Introduction to Apache Spark

- Imagine we have a really big pile of data, like a mountain of information. Apache Spark is like a powerful machine that helps us dig through that mountain super-fast.
- Instead of using just one small shovel (like a regular computer), Spark gives you lots of shovels (like many computers working together). So, we can break up the mountain into smaller pieces and have all the shovels working at the same time to dig through it.
- Spark is really good at handling big data and doing all sorts of tasks with it, like analyzing, processing, and finding important stuff within it. It's like having a team of super-fast data detectives helping you make sense of all that information!

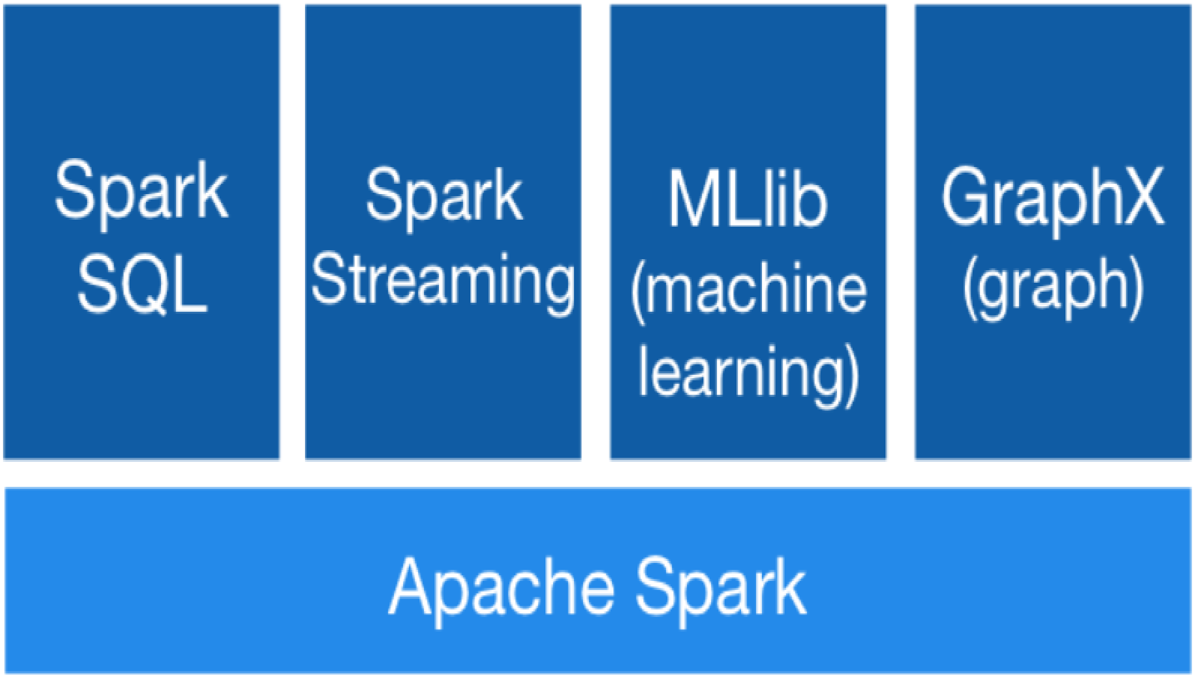
Spark v/s Hadoop?

- Imagine we have a huge garden full of different types of plants. We want to water all the plants efficiently.
- Hadoop is like a sprinkler system. It's effective at covering large areas of the garden with water, but it might take a bit of time to reach all the plants, especially if they're spread out far apart.

- Apache Spark is like having a team of gardeners with hoses. Each gardener focuses on watering a specific section of the garden, and they all work together simultaneously. This way, the entire garden gets watered much faster than with just the sprinkler system.
- In this analogy, Hadoop provides a systematic approach to data processing, while Apache Spark offers parallel processing capabilities, allowing for faster and more efficient handling of large volumes of data.

What is Spark?

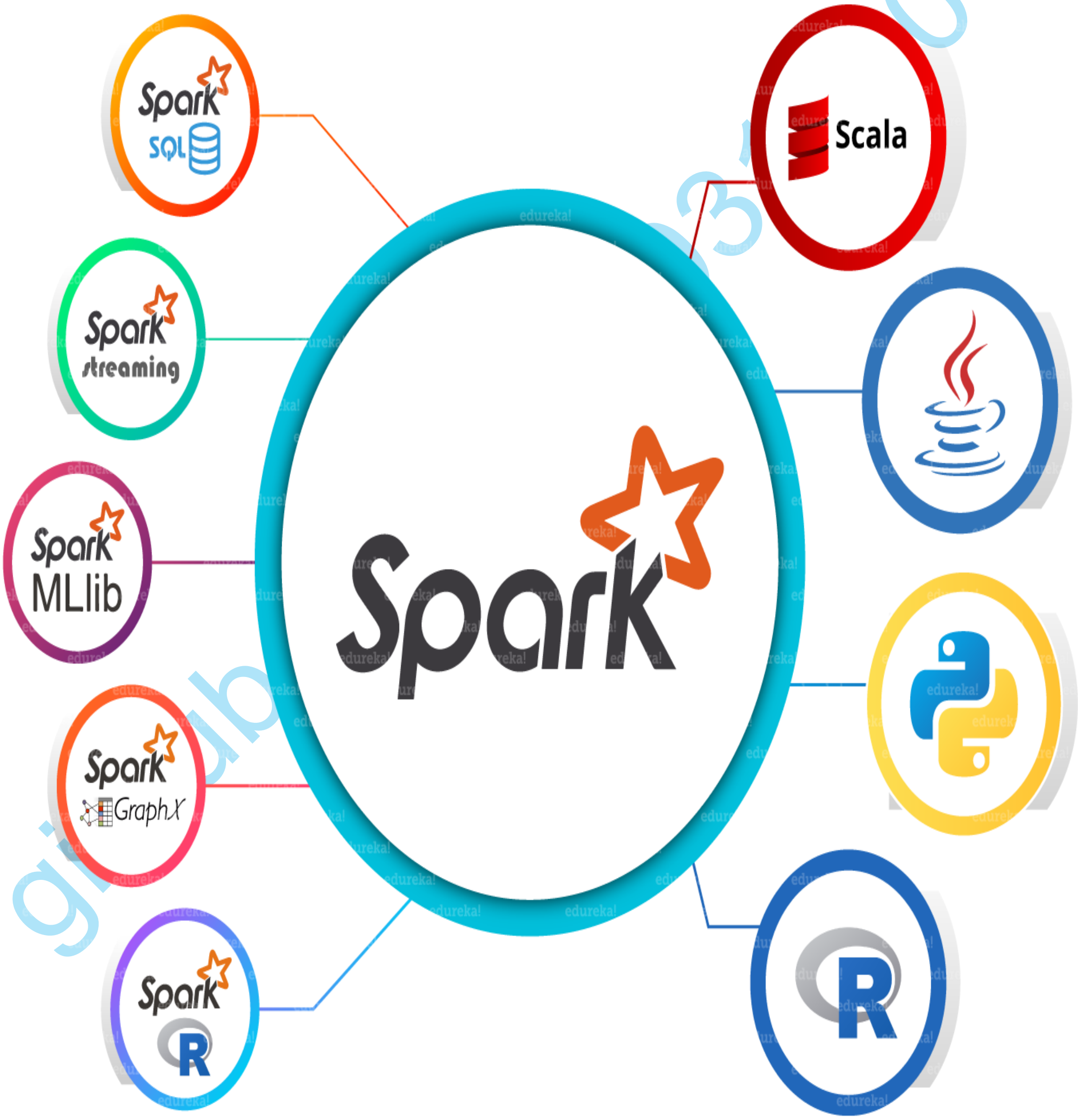
- Apache Spark is an open-source cluster computing framework. Its primary purpose is to handle the real-time generated data.
- Spark takes large data sets, break them down into smaller, manageable parts and then processes these parts across multiple computers at the same time.
- Spark was built on the top of the Hadoop MapReduce. It was optimized to run in memory whereas alternative approaches like Hadoop's MapReduce writes data to and from computer hard drives. So, Spark process the data much quicker than other alternatives.



History of Apache Spark

- The Spark was initiated by Matei Zaharia at UC Berkeley's AMP Lab in 2009. It was open sourced in 2010 under a BSD license.
- In 2013, the project was acquired by Apache Software Foundation. In 2014, the Spark emerged as a Top-Level Apache Project.

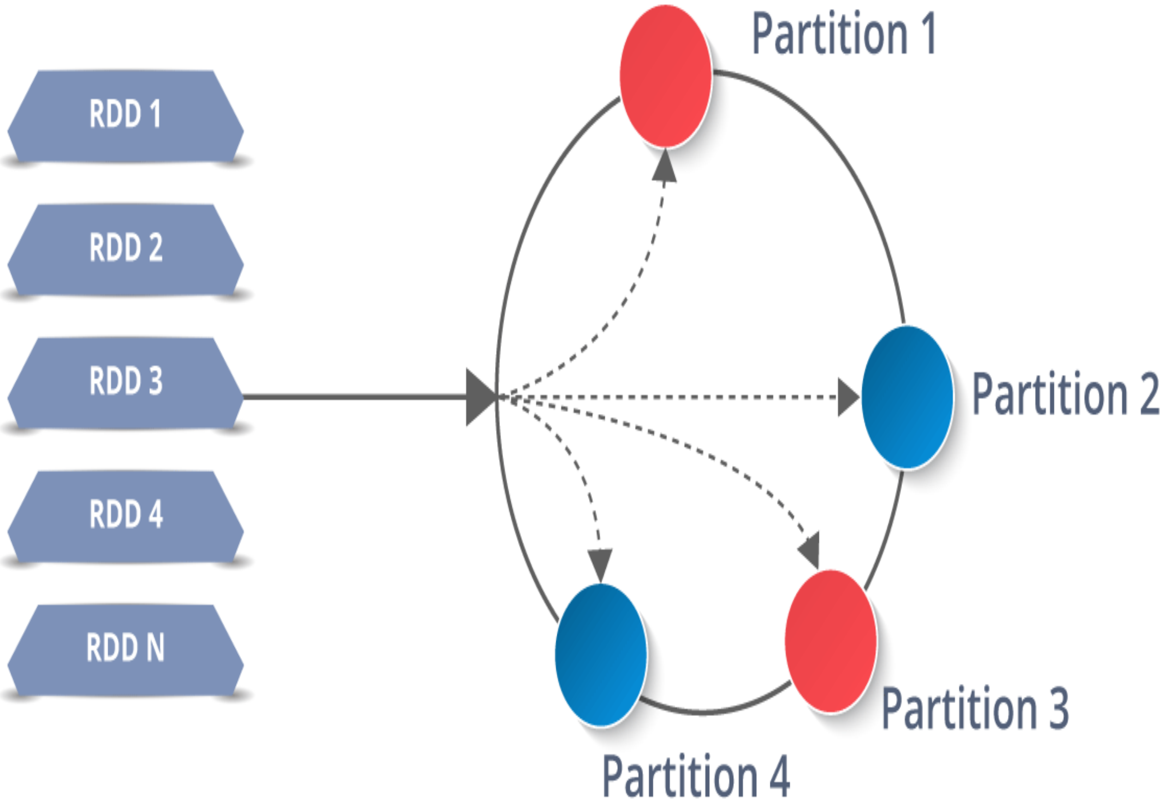
Apache Spark Ecosystem



Resilient Distributed Datasets (RDD)

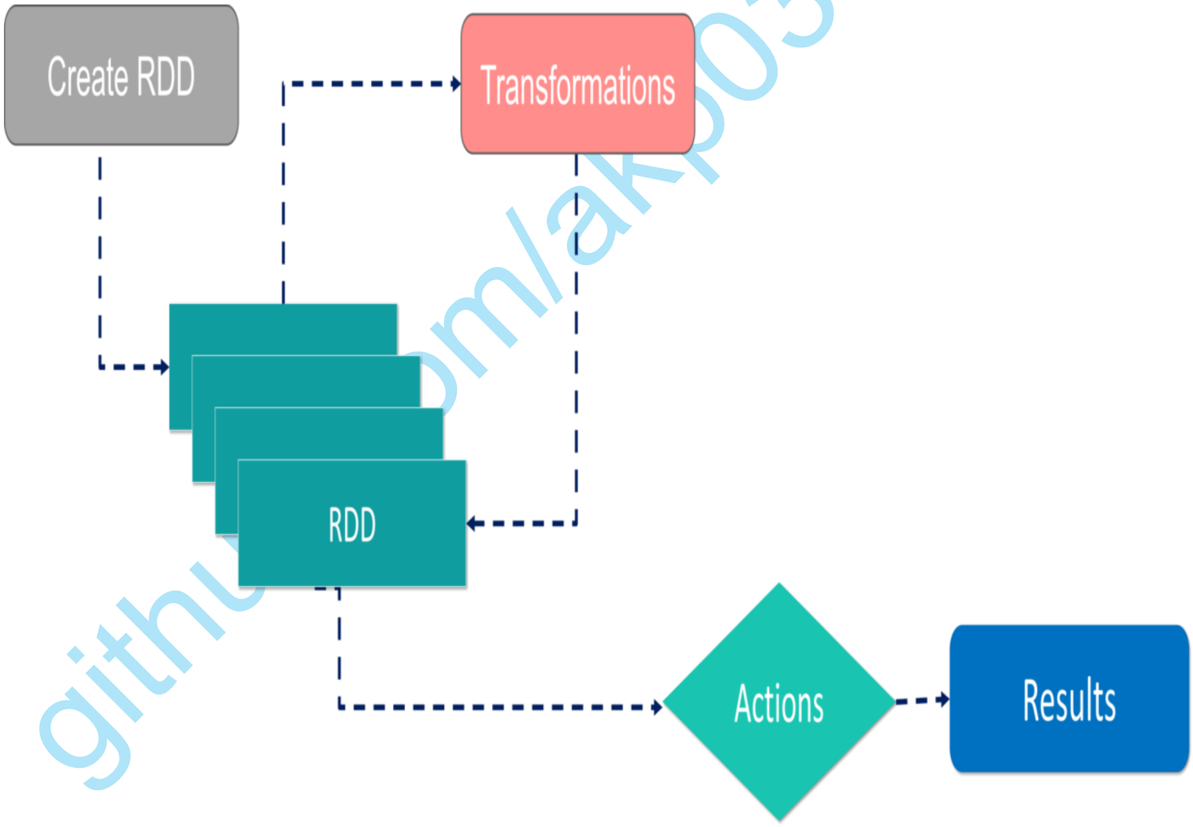
- RDDs are the building blocks of any Spark application. RDDs Stands for:
- Think of an RDD as a big, flexible bag that holds our data. This bag is special because it can be spread out over many computers (distributed) and it's resilient, meaning even if one of those computers goes stopped or malfunctioned, our data is safe and can be reconstructed from the other computers.

    - **Resilient:** If a computer holding part of the RDD fails, Spark can reconstruct that part using the data stored on other computers. It's like having multiple backups, so even if one fails, our data is still accessible.
    - **Distributed:** RDDs can be split into smaller chunks and stored on multiple computers. This allows Spark to process the data in parallel, making things faster.
    - **Dataset:** It's just our data, whether it's numbers, text, or anything else. RDDs can hold any type of data we want to work with.

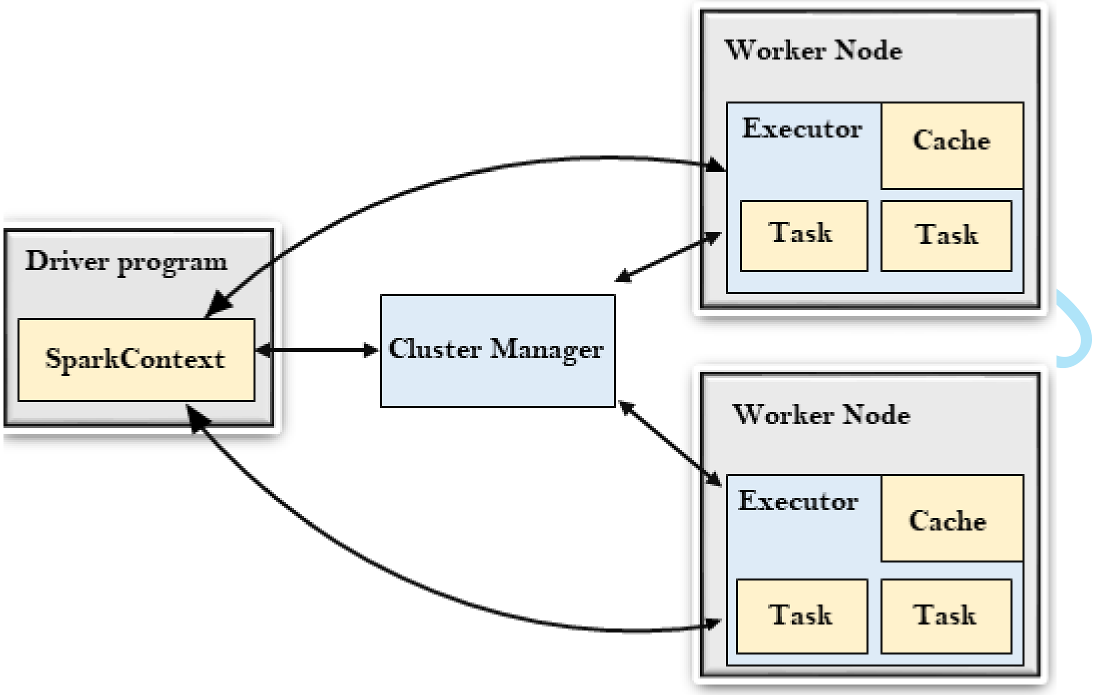


- Features of RDD:
  - In-Memory
  - Immutable
  - Lazy Evaluated
  - Parallel

- With RDDs, we can perform two types of operations:
  - **Transformations:** They are the operations that are applied to create a new RDD.
  - **Actions:** They are applied on an RDD to instruct Apache Spark to apply computation and pass the result back to the driver.



Spark Architecture



- In our master node, we have the **driver program**, which drives our application. The code we are writing behaves as a driver program or if we are using the interactive shell, the shell acts as the driver program.

- Inside the driver program, the first thing we do is, we create a **Spark Context**. Assume that the Spark context is a gateway to all the Spark functionalities. It is similar to our database connection. Any command we execute in our database goes through the database connection. Likewise, anything we do on Spark goes through Spark context.

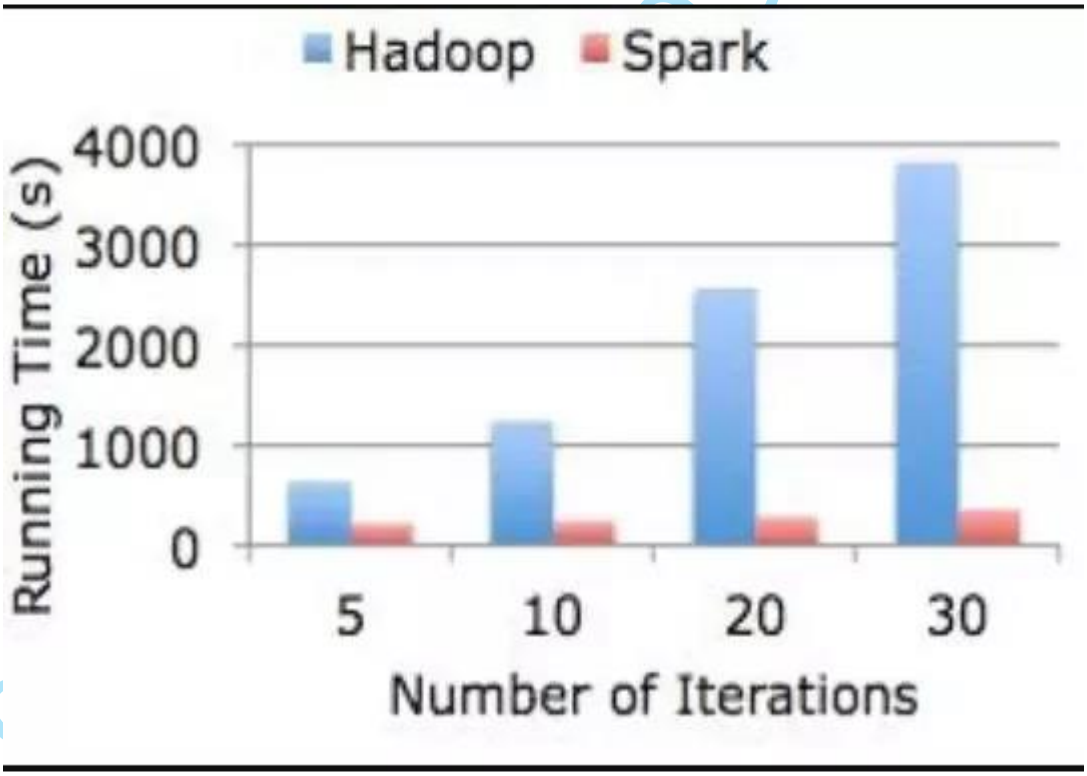
- Now, this Spark context works with the cluster manager to manage various jobs. The driver program & Spark context takes care of the job execution within the cluster. A job is split into multiple tasks which are distributed over the worker node. Anytime an RDD is created in Spark context, it can be distributed across various nodes and can be cached there.

- Worker nodes are the slave nodes whose job is to basically execute the tasks. These tasks are then executed on the partitioned RDDs in the worker node and hence returns back the result to the Spark Context.

- Spark Context takes the job, breaks the job in tasks and distribute them to the worker nodes. These tasks work on the partitioned RDD, perform operations, collect the results and return to the main Spark Context.

- If we increase the number of workers, then we can divide jobs into more partitions and execute them parallelly over multiple systems. It will be a lot faster.

- With the increase in the number of workers, memory size will also increase & we can cache the jobs to execute it faster.

