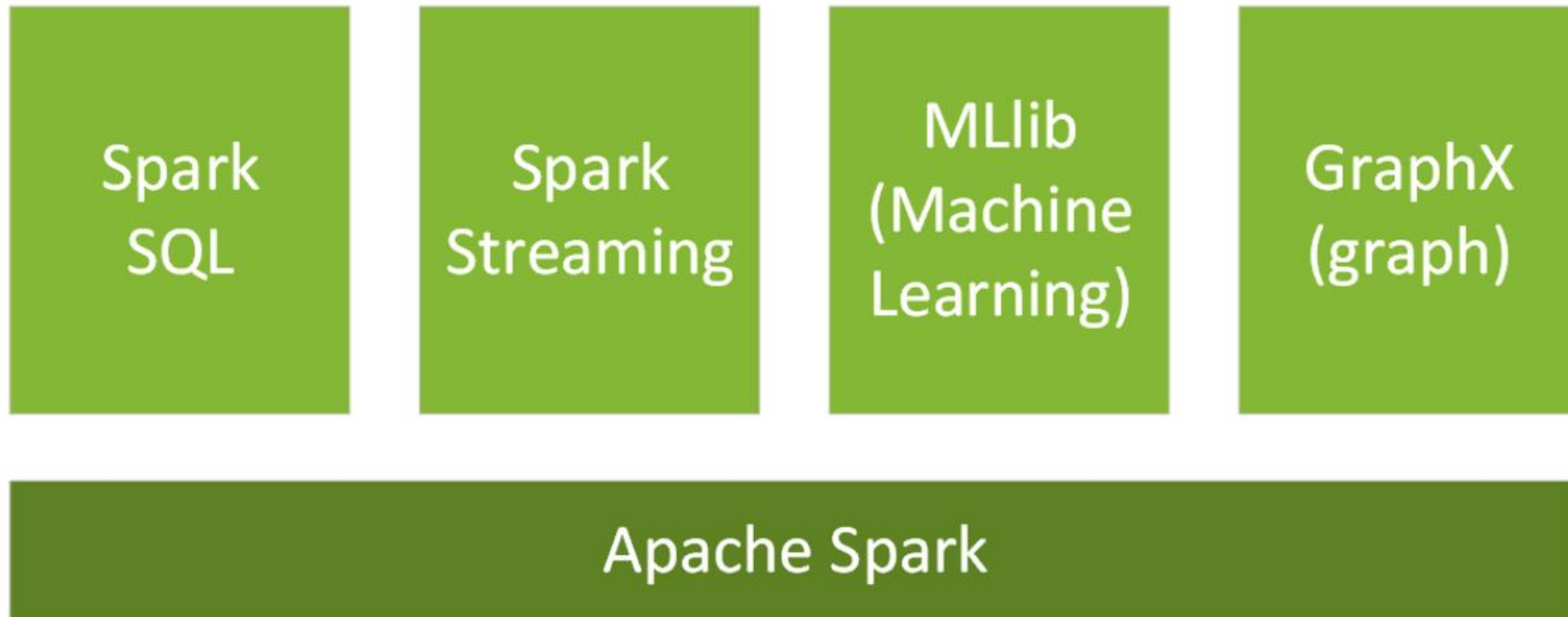
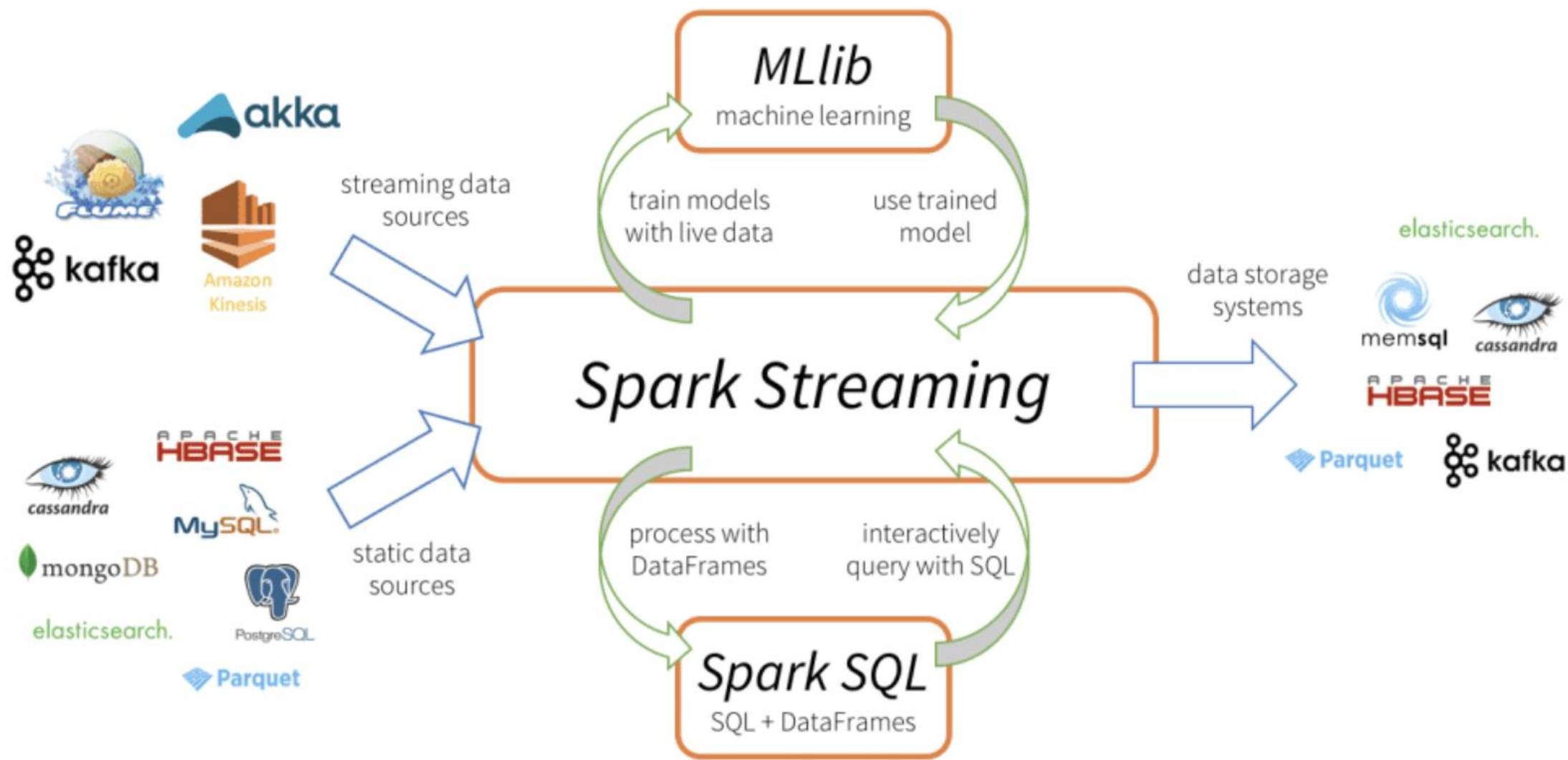


