What is the PySpark RDD tutorial and how can I learn it with examples?

Authored by

stats writer

June 24, 2024

The PySpark RDD tutorial is a comprehensive guide designed to help individuals learn how to use PySpark's Resilient Distributed Datasets (RDDs). RDDs are the fundamental data structures used in PySpark for in-memory data processing, making them a critical component for data analysis and manipulation.

The tutorial provides a step-by-step approach to understanding the basics of RDDs, including how to create, manipulate, and transform them. It also covers more advanced topics such as parallel processing, caching, and partitioning.

To aid in the learning process, the tutorial includes various examples and hands-on exercises to help users gain a practical understanding of how to use RDDs in real-world scenarios. Additionally, it offers tips and best practices for optimizing RDD performance and troubleshooting common errors.

Overall, the PySpark RDD tutorial is an excellent resource for gaining a solid foundation in using RDDs and enhancing your skills in data analysis and processing with PySpark. By following the tutorial and practicing with the examples provided, individuals can quickly become proficient in working with RDDs and leveraging their capabilities for efficient and scalable data processing.

RDD Introduction

RDD (Resilient Distributed Dataset) is a core building block of PySpark. It is a fault-tolerant, immutable, distributed collection of objects. Immutable means that once you create an RDD, you cannot change it. The data within RDDs is segmented into logical partitions, allowing for distributed computation across multiple nodes within the cluster.

This PySpark RDD Tutorial will help you understand what is RDD (Resilient Distributed Dataset) , its advantages, and how to create an RDD and use it, along with GitHub examples. You can find all RDD Examples explained in that article at GitHub PySpark examples project for quick reference.

By the end of this PySpark RDD tutorial, you will have a better understanding of PySpark RDD, how to apply transformations and actions, and how to operate on pair RDD.

1. What is RDD (Resilient Distributed Dataset)?

RDD, or Resilient Distributed Dataset, serves as a core component within PySpark, offering a fault-tolerant, distributed collection of objects. This foundational element boasts immutability, ensuring that once an RDD is created, it remains unchanged. Furthermore, RDDs are partitioned logically, facilitating parallel computation across various nodes within the cluster.

RDDs are collections of objects similar to a list in Python; the difference is that RDD is computed

on several processes scattered across multiple physical servers, also called nodes in a cluster, while a Python collection lives and processes in just one process.

By leveraging RDDs, PySpark users benefit from fault tolerance, scalability, and parallel processing capabilities, enabling efficient handling of large-scale data processing tasks. With RDDs, users can confidently tackle complex data analysis challenges while enjoying the flexibility and resilience offered by this robust distributed computing framework.

Additionally, RDDs provide data abstraction of partitioning and distribution of the data designed to run computations in parallel on several nodes, while doing transformations on RDD we don't have to worry about the parallelism as PySpark by default provides.

Note: RDDs can have a name and unique identifier (id)

2. PySpark RDD Benefits

PySpark is widely adopted in the Machine learning and Data science community due to its advantages over traditional Python programming.

In-Memory Processing

PySpark loads the data from disk and processes it in memory, and keeps the data in memory; this is the main difference between PySpark and MapReduce (I/O intensive). In between the transformations, we can also cache/persists the RDD in memory to reuse the previous computations.

Immutability

PySpark RDDs are immutable in nature meaning, once RDDs are created you cannot modify them. When we apply transformations on RDD, PySpark creates a new RDD and maintains the RDD Lineage.

Fault Tolerance

PySpark operates on fault-tolerant data stores on HDFS, S3 e.t.c. Hence, if any RDD operation fails, it automatically reloads the data from other partitions. Also, when PySpark applications running on a cluster, PySpark task failures are automatically recovered for a certain number of times (as per the configuration) and finish the application seamlessly.

Lazy Evolution

PySpark does not evaluate the RDD transformations as they appear/encountered by Driver instead it keeps the all transformations as it encounters(DAG) and evaluates the all transformation when it sees the first RDD action.

