# Spark Applications

Jorge Acosta Hernández
jorge.acosta@upm.es

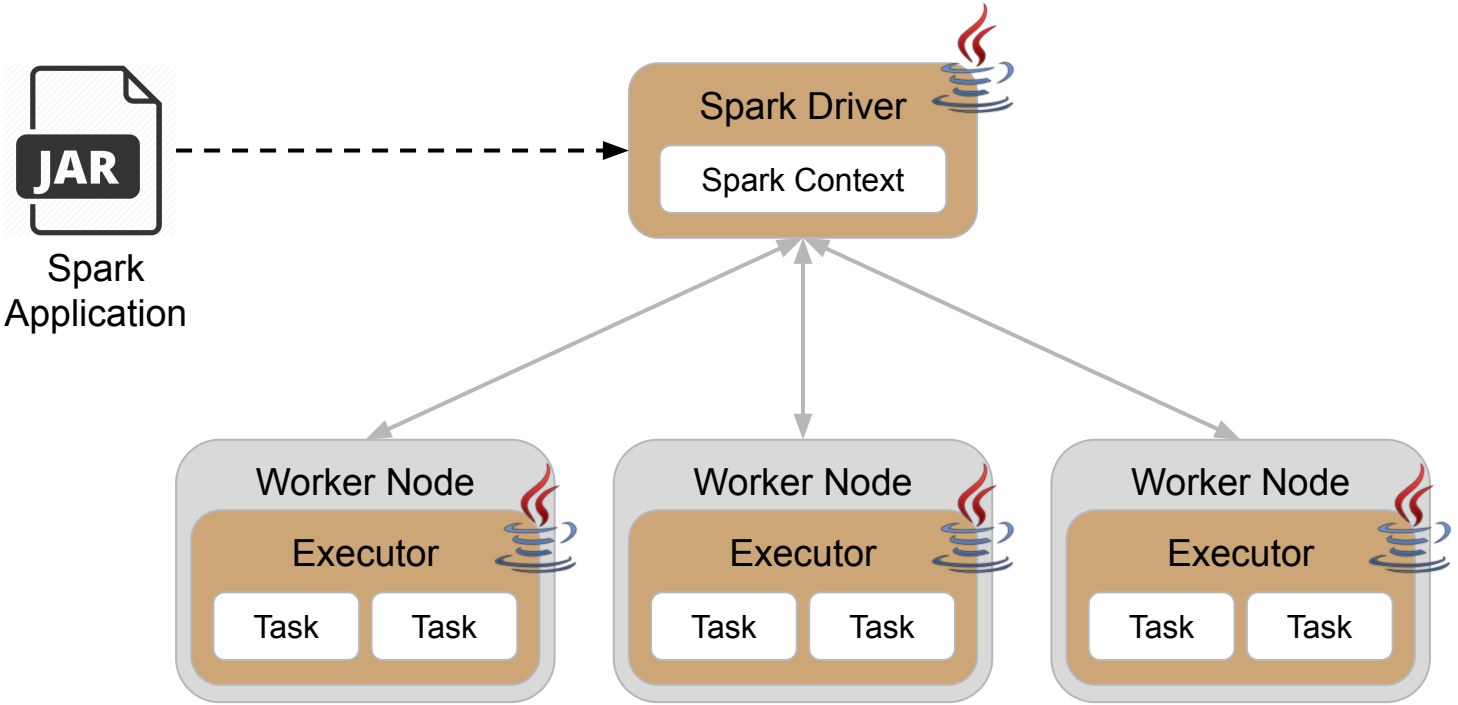
With some slides from Jesús Montes

Nov. 2024

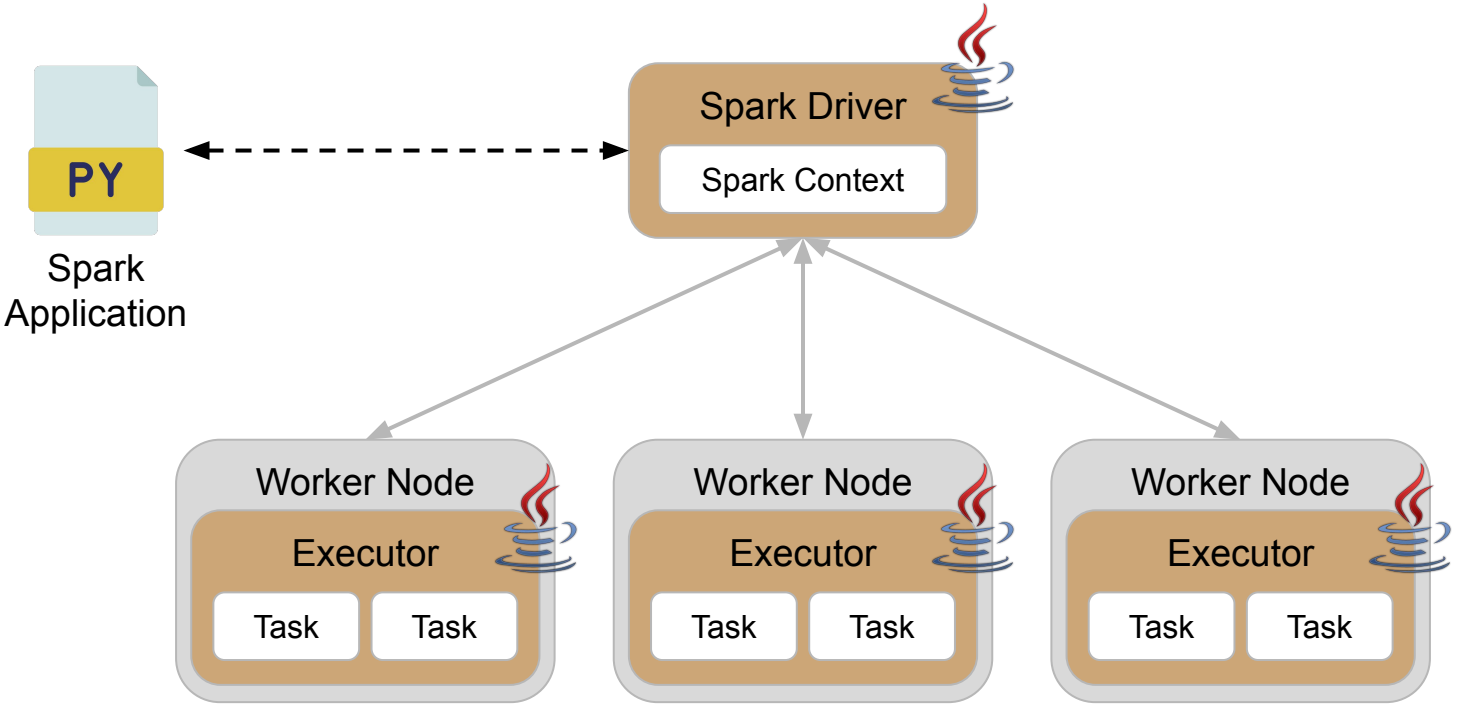
# Spark Applications

- So far, we have seen how to run Spark programs interactively, using the Spark Shell or using Jupyter Notebooks
- Typically, the shell/notebook is used for experimentation and initial data exploration
- Complex procedures are usually developed as independent applications.
  - Batch processing
  - Stream processing
  - ML
  - more...
- Spark applications run inside the JVM, and are distributed as JAR/Py files
- When a Spark application is launched, its JAR/Py is deployed in the cluster

# Spark Application Deployment (Scala)



# Spark Application Deployment (Python)



# Spark Application Deployment

The actual way the application is deployed depends on the environment

- In local mode, no file transference is required
- In cluster mode (mesos, yarn, kubernetes) the JAR/Py file and its dependencies have to be copied to the driver and worker nodes
  - In a Hadoop cluster, HDFS is used as a data repository where all these files are stored.
- Application deployment is usually performed with the help of the `spark-submit` script

# spark-submit

- `spark-submit` is a deployment tool included in the Spark distribution.
- With it, you can launch a Spark application JAR file, indicating the main class, configuration parameters, additional dependencies, etc
- It is a script that wraps the SparkSubmit class

```
# spark-submit
Usage: spark-submit [options] <app jar | python file> [app arguments]
Usage: spark-submit --kill [submission ID] --master [spark://...]
Usage: spark-submit --status [submission ID] --master [spark://...]
Usage: spark-submit run-example [options] example-class [example args]

Options:
  --master MASTER_URL         spark://host:port, mesos://host:port, yarn, or local.
  --deploy-mode DEPLOY_MODE   Whether to launch the driver program locally ("client")
or
                              on one of the worker machines inside the cluster
("cluster")
                              (Default: client).
  --class CLASS_NAME          Your application's main class (for Java / Scala apps).
  --name NAME                 A name of your application.
  --jars JARS                 Comma-separated list of local jars to include on the
driver
                              and executor classpaths.
  --packages                  Comma-separated list of maven coordinates of jars to
include
                              on the driver and executor classpaths. Will search the
local
                              maven repo, then maven central and any additional remote
                              repositories given by --repositories. The format for the
                              coordinates should be groupId:artifactId:version.
```

# Programming Spark Applications

- Spark Applications are like regular Scala/Python/... programs
- They have a main method from a main object/class
  - Part of an object in Scala
  - Static method in Java
  - Main function in Python
- The main difference with using the shell is that the Spark Context must be explicitly configured and created
- For doing this, we use the Spark Core API

```
# Create Spark Context with SparkConf

from pyspark import SparkConf, SparkContext

def main():
      conf = SparkConf()
      conf.setAppName("myApp")
      sc = SparkContext.getOrCreate(conf)
      sc.setLogLevel("ERROR")

      #     ... Do the work ...     #
                  ###

      #  Organize this code has you consider  #

                  ###

      # Stop the SparkContext when done
      sc.stop()

if __name__ == "__main__":
      main()
```

