BIG DATA ANALYTICS

Parallel Programming with Spark

Spark Concepts

Parallel Programming with Spark: *Outline*

- Overview of Spark
- Fundamentals of Scala and Functional Programming
- Spark Concepts
  - Resilient Distributed Datasets (RDD)
  - Creating RDDs
  - Basic Transformations
  - Basic Actions
  - Word Count Example
- Spark operations
- Job execution
- Spark Applications : Cluster computing with working sets

Spark Concepts

- Spark Concepts
  - Resilient Distributed Datasets (RDD)
  - Creating RDDs
  - Basic Transformations
  - Basic Actions
  - Word Count Example

Resilient Distributed Dataset (RDD)

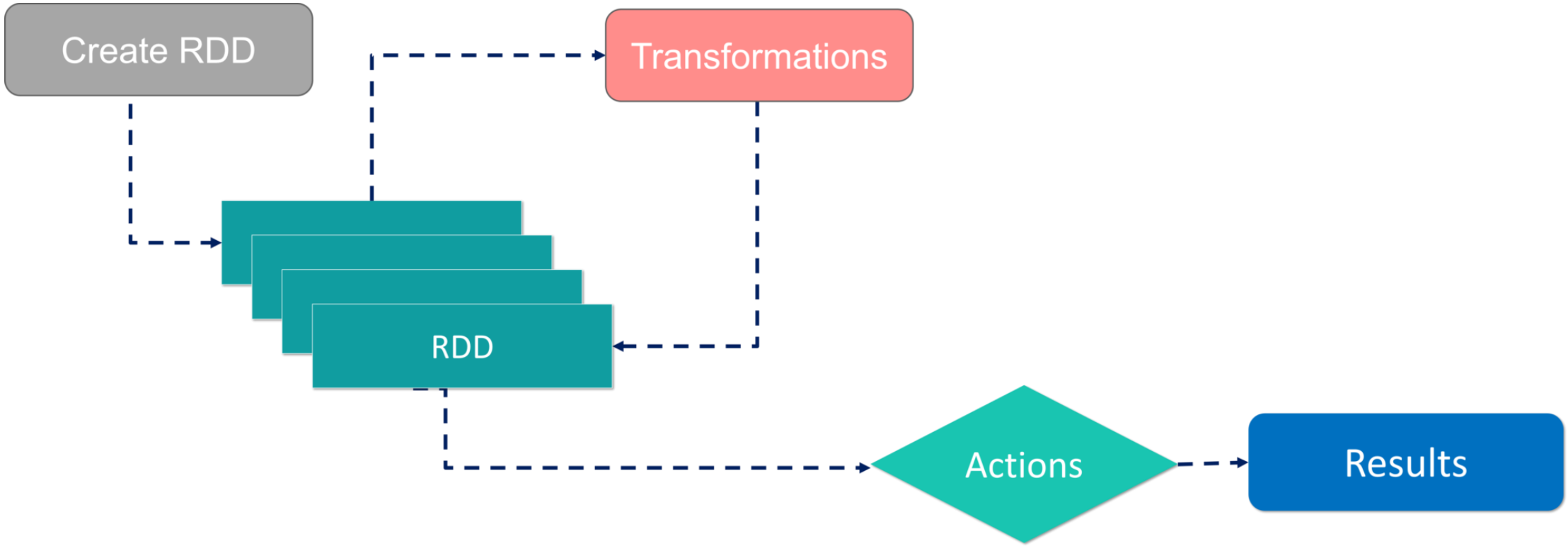
What is the significance of Resilient Distributed Datasets in Spark?

- Resilient Distributed Datasets (RDD) are the fundamental data structure of Apache Spark.
- RDDs are at the heart of every Spark program.
- It is embedded in Spark Core.
- RDDs are immutable, fault-tolerant, distributed collections of objects that can be operated on in parallel.
- RDD's are split into partitions and can be executed on different nodes of a cluster.
- RDDs are created by either transformation of existing RDDs or by loading an external dataset from stable storage like HDFS or HBase.

Resilient Distributed Dataset (RDD)

What is the significance of Resilient Distributed Datasets in Spark?