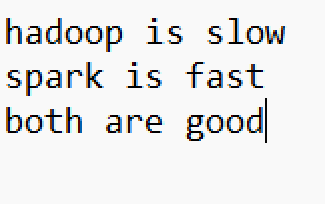
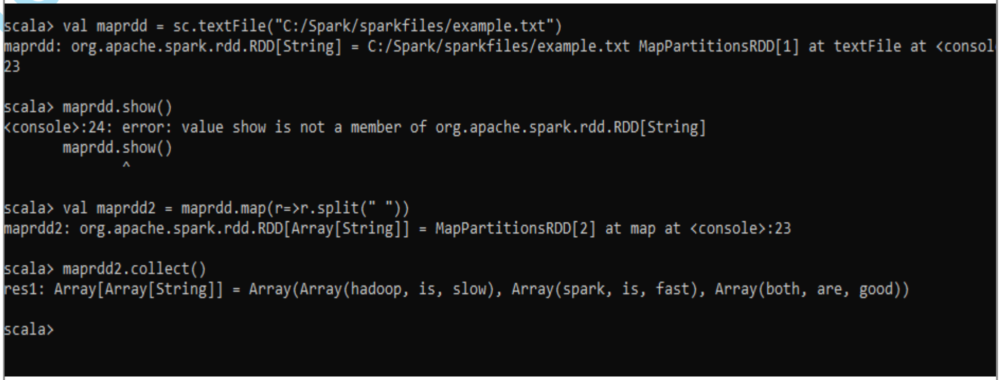


Different Ways to Create Resilient Distributed Datasets (RDD)

Parallelize Method	val data = Seq(("A", 1), ("B", 2), ("C", 3)) val r1 = spark.sparkContext.parallelize(data) val r1 = spark.sparkContext.parallelize(Seq(("A",1),("B",2),("C",3))) val r1 = spark.sparkContext.parallelize(List(("A", 1), ("B", 2), ("C", 3))) val r1 = spark.sparkContext.parallelize(Array(("A", 1), ("B", 2), ("C", 3))) val r1 = spark.sparkContext.parallelize(Vector(("A", 1), ("B", 2), ("C", 3))) val r1 = spark.sparkContext.parallelize(1 to 100) r1.foreach(println) val data = List(("A", 1), ("B", 2), ("C", 3)) val r1 = spark.sparkContext.parallelize(data)
Text File Method	val r2 = spark.sparkContext.textFile("C:/Spark/sparkfiles/g2.txt") r2.collect()
From Existing RDD	• We can use transformations like map, flatmap, filter to create a new RDD from an existing one. val r3 = r2.flatMap(_.split(",")) r3.collect()

From existing DataFrames and DataSet	val r4 = spark.range(20).toDF()**.rdd** val r4 = dataframe**.rdd** r4.collect()

Transformation in Spark RDD API

map()	Transformation applies a function to each row in a Data Frame/Dataset and returns the new transformed Dataset. val maprdd = sc.textFile("C:/Spark/sparkfiles/example.txt")  hadoop is slow spark is fast both are good val maprdd2 = maprdd.map(r=>r.split(" ")) maprdd2.collect()  scala> val maprdd = sc.textFile("C:/Spark/sparkfiles/example.txt") maprdd: org.apache.spark.rdd.RDD[String] = C:/Spark/sparkfiles/example.txt MapPartitionsRDD[1] at textFile at <consol 23 scala> maprdd.show() <console>:24: error: value show is not a member of org.apache.spark.rdd.RDD[String] maprdd.show() ^ scala> val maprdd2 = maprdd.map(r=>r.split(" ")) maprdd2: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at map at <console>:23 scala> maprdd2.collect() res1: Array[Array[String]] = Array(Array(hadoop, is, slow), Array(spark, is, fast), Array(both, are, good)) scala> map maintains a one-to-one relationship between input and output elements

RDD with some numbers, and we want to double each number using the map operation.

// Create an RDD with some numbers

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

// Use map to double each number

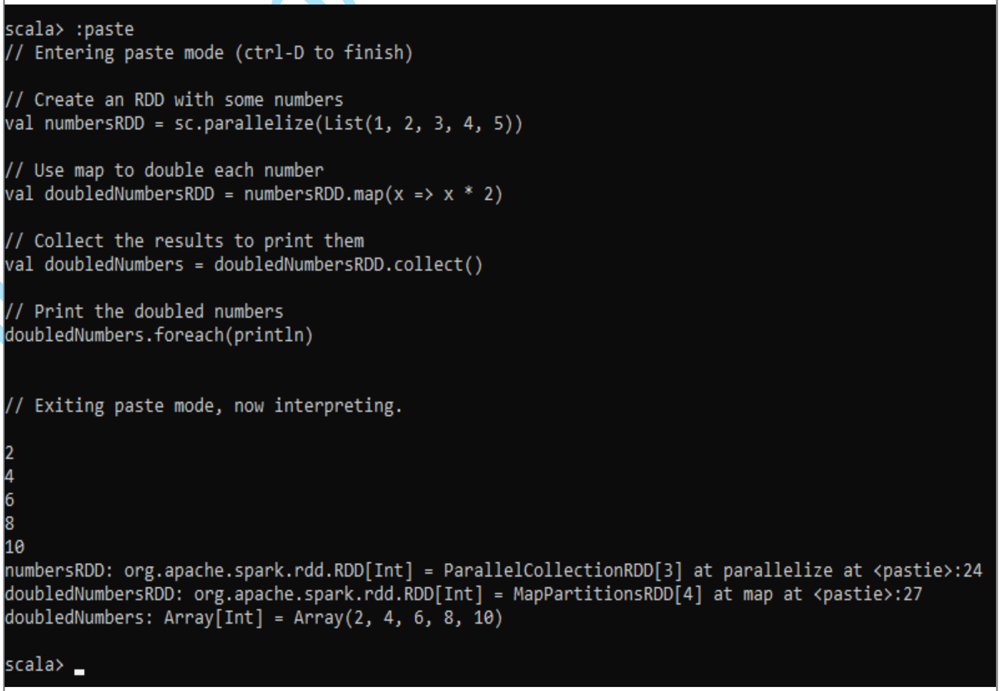
val doubledNumbersRDD = numbersRDD.map(x => x * 2)

// Collect the results to print them

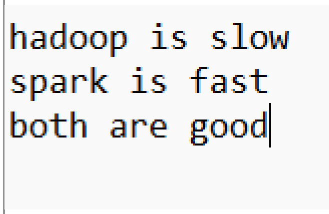
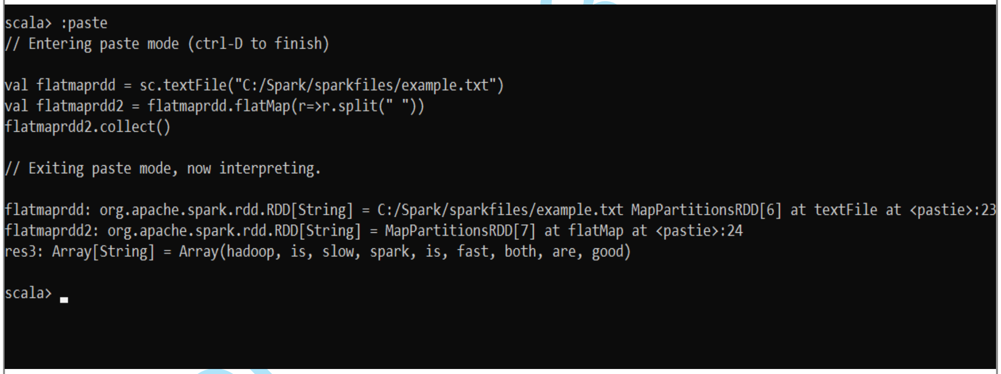
val doubledNumbers = doubledNumbersRDD.collect()

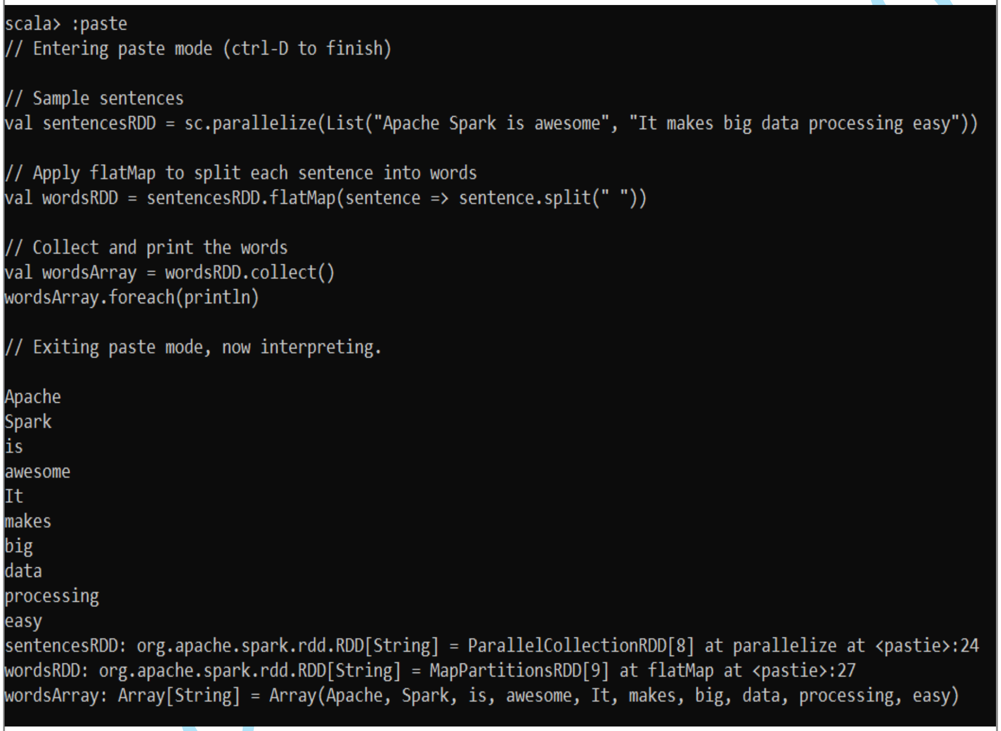
// Print the doubled numbers

doubledNumbers.foreach(println)

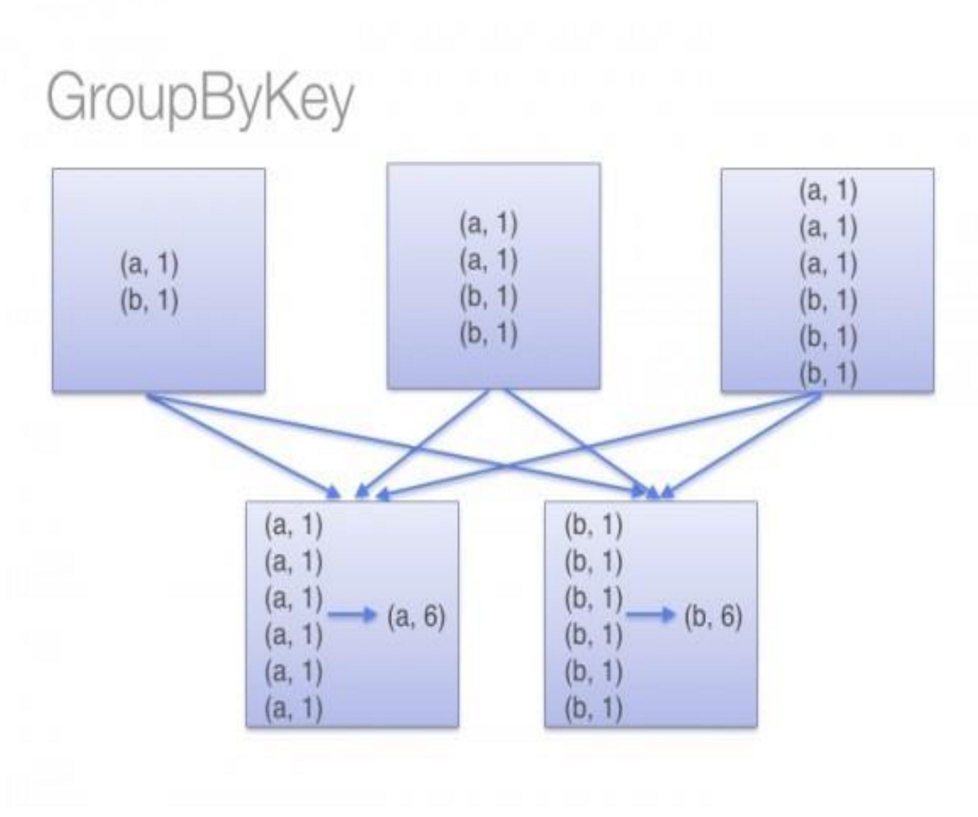


```
scala> :paste
// Entering paste mode (ctrl-D to finish)

// Create an RDD with some numbers
val numbersRDD = sc.parallelize(List(1, 2, 3, 4, 5))

// Use map to double each number
val doubledNumbersRDD = numbersRDD.map(x => x * 2)

// Collect the results to print them
val doubledNumbers = doubledNumbersRDD.collect()

// Print the doubled numbers
doubledNumbers.foreach(println)


// Exiting paste mode, now interpreting.

2
4
6
8
10
numbersRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <pastie>:24
doubledNumbersRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[4] at map at <pastie>:27
doubledNumbers: Array[Int] = Array(2, 4, 6, 8, 10)

scala> _
```

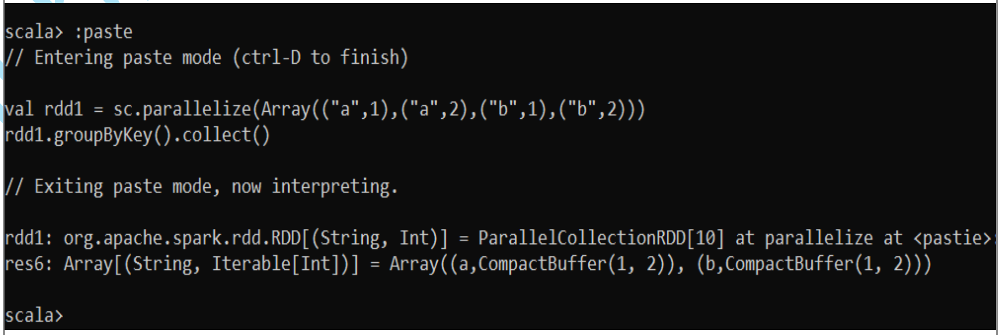
flatMap()	Transformation flattens the Data Frame/Dataset after applying the function on every element and returns a new transformed Dataset.
	
	val flatmaprdd = sc.textFile("C:/Spark/sparkfiles/example.txt")
	
	
	hadoop is slow
	spark is fast
	both are good
	
	val flatmaprdd2 = flatmaprdd.flatMap(r=>r.split(" "))
	flatmaprdd2.collect()
	
	
	scala> :paste
	// Entering paste mode (ctrl-D to finish)
	
	val flatmaprdd = sc.textFile("C:/Spark/sparkfiles/example.txt")
	val flatmaprdd2 = flatmaprdd.flatMap(r=>r.split(" "))
	flatmaprdd2.collect()
	
	// Exiting paste mode, now interpreting.
	
	flatmaprdd: org.apache.spark.rdd.RDD[String] = C:/Spark/sparkfiles/example.txt MapPartitionsRDD[6] at textFile at <pastie>:23
	flatmaprdd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[7] at flatMap at <pastie>:24
	res3: Array[String] = Array(hadoop, is, slow, spark, is, fast, both, are, good)
	
	scala>
	
	flatMap allows for a one-to-many relationship, and can generate zero or more output elements
	
	Suppose we have an RDD with sentences, and we want to split each sentence into words using flatMap.
	
	// Sample sentences
	
	val sentencesRDD = sc.parallelize(List("Apache Spark is awesome", "It makes big data processing easy"))
	
	// Apply flatMap to split each sentence into words

	val wordsRDD = sentencesRDD.flatMap(sentence => sentence.split(" "))
	
	// Collect and print the words
	val wordsArray = wordsRDD.collect()
	wordsArray.foreach(println)
	
	scala> :paste
	// Entering paste mode (ctrl-D to finish)
	
	// Sample sentences
	val sentencesRDD = sc.parallelize(List("Apache Spark is awesome", "It makes big data processing easy"))
	
	// Apply flatMap to split each sentence into words
	val wordsRDD = sentencesRDD.flatMap(sentence => sentence.split(" "))
	
	// Collect and print the words
	val wordsArray = wordsRDD.collect()
	wordsArray.foreach(println)
	
	// Exiting paste mode, now interpreting.
	
	Apache
	Spark
	is
	awesome
	It
	makes
	big
	data
	processing
	easy
	sentencesRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[8] at parallelize at <pastie>:24
	wordsRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[9] at flatMap at <pastie>:27
	wordsArray: Array[String] = Array(Apache, Spark, is, awesome, It, makes, big, data, processing, easy)

groupByKey()	groupByKey shuffles the entire dataset across the network based on the key.



```
val rdd1 = sc.parallelize(Array(("a",1),("a",2),("b",1),("b",2)))
rdd1.groupByKey().collect()
```



```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val rdd1 = sc.parallelize(Array(("a",1),("a",2),("b",1),("b",2)))
rdd1.groupByKey().collect()

// Exiting paste mode, now interpreting.

rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[10] at parallelize at <pastie>
res6: Array[(String, Iterable[Int])] = Array((a,CompactBuffer(1, 2)), (b,CompactBuffer(1, 2)))

scala>
```

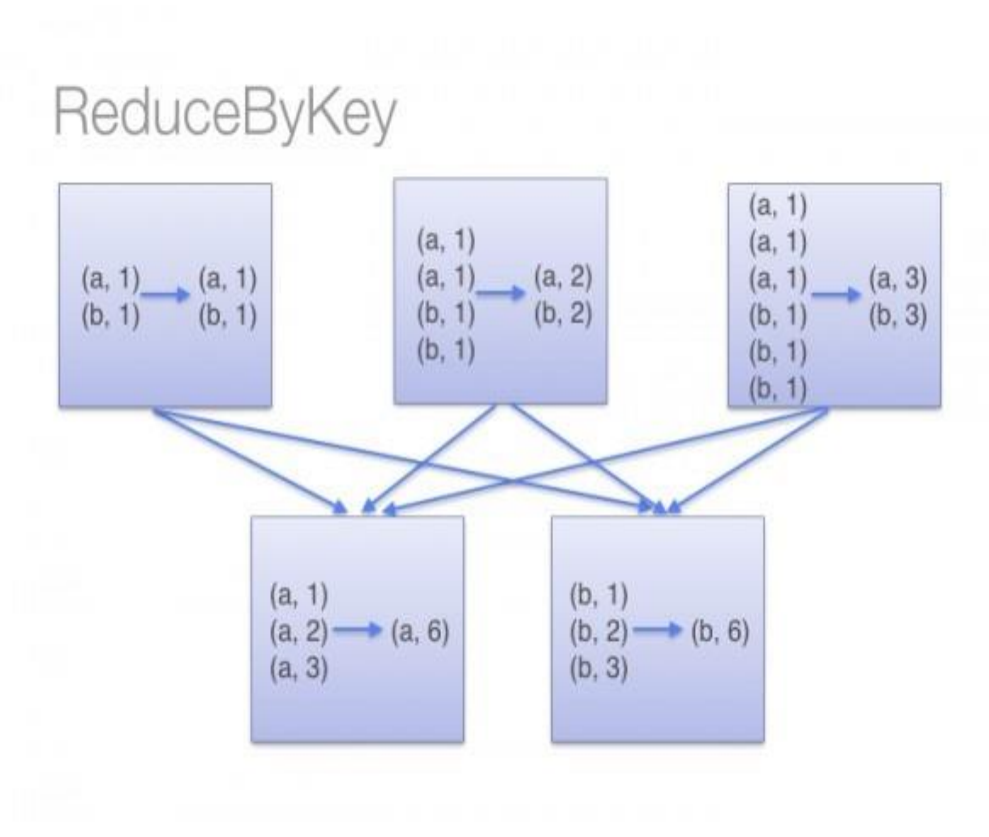
A simple example to understand groupBykey():

// Example RDD of books

<table>
<tr>
<td></td>
<td>
<pre>
val books = sc.parallelize(Seq(
  ("Fiction", "Book1"),
  ("Fiction", "Book2"),
  ("Non-Fiction", "Book3"),
  ("Fiction", "Book4"),
  ("Non-Fiction", "Book5")
))

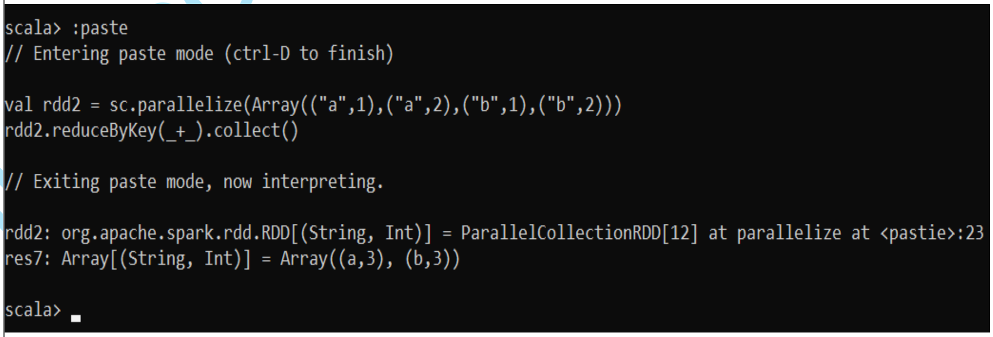
// Applying reduceByKey to count the number of books in each genre
val bookCounts = books.map(genre => (genre._1, 1)).reduceByKey(_ + _)

// Collecting and printing the result
bookCounts.collect().foreach(println)
</pre>
<p><b>Output:</b></p>
<pre>
(Fiction,3)
(Non-Fiction,2)
</pre>
</td>
</tr>
<tr>
<td><b>reduceByKey()</b></td>
<td>reduceByKey will compute local sums for each key in each partition and combine those local sums into larger sums after shuffling.</td>
</tr>
</table>



val rdd2 = sc.parallelize(Array(("a",1),("a",2),("b",1),("b",2)))

rdd2.reduceByKey(_+_).collect()



```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val rdd2 = sc.parallelize(Array(("a",1),("a",2),("b",1),("b",2)))
rdd2.reduceByKey(_+_).collect()

// Exiting paste mode, now interpreting.

rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[12] at parallelize at <pastie>:23
res7: Array[(String, Int)] = Array((a,3), (b,3))

scala> 
```

A simple example of reducing duplicate numbers in an RDD using reduceByKey.

