Aim : Write example for following Spark RDD Actions:
a. aggregate b. treeAggregate c. fold
d. reduce e. collect

Solution :

1. aggregate()

```
val rdd = sc.parallelize(List(1, 2, 3, 4, 5))

// Zero value: 0, seqOp: Add elements, combOp: Combine partitions
val result = rdd.aggregate(0)(_ + _, _ + _)

println(result) // Output: 15
```

---

2. treeAggregate()

```
val rdd = sc.parallelize(List(1, 2, 3, 4, 5), 3)

val result = rdd.treeAggregate(0)(_ + _, _ + _)

println(result) // Output: 15
```

---

3. fold()

```
val rdd = sc.parallelize(List(1, 2, 3, 4, 5))

// Zero value: 0, operation: Add elements
val result = rdd.fold(0)(_ + _)

println(result) // Output: 15
```

4. reduce()

```
val rdd = sc.parallelize(List(1, 2, 3, 4, 5))

// Reduces using a sum operation
val result = rdd.reduce(_ + _)

println(result) // Output: 15
```

5. collect()

```
val rdd = sc.parallelize(List(1, 2, 3, 4, 5))

// Collect all elements from the RDD
val result = rdd.collect()

println(result.mkString(", ")) // Output: 1, 2, 3, 4, 5
```

Write example for following Spark RDD Actions: a. count b. countApproxDistinct c. first d. Top e. Min f.max

a. count

Purpose: Counts the total number of elements in the RDD.

```
val rdd = sc.parallelize(List(10, 20, 30, 40,
50))

val totalCount = rdd.count()

println(s"Total count: $totalCount")  //
Output: 5
```

---

b. countApproxDistinct

Purpose: Counts the approximate number of distinct elements in the RDD — more efficient for very large datasets.

```
val rdd = sc.parallelize(List(1, 2, 2, 3, 3,
3, 4, 4, 5))

val approxDistinctCount =
rdd.countApproxDistinct()

println(s"Approximate distinct count:
$approxDistinctCount")
 // Output: 5
```

`countApproxDistinct` uses a probabilistic algorithm — for huge data it's much faster than exact `distinct().count()`.

---

c. `first`

```
val rdd = sc.parallelize(List(100, 200, 300))

val firstElement = rdd.first()

println(s"First element: $firstElement")  // Output: 100
```

---

d. `top`

Purpose: Returns the top N elements, sorted in descending order.

```
val rdd = sc.parallelize(List(10, 70, 20, 90, 40))

val top3 = rdd.top(3)  // Sorted descending
```

```
println(s"Top 3 elements: ${top3.mkString(", ")}")
// Output: Top 3 elements: 90, 70, 40
```

---

e. min

Purpose: Returns the minimum element in the RDD.

```
val rdd = sc.parallelize(List(25, 10, 30, 5, 15))

val minValue = rdd.min()

println(s"Minimum value: $minValue")//Minimum value :5
```

Aim : Write Spark Pair RDD Functions.

A Pair RDD is an RDD where each element is a key-value pair: `(K, V)`.

1. reduceByKey

Combine values by key with a function (e.g., sum):

```
val data = sc.parallelize(Seq(("a", 1), ("b",
2), ("a", 3)))
val result = data.reduceByKey(_ +
_).collect()
// Output: Array(("a", 4), ("b", 2))
```

mapValues

Transform only the value part, keeping the key unchanged.

```
val data = sc.parallelize(Seq(("x", 2), ("y",
3)))
val squared = data.mapValues(v => v *
v).collect()
// Output: Array(("x", 4), ("y", 9))
```

keys and values

Extract only keys or only values.

```
val data = sc.parallelize(Seq(("a", 1), ("b",
2)))
val keys = data.keys.collect() // Output:
Array("a", "b")
```

```
val values = data.values.collect() // Output: Array(1, 2)
```

sortByKey

Sort by key.

```
val data = sc.parallelize(Seq(("c", 3), ("a", 1), ("b", 2)))
val sorted = data.sortByKey().collect()
// Output: Array(("a", 1), ("b", 2), ("c", 3))
```

join

Join two Pair RDDs on key.

```
val rdd1 = sc.parallelize(Seq(("a", 1), ("b", 2)))
val rdd2 = sc.parallelize(Seq(("a", "x"), ("b", "y")))
val joined = rdd1.join(rdd2).collect()
// Output: Array((a, (1, "x")), ("b", (2, "y")))
```

lookup

Return all values for a given key.

```
val data = sc.parallelize(Seq(("a", 1), ("b", 2), ("a", 3)))
val result = data.lookup("a")
// Output: Seq(1, 3)
```