- Here is how the architecture of RDD looks like:



Resilient Distributed Dataset (RDD)

Explain the types of operations supported by RDDs?

- Resilient Distributed Dataset (RDD) is a basic data structure of Spark.
- RDDs are the immutable Distributed collections of objects of any type.
- It records the data from various nodes and prevents it from significant faults.
- The Resilient Distributed Dataset (RDD) in Spark supports two types of operations. These are:

1. Transformations
2. Actions

Resilient Distributed Dataset (RDD)

Explain the types of operations supported by RDDs?

RDD Transformation:

- The transformation function generates new RDD from the pre-existing RDDs in Spark. Whenever the transformation occurs, it generates a new RDD by taking an existing RDD as input and producing one or more RDD as output. Due to its Immutable nature, the input RDDs don't change and remain constant.
- Along with this, if we apply Spark transformation, it builds RDD lineage, including all parent RDDs of the final RDDs. We can also call this RDD lineage as RDD operator graph or RDD dependency graph. RDD Transformation is the logically executed plan, which means it is a Directed Acyclic Graph (DAG) of the continuous parent RDDs of RDD.

Resilient Distributed Dataset (RDD)

Explain the types of operations supported by RDDs?

RDD Action:

- The RDD Action works on an actual dataset by performing some specific actions. Whenever the action is triggered, the new RDD does not generate as happens in transformation. It depicts that Actions are Spark RDD operations that provide non-RDD values. The drivers and external storage systems store these non-RDD values of action. This brings all the RDDs into motion.
- If appropriately defined, the action is how the data is sent from the Executor to the driver. Executors play the role of agents and the responsibility of executing a task. In comparison, the driver works as a JVM process facilitating the coordination of workers and task execution.

Resilient Distributed Dataset (RDD)

Data Sharing is Slow in MapReduce:

- MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.
- Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex: between two MapReduce jobs) is to write it to an external stable storage system (Ex: HDFS). Although this framework provides numerous abstractions for accessing a cluster's computational resources, users still want more.
- Both Iterative and Interactive applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

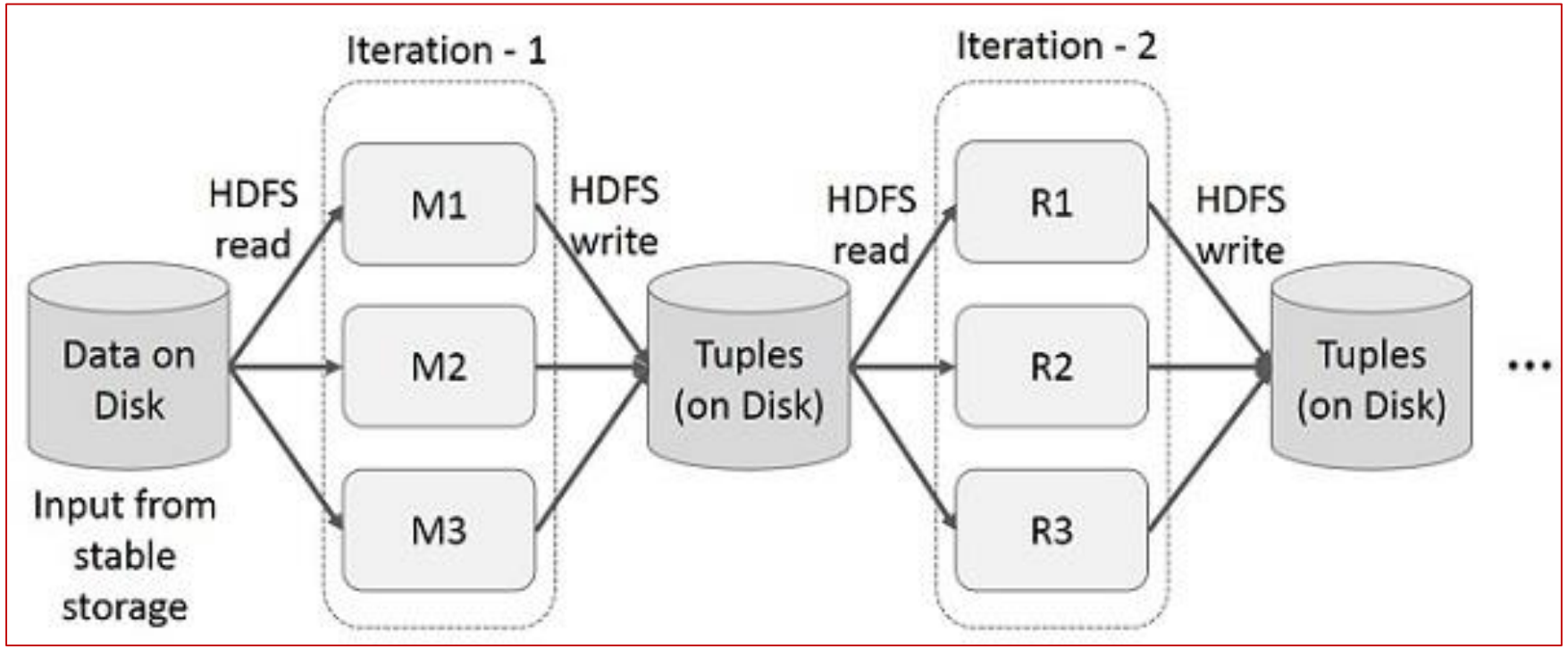
Resilient Distributed Dataset (RDD)

