1. INTRODUCTION TO APACHE SPARK

Apache Spark - December 2021

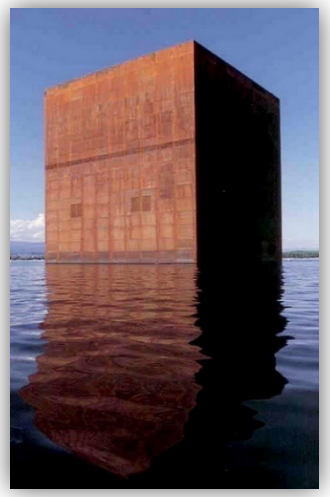
EDEM
Escuela de Empresarios

1. Before Apache Spark
2. Introducing Apache Spark
3. Apache Spark. Architecture
4. RDD
5. Key/Value Pairs RDD
6. Executing Spark
7. Architectures

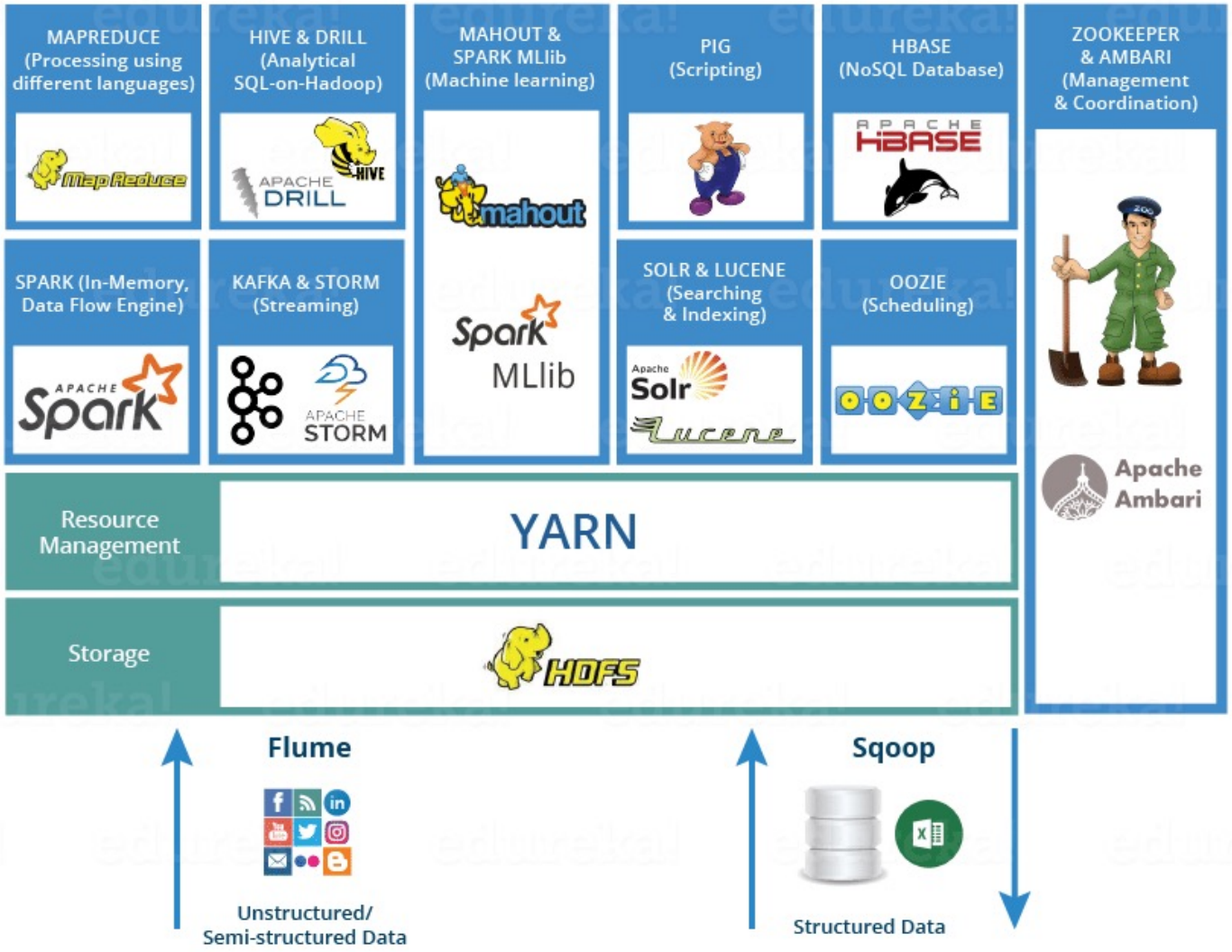
1. Before Apache Spark 
2. Introducing Apache Spark
3. Apache Spark. Architecture
4. RDD
5. Key/Value Pairs RDD
6. Executing Spark
7. Architectures

Hadoop

- **Monolithic Computing**
  - For decades, the goal was a bigger, more powerful machine
  - This approach has limitations
    - High Cost
    - Limited scalability



Hadoop



CLOUDERA

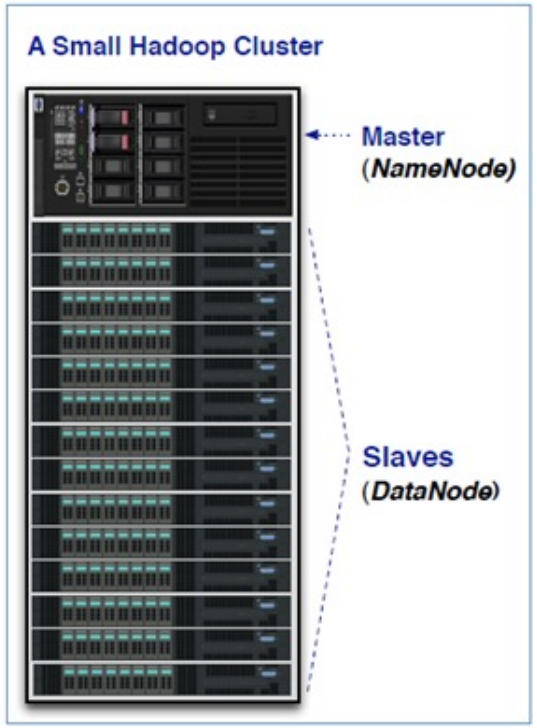
MAPR

HDFS

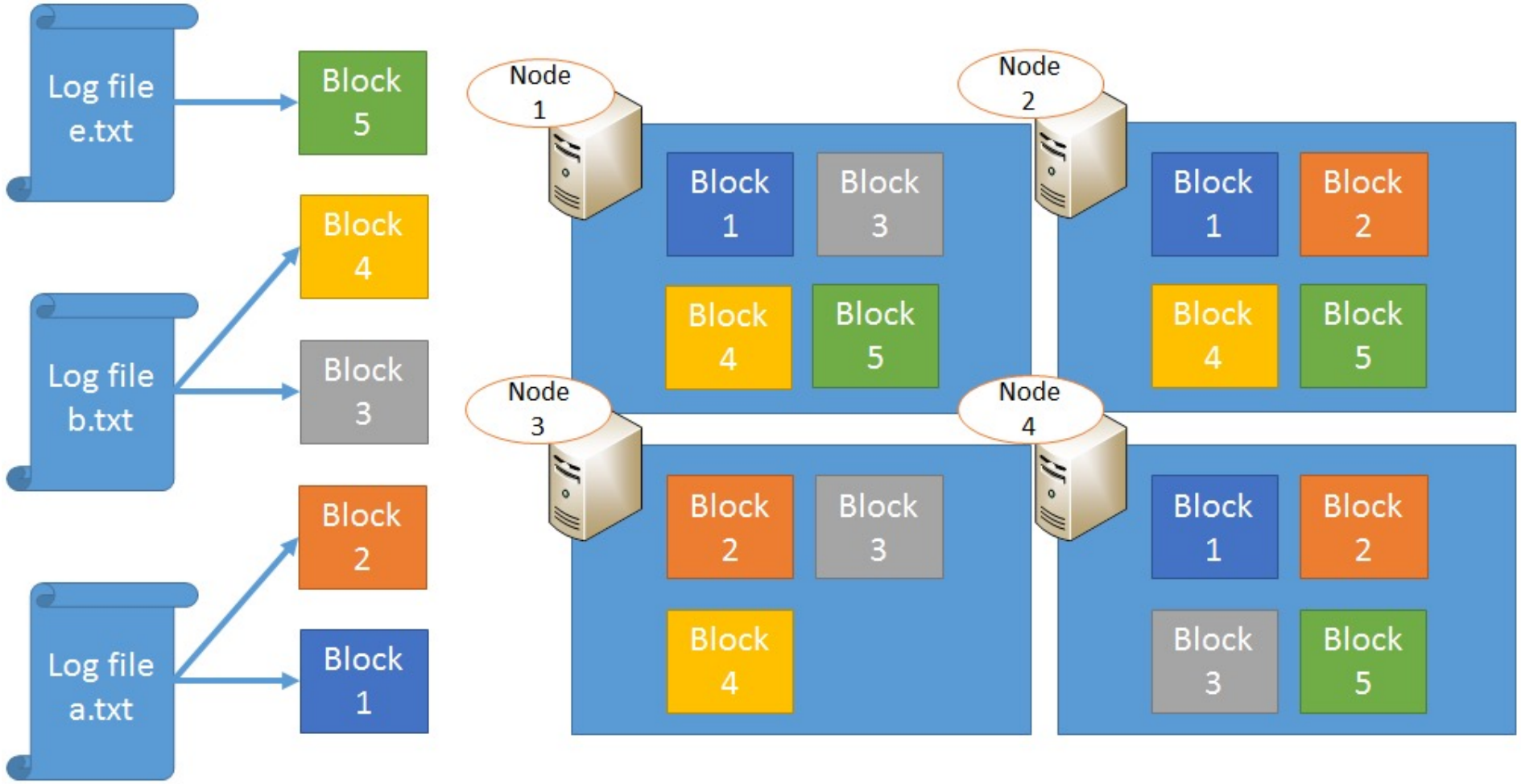
- **HDFS**, the Hadoop Distributed File System, is responsible for storing data on the cluster
- Data is split into blocks and distributed across multiple nodes in the cluster
- **Each block is replicated multiple times:**
  - Replicas are stored on different nodes

HDFS

- Blocks are replicated across multiple machines, known as **DataNodes**
- A master node called **NameNode** keeps track of which blocks make up a file, and where those blocks are located