Partitioning

When you create RDD from a data, It by default partitions the elements in a RDD. By default it partitions to the number of cores available.

3. PySpark RDD Limitations

PySpark RDDs are not much suitable for applications that make updates to the state store such as storage systems for a web application. For these applications, it is more efficient to use systems that perform traditional update logging and data checkpointing, such as databases. The goal of RDD is to provide an efficient programming model for batch analytics and leave these asynchronous applications.

4. RDD Creation

You can create RDD by parallelizing the existing collection and reading data from a disk.

Before we look into examples, first let's initialize SparkSession using the builder pattern method defined in SparkSession class. While initializing, we need to provide the master and application name as shown below. In real-time application, you will pass master from spark-submit instead of hardcoding.

```
# Imports
from pyspark.sql import SparkSession
```

```
# Create SparkSession
spark = SparkSession.builder
.master("local")
.appName(arabpsychology.com)
.getOrCreate()
```

`master()` - If you are running it on the cluster you need to use your master name as an argument to master(). usually, it would be either yarn (Yet Another Resource Negotiator) or `mesos` depends on your cluster setup.

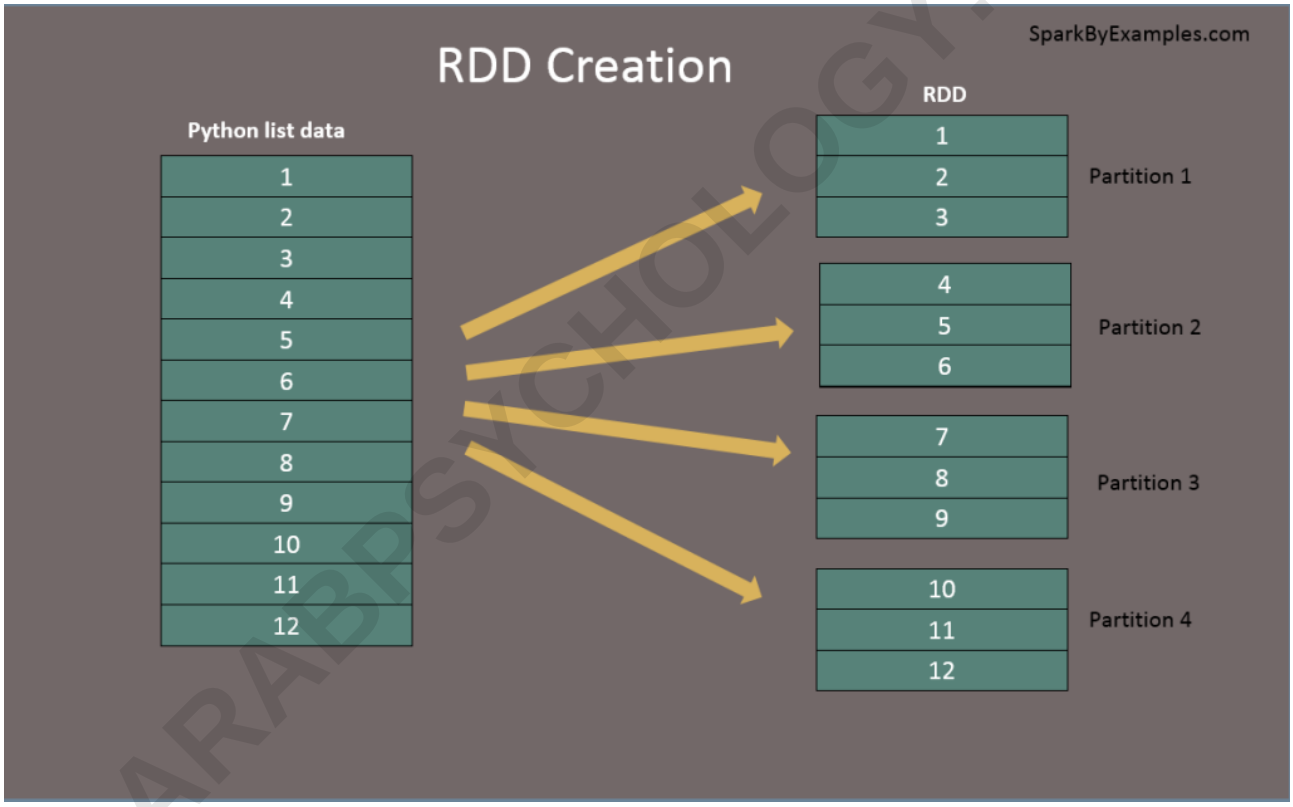
`appName()` - Used to set your application name.