05/08/2025

Aim :Create two dataframes one for employee and other for dept. Perform

a) Left anti join
b) Self join
c) Left semi join

```
val employee = Seq(
  (1, "Alice", 10),
  (2, "Bob", 20),
  (3, "Charlie", 30),
  (4, "David", 40),
  (5, "Eva", 50)
).toDF("emp_id", "emp_name", "dept_id")

val dept = Seq(
  (10, "HR"),
  (20, "Finance"),
  (30, "IT")
).toDF("dept_id", "dept_name")
```

a) Left anti join

Left Anti join: Returns only the rows from the left DataFrame (e.g., `employee`) that do NOT have a match in the right DataFrame (e.g., `dept`).

"Give me employees whose department is not listed in the department table

```
val antiJoin = employee.join(dept, Seq("dept_id"),
"left_anti")
antiJoin.show()
```

b) Self join

Definition:
A join of a DataFrame with itself. It is used to compare rows within the same table, often based on a common column.

"Find pairs of employees who work in the same department."

```
val e1 = employee.as("e1")
val e2 = employee.as("e2")

val selfJoin = e1.join(e2, $"e1.dept_id" === $"e2.dept_id")
  .select($"e1.emp_name".as("emp1"),
$"e2.emp_name".as("emp2"), $"e1.dept_id")
selfJoin.show()
```

c) Left Semi Join

Returns **only the rows from the left DataFrame that have a match** in the right DataFrame — but **only columns from the left table** are returned.

"Give me employees who belong to a valid department (as per the department table)."

```
val semiJoin = employee.join(dept,
Seq("dept_id"), "left_semi")
semiJoin.show()
```

a) Create two case classes – Student and Address
b) Create schema from these case classes

```
case class Address(city: String, state: String, pincode: String)

case class Student(id: Int, name: String, age: Int, address: Address)

val students = Seq(
  Student(1, "Alice", 20, ,Address("Mumbai", "MH", "400001")),
  Student(2, "Bob", 21, “MaharasAddress("Pune", "MH", "411001")),
  Student(3, "Charlie", 22, “Karnataka”,Address("Delhi", "DL", "110001"))
)

val studentDF = spark.createDataFrame(students)
studentDF.printSchema()
```

3/9/2025

a)Create a data frame with today’s date and timestamp
b) Display the hours, minutes and seconds from the timestamp

```
 var df = Seq(1).toDF("id")
df = df.withColumn("today", current_date())
df = df.withColumn("now", current_timestamp())
df.show()
```

b)Display the hours, minutes and seconds from the timestamp

```
var df = Seq(1).toDF("id")
df = df.withColumn("now", current_timestamp())
df = df.withColumn("hour", hour(col("now")))
df = df.withColumn("minute", minute(col("now")))
df = df.withColumn("second", second(col("now")))
df.show(false)
```

a)For the following employee data showing name, dept and salary, perform the
given operations:

Data: ("James", "Sales", 3000),
("Michael", "Sales", 4600),
("Robert", "Sales", 4100),
("Maria", "Finance", 3000),
("James", "Sales", 3000),
("Scott", "Finance", 3300),
("Jen", "Finance", 3900),
("Jeff", "Marketing", 3000),

```
("Kumar", "Marketing", 2000),
("Saif", "Sales", 4100),
(Jason", "Sales", 9000),
("Alice", "Finance", 3700),
("Jenniffer", "Finance", 8900),
("Jenson", "Marketing", 9000)
```

a) Create a data frame for the above data
b) Display average salary
c) Display number of unique departments
d) Display number of employees with unique salary

a)

```
val data = Seq(
  ("James", "Sales", 3000),
  ("Michael", "Sales", 4600),
  ("Robert", "Sales", 4100),
  ("Maria", "Finance", 3000),
  ("James", "Sales", 3000),
  ("Scott", "Finance", 3300),
  ("Jen", "Finance", 3900),
  ("Jeff", "Marketing", 3000),
  ("Kumar", "Marketing", 2000),
  ("Saif", "Sales", 4100),
  ("Jason", "Sales", 9000),
  ("Alice", "Finance", 3700),
  ("Jenniffer", "Finance", 8900),
  ("Jenson", "Marketing", 9000)
)

val df = data.toDF("name", "dept", "salary")
df.show(false)
```

b) Display average salary

```
df.agg(avg("salary")).show()
```

c) Display number of unique departments

```
df.select("dept").distinct().count()
```

d) Display number of employees with unique salary

```
df.groupBy("salary")
  .count()
  .filter($"count" === 1)
  .agg(sum("count"))
  .show()
```

a)For the following employee data showing name, dept and salary, perform the
given operations:

Data: ("James", "Sales", 3000),
("Michael", "Sales", 4600),
("Robert", "Sales", 4100),
("Maria", "Finance", 3000),
("James", "Sales", 3000),
("Scott", "Finance", 3300),
("Jen", "Finance", 3900),
("Jeff", "Marketing", 3000),
("Kumar", "Marketing", 2000),
("Saif", "Sales", 4100),
(Jason", "Sales", 9000),
("Alice", "Finance", 3700),
("Jenniffer", "Finance", 8900),
("Jenson", "Marketing", 9000)
a) Create a data frame for the above data
b) Find the highest salary value
c) Find the lowest salary value
d) Find the standard deviation for the salary

a)

```
val data = Seq(
  ("James", "Sales", 3000),
  ("Michael", "Sales", 4600),
  ("Robert", "Sales", 4100),
  ("Maria", "Finance", 3000),
  ("James", "Sales", 3000),
  ("Scott", "Finance", 3300),
  ("Jen", "Finance", 3900),
  ("Jeff", "Marketing", 3000),
  ("Kumar", "Marketing", 2000),
  ("Saif", "Sales", 4100),
  ("Jason", "Sales", 9000),
  ("Alice", "Finance", 3700),
  ("Jenniffer", "Finance", 8900),
  ("Jenson", "Marketing", 9000)
)

val df = data.toDF("name", "dept", "salary")
df.show(false)
```

b) Find the highest salary

```
df.agg(max("salary")).show()
```

c) Find the lowest salary

```
df.agg(min("salary")).show()
```

d) Find the standard deviation of salary

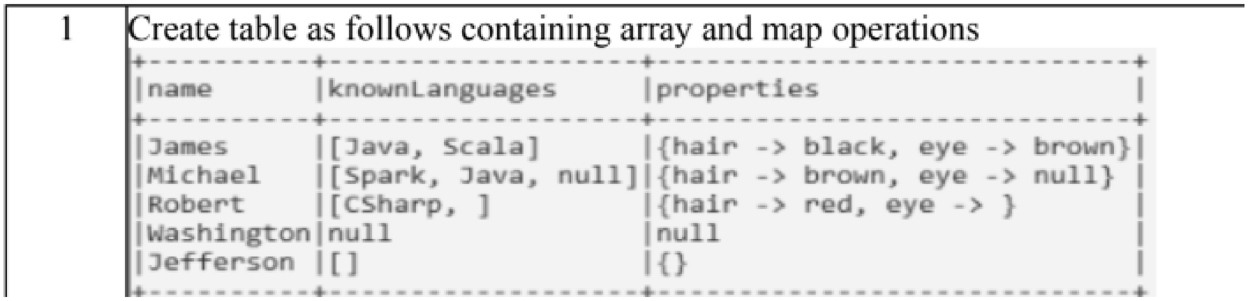
```
df.agg(stddev("salary")).show()
```

b) Create a data frame with data that follows the below given schema
emp_id, dept, properties (a structure containing salary and location)
Return the map keys from spark SQL for this data frame

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import spark.implicits._

// Schema
val schema = StructType(Seq(
  StructField("emp_id", IntegerType, false),
  StructField("dept", StringType, true),
  StructField("properties", StructType(Seq(
    StructField("salary", IntegerType, true),
    StructField("location", StringType, true)
  )))
))

//Data
val data = Seq(
  Row(1, "Sales", Row(3000, "Mumbai")),
  Row(2, "Finance", Row(4000, "Delhi")),
  Row(3, "Marketing", Row(5000, "Pune"))
)

// Create DataFrame df
val df = spark.createDataFrame(
  spark.sparkContext.parallelize(data),
  schema
)
```

```
// Convert struct -> map (df2)
val df2 = df.withColumn("properties_map", map(lit("salary"),
col("properties.salary"),lit("location"),
col("properties.location")))
df.show(false)
```

a)



1 Create table as follows containing array and map operations

```
+----------+-------------------+-----------------------------+
|name      |knownLanguages     |properties                   |
+----------+-------------------+-----------------------------+
|James     |[Java, Scala]      |{hair -> black, eye -> brown}|
|Michael   |[Spark, Java, null]|{hair -> brown, eye -> null} |
|Robert    |[CSharp, ]         |{hair -> red, eye -> }       |
|Washington|null               |null                         |
|Jefferson |[]                 |{}                           |
+----------+-------------------+-----------------------------+
```