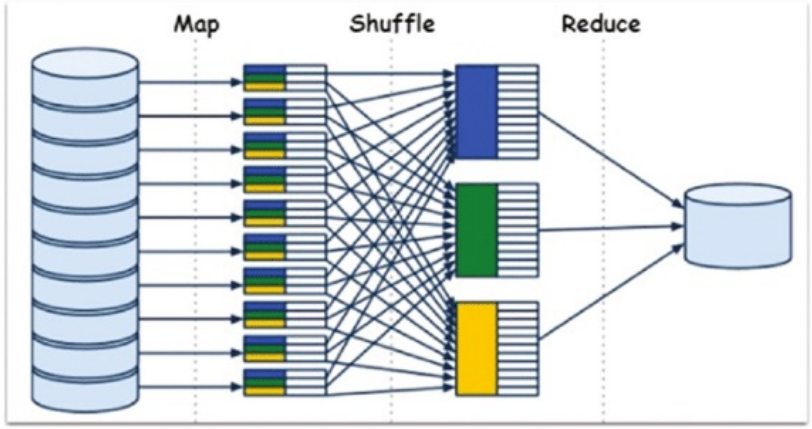


HDFS



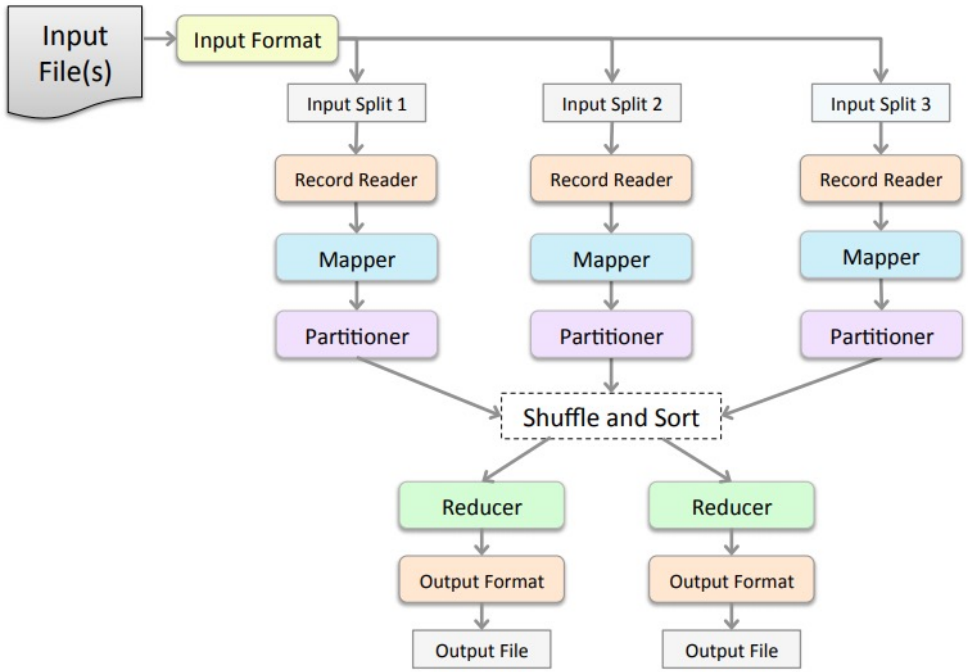
MapReduce

- **MapReduce**
  - Is batch oriented
    - So Hive, Pig and all MR-based systems too
  - Great throughput but high latency
    - Not for BI tools
    - Not for real time (Stream) processing
    - Just ingesting streaming data with Flume
  - Forces to concatenate multiple jobs
    - External orchestration
  - Each job flushes to disk
  - Data sharing requires external storage



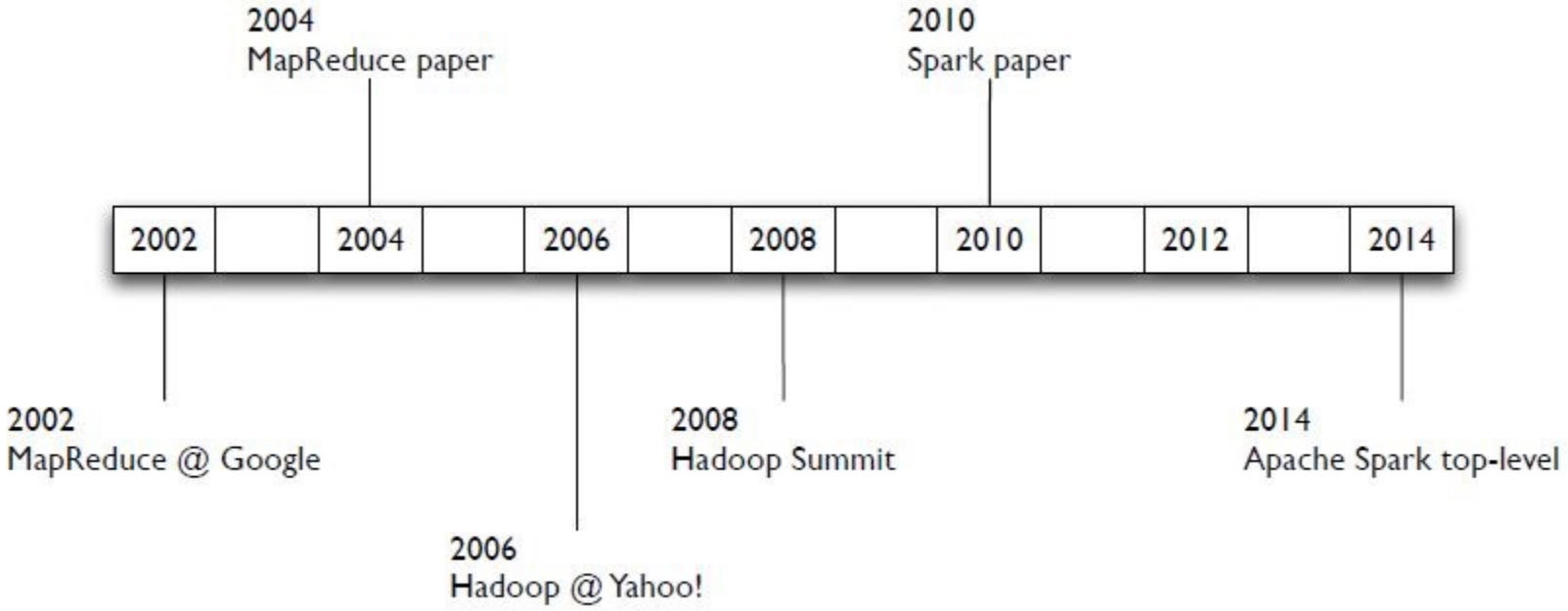
MapReduce

- **Key concepts to keep in mind with MapReduce**
  - The **Mapper** works on an individual record at a time
  - The **Reducer** aggregates results from the Mappers
  - The **intermediate keys** produced by the Mapper are the keys on which the aggregation will be based



1. Before Apache Spark
2. Introducing Apache Spark 
3. Apache Spark. Architecture
4. RDD
5. Key/Value Pairs RDD
6. Executing Spark
7. Architectures

History


2004
MapReduce paper
2010
Spark paper
2002
2004
2006
2008
2010
2012
2014
2002
MapReduce @ Google
2006
Hadoop @ Yahoo!
2008
Hadoop Summit
2014
Apache Spark top-level

What Is Apache Spark?

- Apache Spark is an open-source cluster computing framework

- Requires
  - **Cluster manager**
    - Standalone – a simple cluster manager included with Spark
    - Apache Mesos – a general cluster manager
    - Hadoop YARN – the resource manager in Hadoop 2
    - Kubernetes
  - **Distributed storage system**
    - Hadoop Distributed File System (HDFS)
    - Cassandra
    - Amazon S3

- Also supports pseudo-distributed mode for development and testing
  - Local file system and one worker per CPU core

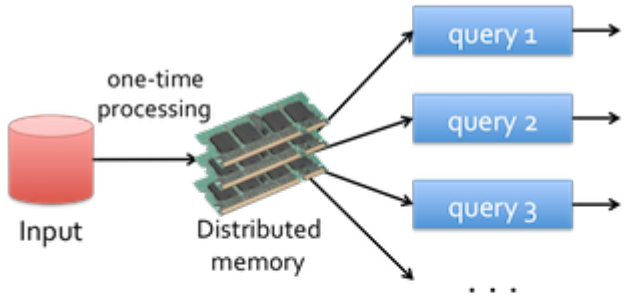
What Is Apache Spark?

- Spark provides the following major benefits
  - **Lightning speed of computation**
    - Data are loaded in distributed memory (RAM) over a cluster of machines
  - **Highly accessible**
    - Through standard APIs built in Java, Scala, Python R or SQL
  - **Compatibility**
    - With the existing Hadoop ecosystems
  - **Convenient**
    - Interactive shells in Scala and Python (REPL)
  - **Enhanced productivity**
    - Due to high level constructs that keep the focus on content of computation

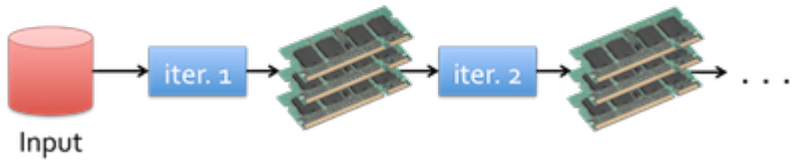
What Is Apache Spark?

