## Java and Spark

**[Java and Spark 1](#_Toc1640)**

**[Youtube playlist 1](#_Toc19020)**

**[Github main 1](#_Toc18796)**

**[References 2](#_Toc3735)**

**[Chapter 01. The Big Picture 3](#_Toc20129)**

**[Big Data 3](#_Toc6736)**

**[Local versus Distributed Systems 4](#_Toc9358)**

**[Apache Hadoop and MapReduce 4](#_Toc19428)**

**[Apache Spark 5](#_Toc6626)**

**[Spark RDDs 5](#_Toc24920)**

**[Youtube 6](#_Toc15963)**

**[Github 6](#_Toc30226)**

**[Chapter 02. Project Setup - Maven 7](#_Toc26589)**

**[Git clone Hadoop to local 7](#_Toc30744)**

**[Youtube 7](#_Toc17731)**

**[Github 7](#_Toc6172)**

**[Chapter 03. Spark RDD - First Program 8](#_Toc5295)**

**[Initializing Spark 8](#_Toc4876)**

**[Youtube 9](#_Toc356)**

**[Github 9](#_Toc26881)**

**[Chapter 04. Spark RDD - Reduces 10](#_Toc3441)**

**[Youtube 10](#_Toc29885)**

**[Github 10](#_Toc19894)**

**[Chapter 05. Spark RDD - Mapping 11](#_Toc12322)**

**[Youtube 11](#_Toc13065)**

**[Github 11](#_Toc5440)**

**[Chapter 06. Spark RDD - Printing elements 12](#_Toc12677)**

**[Youtube 12](#_Toc18667)**

**[Github 13](#_Toc24390)**

**[Chapter 07. Spark RDD - External Datasets 13](#_Toc25343)**

**[Youtube 14](#_Toc11813)**

**[Github 14](#_Toc2485)**

**[Chapter 08. Spark RDD - Tuples 15](#_Toc16429)**

**[Youtube 15](#_Toc188)**

**[Github 15](#_Toc5345)**

**[Chapter 09. Spark RDD - PairRDDs 16](#_Toc4587)**

**[Youtube 16](#_Toc32141)**

**[Github 17](#_Toc9426)**

#### Youtube playlist

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

#### Github main

<https://github.com/backstreetbrogrammer/11_JavaSpark>

#### References

<https://spark.apache.org/>

<https://spark.apache.org/docs/latest/api/java/index.html>

<https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples>

#### Chapter 01. The Big Picture

###### Big Data

Big data is a term that describes large, hard-to-manage volumes of data – both structured and unstructured – that inundate businesses on a day-to-day basis.

These data sets are so voluminous that traditional data processing software just can’t manage them. But these massive volumes of data can be used to address business problems we wouldn’t have been able to tackle before.

The three **Vs** of big data:

**Volume**

The amount of data matters. With big data, we’ll have to process high volumes of low-density, unstructured data. This can be data of unknown value, such as Twitter data feeds, click-streams on a web page or a mobile app, or sensor-enabled equipment. For some organizations, this might be **tens of terabytes** of data. For others, it may be **hundreds of petabytes**.

**Velocity**

Velocity is the fast rate at which data is received and (perhaps) acted on. Normally, the highest velocity of data streams directly into memory versus being written to disk. Some internet-enabled smart products operate in real time or near real time and will require real-time evaluation and action.

**Variety**

Variety refers to the many types of data that are available. Traditional data types were structured and fit neatly in a relational database. With the rise of big data, data comes in new **unstructured** data types. Unstructured and semi-structured data types, such as text, audio, and video, require additional preprocessing to derive meaning and support metadata.

|  |  |  |
| --- | --- | --- |
| ****Name**** | ****Value**** | ****Value**** |
| kilobyte (kB) | 10^3 | 2^10 |
| megabyte (MB) | 10^6 | 2^20 |
| gigabyte (GB) | 10^9 | 2^30 |
| terabyte (TB) | 10^12 | 2^40 |
| petabyte (PB) | 10^15 | 2^50 |
| exabyte (EB) | 10^18 | 2^60 |
| zettabyte (ZB) | 10^21 | 2^70 |
| yottabyte (YB) | 10^24 | 2^80 |

###### Local versus Distributed Systems

Big data can not be processed or stored in a local system or a single node. It requires multiple machines or nodes to store / process it.

