**What is Task 9 all about?**

It is a **hands-on exercise** using **Apache Spark's RDD API** to manipulate and analyze a dataset. You will use **Spark transformations and actions** to perform common data analysis tasks.

**Concepts Practiced in Task 9**

| **Step** | **Concept** | **What it teaches** |
| --- | --- | --- |
| a) | Loading data into an RDD | How to create an RDD from a file or data source |
| b) | count() action | How to count total records |
| c) | filter() transformation | How to filter records using conditions |
| d) | map() transformation | How to extract specific columns |
| e) | groupBy() + mapValues() + mean() logic | How to group by department and calculate average salary |

**Sample Dataset Used**

Here's an example of what the dataset might look like:

E001, Alice, IT, 75000, 2012

E002, Bob, HR, 50000, 2016

E003, Charlie, IT, 82000, 2014

E004, David, Finance, 67000, 2018

E005, Eva, HR, 59000, 2011

**Operations Performed in Task 9**

**a) Load the Dataset into an RDD**

Using sc.textFile("employees.csv") to load a text file where each line is a record.

**b) Count the number of records**

Using .count() to return the number of lines (i.e., employee records).

**c) Filter out employees who joined after 2014**

Using .filter() and splitting the line:

rdd.filter(lambda line: int(line.split(",")[4]) > 2014)

**d) Map to employee names**

Using .map() to return only names:

rdd.map(lambda line: line.split(",")[1])

**e) Calculate average salary per department**

Steps:

1. Extract (Department, Salary) as key-value pairs.
2. Use groupByKey() or reduceByKey() to collect salaries per department.
3. Calculate average.

dept\_salary = rdd.map(lambda line: (line.split(",")[2], int(line.split(",")[3])))

avg\_salary = dept\_salary.groupByKey().mapValues(lambda vals: sum(vals)/len(vals))

**Summary**

By completing Task 9, you will:

* Understand the **core RDD data structure**.
* Practice using **transformations (map, filter)** and **actions (count, collect)**.
* Apply **real-life logic** on a distributed system (like filtering employees by year or computing departmental stats).
* Gain foundational skills to write **Spark-based data processing code**.

**Step-by-Step Guide for Task 9**

**Step 1: Start the Spark Shell**

spark-shell

**Explanation:**

* This command starts the **interactive Scala shell** for Apache Spark.
* You can now type and run Spark commands directly.

Step 2: Create a Sample Dataset File

**Step A: Create the File on Your Local System**

1. Open **Notepad** or any plain text editor on your laptop.
2. Paste the dataset below:

E101,Ali,IT,80000,2012

E102,Beena,HR,50000,2015

E103,Charles,IT,85000,2014

E104,Daisy,Finance,60000,2018

E105,Eshan,HR,55000,2011

E106,Fatima,IT,83000,2017

E107,Ganesh,Finance,62000,2013

E108,Hari,IT,91000,2019

E109,Isha,HR,58000,2020

E110,John,IT,88000,2016

1. Save the file as **employee.txt** on your desktop or in a known folder.

**Step B: Upload to VM Using WinSCP**

1. Open **WinSCP** and connect to your Cloudera VM.
2. On the left side (local), browse to the folder where employee.txt is saved.
3. On the right side (remote), go to your working directory (e.g., /home/cloudera/).
4. **Drag and drop** the file from left to right to upload.

**Step C: Confirm File in VM**

Switch back to the Cloudera terminal and run:

ls -l employee.txt

You should see the file listed, confirming it is ready for use in Spark.

**Put employee.txt into HDFS**

**Step 1: Check the file exists in local file system**

Run in terminal of cloudera:

ls /home/cloudera/employee.txt

If it shows the file, you're good.

**Step 2: Put the file into HDFS**

Now copy the file from local to HDFS using:

hadoop fs -put /home/cloudera/employee.txt /user/cloudera/

🔸 This command uploads employee.txt into HDFS under /user/cloudera/

**Step 3: Confirm the file is now in HDFS**

Run:

hadoop fs -ls /user/cloudera/

You should see employee.txt listed there.