- Apache Spark is a cluster computing platform designed to be **fast**, **highly-accessible** and **general-purpose**

- Speed
  - Spark extends the MapReduce model to:
    - Efficiently support more types of computations
      - *e.g:* interactive queries, stream processing
    - Ability to run computations in memory
      - Also faster than MR for complex applications running on disk



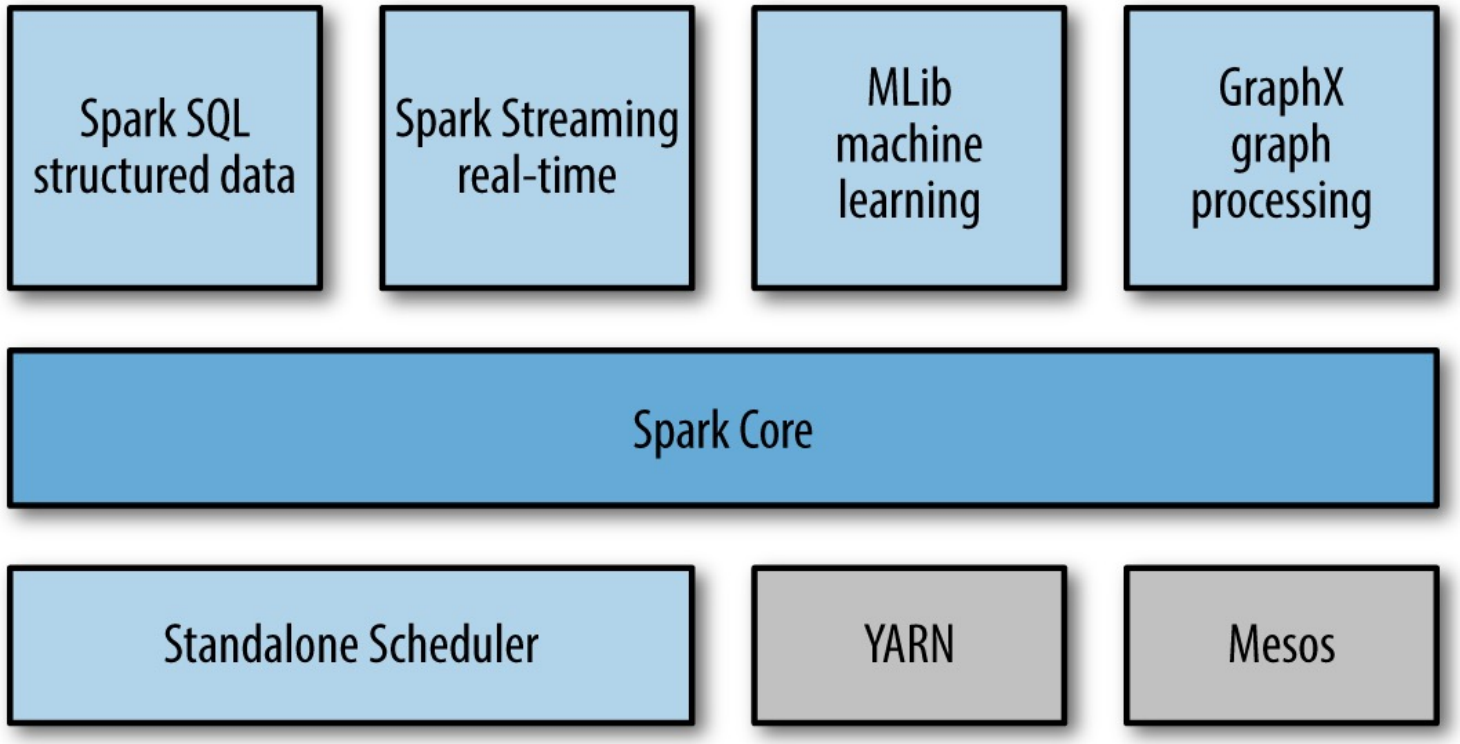
(a) Low-latency computations (queries)



(b) Iterative computations

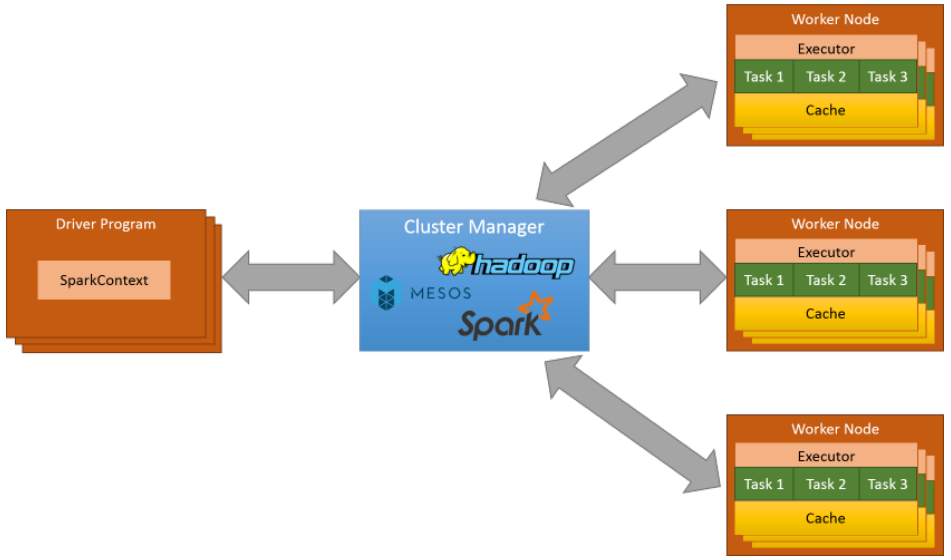
1. Before Apache Spark
2. Introducing Apache Spark
3. Apache Spark. Architecture 
4. RDD
5. Key/Value Pairs RDD
6. Executing Spark
7. Architectures

Architecture

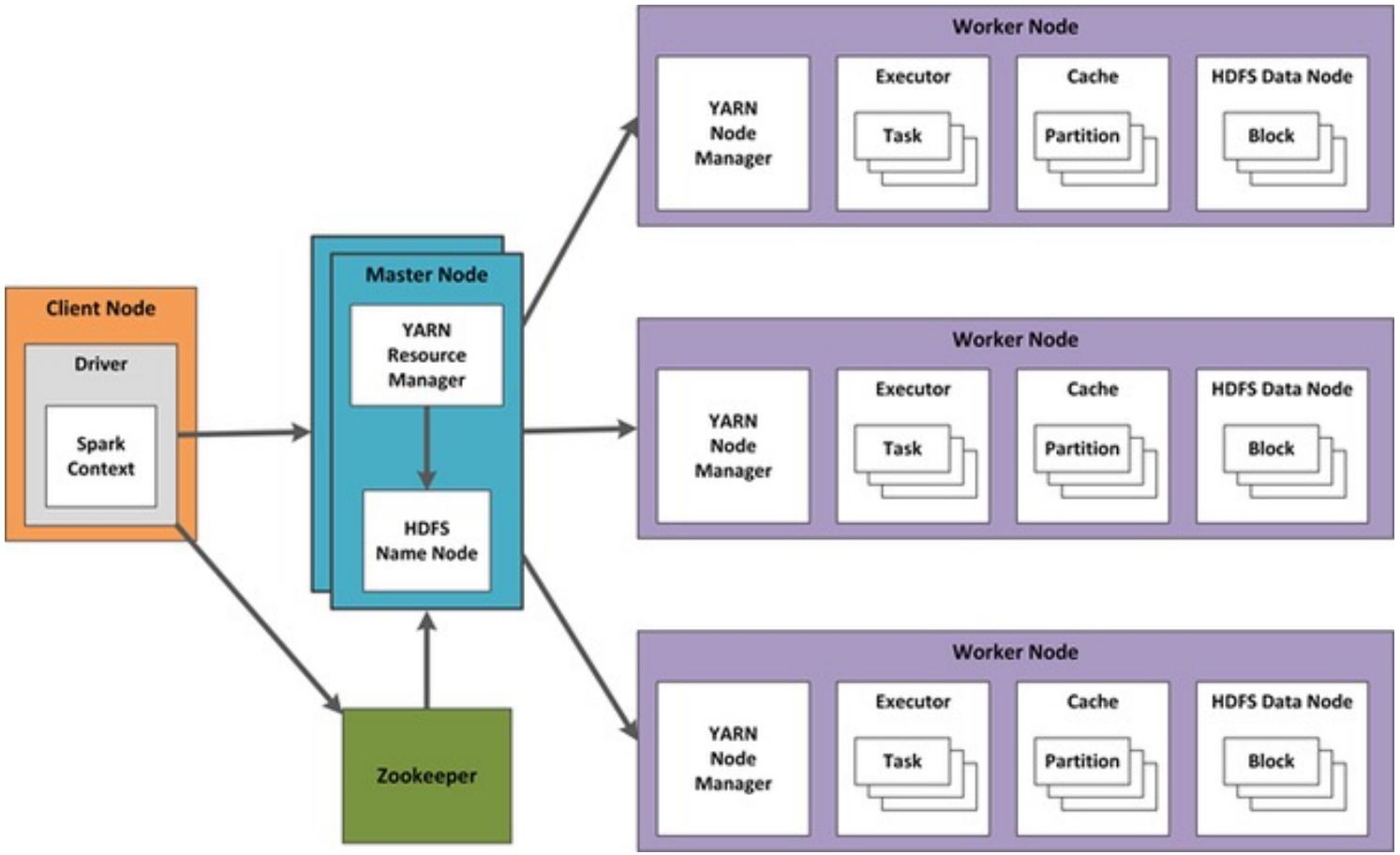


Spark Runtime Architecture

- Master/slave architecture
- Central coodinator **driver** (own java process)
- Daemons on workers called **executor** (own java process)
- **Driver + executors = Spark application**
- Application is launched using a **cluster manager**



