What is PySpark?

PySpark is an API interface that allows you to write Python code to interact with Apache Spark, which is an open source distributed computing framework to handle big data. As the size of data grows year over year, Spark has become a popular framework in the industry to efficiently process large datasets for streaming, data engineering, real-time analytics, exploratory data analysis and machine learning.

Why use PySpark?

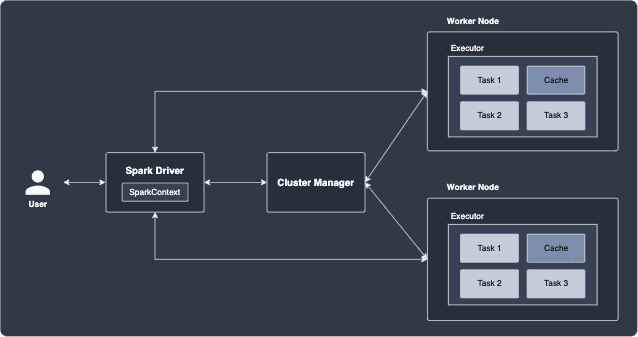
The core value proposition behind PySpark is that:

Spark partitions the dataset into smaller chunks and stores them in multiple machines.

By doing so, Spark can efficiently process massive volumes of data in parallel.

This is extremely useful when you are dealing with large datasets that cannot fit into the memory of a single machine. PySpark can handle a wide array of data formats including Hadoop Distributed File System (HDFS), Cassandra and Amazon S3.

Anatomy of Spark:



Worker nodes:

Worker nodes are machines that host the executors.

A worker node can host multiple executors if CPU and memory are available.

Spark Driver:

Spark Driver is the entry point of the Spark application that receives the user's Spark

program, and is responsible for the following:

creating the SparkContext, which provides the connection to the cluster manager.

SparkContext holds configuration parameters such as the application name, number of CPU cores, memory size of executors running on worker nodes.

Submitting jobs and converting them into tasks.

These tasks are then to be handled by executors.

Coordinates the execution of worker nodes and aggregates data from the worker nodes.

Cluster manager

The cluster manager processes that monitor worker nodes and reserve cluster sources for the Driver to coordinate.

There are many cluster managers to choose from such as YARN, Kubernetes, Mesos and Spark Standalone.

There are of course differences in how resources are allocated for each of these managers,

but they all come with a clean visual web dashboard for live monitoring of your cluster.

Note that the cluster manager does not manage the worker nodes directly (this is the job of the Driver). Instead, the cluster manager simply requests for resources for the Driver to use.

NOTE:

All these components are written in a programming language called Scala,

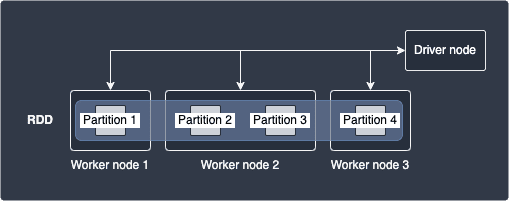
but they are compiled into Java byte-code so that they run in a Java Virtual Machines (JVM),

which is a cross-platform runtime engine.

RDD (Resilient Distributed Dataset):

The basic building block of Spark is the Resilient Distributed Dataset (RDD), which is an immutable data structure that is logically partitioned across multiple nodes in the cluster for parallel computing.

The following diagram illustrates an example of a RDD:



Here, our dataset is represented by a single RDD that consists of 4 partitions that is hosted by 3 separate worker nodes.

Note that worker nodes may hold different number of partitions.

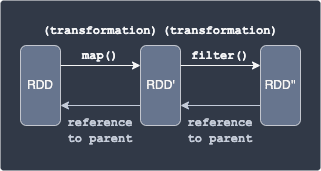
Transformations and Actions

There are two types of operations we can perform on RDDs:

Transformations

Actions

Transformations:

A transformation takes in as input one or more RDDs, and returns a new RDD by applying some function to the data. Examples include map(~), filter(~), sortByKey(~). Transformations can be applied one after another as shown below:

Here, we apply the map(~) transformation to a RDD, which applies a function to each data in RDD to yield RDD'. Next, we apply the filter(~) transformation to select a subset of the data in RDD' to finally obtain RDD''.