Iterative Operations on MapReduce:

- Reuse intermediate results across multiple computations in multi-stage applications.
- The following illustration explains how the current framework works, while doing the iterative operations on MapReduce.
- This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.

Resilient Distributed Dataset (RDD)

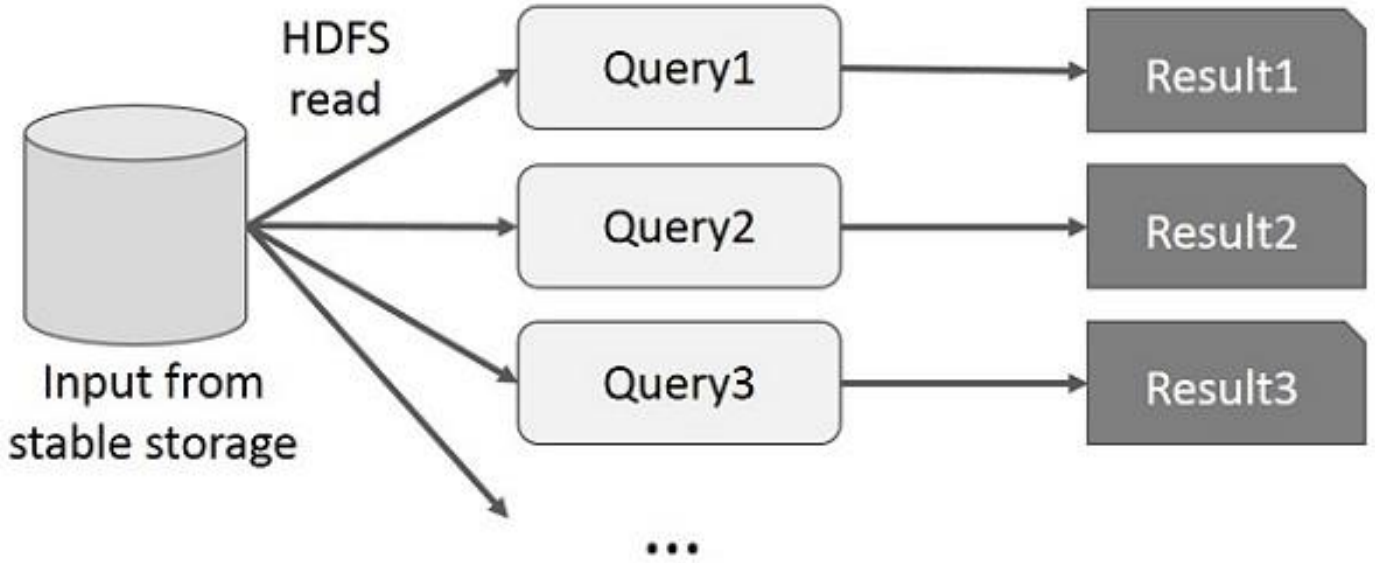
Iterative Operations on MapReduce:



Resilient Distributed Dataset (RDD)

Interactive Operations on MapReduce:

- User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominate application execution time.
- The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



Resilient Distributed Dataset (RDD)

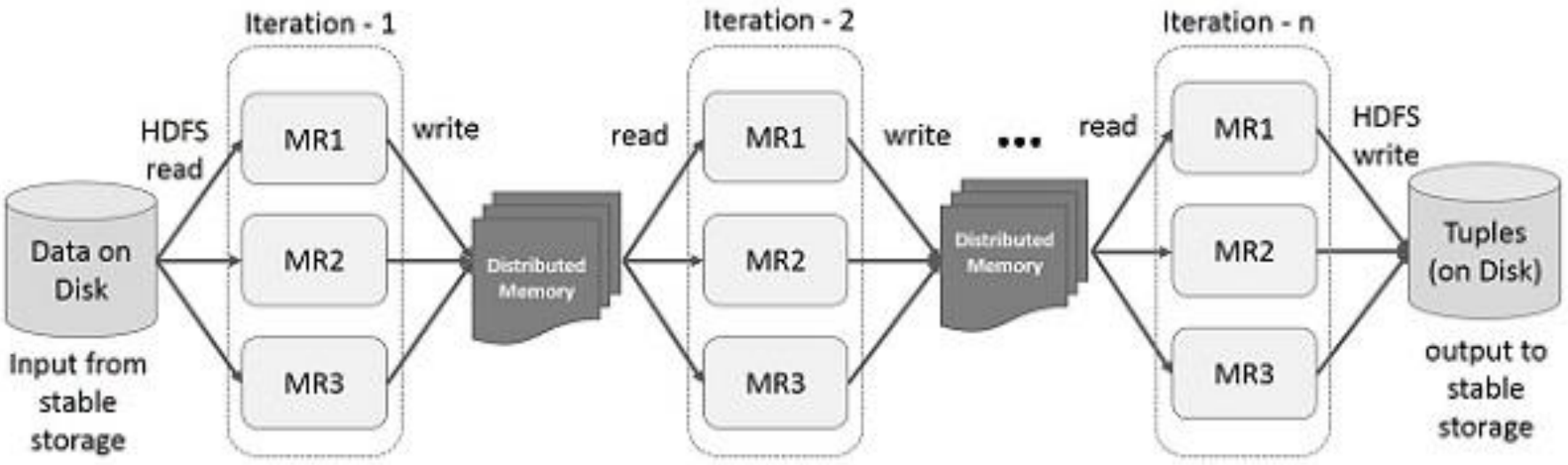
Data Sharing using Spark RDD:

- Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.
- Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is Resilient Distributed Datasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Resilient Distributed Dataset (RDD)

Iterative Operations on Spark RDD:

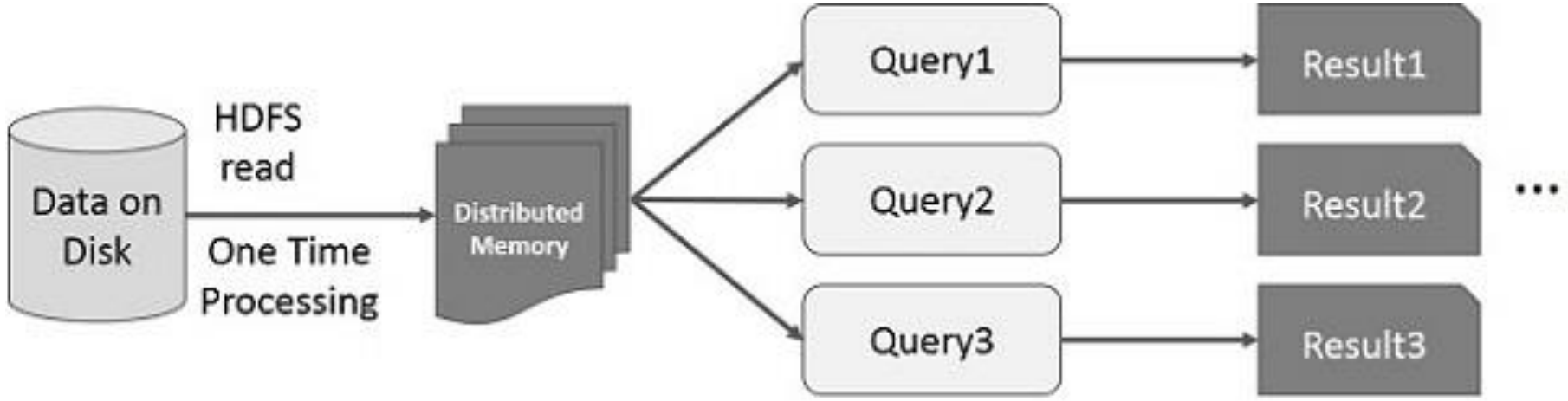
- The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.