Master Node => Slave node**s**

A local single node will use the computation sources (CPU, cores) and storage (memory, hard disk) of a single machine only. Only **vertical scaling** is possible which means we can add powerful CPU or memory to a single machine but there will be a limit to it. Single point of failure if the local node goes down which makes it essential to store the important data in cloud or separate disk.

A distributed system has access to the computation sources (CPU, cores) and storage (memory, hard disk) across a number of machines connected through a network. **Horizontal scaling** is easier by just adding new nodes or systems to the distributed system. It also supports **fault tolerance**, if one machine fails, the whole network can still go on.

###### Apache Hadoop and MapReduce

Apache Hadoop is a collection of open-source software utilities that facilitates using a network of many computers to solve problems involving massive amounts of data and computation. It provides a software framework for distributed storage and processing of big data using the **MapReduce** programming model.

Hadoop uses **Hadoop Distributed File System** (**HDFS**) which is a distributed, scalable, and portable file system written in Java for the Hadoop framework and allows user to work with large data sets. It also duplicates blocks of data for **fault tolerance**.

HDFS uses **MapReduce** which allows computations on that data.

HDFS uses blocks of data of default size 128 MB and replicates it multiple times to the slave nodes for fault tolerance.

**MapReduce** is a way of splitting a computational task to a distributed set of files such as HDFS. It consists of a **Job Tracker** at Master Node and multiple **Task Trackers** in the slave nodes. Job Tracker sends code to run on the Task Trackers. The Task Trackers allocate CPU and memory for the tasks and monitor the tasks on the worker nodes.

To summarize,

* **HDFS** is used to distribute large data sets
* **MapReduce** is used to distribute a computational task to a distributed data set

###### Apache Spark

Apache Spark is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

Key Features:

* SQL analytics using **RDDs** and **SparkSQL**

Execute fast, distributed ANSI SQL queries for dashboarding and ad-hoc reporting. Runs faster than most data warehouses.

* Machine learning using **SparkML**

Train machine learning algorithms on a laptop and use the same code to scale to fault-tolerant clusters of thousands of machines.

* Batch/streaming data using **Spark Streaming**

Unify the processing of data in batches and real-time streaming.

**Spark** is a flexible alternative to **MapReduce**.

**MapReduce** requires files to be stored only in HDFS, while **Spark** can work on data stored in a variety of formats like HDFS, AWS S3, Cassandra, HBase etc.

**Spark** can perform operations up to 100X faster than **MapReduce** because MapReduce writes most of data to **disk** after each map and reduce operation; however Spark keeps most of the data in **memory** after each transformation. Spark will write to disk only when the memory is full.

###### Spark RDDs

**RDD (Resilient Distributed Dataset)** is the fundamental data structure of Apache Spark which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in Spark RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster.

RDD has these main features:

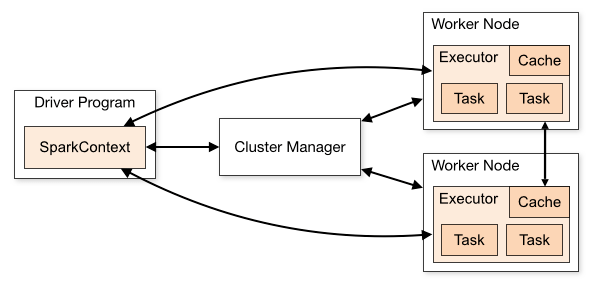
* Distributed collection of data
* Immutable, lazily evaluated and cacheable
* Fault-tolerant “in-memory” computations
* Parallel operation - partitioned
* Ability to use many data sources

RDDs support 2 kinds of operations:

1. **Transformation** – Spark RDD transformation is a function that produces new RDD from the existing RDDs. The transformer takes RDD as input and produces one or more RDD as output. Transformations are lazy in nature i.e., they get execute when we call an action.
2. **Action** – transformations create RDDs from each other, but when we want to work with the actual data set, at that point action is performed. Thus, Actions are Spark RDD operations that give **non-RDD** values. The values of action are stored to drivers or to the external storage system.