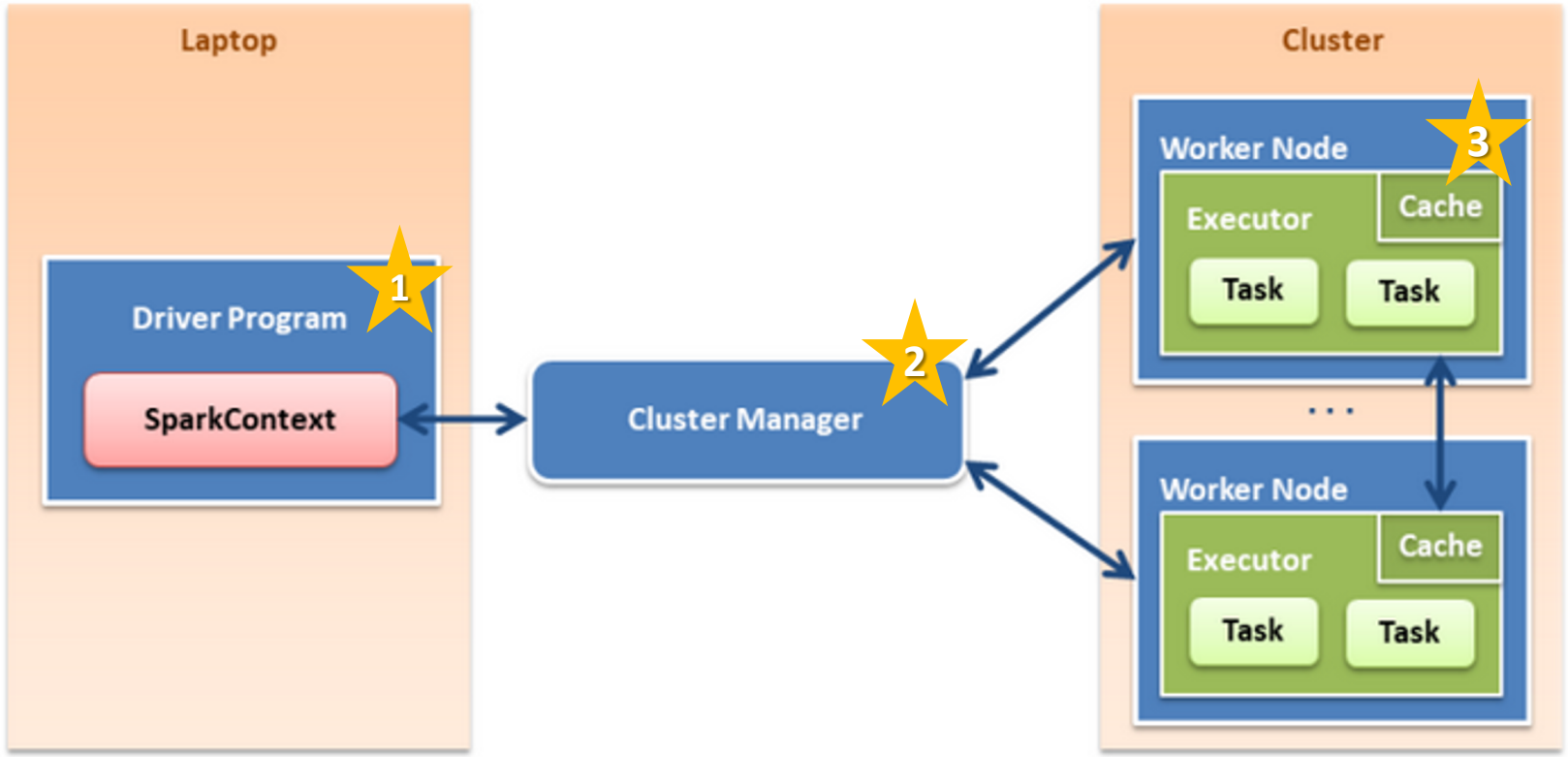
Spark Runtime Architecture



Core Spark Concepts

- Every Spark application consists of a ***driver program*** that defines distributed datasets on a cluster and then launches various parallel operations to them
- Driver can be your own program or the **Spark shell** to type operations you want to run
- Driver access Spark through a **`SparkSession` object** which represents a connection to a cluster

Apache Spark – Dataflow



Apache Spark – Dataflow

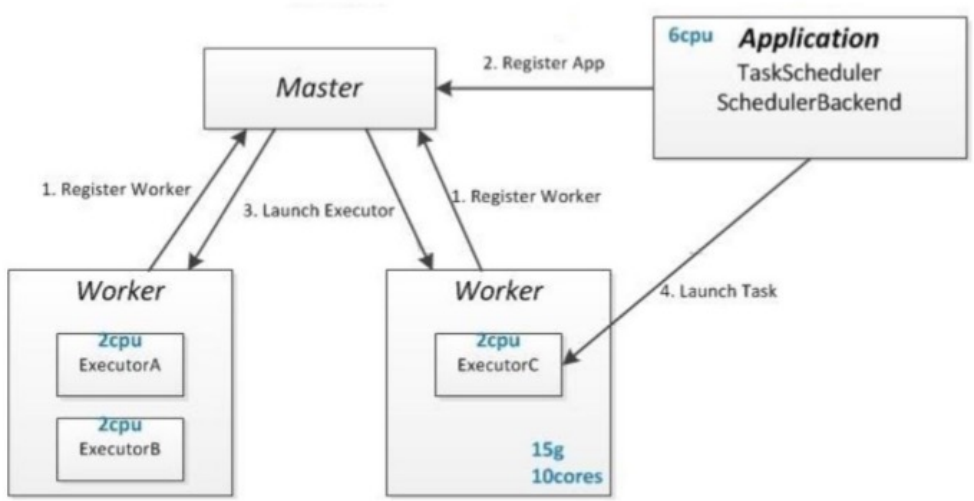
- Spark **code is developed in the Driver program**, connecting cluster through *SparkSession*

- We will define all resources and configuration in *SparkSession*, where RDDs and other structures will be created as well

```
import findspark
findspark.init('/opt/mapr/spark/spark-2.0.1/')
import pyspark
from pyspark.sql import SparkSession

spark=SparkSession.builder.appName("variable_selection")\
                          .config("spark.master","yarn")\
                          .config("spark.eventLog.enabled","true")\
                          .config("spark.executor.instances","5")\
                          .config("spark.executor.cores","3")\
                          .config("spark.executor.memory","5g")\
                          .getOrCreate()



df=spark.read.csv('hdfs:/data/databases/cars.csv')

df.show()
```

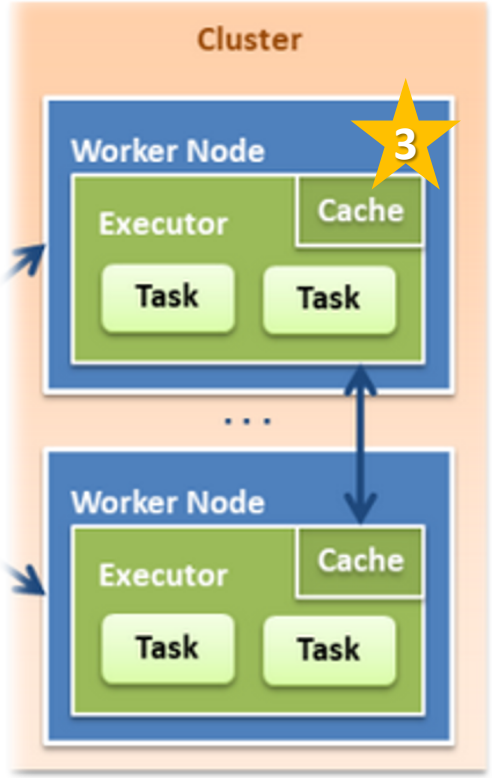
Apache Spark – Dataflow

- Spark works with the following services as cluster manager:
  - **YARN**
  - **Mesos**
  - **Kubernetes** (experimental)
  - **Standalone.** A simple cluster manager included with Spark
    - Requires prior installation of Apache Spark in all nodes



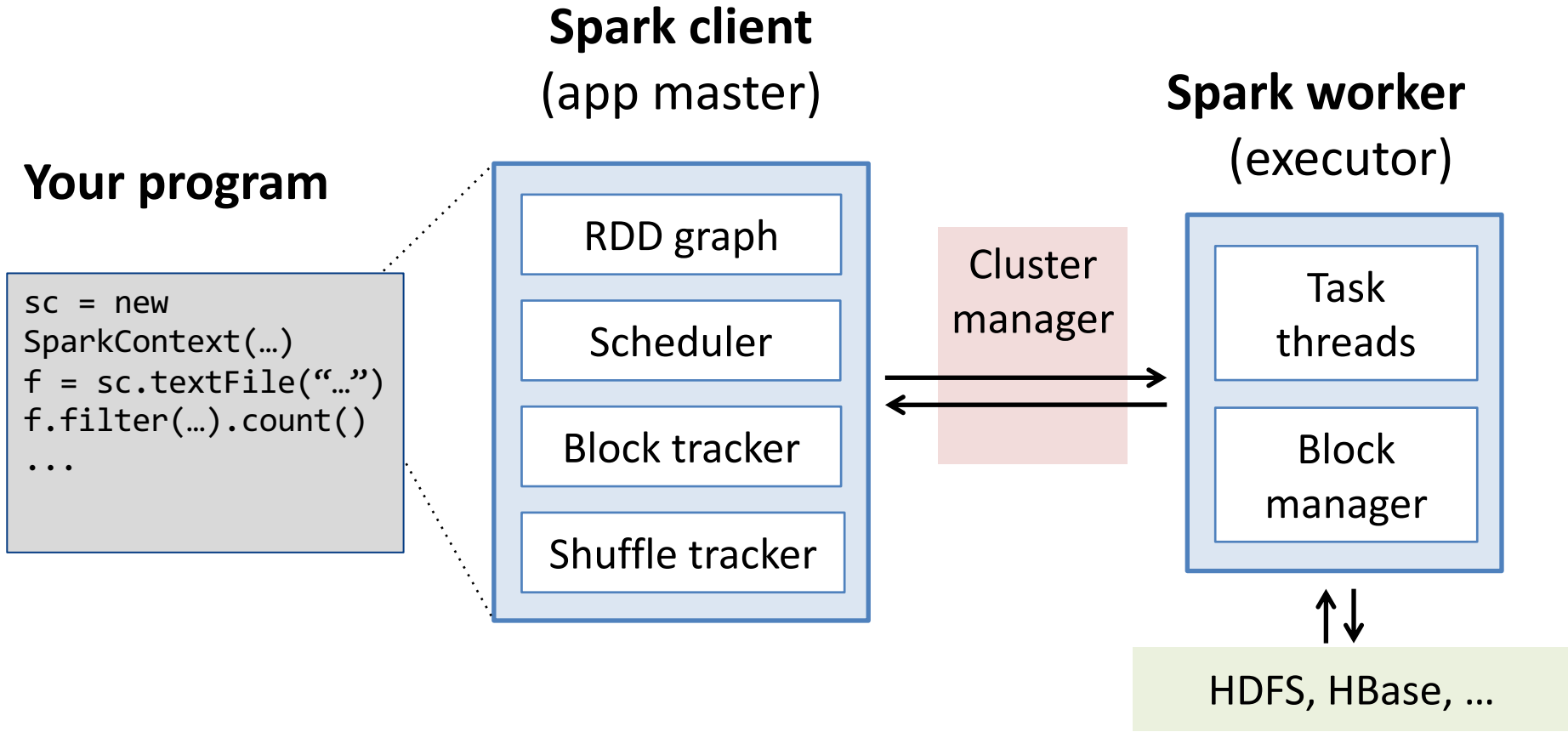
Apache Spark – Dataflow

- There is a background process called **Executor** for each Worker Node
- Executor launches different task for each transformation



Apache Spark – Dataflow

- Zoom-in the architecture



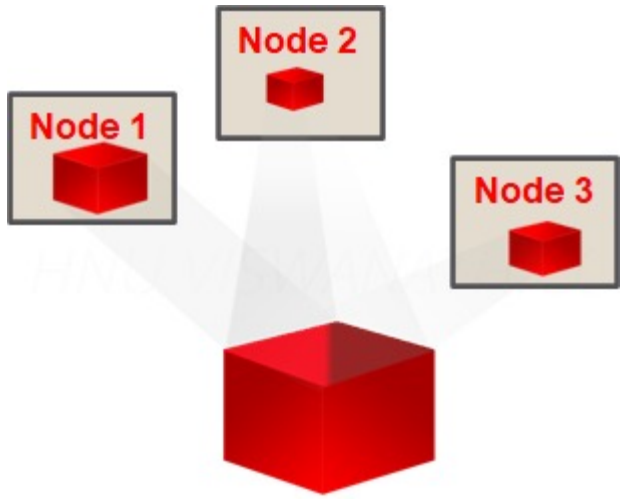
1. Before Apache Spark
2. Introducing Apache Spark
3. Apache Spark. Architecture
4. RDD 
5. Key/Value Pairs RDD
6. Executing Spark
7. Architectures