Spark keeps track of the series of transformations applied to RDD using graphs called RDD lineage or RDD dependency graphs.

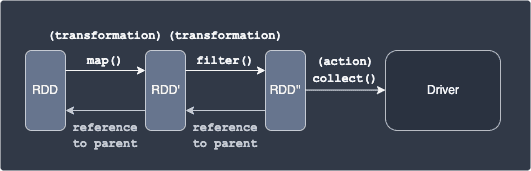
In the above diagram, RDD is considered to be a parent of RDD'.

Every child RDD has a reference to its parent (e.g. RDD' will always have a reference to RDD).

Actions:

Actions are operations that either:

send all the data held by multiple nodes to the driver node. For instance, printing some result in the driver node (e.g. show(~)) or saving some data on an external storage system such as HDFS and Amazon S3. (e.g. saveAsTextFile(~))



After applying transformations, the actual data of the output RDD still reside in different nodes.

Actions are used to gather these scattered results in a single place - either the driver node

or an external data storage.

This should make sense because the data held by the RDD even after applying some transformation is

still partitioned into multiple nodes, and so we would need to aggregate the outputs into a

single place - the driver node in this case.

Examples of actions include show(), reduce() and collect().

WARNING

Since all the data from each node is sent over to the driver with an action, make sure that the driver node has enough RAM to hold all the incoming data - otherwise, an out-of-memory error will occur.

Lazy transformations:

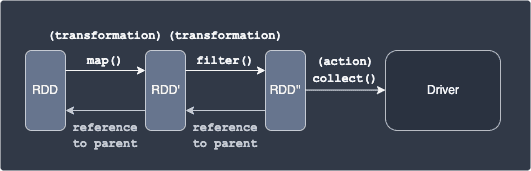
When you execute the transformation, Spark will not immediately perform the transformation.

Instead, RDD will wait until an action link is required, and only then will the transformation fire.

We call this behaviour lazy-execution, and this has the following benefits:

Scheduling - better usage of cluster usage

Some transformations can be grouped together to avoid network traffic



Here, we are first converting each string into uppercase using the transformation map(~),

and then performing a filter(~) transformation to obtain a subset of the data.

Finally, we send the individual results held in different partitions to the driver node

to print the final result on the screen using the action show().

Consider the following RDD with 3 partitions:

rdd = sc.parallelize(["Alex","Bob","Cathy"], numSlices=3)

rdd.collect()

['Alex', 'Bob', 'Cathy']

we are using the parallelize(~) method of SparkContext to create a RDD.

the number of partitions is specified using the numSlices argument.

the collect(~) method is used to gather all the data from each partition to the driver node and print the results on the screen.

Next, we use the map(~) transformation to convert each string (which resides in different partitions) to uppercase.

then use the filter(~) transformation to obtain strings that equal "ALEX":

rdd2 = rdd1.map(lambda x: x.upper())

rdd3 = rdd2.filter(lambda name: name == "ALEX")

rdd3.collect()

['ALEX']

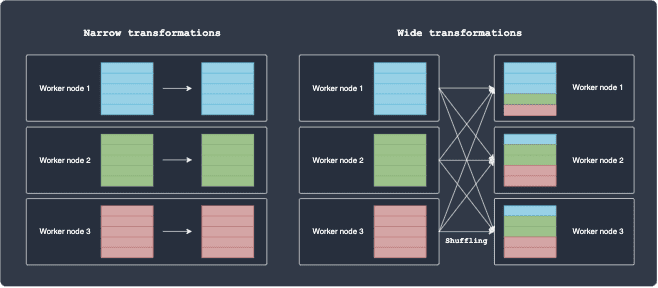
Narrow and wide transformations

There are two types of transformations:

Narrow - no shuffling is needed, which means that data residing in different nodes do not have to be transferred to other nodes

Wide - shuffling is required, and so wide transformations are costly

The difference is illustrated below:



For, narrow transformations, the partition remains in the same node after the transformation

that is, the computation is local. In contrast, wide transformations involve shuffling,

which is slow and expensive because of network latency and bandwidth.