Resilient Distributed Dataset (RDD)

Interactive Operations on Spark RDD:

- This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



Resilient Distributed Dataset (RDD)

Interactive Operations on Spark RDD:

- By default, each transformed RDD may be recomputed each time you run an action on it.
- However, you may also persist an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it.
- There is also support for persisting RDDs on disk, or replicated across multiple nodes.

Programming with RDD

- The RDD (Resilient Distributed Dataset) is the Spark's core abstraction.
- It is a collection of elements, partitioned across the nodes of the cluster so that we can execute various parallel operations on it.
- There are two ways to create RDDs:
  1. Parallelizing an existing data in the driver program
  2. Referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

Programming with RDD

Parallelized Collections:

- To create parallelized collection, call **SparkContext's** parallelize method on an existing collection in the driver program. Each element of collection is copied to form a distributed dataset that can be operated on in parallel.

```
val info = Array(1, 2, 3, 4)

val distinfo = sc.parallelize(info)
```

- Now, we can operate the distributed dataset (distinfo) parallel such like

```
distinfo.reduce((a, b) => a + b)
```

Programming with RDD

External Datasets:

- In Spark, the distributed datasets can be created from any type of storage sources supported by Hadoop such as HDFS, Cassandra, HBase and even our local file system. Spark provides the support for text files, **SequenceFiles**, and other types of Hadoop **InputFormat**.
- **SparkContext's** textFile method can be used to create RDD's text file. This method takes a URI for the file (either a local path on the machine or a hdfs://) and reads the data of the file.

```
scala> val data=sc.textFile("sparkwordcount.txt")

data: org.apache.spark.rdd.RDD[String] = sparkwordcount.txt
MapPartitionsRDD[1] at textFile at <console>:24
```

Creating RDDs

Creating RDDs in Spark using Scala: Code Example and Output

```
// Step 1: Import SparkContext
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
// Step 2: Initialize SparkContext
val conf = new
SparkConf().setAppName("RDDExample").setMaster("local")
val sc = new SparkContext(conf)
```

Creating RDDs

```
// Step 3: Creating RDDs from a collection
val data = Array(1, 2, 3, 4, 5)
val rdd = sc.parallelize(data)
// Step 4: Performing operations on RDDs
val squaredRDD = rdd.map(x => x * x)
squaredRDD.collect().foreach(println) // Output: 1, 4, 9, 16, 25
```

Creating RDDs

// Step 5: Creating RDDs from an external dataset

```
val textRDD = sc.textFile("path/to/your/textfile.txt")

val wordCounts = textRDD.flatMap(line => line.split(" "))
.map(word => (word, 1))

.reduceByKey(_ + _)

wordCounts.collect().foreach(println) // Output: (word, count)
pairs
```

Creating RDDs

// Step 6: Creating RDDs using Hadoop InputFormat

```
import org.apache.hadoop.mapred.TextInputFormat

val hadoopRDD = sc.hadoopFile[LongWritable, Text,
TextInputFormat](    "path/to/hadoop/input",
classOf[TextInputFormat], classOf[LongWritable],
classOf[Text])

val processedRDD = hadoopRDD.map(record => record._2.toString)

processedRDD.take(10).foreach(println) // Output: First 10
records from the Hadoop input
```

// Step 7: Closing SparkContext

```
sc.stop()
```

RDD Operations

- The RDD provides the two types of operations:

  1. Transformation
  2. Action

Transformation:

- In Spark, the role of transformation is to create a new dataset from an existing one.
- The transformations are considered lazy as they only computed when an action requires a result to be returned to the driver program.