Resilient Distributed Dataset (RDD)

- It is an **immutable** **distributed** collection of data, which is **partitioned** across machines in a cluster
- It facilitates two types of operations:
  - **Transformation**
    - An operation such as filter(), map(), or union() on an RDD that yields another RDD
    - **Lazily evaluated**, in that they don't run until an action warrants it
  - **Action**
    - An action is an operation such as count(), first(), take(n), or collect() that triggers a computation, returns a value back to the Master, or writes to a stable storage system
- The Driver remembers the transformations applied to an RDD, so if a partition is lost, that partition can easily be reconstructed on other machine in the cluster
  - That is why is it called "**Resilient**"

RDD

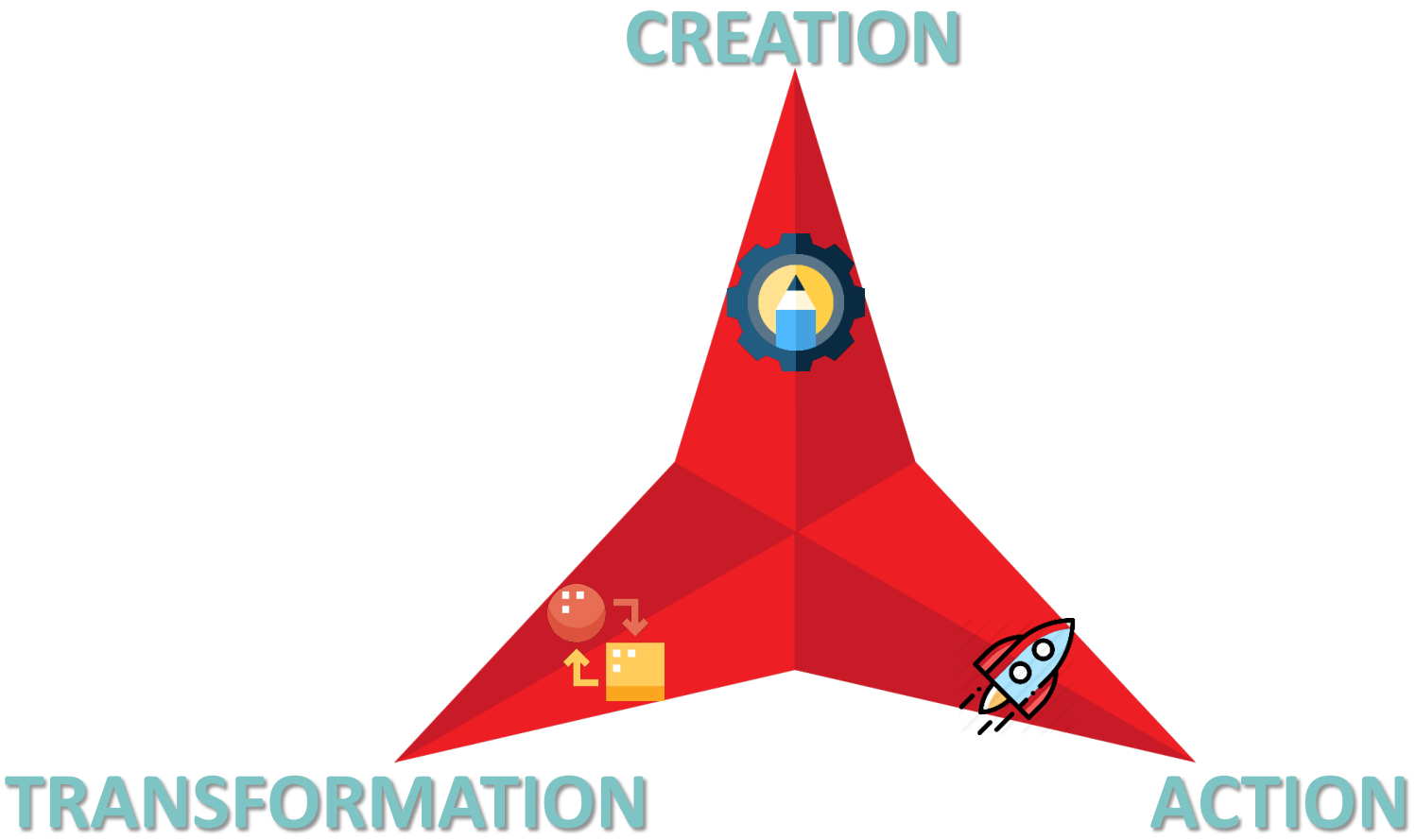
- RDD stands for **Resilient Distributed Datasets**
- ***Resilient*** because RDDs are immutable
  - *They can't be modified once created*
- ***Distributed*** because it is distributed across cluster
- ***Dataset*** because it holds data



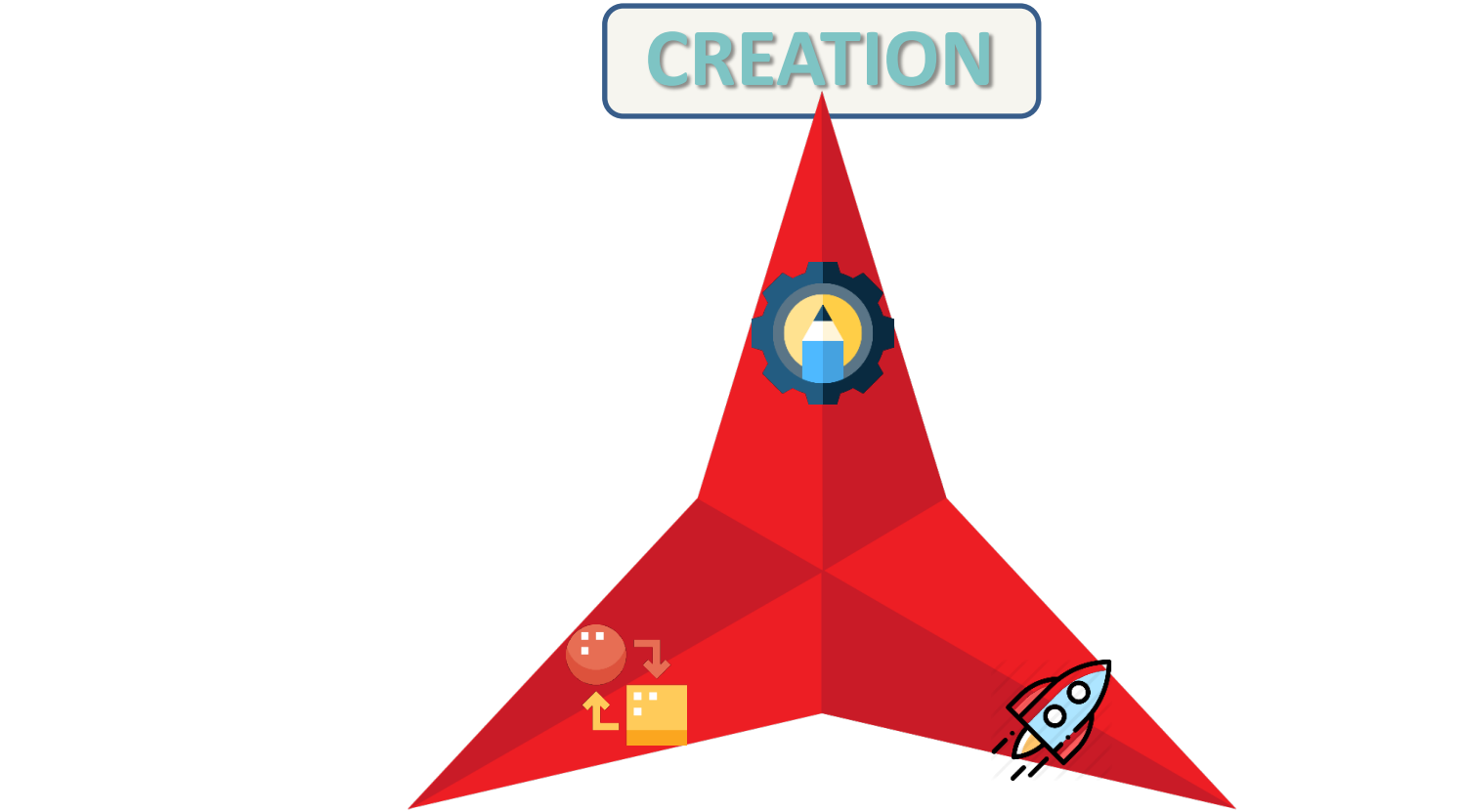
Hands-on

- Open “00.Introduction_to_Apache_Spark.ipynb” in Google Colab:
  - Execute examples 1 and 2
  - Try exercise 1 and 2