// Example RDD with numbers

```
val numbersRDD = sc.parallelize(Seq(
  (1, "one"),
  (2, "two"),
  (3, "three"),
  (1, "uno"),
  (2, "dos"),
  (3, "tres"),
  (1, "eins")
))

// Apply reduceByKey to count the occurrences of each number
val countRDD = numbersRDD.map(num => (num._1, 1)).reduceByKey(_ + _)

// Collect and print the counts
val countArray = countRDD.collect()
countArray.foreach(println)
```

Output:

```
(1,3)
(2,2)
(3,2)
```

Action in Spark RDD API

Apache Spark RDD Actions are operations that trigger the execution of transformations and return results to the driver program or write data to external storage. Unlike transformations, which are lazily evaluated, actions are eagerly executed. Here are some common RDD actions in Apache Spark, along with examples:

collect()	This action retrieves all elements of the RDD and brings them to the driver program as an array. val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.collect() println(result.mkString(", ")) Output: 1, 2, 3, 4, 5 println(result.mkString("/ ")) Output: 1/ 2/ 3/ 4/ 5 **Note:** The **mkString(", ")** method is used in Scala to concatenate the elements of a collection into a single string, with each element separated by a specified delimiter. For example, if we have a list of strings `["one", "two", "three"]` and we call `mkString(", ")`, it will join the elements of the list into a single string with each element separated by a comma and a space, resulting in `"one, two, three"`. In the context of the examples provided earlier, `mkString(", ")` is used to join the elements of an array into a single string with a comma and a space between each element, before printing it out. This is helpful for formatting the output when printing collections.

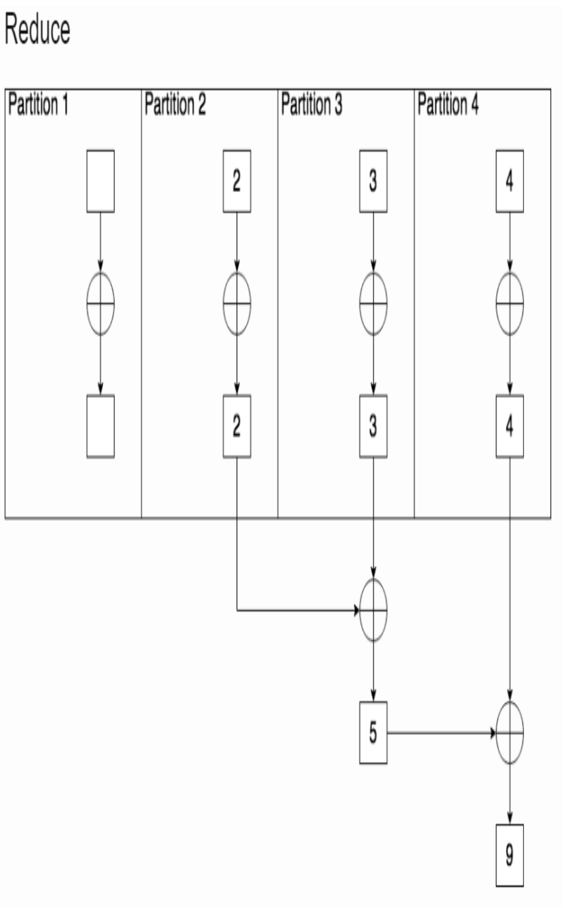
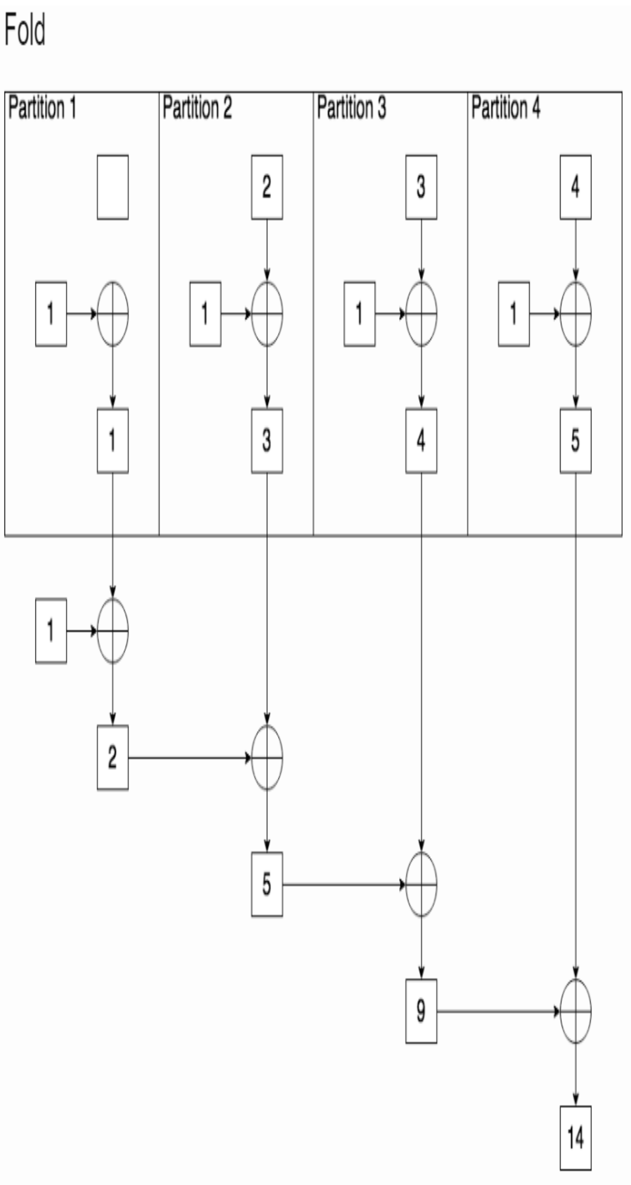
count()	This action counts the number of elements in the RDD. val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.count() println(result)
take(n)	This action retrieves the first n elements of the RDD and returns them as an array. val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.take(3) println(result.mkString(", ")) Output: 1, 2, 3
first()	This action returns the first element of RDD. val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.first() println(result) Output: 1
reduce(func)	This action aggregates the elements of the RDD using a specified binary function func. val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5))

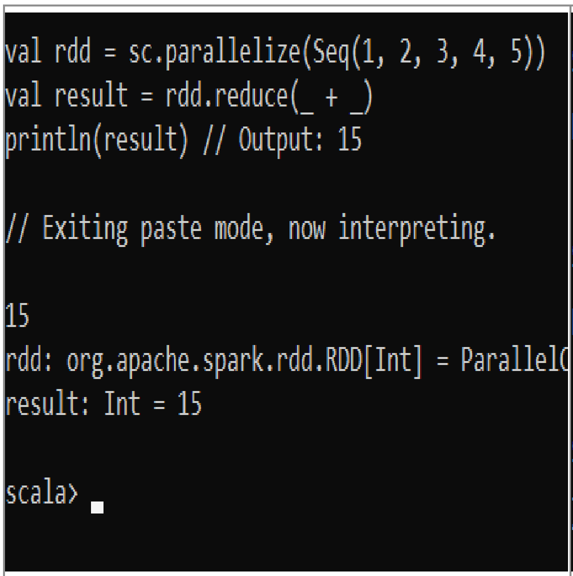
	val result = rdd.reduce(_ + _) println(result) Output: 15
saveAsTextFile(path)	This action saves the RDD as a text file or files in the specified directory. val rdd = sc.parallelize(Seq("Hello", "World", "Apache", "Spark")) rdd.saveAsTextFile("C:/Spark/sparkfiles/output_dir") Note: output_dir must be created automatically, we shouldn't create by own.
max()	This action returns the maximum element in the RDD. val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.max() println(result) Output: 5
min()	This action returns the minimum element in the RDD. val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.min() println(result)

	Output: 1
sum()	This action returns the sum of all elements in the RDD. val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.sum() println(result) Output: 15
standardDeviation()	This action returns the standard deviation of the elements in the RDD. val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.stdev() println(result)
countByKey()	This action is specific to Pair RDDs (RDDs containing key-value pairs). It counts the number of occurrences of each key and returns the result as a Map. val rdd = sc.parallelize(Seq(("a", 1), ("b", 2), ("a", 3), ("c", 4), ("b", 5))) val result = rdd.countByKey() println(result) Output: Map(a -> 2, b -> 2, c -> 1)

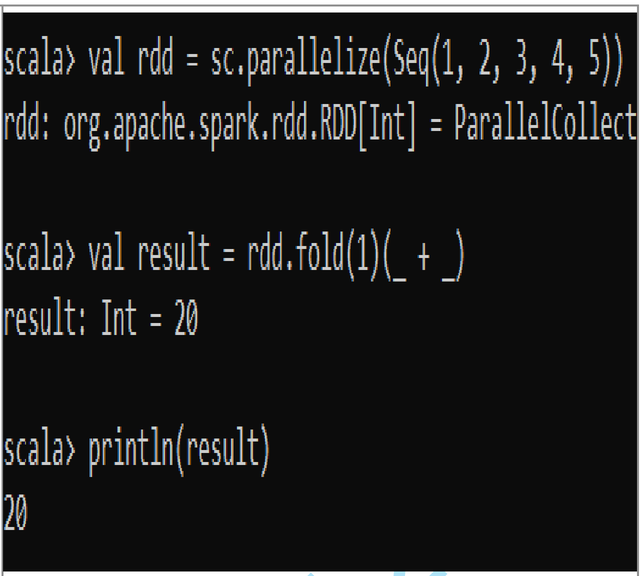
Difference between Reduce and Fold in Apache Spark

Reduce	**Fold**
reduce takes a binary function as input and repeatedly applies it to pairs of elements in the RDD until only a single value remains.	fold also takes a binary function as input and repeatedly applies it to pairs of elements in the RDD until only a single value remains, similar to reduce.
It combines elements pairwise to produce a single result.	However, fold additionally **requires an initial value**, which serves as the starting value for the aggregation.
The result of reduce is the same type as the elements in the RDD.	It combines elements pairwise with the initial value to produce a single result.
It doesn't require an initial value because it uses the first element of the RDD as the initial value for the first invocation of the binary function.	The result of fold can be a different type than the elements in the RDD.

 Reduce Partition 1 Partition 2 Partition 3 Partition 4 2 3 4 2 3 4 5 9 val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.reduce(_ + _) println(result) // Output: 15	 Fold Partition 1 Partition 2 Partition 3 Partition 4 2 3 4 1 1 1 1 1 3 4 5 1 2 5 9 14 val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5)) val result = rdd.fold(0)(_ + _) println(result) // Output: 15

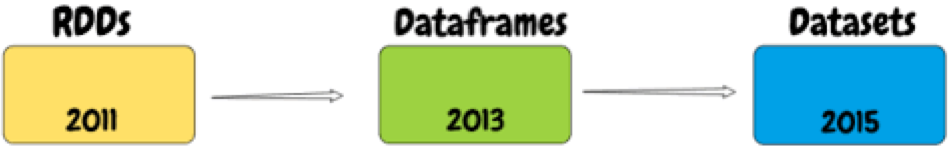


```
val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5))
val result = rdd.reduce(_ + _)
println(result) // Output: 15

// Exiting paste mode, now interpreting.

15
rdd: org.apache.spark.rdd.RDD[Int] = ParallelC
result: Int = 15

scala> 
```

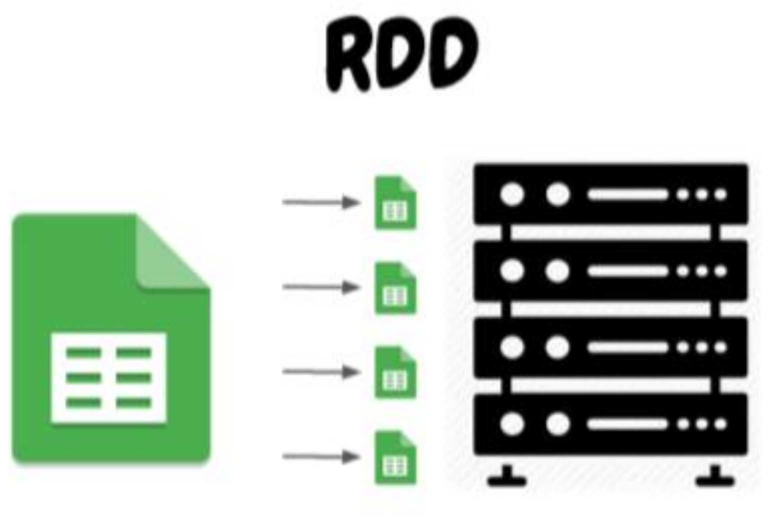


```
scala> val rdd = sc.parallelize(Seq(1, 2, 3, 4, 5))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollect

scala> val result = rdd.fold(1)(_ + _)
result: Int = 20

scala> println(result)
20
```

Difference between RDD, DataFrames, and Datasets



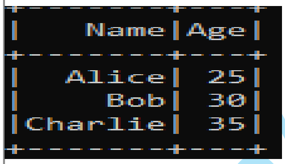
RDD	RDDs or Resilient Distributed Datasets is the fundamental data structure of the Spark. It is the collection of objects which is capable of storing the data partitioned across the multiple nodes of the cluster and also allows them to do processing in parallel. It is fault-tolerant if you perform multiple transformations on the RDD and then due to any reason any node fails. The RDD, in that case, is capable of recovering automatically.

	 • An RDD is like a **box of assorted items**. • Just like a box can contain anything from books to toys, an RDD can contain any type of data, whether structured or unstructured. • We can freely manipulate the contents of the box, moving items around or adding new ones as needed. • However, because the contents are not organized, finding specific items might take some effort.
DataFrames	• DataFrame is a higher-level abstraction built on top of RDDs. • It represents a distributed collection of data organized into named columns, similar to a table in a relational database. • DataFrame provides a more structured and SQL-like interface for data processing. • It uses a **catalyst optimizer for optimization**. • DataFrame API is available in multiple languages like Scala, Java, Python, and R.

	A DataFrame is like a well-organized bookshelf. Each shelf represents a column, and each book represents a row of data. The books are neatly arranged, making it easy to find and analyze information. We can quickly query the bookshelf to find specific books or perform operations on entire rows or columns. Additionally, if we add or remove books, the bookshelf adjusts itself automatically to maintain organization.
Datasets	• DataSet is another higher-level abstraction introduced in Spark 1.6, combining the benefits of RDDs and DataFrames. • It provides the **type-safety and object-oriented programming model of RDDs**, along with the **performance optimizations of DataFrames**. • DataSet allows us to work with strongly typed data structures, providing compile-time type checking and improved developer productivity. • DataSet API is available in Scala and Java, but not in Python or R. A DataSet is like a labelled storage system. Imagine each item in the storage system is labelled with its type (e.g., book, toy, tool). This labelling ensures that each item is handled appropriately and provides clarity on what each item represents. We can confidently interact with the items, knowing their exact type and properties. If we need to perform a specific task, such as finding all books, we can quickly locate and process them based on their labels.

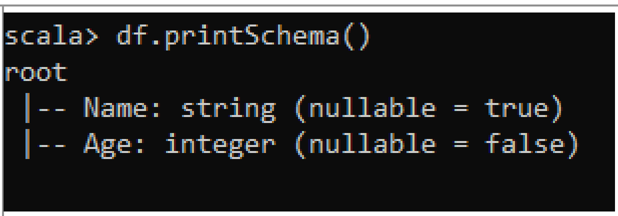
Different Ways to Create DataFrames

import spark.implicits._

From Existing RDD	val rdd = sc.parallelize(Seq(("Alice", 25), ("Bob", 30), ("Charlie", 35))) val df = rdd.**toDF**("Name", "Age") df.show() Output:  +-------+---+ | Name|Age| +-------+---+ | Alice| 25| | Bob| 30| |Charlie| 35| +-------+---+

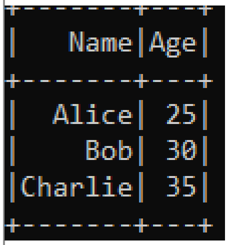
From a Sequence of Tuples	val data = Seq(("Alice", 25), ("Bob", 30), ("Charlie", 35))
	val df = spark.createDataFrame(data).**toDF**("Name", "Age")
	df.show()
	df.printSchema()

<table>
<tr>
<td></td>
<td>



```
scala> df.printSchema()
root
 |-- Name: string (nullable = true)
 |-- Age: integer (nullable = false)
```

Output:



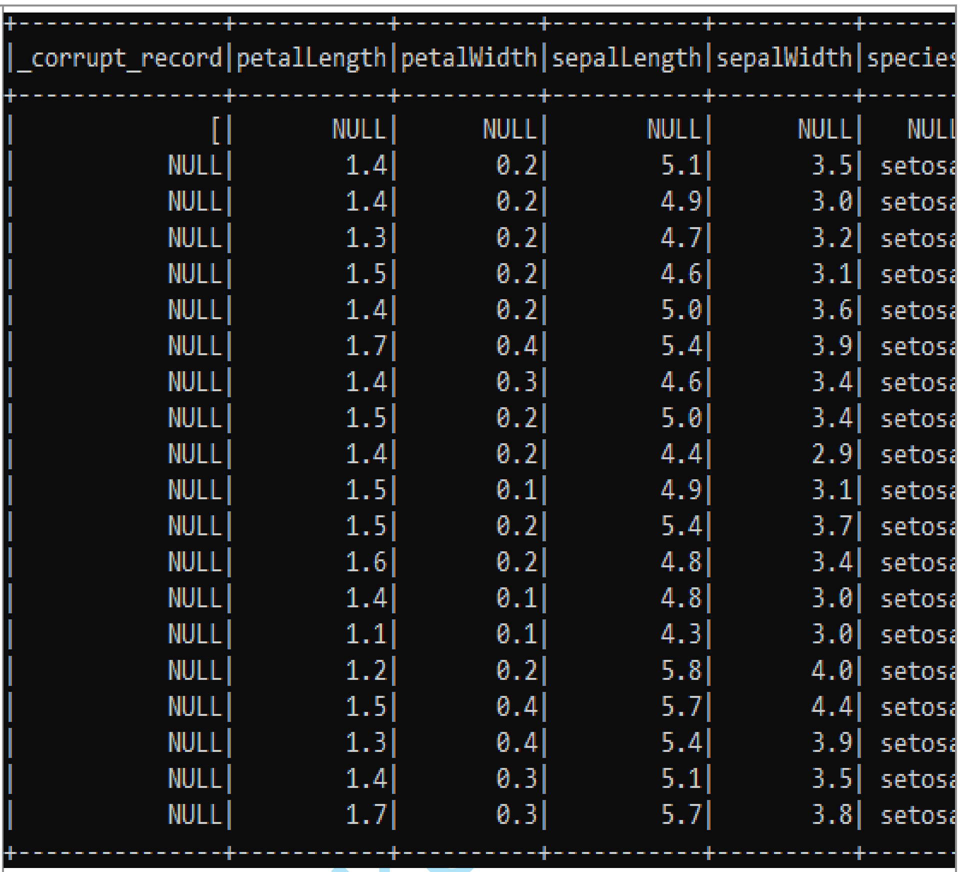
```
+-------+---+
|   Name|Age|
+-------+---+
|  Alice| 25|
|    Bob| 30|
|Charlie| 35|
+-------+---+
```

</td>
</tr>
<tr>
<td><b>From a DataFrameReader (e.g., Reading from a File):</b></td>
<td>

val df4 = spark.read.json("C:/Spark/sparkfiles/iris.json")

df4.show()

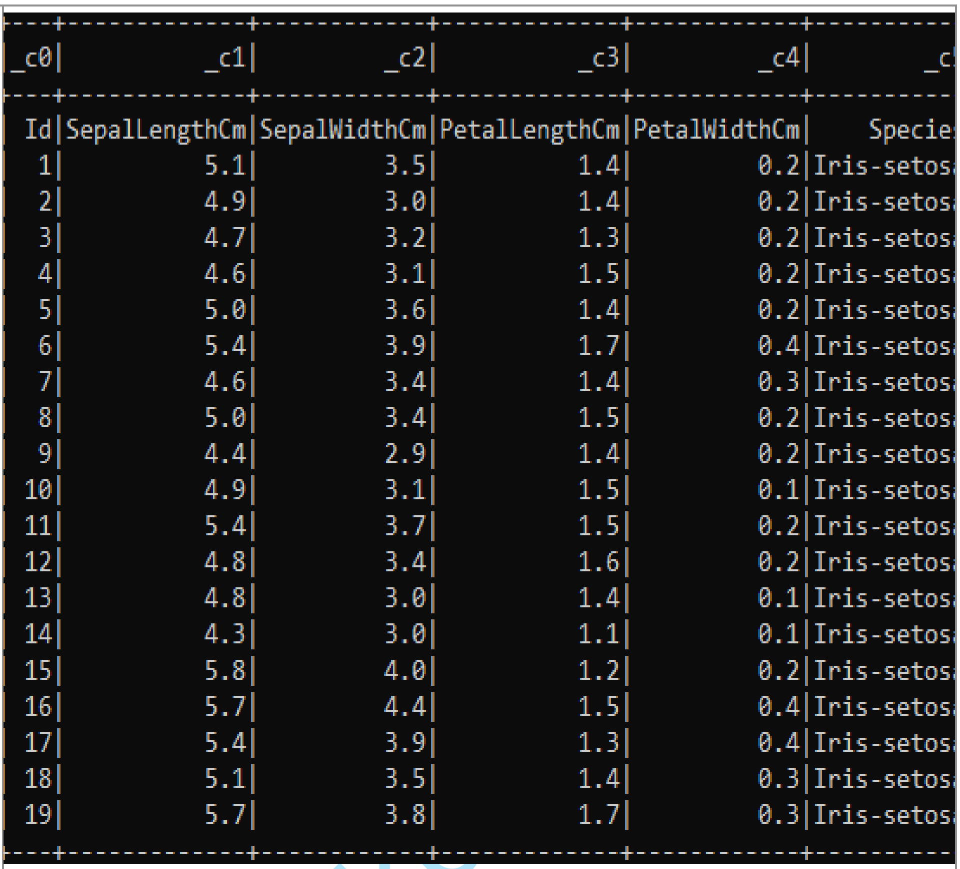
</td>
</tr>
</table>



```
+---------------+-----------+----------+-----------+----------+------
|_corrupt_record|petalLength|petalWidth|sepalLength|sepalWidth|species
+---------------+-----------+----------+-----------+----------+------
|              [|       NULL|      NULL|       NULL|      NULL|   NULL
|           NULL|        1.4|       0.2|        5.1|       3.5| setosa
|           NULL|        1.4|       0.2|        4.9|       3.0| setosa
|           NULL|        1.3|       0.2|        4.7|       3.2| setosa
|           NULL|        1.5|       0.2|        4.6|       3.1| setosa
|           NULL|        1.4|       0.2|        5.0|       3.6| setosa
|           NULL|        1.7|       0.4|        5.4|       3.9| setosa
|           NULL|        1.4|       0.3|        4.6|       3.4| setosa
|           NULL|        1.5|       0.2|        5.0|       3.4| setosa
|           NULL|        1.4|       0.2|        4.4|       2.9| setosa
|           NULL|        1.5|       0.1|        4.9|       3.1| setosa
|           NULL|        1.5|       0.2|        5.4|       3.7| setosa
|           NULL|        1.6|       0.2|        4.8|       3.4| setosa
|           NULL|        1.4|       0.1|        4.8|       3.0| setosa
|           NULL|        1.1|       0.1|        4.3|       3.0| setosa
|           NULL|        1.2|       0.2|        5.8|       4.0| setosa
|           NULL|        1.5|       0.4|        5.7|       4.4| setosa
|           NULL|        1.3|       0.4|        5.4|       3.9| setosa
|           NULL|        1.4|       0.3|        5.1|       3.5| setosa
|           NULL|        1.7|       0.3|        5.7|       3.8| setosa
+---------------+-----------+----------+-----------+----------+------
```

```
val df4 = spark.read.csv("C:/Spark/sparkfiles/iris.csv")
```

```
df4.show()
```



```
---+-------------+------------+-------------+------------+----------
|_c0|          _c1|         _c2|          _c3|         _c4|         _c
---+-------------+------------+-------------+------------+----------
| Id|SepalLengthCm|SepalWidthCm|PetalLengthCm|PetalWidthCm|     Specie
|  1|          5.1|         3.5|          1.4|         0.2|Iris-setos
|  2|          4.9|         3.0|          1.4|         0.2|Iris-setos
|  3|          4.7|         3.2|          1.3|         0.2|Iris-setos
|  4|          4.6|         3.1|          1.5|         0.2|Iris-setos
|  5|          5.0|         3.6|          1.4|         0.2|Iris-setos
|  6|          5.4|         3.9|          1.7|         0.4|Iris-setos
|  7|          4.6|         3.4|          1.4|         0.3|Iris-setos
|  8|          5.0|         3.4|          1.5|         0.2|Iris-setos
|  9|          4.4|         2.9|          1.4|         0.2|Iris-setos
| 10|          4.9|         3.1|          1.5|         0.1|Iris-setos
| 11|          5.4|         3.7|          1.5|         0.2|Iris-setos
| 12|          4.8|         3.4|          1.6|         0.2|Iris-setos
| 13|          4.8|         3.0|          1.4|         0.1|Iris-setos
| 14|          4.3|         3.0|          1.1|         0.1|Iris-setos
| 15|          5.8|         4.0|          1.2|         0.2|Iris-setos
| 16|          5.7|         4.4|          1.5|         0.4|Iris-setos
| 17|          5.4|         3.9|          1.3|         0.4|Iris-setos
| 18|          5.1|         3.5|          1.4|         0.3|Iris-setos
| 19|          5.7|         3.8|          1.7|         0.3|Iris-setos
---+-------------+------------+-------------+------------+----------
```