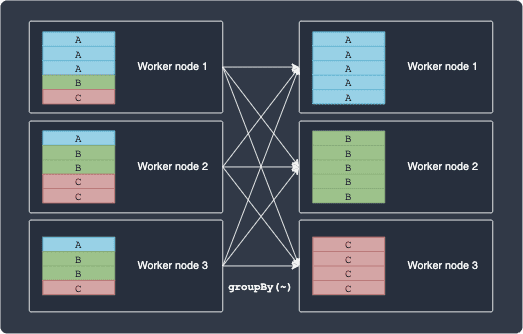
Some of narrow transformations include map(~) and filter(~).

Consider a simple map operation where we increment an integer of some data by one.

It's clear that the each worker node can perform this on their own since there is no dependency between the partitions living on other worker nodes.

Some examples of wide transformations include groupBy(~) and sort(~).

Suppose we wanted to perform a groupBy(~) operation on some column, say a categorical variable consisting of 3 classes: A, B and C.

following diagram illustrates how Spark will execute this operation:

Notice how groupBy(~) cannot be computed locally because the operation requires dependency between partitions lying in different nodes.

Spark jobs, stages and tasks

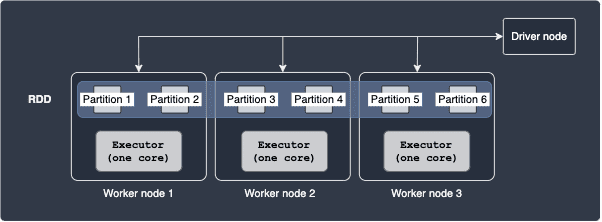
When you invoke an action (e.g. count(), take(), collect()) on an RDD, a job is created.

Spark will then internally decompose a job into a single or multiple stages.

Next, Spark splits each stage into tasks, which are units of work that the Spark driver’s scheduler ships to executors on the worker nodes to handle. Each task processes one unit of partitioned dataset in its memory.

Executors with one core

As an example, consider the following setup:



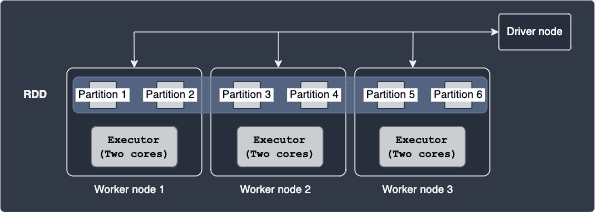
Here, our RDD is composed of 6 partitions, with 2 partitions on each worker node.

The executor threads are equipped with one CPU core, which means that only one task can be performed by each executor at any given time.

The total number of tasks is equal to the number of partitions, which means that there are 6 tasks.

Executors with multiple cores

Multiple tasks can run in parallel on the same executor if you allocate more than one core to each executor. Consider the following case:



Here, each executor is equipped with 2 cores. The total number of tasks here is 6, which is the same as the previous case since there are still 6 partitions. With 2 cores, each executor can handle 2 tasks in parallel.

As you can tell from this example, the more number of cores you allocate to each executor, the more tasks you can perform in parallel.

Number of partitions:

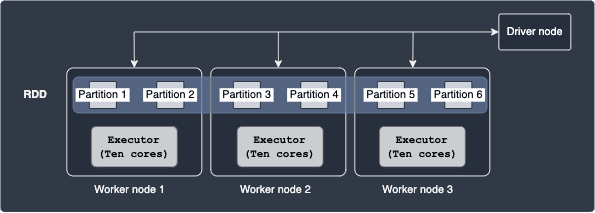
In Spark, we can choose the number of partitions by which to divide our dataset.