RDD – OPERATIONS



RDD – OPERATIONS



RDD – Creation

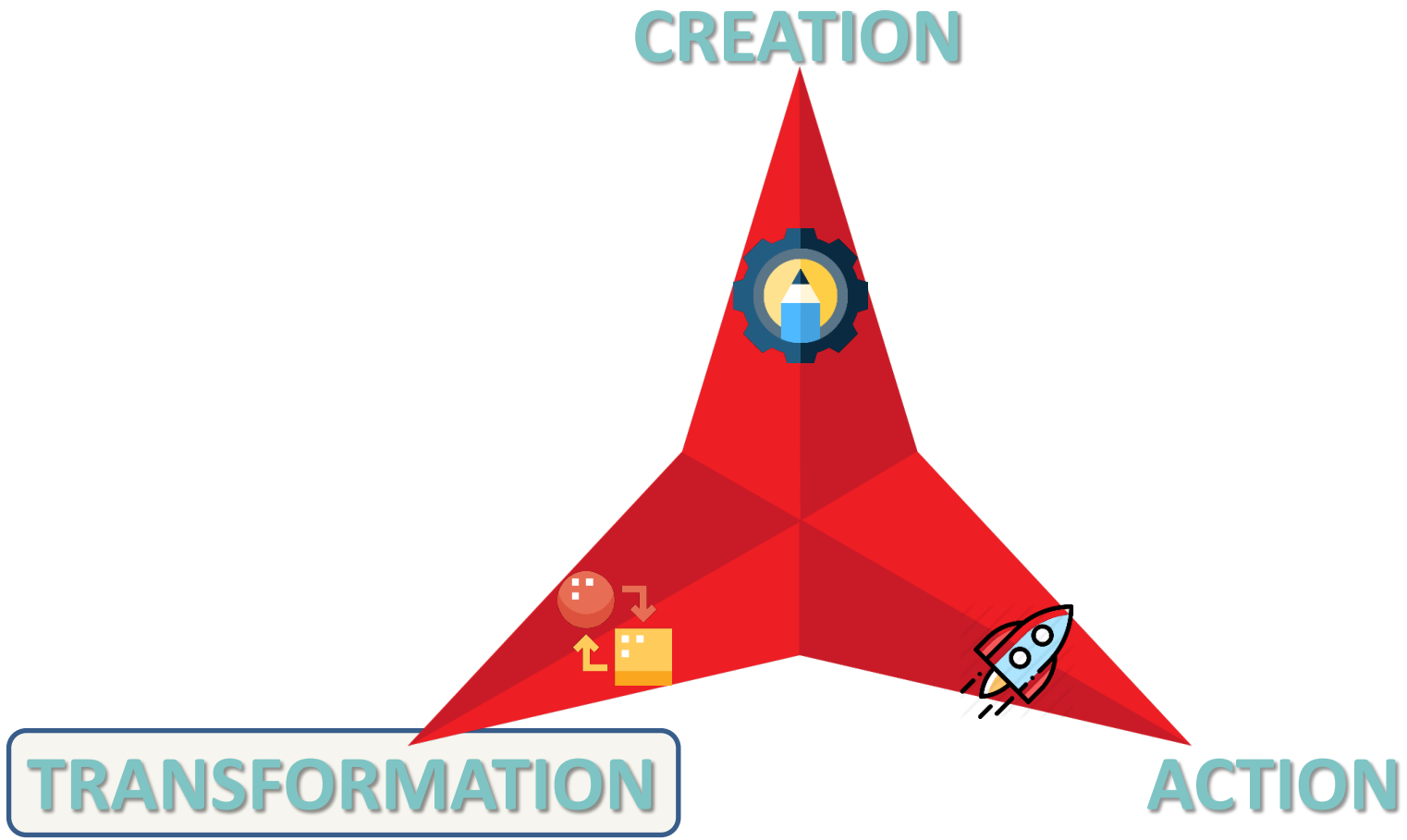
- Three ways to create a RDD:
  - From external source
    - File
    - Kafka, Mysql, ….

    ```
    spark.sparkContext().textFile("file.csv")
    ```
  - From an internal structure

    ```
    spark.sparkContext().parallelize(List(1, 2, 3, 4))
    ```
  - From other RDD

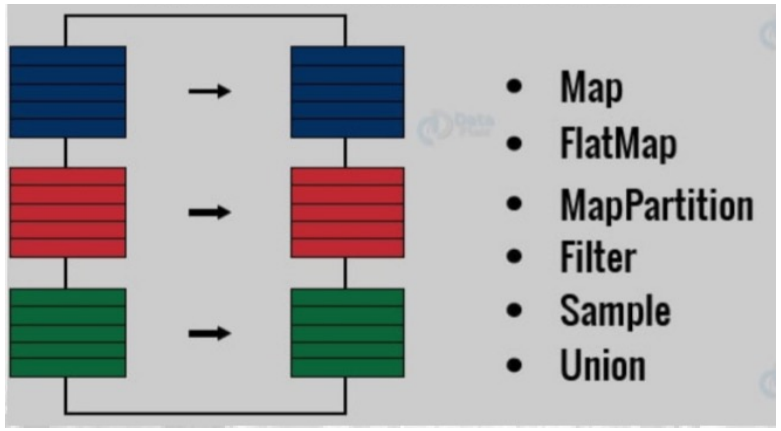
    ```
    otherRDD.map(word => (word, word.length)
    ```

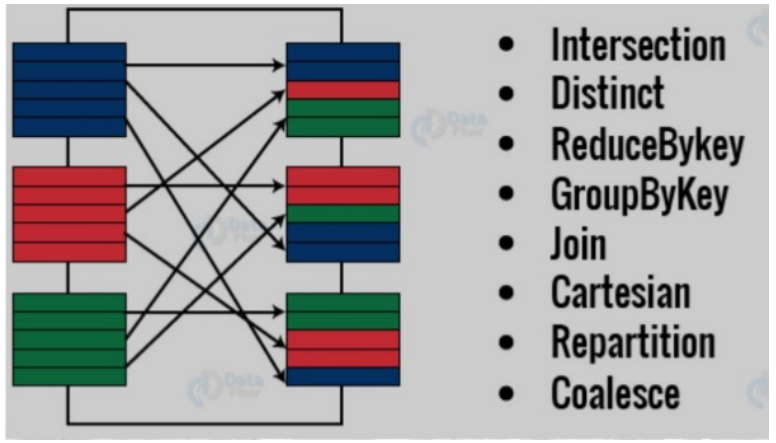
RDD – OPERATIONS



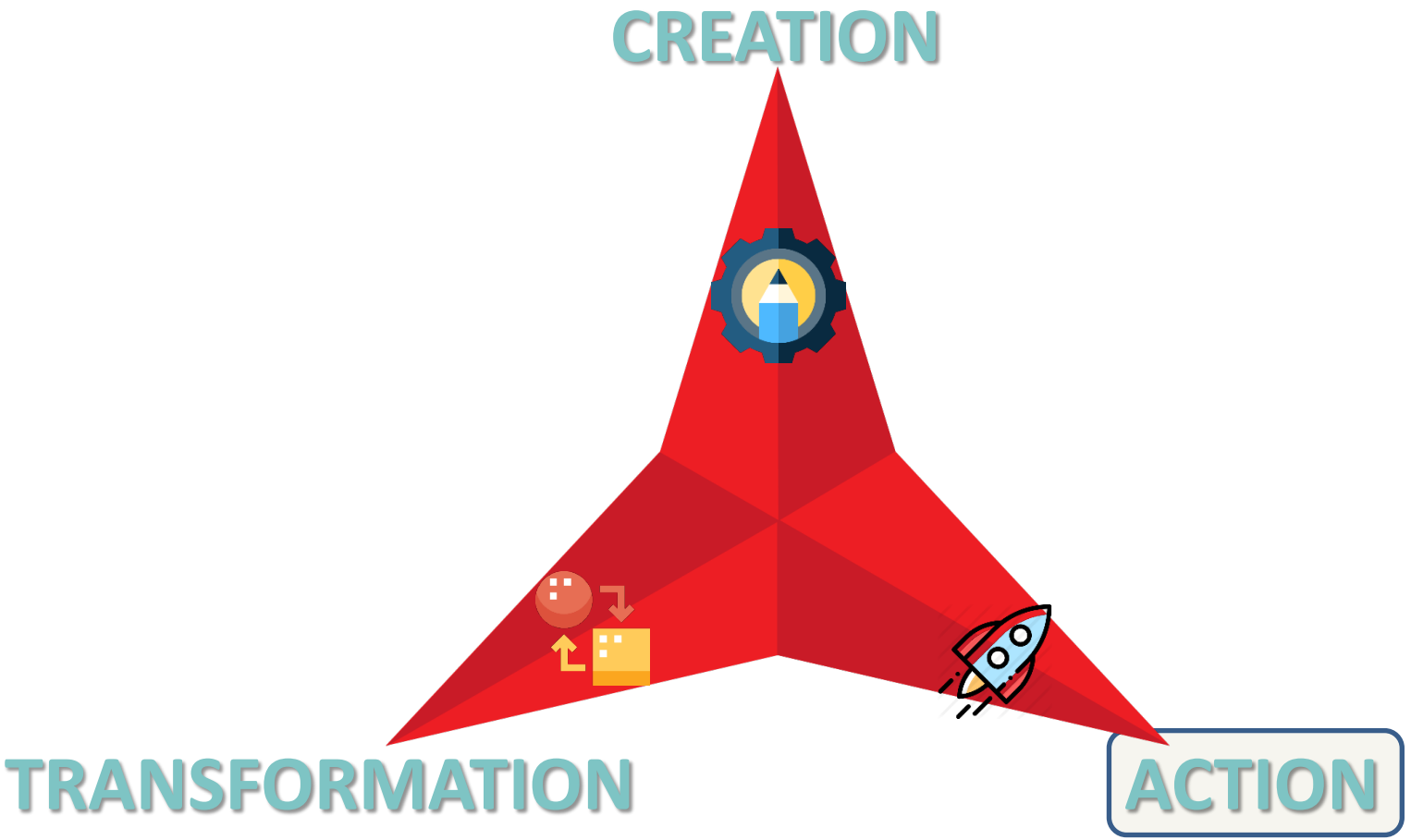
RDD – Transformation

- Operations over RDDs which return a new RDD
- Transformations are **lazy**(*)
  - Only computed when an action requires a result to be returned to the driver program
  - (*) Some transformations like sortByKey are not lazy



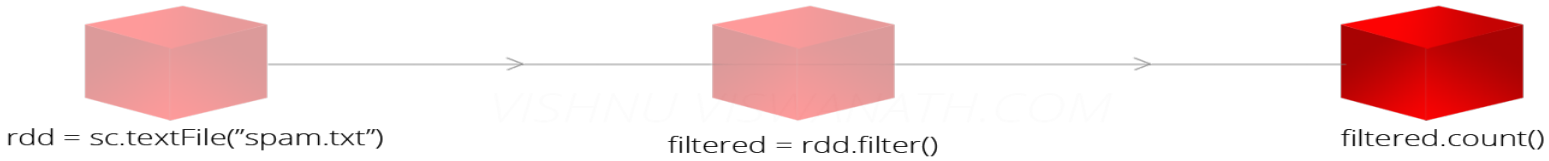


RDD – OPERATIONS

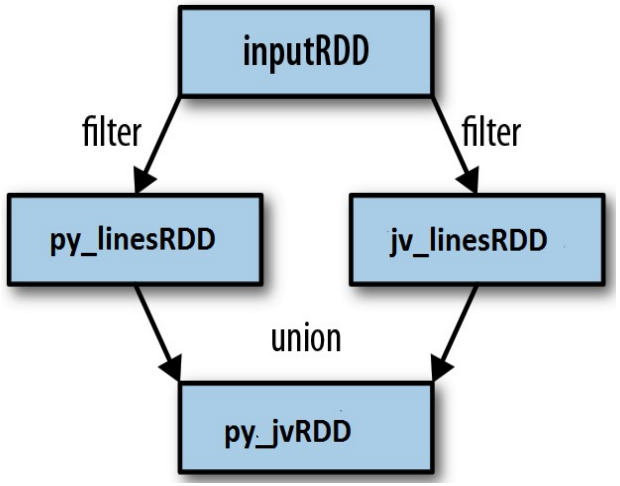


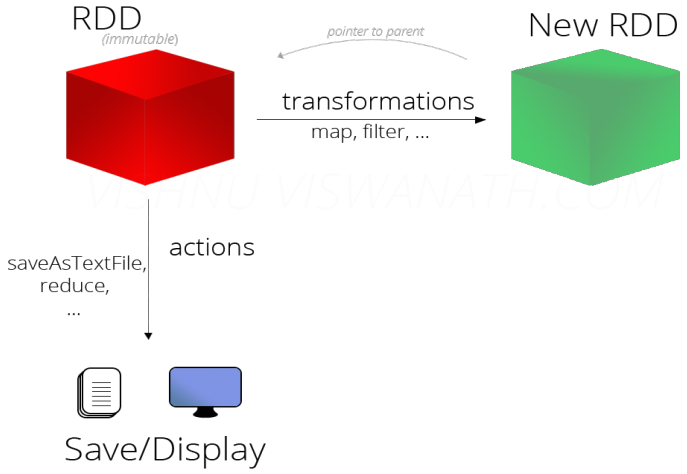
RDD – Actions

- Actions, which **return a value to the driver program** after running a computation on the dataset
  - Eg. Reduce, count, ….
- Action is used to either **save a result to some location or to display it**



