Big Data Hadoop and Spark Developer

Working with Spark RDDs

Learning Objectives

By the end of this lesson, you will be able to:

- Identify the challenges in existing computing methods
- Apply RDD with its operations, transformations, and actions
- Load and save data through RDD
- Work with Spark Pair RDDs
- Implement RDD lineage and RDD persistence

Learning Objectives

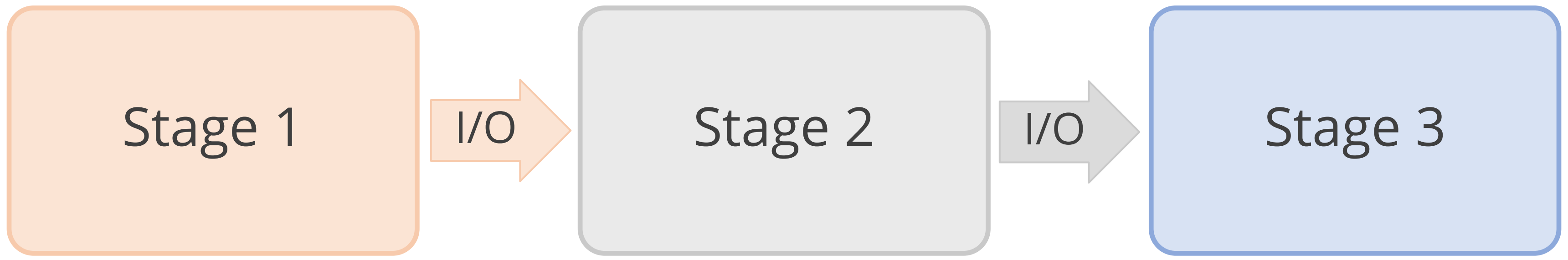
By the end of this lesson, you will be able to:

- Implement Word Count program using RDD
- Apply RDD partitioning to achieve parallelization
- Pass functions to Apache Spark

Challenges in Existing Computing Methods

Challenges in Existing Computing Methods

The processing in distributed computing occurs in numerous phases, with the output of the first stage becoming input to the second stage and the data being shared between the two.



The intermediate data is saved in file base storage in-between stages. Stage 1 saves the data on disk, and Stage 2 reads the same data for further processing. This kind of processing involves a lot of input-output overhead and makes the overall computation slower.

Probable Solution

- Reduction of the input-output operations that slow the computation can be accomplished through in-memory data sharing.
- The data transferred between in-memory is 10–100 times faster than data sharing through a network or a disc.

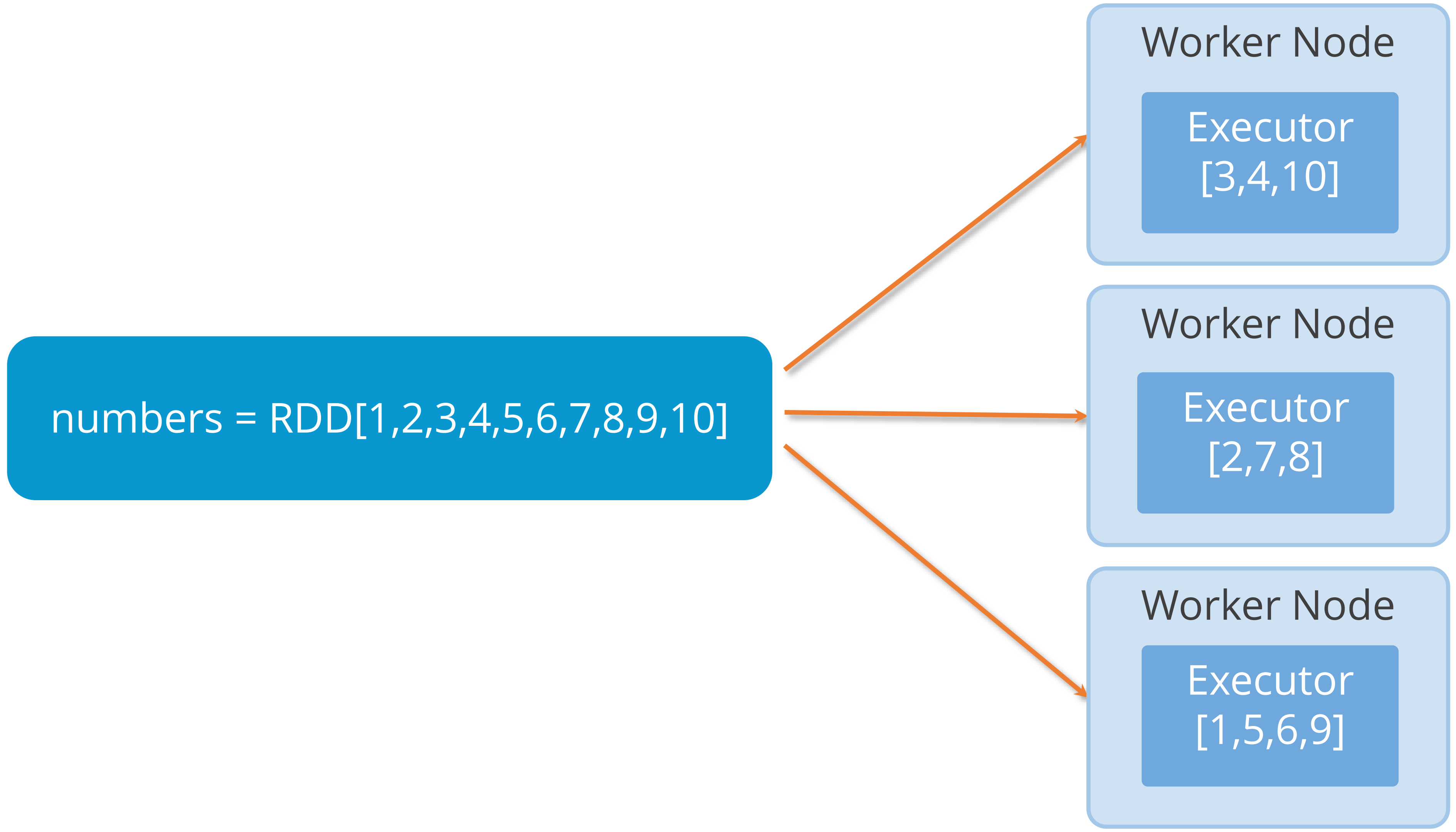
Ideal Solution: RDD

- RDD stands for Resilient Distributed Dataset.
- RDD enables fault-tolerant, distributed in-memory computations.
- RDD supports lazy evaluation.

Resilient Distributed Dataset

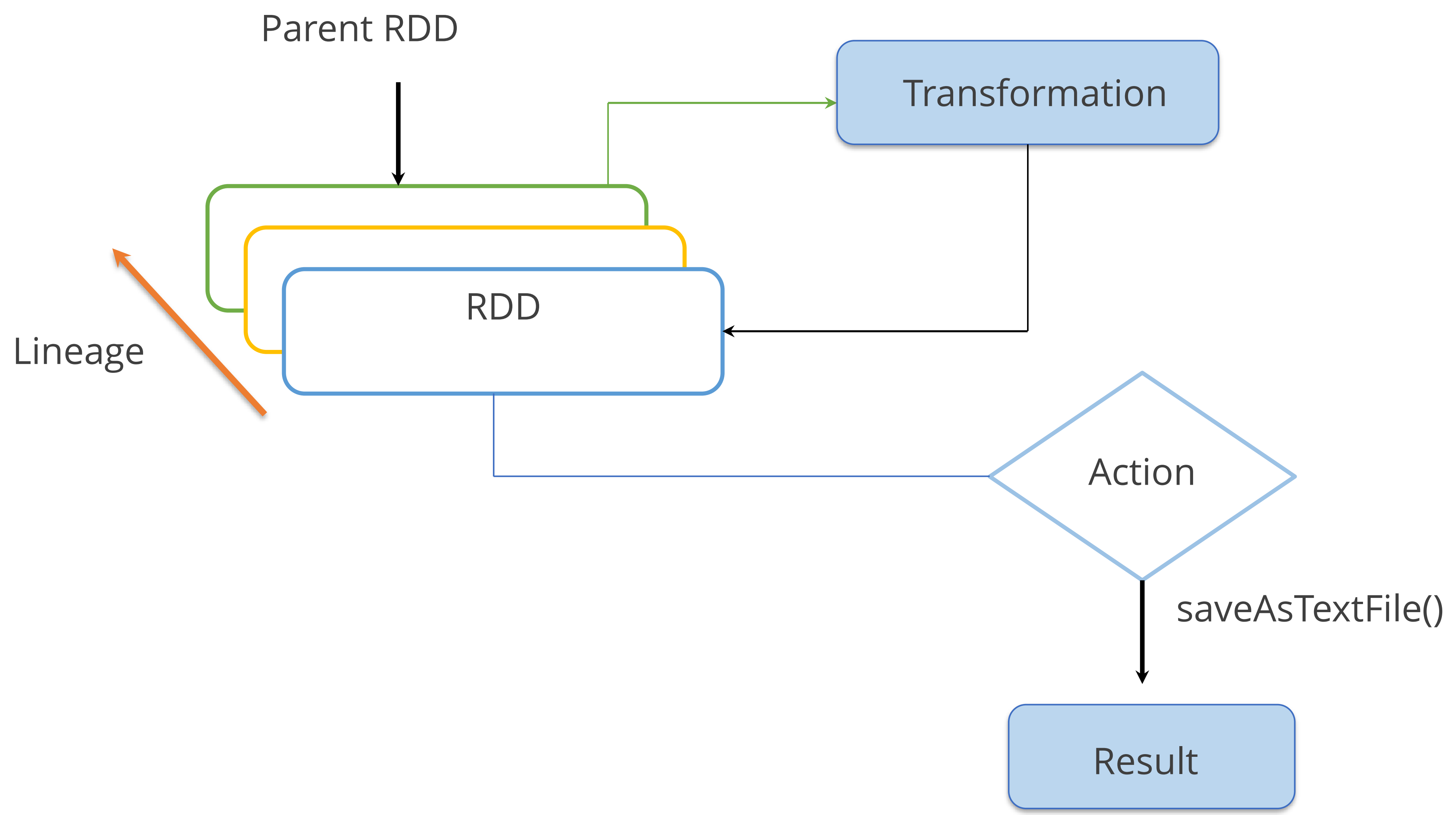
What Is RDD?

An RDD is a collection of objects that are distributed across nodes in a cluster. RDD supports lazy evaluation.