For instance, should we divide up our data into just a few partitions, or into hundreds of partitions?

We should choose carefully because the number of partitions has an immense impact on the cluster's performance.

Examples, let's explore the case of over-partitioning and under-partitioning.

Under-partitioning:

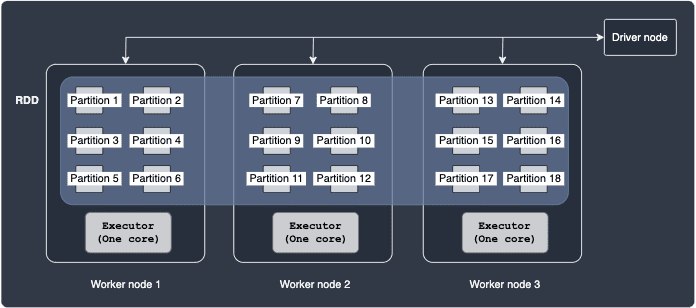


Here, each of our executors is equipped with 10 cores, but only 2 partitions reside at each node.

This means that each executor can tackle the two tasks assigned to it in parallel using just 2 cores - the other 8 cores remain unused here. In other words, we are not making use of the available cores since the number of partitions is too small, that is, we are underutilising our resources

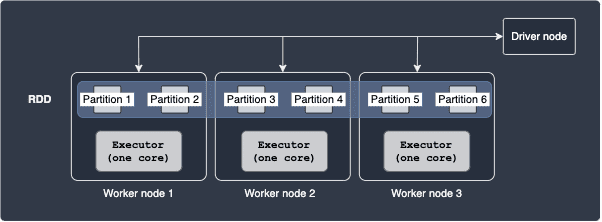
A better configuration would be to have 10 partitions on each worker node so that each executor can parse all 10 partitions on their node in parallel.

Excess partitioning



Here, we have 6 partitions residing in each worker node, which is equipped with only one CPU core. The driver would need to create and schedule the same number of tasks as there are partitions (16 in this case). There is considerable overhead in having to manage and coordinate many small tasks. Therefore, having a large number of partitions is also not desirable.

Recommended number of partitions:

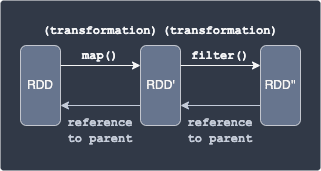


The official PySpark documentationopen\_in\_new recommends that there should be 2 to 4 partitions for each core in the executor

Fault tolerance property:

The R in RDD stands for resilient, meaning that even if a worker node fails, the missing partition can still be

recomputed to recover the RDD with the help of RDD lineage. For instance, consider the following example:



Suppose RDD'' is "damaged" because of a node failure.

Since Spark knows that RDD' is the parent of RDD'', Spark will be able to re-compute RDD'' from RDD'.

Difference between RDD and DataFrames:

When working with PySpark, we usually use DataFrames instead of RDDs.

Similar to RDDs, DataFrames are also an immutable collection of data, but the key difference is that DataFrames can be thought of as a spreadsheet-like table where the data is organised into columns.

This does limit the use-case of DataFrames to only structured or tabular data, but the added benefit is that we can work with our data at a much higher level of abstraction.

If you've ever used a Pandas DataFrame, you'll understand just how easy it is to interact with your data.

DataFrames are actually built on top of RDDs, but there are still cases when you would rather work at a lower level and tinker directly with RDDs.

For instance, if you are dealing with unstructured data (e.g. audio and streams of data), you would use RDDs rather than DataFrames.

NOTE:

If you are dealing with structured data, we highly recommend that you use DataFrames instead of RDDs.

This is because Spark will optimise the series of operations you perform on DataFrames under the hood,

but will not do so in the case of RDDs.