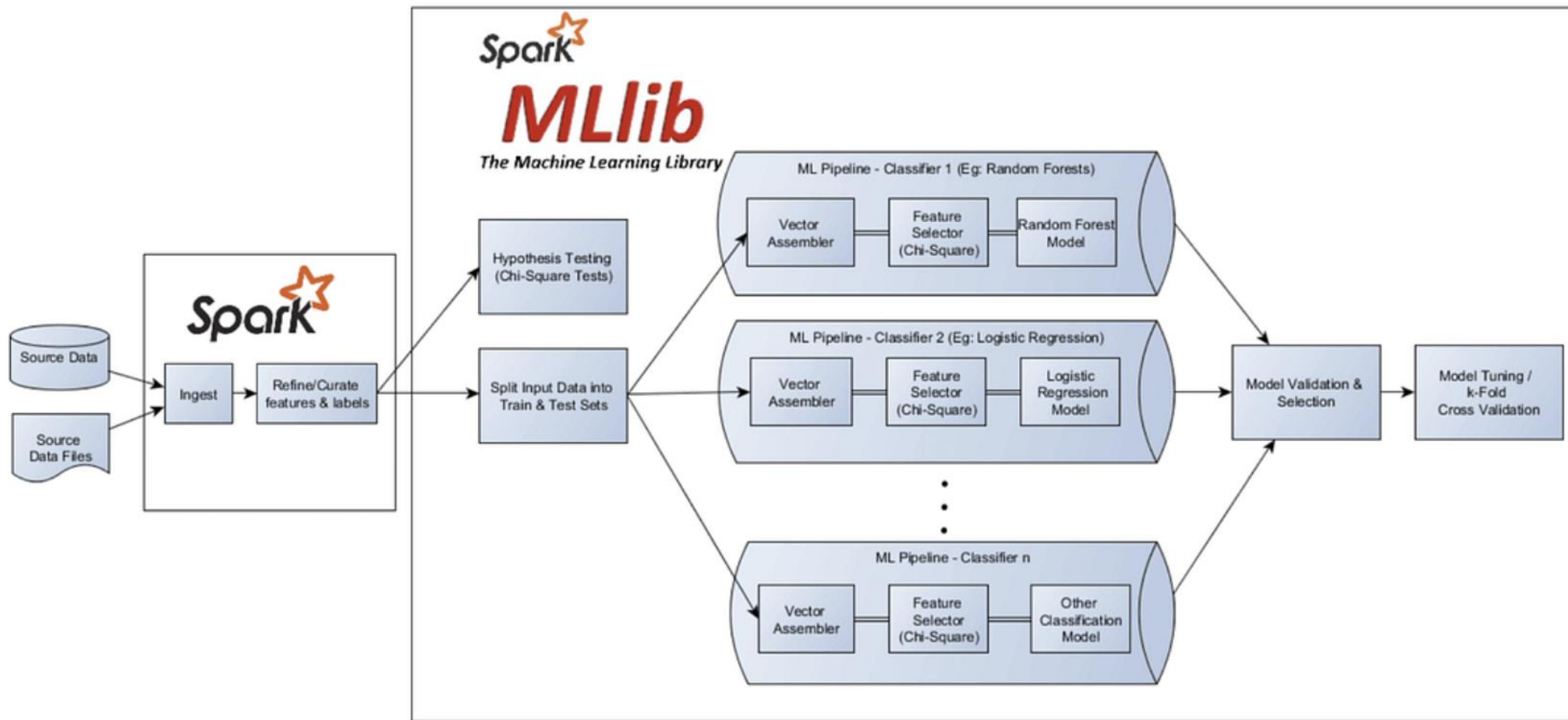
Spark and PySpark: internals, environment and examples