An action is one of the ways of sending data from **Executor** to the driver.

**Executors** are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. Some actions of Spark are count and collect.



###### Youtube

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

###### Github

<https://github.com/backstreetbrogrammer/11_JavaSpark>

#### Chapter 02. Project Setup - Maven

We can create a Maven project and add spark dependencies.

<dependency>  
 <groupId>org.apache.spark</groupId>  
 <artifactId>spark-core\_2.13</artifactId>  
 <version>${apache-spark.version}</version>  
</dependency>  
  
<dependency>  
 <groupId>org.apache.spark</groupId>  
 <artifactId>spark-sql\_2.13</artifactId>  
 <version>${apache-spark.version}</version>  
</dependency>  
  
<dependency>  
 <groupId>org.apache.hadoop</groupId>  
 <artifactId>hadoop-hdfs</artifactId>  
 <version>3.3.2</version>  
</dependency>

Latest apache spark version as of this writing is **3.2.2**

Complete **pom.xml** can be found at Github: <https://github.com/backstreetbrogrammer/11_JavaSpark/pom.xml>

###### Git clone Hadoop to local

1. Go and clone this repo: **https://github.com/cdarlint/winutils** in Windows machine.

Please clone the entire repo because doing that will help to use any version of **winutils.exe**

1. Set **HADOOP\_HOME** environment variable to the root path of latest version
2. Add to the **PATH** environment, the **%HADOOP\_HOME%\bin**
3. Edit **Application** and **Junit** templates in **IntelliJ** and add **VM arguments** as:

-Dhadoop.home.dir=C:\\Users\\rishi\\Downloads\\BuildWithTech\\winutils\\hadoop-3.2.2

###### Youtube

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

###### Github

<https://github.com/backstreetbrogrammer/11_JavaSpark>

#### Chapter 03. Spark RDD - First Program

Every Spark application consists of a **driver program** that runs the user’s *main* function and executes various parallel operations on a cluster. Spark provides **RDD**, which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.

RDDs are created by starting with a file in **HDFS** or other supported file systems, or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in **memory**, allowing it to be reused efficiently across parallel operations. Also, RDDs automatically recover from node failures.

Spark uses **shared variables** 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, a variable needs to be shared across tasks, or between tasks and the driver program.

Spark supports 2 types of shared variables:

* **broadcast variables** => to cache a value in memory on all nodes
* **accumulators** => variables that are only “added” to, such as counters and sums

###### Initializing Spark

1. Build a **SparkConf** object that contains information about the application

```

final var conf = new SparkConf().setAppName("SparkFirstProgram").setMaster("local[\*]");

```

The `**appName**` parameter is a name for the application to show on the cluster UI.

The `**master**` is a Spark, Mesos or YARN cluster URL, or a special “local” string to run in local mode. When running on a cluster, we will not want to hardcode master in the program, but rather launch the application with `**spark-submit**` and receive it there. However, for local testing and unit tests, we can pass “local” to run Spark in-process.

2. Create a **JavaSparkContext** object which tells Spark how to access a cluster, by passing the **SparkConf** object to its constructor

```

final var sc = new JavaSparkContext(conf);

```

3. Create RDD which is a fault-tolerant collection of elements that can be operated on in parallel

There are two ways to create RDDs:

* **parallelizing** an existing collection in the driver program
* referencing a dataset in an **external storage system**, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop `**InputFormat**`

Parallelized collections are created by calling JavaSparkContext’s `**parallelize()**` method on an existing `**Collection**` in the driver program. The elements of the collection are copied to form a **RDD** that can be operated on in parallel.

```

final var data = List.of(165, 254, 124656, 356838, 64836);

final var myRdd = sc.parallelize(data);

```

RDD created `**myRdd**` can be operated on in parallel. These operations can be to **reduce**, **map**, etc.

```

final var max = myRdd.reduce(Integer::max);

final var min = myRdd.reduce(Integer::min);

final var sum = myRdd.reduce(Integer::sum);

```

One important parameter for parallel collections is the number of partitions to cut the dataset into. Spark will run one task for each partition of the cluster.

We may want 2-4 partitions for each CPU in the cluster. Spark tries to set the number of partitions automatically based on our cluster.

