Spark

Apache Spark is a cluster computing framework for large-scale data processing. Unlike most of the other processing frameworks, Spark does not use MapReduce as an execution engine; instead, it uses its own distributed runtime for executing work on a cluster. However, Spark has many parallels with MapReduce, in terms of both API and runtime Spark is closely integrated with Hadoop: it can run on YARN and works with Hadoop file formats and storage backends like HDFS.

Spark is best known for its ability to keep large working datasets in memory *between jobs*. This capability allows Spark to outperform the equivalent MapReduce workflow (by an order of magnitude or more in some cases1), where datasets are always loaded from disk. Two styles of application that benefit greatly from Spark's processing model are iterative algorithms (where a function is applied to a dataset repeatedly until an exit condition is met) and interactive analysis (where a user issues a series of ad hoc exploratory queries on a dataset).

Even if you don't need in-memory caching, Spark is very attractive for a couple of other reasons: its DAG engine and its user experience. Unlike MapReduce, Spark's DAG engine can process arbitrary pipelines of operators and translate them into a single job for the user. Spark's user experience is also second to none, with a rich set of APIs for performing many common data processing tasks, such as joins. Spark provides APIs in three languages: Scala, Java, and Python.

Spark also comes with a REPL (read—eval—print loop) for both Scala and Python, which makes it quick and easy to explore datasets.

Spark is proving to be a good platform on which to build analytics tools, too, and to this end the Apache Spark project includes modules for machine learning (MLlib), graph processing (GraphX), stream processing (Spark Streaming), and SQL (Spark SQL).

SparkContext

SparkContext is the internal engine that allows the connections with the clusters.If you want to run an operation, you need a SparkContext.

```
from pyspark import SparkContext
sc=SparkContext()
```

SparkContext structure:

```
class pyspark.SparkContext (
master = None,
appName = None,
sparkHome = None,
pyFiles = None,
environment = None,
batchSize = 0,
serializer = PickleSerializer(),
conf = None,
gateway = None,
jsc = None,
profiler_cls = <class 'pyspark.profiler.BasicProfiler'>
)
```

Parameters:

- **Master :** It is the URL of the cluster it connects to.
- **appName** – Name of your job.
- **sparkHome** – Spark installation directory.
- **pyFiles** – The .zip or .py files to send to the cluster and add to the PYTHONPATH.
- **Environment** – Worker nodes environment variables.
- **batchSize** – The number of Python objects represented as a single Java object. Set 1 to disable batching, 0 to automatically choose the batch size based on object sizes, or -1 to use an unlimited batch size.
- **Serializer** – RDD serializer.
- **Conf** – An object of L{SparkConf} to set all the Spark properties.
- **Gateway** – Use an existing gateway and JVM, otherwise initializing a new JVM.
- **JSC** – The JavaSparkContext instance.
- **profiler_cls** – A class of custom Profiler used to do profiling (the default is pyspark.profiler.BasicProfiler).

Cluster Managers

The SparkContext can work with various Cluster Managers, like Standalone Cluster Manager, Yet Another Resource Negotiator (YARN), or Mesos, which allocate resources to containers in the worker nodes. The work is done inside these containers.

Standalone Cluster

Standalone Master is the Resource Manager and **Standalone Worker** is the worker in the Spark Standalone Cluster.

In the Standalone Cluster mode, there is only one executor to run the tasks on each worker node.

A client establishes a connection with the Standalone Master, asks for resources, and starts the execution process on the worker node.

Here, the client is the application master, and it requests the resources from the Resource Manager. In this Cluster Manager, we have a Web UI to view all clusters and job statistics.

Hadoop YARN (Yet Another Resource Negotiator)

YARN takes care of resource management for the Hadoop ecosystem. It has two components:

- **Resource Manager:** It manages resources on all applications in the system. It consists of a Scheduler and an Application Manager. The Scheduler allocates resources to various applications.
- **Node Manager:** Node Manager consists of an Application Manager and a Container. Each task of MapReduce runs in a container. An application or job thus requires one or more containers, and the Node Manager monitors these containers and resource usage. This is reported to the Resource Manager.