RDD Operations

Transformation:

Some of the frequently used RDD Transformations are shown below:

Transformation	Description
`map`(*`func`*)	It returns a new distributed dataset formed by passing each element of the source through a function `func`.
`filter`(*`func`*)	It returns a new dataset formed by selecting those elements of the source on which `func` returns true.
`flatMap`(*`func`*)	Here, each input item can be mapped to zero or more output items, so `func` should return a sequence rather than a single item.

RDD Operations

Transformation:

Transformation	Description
mapPartitions(*func*)	It is similar to `map`, but runs separately on each partition (block) of the RDD, so `func` must be of type `Iterator<T> => Iterator<U>` when running on an RDD of type `T`. Here `T` and `U` are `Scala types`.
mapPartitionsWithIndex(*func*)	It is similar to `mapPartitions` that provides `func` with an integer value representing the index of the partition, so `func` must be of type `(Int, Iterator<T>) => Iterator<U>` when running on an RDD of type `T`.

RDD Operations

Transformation:

Transformation	Description
sample(*withReplacement, fraction, seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.

RDD Operations

Transformation:

Transformation	Description
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. *Note:* If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. *Note:* By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.

RDD Operations

Transformation:

Transformation	Description
reduceByKey(*func*, *[numPartitions]*)	When called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function `func`, which must be of type `(V,V) => V`. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.

RDD Operations

Transformation:

Transformation	Description
aggregateByKey(*zeroValue*)(*seqOp, combOp,* [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

RDD Operations

Transformation:

Transformation	Description
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. *Outer joins* are supported through *leftOuterJoin*, *rightOuterJoin*, and *fullOuterJoin*.

RDD Operations

Transformation:

Transformation	Description
cogroup(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called *groupWith*.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, [*envVars*])	Pipe each partition of the RDD through a shell command, e.g. a *Perl* or bash script. RDD elements are written to the process's *stdin* and *lines* output to its *stdout* are returned as an RDD of strings.

RDD Operations

Transformation:

Transformation	Description
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to *numPartitions*. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

RDD Operations: Basic Transformations (Example)

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x) # => {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) # => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x)) # => {0, 0, 1, 0, 1, 2}
```

RDD Operations

Actions:

- In Spark, the role of action is to return a value to the driver program after running a computation on the dataset.
- The following table lists some of the common actions supported by Spark:

Transformation	Meaning
`reduce``(`*`func`*`)`	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
`collect``()`	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

RDD Operations

Actions:

Transformation	Meaning
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to *take(1)*).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement, num, [seed]*)	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n, [ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.

RDD Operations

Actions:

Transformation	Meaning
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).

RDD Operations

Actions:

Transformation	Meaning
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior.

RDD Operations: Basic Actions (Example)

```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection

nums.collect() # => [1, 2, 3]

# Return first K elements

nums.take(2) # => [1, 2]

# Count number of elements

nums.count() # => 3

# Merge elements with an associative function

nums.reduce(lambda x, y: x + y) # => 6

# Write elements to a text file

nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

Spark's "distributed reduce" transformations act on RDDs of key-value pairs.

Python:

```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

Scala:

```
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```

Java:

```
Tuple2 pair = new Tuple2(a, b); // class scala.Tuple2
pair._1 // => a
pair._2 // => b
```

Some Key-Value Operations

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])
pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}
pets.groupByKey()
# => {(cat, Seq(1, 2)), (dog, Seq(1)}
pets.sortByKey()
# => {(cat, 1), (cat, 2), (dog, 1)}
```

reduceByKey also automatically implements combiners on the map side

Spark Word Count Example

In Spark word count example, we find out the frequency of each word exists in a particular file. Here, we use Scala language to perform Spark operations.

Demo in Ubuntu:

Steps to execute Spark word count example:

In this example, we find and display the number of occurrences of each word.

- Create a text file in our local machine and write some text into it.