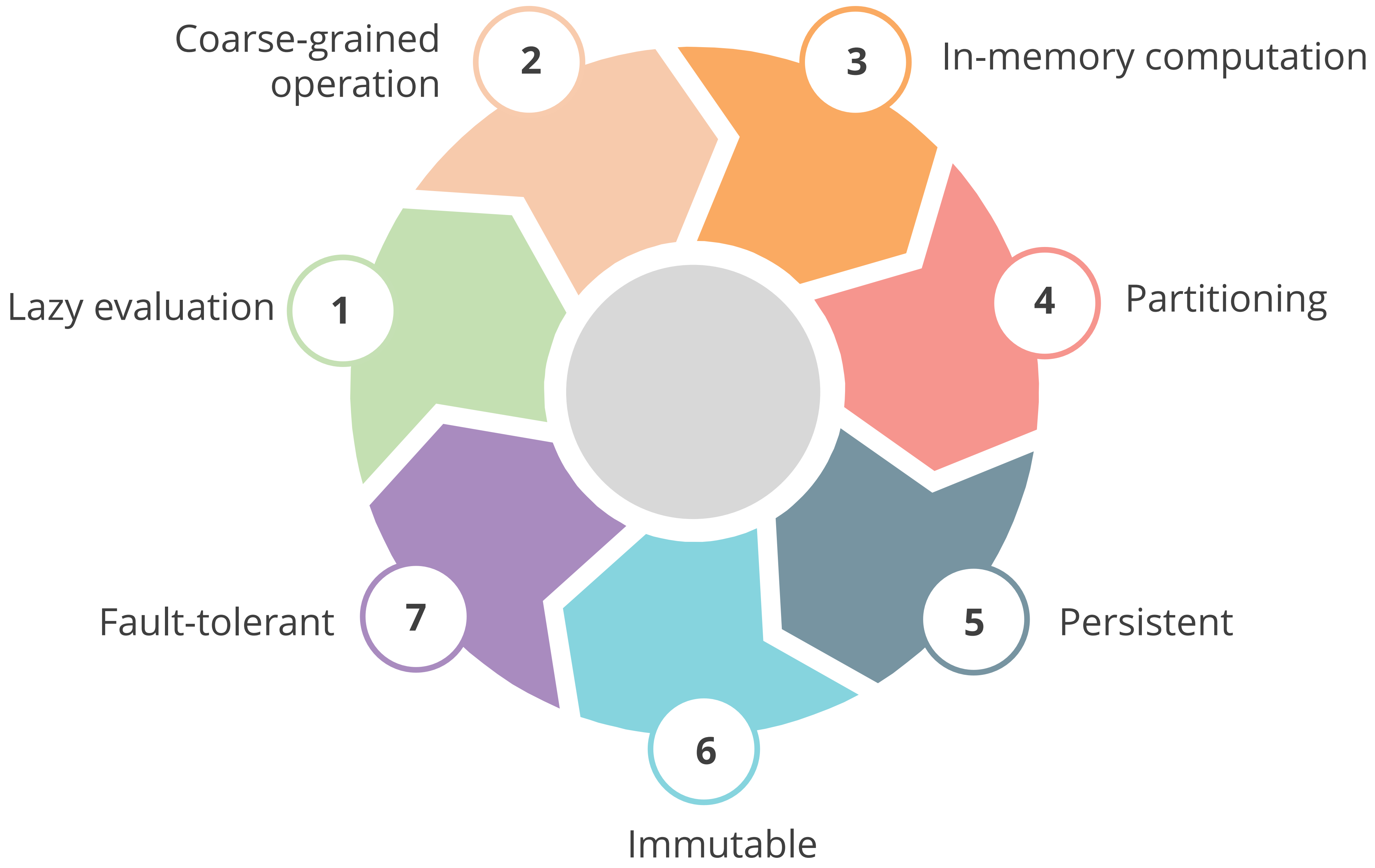
Lazy Evaluation

Lazy evaluation in Spark is a mechanism where the execution of an action will not begin until the action is initiated. Lazy evaluation occurs during Spark transformation.



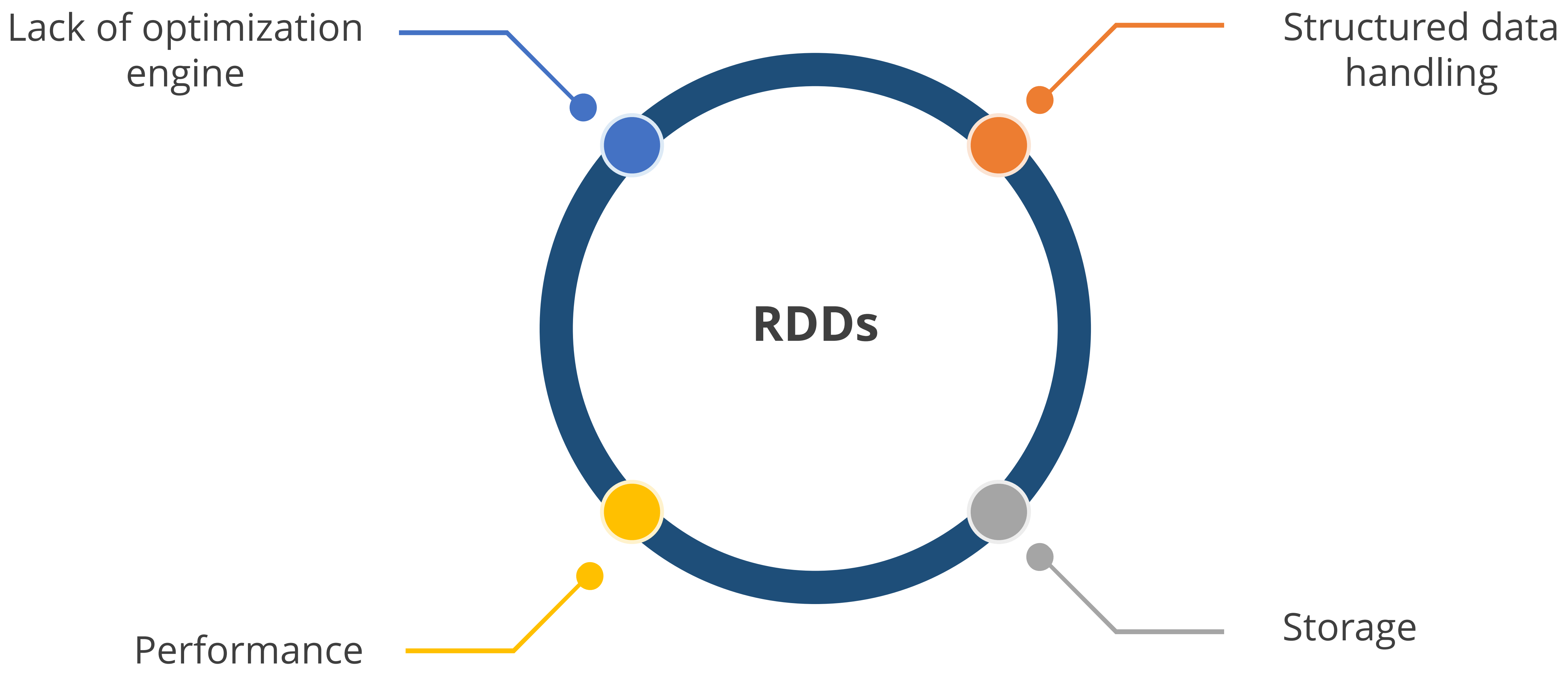
Features of RDD

Following are the features of Spark RDD:



Limitations of RDD

Following are the limitations of Spark RDD:



Different Ways to Create Spark RDD

Spark RDD can be created in a variety of methods.

Methods	Ways to use the method	Example
Text File	File or set of files Local or Distributed	textFile = sc.textFile("README.md")
Parallelized Collection	Memory	input = sc.parallelize((1, 2, 3, 4))
From existing RDD	Another RDD	newRdd = input.map(lambda e : (e, 1))

Different Ways to Create Spark RDD

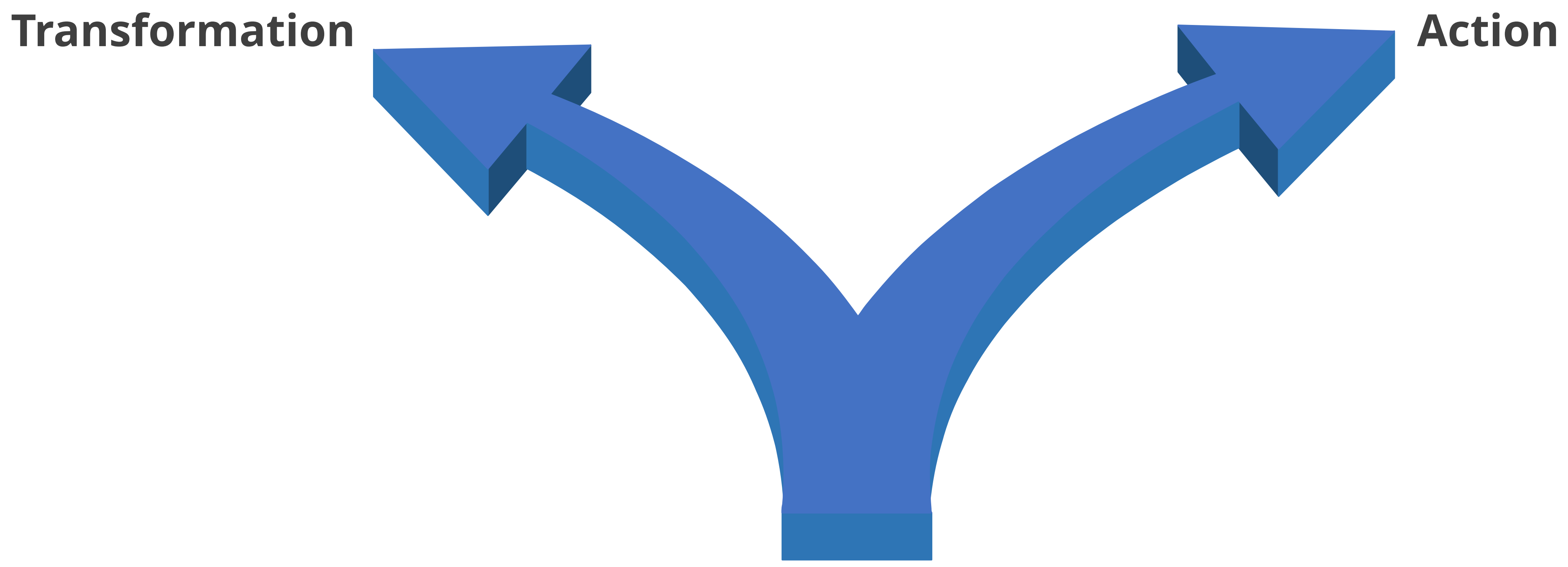
Spark RDD can be created in a variety of methods.

Methods	Ways to use the method	Example
From external datasets	Referencing a dataset in the external storage system	`data = spark.read \ .format("org.apache.spark.sql.cassandra")\ .options(table="test", keyspace="test") \ .load()`

RDD Operations

RDD Operations

RDD supports the following types of operations:



RDD Transformation

RDD Transformation

The operation performed on an RDD to create a new RDD is referred to as a transformation. The transformed RDD will execute only when an action is encountered.