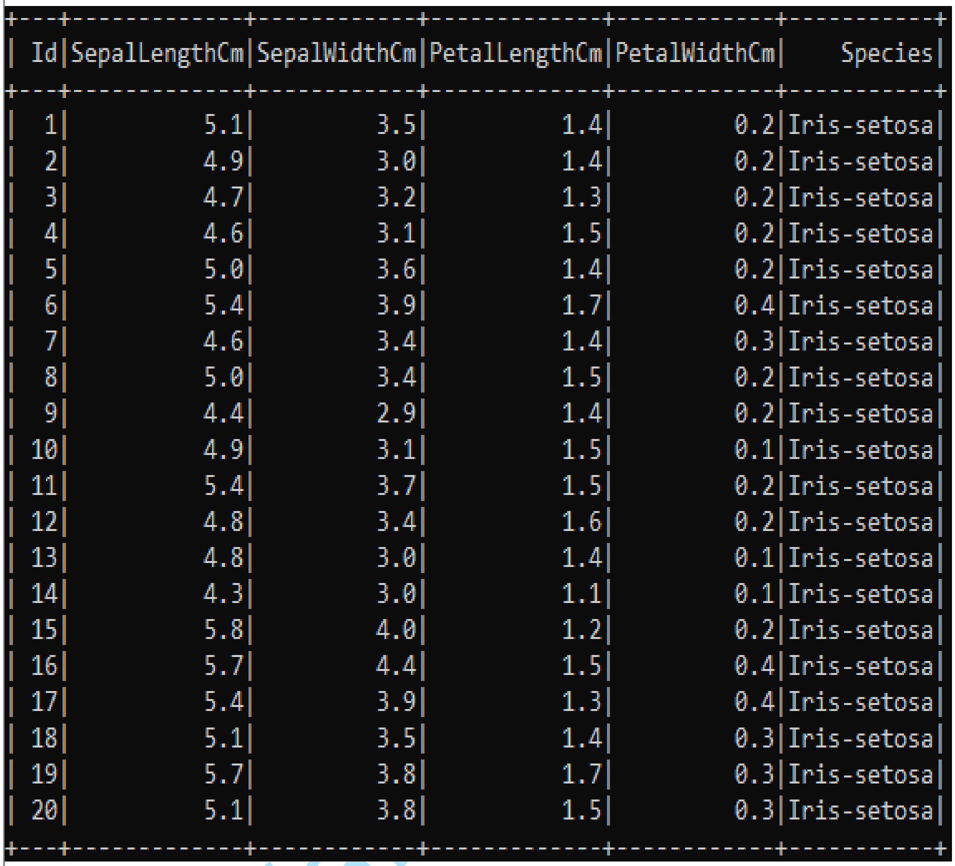
val df4 = spark.read.text("C:/Spark/sparkfiles/iris.txt")

df4.show()

When reading a CSV file with Spark, we can specify whether the file has a header or not using the option method with the header parameter set to true or false. If the file has a header, setting header to true will instruct Spark to use the first line of the file as the header and skip it when reading the data.

val df = spark.read.option("header", "true").csv("C:/Spark/sparkfiles/iris.csv")

val df = spark.read.option("header", "true").option("inferSchema", "true").csv("C:/Spark/sparkfiles/iris.csv")



```
+---+-------------+------------+-------------+------------+-----------+
| Id|SepalLengthCm|SepalWidthCm|PetalLengthCm|PetalWidthCm|    Species|
+---+-------------+------------+-------------+------------+-----------+
|  1|          5.1|         3.5|          1.4|         0.2|Iris-setosa|
|  2|          4.9|         3.0|          1.4|         0.2|Iris-setosa|
|  3|          4.7|         3.2|          1.3|         0.2|Iris-setosa|
|  4|          4.6|         3.1|          1.5|         0.2|Iris-setosa|
|  5|          5.0|         3.6|          1.4|         0.2|Iris-setosa|
|  6|          5.4|         3.9|          1.7|         0.4|Iris-setosa|
|  7|          4.6|         3.4|          1.4|         0.3|Iris-setosa|
|  8|          5.0|         3.4|          1.5|         0.2|Iris-setosa|
|  9|          4.4|         2.9|          1.4|         0.2|Iris-setosa|
| 10|          4.9|         3.1|          1.5|         0.1|Iris-setosa|
| 11|          5.4|         3.7|          1.5|         0.2|Iris-setosa|
| 12|          4.8|         3.4|          1.6|         0.2|Iris-setosa|
| 13|          4.8|         3.0|          1.4|         0.1|Iris-setosa|
| 14|          4.3|         3.0|          1.1|         0.1|Iris-setosa|
| 15|          5.8|         4.0|          1.2|         0.2|Iris-setosa|
| 16|          5.7|         4.4|          1.5|         0.4|Iris-setosa|
| 17|          5.4|         3.9|          1.3|         0.4|Iris-setosa|
| 18|          5.1|         3.5|          1.4|         0.3|Iris-setosa|
| 19|          5.7|         3.8|          1.7|         0.3|Iris-setosa|
| 20|          5.1|         3.8|          1.5|         0.3|Iris-setosa|
+---+-------------+------------+-------------+------------+-----------+
```

DataFrames API Queries

```
import org.apache.spark.sql.SparkSession


Create SparkSession
val spark = SparkSession.builder()
  .appName("DataFrame API Example")
  .master("local[*]")
  .getOrCreate()


Synthetic data
```

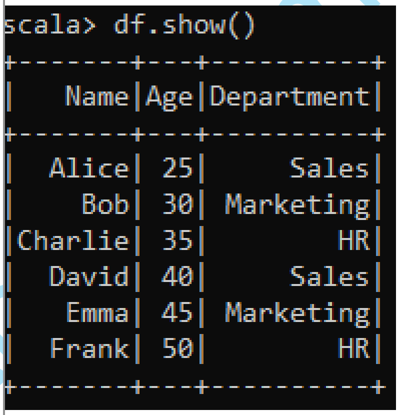
```
val data = Seq(
  ("Alice", 25, "Sales"),
  ("Bob", 30, "Marketing"),
  ("Charlie", 35, "HR"),
  ("David", 40, "Sales"),
  ("Emma", 45, "Marketing"),
  ("Frank", 50, "HR")
)
```

Create DataFrame

```
val df = spark.createDataFrame(data).toDF("Name", "Age", "Department")
```

Show DataFrame

```
df.show()
```



```
scala> df.show()
+-------+---+----------+
|   Name|Age|Department|
+-------+---+----------+
|  Alice| 25|     Sales|
|    Bob| 30| Marketing|
|Charlie| 35|        HR|
|  David| 40|     Sales|
|   Emma| 45| Marketing|
|  Frank| 50|        HR|
+-------+---+----------+
```

```
import org.apache.spark.sql.{SparkSession, DataFrame}
```

<table>
<tr>
<td><b>Select Columns</b></td>
<td>
Select specific columns

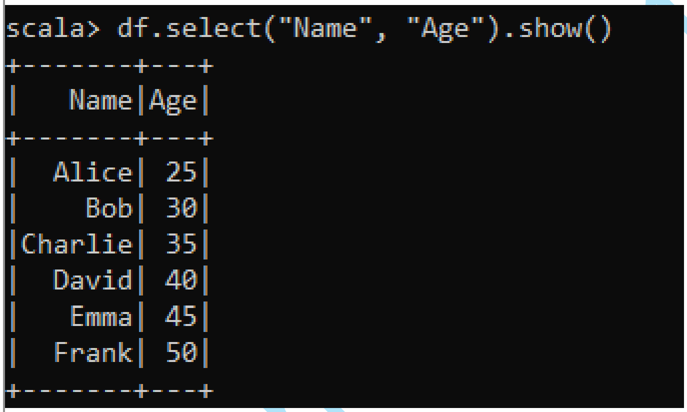
df.select("col1", "col2")

Select all columns

df.select("*")

<b>Example-</b>

df.select("Name", "Age").show()



<pre>
scala> df.select("Name", "Age").show()
+-------+---+
| Name|Age|
+-------+---+
| Alice| 25|
| Bob| 30|
|Charlie| 35|
David	40
Emma	45
Frank	50
+-------+---+
</pre>
</td>
</tr>
<tr>
<td><b>Filter Rows</b></td>
<td>
Filter rows based on a condition

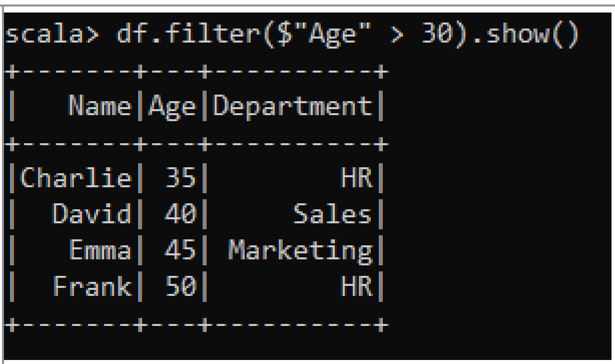
df.filter($"col1" > 10)

Filter rows using SQL expression

df.filter("col1 > 10")

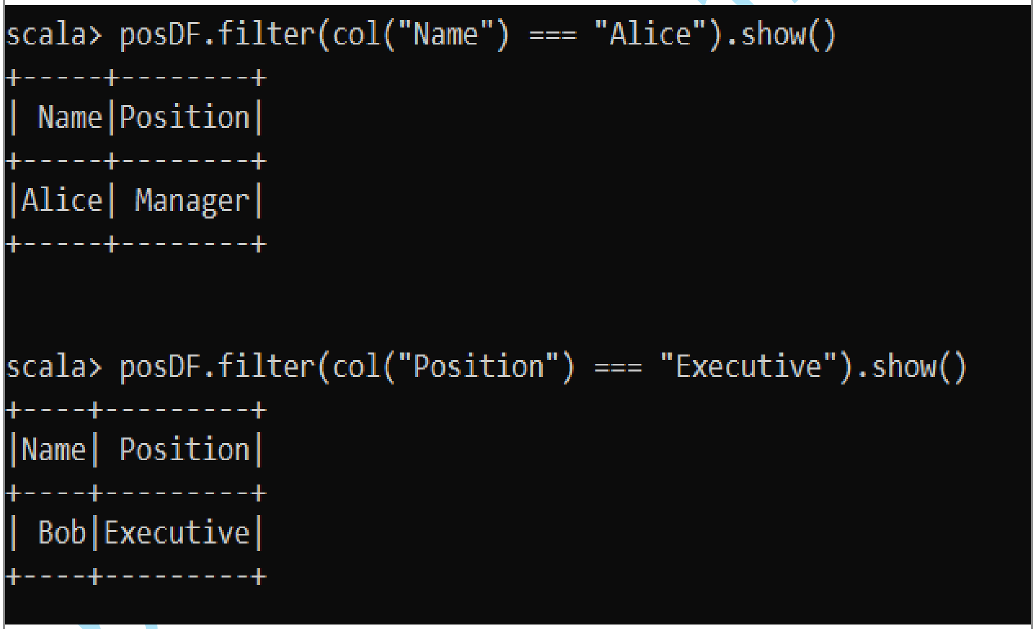
<b>Example-</b>

df.filter($"Age" > 30).show()
</td>
</tr>
</table>



```
scala> df.filter($"Age" > 30).show()
+-------+---+----------+
|   Name|Age|Department|
+-------+---+----------+
|Charlie| 35|        HR|
|  David| 40|     Sales|
|   Emma| 45| Marketing|
|  Frank| 50|        HR|
+-------+---+----------+
```

Using Column Expression: We can construct the condition using column expressions. This involves using the col function to refer to the column:



```
scala> posDF.filter(col("Name") === "Alice").show()
+-----+--------+
| Name|Position|
+-----+--------+
|Alice| Manager|
+-----+--------+


scala> posDF.filter(col("Position") === "Executive").show()
+----+---------+
|Name| Position|
+----+---------+
| Bob|Executive|
+----+---------+
```

Group By and Aggregate

Group by a column and count the occurrences

df.groupBy("col1").count()

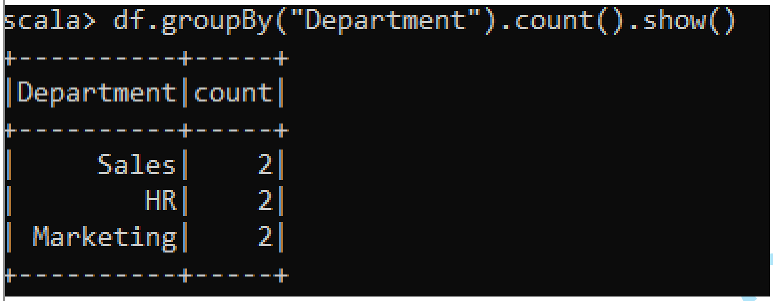
Aggregate with custom aggregation functions

import org.apache.spark.sql.functions._

df.groupBy("col1").agg(avg("col2"), sum("col3"))

Example-

df.groupBy("Department").count().show()



```
scala> df.groupBy("Department").count().show()
+----------+-----+
|Department|count|
+----------+-----+
|     Sales|    2|
|        HR|    2|
| Marketing|    2|
+----------+-----+
```

Sort Data

Sort data by one or more columns

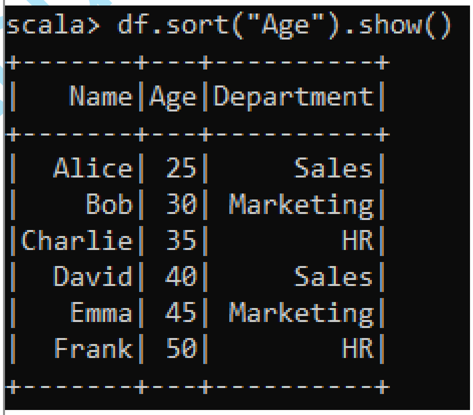
df.sort("col1")

Sort data in descending order

df.sort(desc("col1"))

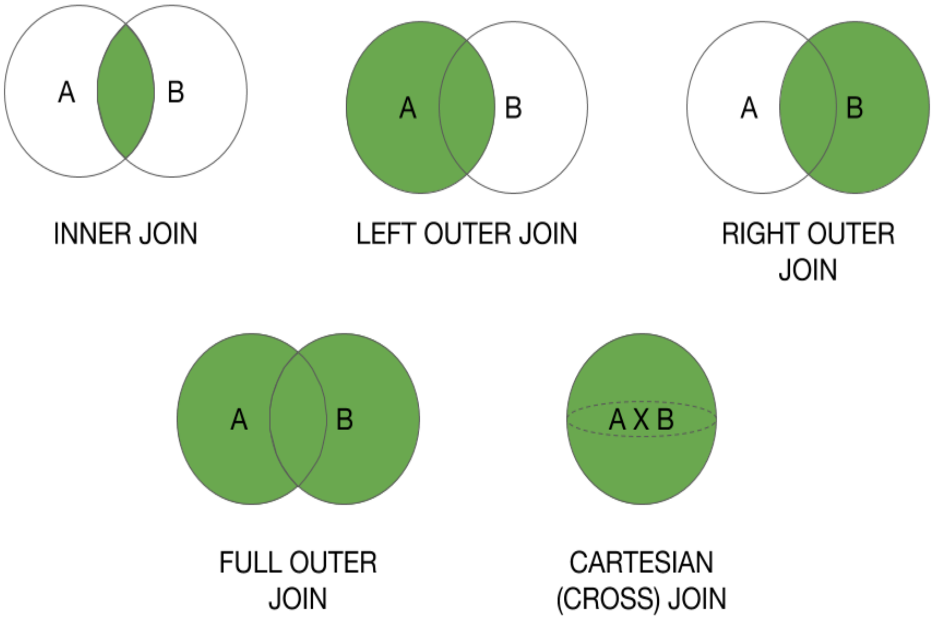
Example-

df.sort("Age").show()



```
scala> df.sort("Age").show()
+-------+---+----------+
|   Name|Age|Department|
+-------+---+----------+
|  Alice| 25|     Sales|
|    Bob| 30| Marketing|
|Charlie| 35|        HR|
|  David| 40|     Sales|
|   Emma| 45| Marketing|
|  Frank| 50|        HR|
+-------+---+----------+
```

Join DataFrames



Inner join with another DataFrame

```
df.join(otherDf, df("col1") === otherDf("col1"), "inner")
```

Left outer join

```
df.join(otherDf, Seq("col1"), "left_outer")
```

```
import org.apache.spark.sql.{SparkSession, DataFrame}
```

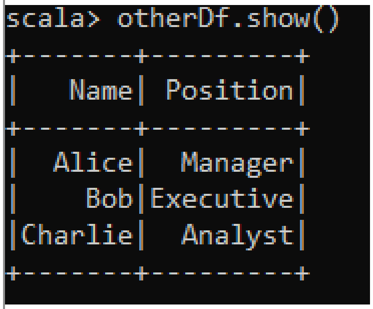
Example-

```
val otherData = Seq(
  ("Alice", "Manager"),
  ("Bob", "Executive"),
  ("Charlie", "Analyst")
```

)

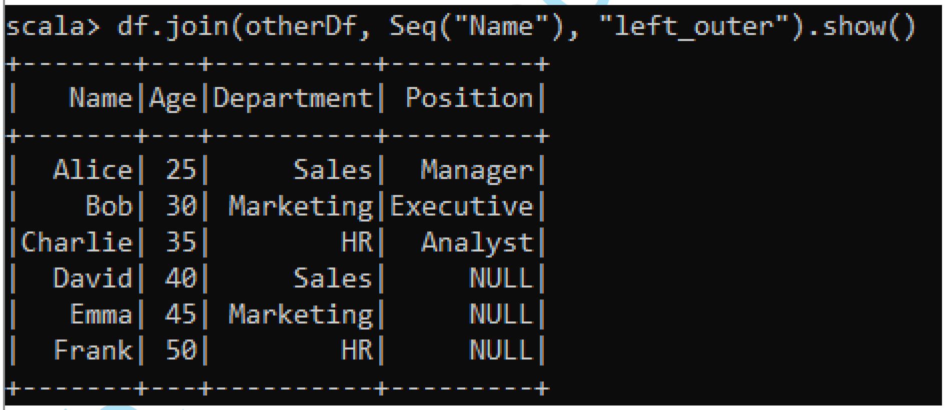
val otherDf = spark.createDataFrame(otherData).toDF("Name", "Position")

otherDf.show()



```
scala> otherDf.show()
+-------+---------+
|   Name| Position|
+-------+---------+
|  Alice|  Manager|
|    Bob|Executive|
|Charlie|  Analyst|
+-------+---------+
```

df.join(otherDf, Seq("Name"), "left_outer").show()



```
scala> df.join(otherDf, Seq("Name"), "left_outer").show()
+-------+---+----------+---------+
|   Name|Age|Department| Position|
+-------+---+----------+---------+
|  Alice| 25|     Sales|  Manager|
|    Bob| 30| Marketing|Executive|
|Charlie| 35|        HR|  Analyst|
|  David| 40|     Sales|     NULL|
|   Emma| 45| Marketing|     NULL|
|  Frank| 50|        HR|     NULL|
+-------+---+----------+---------+
```

Union DataFrames

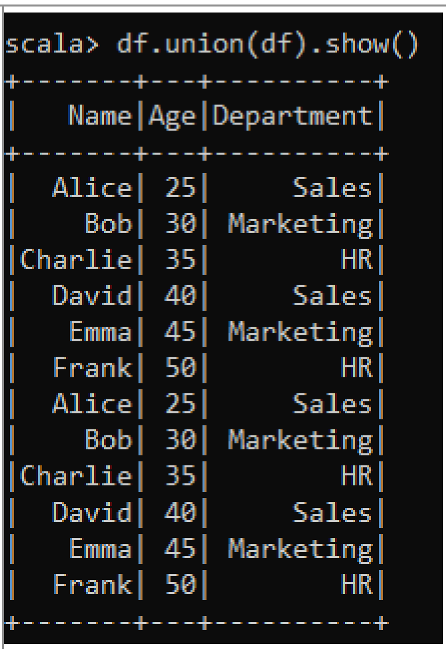
Union two DataFrames

df.union(otherDf)

Example-

df.union(df).show()

