Spark Programs

1)

```
//fold
val rdd = sc.parallelize(Seq(10, 20, 30, 40))
val result = rdd.map(_ + 100).fold(0)(_ + _)
//aggregate
val rdd = sc.parallelize(Seq(10, 20, 30, 40))
val result = rdd.aggregate(0)(
  (acc, value) => acc + (value + 100),  // seqOp: update & sum within partition
  (acc1, acc2) => acc1 + acc2           // combOp: combine results across partitions
)
```

2)

```
import org.apache.spark.{SparkConf, SparkContext}

object pro {
def main(args: Array[String]): Unit = {

// Set up the Spark configuration and context
val conf = new SparkConf().setAppName("WordCountApp").setMaster("local[*]")
val sc = new SparkContext(conf)

// Path to input file
val pathToFile = "log.txt"

// Read file and split into words
val wordsRdd = sc.textFile(pathToFile).flatMap(_.split("\\s+"))

// Create initial word count RDD
val wordCountInitRdd = wordsRdd.map(word => (word, 1))

// Reduce by key to get total counts
val wordCountRdd = wordCountInitRdd.reduceByKey((v1, v2) => v1 + v2)
```

```
// Filter words that occur more than 4 times
val highFreqWords = wordCountRdd.filter(x => x._2 > 4)

// Save the result
highFreqWords.saveAsTextFile("wordcountsDir")

// Stop the SparkContext
sc.stop()
}
}
```

3)

```
import org.apache.spark.{SparkConf, SparkContext}

object WordCountApp {
  def main(args: Array[String]): Unit = {
    // Step 1: Initialize Spark Context
    val conf = new SparkConf().setAppName("WordCountApp").setMaster("local")
    val sc = new SparkContext(conf)

    // Step 2: Load the text file into an RDD
    val textRDD = sc.textFile("text.txt")
    // Step 3: Split each line into words
    val wordsRDD = textRDD.flatMap(line => line.split("\\s+"))

    // Step 4: Convert each word to a pair (word, 1)
    val pairsRDD = wordsRDD.map(word => (word, 1))

    // Step 5: Reduce by key to count occurrences
    val wordCountsRDD = pairsRDD.reduceByKey(_ + _)

    // Step 6: Collect and print the results to console
    wordCountsRDD.collect().foreach { case (word, count) =>
      println(s"$word: $count")
```

```
    }

    // Step 7: Save the word counts to output directory
    wordCountsRDD.saveAsTextFile("output_directory")

    // Stop the Spark Context
    sc.stop()
  }
```

4)import org.apache.spark.{SparkConf, SparkContext}

```
object AverageMarksApp {
  def main(args: Array[String]): Unit = {
    // Initialize Spark Context
    val conf = new SparkConf().setAppName("AverageMarks").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Sample data: (Student, Subject, Marks)
    val data = Array(
      ("Joe", "Maths", 83), ("Joe", "Physics", 74), ("Joe", "Chemistry", 91), ("Joe", "Biology", 82),
      ("Nik", "Maths", 69), ("Nik", "Physics", 62), ("Nik", "Chemistry", 97), ("Nik", "Biology", 80)
    )

    // Parallelize the data into an RDD
    val rdd = sc.parallelize(data)

    // Extract (Student, Marks) pairs, trimming spaces in student names
    val marksRDD = rdd.map { case (student, subject, marks) => (student.trim, marks) }

    // Use combineByKey to calculate (sum, count) for each student
    val combined = marksRDD.combineByKey(
```

```
    (marks: Int) => (marks, 1),                    // createCombiner
    (acc: (Int, Int), marks: Int) => (acc._1 + marks, acc._2 + 1),  // mergeValue
    (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)  // mergeCombiners
  )

  // Calculate average marks
  val averages = combined.mapValues { case (sum, count) => sum.toDouble / count }

  // Print averages to console
  averages.collect().foreach { case (student, avg) =>
   println(f"$student%s -> Average Marks: $avg%.2f")
  }

  // Save averages to output directory
  averages.saveAsTextFile("output_directory")

  // Stop Spark Context
  sc.stop()
 }
}
```

5)