`getOrCreate()` - This returns a SparkSession object if already exists, and creates a new one if not exist.

Note: Creating SparkSession object, internally creates one SparkContext per JVM.

Using sparkContext.parallelize()

By using `parallelize()` function of SparkContext (sparkContext.parallelize()) you can create an RDD. This function loads the existing collection from your driver program into parallelizing RDD. This method of creating an RDD is used when you already have data in memory that is either loaded from a file or from a database. and all data must be present in the driver program prior to creating RDD.



RDD from list

```
# Create RDD from parallelize
data =
rdd = spark.sparkContext.parallelize(data)
```

For production applications, we mostly create RDD by using external storage systems like `HDFS`, `S3`, `HBase` e.t.c. To make it simple for this PySpark RDD tutorial we are using files from the local

system or loading it from the python list to create RDD.

Using sparkContext.textFile()

Use the textFile() method to read a .txt file into RDD.

```
# Create RDD from external Data source
rdd2 = spark.sparkContext.textFile("/path/textFile.txt")
```

Using sparkContext.wholeTextFiles()

wholeTextFiles() function returns a PairRDD with the key being the file path and the value being file content.

```
# Read entire file into a RDD as single record.
rdd3 = spark.sparkContext.wholeTextFiles("/path/textFile.txt")
```

Besides using text files, we can also create RDD from CSV file, JSON, and more formats.

Create empty RDD using sparkContext.emptyRDD

Using `emptyRDD()` method on sparkContext we can create an RDD with no data. This method creates an empty RDD with no partition.

```
# Create an empty RDD with no partition
rdd = spark.sparkContext.emptyRDD
```

Output:
rddString = spark.sparkContext.emptyRDD

Creating empty RDD with partition

Sometimes we may need to write an empty RDD to files by partition, In this case, you should create an empty RDD with partition.

```
# Create empty RDD with partition
rdd2 = spark.sparkContext.parallelize(,10) #This creates 10 partitions
```

5. RDD Partitions

When we use `parallelize()`, `textFile()` or `wholeTextFiles()` methods of SparkContxt to initiate RDD, it automatically splits the data into partitions based on resource availability. When you run it on a laptop, it creates partitions as the same number of cores available on your system.

getNumPartitions() - This is an RDD function that returns a number of partitions your dataset split into.

```
# Get partition count
print("Initial partition count:"+str(rdd.getNumPartitions()))
```

Outputs: Initial partition count:2

Set parallelize manually - We can also set a number of partitions manually, all we need is to pass a number of partitions as the second parameter to these functions for example;

```
# Set partitions manually
sparkContext.parallelize(, 10)
```

6. Repartition and Coalesce

Sometimes, we may need to repartition the RDD, PySpark provides two ways to repartition; first using `repartition()` method, which shuffles data from all nodes also called full shuffle and second coalesce() method which shuffles data from minimum nodes, for examples if you have data in 4 partitions and doing `coalesce(2)` moves data from just 2 nodes.

Both of these functions take the number of partitions to repartition RDD as shown below. Note that repartition() method is a very expensive operation as it shuffles data from all nodes in a cluster.

```
# Repartition the RDD
reparRdd = rdd.repartition(4)
print("re-partition count:"+str(reparRdd.getNumPartitions()))
```

Outputs:
re-partition count:4

Note: repartition() or coalesce() methods also return a new RDD.