<table>
<tr><td></td><td>



```
scala> df.union(df).show()
+-------+---+----------+
|   Name|Age|Department|
+-------+---+----------+
|  Alice| 25|     Sales|
|    Bob| 30| Marketing|
|Charlie| 35|        HR|
|  David| 40|     Sales|
|   Emma| 45| Marketing|
|  Frank| 50|        HR|
|  Alice| 25|     Sales|
|    Bob| 30| Marketing|
|Charlie| 35|        HR|
|  David| 40|     Sales|
|   Emma| 45| Marketing|
|  Frank| 50|        HR|
+-------+---+----------+
```

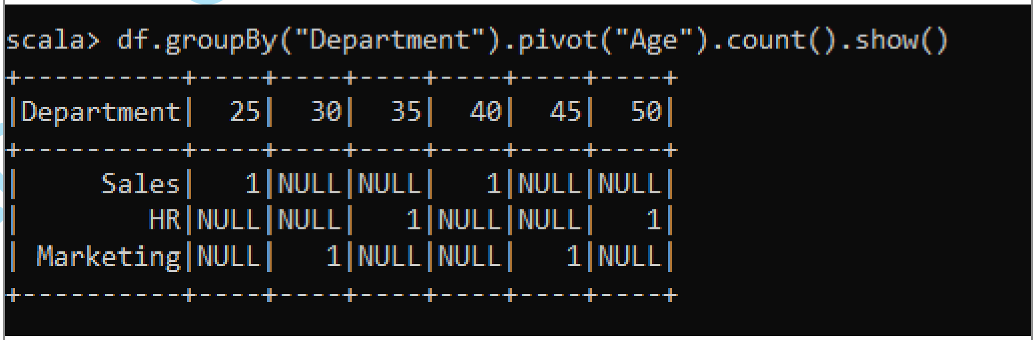
</td></tr>
<tr><td><b>Pivot</b></td><td>

Pivot DataFrame based on a column

df.groupBy("col1").pivot("col2").agg(sum("col3"))

Example-

df.groupBy("Department").pivot("Age").count().show()



```
scala> df.groupBy("Department").pivot("Age").count().show()
+----------+----+----+----+----+----+----+
|Department|  25|  30|  35|  40|  45|  50|
+----------+----+----+----+----+----+----+
|     Sales|   1|NULL|NULL|   1|NULL|NULL|
|        HR|NULL|NULL|   1|NULL|NULL|   1|
| Marketing|NULL|   1|NULL|NULL|   1|NULL|
+----------+----+----+----+----+----+----+
```

</td></tr>
<tr><td><b>Window Functions</b></td><td>

Calculate row number over a window

import org.apache.spark.sql.expressions.Window

</td></tr>
</table>

val windowSpec = Window.orderBy("col1")

df.withColumn("row_number", row_number().over(windowSpec))

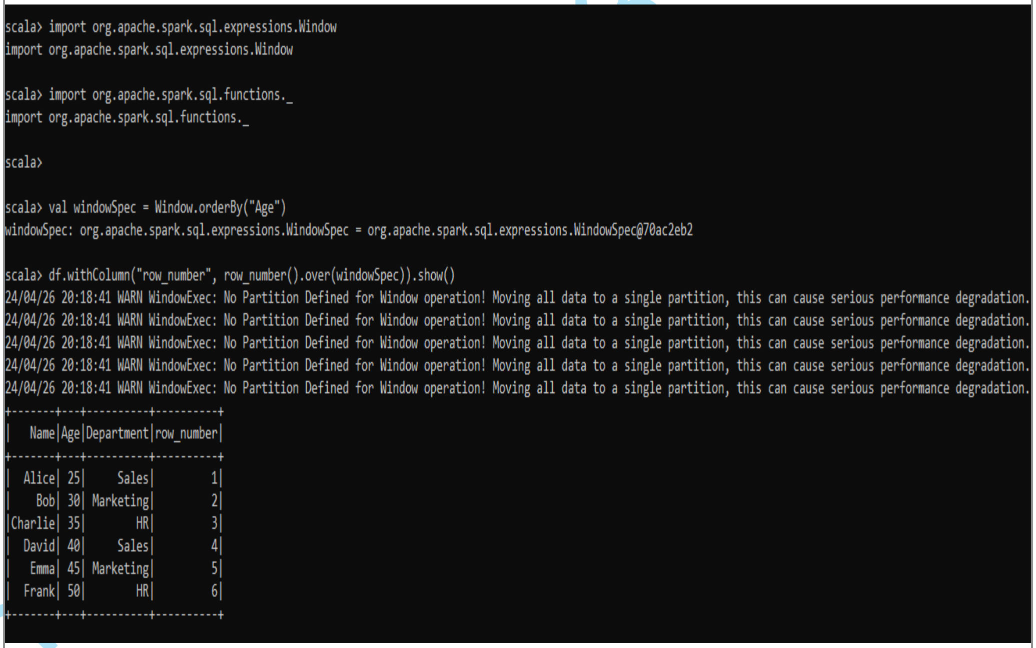
Example-

import org.apache.spark.sql.expressions.Window

import org.apache.spark.sql.functions._

val windowSpec = Window.orderBy("Age")

df.withColumn("row_number", row_number().over(windowSpec)).show()



```
scala> import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.expressions.Window

scala> import org.apache.spark.sql.functions._
import org.apache.spark.sql.functions._

scala>

scala> val windowSpec = Window.orderBy("Age")
windowSpec: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@70ac2eb2

scala> df.withColumn("row_number", row_number().over(windowSpec)).show()
24/04/26 20:18:41 WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can cause serious performance degradation.
24/04/26 20:18:41 WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can cause serious performance degradation.
24/04/26 20:18:41 WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can cause serious performance degradation.
24/04/26 20:18:41 WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can cause serious performance degradation.
24/04/26 20:18:41 WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition, this can cause serious performance degradation.
+-------+---+----------+----------+
|   Name|Age|Department|row_number|
+-------+---+----------+----------+
|  Alice| 25|     Sales|         1|
|    Bob| 30| Marketing|         2|
|Charlie| 35|        HR|         3|
|  David| 40|     Sales|         4|
|   Emma| 45| Marketing|         5|
|  Frank| 50|        HR|         6|
+-------+---+----------+----------+
```

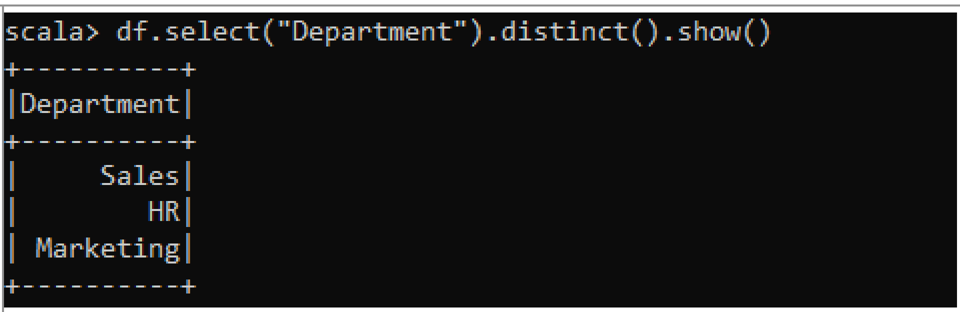
Distinct Values

Get distinct values of a column

df.select("col1").distinct()

Example-

df.select("Department").distinct().show()



```
scala> df.select("Department").distinct().show()
+----------+
|Department|
+----------+
|     Sales|
|        HR|
| Marketing|
+----------+
```

Show Data

Display first n rows

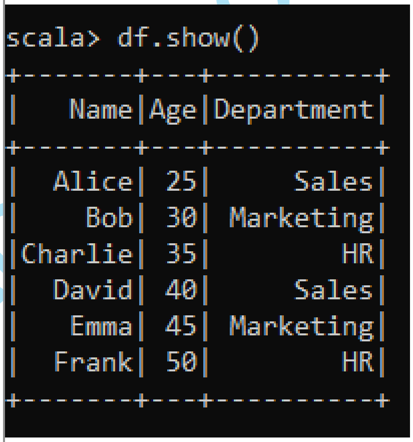
df.show()

Display only selected columns

df.select("col1", "col2").show()

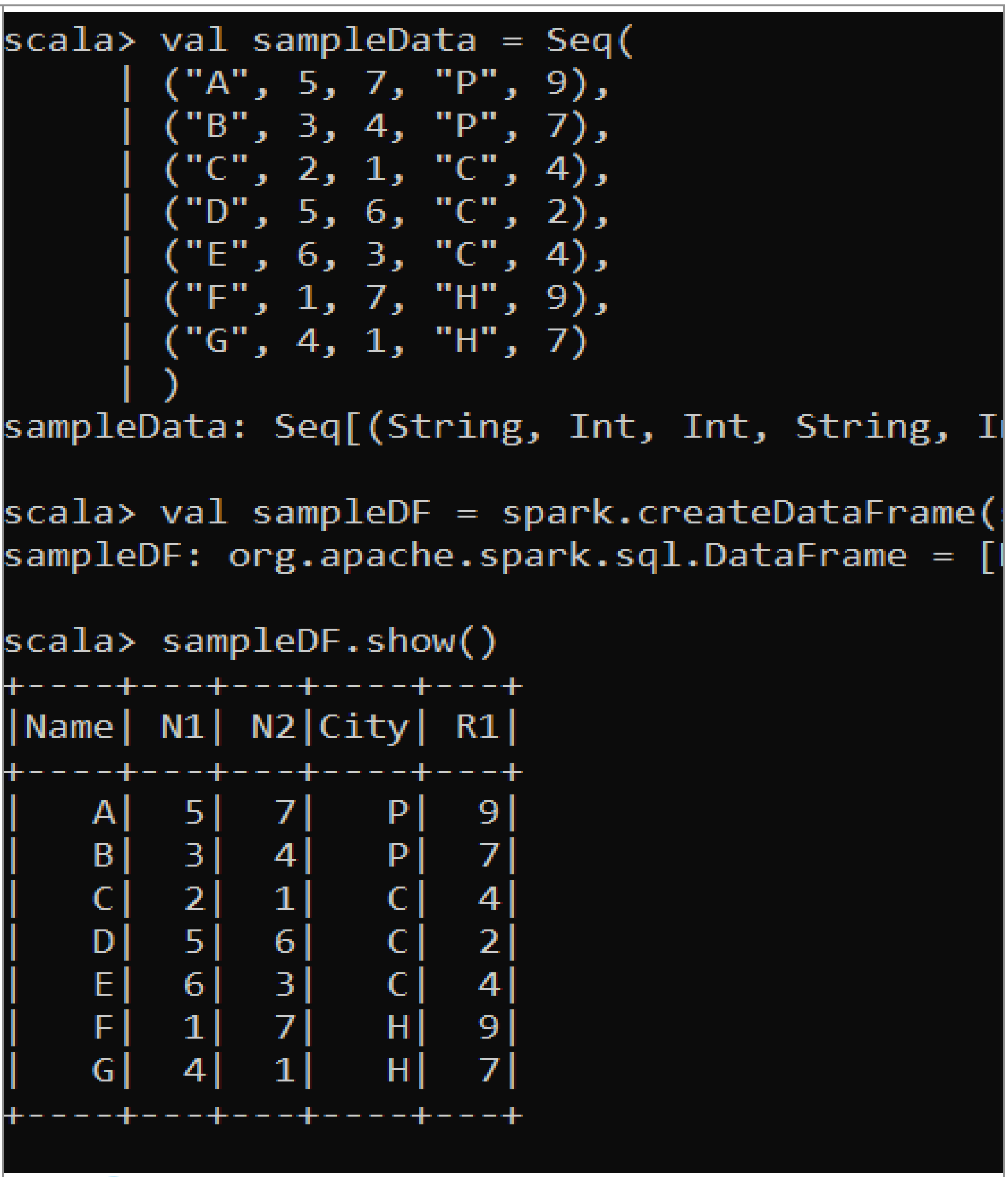
Example-

df.show()

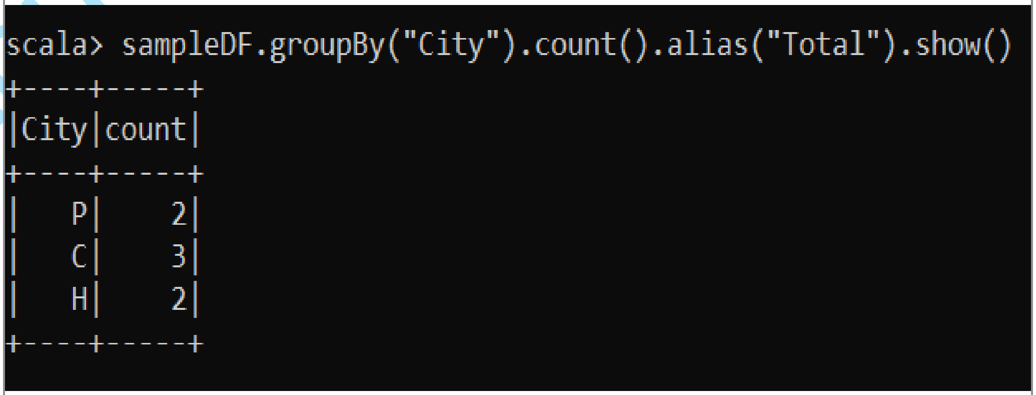


```
scala> df.show()
+-------+---+----------+
|   Name|Age|Department|
+-------+---+----------+
|  Alice| 25|     Sales|
|    Bob| 30| Marketing|
|Charlie| 35|        HR|
|  David| 40|     Sales|
|   Emma| 45| Marketing|
|  Frank| 50|        HR|
+-------+---+----------+
```

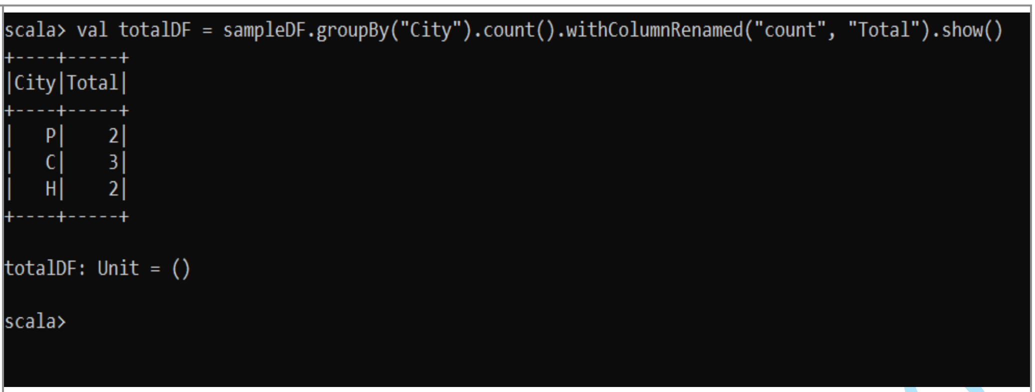
Column Rename



```
scala> val sampleData = Seq(
     | ("A", 5, 7, "P", 9),
     | ("B", 3, 4, "P", 7),
     | ("C", 2, 1, "C", 4),
     | ("D", 5, 6, "C", 2),
     | ("E", 6, 3, "C", 4),
     | ("F", 1, 7, "H", 9),
     | ("G", 4, 1, "H", 7)
     | )
sampleData: Seq[(String, Int, Int, String, I

scala> val sampleDF = spark.createDataFrame(
sampleDF: org.apache.spark.sql.DataFrame = [

scala> sampleDF.show()
+----+---+---+----+---+
|Name| N1| N2|City| R1|
+----+---+---+----+---+
|   A|  5|  7|   P|  9|
|   B|  3|  4|   P|  7|
|   C|  2|  1|   C|  4|
|   D|  5|  6|   C|  2|
|   E|  6|  3|   C|  4|
|   F|  1|  7|   H|  9|
|   G|  4|  1|   H|  7|
+----+---+---+----+---+
```



```
scala> sampleDF.groupBy("City").count().alias("Total").show()
+----+-----+
|City|count|
+----+-----+
|   P|    2|
|   C|    3|
|   H|    2|
+----+-----+
```

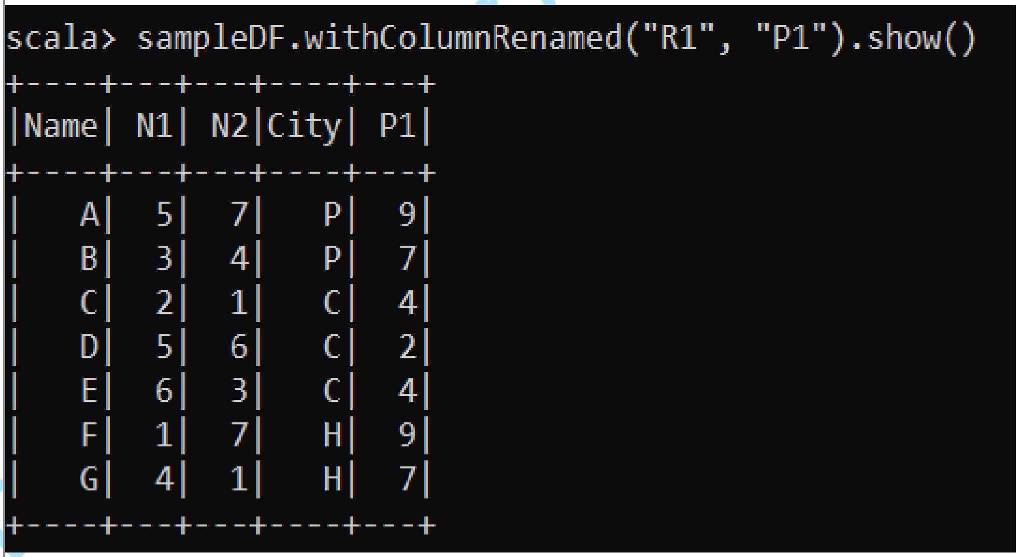


```
scala> val totalDF = sampleDF.groupBy("City").count().withColumnRenamed("count", "Total").show()
+----+-----+
|City|Total|
+----+-----+
|   P|    2|
|   C|    3|
|   H|    2|
+----+-----+

totalDF: Unit = ()

scala>
```

val totalDF = sampleDF.groupBy("City").count().**withColumnRenamed**("count", "Total")

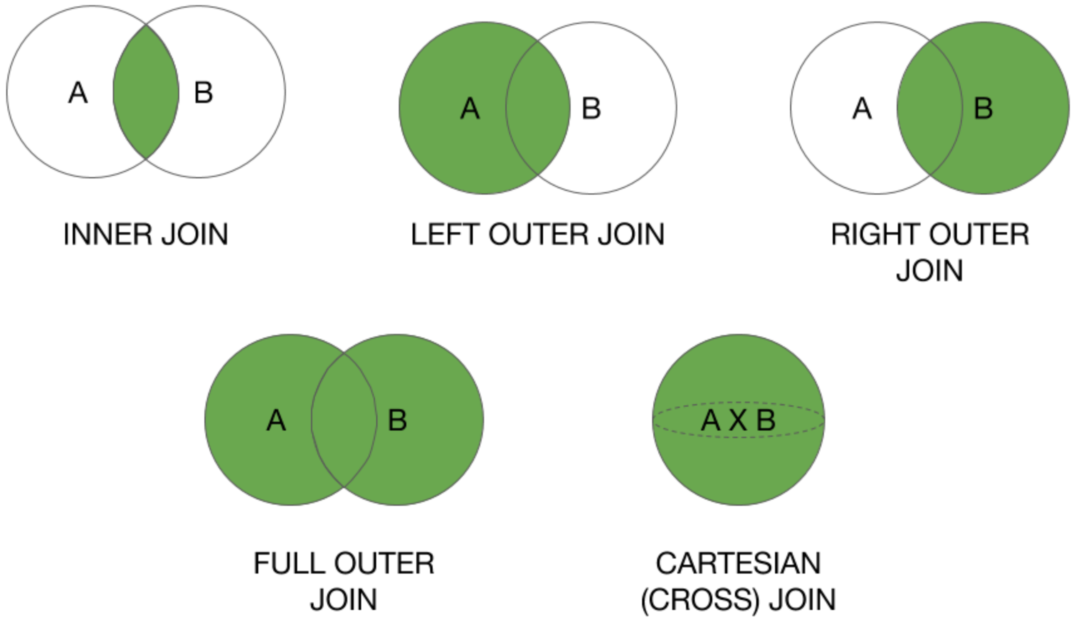
totalDF.show()

sampleDF.withColumnRenamed("R1", "P1").show()



```
scala> sampleDF.withColumnRenamed("R1", "P1").show()
+----+---+---+----+---+
|Name| N1| N2|City| P1|
+----+---+---+----+---+
|   A|  5|  7|   P|  9|
|   B|  3|  4|   P|  7|
|   C|  2|  1|   C|  4|
|   D|  5|  6|   C|  2|
|   E|  6|  3|   C|  4|
|   F|  1|  7|   H|  9|
|   G|  4|  1|   H|  7|
+----+---+---+----+---+
```

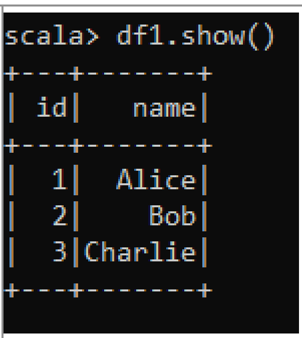
Joins



Before Running Queries Import:

import org.apache.spark.sql.{SparkSession, DataFrame}

	Sample data for DataFrame 1 val data1 = Seq((1, "Alice"), (2, "Bob"), (3, "Charlie")) Create DataFrame 1 val df1 = spark.createDataFrame(data1).toDF("id", "name")



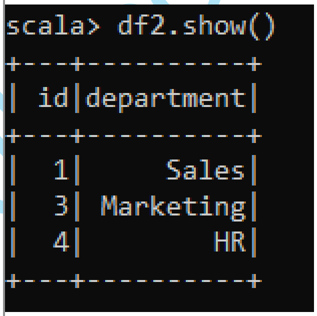
```
scala> df1.show()
+---+-------+
| id|   name|
+---+-------+
|  1|  Alice|
|  2|    Bob|
|  3|Charlie|
+---+-------+
```

Sample data for DataFrame 2

```
val data2 = Seq(
  (1, "Sales"),
  (3, "Marketing"),
  (4, "HR")
)
```