```
// 1. Sample data
val data = Seq(
  ("James",    Seq("Java", "Scala"),   Map("hair" -> "black",
"eye" -> "brown")),
  ("Michael",  Seq("Spark", "Java", null), Map("hair" -> "brown",
"eye" -> null)),
  ("Robert",   Seq("CSharp", ""),     Map("hair" -> "red", "eye"
-> "")),
  ("Washington", null,               null),
  ("Jefferson", Seq.empty[String],    Map.empty[String, String])
)

// 2. Create DataFrame
val df = spark.createDataFrame(data).toDF("name",
"knownLanguages", "properties")
```

```
// 3. Show the table
df.show(false)
```

b)
Find current timestamp and hour, Minute, second separately for today's date

```
val df = spark.range(1).select(
  current_timestamp().alias("current_ts"),
  hour(current_timestamp()).alias("hour"),
  minute(current_timestamp()).alias("minute"),
  second(current_timestamp()).alias("second")
)
df.show(false)
```

10/09/2025

1. a) Create a data frame containing today’s date, date 2022-01-31, date 2021-03-22, date 2024-01-31, date 2023-11-11.

b) Store the date in the format MM-DD-YYYY.

c) Display the dates in the format DD/MM/YYYY

d) Find the number of months between each of the dates and today’s date

Solution :

a)

```
import java.time.LocalDate

// Today's date
val today = LocalDate.now().toString  // e.g. "2025-09-09"

// Create a Seq of dates
val dates = Seq(
  today,
  "2022-01-31",
  "2021-03-22",
  "2024-01-31",
  "2023-11-11"
)

// Convert to DataFrame
val df = dates.toDF("date_str")
df.show()
```

b)

```
// Convert to DateType first
val df1 = df.withColumn("date", to_date($"date_str", "yyyy-MM-dd"))

// Overwrite with MM-dd-yyyy as String
val df2 = df1.withColumn("date", date_format($"date", "MM-dd-yyyy"))

df2.show(false)
```

c)

```
// Convert to DateType
val df1 = df.withColumn("date", to_date($"date_str", "yyyy-MM-dd"))

// Display as DD/MM/YYYY
val df2 = df1.withColumn("date_DDMMYYYY", date_format($"date", "dd/MM/yyyy"))

df2.show(false)
```

d)

```
// Step 1: Convert to DateType

 val df1 = df.withColumn("date", to_date($"date_str",
"yyyy-MM-dd"))

 // Step 2: Find months difference between today's date and
each dateval df2 = df1.withColumn("months_diff",
months_between(current_date(), $"date"))

df2.show(false)
```

2) Refer to the employee.json file. Perform the following operations:

a) Print the names of employees above 25 years of age.

b) Print the number of employees of different ages.

<u>employee.json(file)</u>

```
[

  {"name":"Alice", "age":25, "dept":"IT"},

  {"name":"Bob", "age":30, "dept":"HR"},
  {"name":"Charlie", "age":28, "dept":"Finance"},
  {"name":"David", "age":24, "dept":"IT"}
]
```

```
val df = spark.read.option("multiline", "true").json("C:/Users/Lenovo/Documents/employee.json")
df.show(false)
```

a)

```
df.filter($"age" > 25).select("name").show(false)
```

b)

```
df.groupBy("age").count().show(false)
```

3. a)Get new dates by adding 4 days, and subtracting 7 days in below dates "2020-01-02","2023-01-15","2025-01-30"

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._
 // Create Spark session
```

```
val spark = SparkSession.builder()
  .appName("DateExample")
  .master("local[*]")
  .getOrCreate()

// Step 1: Create DataFrame
val df = spark.createDataFrame(Seq(
  ("2020-01-02"),
  ("2023-01-15"),
  ("2025-01-30")
)).toDF("date_str")

// Step 2: Convert to DateType
val df2 = df.withColumn("date", to_date(df("date_str"),
"yyyy-MM-dd"))

// Step 3: Add 4 days
val df3 = df2.withColumn("date_plus_4", date_add(df2("date"),
4))
```

```
// Step 4: Subtract 7 days

val df_final = df3.withColumn("date_minus_7",
date_sub(df2("date"), 7))


// Step 5: Show all columns

df_final.show(false)
```

b)

Use the Operation Read CSV file on RDD with Scala operation

```
// Read CSV file as RDD[String]

val rdd =
spark.sparkContext.textFile("path/to/your/employee.csv")


// Print first 5 lines

rdd.take(5).foreach(println)
```