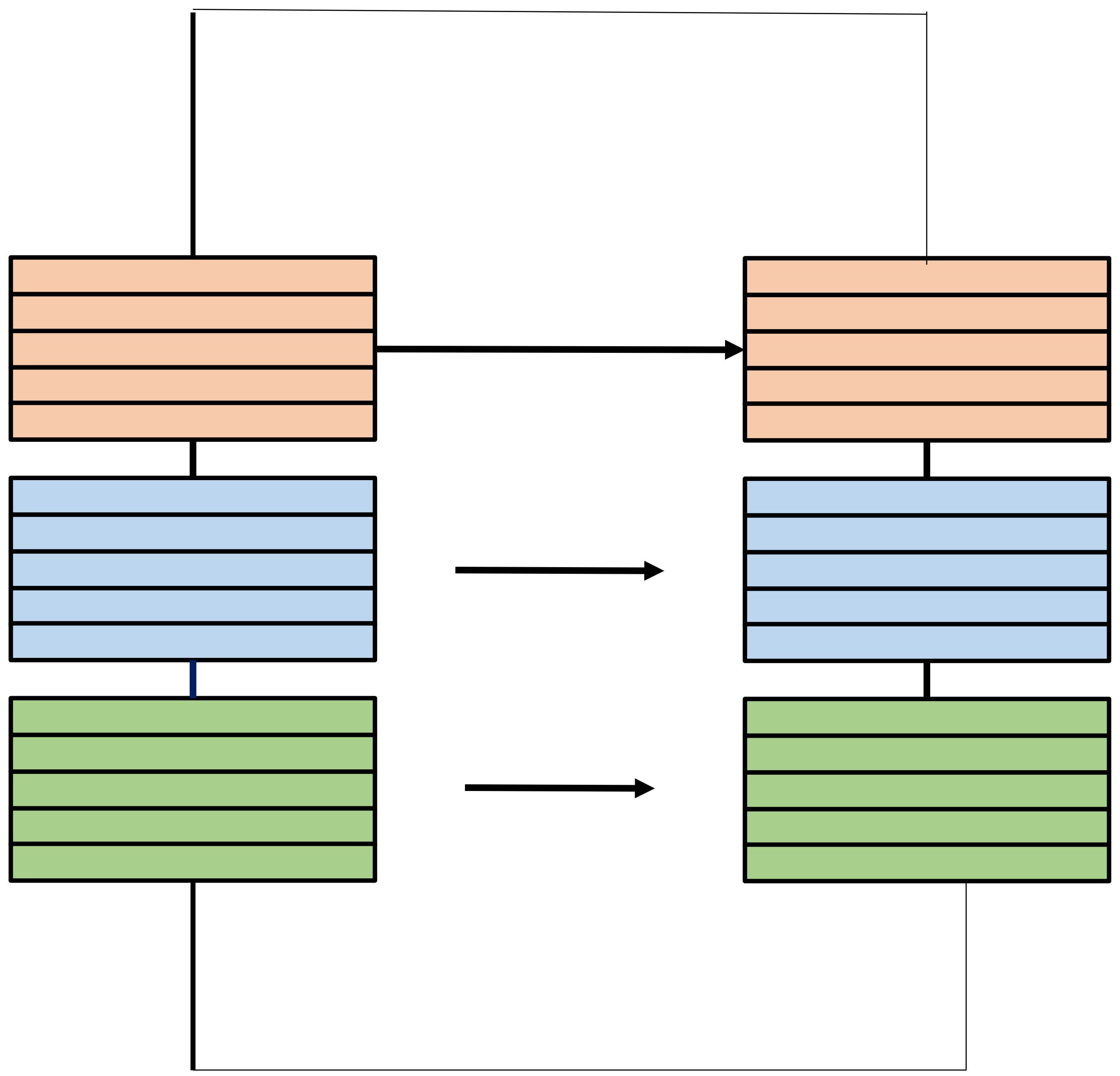
There are two types of transformation:

Narrow Transformation

Wide Transformation

Narrow Transformation

Narrow transformation is self-sufficient. It is the result of a map() and filter() functions, such that the data is from a single partition only.



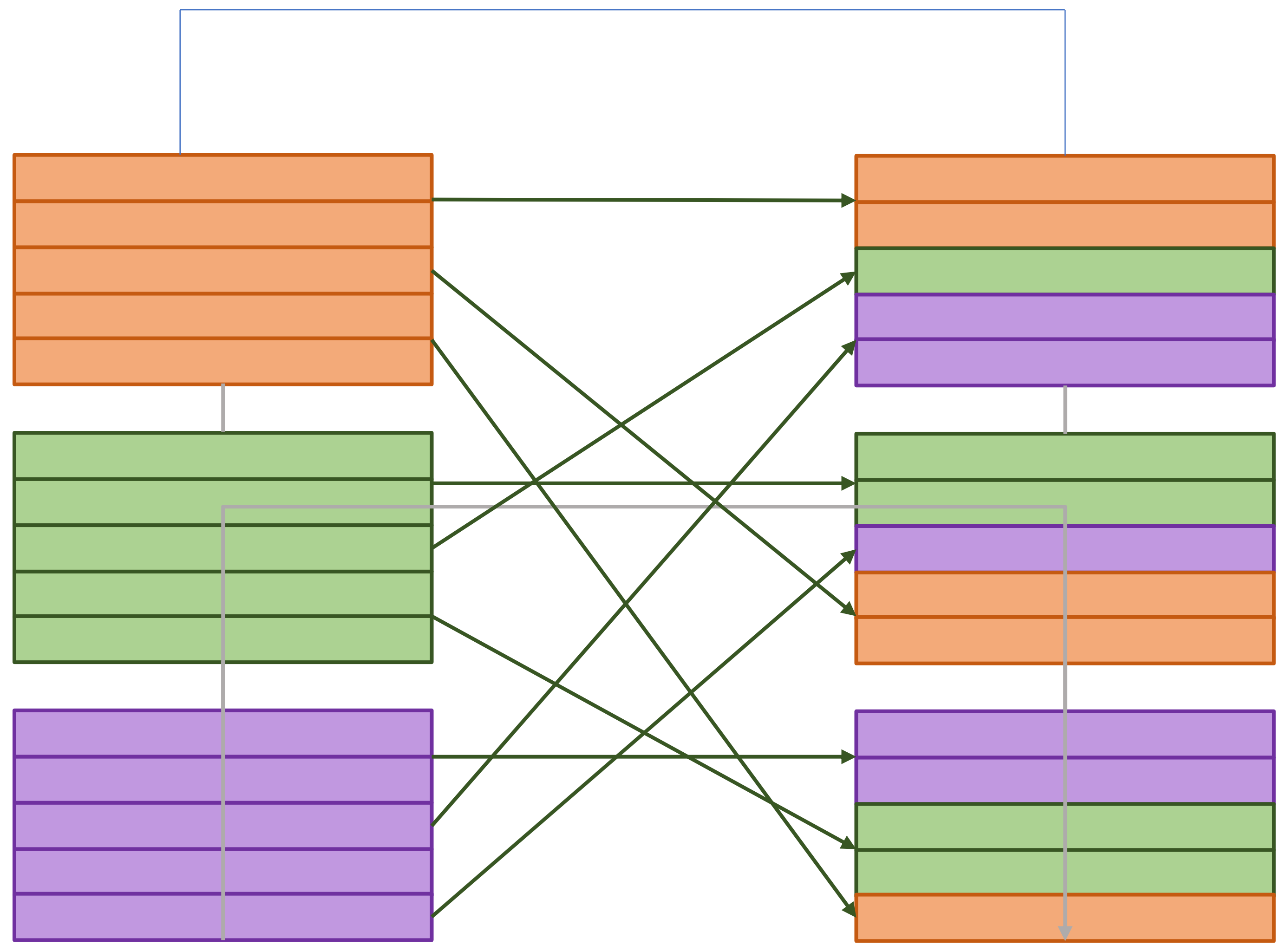
Narrow Transformation: Function

Narrow transformations include map(), mapPartition(), flatMap(), filter(), and union() functions.

- Map
- FlatMap
- MapPartition
- Filter
- Sample
- Union

Wide Transformation

Wide transformation is not self-sufficient. It is the result of GroupByKey() and ReduceByKey() functions, such that the data can be from multiple partitions.



Wide Transformation: Function

Wide transformations include groupByKey(), aggregateByKey(), aggregate(), join(), and repartition().

● **Cartesian**	● **ReduceByKey**
● **Join**	● **GroupByKey**
● **Intersection**	● **Coalesce**
● **Repartition**	● **Distinct**

Wider transformations are more expensive than narrow transformations because they require shuffling.

RDD Transformation: Examples

RDD Transformation: map()

It returns a new distributed dataset that is created by passing each element of the source through the function.

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
Sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

words = [“January", “February", “March", “April", “May", “June", “July"]
wordsRDD = sc.parallelize(words)
wordsRDD = wordsRDD.map(lambda word: (word, len(word)))

for word in wordsRDD.collect():
    print(word)
```

Output:

```
('January', 7)
('February', 8)
('March', 5)
('April', 5)
('May', 3)
('June', 4)
('July', 4)
```

RDD Transformation: flatMap()

Each input item can be mapped to 0 or more output items, similar to the map function (so the function returns a sequence rather than a single item).

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
Sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

words = ["Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday"]

wordsRDD = sc.parallelize(words)

wordsRDD = wordsRDD.flatMap(lambda word: word.split(",")) \
    .map(lambda word: (word, len(word)))

for word in wordsRDD.collect():
    print(word)
```

Output:

```
('Sunday', 6)
('Monday', 6)
('Tuesday', 7)
('Wednesday', 9)
('Thursday', 8)
('Friday', 6)
('Saturday', 8)
```

RDD Transformation: filter()

It returns a new dataset made up of the source elements for which the function returns true.

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

words = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

wordsRDD = sc.parallelize(words)

# Filter only even numbers.
wordsRDD = wordsRDD.filter(lambda word: word % 2 == 0)

for word in wordsRDD.collect():
    print(word)
```

Output:

```
2
4
6
8
10
```

RDD Transformation: reduceByKey()

It returns a dataset of key-value pairs for which the values for each key have been aggregated using the reduce function func.

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
     .builder \
     .appName("Simplilearn Examples") \
     .getOrCreate() \
     .sparkContext

words = ["cat", "cat", "dog", "rat", "cat", "cat", "dog"]

wordsRDD = sc.parallelize(words)

wordsRDD = wordsRDD.map(lambda word: (word, 1)) \
     .reduceByKey(lambda x, y: x + y)

for word in wordsRDD.collect():
    print(word)
```

Output:

```
('rat', 1)
('dog', 2)
('cat', 4)
```

RDD Transformation: groupBy()

It groups the data in the original RDD. It generates a set of key-value pairs with the key representing the output of a user function and the value representing all items for which the function returns this key.

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

words = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"]

wordsRDD = sc.parallelize(words)

# Group all words with same length.
wordsRDD = wordsRDD.groupBy(lambda word: len(word))

for word in wordsRDD.collect():
    print(word[0], list(word[1]))
```

Output:

```
8 ['Thursday', 'Saturday']
9 ['Wednesday']
6 ['Sunday', 'Monday', 'Friday']
7 ['Tuesday']
```

RDD Transformation: groupByKey()

It is used to group the values for each key in the original RDD. It creates a new pair in which the original key corresponds to the collected group of values.

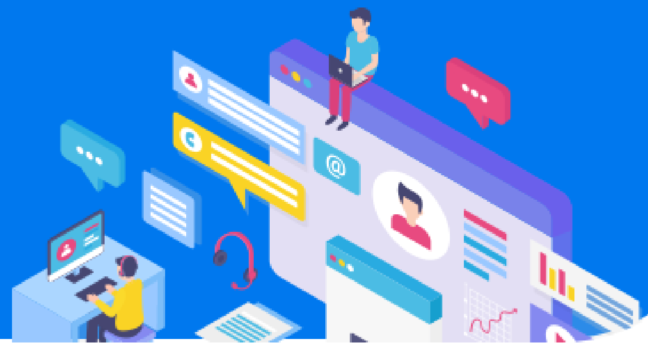
Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

words = [("a", 1), ("b", 1), ("a", 1)]

wordsRDD = sc.parallelize(words)

wordsRDD = wordsRDD.groupByKey() \
    .mapValues(list)

for word in wordsRDD.collect():
    print(word)
```

Output:

```
('a', [1, 1])
('b', [1])
```

Assisted Practice 15.1: RDD Partitions Using Coalesce Transformation



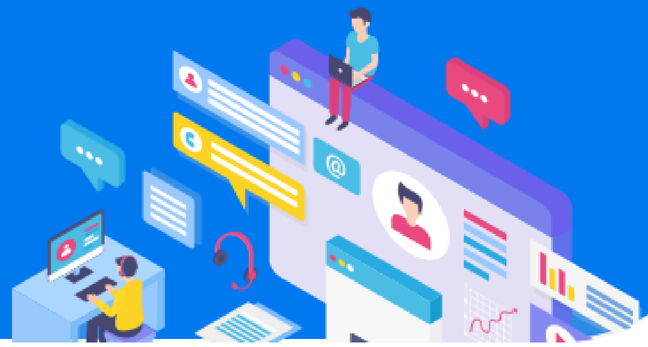
Duration: 10 mins

ASSISTED PRACTICE

Problem Scenario: Create an RDD to check the number of partitions after applying the coalesce transformation

Objective: In this demonstration, you will create an RDD and perform coalesce transformation.