7. PySpark RDD Operations

RDD operations are the core transformations and actions performed on RDDs

RDD transformations - Transformations are lazy operations; instead of updating an RDD, these operations return another RDD.
RDD actions - operations that trigger computation and return RDD values.

RDD Transformations with example

Transformations on PySpark RDD return another RDD, and transformations are lazy, meaning they don't execute until you call an action on RDD. Some transformations on RDDs are `flatMap()`, `map()`, `reduceByKey()`, `filter()`, `sortByKey()` and return a new RDD instead of updating the current.

Let's use some of the most common transformations to perform word count example.

First, by using textFile(), read the text file into RDD. The text file used in this example is available at the GitHub project.

```
# Read data from text file
rdd = spark.sparkContext.textFile("/tmp/test.txt")
```

flatMap - The `flatMap()` transformation in the RDD API flattens the resulting RDD after applying a function to each element, producing a new RDD. In the provided example below, each record is initially split by space within an RDD, and subsequently, the transformation flattens it. The resulting RDD comprises individual records, each containing a single word."

```
# split the data by spact and flatten it.
rdd2 = rdd.flatMap(lambda x: x.split(" "))
```

map - The `map()` transformation is used to perform various complex operations, such as adding or updating an element. The result of map transformations retains the same number of records as the input.

In our word count illustration, a new column with a value of 1 is added to each word. The resulting RDD is transformed into PairRDDFunctions, comprising key-value pairs. Each key, representing a word of type String, is associated with a value of 1, type Int."

```
# Apply the mpa() transformation
# Add a new element with value 1 to each word
rdd3 = rdd2.map(lambda x: (x,1))
```

reduceByKey - The `reduceByKey()` combines the values associated with each key using the provided function. In our scenario, it aggregates the word strings by using the sum function on the corresponding values. The result of our RDD outcome comprises distinct words along with their respective counts.

```
# Use reduceByKey()
rdd4 = rdd3.reduceByKey(lambda a,b: a+b)
```

sortByKey - The `sortByKey()` transformation arranges the elements of an RDD based on their keys. In our scenario, we initially convert the RDD from `(String, Int)` to `(Int, String)` using the `map` transformation. Subsequently, `sortByKey()` sorts the RDD primarily based on integer values.

```
# Using sortByKey()
rdd5 = rdd4.map(lambda x: (x,x)).sortByKey()
```

# Print rdd5 result to console
print(rdd5.collect())

Please refer to this page for the full list of RDD transformations.

RDD Actions with example

RDD Action operations trigger the execution of transformations on RDDs (Resilient Distributed Datasets) and produce a result that can be either returned to the driver program or saved to an external storage system.

We will continue to use our word count example and perform some actions on it.

count() - Returns the number of records in an RDD

```
# Action - count
print("Count : "+str(rdd6.count()))
```

first() - Returns the first record.

```
# Action - first
firstRec = rdd6.first()
print("First Record : "+str(firstRec) + ","+ firstRec)
```

max() - Returns max record.

```
# Action - max
datMax = rdd6.max()
print("Max Record : "+str(datMax) + ","+ datMax)
```

reduce() - Reduces the records to single, we can use this to count or sum.

```
# Action - reduce
totalWordCount = rdd6.reduce(lambda a,b: (a+b,a))
print("dataReduce Record : "+str(totalWordCount))
```

take() - Returns the record specified as an argument.

```
# Action - take
data3 = rdd6.take(3)
for f in data3:
print("data3 Key:"+ str(f) +", Value:"+f)
```

collect() - Returns all data from RDD as an array. Be careful when you use this action when you are working with huge RDD with millions and billions of data as you may run out of memory on the driver.

```
# Action - collect
data = rdd6.collect()
for f in data:
print("Key:"+ str(f) +", Value:"+f)
```

saveAsTextFile() - Using saveAsTestFile action, we can write the RDD to a text file.

```
rdd6.saveAsTextFile("/tmp/wordCount")
```

Note: Please refer to this page for a full list of RDD actions.