Apache Spark Stack



Application



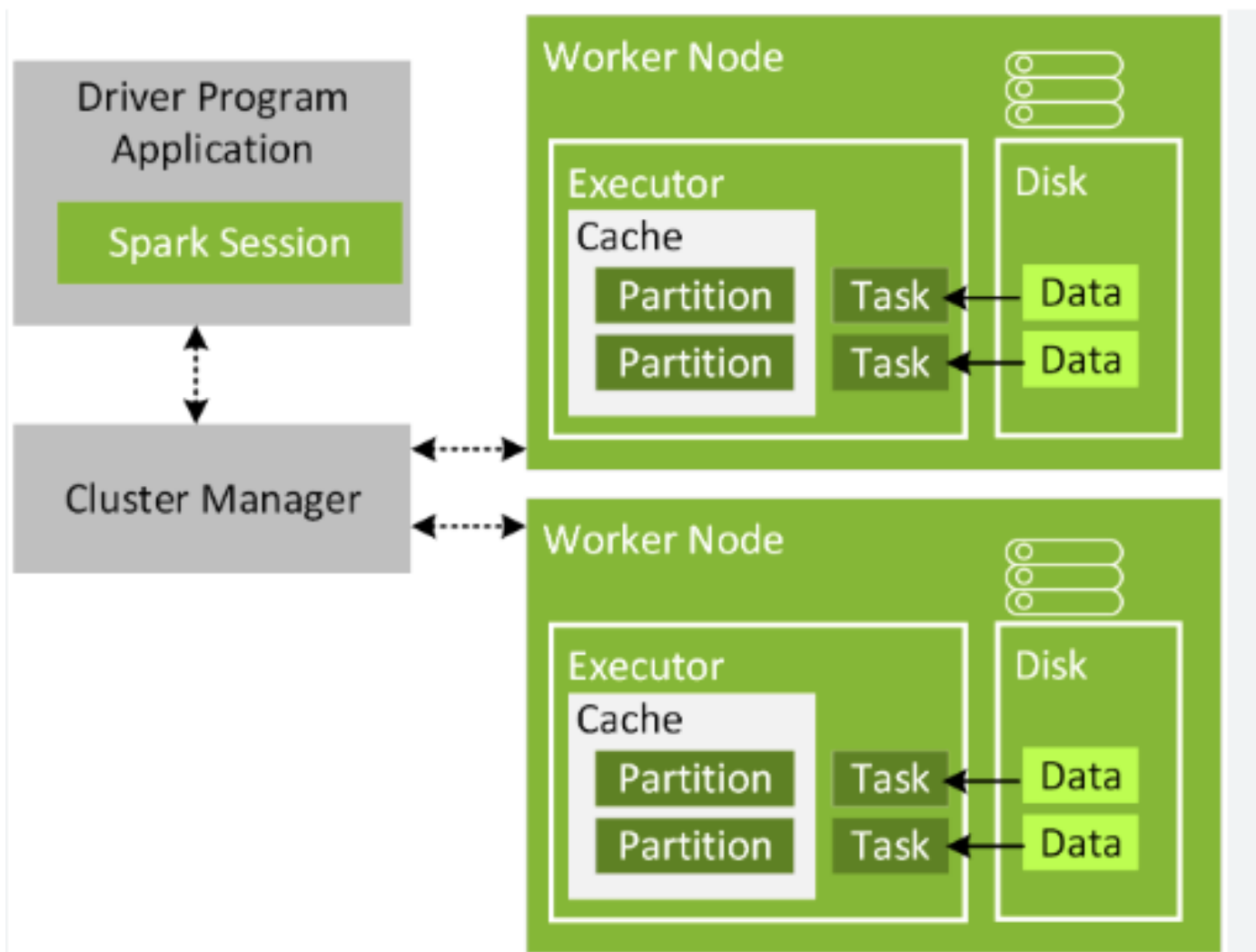


MLlib is Spark's machine learning (ML):

