# Spark setup

**The best materials:**

* [www.cloudduggu.com](https://www.cloudduggu.com/spark/installation-multi-node/) - spark multi node set up with hadoop.
* [www.confessionsofadataguy.com](https://www.confessionsofadataguy.com/create-your-very-own-apache-spark-hadoop-cluster-then-do-something-with-it/) - Setup of both Hadoop and Spark. there is no info about on which nodes (servers) execute which commands in terminal for setting up hadoop. But the part for Spark seems fine.

**Other materials:**

* [youtube](https://www.youtube.com/watch?v=f_XsaYcETnI) – spark multi node set up on hadoop
* [youtube](https://www.youtube.com/watch?v=-5TSKMXAygc) – spark multi node set up. I am not sure if it is using hadoop.
* [data-flair.training/blogs](https://data-flair.training/blogs/install-apache-spark-multi-node-cluster/) - spark multi node set up on hadoop

# Materials

## Repositories

Here are my repositories related to Spark:

* [github - hadoop\_spark](https://github.com/bulka4/hadoop_spark) – Running a multinode HDFS, Yarn and Spark cluster on Azure Linux VMs.

## Spark theory

Here are videos explaining Spark theory:

* [youtube - Data Engineering](https://www.youtube.com/watch?v=Tyg1FVNq40g&list=PLGhXxbu7qYooyn_aWk1DqpIF1CjBzaSUn&index=3) – Hadoop and Spark (9h video)
* [youtube - Data Engineering](https://www.youtube.com/watch?v=OgS0noWVPJ4&list=PLLa_h7BriLH0FzTY5aBFpH-vciOiEf4Br&index=4) – Spark

## Submitting Spark Jobs

* [youtube - codeWithYu](https://www.youtube.com/watch?v=o_pne3aLW2w) – submit spark job through Airflow.
* [stackoverflow](https://stackoverflow.com/questions/53344285/is-there-a-way-to-submit-spark-job-on-different-server-running-master) - submit spark job through Airflow.

# Spark theory

## DAG

DAG stands for Directed Acyclic Graph and in Spark it represents the ligical execution plan of a job.

It consists of stages and tasks that define what calculations on data are being done.

## SparkContext

It represents a connection between our Spark application and the Spark cluster.

It initializes the Spark application and connects it to the cluster resource manager (like YARN or Kubernetes).

SparkContext manages the job scheduling, task dispatching to executors, and monitoring**.**

In Python (and other languages), you typically use SparkSession, which internally manages the SparkContext.

## Data partitioning

When we make calculations on data in Spark, that data is divided into smaller chunks called partitions.

Each partition contains a subset of table records.

On each partition calculations are done separately and in parallel.

For example we might have a table with columns clientName and amount which will be broken into two partitions like that:

A screenshot of a computer

AI-generated content may be incorrect.

## Spark Jobs, Stages and Tasks

A job is trigerred by a Spark function (like count(), collect() or save()). It represents the entire computation that needs to be done.

Each job is broken down into stages. Each stage is a set of tasks that can be executed in parallel without requring any shuffle between them (more about shuffling later on in the ‘Shuffling’ section).

A stage ends before a shuffle is needed and a new stage begins after the shuffle.

An example of how stages look like is in the ‘Wide transformation example – aggregateByKey() function’ section later on.

A task is the smallest unit of work. Each task is executed on a single data partition. They are executed in parallel across the cluster.

## Input and output partitions, and final output

Each task performs transformations on a single data partition which is called an **input partition.**

A result of those transformations is also a single data partition and it is called an **output partition.**

How the final output is created:

* A single job can consist of multiple tasks
* Each task takes as input a single data partition and produces a single output partition
* All the output partitions are combined together to create a final job output.

## Narrow and wide transformation, shuffling

We talk about narrow transformations if every output partition depends only on one input partition.

We talk about wide transformations if every output partition can depend on multiple input partitions.

In the below sections are examples to illustrate this.

### Narrow transformation example – filter() function

For example if we have input partitions like that, for a table with clientName and amount columns:

A screenshot of a computer

AI-generated content may be incorrect.

Then the filter() function will be a narrow transformation.

If we filter for ‘Alice’, then it will produce output partitions like that:

A screen shot of a computer

AI-generated content may be incorrect.

So values of every output partition depends only on one input partition.

As the final output we join both output partitions:

A screenshot of a computer

AI-generated content may be incorrect.

### Wide transformation example – aggregateByKey() function

If we take the same input partitions as in the previous example and use the aggregateByKey() function in order to calculate a sum for each key (a client name) then it will perform the following actions:

* Stage 1 – Local aggregation + Shuffle write
* Stage 2 - Shuffle read + grouping + final aggregation

More information about shuffling can be found in the next section ‘Shuffling’.

Here is detailed explanation of performed stages and tasks:

* Stage 1 – Local aggregation + Shuffle write
  + One map task per input partition
  + Each task does:
    - Local aggregation by key (e.g., sum values for Alice, John)
    - Writes records for each key to separate shuffle files

A computer screen shot of a black background with white text

AI-generated content may be incorrect.