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.HashPartitioner

object EmployeePartitionApp {
 def main(args: Array[String]): Unit = {
  // Step 1: Create Spark configuration and context
  val conf = new SparkConf().setAppName("EmployeePartitionApp").setMaster("local[*]")
  val sc = new SparkContext(conf)

  // Step 2: Sample Employee data (EmpID, Dept, EmpDesg)
  val employeeData = Array(
```

```
  (1, "HR", "Manager"),
  (2, "Finance", "Analyst"),
  (3, "HR", "Recruiter"),
  (4, "IT", "Developer"),
  (5, "IT", "Tester"),
  (6, "Finance", "Accountant"),
  (7, "Sales", "Executive"),
  (8, "Sales", "Manager")
)

// Step 3: Parallelize the employee data
val empRDD = sc.parallelize(employeeData)

// Step 4: Create a PairRDD using Dept as key
val pairRDD = empRDD.map { case (empId, dept, desg) =>
  (dept, (empId, desg))
}

// Step 5: Partition the RDD using HashPartitioner with 4 partitions
val partitionedRDD = pairRDD.partitionBy(new HashPartitioner(4))

// Step 6: Save the partitioned RDD to output directory
val outputDir = "employee_partition_output"
partitionedRDD.saveAsTextFile(outputDir)

// Step 7: Debug output — show which records went to which partition
println(s"\n--- Partitioned Output (Also saved to $outputDir) ---")
val debugRDD = partitionedRDD.mapPartitionsWithIndex {
  case (index, iter) => iter.map { case (dept, (id, desg)) =>
    s"Partition $index: (Dept: $dept, EmpID: $id, EmpDesg: $desg)"
  }
}
```

```
    debugRDD.collect().foreach(println)

    // Step 8: Stop Spark context
    sc.stop()
  }
}
```

6)

```
import org.apache.spark.{SparkConf, SparkContext}

object PartitionWithIndexExample {
  def main(args: Array[String]): Unit = {
    // Step 1: Spark Context setup
    val conf = new SparkConf().setAppName("PartitionWithIndex").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Step 2: Input collection and create RDD with 3 partitions
    val data = Seq(11, 34, 45, 67, 3, 4, 90)
    val rdd = sc.parallelize(data, 3)

    // Step 3: Use mapPartitionsWithIndex to increment elements and show partition info
    val resultRDD = rdd.mapPartitionsWithIndex {
      case (index, iter) =>
        val incremented = iter.map(_ + 1).toArray
        Iterator(s"Partition $index: [${incremented.mkString(", ")}]")
    }

    // Step 4: Save output to directory
    val outputDir = "partition_output"
    resultRDD.saveAsTextFile(outputDir)

    // Step 5: Optional: also print to console
```

```
    resultRDD.collect().foreach(println)

    // Step 6: Stop Spark context
    sc.stop()
  }
}
```

7)

```
import org.apache.spark.{SparkConf, SparkContext}

object ItemPartitionExample {
  def main(args: Array[String]): Unit = {
    // Step 1: Initialize SparkContext
    val conf = new SparkConf().setAppName("ItemPartitionExample").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Step 2: Define the Item map
    val Item = Map(
      "Ball" -> 10,
      "Ribbon" -> 50,
      "Box" -> 20,
      "Pen" -> 5,
      "Book" -> 8,
      "Dairy" -> 4,
      "Pin" -> 20
    )

    // Step 3: Create RDD from map with default partitioning
    val rdd = sc.parallelize(Item.toSeq)

    // Step 4: Print number of partitions
    val numPartitions = rdd.getNumPartitions
    println(s"Number of partitions: $numPartitions")
```

```
    // Step 5: Print full RDD content
    println("Full RDD content:")
    rdd.collect().foreach(println)

    // Step 6: Use mapPartitionsWithIndex to show each partition's content
    val partitionedRDD = rdd.mapPartitionsWithIndex {
      case (index, iter) =>
        val data = iter.map { case (k, v) => s"($k -> $v)" }.toArray
        Iterator(s"Partition $index: [${data.mkString(", ")}]")
    }