You can now directly use this file in the spark-shell using:

val empRDD = sc.textFile("employee.txt")

Explanation of the above command :

| **Part** | **Meaning** |
| --- | --- |
| val empRDD | You are declaring a variable named empRDD to hold the RDD (Resilient Distributed Dataset). |
| sc | The SparkContext object that lets you connect with the Spark cluster — already available in spark-shell. |
| textFile("employee.txt") | This loads the text file as an RDD, with each **line** in the file becoming **one element** in the RDD. |

So, if your employee.txt looks like this:

101,John,IT,50000,2015

102,Jane,HR,45000,2016

103,Ravi,IT,52000,2013

Then your RDD will have **3 lines**, each stored as a **String**.

Basically, You are asking Spark to:

* Read the file line-by-line.
* Treat **each line** as one **data item** (called an **element**) in the RDD.
* And store it as a **string of text** — because the text file is unstructured.

**Example:**

Assume your file employee.txt contains:

101,John,IT,50000,2015

102,Jane,HR,45000,2016

103,Ravi,IT,52000,2013

Then after loading, the empRDD will look like:

[

"101,John,IT,50000,2015", // Line 1 → Element 1 (String)

"102,Jane,HR,45000,2016", // Line 2 → Element 2 (String)

"103,Ravi,IT,52000,2013" // Line 3 → Element 3 (String)

]

So it's an RDD containing 3 **strings** — one per line in the file.

**Step 3: Count the Number of Records in the RDD**

**Command:**

empRDD.count()

**Explanation:**

| **Part of the Command** | **Meaning** |
| --- | --- |
| empRDD | This is the RDD variable holding your employee dataset. |
| .count() | This is an **action** (not a transformation). It **triggers computation** and returns the **total number of lines** (or records) present in the RDD. |

**What This Does:**

* Spark looks at all the lines in the employee.txt file.
* Counts them one by one across all nodes (if it's distributed).
* Returns the total number of records to your terminal.

**Example Output:**

If your file contains 10 employee records, you'll see:

res0: Long = 10

res0 is a temporary variable storing the result.  
 Long = 10 means 10 records were counted.

**Step 4: Filter Employees Who Joined After 2014**

**Objective**

We want to select only those employees **whose year of joining is greater than 2014**, from the existing RDD.

**Recap: What does your empRDD contain?**

If your dataset (employee.txt) looks like this:

101,John,IT,60000,2012

102,Alice,HR,50000,2015

103,David,IT,70000,2018

104,Emma,Sales,55000,2013

Then, each line in the RDD is **a single string**, for example:

"101,John,IT,60000,2012"

**Step-by-Step Code with Explanation**

**Scala Command**

val empFilteredRDD = empRDD.filter(line => line.split(",")(4).toInt > 2014)

**Explanation of the Command:**

**empRDD.filter(...)**

* filter() is a **transformation** operation in Spark.
* It is used to keep only those records that match a **condition**.
* Here, we apply the filter to the existing RDD empRDD.

**line => ...**

* This is a **lambda expression** (anonymous function).
* It means: for **each line** (which is a string), apply the logic inside {}.

**line.split(",")**

* This **splits the string** by commas.
* For example: "101,John,IT,60000,2012" becomes an array:
* Array("101", "John", "IT", "60000", "2012")

**(4)**

* Accesses the **5th element** of the array (index starts from 0).
* In our example, 2012 is at index 4.

**.toInt**

* Converts the year from string to integer so we can compare it using >.

**> 2014**

* Keeps only the records where year of joining is after 2014.

**Output RDD**

empFilteredRDD now contains only:

102,Alice,HR,50000,2015

103,David,IT,70000,2018

**To print the output**

Use this:

empFilteredRDD.collect().foreach(println)

It will display the filtered lines.

### Explanation :

**empFilteredRDD**

* This is your RDD that was filtered in the previous step.
* It contains only those employee records where Year of Joining > 2014.