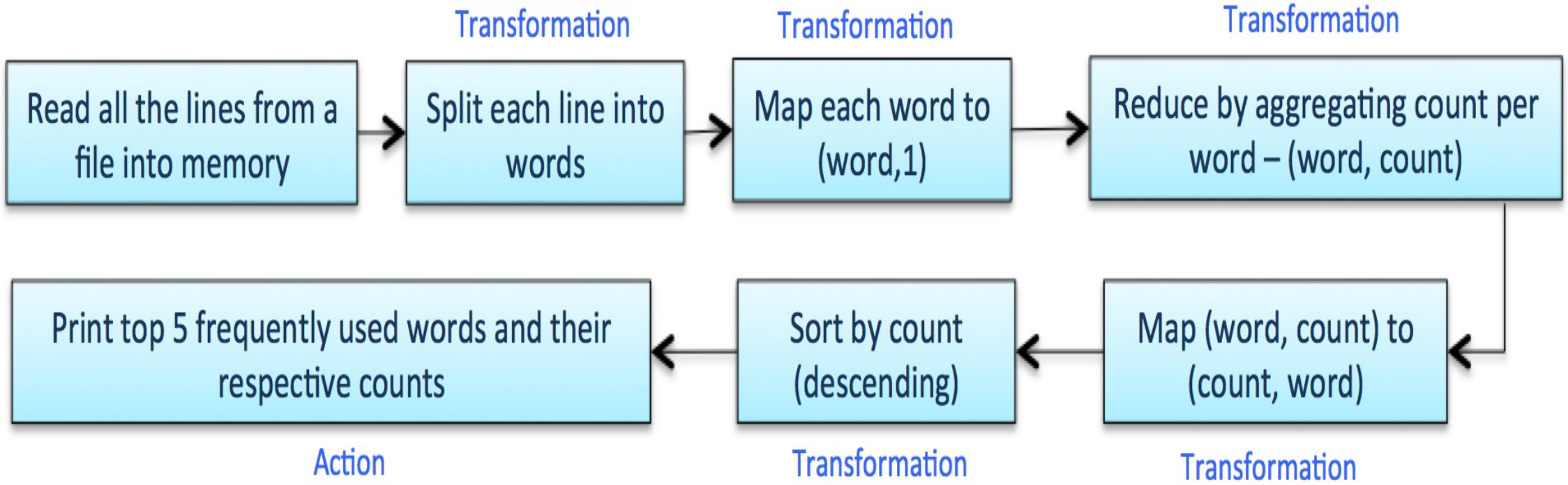
Resilient Distributed Dataset (RDD)





inputRDD = sc.textFile("README.md") **-> Transformation**
py_linesRDD = inputRDD.filter(**lambda** line: "Python" **in** line) **-> Transformation**
jv_linesRDD = inputRDD.filter(**lambda** line: "Java" **in** line) **-> Transformation**
py_jvRDD **=** py_linesRDD.union(jv_linesRDD) **-> Transformation**
print py_jvRDD.count() -> Action

Resilient Distributed Dataset (RDD)



Resilient Distributed Dataset (RDD)

Transformation	Meaning
➢ **map**(func)	Return a new RDD formed by passing each element of the source through a function *func*
➢ **filter**(func)	Return a new RDD 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
➢ **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 of the data, with or without replacement, using a given random number generator seed
➢ **union**(otherRDD)	Return a new RDD that contains the union of the elements in the calling RDD and the argument
➢ **intersection**(otherRDD)	Return a new RDD that contains the intersection of the elements in the calling RDD and the argument

Resilient Distributed Dataset (RDD)

Transformation	Meaning
➢ **distinct**()	Return a new RDD that contains the distinct elements of the source dataset
➢ **groupByKey**()	When called on a dataset of *(K, V)* pairs, returns a dataset of *(K, Iterable<V>)* pairs
➢ **reduceByKey**(func)	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 *func*, which must be of type *(V,V) => V*
aggregateByKey(zeroValue)(seqOp, combOp)	When called on an RDD of *(K, V)* pairs, returns an RDD of *(K, U)* pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value; **allows an aggregated value type that is different than the input value type**, while avoiding unnecessary allocations
➢ **sortByKey**([ascending])	When called on an RDD of *(K, V)* pairs where *K* implements Ordered, returns an RDD of *(K, V)* pairs **sorted by keys** in ascending or descending order
➢ **join**(otherRDD)	When called on RDDs of type *(K, V)* and *(K, W)*, returns an RDD of *(K, (V, W))* pairs with all pairs of elements for each key
➢ **cogroup**(otherRDD)	When called on RDDs of type *(K, V)* and *(K, W)*, returns an RDD of *(K, Iterable<V>, Iterable<W>)* tuples; this operation is also called *groupWith*

Resilient Distributed Dataset (RDD)

Transformation	Meaning
cartesian(otherRDD)	When called on RDDs of types *T* and *U*, returns an RDD of all *(T, U)* pairs
pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script
➢ **coalesce**(numPartitions)	Decrease the number of partitions in the RDD to *numPartitions*; useful for running operations more efficiently after filtering down a large dataset
repartition(numPartitions)	Reshuffle 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**
repartitionAndSortWithinPartitions (partitioner)	Repartition the RDD according to the given *partitioner* and, within each resulting partition, sort records by their keys

Resilient Distributed Dataset (RDD)

Action	Meaning
➢ **reduce**(func)	Aggregate the elements of the RDD using a function *func* (which takes two arguments and returns one)
➢ **collect**()	Return all the elements of the RDD as an array at the driver program
➢ **count**()	Return the number of elements in the RDD
➢ **first**()	Return the first element of the RDD (similar to *take(1)*)
➢ **take**(n)	Return an array with the first *n* elements of the RDD
takeSample(withReplacement, num, [seed])	Return an array with a random sample of *num* elements of the RDD, with or without *replacement*, optionally pre-specifying a random number generator *seed*
takeOrdered(n, [ordering])	Return the first *n* elements of the RDD using either their natural order or a custom comparator

Resilient Distributed Dataset (RDD)

Action	Meaning
➢ **saveAsTextFile**(path)	Write the elements of the RDD 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 **Note**: Spark will call toString on each element to convert it to a line of text in the file
saveAsSequenceFile(path) (Java and Scala)	Write the elements of the RDD as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system
saveAsObjectFile(path) (Java and Scala)	Write the elements of the RDD in a simple format using Java serialization, which can then be loaded using ***SparkContext.objectFile()***
➢ **countByKey**()	Returns a hashmap of *(K, Int)* pairs with the count of each key **Note**: Only available on RDDs of type *(K, V)*
➢ **foreach**(func)	Run a function *func* on each element of the RDD

Hands-on

- Open “00.Introduction_to_Apache_Spark.ipynb” in Google Colab:
  - Execute examples 3, 4 and 5
  - Try exercise 3 and 4

1. Before Apache Spark
2. Introducing Apache Spark
3. Apache Spark. Architecture
4. RDD
5. Key/Value Pairs RDD 
6. Executing Spark
7. Architectures

Key/Value Pairs RDD

- Spark provides special operations on RDDs containing **(key, value)** pairs
- They expose operations that allow you to act on each key in parallel, regroup or agregate data across the network.
- Creating pair RDD:
  - Many formats loading return pair RDDs for key/value data (e.g. Avro,Json)
  - Parallelize (for testing and PoCs)

```
val pairs = sc.parallelize (List((1,"a"),(2,"b"),(3,"c")))
```

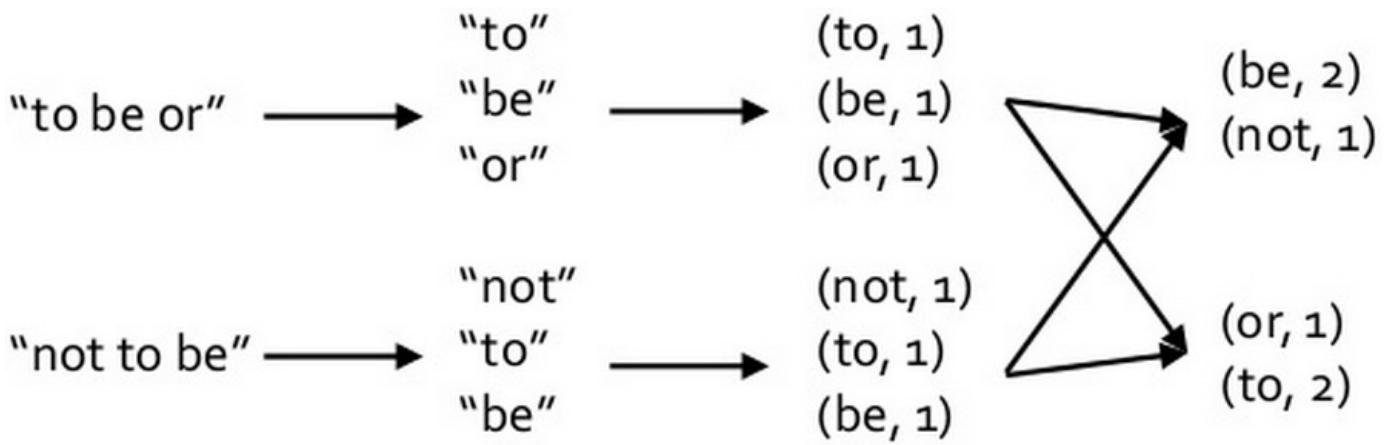
  - Running transformations over regular RDDs

```
//pair RDD using the first word as the key
val pairs = lines.map(linea => (linea.split(" ")(0), linea))
```