* Stage 2 - Shuffle read + grouping + final aggregation
  + Read all data for each key (from shuffle files)

A group of green and white text

AI-generated content may be incorrect.

* + Group values for each key

A screenshot of a computer

AI-generated content may be incorrect.

* + Aggregate data in each partition (that creates an output partitions)

A screenshot of a computer screen

AI-generated content may be incorrect.

* + All the 3 above operations are the same task (reduce)

So in this case values in each output partition depends on values from both input partitions and that’s why it is a wide transformation and we need to use shuffling.

### Shuffling

Shuffling is used in wide transformations, such as aggregateByKey(), groupByKey() or join(), where we perform aggregations or join tables.

A key is a value from a column on which we are aggregating or joining.

In those transformations, records with the same key needs to be processed together, so they need to be in the same partition.

That’s why we need shuffling which involves:

* Shuffle write – saving records for each key and partition to a separate shuffle file
* Shuffle read – reading records from shuffle files such that records for each key goes into the same partition

Example of shuffling is shown in the previous section ‘Wide transformation example’.

## Calculations performance

Generaly in order to improve performance we want to avoid moving records between partitions (shuffling).

There are different strategies for efficient data partitioning which can increase performance.

For now this documentation doesn’t cover it in more detail.

## Saving data

When we save data using df.write, then Spark always saves this data as set of files in a folder.

So folder represents a table and each time we append data to that table using df.write, we create a new file in that folder.

### Data partitioning

When we want to save a big dataframe in a file using df.write, then Spark might split this table into multiple files.

### Data corruption

If we have already saved a table, that is a folder consisting of mutiple files, and then we try to add more that to that folder which have different columns and data types, it will work.

We will create a new file which have different columns and data types than other files in that folder creating a table and that will cause problems. If we want to read data from that table after that we might not be able to do that.

That’s why it is recommended to use additional tools which can prevent that, for example Delta lake.

# Submitting Spark jobs

We can submit Spark jobs in two primary deployment modes:

* Local mode
* Cluster mode

Depending on environment we can submit Spark jobs using different tools, for example:

* Running a Python script
* Using spark-submit or pyspark CLI tool
* Using Kubernetes Spark Operator and SparkApplication CRD.

## Local mode

When we run Spark in a local mode, that means it is running on a single machine. Then we have only one JVM (Java virtual machine) process.

In order to run Spark in a local mode, we can use one of the following commands:

* Python script.py
* spark-submit --master local[\*]" my\_script.py
* pyspark --master local[\*]" my\_script.py

Instead of specifying a master in the above commands we can do it in the script when creating a SparkSession:

* SparkSession.builder.master("local[\*]").getOrCreate()

## Cluster mode

When we run Spark in a cluster mode, that means it is running on multiple machines in a distributed way. We have then multiple JVM porcesses (master and workers) running on different machines.

Cluster mode requires to use a resource manager like YARN, Kubernetes or Standalone Spark.

How we submit Spark jobs depends on which resource manager we are using.

### Standalone Spark

In order to run Spark using the Standalone Spark resource manager (i.e run it in the Standalone mode) we need to use one of those commands:

* spark-submit --master spark://<master-ip>:7077 my\_script.py
* pyspark --master spark://<master-ip>:7077 my\_script.py

Where master-ip is a IP address of the Spark Master node.

We can specify master in the command like shown above or we can specify it in the script when creating a SparkSession:

* SparkSession.builder.master("spark://<master-node-ip-address>:7077").getOrCreate()

### YARN

In order to submit a Spark job using YARN we use the same methods as in the Standalone Spark, that is we use either spark-submit or pyspark CLI tools. We just need to specify a different master:

* spark-submit --master yarn my\_script.py

More information about running Spark with YARN can be found in the ‘Spark on YARN’ document in the same folder as this one.

### Kubernetes

In order to submit a Spark job using Kubernetes we can use the spark-submit CLI tool like in case of YARN:

A computer screen with white text

AI-generated content may be incorrect.

Or we can also use the Spark Operator and SparkApplication CRD.

In that case we create a YAML file where we provide parameters specifying how the Spark job will be executed.

More information about running Spark with Kubernetes can be found in the ‘Spark on Kubernetes document in the same folder as this one.

# Monitoring

We can use Prometheus, Grafana and Spark History Server to monitor our Spark jobs.

Prometheus will gather metrics about Spark Jobs.

Grafana will create visual dashboards using metrics gathered by Prometheus.

Spark History Server gives more detail about Spark jobs.

# Code development

In order to develop Spark code we can run Spark in a Local mode (on a single machine) and Jupyter Notebook (or JupyterLab or JupyterHub) in a Docker container running on a Linux VM in cloud.

Then we can connect to the Jupyter through a browser from our local computer to develop and run a Spark code.

Additionally we can use VS code with the following extensions:

* Remote-SSH - In order to connect VS code to that VM
* Dev Containers – In order to access a filesystem inside of the Docker container running on the VM.

This will give us a possibility to browse a filesystem inside of the Docker container and use terminal through VS code.

It is used in the spark\_kubernetes repository.