Assisted Practice 15.1: RDD Partitions Using Coalesce Transformation



Duration: 10 mins

Tasks to Perform:

Step 1: Login into the **"console"** and open the PySpark shell

Step 2: Import the required libraries and create a Spark Session

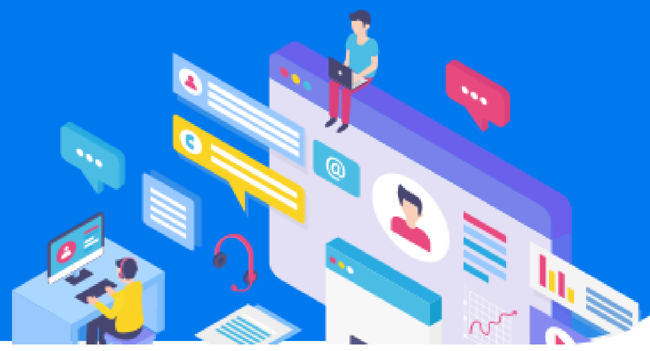
Step 3: Create an RDD using parallelize method and define eight different values

Step 4: Check the number of partitions using getNumpartitions function and then use coalesce function over the given RDD

Note: The solution to this assisted practice is provided under the Reference Materials section.

ASSISTED PRACTICE

Unassisted Practice 15.2: Spark Transformation Exploration



Duration: 10 mins

Problem Scenario: Create and explore Spark transformation examples.

Objective: In this demonstration, you will create map, flatMap, filter, reduceByKey, and groupBY

Tasks to Perform:

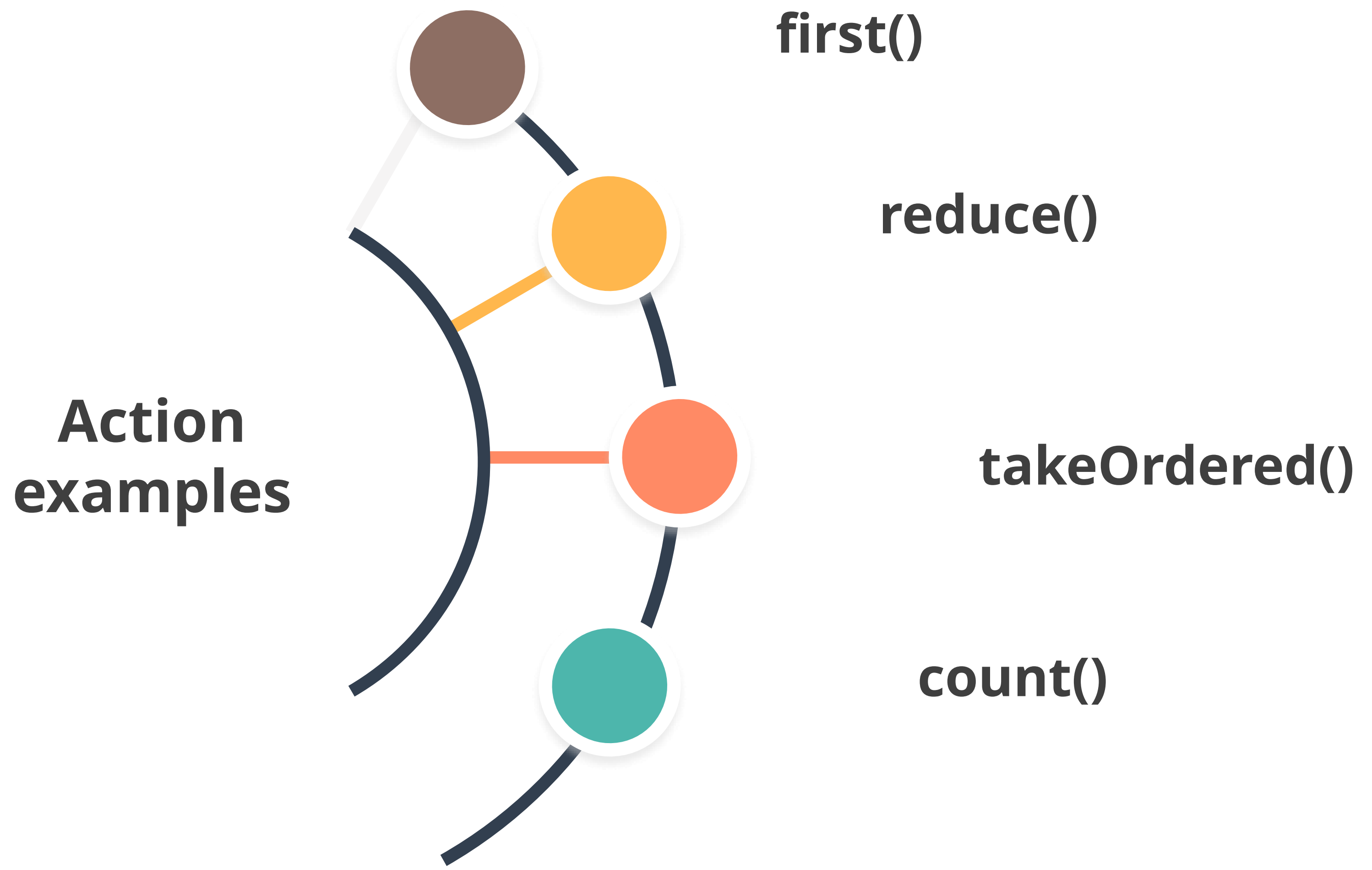
Step 1: Login into the **“webconsole”** and open the PySpark shell

Step 2: Use different transformation functions such as map, flatMap, filter, reduceByKey, and groupBy

RDD Action

RDD Action

Actions is an RDD operation that instructs Spark to perform computations and return the results back to the driver. It will be executed only when an action is encountered.



RDD Action: Examples

RDD Action: first()

It returns the dataset's first element (equivalent to take(1)).

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
     .builder \
     .appName("Simplilearn Examples") \
     .getOrCreate() \
     .sparkContext

words = [("a", 1), ("b", 1), ("a", 1)]

wordsRDD = sc.parallelize(words)

print(wordsRDD.first())
```

Output:

```
('a’, 1)
```

RDD Action: reduce()

It accepts two arguments and returns one and aggregates all the elements of the dataset.

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
     .builder \
     .appName("Simplilearn Examples") \
     .getOrCreate() \
     .sparkContext

dataRDD = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

reduceRDD = dataRDD.reduce(lambda x, y: x + y)

print(reduceRDD)
```

Output:

```
55
```

RDD Action: takeOrdered()

It returns the first n elements of the RDD in their natural order or using a custom comparator.

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

dataRDD = sc.parallelize(["cat", "rat", "bat", "dog",
"elephant"])

print(dataRDD.takeOrdered(3))
```

Output:

```
['bat', 'cat', 'dog']
```

RDD Action: count()

It returns the number of elements present in the dataset.

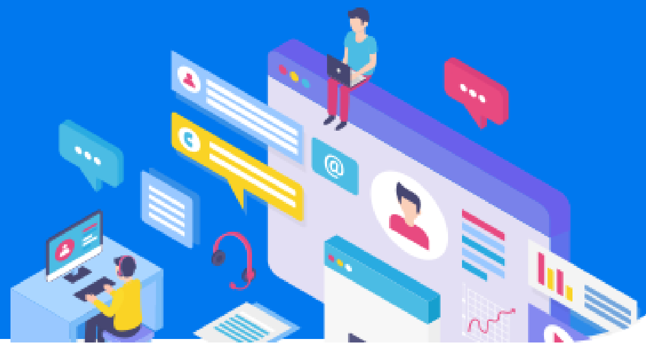
Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

dataRDD = sc.parallelize([11, 20, 33, 4, 5, 6 ])

print(dataRDD.count())
```

Output:

```
6
```

Unassisted Practice 15.3: Spark Action Exploration



Duration: 10 mins

Problem Statement: Create and explore Spark Action examples.

Objective: In this demonstration, you will work with different actions such as count, first, reduce, and takeOrdered

Tasks to Perform:

Step 1: Login into the **“console”** and open the PySpark shell

Step 2: Use different actions such as count, first, reduce, and takeOrdered

Loading and Saving Data into an RDD

Data Loading in RDD

The simplest way of loading data into an RDD is the SparkContext.parallelize function. SparkContext provides the parallelize function, which allows Spark to distribute the data across multiple nodes instead of depending on a single node to process the data.

Code Snippet:

```
dataRDD = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Data Loading in RDD

CSV file can also be used to load data into an RDD.

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

fileRDD =
sc.textFile("/user/testdemomay1301mailinator/data-
files/vehicles.csv")

for x in fileRDD.collect():
    print(x)
```

Output:

```
name,year,selling_price,km_driven,fuel,seller_type,transmis
sion,owner
Maruti 800
AC,2007,60000,70000,Petrol,Individual,Manual,First Owner
Maruti Wagon R LXI
Minor,2007,135000,50000,Petrol,Individual,Manual,First
Owner
Hyundai Verna 1.6
SX,2012,600000,100000,Diesel,Individual,Manual,First Owner
Datsun RediGO T
Option,2017,250000,46000,Petrol,Individual,Manual,First
Owner
Honda Amaze VX i-
DTEC,2014,450000,141000,Diesel,Individual,Manual,Second
Owner
```

Data Loading in RDD

Text file can also be used to load data into an RDD.