YARN also provides security for authorization and authentication of web consoles for data confidentiality. Hadoop uses Kerberos to authenticate its users and services

Working with SparkContext

```
products = sc.textFile("file:///home/hadoop/product.csv")
products.count()
```

Resilient Distributed Datasets (RDDs)

RDDs are the main logical data units in Spark. They are a distributed collection of objects, which are stored in memory or on disks of different machines of a cluster. A single RDD can be divided into multiple logical partitions so that these partitions can be stored and processed on different machines of a cluster.

RDDs are immutable (read-only) in nature. You cannot change an original RDD, but you can create new RDDs by performing coarse-grain operations, like transformations, on an existing RDD.

An RDD in Spark can be cached and used again for future transformations, which is a huge benefit for users. RDDs are said to be lazily evaluated, i.e., they delay the evaluation until it is really needed. This saves a lot of time and improves efficiency.

Features of an RDD in Spark

- **Resilience:** RDDs track data lineage information to recover lost data, automatically on failure. It is also called fault tolerance.
- **Distributed:** Data present in an RDD resides on multiple nodes. It is distributed across different nodes of a cluster.
- **Lazy evaluation:** Data does not get loaded in an RDD even if you define it. Transformations are actually computed when you call an action, such as count or collect, or save the output to a file system.
- **Immutability:** Data stored in an RDD is in the read-only mode—you cannot edit the data which is present in the RDD. But, you can create new RDDs by performing transformations on the existing RDDs.
- **In-memory computation:** An RDD stores any immediate data that is generated in the memory (RAM) than on the disk so that it provides faster access.
- **Partitioning:** Partitions can be done on any existing RDD to create logical parts that are mutable. You can achieve this by applying transformations on the existing partitions.

Creating an RDD

An RDD can be created in three ways. Let’s discuss them one by one.

By Loading an External Dataset

You can load an external file onto an RDD. The types of files you can load are csv, txt, JSON, etc.

```
products = sc.textFile("file:///home/hadoop/product.csv")
products.foreach(print)
```

By Parallelizing the Collection of Objects

When Spark's parallelize method is applied to a group of elements, a new distributed dataset is created. This dataset is an RDD.

```
numbers = sc.parallelize(range(1,11))
numbers.foreach(print)
```

By Performing Transformations on the Existing RDDs

One or more RDDs can be created by performing transformations on the existing RDDs as mentioned earlier in this tutorial page. The below figure shows how a map() function can be used to create an RDD:

```
new_numbers = numbers.map(lambda x : x+5)
new_numbers.foreach(print)
```

RDD Transformations and Actions

Transformations

These are functions that accept the existing RDDs as input and outputs one or more RDDs. However, the data in the existing RDD in Spark does not change as it is immutable. Some of the transformation operations are provided in the table below:

Function	Description
map()	Returns a new RDD by applying the function on each data element
filter()	Returns a new RDD formed by selecting those elements of the source on which the function returns true
reduceByKey()	Aggregates the values of a key using a function
groupByKey()	Converts a (key, value) pair into a (key, <iterable value>) pair
union()	Returns a new RDD that contains all elements and arguments from the source RDD

intersection()	Returns a new RDD that contains an intersection of the elements in the datasets

These transformations are executed when they are invoked or called. Every time transformations are applied, a new RDD is created.

Actions

Actions in Spark are functions that return the end result of RDD computations. It uses a lineage graph to load data onto the RDD in a particular order. After all of the transformations are done, actions return the final result to the Spark Driver. Actions are operations that provide non-RDD values. Some of the common actions used in Spark are given below:

Function	Description
count()	Gets the number of data elements in an RDD
collect()	Gets all the data elements in an RDD as an array
reduce()	Aggregates data elements into an RDD by taking two arguments and returning one
take(n)	Fetches the first *n* elements of an RDD
foreach(operation)	Executes the **operation** for each data element in an RDD
first()	Retrieves the first data element of an RDD