    // Step 7: Save partitioned content to output directory
    val outputDir = "item_partition_output"
    partitionedRDD.saveAsTextFile(outputDir)

    // Step 8: Optional: Print the partitioned data to console
    partitionedRDD.collect().foreach(println)

    // Step 9: Stop SparkContext
    sc.stop()
  }
}
```

8)

```
import org.apache.spark.{SparkConf, SparkContext}

object CustomPartitionExample {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("CustomPartitionExample").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Manually create the partitions as Seq of Seq
    val partition1 = Seq(("Ball", 10), ("Ribbon", 50), ("Box", 20))
```

```
val partition2 = Seq(("Pen", 5), ("Book", 8))
val partition3 = Seq(("Dairy", 4), ("Pin", 20))

// Create an RDD for each partition
val rdd1 = sc.parallelize(partition1, 1)
val rdd2 = sc.parallelize(partition2, 1)
val rdd3 = sc.parallelize(partition3, 1)

// Union all three RDDs (each with 1 partition) to get RDD with 3 partitions
val rdd = rdd1.union(rdd2).union(rdd3)

// Step i: Find number of partitions
val numPartitions = rdd.getNumPartitions
println(s"Number of partitions: $numPartitions")

// Step ii: Display full RDD content
println("Full RDD content:")
rdd.collect().foreach(println)

// Display content of each partition separately
val partitionedRDD = rdd.mapPartitionsWithIndex {
  case (index, iter) =>
    val data = iter.map { case (k, v) => s"($k -> $v)" }.toArray
    Iterator(s"Partition $index: [${data.mkString(", ")}]")
}

// Print partition contents
partitionedRDD.collect().foreach(println)

// === Save partitioned content to output directory ===
val outputDir = "item_partition_output"
partitionedRDD.saveAsTextFile(outputDir)
```

```
    sc.stop()
  }
}
```

9)

```
import org.apache.spark.{SparkConf, SparkContext}

object WordCountExample {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("WordCountExample").setMaster("local[*]")
    val sc = new SparkContext(conf)

    // Read the text file
    val lines = sc.textFile("words.txt")

    // Step i) Count occurrences of each word
    val words = lines
      .flatMap(_.split("\\W+"))    // split on non-word chars
      .filter(_.nonEmpty)
      .map(_.toLowerCase)

    val wordCounts = words.map(word => (word, 1)).reduceByKey(_ + _)

    // Step ii) Sort by word ascending
    val sortedWordCounts = wordCounts.sortByKey(ascending = true)

    // Step iii) Filter words that start with 's'
    val sWords = sortedWordCounts.filter { case (word, _) => word.startsWith("s") }

    // Print results on console
    println("Word Counts (sorted):")
    sortedWordCounts.collect().foreach(println)

    println("\nWords starting with 's':")
```

```
    sWords.collect().foreach(println)

    // Save outputs to directories
    sortedWordCounts.saveAsTextFile("output/sorted_word_counts")
    sWords.saveAsTextFile("output/words_starting_with_s")

    sc.stop()
  }
}
```

10)

```
import org.apache.spark.{SparkConf, SparkContext}

object CombineByKeyExample {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("CombineByKeyExample").setMaster("local[*]")
    val sc = new SparkContext(conf)

    val data = Array(("coffee", 2), ("cappuccino", 5), ("tea", 3), ("coffee", 10), ("cappuccino", 15))

    val rdd = sc.parallelize(data)

    // combineByKey to collect all values in a List[Int]
    val combined = rdd.combineByKey(
      (v: Int) => List(v),                    // createCombiner
      (acc: List[Int], v: Int) => v :: acc,    // mergeValue
      (acc1: List[Int], acc2: List[Int]) => acc1 ++ acc2 // mergeCombiners
    )

    // Reverse the lists to preserve original order
    val combinedOrdered = combined.mapValues(_.reverse)

    // Print the combined values
```

```
    println("Combined values by key:")
    combinedOrdered.collect().foreach{ case (k,v) => println(s"$k -> $v") }

    // Save the combined result as text files to output directory
    combinedOrdered.saveAsTextFile("output/combined_values")

    sc.stop()
  }
}



}
```