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

fileRDD = sc.textFile("/user/testdemomay1301mailinator/
mapreducedemo/wordcount.txt")

for x in fileRDD.collect():
    print(x)
```

Output:

```
Apache Spark is an open-source cluster-computing framework.
Originally developed at the University of California,
Berkeley's AMPLab, the Spark codebase was later d
onated to the Apache Software Foundation, which has
maintained it since. Spark provides an interface for
programming entire clusters with implicit data parallelism
 and fault tolerance.
```

Saving Data Using RDD

There are multiple methods present in an RDD class to save data, in multiple partitions on the disk.

Syntax:

```
# Saving a text file.
fileRDD.saveAsTextFile("/path/")

# Saving file as a sequence file.
fileRDD.saveAsSequenceFile("/path/")
```

Saving Data Using RDD

Multiple methods are available to save data into an RDD class.

1. saveAsHadoopDataset()
2. saveAsNewAPIHadoopDataset()
3. saveAsNewAPIHadoopFile()
4. saveAsPickleFile()

Pair RDDs

What Is Pair RDD?

A pair RDD can be created by calling the map() function over an RDD that returns a key-value pair.

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
     .builder \
     .appName("Simplilearn Examples") \
     .getOrCreate() \
     .sparkContext
words = [("a", 1), ("b", 1), ("a", 1)]

wordsRDD = sc.parallelize(words)

wordsRDD = wordsRDD.groupByKey() \
     .mapValues(list)

for word in wordsRDD.collect():
    print(word)
```

Output:

```
('a', [1, 1])
('b', [1])
```

Collect values for the same key in an array

Pair RDD: Example

Example:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Create Spark Session.
sc = SparkContext = SparkSession \
     .builder \
     .appName("Simplilearn Examples") \
     .getOrCreate() \
     .sparkContext
# Python collection containing words.
words = ["cat", "cat", "dog", "rat", "cat", "cat",
"dog"]

# Converting the collection to RDD
wordsRDD = sc.parallelize(words)

# Using map and reduceByKey function to calculate
word count.
wordsRDD = wordsRDD \
     .map(lambda word: (word, 1)) \
     .reduceByKey(lambda x, y: x + y)
for word in wordsRDD.collect():
    print(word)
```

Output:

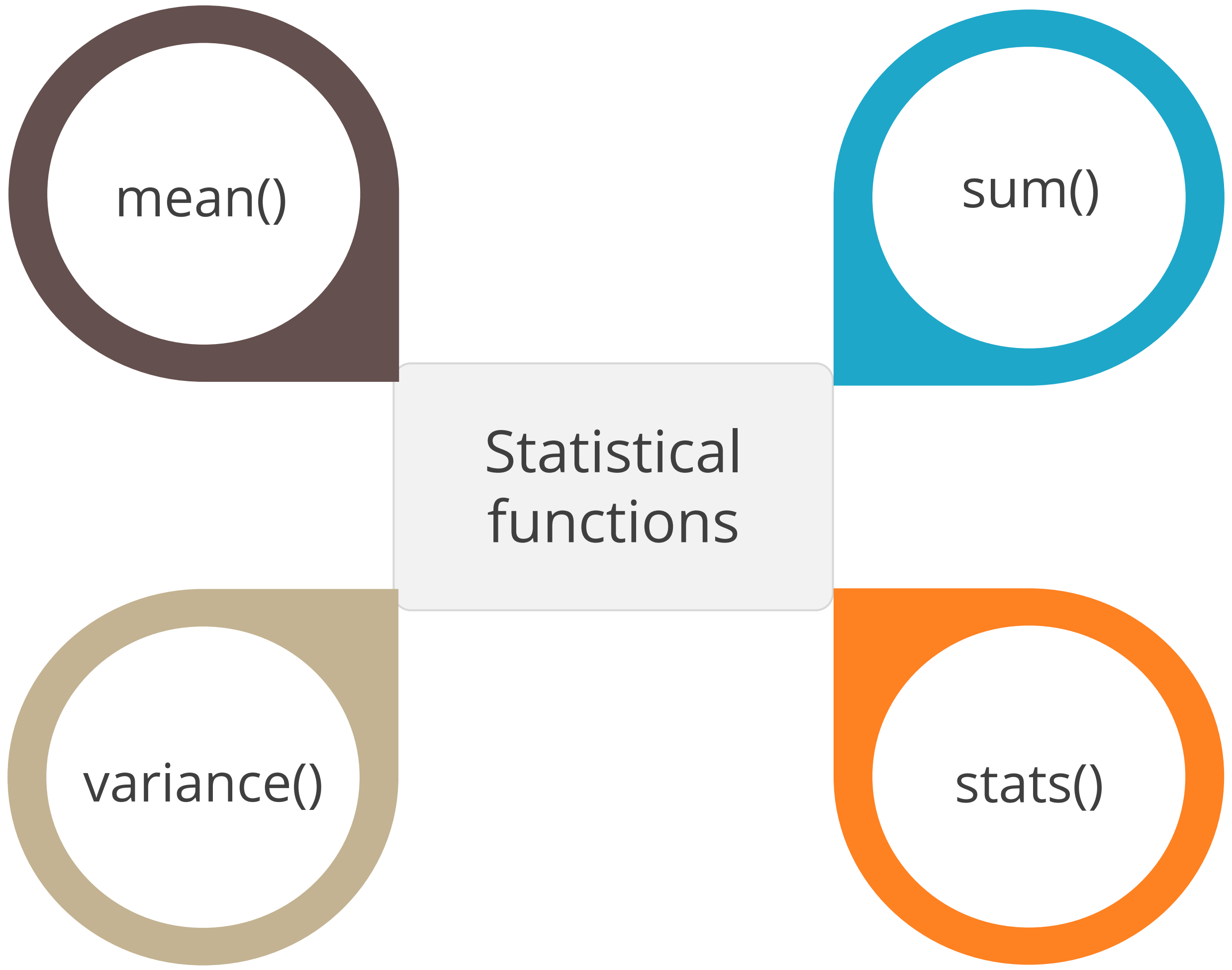
```
('rat', 1)
('dog', 2)
('cat', 4)
```

Collect values for the same key in an array

Double RDD and Its Functions

Double RDD

An RDD with a collection of double values is known as a DoubleRDD. Due to this characteristic, many statistical functions can be used with the DoubleRDD.



Double RDD Function: mean()

Mean is the total sum of values divided by the number of values in RDD. It is used to identify the mean value of the given RDD.

Example:

```
from pyspark.sql import SparkSession
from pyspark import SparkContext

sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

data = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9,
10])

print(data.mean())
```

Output:

```
5.5
```

Double RDD Function: sum()

It calculates the sum of all values present in the RDD.

Example:

```
from pyspark.sql import SparkSession
from pyspark import SparkContext

sc = SparkContext = SparkSession \
     .builder \
     .appName("Simplilearn Examples") \
     .getOrCreate() \
     .sparkContext

data = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9,
10])

print(data.sum())
```

Output:

```
55
```

Double RDD Function: variance()

It calculates the variance of all values present in the RDD.

Example:

```
from pyspark.sql import SparkSession
from pyspark import SparkContext

sc = SparkContext = SparkSession \
     .builder \
     .appName("Simplilearn Examples") \
     .getOrCreate() \
     .sparkContext

data = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9,
10])

print(data. variance())
```

Output:

```
8.25
```

Double RDD Function: stats()

stats() returns an instance of a multivariate statistical summary, which contains the count, mean, standard deviation, max, and min.

Example:

```
from pyspark.sql import SparkSession
from pyspark import SparkContext

sc = SparkContext = SparkSession \
    .builder \
    .appName("Simplilearn Examples") \
    .getOrCreate() \
    .sparkContext

data = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9,
10])

print(data. stats())
```

Output:

```
(count: 10, mean: 5.5, stdev:
2.8722813232690143, max: 10.0,
min: 1.0)
```

DAG and RDD Lineage

Directed Acyclic Graph

A DAG is a graphical representation of how Spark will execute the program. Each vertex on the graph represents a separate operation, and the edges represent the operation dependencies.

Directed

One node in a graph is directly connected to another, which creates a sequence.

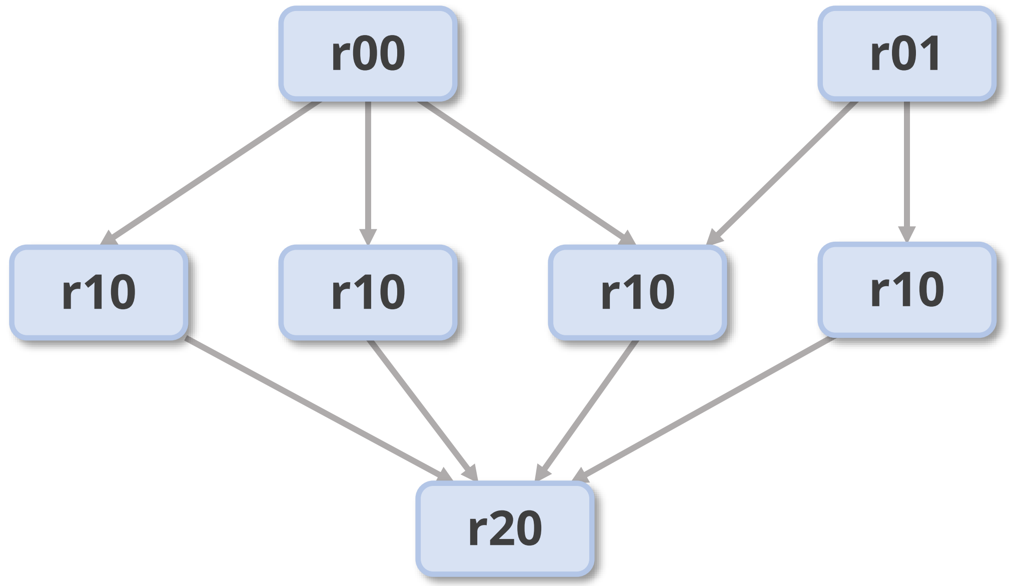
Acyclic

It defines that there is no cycle or loop available.

Graph

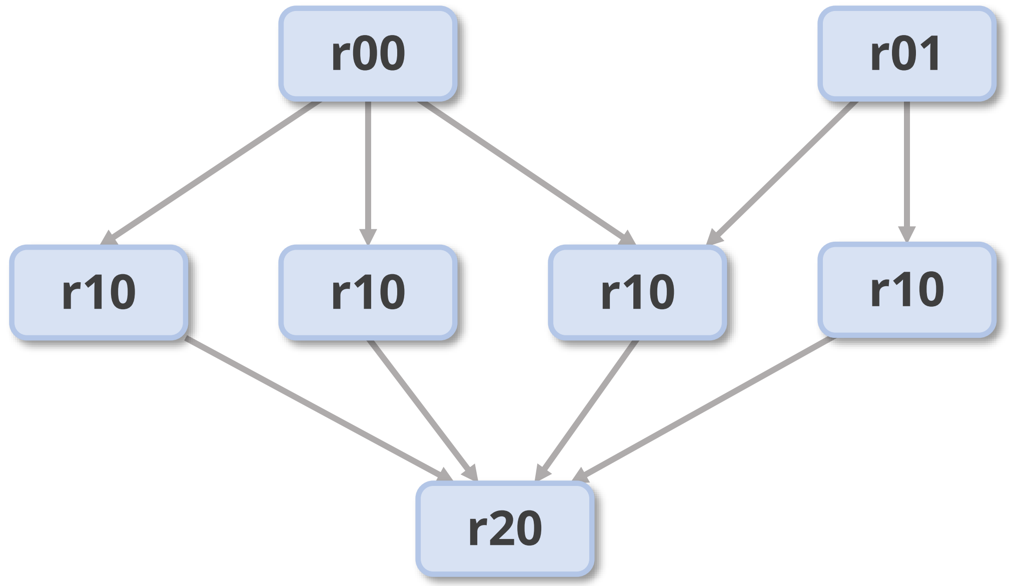
From graph theory, it is a combination of vertices and edges.

RDD Lineage



- RDD lineage (also known as RDD operator graph or RDD dependency graph) is a graph that represents all the parent RDD of an RDD.
- It is built as a result of applying transformations to the RDD and creates a logical execution plan.
- RDD supports lazy evaluation. It only creates a lineage when a transformation is called on top of it.
- It can operate on multiple entities (RDDs, DataFrames, and so on.)

RDD Lineage



- RDD Lineage is just a portion of a DAG (one or more operations) that lead to the creation of that RDD.
- One DAG can create multiple RDDs and each RDD will have its lineage i.e., that path in DAG that leads to that RDD.
- If some partitions of RDD get corrupted or lost, then Spark may return that part of the DAG that leads to the creation of those partitions.

RDD Persistence and Its Storage Levels

RDD Persistence

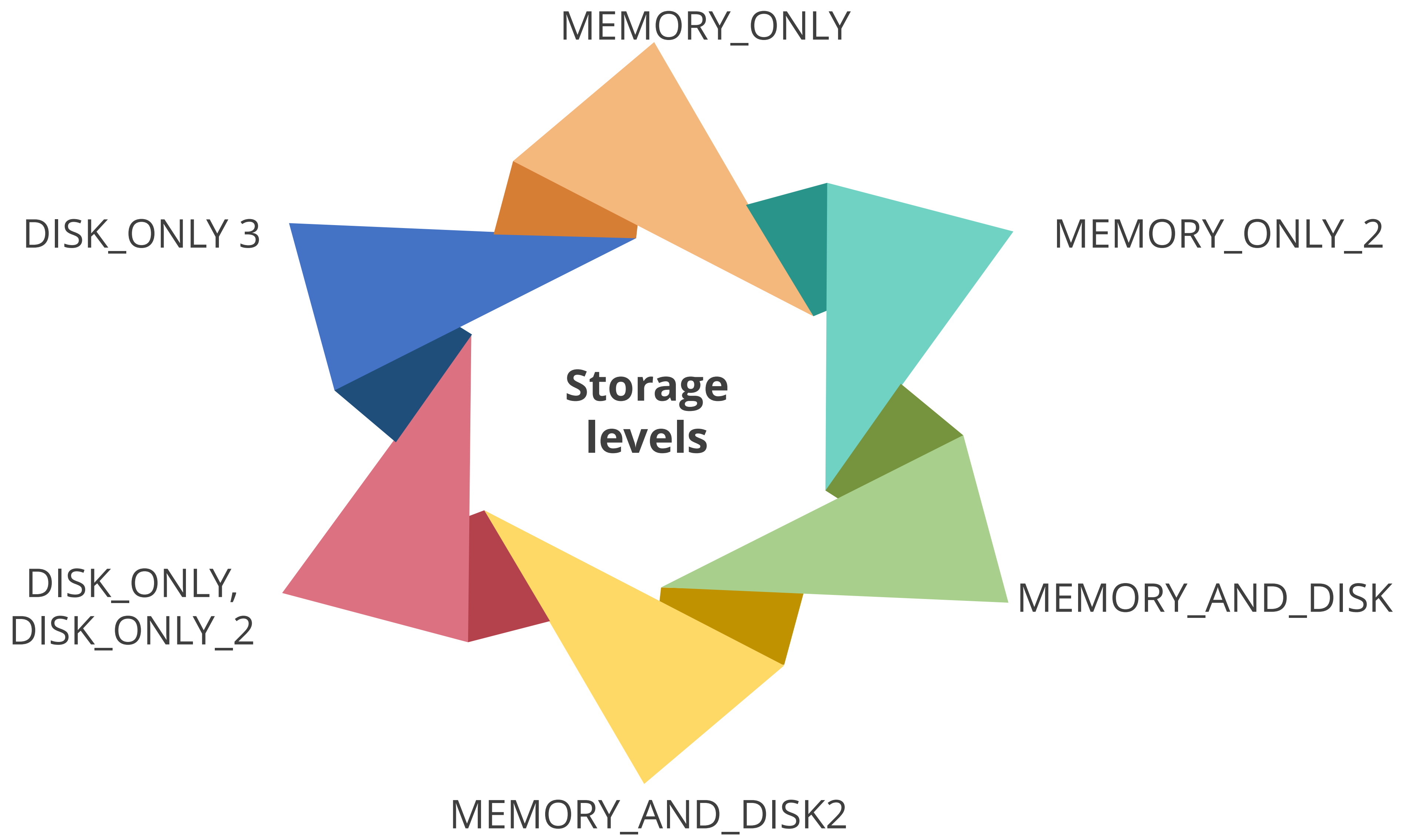
Code Snippet:

```
words = ["cat", "cat", "dog", "rat", "cat", "cat",
"dog"]

wordsRDD = sc.parallelize(words)

wordsRDD = wordsRDD.map(lambda word: (word, 1)) \
     .reduceByKey(lambda x, y: x + y)

wordsRDD.cache()
```

- Spark RDD is recomputed once it is materialized. Hence, it is good to persist RDD in memory.
- When too much data is cached in memory, Spark will automatically evict old partitions using LRU (Least Recently Used) cache policy.
- RDD provides method cache() or persist() to save data in memory.
- RDD provides unpersist() method to discard the persisted data.

RDD Persistence: Storage Levels

The storage level determines how and where to persist or cache a PySpark RDD, DataFrame, or Dataset. All these storage levels are passed as an argument to the persist() method of the PySpark RDD, DataFrame, and Dataset.



RDD Persistence: Storage Levels

MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they are needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that do not fit on disk, and read them from there when they are needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one-byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.

RDD Persistence: Storage Levels

DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2	Same as the levels above but replicate each partition on two cluster nodes.
OFF_HEAP	Similar to MEMORY_ONLY_SER but store the data in off-heap memory. This requires off-heap memory to be enabled.

Selection of Storage Level

The storage levels in Spark are designed to provide various trade-offs between memory utilization and CPU efficiency. It is suggested to follow these steps to choose one:

1. If RDD fits comfortably with the default storage level (MEMORY_ONLY), leave them as they are. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.

2. If it does not work, consider using MEMORY_ONLY_SER and a fast serialization library to make the objects much more space-efficient, but still reasonably fast to access.

Selection of Storage Level

3 If the functions that compute user datasets are expensive or filter a large quantity of data, do not spill to the disc. Otherwise, recomputing a partition could be as fast as reading it from the disc.

4 If a user needs fast fault recovery, use replicated storage levels. All storage levels provide full fault tolerance by recomputing lost data. However, replicated storage levels allow to continue running tasks on the RDD without waiting to recompute a lost partition.

Word Count Program

Word Count Program

Step 1: Import required package

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
```

Step 2: Create SparkContext

```
sc = SparkContext = SparkSession \
     .builder \
     .appName("Simplilearn Examples") \
     .getOrCreate() \
     .sparkContext
```

Word Count Program

Step 3: Create a Python collection using some words

```
words = ["cat", "cat", "dog", "rat", "cat", "cat",
"dog"]
```

Step 4: Convert collection into an RDD

```
wordsRDD = sc.parallelize(words)
```

Word Count Program

Step 5: Use map and reduceByKey to calculate word count

```
wordsRDD = wordsRDD \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda x, y: x + y)
```

Step 6: Cache the data in-memory

```
wordsRDD.cache()
```

Word Count Program

Step 7: Collect and print the data

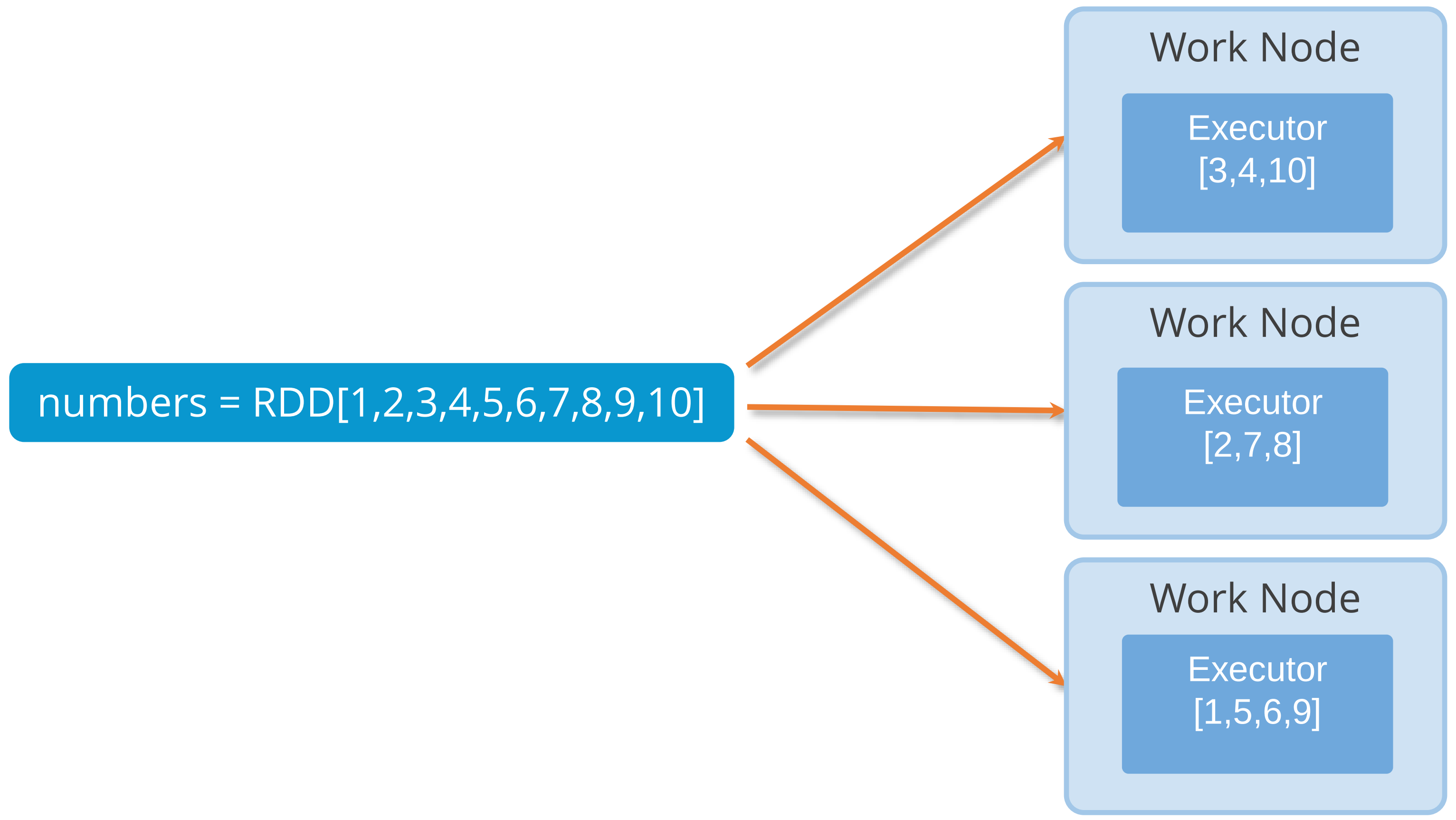
```
for word in wordsRDD.collect():
    print(word)
```