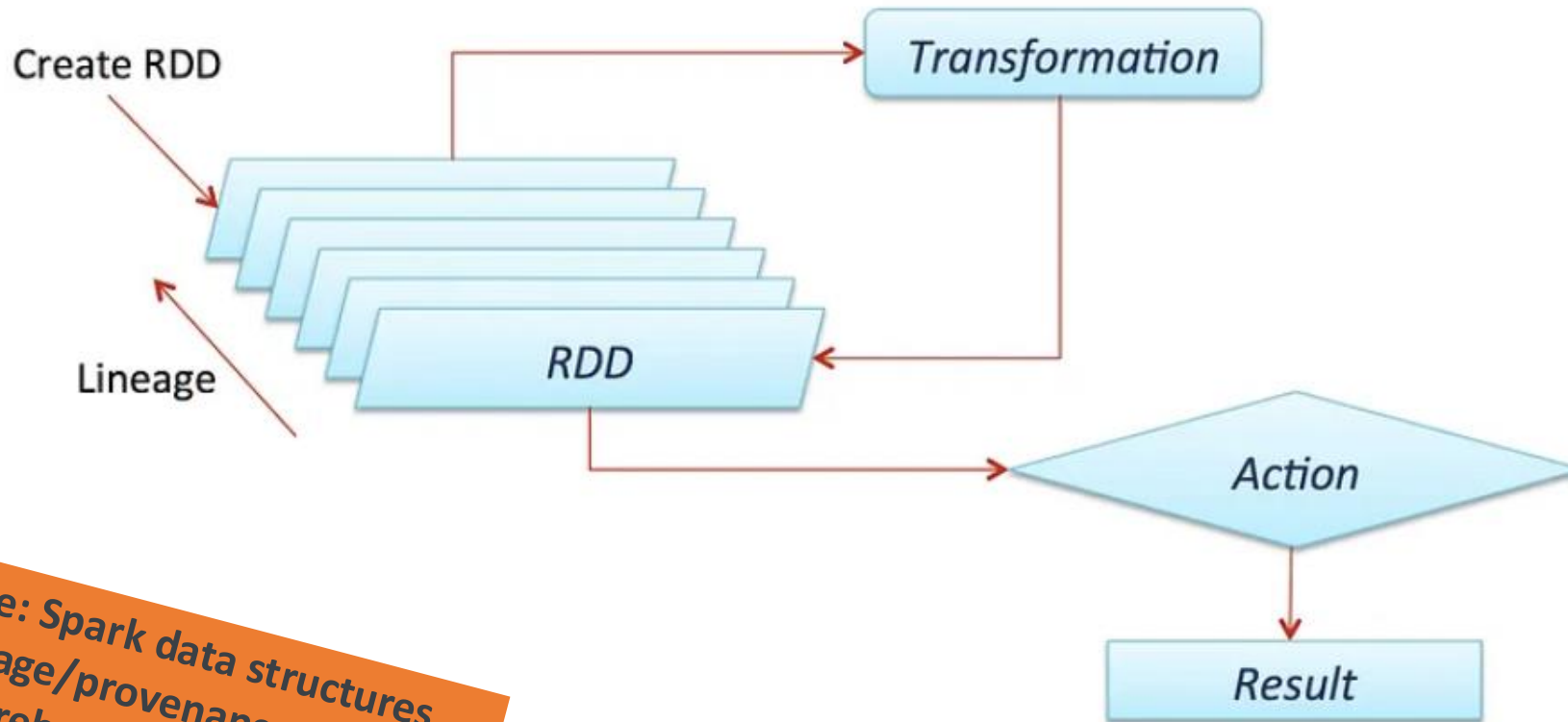
ML Algorithms: classification, regression, clustering, and collaborative filtering

- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- Persistence: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc.

Spark Internals



Lazy evaluation



Fault tolerance: Spark data structures track data lineage/provenance information to rebuild lost data automatically on failure