```
hadoop@ubuntu22:~/spark-3.5.2-bin-hadoop3/bin$ gedit
Sparkwordcount.txt
```

```
Dear Bear River

Car Car River

Deer Car Bear

Dear Deer River
```

Spark Word Count Example

Demo in Ubuntu:

```
hadoop@ubuntu22:~/spark-3.5.2-bin-hadoop3/bin$ ls
```

Check the text written in the Sparkwordcount.txt file.

```
hadoop@ubuntu22:~/spark-3.5.2-bin-hadoop3/bin $ cat Sparkwordcount.txt
Dear Bear River
Car Car River
Deer Car Bear
Dear Deer River
```

Spark Word Count Example

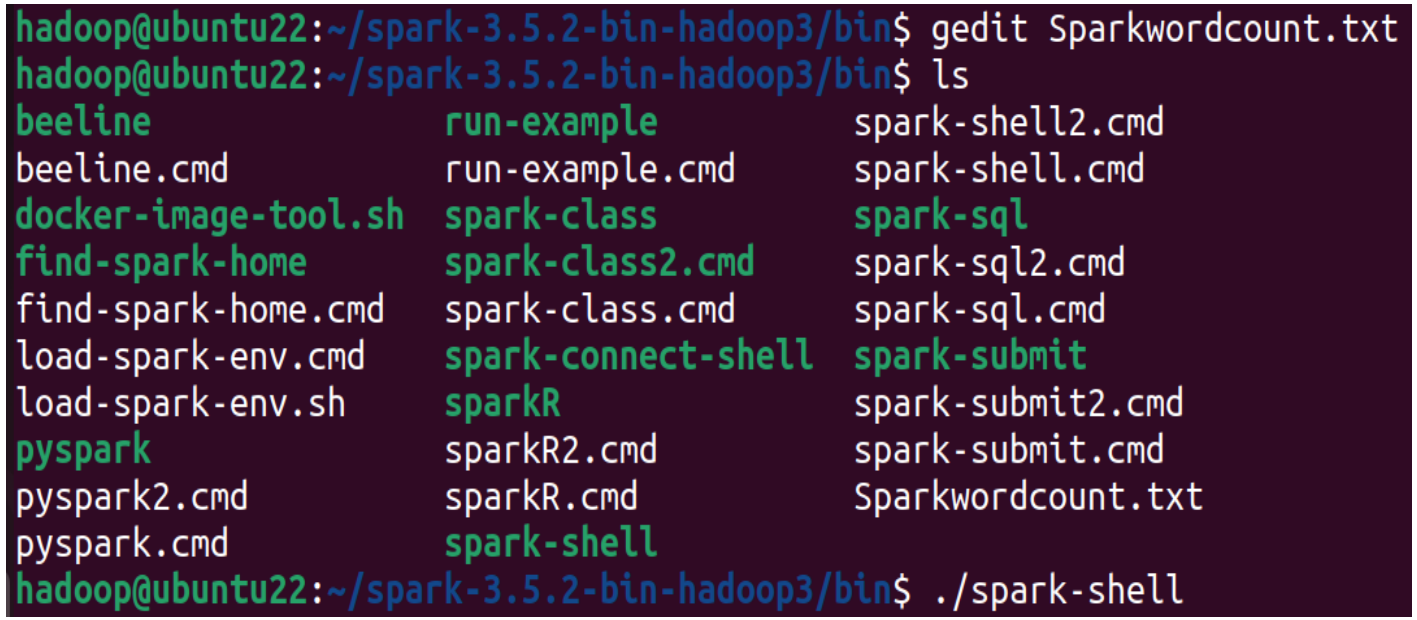
- Now, follow the below command to open the spark in Scala mode.