Output:

```
('rat', 1)
('dog', 2)
('cat', 4)
```

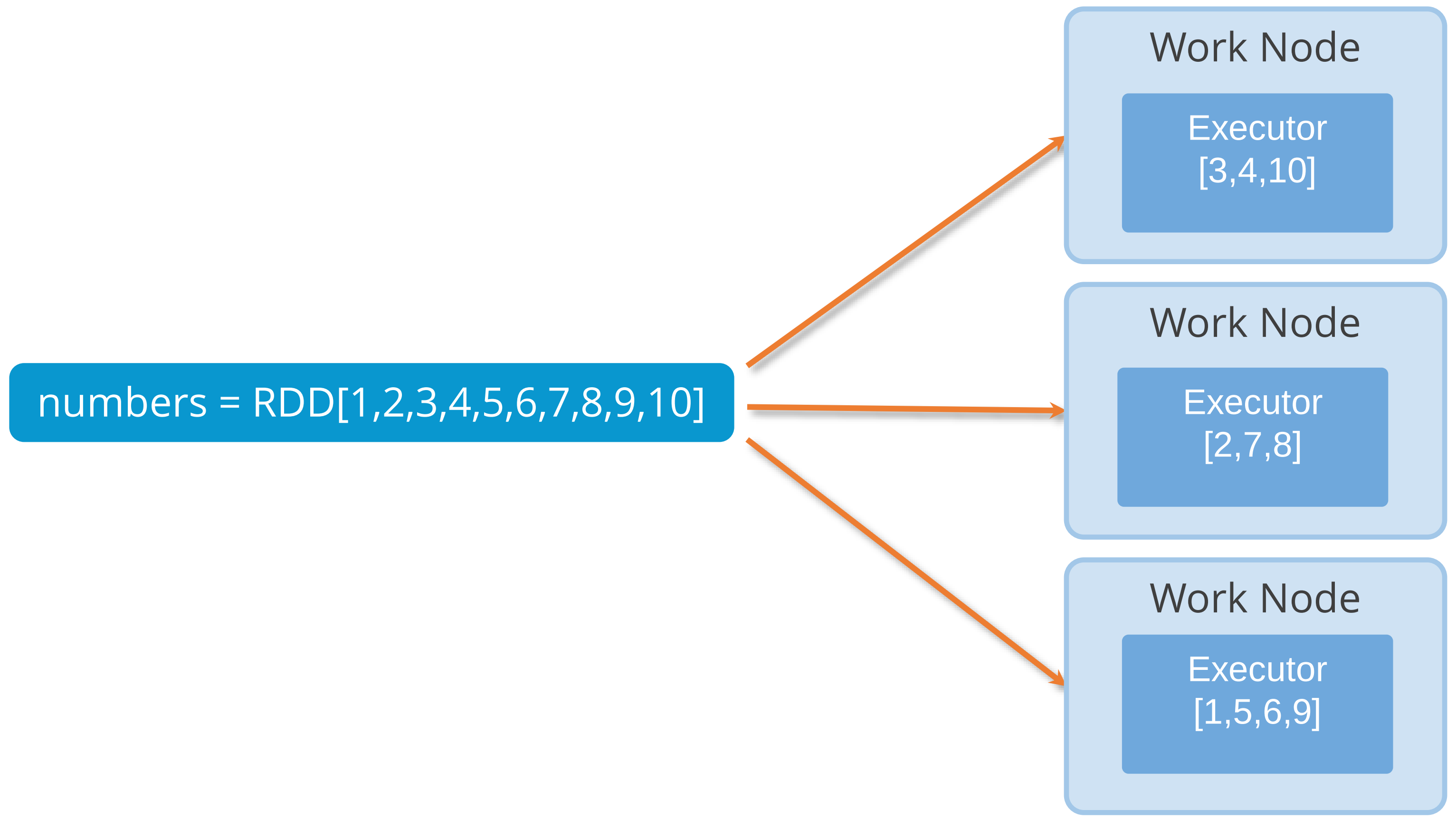
RDD Partitioning

RDD Partitioning



- RDD is partitioned across multiple worker nodes.
- Spark automatically partitions RDD and distributes the partitions across different nodes.
- RDD partitioning enables parallel processing.
- It assigns one task per partition, and each worker node can process one task at a time.

RDD Partitioning



- Every worker node may contain one or more partitions.
- The number of partitions is configurable. It can be changed using the below methods:

1. Repartition()
2. Coalesce()

RDD Partitioning

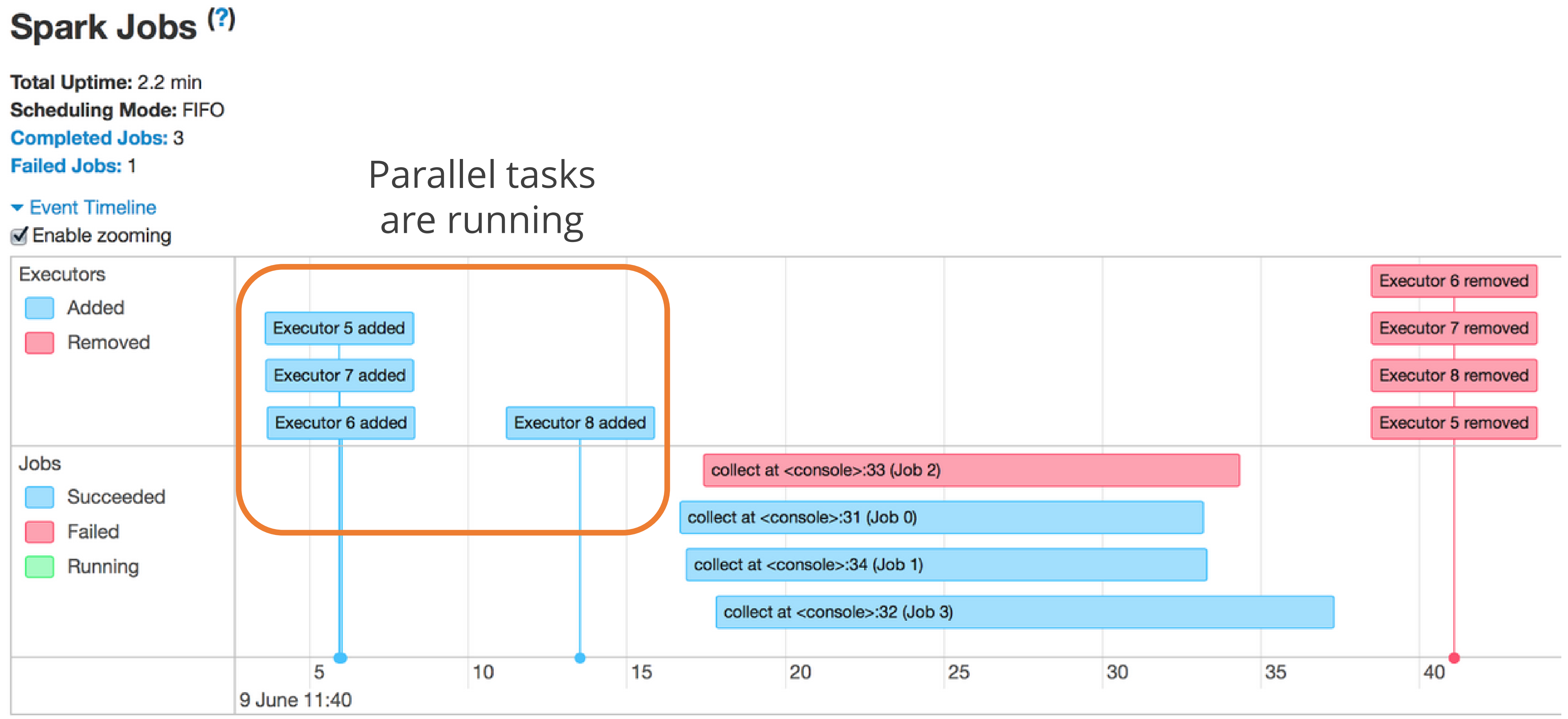
Apache Spark supports two types of RDD partitioning:

Hash partitioning

Range partitioning

RDD Partitioning and Parallelism

Spark creates a separate task for a partition, which enables parallel processing of data and optimizes the overall computation. To check the number of partitions of an RDD, the def getNumPartitions() method is used.

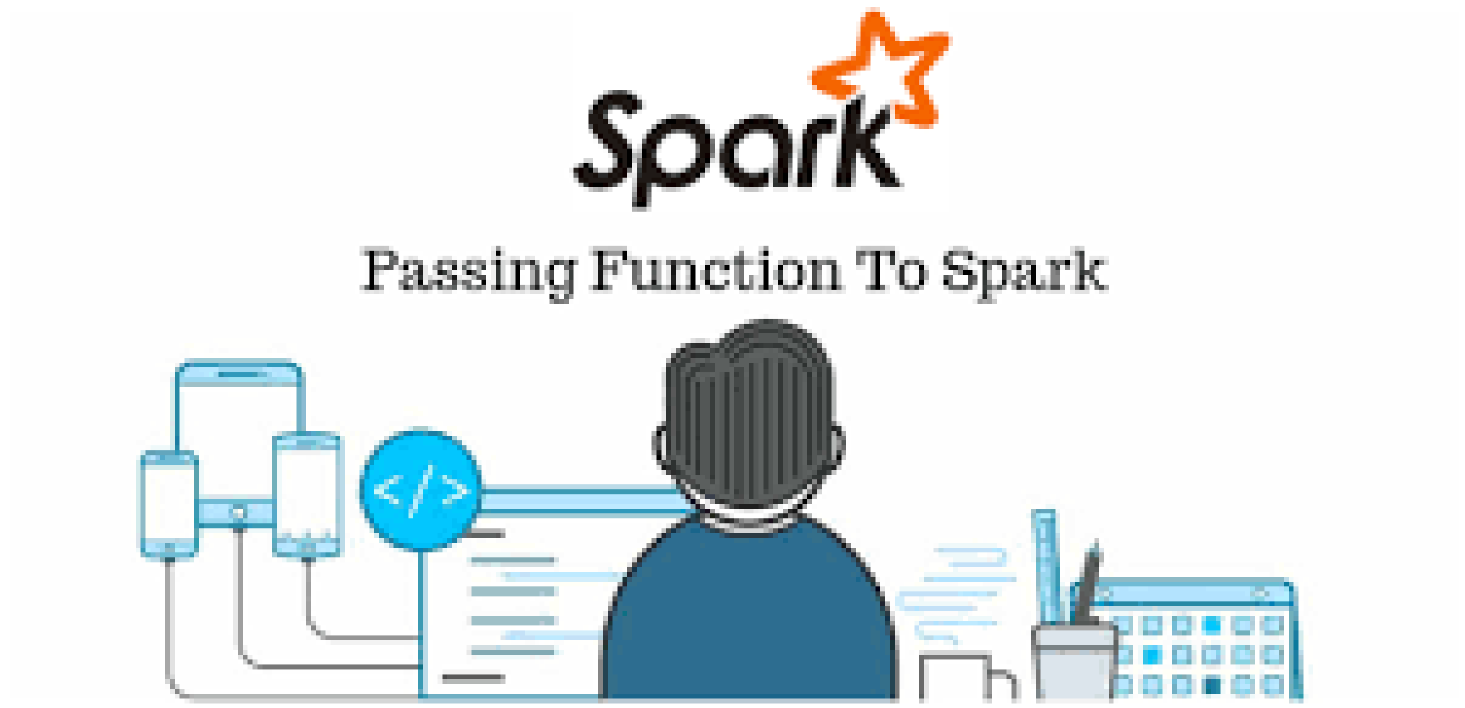

Spark Jobs (?)