# Spark Configuration

There are 5 levels of configuration in spark, from the highest to the lowest priority we have:

1. Programmatically, using SparkConf()
2. Command-Line, using spark-submit flags
3. Configuration files, there may be many of them. Even one per node
4. Cluster Manager configuration
5. Spark Built-in default configuration files

# SparkConf

- Objects of the SparkConf class serve as configuration for a Spark application
- They are used to set various Spark parameters as key-value pairs
- Most of the time, a SparkConf object can be created simply with `new SparkConf()` which will load values from any spark.* Java system properties set in the application as well
- Parameters set directly on the SparkConf object take priority over system properties

# SparkConf methods and configuration variables

Methods:

- `setAppName(name: String): SparkConf`
  Set a name for your application
- `set(key: String, value: String): SparkConf`
  Set a configuration variable
- `setJars(jars: Array[String]): SparkConf`
  Set JAR files to distribute to the cluster
- `setMaster(master: String): SparkConf`
  The master URL to connect to, such as "local" to run locally with one thread, or "spark://master:7077" to run on a Spark standalone cluster

Variables:

- spark.app.name
- spark.driver.cores
- spark.driver.memory
- spark.driver.extraClassPath
- spark.driver.extraJavaOptions
- spark.executor.memory
- spark.executor.extraClassPath
- spark.executor.extraJavaOptions
- spark.executor.cores
- ...

Take a look at http://spark.apache.org/docs/latest/configuration.html

# Submitting your first Spark Application

1. Download the file: **app.py** from moodle and paste it **on the folder with pagecounts**.
2. Let's check the code:

```
conf = SparkConf()
sc = SparkContext.getOrCreate(conf)
sc.setLogLevel("ERROR")

inputFilePath = "/home/user/pagecounts"
rdd = sc.textFile(inputFilePath)

enPages = rdd.filter(lambda line: line.startswith('en '))
numEnLines = enPages.count()
print('Number of EN pages:', numEnLines)

enPagesTuples = enPages.flatMap(lambda line: [(pieces[0], pieces[1], int(pieces[2]), int(pieces[3]))
                                              for pieces in [line.split(" ")] if len(pieces) == 4])
for line in enPagesTuples.take(10):
        print(line)
topSortedEnPages = enPagesTuples.sortBy(lambda x: x[2], ascending=False) \
         .take(10) \
for page in topSortedEnPages:
         print(page)
sc.stop()
```

3. Submit your application: `[user]$` spark-submit --master local[*] app.py

# Web UI

localhost:4040

# Shared variables in Spark

- When a function passed to a Spark operation is executed on a remote cluster node, it works on separate copies of all the variables used in the function
- These variables are copied to each machine
- No updates to the variables on the remote machine are propagated back to the driver program
- Supporting general, read-write shared variables across tasks would be inefficient
- However, Spark does provide two limited types of shared variables for two common usage patterns: *broadcast variables* and *accumulators*

# Broadcast Variables

- Read-only variables cached on each machine
- They can be used, for example, to give every node a copy of a large input dataset in an efficient manner
- Spark attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost
- Explicitly creating broadcast variables is only useful when:
  - Tasks across multiple stages need the same data
  - Caching the data in deserialized form is important

Creating a broadcast variable:

```
bcVar = sc.broadcast("hello!")
```

Accessing a broadcast variable:

```
bcVar.value
```

After the broadcast variable is created:

- It should be used instead of the value v
- v should not be modified, to ensure that all nodes get the same value

# Accumulators

- Accumulators are variables that are only “added” to through an associative and commutative operation
- They can be used to implement counters (as in MapReduce) or sums
- Spark natively supports accumulators of numeric types, and programmers can add support for new types
- If accumulators are created with a name, they will be displayed in Spark’s UI. This can be useful for understanding the progress of running stages

Using an accumulator:

```
accum=sc.accumulator(0)
RDD=sc.parallelize([1,2,3,4,5])
RDD.foreach(lambda
x:accum.add(x))
```

Acces accumulator value:

```
accum.value
```

What will be the value of `accum`?

# Repartition

```
repartition(numPartitions: Int): RDD[T]
```

- An RDD transformation that returns a new RDD that has exactly numPartitions partitions
- This operation reshuffles the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network
- It can be used to optimize cluster utilization

# Cache/Persist and Checkpoint

```
cache()
persist()
persist(newLevel: StorageLevel)
```

- RDD method that marks the RDD to be persisted
- The first time the RDD is computed in an action, it will be kept in memory on the nodes
- Each persisted RDD can be stored using a different storage level (MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY, MEMORY_ONLY_SER,...)

```
checkpoint()
```

- RDD method that marks the RDD to be saved to a file inside the checkpoint directory set with SparkContext method `setCheckpointDir`
- All references to its parent RDDs will be removed (lineage is lost)
- It is strongly recommended that the RDD is persisted in memory, otherwise saving it on a file will require recomputation