**.collect()**

* **Collect is an Action** in Spark.
* It gathers all the data from the distributed RDD (across cluster nodes) **back to the driver program** (your main computer).

Example:

val sample = sc.parallelize(Array(1, 2, 3))

sample.collect()

gives:

Array(1, 2, 3)

**.foreach(println)**

* Once the data is collected into an array, this part prints each element **line by line**.
* foreach means: “for each element in the array, apply the function println()”.
* In our case, each element is a string like:
* 102,Alice,HR,50000,2015

**Final Outcome:**

This command simply **prints all the filtered records** from the RDD to your screen — one per line — like this:

102,Alice,HR,50000,2015

103,David,IT,70000,2018

**Step 5: Map the RDD to Only Employee Names**

This step creates a new RDD that **contains only the names** of the employees who joined after 2014.

**Command:**

val empNamesRDD = empFilteredRDD.map(line => line.split(",")(1))

**Explanation:**

Let’s break it down part-by-part:

**1. empFilteredRDD**

* This is the RDD that contains only records where employees joined after 2014.
* Each record looks like this (as a string):
* 102,Alice,HR,50000,2015
* 103,David,IT,70000,2018

**2. .map(...)**

* This is a **Transformation** in Spark.
* It means: for each line in the RDD, apply a function that transforms the data.
* The result is a new RDD of transformed values (without changing the original RDD).

**3. line => line.split(",")(1)**

This is a **lambda function** (anonymous function) that does the following:

* line refers to a single row in the RDD, like:
* "102,Alice,HR,50000,2015"
* line.split(",") splits the string into an array:
* Array("102", "Alice", "HR", "50000", "2015")
* line.split(",")(1) takes the second item (index 1), which is the **employee name**.

So the result is:

Alice

David

...

**Final Result:**

You get a new RDD called empNamesRDD that holds only:

Alice

David

...

**To Display the Names:**

You can print the result:

empNamesRDD.collect().foreach(println)

## **Step 6: Calculate the Average Salary of Employees per Department**

### Goal:

We want to group employees **by department** and calculate the **average salary** for each department.

To do this in Spark, we’ll:

1. Extract department and salary fields.
2. Convert salary from String to Int.
3. Group by department.
4. Calculate the average.

### Step-by-Step Commands and Explanations:

#### Step 6.1: Extract Department and Salary from Filtered Data

val deptSalRDD = empFilteredRDD.map(line => {

val parts = line.split(",")

(parts(2), parts(3).toInt)

})

#### Explanation:

* empFilteredRDD: has lines like:
* 102,Alice,HR,50000,2015
* 103,David,IT,70000,2018
* .map(...):
  + line.split(",") → turns each line into an array: Array("102", "Alice", "HR", "50000", "2015")
  + parts(2) → department (e.g., "HR")
  + parts(3).toInt → salary converted from string to integer (e.g., 50000)
* Result: an RDD of pairs like:
* ("HR", 50000)
* ("IT", 70000)

#### Step 6.2: Calculate Total Salary and Count per Department

val deptSalCountRDD = deptSalRDD.mapValues(sal => (sal, 1))

.reduceByKey((a, b) => (a.\_1 + b.\_1, a.\_2 + b.\_2))

#### Explanation:

* mapValues(sal => (sal, 1)):
  + Changes each item from:
  + ("HR", 50000) → ("HR", (50000, 1))
  + Here, 1 represents 1 employee.
* reduceByKey(...):
  + Groups by department and adds both salary and count:
  + ("HR", (50000, 1)) + ("HR", (55000, 1)) → ("HR", (105000, 2))

#### Step 6.3: Calculate Average Salary

val deptAvgSalRDD = deptSalCountRDD.mapValues {

case (totalSalary, count) => totalSalary / count

}

#### Explanation:

* For each department like:
* ("HR", (105000, 2))

It calculates:

105000 / 2 = 52500

#### Step 6.4: Display the Result

deptAvgSalRDD.collect().foreach(println)

#### Sample Output:

(HR,52500)

(IT,70000)