Create DataFrame 2

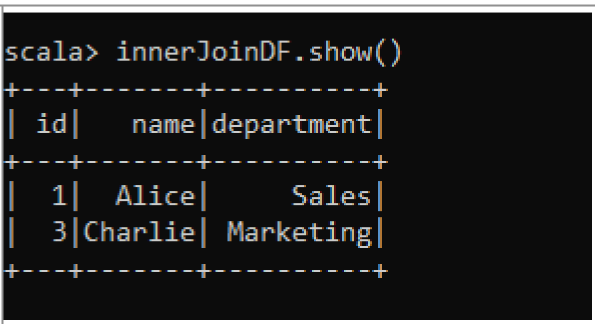
```
val df2 = spark.createDataFrame(data2).toDF("id", "department")
```



```
scala> df2.show()
+---+----------+
| id|department|
+---+----------+
|  1|     Sales|
|  3| Marketing|
|  4|        HR|
+---+----------+
```

Inner Join

```
val innerJoinDF: DataFrame = df1.join(df2, "id", "inner")
innerJoinDF.show()
```

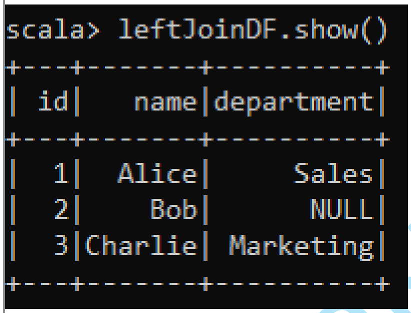


```
scala> innerJoinDF.show()
+---+-------+----------+
| id|   name|department|
+---+-------+----------+
|  1|  Alice|     Sales|
|  3|Charlie| Marketing|
+---+-------+----------+
```

Left Outer Join

val leftJoinDF: DataFrame = df1.join(df2, Seq("id"), "left_outer")

leftJoinDF.show()

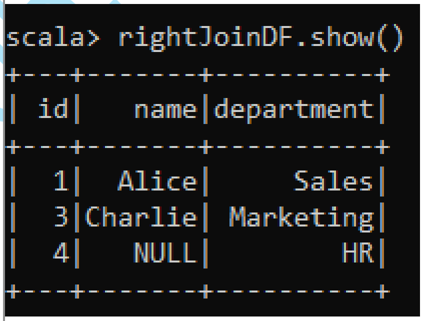


```
scala> leftJoinDF.show()
+---+-------+----------+
| id|   name|department|
+---+-------+----------+
|  1|  Alice|     Sales|
|  2|    Bob|      NULL|
|  3|Charlie| Marketing|
+---+-------+----------+
```

Right Outer Join

val rightJoinDF: DataFrame = df1.join(df2, Seq("id"), "right_outer")

rightJoinDF.show()



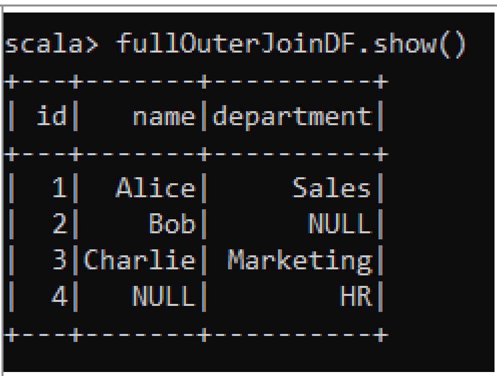
```
scala> rightJoinDF.show()
+---+-------+----------+
| id|   name|department|
+---+-------+----------+
|  1|  Alice|     Sales|
|  3|Charlie| Marketing|
|  4|   NULL|        HR|
+---+-------+----------+
```

Full Outer Join

val fullOuterJoinDF: DataFrame = df1.join(df2, Seq("id"), "outer")

fullOuterJoinDF.show()

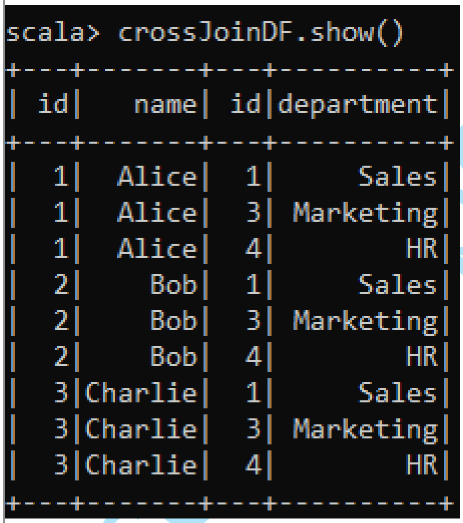
<table>
<tr>
<td></td>
<td>



```
scala> fullOuterJoinDF.show()
+---+-------+----------+
| id|   name|department|
+---+-------+----------+
|  1|  Alice|     Sales|
|  2|    Bob|      NULL|
|  3|Charlie| Marketing|
|  4|   NULL|        HR|
+---+-------+----------+
```

</td>
</tr>
<tr>
<td><b>Inner Join</b></td>
<td>

val crossJoinDF: DataFrame = df1.crossJoin(df2)
crossJoinDF.show()



```
scala> crossJoinDF.show()
+---+-------+---+----------+
| id|   name| id|department|
+---+-------+---+----------+
|  1|  Alice|  1|     Sales|
|  1|  Alice|  3| Marketing|
|  1|  Alice|  4|        HR|
|  2|    Bob|  1|     Sales|
|  2|    Bob|  3| Marketing|
|  2|    Bob|  4|        HR|
|  3|Charlie|  1|     Sales|
|  3|Charlie|  3| Marketing|
|  3|Charlie|  4|        HR|
+---+-------+---+----------+
```

</td>
</tr>
</table>

Running SQL Queries Using Spark SQL

Querying structured data using Spark SQL allows you to leverage SQL queries to interact with DataFrames and perform various operations.

import org.apache.spark.sql.SparkSession

Sample data for DataFrame

```
val data = Seq(
  ("Alice", 25, "Sales", 50000),
  ("Bob", 30, "Marketing", 60000),
  ("Charlie", 35, "HR", 55000),
  ("David", 40, "Sales", 52000),
  ("Emma", 45, "Marketing", 62000),
  ("Frank", 50, "HR", 58000)
)
```

Create DataFrame

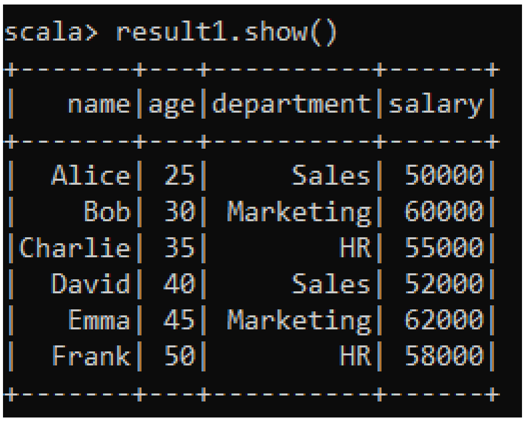
```
val df = spark.createDataFrame(data).toDF("name", "age", "department", "salary")
```

Register DataFrame as a temporary table

```
df.createOrReplaceTempView("employee")
```

Select all Columns

```
val result1 = spark.sql("SELECT * FROM employee")
result1.show()
```

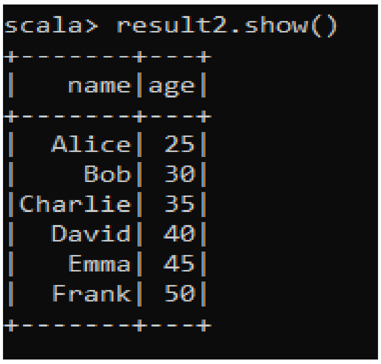


```
scala> result1.show()
+-------+---+----------+------+
|   name|age|department|salary|
+-------+---+----------+------+
|  Alice| 25|     Sales| 50000|
|    Bob| 30| Marketing| 60000|
|Charlie| 35|        HR| 55000|
|  David| 40|     Sales| 52000|
|   Emma| 45| Marketing| 62000|
|  Frank| 50|        HR| 58000|
+-------+---+----------+------+
```

Select Specific Columns

val result2 = spark.sql("SELECT name, age FROM employee")

result2.show()

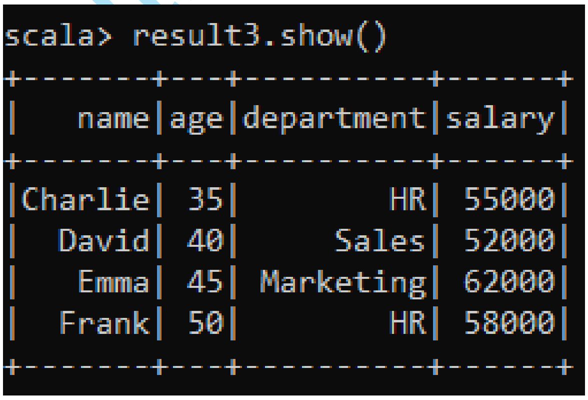


```
scala> result2.show()
+-------+---+
|   name|age|
+-------+---+
|  Alice| 25|
|    Bob| 30|
|Charlie| 35|
|  David| 40|
|   Emma| 45|
|  Frank| 50|
+-------+---+
```

Filter Rows With WHERE

val result3 = spark.sql("SELECT * FROM employee WHERE age > 30")

result3.show()

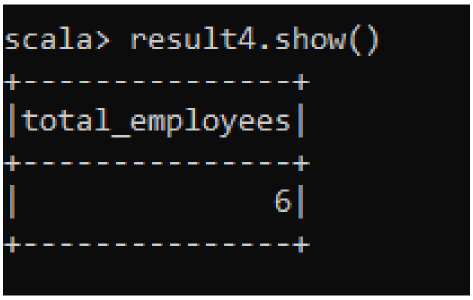


```
scala> result3.show()
+-------+---+----------+------+
|   name|age|department|salary|
+-------+---+----------+------+
|Charlie| 35|        HR| 55000|
|  David| 40|     Sales| 52000|
|   Emma| 45| Marketing| 62000|
|  Frank| 50|        HR| 58000|
+-------+---+----------+------+
```

Count Rows

val result4 = spark.sql("SELECT COUNT(*) AS total_employees FROM employee")

result4.show()

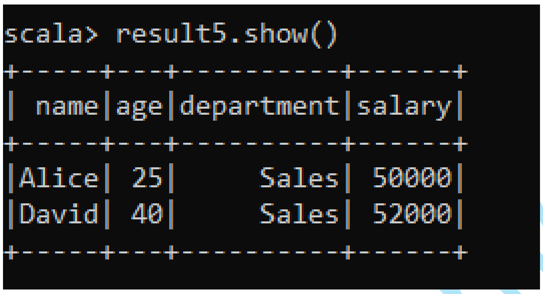


```
scala> result4.show()
+---------------+
|total_employees|
+---------------+
|              6|
+---------------+
```

Using LIKE Operator

val result5 = spark.sql("SELECT * FROM employee WHERE department LIKE 'Sa%'")

result5.show()

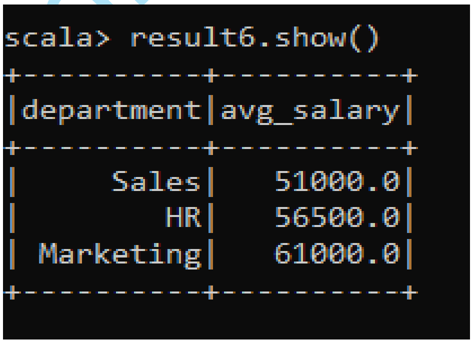


```
scala> result5.show()
+-----+---+----------+------+
| name|age|department|salary|
+-----+---+----------+------+
|Alice| 25|     Sales| 50000|
|David| 40|     Sales| 52000|
+-----+---+----------+------+
```

Group By with Aggregate Functions

val result6 = spark.sql("SELECT department, AVG(salary) AS avg_salary FROM employee GROUP BY department")

result6.show()

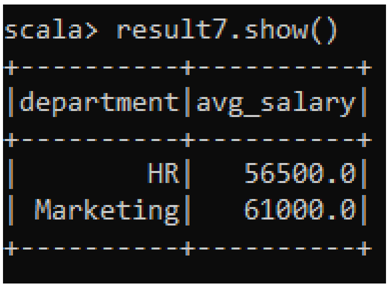


```
scala> result6.show()
+----------+----------+
|department|avg_salary|
+----------+----------+
|     Sales|   51000.0|
|        HR|   56500.0|
| Marketing|   61000.0|
+----------+----------+
```

Having Clause

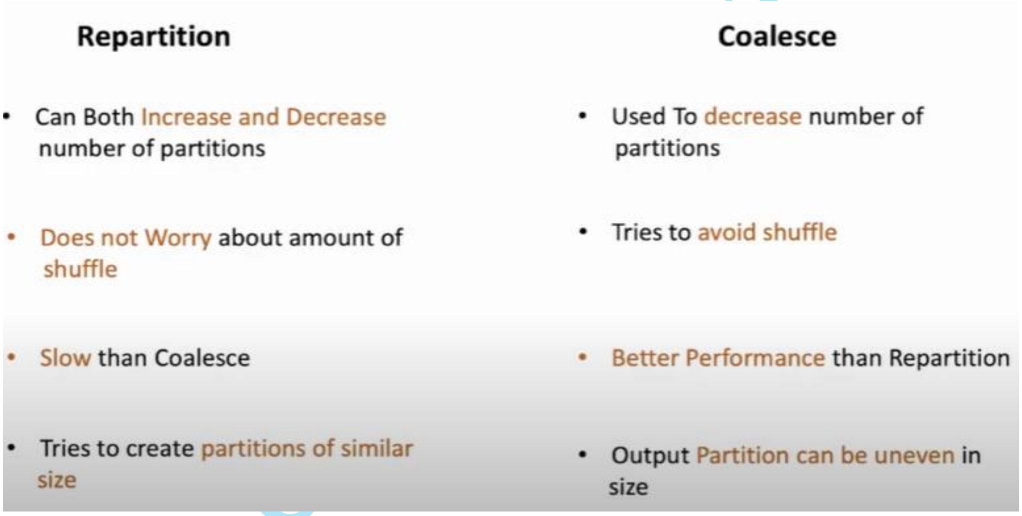
val result7 = spark.sql("SELECT department, AVG(salary) AS avg_salary FROM employee GROUP BY department HAVING AVG(salary) > 55000")

result7.show()

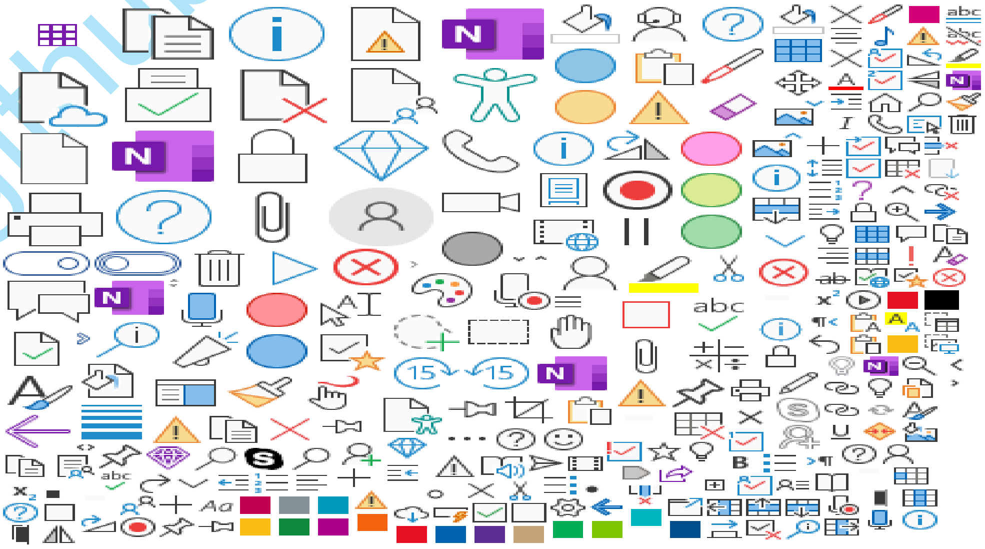


```
scala> result7.show()
+----------+----------+
|department|avg_salary|
+----------+----------+
|        HR|   56500.0|
| Marketing|   61000.0|
+----------+----------+
```

Managing Spark Partitions with Coalesce and Repartition



Repartition	Coalesce
Can Both Increase and Decrease number of partitions	Used To decrease number of partitions
Does not Worry about amount of shuffle	Tries to avoid shuffle
Slow than Coalesce	Better Performance than Repartition
Tries to create partitions of similar size	Output Partition can be uneven in size



Data

```
val x = (1 to 10).toList

val numberDf = x.toDF("number")

numberDf.show()

numberDf.rdd.partitions.size

numberDf.write.csv("C:/Spark/sparkfiles/partinfo")
```

Coalesce (Merging)

```
val numberDf1 = numberDf.coalesce(3)

numberDf1.rdd.partitions.size

numberDf1.write.csv("C:/Spark/sparkfiles/partinfo1")
```

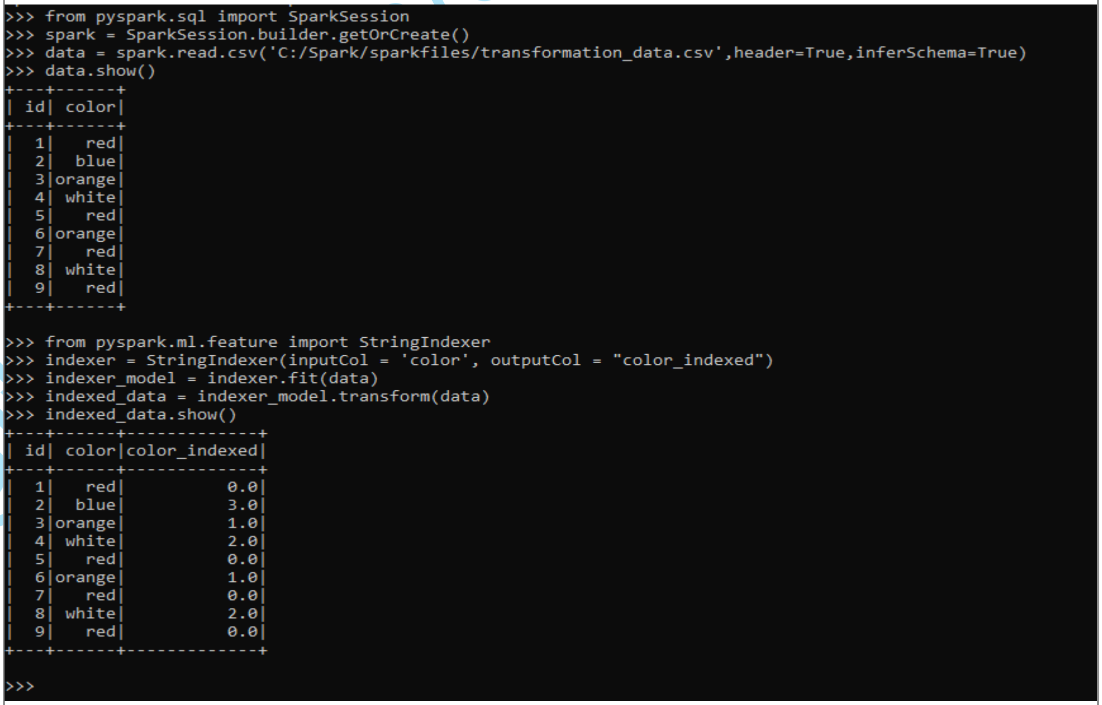
Repartition (Dividing)

```
val numberDf2 = numberDf.repartition(2)

numberDf2.rdd.partitions.size

numberDf2.write.csv("C:/Spark/sparkfiles/partinfo2")

val numberDf3 = numberDf.repartition(6)

numberDf3.rdd.partitions.size

numberDf3.write.csv("C:/Spark/sparkfiles/partinfo3")
```

Feature Transformation in Spark

Method-1

```
from pyspark.sql import SparkSession


spark = SparkSession.builder.appName("Name anything")getOrCreate()

data =
spark.read.csv('C:/Spark/sparkfiles/transformation_data.csv',header=True,inferSchema=True)

data.show()


from pyspark.ml.feature import StringIndexer


indexer = StringIndexer(inputCol = 'color', outputCol = "color_indexed")

indexer_model = indexer.fit(data)

indexed_data = indexer_model.transform(data)

indexed_data.show()
```



```
>>> from pyspark.sql import SparkSession
>>> spark = SparkSession.builder.getOrCreate()
>>> data = spark.read.csv('C:/Spark/sparkfiles/transformation_data.csv',header=True,inferSchema=True)
>>> data.show()
+---+------+
| id| color|
+---+------+
|  1|   red|
|  2|  blue|
|  3|orange|
|  4| white|
|  5|   red|
|  6|orange|
|  7|   red|
|  8| white|
|  9|   red|
+---+------+

>>> from pyspark.ml.feature import StringIndexer
>>> indexer = StringIndexer(inputCol = 'color', outputCol = "color_indexed")
>>> indexer_model = indexer.fit(data)
>>> indexed_data = indexer_model.transform(data)
>>> indexed_data.show()
+---+------+-------------+
| id| color|color_indexed|
+---+------+-------------+
|  1|   red|          0.0|
|  2|  blue|          3.0|
|  3|orange|          1.0|
|  4| white|          2.0|
|  5|   red|          0.0|
|  6|orange|          1.0|
|  7|   red|          0.0|
|  8| white|          2.0|
|  9|   red|          0.0|
+---+------+-------------+

>>>
```

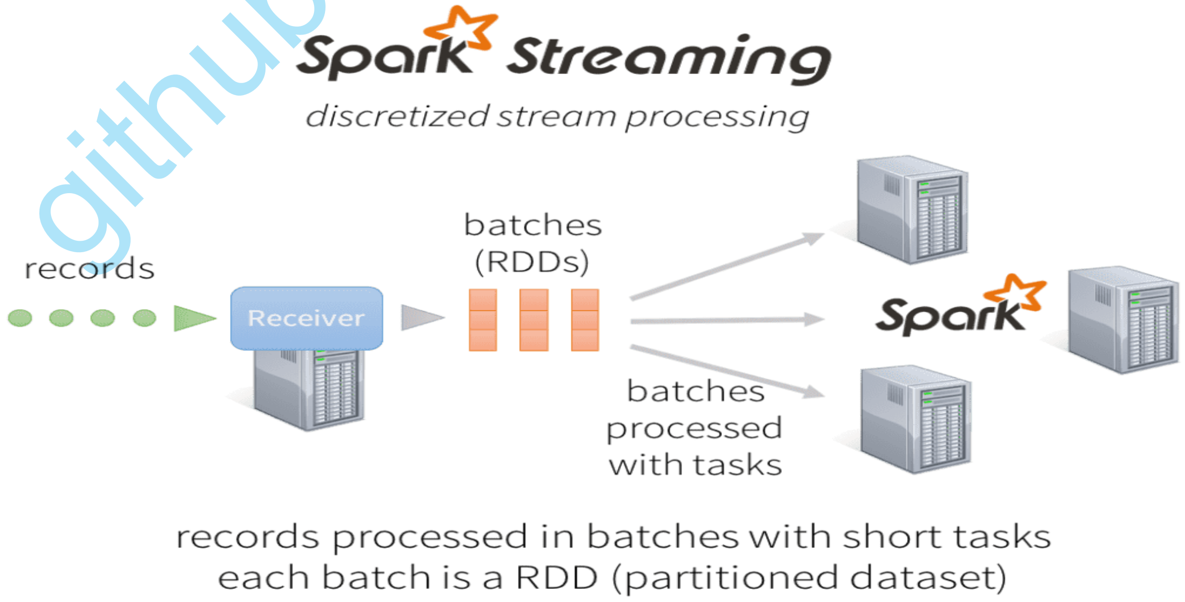
	Suppose you have a dataset with a categorical feature like "Color" with values "Red", "Blue", and "Green". Before applying a machine learning algorithm to predict some outcome based on this data, you need to convert these string values into numerical indices. This is where StringIndexer comes into play, transforming "Red" to 0, "Blue" to 1, and "Green" to 2, for example, allowing the algorithm to work with the data effectively.
Method-2	Feature Scaling

Spark Streaming

Apache Spark Streaming is a component of the Apache Spark ecosystem that enables scalable, high-throughput, fault-tolerant stream processing of real-time data.