However, we can also set it manually by passing it as a second parameter to **parallelize()** method.

```

sc.parallelize(data, 10)

```

###### Youtube

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

###### Github

<https://github.com/backstreetbrogrammer/11_JavaSpark>

#### Chapter 04. Spark RDD - Reduces

As discussed, once `**JavaRDD**` object is created, it can be used to perform `**reduce**` operation.  
  
In functional programming-language jargon, this is referred to as a **fold** because we can view this operation as repeatedly folding a long piece of paper (our stream) until it forms a small square, which is the result of the **fold** operation.  
  
Example:  
  
```  
final var max = myRdd.reduce(Integer::max);  
final var min = myRdd.reduce(Integer::min);  
final var sum = myRdd.reduce(Integer::sum);  
```

###### Youtube

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

###### Github

<https://github.com/backstreetbrogrammer/11_JavaSpark>

#### Chapter 05. Spark RDD - Mapping

**Transformation: map(func)**  
> Return a new distributed dataset formed by passing each element of the source through a function `**func**`.  
  
As RDDs are immutable, after applying the `**map**` transformation, **new RDD** is created.  
  
The `**func**` is applied to each element, mapping it into a new element (the word mapping is used because it has a meaning similar to transforming but with the nuance of “creating a new version of” rather than “modifying”).  
  
Example:  
  
```  
final var myList = myRdd.map(String::length).collect();  
final var count = myRdd.map(String::length).count();  
final var count = myRdd.map(String::length).map(v -> 1L).reduce(Long::sum);  
```

###### Youtube

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

###### Github

<https://github.com/backstreetbrogrammer/11_JavaSpark>

#### Chapter 06. Spark RDD - Printing elements

We can print out the elements of an RDD using:  
  
```  
rdd.foreach(println)

OR

rdd.map(println)   
```  
  
On a **single** machine, this will generate the expected output and print all the RDD’s elements.  
  
However, in **cluster** mode, the output to *stdout* being called by the **executors** is now writing to the **executor’s** *stdout* instead, not the one on the **driver**, so *stdout* on the **driver** won’t show these.  
  
Also, we may see `**NotSerializableException**` in few cases when the data is huge.  
  
To print all elements on the **driver** one can use the `**collect()**` method to first bring the RDD to the **driver** node:  
  
```  
rdd.collect().foreach(println)  
```  
  
This can cause the driver to run **out of memory**, though, because `**collect()**` fetches the entire RDD to a **single** machine; if we only need to print a few elements of the RDD, a safer approach is to use the `**take()**` method:  
  
```  
rdd.take(100).foreach(println)  
```

###### Youtube

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

###### Github

<https://github.com/backstreetbrogrammer/11_JavaSpark>

#### Chapter 07. Spark RDD - External Datasets

Spark can create distributed datasets from any storage source supported by Hadoop like:  
  
- Local file system  
- HDFS  
- Cassandra  
- HBase  
- Amazon S3, etc.  
  
Spark supports:  
  
- Text files  
- SequenceFiles  
- Any Hadoop *InputFormat*  
  
**Text file** RDDs can be created using `**SparkContext**`’s `**textFile()**` method. This method takes a **URI** for the file (either a local path on the machine, or a `**hdfs://**`, `**s3a://**`, etc. URI) and reads it as a collection of lines.  
  
Example:  
  
```  
JavaRDD<String> dataFile = sc.textFile("data.txt");  
```  
  
Once created, `**dataFile**` can be acted on by dataset operations like **map** or **reduce**.  
  
Few important points to read files in Spark:  
  
- If using a path on the local filesystem, the file must also be accessible at the same path on **worker nodes**. Either copy the file to all workers or use a network-mounted shared file system.

- All of Spark’s file-based input methods, including `**textFile**`, support running on **directories**, **compressed** files, and **wildcards** as well. For example, we can use:  
  
```  
sc.textFile("/my/directory")  
sc.textFile("/my/directory/\*.txt")  
sc.textFile("/my/directory/\*.gz")  
```  
  
- The `**textFile()**` method also takes an optional **second** argument for controlling the number of partitions of the file. By default, Spark creates **one** partition for each block of the file (blocks being `**128MB**` by default in HDFS), but we can also ask for a higher number of partitions by passing a larger value. Note that we cannot have fewer partitions than blocks.  
  