PySpark

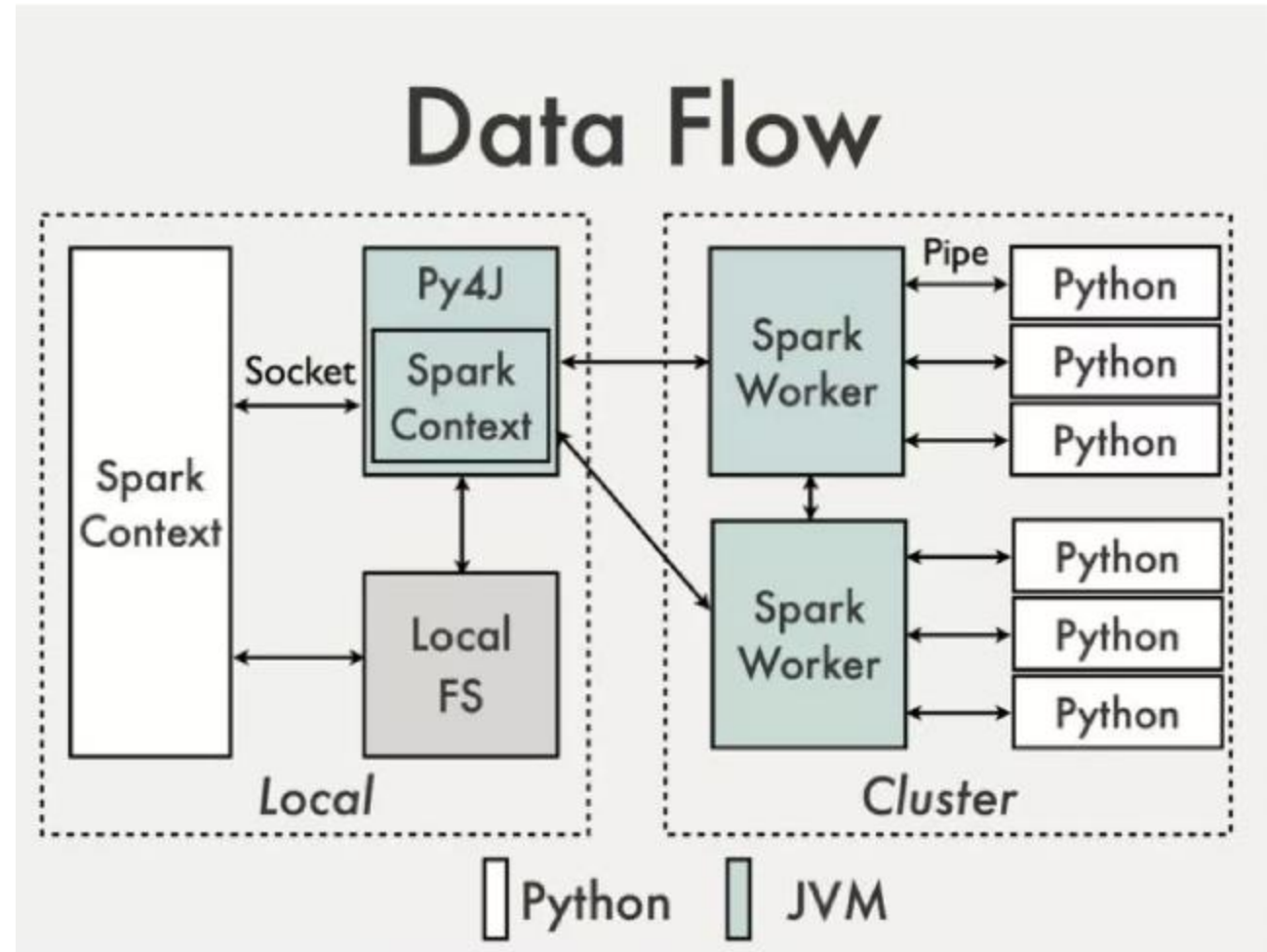


PySpark is the Python API for Apache Spark to perform real-time, large-scale data processing in a distributed environment using Python.

It also provides a PySpark shell for interactively analyzing your data.

Current Version: 3.5.3

<https://pypi.org/project/pyspark/>



Spark Architecture

SparkSession is the entry point to Spark

SparkSession internally creates SparkConfig and SparkContext with the configuration provided with SparkSession.

Example of configuration:

master(String master):
sets the Spark master URL to connect to

"local[*]" run locally and use as many threads as possible

"local[K]" to run locally with K cores,

"yarn" to run on a cluster with **Yarn** as **cluster manager**

```
1 import findspark
2 findspark.init()
3 findspark.find()
4
5 import pyspark
6 from pyspark.sql import SparkSession
7
8 spark= SparkSession \
9     .builder \
10    .appName("Our First Spark Example") \
11    .getOrCreate()
12 sc=spark.sparkContext
```

```
[ ] 1 sc
```

SparkContext

[Spark UI](#)

Version

v3.5.0

Master

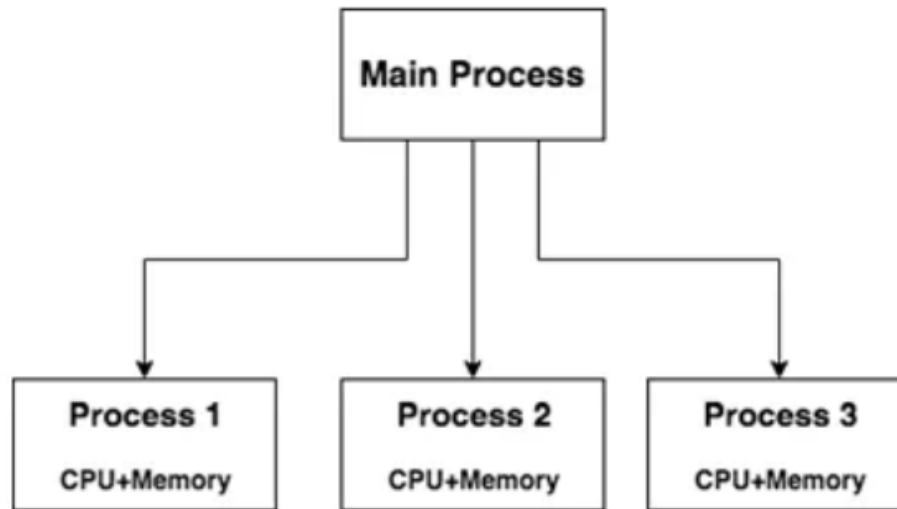
local[*]