8. Types of RDD

PairRDDFunctions or PairRDD - Pair RDD is a key-value pair This is mostly used RDD type,

ShuffledRDD -

DoubleRDD -

SequenceFileRDD -

HadoopRDD -

ParallelCollectionRDD -

9. Shuffle Operations

Shuffling, in PySpark, serves as a means to redistribute data among various executors and potentially across multiple machines. It occurs when specific transformation operations like `groupByKey()`, `reduceByKey()`, and `join()` are applied to RDDs.

PySpark's shuffling process incurs significant costs due to the following factors:

Disk I/O activitiesInvolvement of data serialization and deserializationNetwork I/O operations

When creating an RDD, PySpark doesn't necessarily store the data for all keys in a partition since at the time of creation there is no way we can set the key for data set.

Hence, when we run the `reduceByKey()` operation to aggregate the data on keys, PySpark does the following. needs to first run tasks to collect all the data from all partitions and

For example, when we perform `reduceByKey()` operation, PySpark does the following

PySpark RDD triggers shuffle and repartition for several operations like `repartition()` and `coalesce()`, `groupByKey()`, `reduceByKey()`, `cogroup()` and `join()` but not `countByKey()` .

Shuffle partition size & Performance

Depending on your dataset size, the number of cores, and the memory available, PySpark shuffling can either optimize or harm your job performance. When dealing with smaller datasets,

it's advisable to reduce the shuffle partitions to avoid ending up with numerous partitioned files containing fewer records per partition. This situation leads to running many tasks with minimal data to process.

Conversely, excessive data with fewer partitions results in longer-running tasks and may even lead to out-of-memory errors. Determining the optimal shuffle partition size is challenging and often requires multiple iterations with different values to achieve the desired optimization. This parameter is crucial to address when encountering performance issues in PySpark jobs.

10. Persist RDD

PySpark *Cache* and P*ersist* are optimization techniques to improve the performance of the RDD jobs that are iterative and interactive. In this PySpark RDD Tutorial section, I will explain how to use persist() and cache() methods on RDD with examples.

Although PySpark boasts computation speeds up to 100 times faster than traditional MapReduce jobs, performance degradation may occur when jobs fail to leverage repeated computations, particularly when handling massive datasets in the billions or trillions. Hence, optimizing computations becomes imperative to enhance performance.

PySpark offers optimization techniques such as the `cache()` and `persist()` methods. These mechanisms enable the storage of intermediate RDD computations, facilitating their reuse in subsequent actions.

Upon persisting or caching an RDD, each worker node stores its partitioned data in memory or on disk, allowing for reuse in subsequent RDD actions. Moreover, Spark's persisted data on nodes exhibit fault tolerance, ensuring that any lost partitions are automatically recomputed using the original transformations that generated them.

RDD Cache

PySpark RDD `cache()` method by default saves RDD computation to storage level `` `MEMORY_ONLY` `` meaning it will store the data in the JVM heap as unserialized objects.

PySpark `cache()` method in RDD class internally calls `persist()` method which in turn uses `sparkSession.sharedState.cacheManager.cacheQuery` to cache the result set of RDD. Let's look at an example.

```
# cache()
cachedRdd = rdd.cache()
```

RDD Persist

Using persist() method, you can store the RDD in one of the storage levels

`MEMORY_ONLY`, `MEMORY_ONLY_SER`, `MEMORY_AND_DISK`, `MEMORY_AND_DISK_SER`, `DISK_ONLY`, `MEMORY_ONLY_2`, `MEMORY_AND_DISK_2` and more.

```
# persist()
import pyspark
dfPersist = rdd.persist(pyspark.StorageLevel.MEMORY_ONLY)
dfPersist.show(false)
```

RDD Unpersist

PySpark systematically monitors each invocation of `persist()` and `cache()`, scrutinizing usage across every node. It automatically discards persisted data that remains unused or employs the least-recently-used (LRU) algorithm. Additionally, users can manually remove persisted data using the `unpersist()` method. This action marks the RDD as non-persistent and eradicates all associated blocks from both memory and disk.

```
# unpersist()
rddPersist2 = rddPersist.unpersist()
```

11. RDD Shared Variables

In this section of the PySpark RDD tutorial, let's learn what are the different types of PySpark Shared variables and how they are used in PySpark transformations.