```
hadoop@ubuntu22:~/spark-3.5.2-bin-hadoop3/bin$ ./spark-shell

Welcome to Spark version 2.4.4

Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server
VM, Java 1.8.0_171)Type in expressions to have them
evaluated.Type :help for more information.
```

scala>



Spark Word Count Example

- Let's create an RDD by using the following command.

```
scala> val data=sc.textFile("/home/hadoop/spark-3.5.2-bin-hadoop3/bin/Sparkwordcount.txt")
```

Here, pass any file name that contains the data.

```
data: org.apache.spark.rdd.RDD[String] = sparkwordcount.txt MapPartitionsRDD[1] at textFile at <console>:23
```

Spark Word Count Example

- Now, we can read the generated result by using the following command.

```
scala> data.collect;

res3: Array[String] = Array(Dear Bear River, Car Car River,
Deer Car Bear, Dear Deer River)
```

```
scala> val data=sc.textFile("/home/hadoop/spark-3.5.2-bin-hadoop3/bin/Sparkword
count.txt")
data: org.apache.spark.rdd.RDD[String] = /home/hadoop/spark-3.5.2-bin-hadoop3/b
in/Sparkwordcount.txt MapPartitionsRDD[9] at textFile at <console>:23

scala> data.collect;
res3: Array[String] = Array(Dear Bear River, Car Car River, Deer Car Bear, Dear
 Deer River)
```

Spark Word Count Example

- Here, we split the existing data in the form of individual words by using the following command.

```
scala> val splitdata = data.flatMap(line => line.split(" "));

splitdata: org.apache.spark.rdd.RDD[String] =
MapPartitionsRDD[10] at flatMap at <console>:23
```

Spark Word Count Example

- Now, we can read the generated result by using the following command.

```
scala> splitdata.collect;

res4: Array[String] = Array(Dear, Bear, River, Car, Car,
River, Deer, Car, Bear, Dear, Deer, River)
```

```
scala> val splitdata = data.flatMap(line => line.split(" "));
splitdata: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[10] at flatMap a
t <console>:23

scala> splitdata.collect;
res4: Array[String] = Array(Dear, Bear, River, Car, Car, River, Deer, Car, Bear
, Dear, Deer, River)
```

Spark Word Count Example

- Now, perform the map operation

```
scala> val mapdata = splitdata.map(word => (word,1));
```

Here, we are assigning a value 1 to each word.

```
mapdata: org.apache.spark.rdd.RDD[(String, Int)] =
MapPartitionsRDD[11] at map at <console>:23
```

Spark Word Count Example

- Now, we can read the generated result by using the following command.

```
scala> mapdata.collect;
```

Here, we are assigning a value 1 to each word.

```
res5: Array[(String, Int)] = Array((Dear,1), (Bear,1),
(River,1), (Car,1), (Car,1), (River,1), (Deer,1), (Car,1),
(Bear,1), (Dear,1), (Deer,1), (River,1))
```

```
scala> val mapdata = splitdata.map(word => (word,1));
mapdata: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[11] at map
at <console>:23

scala> mapdata.collect;
res5: Array[(String, Int)] = Array((Dear,1), (Bear,1), (River,1), (Car,1), (Car
,1), (River,1), (Deer,1), (Car,1), (Bear,1), (Dear,1), (Deer,1), (River,1))
```

Spark Word Count Example

- Now, perform the reduce operation.

```
scala> val reducedata = mapdata.reduceByKey(_+_);
```

Here, we are summarizing the generated data.

```
reducedata: org.apache.spark.rdd.RDD[(String, Int)] =
ShuffledRDD[12] at reduceByKey at <console>:23
```

Spark Word Count Example

- Now, we can read the generated result by using the following command.

```
scala> reducedata.collect;

res6: Array[(String, Int)] = Array((Dear,2), (Deer,2),
(Bear,2), (Car,3), (River,3))
```

```
scala> val reducedata = mapdata.reduceByKey(_+_);
reducedata: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[12] at reduce
ByKey at <console>:23

scala> reducedata.collect;
res6: Array[(String, Int)] = Array((Dear,2), (Deer,2), (Bear,2), (Car,3), (Rive
r,3))
```