AppName

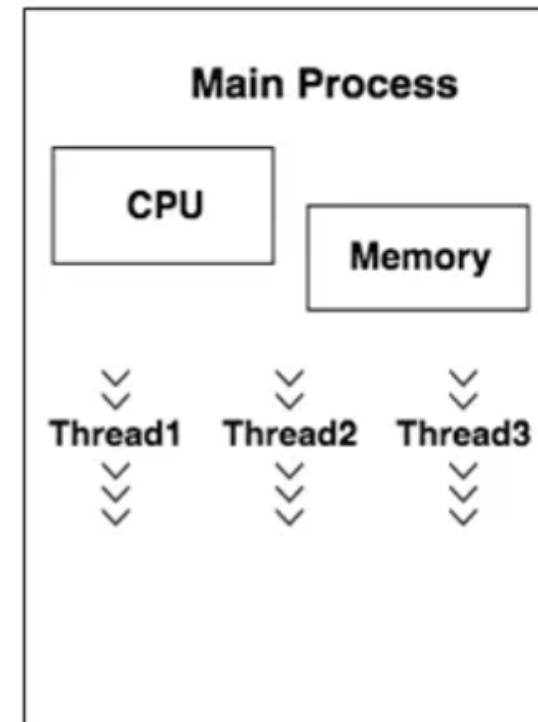
Our First Spark Example

Inciso: Multiprocessing vs multithreading

Multiprocessing

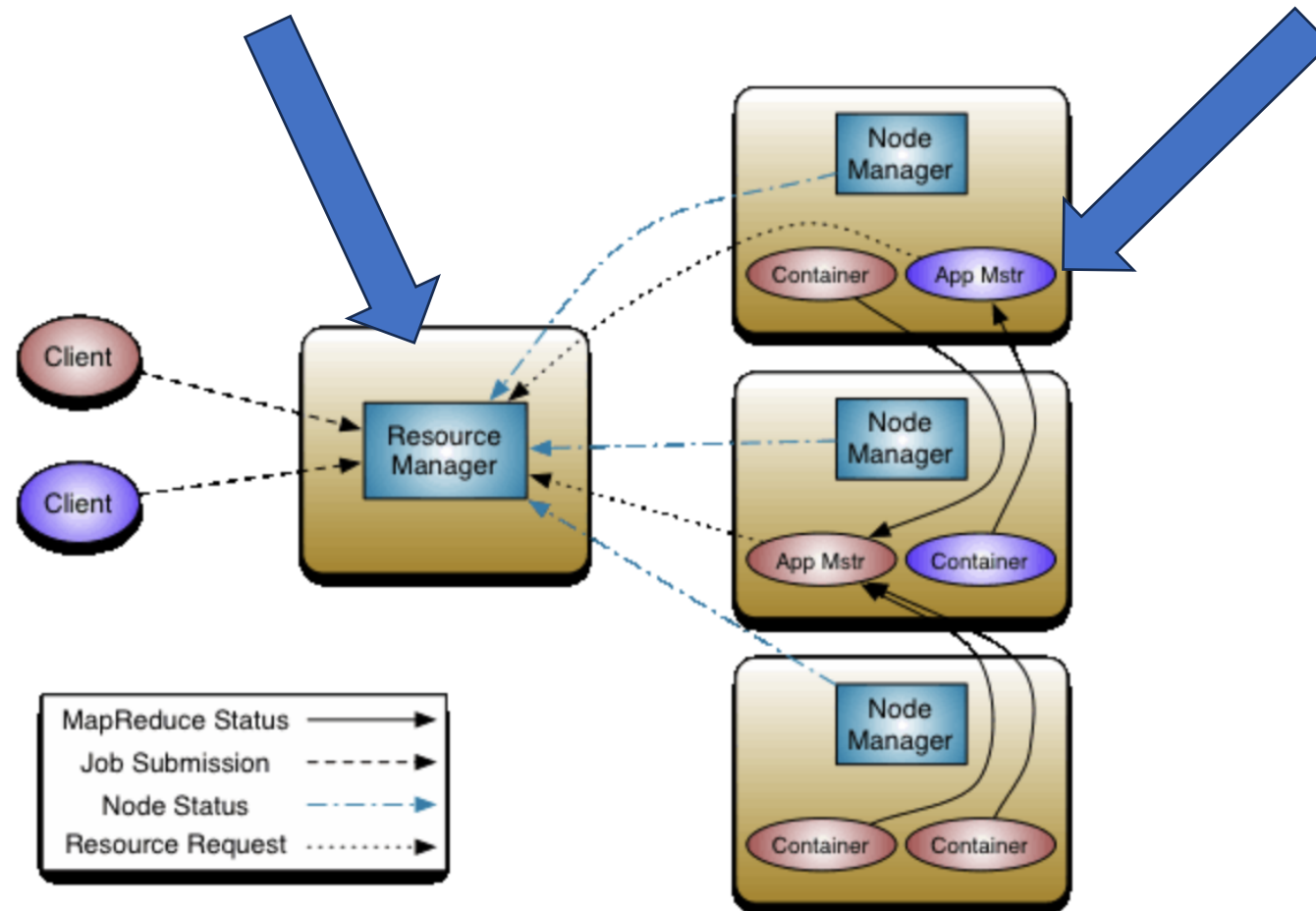


Multithreading



What is Apache Hadoop Yarn?

A cluster manager that split up the functionalities of resource management and job scheduling/monitoring into separate daemons



A container is the physical instance of a process executed by YARN on a worker node within the cluster. The context of the container defines the executable to run, the environment, arguments to the executable, and other information set by the client.

Yarn components

NodeManager is the per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager/Scheduler.

The ResourceManager is the authority that arbitrates resources among all the applications in the system