Imagine we have a continuous stream of data coming in, like tweets from Twitter or sensor data from IoT devices. Spark Streaming allows us to process this data in real-time, meaning we can analyze and react to it as soon as it arrives.

How it Works:


Spark Streaming
discretized stream processing
records
Receiver
batches
(RDDs)
batches
processed
with tasks
Spark
records processed in batches with short tasks
each batch is a RDD (partitioned dataset)

Micro-Batching: Spark Streaming divides the incoming stream of data into small batches or micro-batches. Each batch represents a short interval of time, such as a few seconds.

Processing: Once the data is divided into batches, Spark processes each batch using the same powerful distributed computing engine used for batch processing in Apache Spark.

Transformation and Analysis: Within each batch, we can apply transformations, filters, aggregations, and machine learning algorithms to analyze the data and derive insights from it.

Output: After processing, we can store the results in various data stores, generate alerts, update dashboards, or trigger actions based on the analyzed data.

Benefits

Real-Time Insights: Spark Streaming enables us to analyze and respond to data in real-time, providing timely insights and actionable information.

Scalability: It leverages the scalability and fault-tolerance of Apache Spark, allowing us to process large volumes of data efficiently.

Integration: It integrates seamlessly with other components of the Spark ecosystem, such as Spark SQL, MLlib, and GraphX, enabling us to perform complex analytics and machine learning on streaming data.

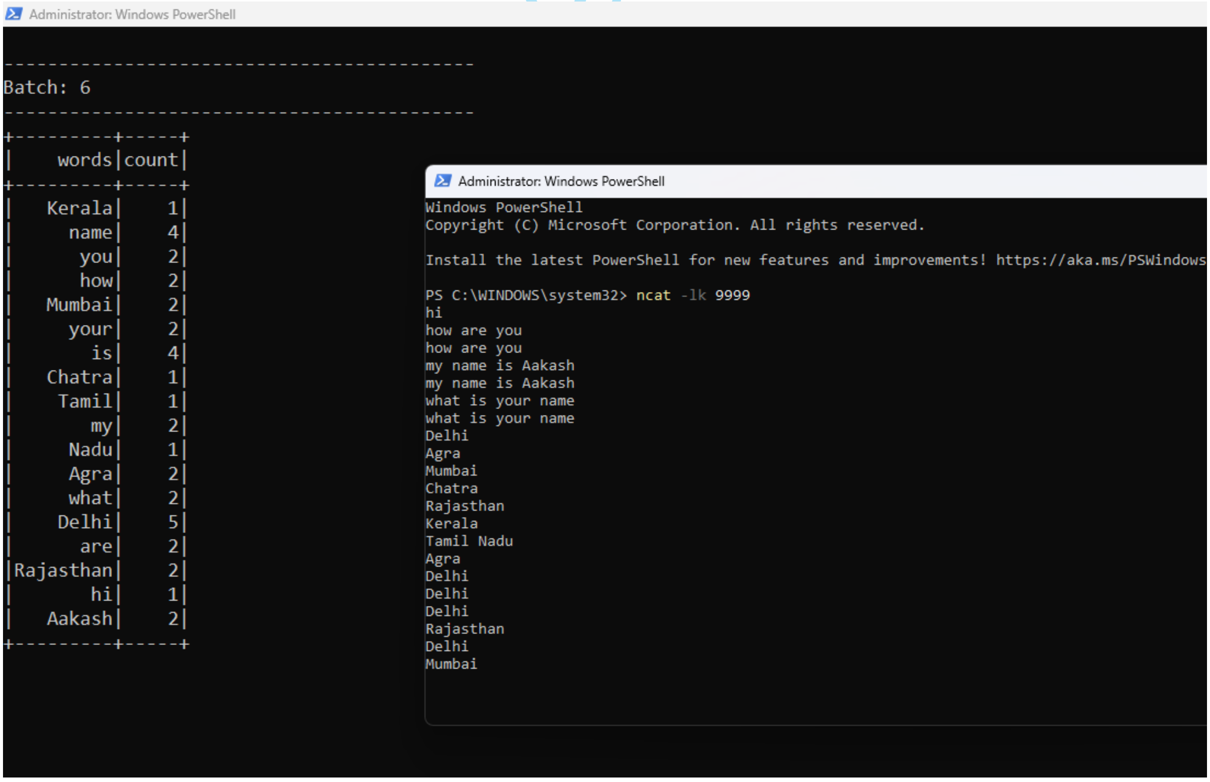
Streaming

```
import org.apache.spark.sql.{SparkSession}

import org.apache.spark.sql.functions._

val spark = SparkSession.builder().master("local").appName("mysource").getOrCreate()

val initDF = (spark.readStream.format("rate").option("rowsPerSecond",1).load())


println("Streaming DataFrame: " +initDF.isStreaming)


val resultDF = initDF.withColumn("result",col("value")+lit(1))


resultDF.writeStream.outputMode("append").option("truncate",false).format("console").start().awaitTermination()
```

Structures Streaming - Socket Source

```
//Powershell-1

// starting ncat server

ncat -lk 9999


//Powershell-2

// Import Libraries

import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.functions._


// Create Spark Session

val spark = SparkSession

  .builder()

  .master("local")

  .appName("Socket Source")

  .getOrCreate()


// Define host and port number to Listen.

val host = "127.0.0.1"

val port = "9999"


// Create Streaming DataFrame by reading data from socket.

val initDF = (spark

  .readStream

  .format("socket")

  .option("host", host)

  .option("port", port)

  .load())
```

```
// Perform word count on streaming DataFrame
val wordCount = (initDF
  .select(explode(split(col("value"), " ")).alias("words"))
  .groupBy("words")
  .count()
  )

wordCount
  .writeStream
  .outputMode("complete")
  .format("console")
  .start()
  .awaitTermination()
```



Administrator: Windows PowerShell

```
-------------------------------------------
Batch: 6
-------------------------------------------
+---------+-----+
|    words|count|
+---------+-----+
|   Kerala|    1|
|     name|    4|
|      you|    2|
|      how|    2|
|   Mumbai|    2|
|     your|    2|
|       is|    4|
|   Chatra|    1|
|    Tamil|    1|
|       my|    2|
|     Nadu|    1|
|     Agra|    2|
|     what|    2|
|    Delhi|    5|
|      are|    2|
|Rajasthan|    2|
|       hi|    1|
|   Aakash|    2|
+---------+-----+
```

Administrator: Windows PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\WINDOWS\system32> ncat -lk 9999
hi
how are you
how are you
my name is Aakash
my name is Aakash
what is your name
what is your name
Delhi
Agra
Mumbai
Chatra
Rajasthan
Kerala
Tamil Nadu
Agra
Delhi
Delhi
Delhi
Rajasthan
Delhi
Mumbai
```

kafka

Apache Kafka is an open-source distributed event streaming platform used for building real-time data pipelines and streaming applications. It is designed to handle high-throughput, fault-tolerant, and scalable ingestion, storage, and processing of real-time data streams.

Imagine we're at a busy post office where letters are constantly arriving and need to be sorted, processed, and delivered to the right destinations. The post office has several departments working together efficiently to handle this continuous flow of mail. Apache Kafka works similarly but with data instead of letters.

- **Post Office:** Think of Kafka as a modern post office. Just like the post office receives, processes, and delivers mail, Kafka receives, processes, and delivers data.
- **Data Pipelines:** Data pipelines are like conveyor belts in the post office, moving mail from one department to another. Kafka acts as a reliable conveyor belt for your data, ensuring it gets from one system to another without loss or delay.

Why Kafka is Used:

Real-Time Data Processing: Kafka enables organizations to process and react to data in real-time, allowing them to make timely decisions and provide immediate responses.

Scalability: Kafka is designed to handle large volumes of data and can scale horizontally by adding more servers to the cluster, making it suitable for high-throughput applications.

Fault Tolerance: Kafka is fault-tolerant, meaning it can continue to function even if some of its servers fail. This ensures that data is not lost and processing can continue uninterrupted.

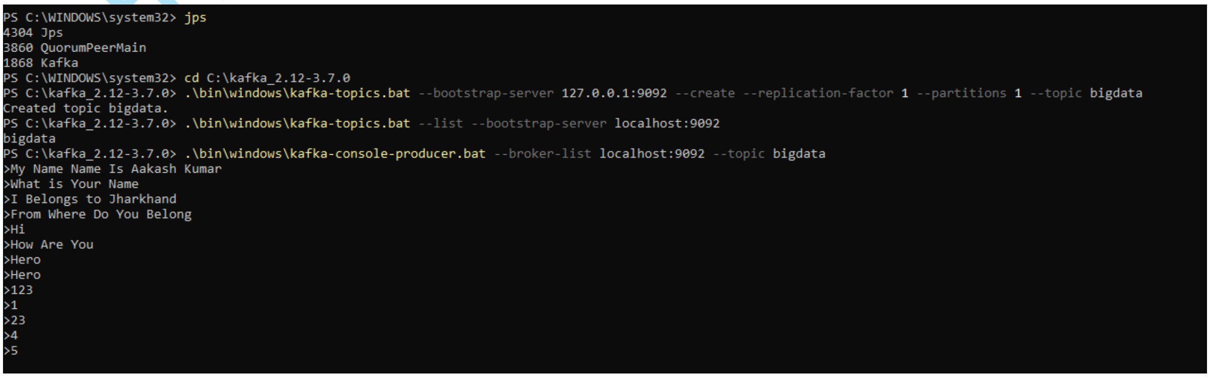
Data Integration: Kafka serves as a central hub for integrating different systems and applications, allowing data to flow seamlessly between them.

Event Streaming: Kafka facilitates event-driven architectures where applications can react to events in real-time, enabling use cases like real-time analytics, monitoring, and fraud detection.

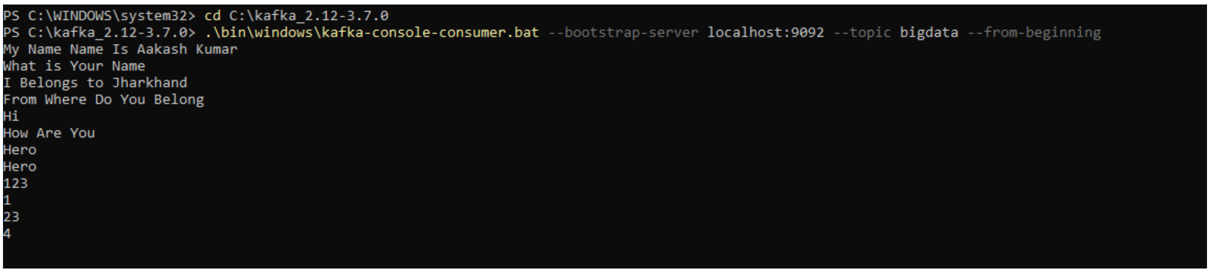
Producer	**Consumer**
• A Producer is a component or application responsible for sending data (messages) to Kafka topics. • It's like a publisher or sender of messages. • Producers are typically used to push data from various sources into Kafka topics. • For example, imagine a sensor sending temperature readings to a Kafka topic every second. In this scenario, the sensor acts as a Producer, continuously sending temperature data to Kafka.	• A Consumer is a component or application responsible for reading data (messages) from Kafka topics. • It's like a subscriber or receiver of messages. • Consumers retrieve data from Kafka topics and process it according to their requirements. • For example, imagine an analytics application that analyzes temperature trends. This application would consume temperature data from a Kafka topic, process it to detect trends, and possibly visualize the results.

Commands	Start Zookeeper 1. cd C:\kafka_2.12-3.7.0 2. .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties Start Server 1. cd C:\kafka_2.12-3.7.0 2. .\bin\windows\kafka-server-start.bat .\config\server.properties Go to new powershell window 1. jps

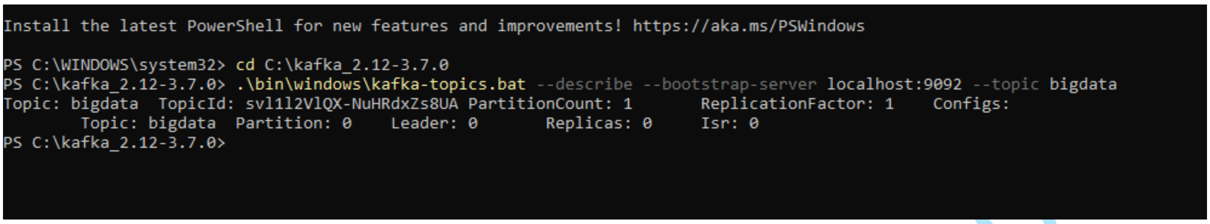
Create Topic	.\bin\windows\kafka-topics.bat --bootstrap-server 127.0.0.1:9092 --create --replication-factor 1 --partitions 1 --topic bigdata
Topic List	.\bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092
Producer	.\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic bigdata
Consumer	.\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic bigdata --from-beginning
Describe	.\bin\windows\kafka-topics.bat --describe --bootstrap-server localhost:9092 --topic bigdata
Delete	.\bin\windows\kafka-topics.bat --bootstrap-server localhost:9092 --delete --topic bigdata



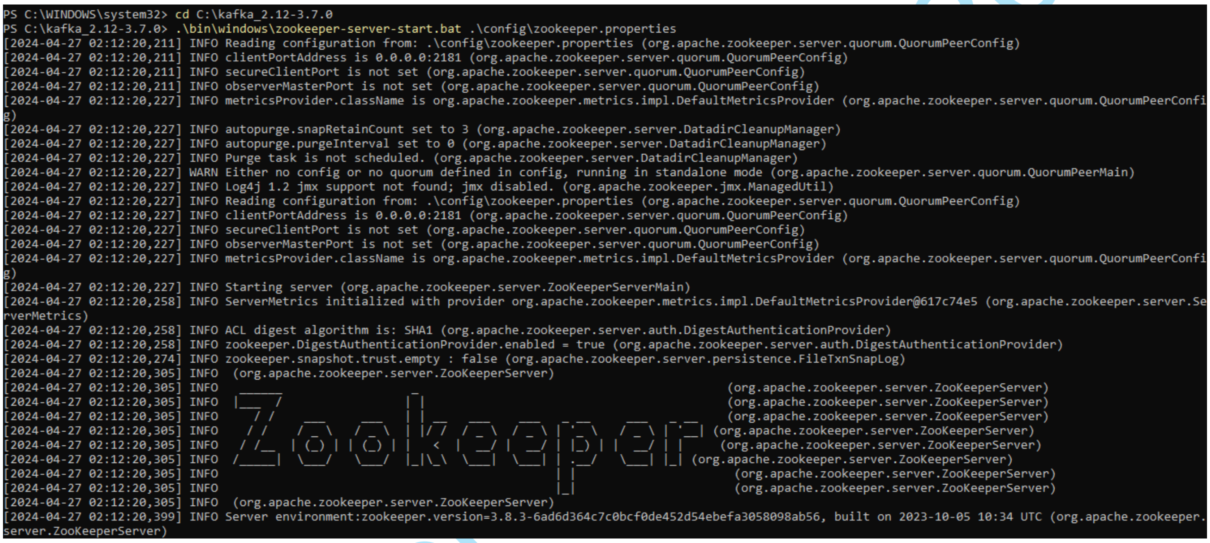
```
PS C:\WINDOWS\system32> jps
4304 Jps
3860 QuorumPeerMain
1868 Kafka
PS C:\WINDOWS\system32> cd C:\kafka_2.12-3.7.0
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-topics.bat --bootstrap-server 127.0.0.1:9092 --create --replication-factor 1 --partitions 1 --topic bigdata
Created topic bigdata.
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092
bigdata
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic bigdata
>My Name Name Is Aakash Kumar
>What is Your Name
>I Belongs to Jharkhand
>From Where Do You Belong
>Hi
>How Are You
>Hero
>Hero
>123
>1
>23
>4
>5
```



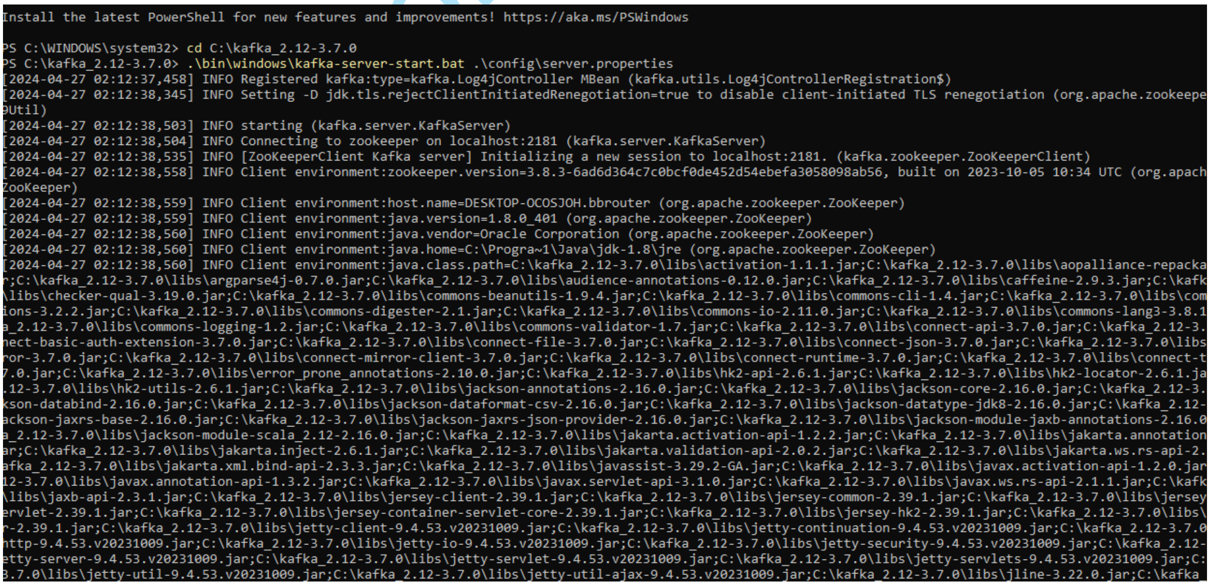
```
PS C:\WINDOWS\system32> cd C:\kafka_2.12-3.7.0
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic bigdata --from-beginning
My Name Name Is Aakash Kumar
What is Your Name
I Belongs to Jharkhand
From Where Do You Belong
Hi
How Are You
Hero
Hero
123
1
23
4
```



```
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\WINDOWS\system32> cd C:\kafka_2.12-3.7.0
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-topics.bat --describe --bootstrap-server localhost:9092 --topic bigdata
Topic: bigdata  TopicId: svl1l2VlQX-NuHRdxZs8UA PartitionCount: 1       ReplicationFactor: 1    Configs:
        Topic: bigdata  Partition: 0    Leader: 0       Replicas: 0     Isr: 0
PS C:\kafka_2.12-3.7.0>
```



```
PS C:\WINDOWS\system32> cd C:\kafka_2.12-3.7.0
PS C:\kafka_2.12-3.7.0> .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
[2024-04-27 02:12:20,211] INFO Reading configuration from: .\config\zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-04-27 02:12:20,211] INFO clientPortAddress is 0.0.0.0:2181 (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-04-27 02:12:20,211] INFO secureClientPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-04-27 02:12:20,211] INFO observerMasterPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-04-27 02:12:20,227] INFO metricsProvider.className is org.apache.zookeeper.metrics.impl.DefaultMetricsProvider (org.apache.zookeeper.server.quorum.QuorumPeerConfi
g)
[2024-04-27 02:12:20,227] INFO autopurge.snapRetainCount set to 3 (org.apache.zookeeper.server.DatadirCleanupManager)
[2024-04-27 02:12:20,227] INFO autopurge.purgeInterval set to 0 (org.apache.zookeeper.server.DatadirCleanupManager)
[2024-04-27 02:12:20,227] INFO Purge task is not scheduled. (org.apache.zookeeper.server.DatadirCleanupManager)
[2024-04-27 02:12:20,227] WARN Either no config or no quorum defined in config, running in standalone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)
[2024-04-27 02:12:20,227] INFO Log4j 1.2 jmx support not found; jmx disabled. (org.apache.zookeeper.jmx.ManagedUtil)
[2024-04-27 02:12:20,227] INFO Reading configuration from: .\config\zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-04-27 02:12:20,227] INFO clientPortAddress is 0.0.0.0:2181 (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-04-27 02:12:20,227] INFO secureClientPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-04-27 02:12:20,227] INFO observerMasterPort is not set (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2024-04-27 02:12:20,227] INFO metricsProvider.className is org.apache.zookeeper.metrics.impl.DefaultMetricsProvider (org.apache.zookeeper.server.quorum.QuorumPeerConfi
g)
[2024-04-27 02:12:20,227] INFO Starting server (org.apache.zookeeper.server.ZooKeeperServerMain)
[2024-04-27 02:12:20,258] INFO ServerMetrics initialized with provider org.apache.zookeeper.metrics.impl.DefaultMetricsProvider@617c74e5 (org.apache.zookeeper.server.Se
rverMetrics)
[2024-04-27 02:12:20,258] INFO ACL digest algorithm is: SHA1 (org.apache.zookeeper.server.auth.DigestAuthenticationProvider)
[2024-04-27 02:12:20,258] INFO zookeeper.DigestAuthenticationProvider.enabled = true (org.apache.zookeeper.server.auth.DigestAuthenticationProvider)
[2024-04-27 02:12:20,274] INFO zookeeper.snapshot.trust.empty : false (org.apache.zookeeper.server.persistence.FileTxnSnapLog)
[2024-04-27 02:12:20,305] INFO  (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,305] INFO   ______                  _                                          (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,305] INFO  |___  /                 | |                                         (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,305] INFO     / /    ___     ___   | | __   ___    ___   _ __     ___   _ __   (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,305] INFO    / /    / _ \   / _ \  | |/ /  / _ \  / _ \ | '_ \   / _ \ | '__| (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,305] INFO   / /__  | (_) | | (_) | |   <  |  __/ |  __/ | |_) | |  __/ | |    (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,305] INFO  /_____|  \___/   \___/  |_|\_\  \___|  \___| | .__/   \___| |_| (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,305] INFO                                               | |                     (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,305] INFO                                               |_|                     (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,305] INFO  (org.apache.zookeeper.server.ZooKeeperServer)
[2024-04-27 02:12:20,399] INFO Server environment:zookeeper.version=3.8.3-6ad6d364c7c0bcf0de452d54ebefa3058098ab56, built on 2023-10-05 10:34 UTC (org.apache.zookeeper.
server.ZooKeeperServer)
```