Total Uptime: 2.2 min
Scheduling Mode: FIFO
Completed Jobs: 3
Failed Jobs: 1

▾ Event Timeline
☑ Enable zooming

Passing Function to Spark

Passing Function to Spark



- Spark API relies heavily on passing functions in the driver program to run on the cluster.
- In Spark, most of the transformations and some actions require some functions to be provided.
- Functions define the logic to transform the RDD elements.

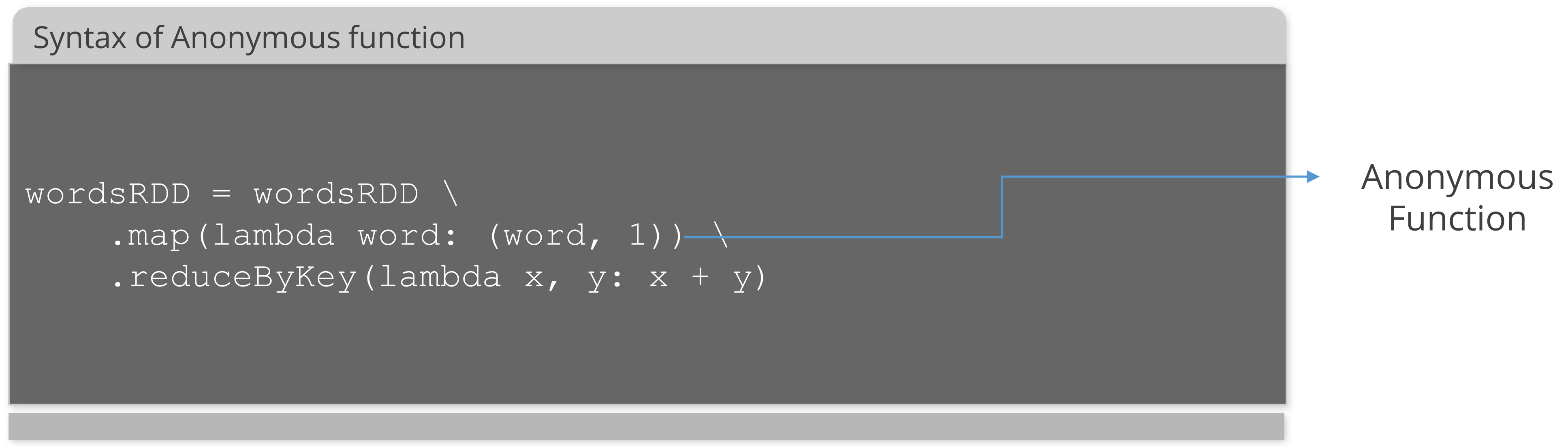
Passing Function to Spark

This can be done in one of three ways:

- Anonymous functions
- Local defined functions
- Top-level functions in a module

Anonymous Function

The anonymous function is a function that does not have any names associated with it. It can be defined using the lambda keyword, as shown below:
reduceByKey(lambda x, y: x + y)



```
wordsRDD = wordsRDD \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda x, y: x + y)
```

Here, the anonymous function is adding up all values for the same key.

Local Defined Function

This function can be defined in PySpark and then the same function can be passed as an argument to a transformation function.

Syntax of Local defined function

```
if __name__ == "__main__":
    def myFunc(s):
        words = s.split(" ")
        return len(words)

    sc = SparkContext(...)

    sc.textFile("file.txt").map(myFunc)
```

Here, myFunc() is a local-defined function that splits the input string by space (“ ”) and then returns the length of the output array.

Top-Level Function

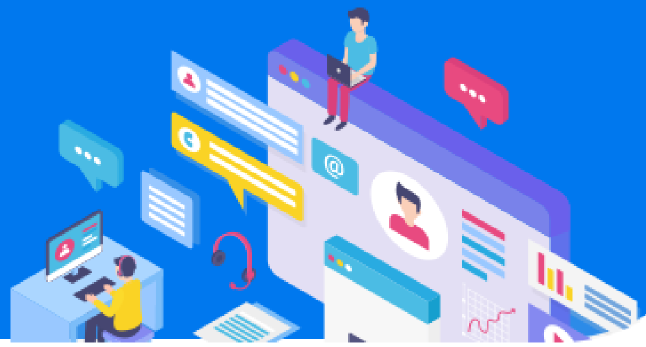
Top-level functions are the ones which are defined in a class.

Syntax of Top-level function in a module

```
class Demo(object):
    def func(self, s):
        return s

    def doStuff(self, rdd):
        return rdd.map(self.func)
```

func() and doStuff() in the class Demo are the Top-level functions.

Assisted Practice 15.4: Create an RDD in Spark



Duration: 10 mins

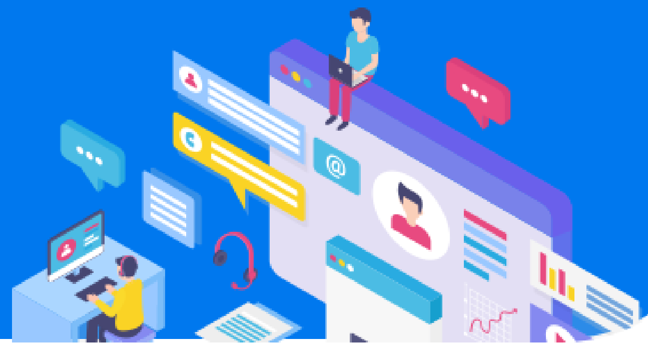
Problem Scenario: Create an RDD with a real-world retail business dataset of different categories

Objective: To read the data from HDFS and print the distinct categories.

Dataset Name: “part-m-00000”

ASSISTED PRACTICE

Assisted Practice 15.4: Create an RDD in Spark



Step 1: Download the **“part-m-00000”** dataset from the categories folder from the reference materials section and upload into the HDFS using **“Hue”**

Step 2: Login into the **“console”** and open the PySpark shell

Step 3: Create an RDD using textFile and update the path of the dataset

Step 4: Create a lambda function that will split the line

Step 5: Print each element using collect() method.

Note: The solution to this assisted practice is provided under the Reference Materials section.

Key Takeaways

- RDD is a collection of objects that are distributed across nodes in a cluster.
- The operation performed on an RDD to create a new RDD is referred to as transformation.
- An RDD with a collection of double values is known as a DoubleRDD.
- DAG is a graphical representation of how Spark will execute the program.
- Apache Spark supports hash and range partitioning.

Key Takeaways

- RDD provides cache() or persist() method to save data in memory.
- RDD class provides multiple methods to save data in multiple partitions on the disk.
- A lineage graph is a graph between an existing RDD and a new RDD. It means that all the dependencies between the RDDs will be recorded in a graph rather than their original data.

Knowledge Check

Knowledge Check

Which of the following describes the role of tasks in the Spark execution hierarchy?

A. Stages with narrow dependencies can be grouped into one task.

B. Tasks with wide dependencies can be grouped into one stage.

C. Within one task, the slots are the unit of work done for each partition of the data.

D. Tasks are the smallest element in the execution hierarchy.

Knowledge Check

Which of the following describes the role of tasks in the Spark execution hierarchy?

A. Stages with narrow dependencies can be grouped into one task.

B. Tasks with wide dependencies can be grouped into one stage

C. Within one task, the slots are the unit of work done for each partition of the data.

D. Tasks are the smallest element in the execution hierarchy.

The correct answer is **D**

Tasks are the smallest element in the execution hierarchy

Knowledge Check

2

Which of the following describes a narrow transformation?

A. A narrow transformation is an operation in which data is exchanged across partitions.

B. A narrow transformation is an operation in which data is exchanged across the cluster.

C. A narrow transformation is a process in which 32-bit float variables are cast into smaller float variables like 16-bit or 8-bit float variables.

D. A narrow transformation is an operation in which no data is exchanged across the cluster.

Knowledge Check

2

Which of the following describes a narrow transformation?

A. A narrow transformation is an operation in which data is exchanged across partitions.

B. A narrow transformation is an operation in which data is exchanged across the cluster.

C. A narrow transformation is a process in which 32-bit float variables are cast into smaller float variables like 16-bit or 8-bit float variables.

D. A narrow transformation is an operation in which no data is exchanged across the cluster.

The correct answer is **D**

In narrow transformation, no data is exchanged across the cluster, since these transformations do not require any data from outside of the partition they are applied on. It includes filter, drop, and coalesce.

Knowledge Check

3

Which of the following statements is correct about the difference between action and transformation?

A. Action triggers the actual execution, while transformation cannot.

B. Actions can be queued for delayed execution, while transformations can only be processed immediately.

C. Actions are lazy in nature, whereas transformations are not.

D. Actions do not send results to the driver, while transformations do.

Knowledge Check

3

Which of the following statements is correct about the difference between action and transformation?

A. Action triggers the actual execution, while transformation cannot.

B. Actions can be queued for delayed execution, while transformations can only be processed immediately.

C. Actions are lazy in nature, whereas transformations are not.

D. Actions do not send results to the driver, while transformations do.

The correct answer is **A**

Transformations are lazy evaluated in Spark. That means when we call transformation, nothing happens. The actual work happens when an action is encountered in Spark.

Knowledge Check

4

Which of the following statements are correct about the executor?

A. An executor can serve multiple applications

B. Executors stop by default on application completion.

C. Executors store data in memory only

D. Executors are launched by the driver

Knowledge Check

4

Which of the following statements are correct about the executor?

A. An executor can serve multiple applications.

B. Executors stop by default on application completion

C. Executors store data in memory only

D. Executors are launched by the driver.

The correct answer is **A,B**

An executor can serve multiple submitted jobs and executors are stopped when a job completes.

Lesson-End Project: Telecom Log Parsing

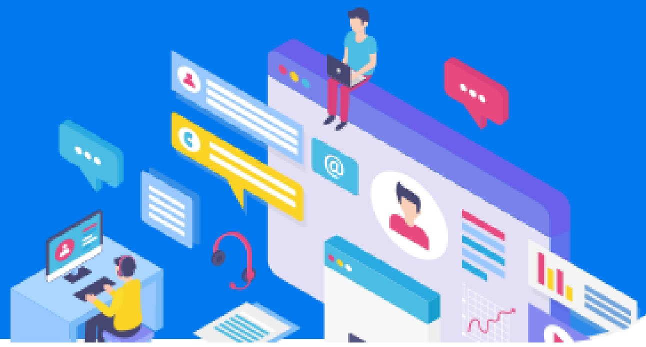
Problem Scenario:
A telecom software provider, Ericsson, is building an application to analyze different components in the production environment. For monitoring purposes, the application relies on log files parsing and looking for potential warnings or exceptions in the logs and reporting them. As a Spark developer, you are required to work on the input dataset containing the log files from different components and use it in the overall telecom application.

Objective:
The objective is to analyze the log files in the PySpark shell and perform some operations to find the 404 HTTP codes.

Dataset Name: mobile-input-data.csv, Access.log

LESSON-END PROJECT

Tasks to Perform



1. Download the dataset “**mobile-input-data.csv“** and **“access.log”** file from the reference materials section
2. Create a directory in HUE named **“data-files”** and upload the **“mobile-input-data.csv”**dataset
3. Create a new directory named **“apache”** and upload the **“access.log”** file into the directory
4. Open the PySpark shell in “**console”**
5. Create an RDD to read the log data in PySpark Shell
6. Read the CSV file from HDFS in PySpark shell
7. Create a lambda function to find the number of404 HTTP codes present in **“access.log”**

ASSISTED PRACTICE

Thank You