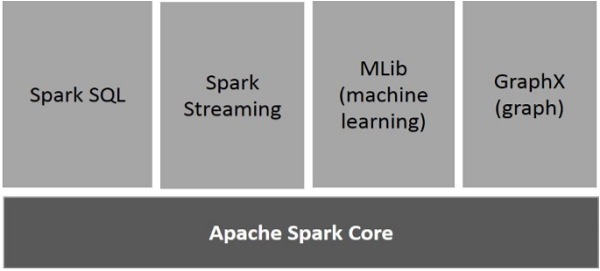
# Spark

**Intro**

* It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing
* The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application.
* three ways of how Spark can be built with Hadoop components.
  + Standalone − Spark Standalone deployment means Spark occupies the place on top of HDFS(Hadoop Distributed File System) and space is allocated for HDFS, explicitly. Here, Spark and MapReduce will run side by side to cover all spark jobs on cluster.
  + Hadoop Yarn − Hadoop Yarn deployment means, simply, spark runs on Yarn without any pre-installation or root access required. It helps to integrate Spark into Hadoop ecosystem or Hadoop stack. It allows other components to run on top of stack.
  + Spark in MapReduce (SIMR) − Spark in MapReduce is used to launch spark job in addition to standalone deployment. With SIMR, user can start Spark and uses its shell without any administrative access.
* The main abstraction Spark provides a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated in parallel.
* RDDs are created by starting with a file in the Hadoop file and transforming it.
* Spark will also persist an RDD in memory, allowing it to be reused efficiently across parallel operations.
* Finally, RDDs automatically recover from node failures.
* A second abstraction in Spark is shared variables that can be used in parallel operations.
* By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task.
* Sometimes, some variable needs to be shared across tasks, or between tasks and the driver program.
* Spark supports two types of shared variables:
  + broadcast variables, which can be used to cache a value in memory on all nodes
  + accumulators, which are variables that are only “added” to, such as counters and sums.

**Components of Spark**

The following illustration depicts the different components of Spark.



**Resilient Distributed Dataset (RDD)**

* is a immutable  / fault-tolerant collection of elements that can be operated in parallel.
* There are currently two types of RDDs:
  + parallelized collections, which take an existing Scala collection and run functions on it in parallel. - are created by calling SparkContext’s parallelize method on an existing Scala collection
  + Hadoop datasets, which run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop.
  + There are two ways to create RDDs − parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.
* Both types of RDDs can be operated on through the same methods. - Spark supports text files(textFile method), [SequenceFiles](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapred/SequenceFileInputFormat.html), (sequenceFile[K, V] method) and any other Hadoop InputFormat
* RDDs support two types of operations:
  + transformations, which create a new dataset from an existing one
  + actions, which return a value to the driver program after running a computation on the dataset.
* For example, map is a transformation that passes each dataset element through a function and returns a new distributed dataset representing the results. On the other hand, reduce is an action that aggregates all the elements of the dataset using some function and returns the final result to the driver program
* All transformations in Spark are *lazy*, in that they do not compute their results right away. The transformations are only computed when an action requires a result to be returned to the driver program.

|  |  |
| --- | --- |
| **Transformation** | **Meaning** |
| **map**(*func*) | Return a new distributed dataset formed by passing each element of the source through a function *func*. |
| **filter**(*func*) | Return a new dataset formed by selecting those elements of the source on which *func* returns true. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item). |
| **mapPartitions**(*func*) | 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. |
| **mapPartitionsWithIndex**(*func*) | Similar to mapPartitions, but also 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. |
| **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. |
| **distinct**([*numTasks*])) | Return a new dataset that contains the distinct elements of the source dataset. |
| **groupByKey**([*numTasks*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs.  **Note:** By default, this uses only 8 parallel tasks to do the grouping. You can pass an optional numTasksargument to set a different number of tasks. |
| **reduceByKey**(*func*, [*numTasks*]) | 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. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. |
| **sortByKey**([*ascending*], [*numTasks*]) | 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*, [*numTasks*]) | 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. |
| **cogroup**(*otherDataset*, [*numTasks*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[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). |

A complete list of transformations is available in the [RDD API doc](https://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.rdd.RDD).