Key/Value Pairs RDD

- Word count example

```
text_file = sc.textFile("hamlet.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda a, b: a + b)  \
                  .counts.saveAsTextFile("hdfs://...")
```



Key/Value Pairs RDD

- Join and Cogroup examples

```
> visits = sc.parallelize([ ("index.html", "1.2.3.4"),
                            ("about.html", "3.4.5.6"),
                            ("index.html", "1.3.3.1") ])

> pageNames = sc.parallelize([ ("index.html", "Home"),
                               ("about.html", "About") ])

> visits.join(pageNames)
  # ("index.html", ("1.2.3.4", "Home"))
  # ("index.html", ("1.3.3.1", "Home"))
  # ("about.html", ("3.4.5.6", "About"))

> visits.cogroup(pageNames)
  # ("index.html", (["1.2.3.4", "1.3.3.1"], ["Home"]))
  # ("about.html", (["3.4.5.6"], ["About"]))
```

Hands-on

- Open “00_Introduction_to_Apache_Spark.ipynb” in Google Colab:
  - Execute examples 6
  - Try exercise 5

1. Before Apache Spark
2. Introducing Apache Spark
3. Apache Spark. Architecture
4. RDD
5. Key/Value Pairs RDD
6. Executing Spark 
7. Architectures

Executing Spark out of Notebook

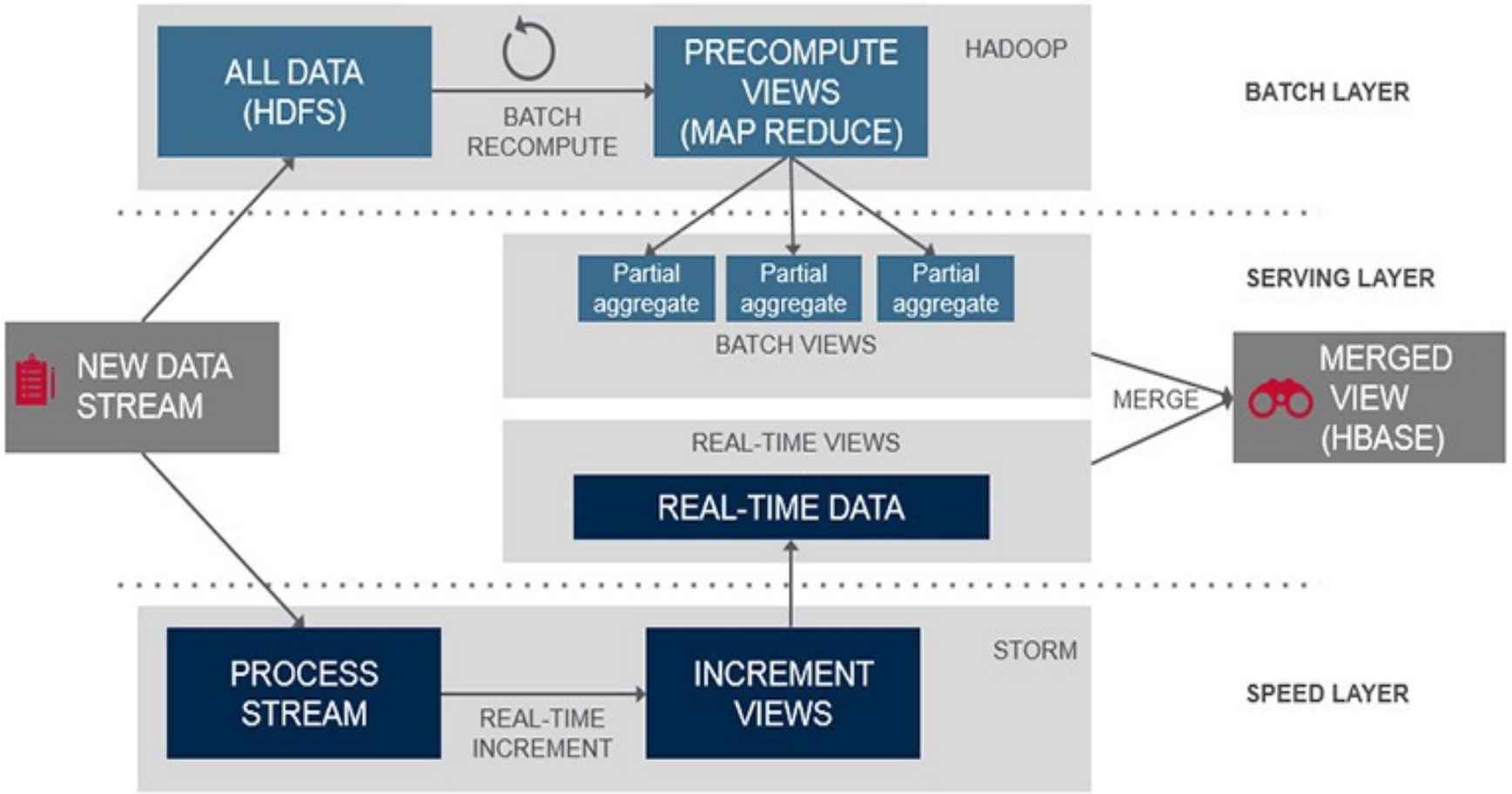
- **Spark-submit** is the command to launch Spark in a cluster/single machine
- Example:

  $ bin/spark-submit --master yarn --deploy-mode cluster --py-files otralib.zip,otrofich.py --num-executors 10 --executor-cores 2 mi-script.py script-options

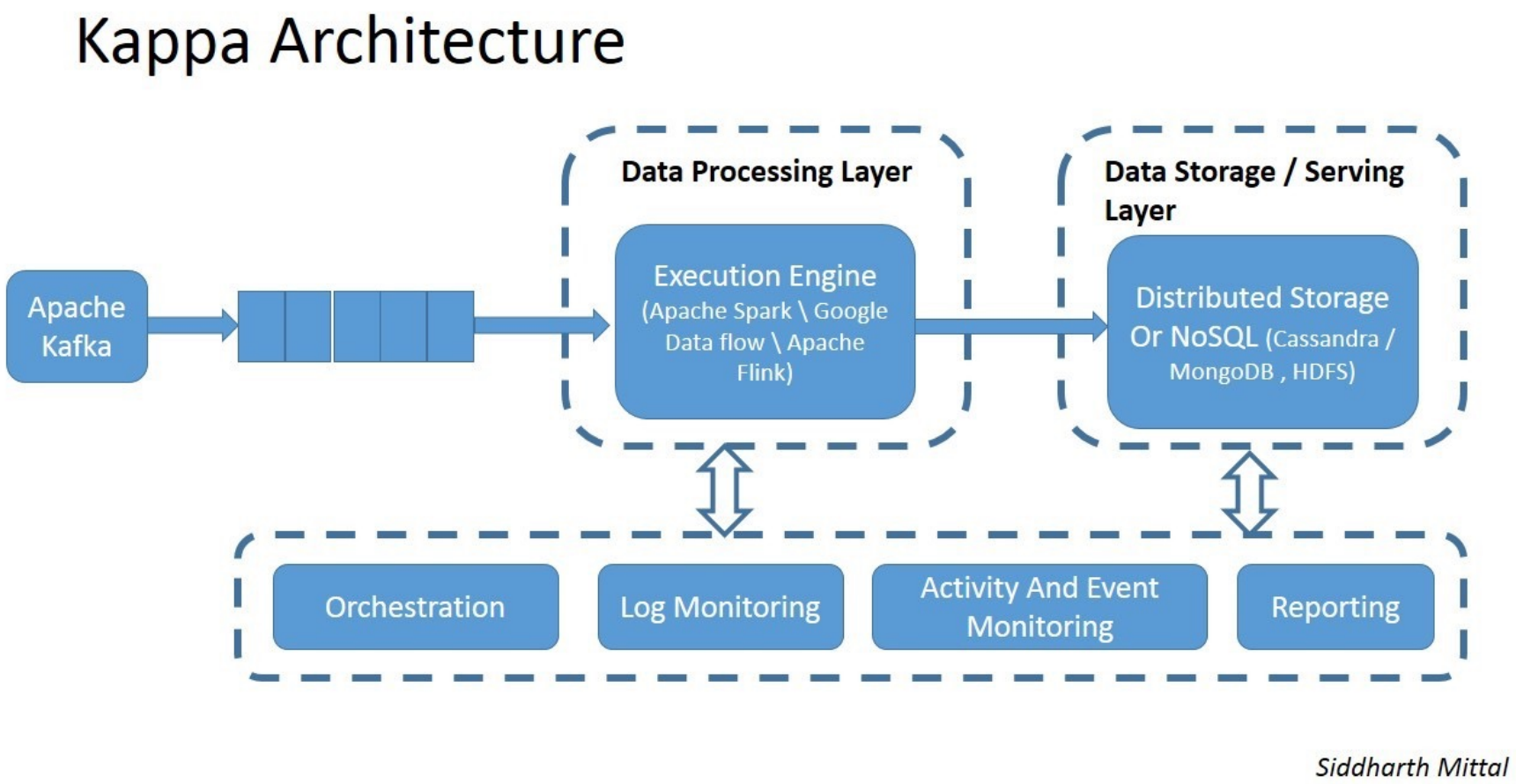
- Spark-submit options
  - master: cluster manager a usar (opciones: yarn, mesos://host:port, spark://host:port, local[n])
  - deploy-mode: dos modos de despliegue (client: local // cluster: cluster)
  - class: clase a ejecutar (Java o Scala)
  - name: nombre de la aplicación (se muestra en el Spark web)
  - jars: ficheros jar a añadir al classpath (Java o Scala)
  - py-files: archivos a añadir al PYTHONPATH (.py,.zip,.egg)
  - files: ficheros de datos para la aplicación
  - executor-memory: memoria total de cada ejecutor
  - driver-memory: memoria del proceso driver

1. Before Apache Spark
2. Introducing Apache Spark
3. Apache Spark. Architecture
4. RDD
5. Key/Value Pairs RDD
6. Executing Spark
7. Architectures

Lambda Architecture



Kappa Architecture



Siddharth Mittal

Modern Streaming Architecture