- The Scheduler is responsible for allocating resources to the various running applications using the abstract notion of a resource *Container* which incorporates elements such as memory, cpu, disk, network etc. The Scheduler applies a policy for partitioning the cluster resources among the various queues, applications etc.
- The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks. The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific and provides the service for restarting the ApplicationMaster container on failure.

Where to use PySpark

Local machine

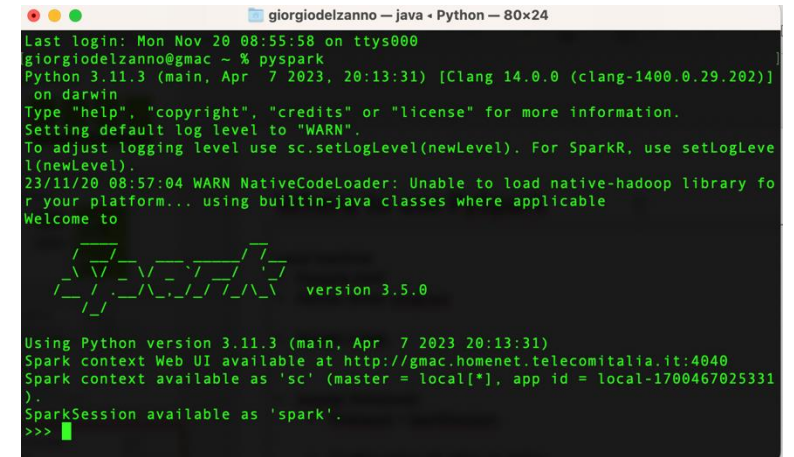
- Pyspark shell
- Python driver program

Google Colab

Jupyter Notebook:

- findspark + SparkSession
- Configuration of .zshrc or .bashrc

```
export PYSARK_DRIVER_PYTHON=jupyter
export PYSARK_DRIVER_PYTHON_OPTS='notebook'
```