### Actions

|  |  |
| --- | --- |
| **Action** | **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. |
| **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. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements. |
| **takeSample**(*withReplacement*, *num*, *seed*) | Return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed. |
| **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*) | 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 only available on RDDs of key-value pairs that either implement Hadoop's Writable interface or are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). |
| **countByKey**() | Only available on RDDs of type (K, V). Returns a `Map` 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 variable (see below) or interacting with external storage systems. |

**Spark with Java**

The Spark Java API is defined in the [org.apache.spark.api.java](https://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.api.java.package) package, and includes a [JavaSparkContext](https://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.api.java.JavaSparkContext) for initializing Spark and [JavaRDD](https://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.api.java.JavaRDD) classes, which support the same methods as their Scala counterparts but take Java functions and return Java data and collection types.

There are a few key differences between the Java and Scala APIs:

* Java does not support anonymous or first-class functions, so functions must be implemented by extending the [org.apache.spark.api.java.function.Function](https://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.api.java.function.Function), [Function2](https://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.api.java.function.Function2), etc. classes.
* To maintain type safety, the Java API defines specialized Function and RDD classes for key-value pairs and doubles. For example,[JavaPairRDD](https://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.api.java.JavaPairRDD) stores key-value pairs.
* RDD methods like collect() and countByKey() return Java collections types, such as java.util.List and java.util.Map.
* Key-value pairs, which are simply written as (key, value) in Scala, are represented by the scala.Tuple2 class, and need to be created using new Tuple2<K, V>(key, value).

Spark Deployment – spark-submit

spark-submit [options] <app jar | python file> [app arguments]

spark-submit --class SparkWordCount --master local wordcount.jar

If you carefully read the following output, you will find different things, such as −

* successfully started service 'sparkDriver' on port 42954
* MemoryStore started with capacity 267.3 MB
* Started SparkUI at http://192.168.1.217:4040
* Added JAR file:/home/hadoop/piapplication/count.jar
* ResultStage 1 (saveAsTextFile at SparkPi.scala:11) finished in 0.566 s
* Stopped Spark web UI at http://192.168.1.217:4040
* MemoryStore cleared

**Spark Streaming**

* Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
* Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark’s [machine learning](https://spark.apache.org/docs/2.2.0/ml-guide.html) and [graph processing](https://spark.apache.org/docs/2.2.0/graphx-programming-guide.html) algorithms on data streams
* Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data.

Word Count

* we create a [JavaStreamingContext](https://spark.apache.org/docs/2.2.0/api/java/index.html?org/apache/spark/streaming/api/java/JavaStreamingContext.html) object, which is the main entry point for all streaming functionality.

SparkConf conf = **new** SparkConf().setMaster("local[2]").setAppName("NetworkWordCount");

JavaStreamingContext jssc = **new** JavaStreamingContext(conf, Durations.seconds(1));

JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999)

JavaDStream<String> words = lines.flatMap(x -> Arrays.asList(x.split(" ")).iterator());

JavaPairDStream<String, Integer> pairs = words.mapToPair(s -> **new** Tuple2<>(s, 1));

JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey((i1, i2) -> i1 + i2);

*// Print the first ten elements of each RDD generated in this DStream to the console*

wordCounts.print();

jssc.start(); *// Start the computation*

jssc.awaitTermination();

## Discretized Streams (DStreams)

**Discretized Stream** or **DStream** is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream.

Input DStreams are DStreams representing the stream of input data received from streaming sources.

SparkConf conf = **new** SparkConf().setAppName("DGAAppZone");

Duration duration = **new** Duration(600);

JavaStreamingContext streamingContext = **new** JavaStreamingContext(conf, duration);

String topic = kafkaProperties.getKafkaPmtTopicIn();

Collection<String> topics = Arrays.*asList*(topic);

Map<String, Object> kafkaParams = getKafkaParms();

JavaInputDStream<ConsumerRecord<String, String>> stream = KafkaUtils.*createDirectStream*(streamingContext,

LocationStrategies.*PreferConsistent*(),

ConsumerStrategies.<String, String>*Subscribe*(topics, kafkaParams));

stream.foreachRDD(rdd -> {

**try** {

System.***out***.println("Starting foreachRDD. Count: " + rdd.count());

**if** (rdd != **null** && !rdd.isEmpty()) { rdd.foreach(record -> {

}

}

}

streamingContext.start();

streamingContext.awaitTermination();