```
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\WINDOWS\system32> cd C:\kafka_2.12-3.7.0
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-server-start.bat .\config\server.properties
[2024-04-27 02:12:37,458] INFO Registered kafka:type=kafka.Log4jController MBean (kafka.utils.Log4jControllerRegistration$)
[2024-04-27 02:12:38,345] INFO Setting -D jdk.tls.rejectClientInitiatedRenegotiation=true to disable client-initiated TLS renegotiation (org.apache.zookeepe
9Util)
[2024-04-27 02:12:38,503] INFO starting (kafka.server.KafkaServer)
[2024-04-27 02:12:38,504] INFO Connecting to zookeeper on localhost:2181 (kafka.server.KafkaServer)
[2024-04-27 02:12:38,535] INFO [ZooKeeperClient Kafka server] Initializing a new session to localhost:2181. (kafka.zookeeper.ZooKeeperClient)
[2024-04-27 02:12:38,558] INFO Client environment:zookeeper.version=3.8.3-6ad6d364c7c0bcf0de452d54ebefa3058098ab56, built on 2023-10-05 10:34 UTC (org.apach
ZooKeeper)
[2024-04-27 02:12:38,559] INFO Client environment:host.name=DESKTOP-OCOSJOH.bbrouter (org.apache.zookeeper.ZooKeeper)
[2024-04-27 02:12:38,559] INFO Client environment:java.version=1.8.0_401 (org.apache.zookeeper.ZooKeeper)
[2024-04-27 02:12:38,560] INFO Client environment:java.vendor=Oracle Corporation (org.apache.zookeeper.ZooKeeper)
[2024-04-27 02:12:38,560] INFO Client environment:java.home=C:\Progra~1\Java\jdk-1.8\jre (org.apache.zookeeper.ZooKeeper)
[2024-04-27 02:12:38,560] INFO Client environment:java.class.path=C:\kafka_2.12-3.7.0\libs\activation-1.1.1.jar;C:\kafka_2.12-3.7.0\libs\aopalliance-repacka
r;C:\kafka_2.12-3.7.0\libs\argparse4j-0.7.0.jar;C:\kafka_2.12-3.7.0\libs\audience-annotations-0.12.0.jar;C:\kafka_2.12-3.7.0\libs\caffeine-2.9.3.jar;C:\kafk
\libs\checker-qual-3.19.0.jar;C:\kafka_2.12-3.7.0\libs\commons-beanutils-1.9.4.jar;C:\kafka_2.12-3.7.0\libs\commons-cli-1.4.jar;C:\kafka_2.12-3.7.0\libs\com
ions-3.2.2.jar;C:\kafka_2.12-3.7.0\libs\commons-digester-2.1.jar;C:\kafka_2.12-3.7.0\libs\commons-io-2.11.0.jar;C:\kafka_2.12-3.7.0\libs\commons-lang3-3.8.1
a_2.12-3.7.0\libs\commons-logging-1.2.jar;C:\kafka_2.12-3.7.0\libs\commons-validator-1.7.jar;C:\kafka_2.12-3.7.0\libs\connect-api-3.7.0.jar;C:\kafka_2.12-3.
nect-basic-auth-extension-3.7.0.jar;C:\kafka_2.12-3.7.0\libs\connect-file-3.7.0.jar;C:\kafka_2.12-3.7.0\libs\connect-json-3.7.0.jar;C:\kafka_2.12-3.7.0\libs
ror-3.7.0.jar;C:\kafka_2.12-3.7.0\libs\connect-mirror-client-3.7.0.jar;C:\kafka_2.12-3.7.0\libs\connect-runtime-3.7.0.jar;C:\kafka_2.12-3.7.0\libs\connect-t
7.0.jar;C:\kafka_2.12-3.7.0\libs\error_prone_annotations-2.10.0.jar;C:\kafka_2.12-3.7.0\libs\hk2-api-2.6.1.jar;C:\kafka_2.12-3.7.0\libs\hk2-locator-2.6.1.ja
.12-3.7.0\libs\hk2-utils-2.6.1.jar;C:\kafka_2.12-3.7.0\libs\jackson-annotations-2.16.0.jar;C:\kafka_2.12-3.7.0\libs\jackson-core-2.16.0.jar;C:\kafka_2.12-3.
kson-databind-2.16.0.jar;C:\kafka_2.12-3.7.0\libs\jackson-dataformat-csv-2.16.0.jar;C:\kafka_2.12-3.7.0\libs\jackson-datatype-jdk8-2.16.0.jar;C:\kafka_2.12-
ackson-jaxrs-base-2.16.0.jar;C:\kafka_2.12-3.7.0\libs\jackson-jaxrs-json-provider-2.16.0.jar;C:\kafka_2.12-3.7.0\libs\jackson-module-jaxb-annotations-2.16.0
a_2.12-3.7.0\libs\jackson-module-scala_2.12-2.16.0.jar;C:\kafka_2.12-3.7.0\libs\jakarta.activation-api-1.2.2.jar;C:\kafka_2.12-3.7.0\libs\jakarta.annotation
ar;C:\kafka_2.12-3.7.0\libs\jakarta.inject-2.6.1.jar;C:\kafka_2.12-3.7.0\libs\jakarta.validation-api-2.0.2.jar;C:\kafka_2.12-3.7.0\libs\jakarta.ws.rs-api-2.
afka_2.12-3.7.0\libs\jakarta.xml.bind-api-2.3.3.jar;C:\kafka_2.12-3.7.0\libs\javassist-3.29.2-GA.jar;C:\kafka_2.12-3.7.0\libs\javax.activation-api-1.2.0.jar
12-3.7.0\libs\javax.annotation-api-1.3.2.jar;C:\kafka_2.12-3.7.0\libs\javax.servlet-api-3.1.0.jar;C:\kafka_2.12-3.7.0\libs\javax.ws.rs-api-2.1.1.jar;C:\kafk
\libs\jaxb-api-2.3.1.jar;C:\kafka_2.12-3.7.0\libs\jersey-client-2.39.1.jar;C:\kafka_2.12-3.7.0\libs\jersey-common-2.39.1.jar;C:\kafka_2.12-3.7.0\libs\jersey
ervlet-2.39.1.jar;C:\kafka_2.12-3.7.0\libs\jersey-container-servlet-core-2.39.1.jar;C:\kafka_2.12-3.7.0\libs\jersey-hk2-2.39.1.jar;C:\kafka_2.12-3.7.0\libs\
r-2.39.1.jar;C:\kafka_2.12-3.7.0\libs\jetty-client-9.4.53.v20231009.jar;C:\kafka_2.12-3.7.0\libs\jetty-continuation-9.4.53.v20231009.jar;C:\kafka_2.12-3.7.0
http-9.4.53.v20231009.jar;C:\kafka_2.12-3.7.0\libs\jetty-io-9.4.53.v20231009.jar;C:\kafka_2.12-3.7.0\libs\jetty-security-9.4.53.v20231009.jar;C:\kafka_2.12-
etty-server-9.4.53.v20231009.jar;C:\kafka_2.12-3.7.0\libs\jetty-servlet-9.4.53.v20231009.jar;C:\kafka_2.12-3.7.0\libs\jetty-servlets-9.4.53.v20231009.jar;C:
3.7.0\libs\jetty-util-9.4.53.v20231009.jar;C:\kafka_2.12-3.7.0\libs\jetty-util-ajax-9.4.53.v20231009.jar;C:\kafka_2.12-3.7.0\libs\jline-3.22.0.jar;C:\kafka_
```

Create a Java project with name MyProject to configure the consumer in Apache Kafka. Link that consumer with producer via command line and show the communication. (Using Eclipse IDE)

Step-01	Create a New Java Project: • Open Eclipse IDE. • Go to File > New > Java Project. • Enter "MyProject" as the project name. • Click "Finish" to create the project.
Step-02	Add Kafka Dependencies: • Right-click on the project in the Project Explorer. • Navigate to Build Path > Configure Build Path. • In the Libraries tab, click on "Add External JARs...". • Add the Kafka client JARs (e.g., kafka-clients-x.x.x.jar, slf4j-api-x.x.x.jar, etc.) from the Kafka installation directory. • Click "Apply and Close".
Step-03	Create a Kafka Consumer Class: • Right-click on the src folder in the project. • Go to New > Class. • Name the class "KafkaConsumerExample" and click "Finish". • Add code to configure and run the Kafka consumer: import org.apache.kafka.clients.consumer.ConsumerConfig; import org.apache.kafka.clients.consumer.ConsumerRecord; import org.apache.kafka.clients.consumer.ConsumerRecords; import org.apache.kafka.clients.consumer.KafkaConsumer;

```
import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class KafkaConsumerExample {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Collections.singletonList("my-topic"));

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("Received message: key = %s, value = %s%n", record.key(), record.value());
            }
        }
    }
}
```

Step-04

Run Producer and Consumer via Command Line:

- Start Zookeeper and Kafka server if not already running.
- Create a topic named "my-topic" using the following command:

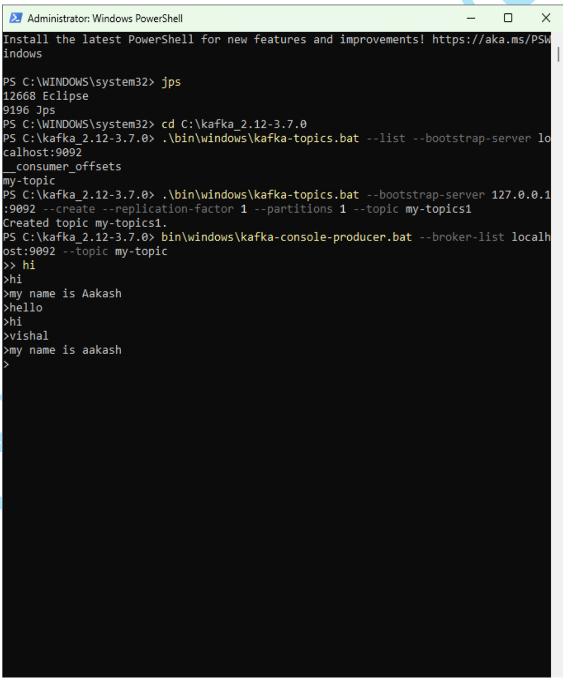
```
.\bin\windows\kafka-topics.bat --bootstrap-server 127.0.0.1:9092 --create --replication-factor 1 --partitions 1 --topic my-topic
```

- Run a Kafka producer in one Command Prompt window:

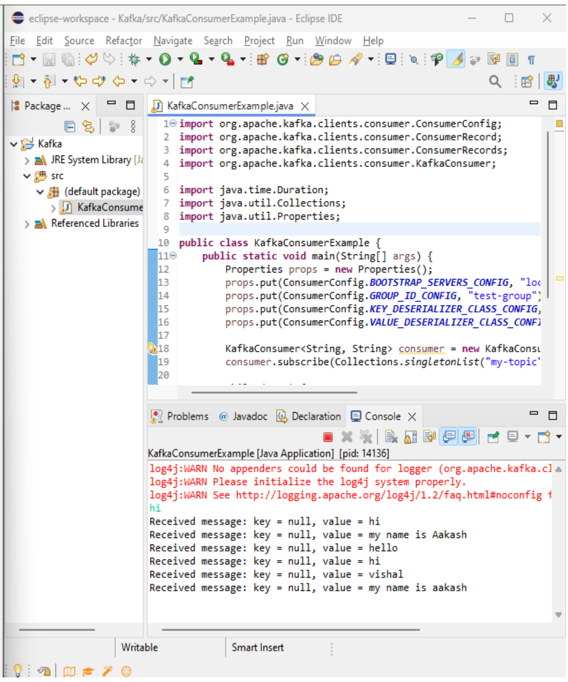
```
.\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic my-topic
```

- Enter some messages in the producer terminal.
- Run the Kafka consumer class in eclipse IDE

Step-05



```
Install the latest PowerShell for new features and improvements! https://aka.ms/PSW
indows

PS C:\WINDOWS\system32> jps
12668 Eclipse
9196 Jps
PS C:\WINDOWS\system32> cd C:\kafka_2.12-3.7.0
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-topics.bat --list --bootstrap-server lo
calhost:9092
__consumer_offsets
my-topic
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-topics.bat --bootstrap-server 127.0.0.1
:9092 --create --replication-factor 1 --partitions 1 --topic my-topics1
Created topic my-topics1.
PS C:\kafka_2.12-3.7.0> bin\windows\kafka-console-producer.bat --broker-list localh
ost:9092 --topic my-topic
>> hi
>hi
>my name is Aakash
>hello
>hi
>vishal
>my name is aakash
>
```


eclipse-workspace - Kafka/src/KafkaConsumerExample.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package ...
Kafka
JRE System Library [J
src
(default package)
KafkaConsume
Referenced Libraries
KafkaConsumerExample.java
Problems
Javadoc
Declaration
Console
Writable
Smart Insert

```java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class KafkaConsumerExample {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "loc
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group")
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFI

        KafkaConsumer<String, String> consumer = new KafkaConsu
        consumer.subscribe(Collections.singletonList("my-topic'
```

KafkaConsumerExample [Java Application] [pid: 14136]

```
log4j:WARN No appenders could be found for logger (org.apache.kafka.cl
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig f
hi
Received message: key = null, value = hi
Received message: key = null, value = my name is Aakash
Received message: key = null, value = hello
Received message: key = null, value = hi
Received message: key = null, value = vishal
Received message: key = null, value = my name is aakash
```

Create a Java project with name "MyProject" to configure the producer in Apache Kafka. Link that Producer with Consumer via command line and show the communication. (Using Eclipse)

Step-01	Create a New Java Project in Eclipse: • Open Eclipse IDE. • Go to File > New > Java Project. • Enter "MyProject" as the project name. • Click "Finish" to create the project.
Step-02	Add Kafka Dependencies: • Right-click on the project in the Project Explorer. • Navigate to Build Path > Configure Build Path. • In the Libraries tab, click on "Add External JARs...". • Add the Kafka client JARs (e.g., kafka-clients-x.x.x.jar, slf4j-api-x.x.x.jar, etc.) from the Kafka installation directory. • Click "Apply and Close".
Step-03	Create a Kafka Producer Class: Right-click on the src folder in the project. Go to New > Class. Name the class "KafkaProducerExample" and click "Finish". Add the following code to configure and run the Kafka producer: **Code1**: `import org.apache.kafka.clients.producer.*;`

```
import java.util.Properties;


public class KafkaProducerExample {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");


        Producer<String, String> producer = new KafkaProducer<>(props);


        String topic = "my-topic";
        for (int i = 0; i < 10; i++) {
            String key = "key" + i;
            String value = "value" + i;
            ProducerRecord<String, String> record = new ProducerRecord<>(topic, key,
value);
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception exception) {
                    if (exception != null) {
                        exception.printStackTrace();
                    } else {
                        System.out.printf("Sent message: topic = %s, partition = %d, offset =
%d, key = %s, value = %s%n",
                                metadata.topic(), metadata.partition(), metadata.offset(), key,
value);
                    }
            }
        });
    }

    producer.flush();
    producer.close();
  }
}
```

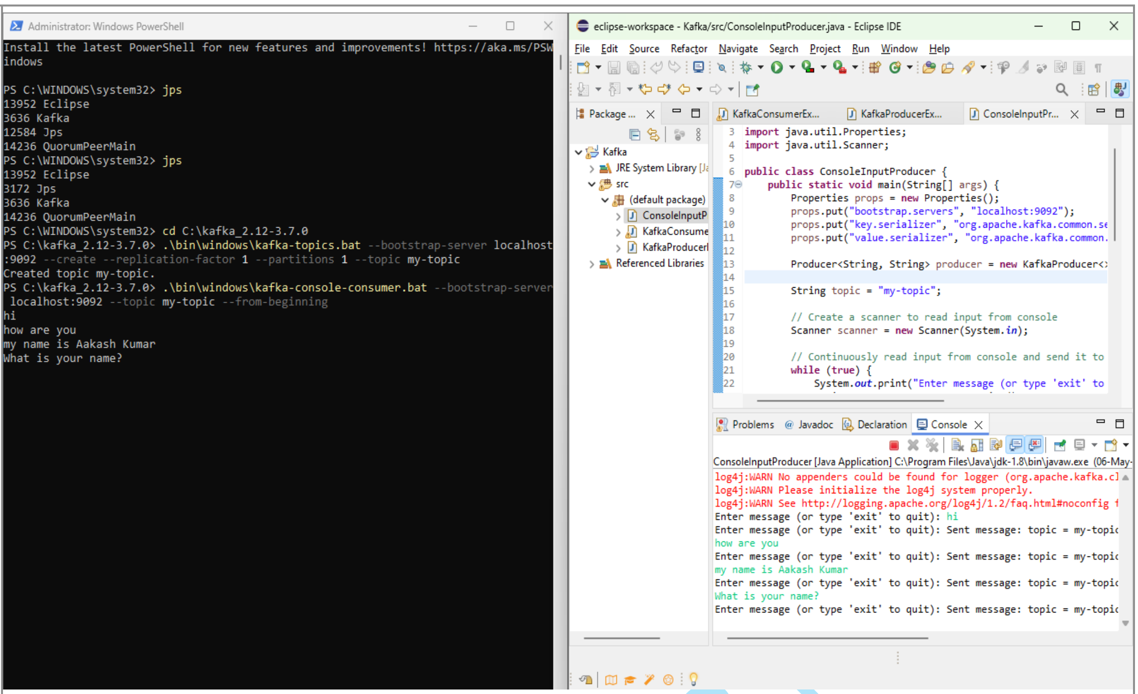
Code-2

```
import org.apache.kafka.clients.producer.*;

import java.util.Properties;
import java.util.Scanner;

public class ConsoleInputProducer {
  public static void main(String[] args) {
    Properties props = new Properties();
    props.put("bootstrap.servers", "localhost:9092");
    props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
    props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

    Producer<String, String> producer = new KafkaProducer<>(props);

    String topic = "my-topic";
```

```
        // Create a scanner to read input from console
        Scanner scanner = new Scanner(System.in);

        // Continuously read input from console and send it to Kafka
        while (true) {
            System.out.print("Enter message (or type 'exit' to quit): ");
            String message = scanner.nextLine();

            // If user types 'exit', break the loop and close the producer
            if (message.equalsIgnoreCase("exit")) {
                break;
            }

            // Create a producer record with the input message
            ProducerRecord<String, String> record = new ProducerRecord<>(topic, message);

            // Send the record to Kafka
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception exception) {
                    if (exception != null) {
                        exception.printStackTrace();
                    } else {
                        System.out.printf("Sent message: topic = %s, partition = %d, offset = %d, value = %s%n",
                                metadata.topic(), metadata.partition(), metadata.offset(), message);
                    }
                }
```

	});
	}
	// Close the producer
	producer.close();
	scanner.close();
	}
	}

Step-04	Run Producer and Consumer via Command Line: • Start Zookeeper and Kafka server if not already running. Navigate to the Kafka installation directory in the Command Prompt. • Create a topic named "my-topic" using the following command: .\bin\windows\kafka-topics.bat --bootstrap-server localhost:9092 --create --replication-factor 1 --partitions 1 --topic my-topic • Run the Kafka consumer in one Command Prompt window: .\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic my-topic --from-beginning • Run the Kafka producer code in Eclipse IDE

Step-05	

Administrator: Windows PowerShell

```
Install the latest PowerShell for new features and improvements! https://aka.ms/PSW
indows

PS C:\WINDOWS\system32> jps
13952 Eclipse
3636 Kafka
12584 Jps
14236 QuorumPeerMain
PS C:\WINDOWS\system32> jps
13952 Eclipse
3172 Jps
3636 Kafka
14236 QuorumPeerMain
PS C:\WINDOWS\system32> cd C:\kafka_2.12-3.7.0
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-topics.bat --bootstrap-server localhost
:9092 --create --replication-factor 1 --partitions 1 --topic my-topic
Created topic my-topic.
PS C:\kafka_2.12-3.7.0> .\bin\windows\kafka-console-consumer.bat --bootstrap-server
 localhost:9092 --topic my-topic --from-beginning
hi
how are you
my name is Aakash Kumar
What is your name?
```

eclipse-workspace - Kafka/src/ConsoleInputProducer.java - Eclipse IDE

```java
import java.util.Properties;
import java.util.Scanner;

public class ConsoleInputProducer {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer", "org.apache.kafka.common.se
        props.put("value.serializer", "org.apache.kafka.common.

        Producer<String, String> producer = new KafkaProducer<>

        String topic = "my-topic";

        // Create a scanner to read input from console
        Scanner scanner = new Scanner(System.in);

        // Continuously read input from console and send it to
        while (true) {
            System.out.print("Enter message (or type 'exit' to
```

ConsoleInputProducer [Java Application] C:\Program Files\Java\jdk-1.8\bin\javaw.exe (06-May-

```
log4j:WARN No appenders could be found for logger (org.apache.kafka.cl
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig f
Enter message (or type 'exit' to quit): hi
Enter message (or type 'exit' to quit): Sent message: topic = my-topic
how are you
Enter message (or type 'exit' to quit): Sent message: topic = my-topic
my name is Aakash Kumar
Enter message (or type 'exit' to quit): Sent message: topic = my-topic
What is your name?
Enter message (or type 'exit' to quit): Sent message: topic = my-topic
```