When PySpark executes transformation using `map()` or `reduce()` operations, It executes the transformations on a remote node by using the variables that are shipped with the tasks and these variables are not sent back to PySpark Driver hence there is no capability to reuse and sharing the variables across tasks. PySpark shared variables solve this problem using the below two techniques. PySpark provides two types of shared variables.

Broadcast read-only Variables

Broadcast variables are shared, read-only variables cached and accessible across all nodes in a cluster for use by tasks. Rather than transmitting this data with every task, PySpark employs

efficient broadcast algorithms to distribute broadcast variables to machines, thereby reducing communication costs.

A prime application of PySpark RDD Broadcast is with lookup data, such as zip codes, states, or country lookups.

When executing a PySpark RDD job utilizing Broadcast variables, PySpark undertakes the following steps:

PySpark partitions the job into stages, each with distributed shuffling, and executes actions within each stage.Subsequently, later stages are subdivided into tasks.PySpark broadcasts common data required by tasks within each stage.The broadcasted data is cached in serialized format and deserialized prior to executing each task.

The PySpark Broadcast is created using the `broadcast(v)` method of the SparkContext class. This method takes the argument v that you want to broadcast.

```
# Create broadcast variable
broadcastVar = sc.broadcast()
broadcastVar.value
```

Note that broadcast variables are not sent to executors with sc.broadcast(variable) call instead, they will be sent to executors when they are first used.

Refer to PySpark RDD Broadcast shared variable for more detailed example.

Accumulators

PySpark Accumulators represent another form of shared variable exclusively "added" via an associative and commutative operation. They serve to perform counters similar to MapReduce counters.

By default, PySpark supports the creation of accumulators of any numeric type and offers the flexibility to incorporate custom accumulator types. Programmers can generate the following types of accumulators:

Named accumulatorsUnnamed accumulators

Creating a named accumulator becomes visible on the PySpark web UI under the "Accumulator" tab. Within this tab, two tables are present: the initial table, labeled "accumulable," encompasses all named accumulator variables alongside their respective values. Meanwhile, the subsequent

table, labeled "Tasks," showcases the value of each accumulator altered by a task.

On the other hand, unnamed accumulators do not appear on the PySpark web UI. For pragmatic purposes, it is advisable to utilize named accumulators in most scenarios.

Accumulator variables are created using `SparkContext.longAccumulator(v)`

```
# Create accumulator variable
accum = sc.longAccumulator("SumAccumulator")
sc.parallelize().foreach(lambda x: accum.add(x))
```

12. Advanced API - DataFrame & DataSet

Creating RDD from DataFrame and vice-versa

Though we have more advanced API's over RDD, we would often need to convert DataFrame to RDD or RDD to DataFrame. Below are several examples.

```
# Converts RDD to DataFrame
dfFromRDD1 = rdd.toDF()
```

# Converts RDD to DataFrame with column names
dfFromRDD2 = rdd.toDF("col1","col2")

# using createDataFrame() - Convert DataFrame to RDD
df = spark.createDataFrame(rdd).toDF("col1","col2")

# Convert DataFrame to RDD
rdd = df.rdd

13. Where to go from here?

Learn PySpark DataFrame Tutorial with examples

Naveen Nelamali

Naveen Nelamali (NNK) is a Data Engineer with 20+ years of experience in transforming data into actionable insights. Over the years, He has honed his expertise in designing, implementing, and maintaining data pipelines with frameworks like Apache Spark, PySpark, Pandas, R, Hive and Machine Learning. Naveen journey in the field of data engineering has been a continuous learning, innovation, and a strong commitment to data integrity. In this blog, he shares his experiences with the data as he come across. Follow Naveen @ **LinkedIn** and **Medium**