Apart from **text files**, Spark’s Java API also supports several other data formats:  
  
- `**JavaSparkContext.wholeTextFiles()**` lets us read a directory containing multiple small text files, and returns each of them as **(filename, content)** pairs. This is in contrast with `**textFile()**`, which would return one record per line in each file.

- For **SequenceFiles**, use SparkContext’s `**sequenceFile[K, V]**` method where `K` and `V` are the types of key and values in the file. These should be subclasses of Hadoop’s **Writable** interface, like **IntWritable** and **Text**.

- For other Hadoop `**InputFormats**`, we can use the `**JavaSparkContext.hadoopRDD()**` method, which takes an arbitrary `**JobConf**` and input format class, key class and value class. Set these the same way we would for a Hadoop job with our input source. We can also use `**JavaSparkContext.newAPIHadoopRDD()**` for `**InputFormats**` based on the “new” MapReduce API (`**org.apache.hadoop.mapreduce**`).

- `**JavaRDD.saveAsObjectFile()**` and `**JavaSparkContext.objectFile()**` support saving an RDD in a simple format consisting of **serialized** Java objects. While this is not as efficient as specialized formats like **Avro**, it offers an easy way to save any RDD.

###### Youtube

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

###### Github

<https://github.com/backstreetbrogrammer/11_JavaSpark>

#### Chapter 08. Spark RDD - Tuples

In Scala, a **Tuple** is a value that contains a fixed number of elements, each with its own type. Tuples are **immutable**.

Tuples are especially handy for returning **multiple** values from a method.  
  
A tuple with two elements can be created as follows:  
  
```  
val person = ("John", 25)  
```  
  
This creates a tuple containing a `**String**` element and an `**Int**` element. The inferred type of `**person**` is `**(String, Int)**`.  
  
Tuples are of type `**Tuple1**`, `**Tuple2**`, `**Tuple3**` and so on. There currently is an upper limit of **22** in the Scala if we need more. For each `**TupleN**` type, where `**1 <= N <= 22**`, Scala defines a number of element-access methods.  
  
Example:  
  
```  
final var tuple2JavaRDD = myRdd.map(line -> new Tuple2<>(line, line.length()));  
```

###### Youtube

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

###### Github

<https://github.com/backstreetbrogrammer/11_JavaSpark>

#### Chapter 09. Spark RDD - PairRDDs

While most Spark operations work on RDDs containing any type of objects, a few special operations are only available on RDDs of **key-value pairs**. The most common ones are distributed “**shuffle**” operations, such as grouping or aggregating the elements by a key.  
  
In Java, key-value pairs are represented using the `**scala.Tuple2**` class from the Scala standard library. We can simply call `**new Tuple2(a, b)**` to create a tuple, and access its fields later with `**tuple.\_1()**` and `**tuple.\_2()**`.  
  
RDDs of key-value pairs are represented by the `**JavaPairRDD**` class. We can construct `**JavaPairRDDs**` from `**JavaRDDs**` using special versions of the **map** operations, like `**mapToPair**` and `**flatMapToPair**`. The `**JavaPairRDD**` will have both standard RDD functions and special key-value ones.  
  
For example, the following code uses the `**reduceByKey**` operation on key-value pairs to count how many times each line of text occurs in a file:  
  
```  
final var lines = sc.textFile("data.txt");  
final var pairs = lines.mapToPair(s -> new Tuple2(s, 1));  
final var counts = pairs.reduceByKey((a, b) -> a + b);  
```  
  
We could also use `**counts.sortByKey()**`, for example, to sort the pairs alphabetically, and finally `**counts.collect()**` to bring them back to the driver program as an array of objects.  
  
When using custom objects as the key in key-value pair operations, we must be sure that a custom `**equals()**` method is accompanied by a matching `**hashCode()**` method.

###### Youtube

<https://www.youtube.com/playlist?list=PLQDzPczdXrTgqEc0uomGYDS0SFu7qY3g3>

###### Github

<https://github.com/backstreetbrogrammer/11_JavaSpark>