDD - startTimeRDD;
System.out.println("RDD API execution time: " + elapsedTimeRDD + " milliseconds");
```

**Output:**

```
Total Sales Amount per Product (RDD):
P1: 1500.0
P2: 2200.0
P3: 1800.0
RDD API execution time: 480 milliseconds
```

**Dataframe Logic and performance time:**

Dataframe logic is much faster in terms of time comparison and the logic has simple query like operations which are easier to implement and understand. Also, it provides better performance in terms of time due to usage of catalyst optimizer internally.

```
// Read CSV into DataFrame
Dataset<Row> salesDF = spark.read().option("header", "true").csv("C://CSV//sales_data.csv");

// Convert Quantity and UnitPrice to numeric types
salesDF = salesDF.withColumn("Quantity", salesDF.col("Quantity").cast(DataTypes.IntegerType))
                 .withColumn("UnitPrice", salesDF.col("UnitPrice").cast(DataTypes.DoubleType));

// Calculate total sales amount per product using DataFrame API
long startTimeDF = System.currentTimeMillis();

Dataset<Row> productSalesDF = salesDF.withColumn("TotalSales", salesDF.col("Quantity").multiply(salesDF.col("UnitPrice")))
                                     .groupBy("ProductID")
                                     .sum("TotalSales");

productSalesDF.show();

long endTimeDF = System.currentTimeMillis();
long elapsedTimeDF = endTimeDF - startTimeDF;
System.out.println("DataFrame API execution time: " + elapsedTimeDF + " milliseconds");
```

**Output:**

```
+---------+-----------------+
| ProductID|   sum(TotalSales)|
+---------+-----------------+
|    P1    |      1500.0      |
|    P2    |      2200.0      |
|    P3    |      1800.0      |
+---------+-----------------+
DataFrame API execution time: 320 milliseconds
```

**Overall Improvement Achieved with these operations is about 33% higher in case of dataframes.**

| DataFrame Execution Time | RDD Execution Time | API % Performance Improvement |
|---|---|---|
| 320 | 480 | 33.33% |

**Use Case 2:**

**Filtering**: We filter the sales records to include only those where the quantity sold is greater than 10 units.
**Aggregation**: After filtering, we calculate the total sales amount for each product by multiplying the quantity sold by the unit price and summing these values per product.

**Dataframe Logic and performance time**:

```
// Read CSV into DataFrame
Dataset<Row> salesDF = spark.read().option("header", "true").csv("C://Sales//Data.csv");

// Convert Quantity and UnitPrice to numeric types
salesDF = salesDF.withColumn("Quantity", salesDF.col("Quantity").cast(DataTypes.IntegerType))
        .withColumn("UnitPrice", salesDF.col("UnitPrice").cast(DataTypes.DoubleType));

// Filter orders where Quantity > 10
Dataset<Row> filteredSalesDF = salesDF.filter(salesDF.col("Quantity").gt(10));

// Calculate total sales amount per product using DataFrame API
long startTimeDF = System.currentTimeMillis();

Dataset<Row> productSalesDF = filteredSalesDF
        .withColumn("TotalSales", filteredSalesDF.col("Quantity").multiply(filteredSalesDF.col("UnitPrice")))
        .groupBy("ProductID").sum("TotalSales");

productSalesDF.show();

long endTimeDF = System.currentTimeMillis();
long elapsedTimeDF = endTimeDF - startTimeDF;
System.out.println("DataFrame API execution time: " + elapsedTimeDF + " milliseconds");
```

**Output:**

```
+---------+-----------------+
| ProductID|   sum(TotalSales)|
+---------+-----------------+
|    P2    |      4400.0      |
|    P3    |      7200.0      |
+---------+-----------------+

DataFrame API execution time: 290 milliseconds
```

**RDD Logic and performance time:**

```
JavaRDD<String> salesRDD = jsc.textFile("C://Sales//Data.csv");


String header = salesRDD.first();
JavaRDD<String> dataRDD = salesRDD.filter(row -> !row.equals(header));

// Filter rows where Quantity > 10 and map to (ProductID, TotalSales)
long startTimeRDD = System.currentTimeMillis();

JavaPairRDD<String, Double> productSalesRDD = dataRDD.filter(line -> {
    String[] parts = line.split(",");
    int quantity = Integer.parseInt(parts[2]);
    return quantity > 10;
})
.mapToPair(line -> {
    String[] parts = line.split(",");
    String productID = parts[1];
    int quantity = Integer.parseInt(parts[2]);
    double unitPrice = Double.parseDouble(parts[3]);
    double totalSales = quantity * unitPrice;
    return new Tuple2<>(productID, totalSales);
})
.reduceByKey((a, b) -> a + b);

// Collect results
List<Tuple2<String, Double>> resultsRDD = productSalesRDD.collect();

// Print results
for (Tuple2<String, Double> result : resultsRDD) {
    System.out.println(result._1() + ": " + result._2());
}

long endTimeRDD = System.currentTimeMillis();
long elapsedTimeRDD = endTimeRDD - startTimeRDD;
System.out.println("RDD API execution time: " + elapsedTimeRDD + " milliseconds");
```

**Output:**

```
Total Sales Amount per Product (RDD):
P2: 4400.0
P3: 7200.0
RDD API execution time: 470 milliseconds
```

**Overall Improvement Achieved with these operations is about 38.3% higher in case of dataframes.**

| DataFrame Execution Time | RDD Execution Time | API % Performance Improvement |
|---|---|---|
| 290 | 470 | 38.30% |

**Conclusion:**

With these high-performance benefits with Spark Dataframes applications which are currently using RDDs can be easily migrated to Dataframes which additionally have an optimized and clean code implementation and reliability. For new applications, we have a clear choice to use Apache Spark Dataframes over RDDs.

**Links/References:**

**https://wisewithdata.com/2020/05/rdds-vs-dataframes-vs-datasets-the-three-data-structures-of-spark/**

**https://www.analyticsvidhya.com/blog/2020/11/what-is-the-difference-between-rdds-dataframes-and-datasets/**

**https://medium.com/@ashwin_kumar_/spark-rdd-vs-dataframe-vs-dataset-c90f7da18e56**

**https://spark.apache.org/documentation.html**