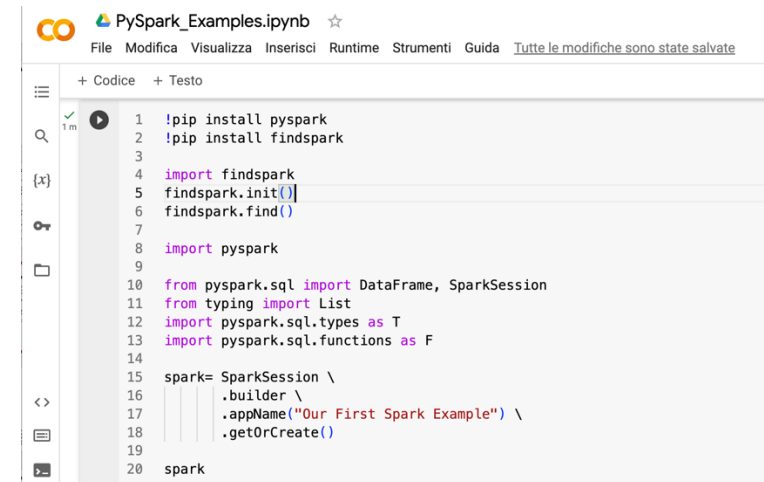


```
giorgiodelzanno — java • Python — 80x24
Last login: Mon Nov 20 08:55:58 on ttys000
giorgiodelzanno@mac ~ % pyspark
Python 3.11.3 (main, Apr 7 2023, 20:13:31) [Clang 14.0.0 (clang-1400.0.29.202)]
on darwin
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/11/20 08:57:04 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Welcome to

      _ _ _ _ _
     / _ _ _ \
    / _ _ _ \
   / _ _ _ \
  / _ _ _ \
 / _ _ _ \
/_ _ _ _ \

version 3.5.0

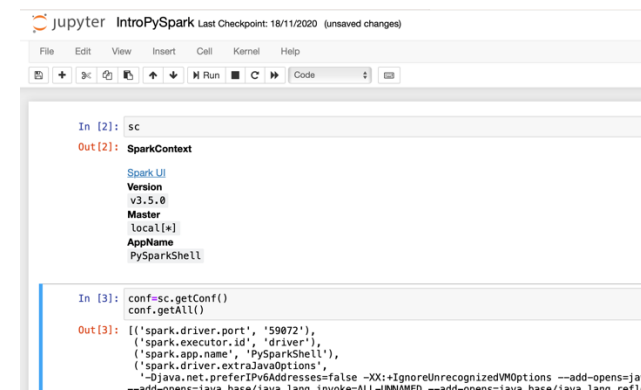
Using Python version 3.11.3 (main, Apr 7 2023 20:13:31)
Spark context Web UI available at http://gmac.homenet.telecomitalia.it:4040
Spark context available as 'sc' (master = local[*], app id = local-1700467025331).
SparkSession available as 'spark'.
>>>
```



```
PySpark_Examples.ipynb
File Modifica Visualizza Inserisci Runtime Strumenti Guida Tutte le modifiche sono state salvate

+ Codice + Testo

1 !pip install pyspark
2 !pip install findspark
3
4 import findspark
5 findspark.init()
6 findspark.find()
7
8 import pyspark
9
10 from pyspark.sql import DataFrame, SparkSession
11 from typing import List
12 import pyspark.sql.types as T
13 import pyspark.sql.functions as F
14
15 spark= SparkSession \
16     .builder \
17     .appName("Our First Spark Example") \
18     .getOrCreate()
19
20 spark
```



```
jupyter IntroPySpark Last Checkpoint: 18/11/2020 (unsaved changes)
File Edit View Insert Cell Kernel Help

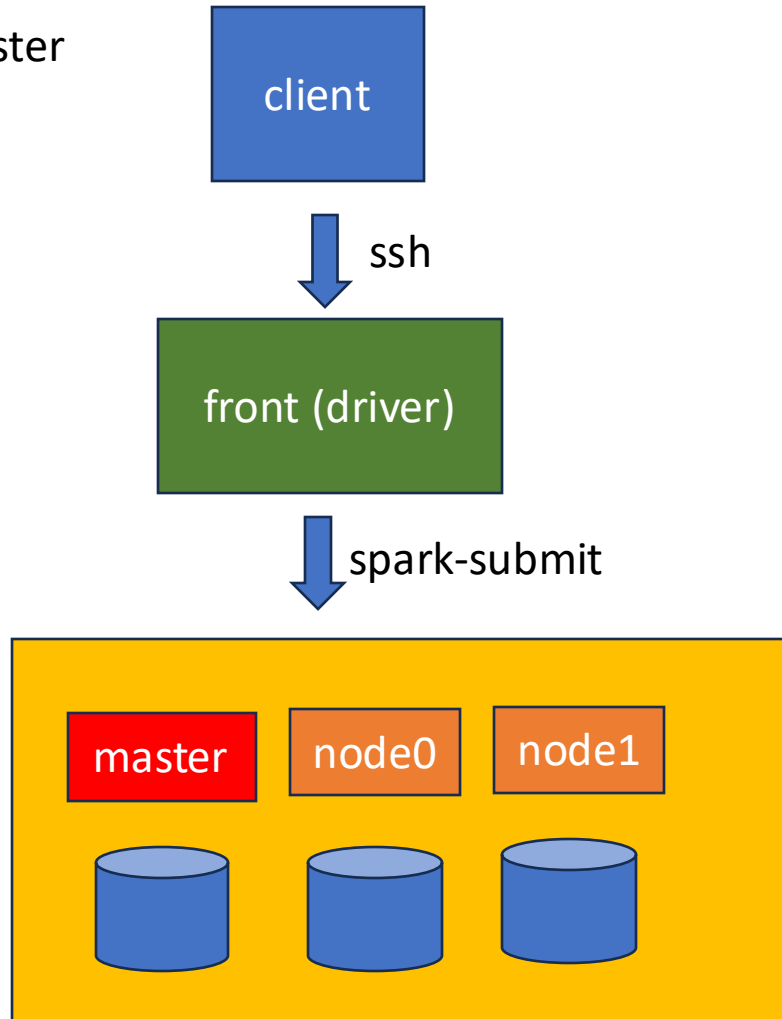
In [2]: sc
Out[2]: SparkContext

Spark UI
Version
v3.5.0
Master
local[*]
AppName
PySparkShell

In [3]: conf=sc.getConf()
conf.getAll()
Out[3]: [{'spark.driver.port', '59072'},
{'spark.executor.id', 'driver'},
{'spark.app.name', 'PySparkShell'},
{'spark.driver.extraJavaOptions',
'-Djava.net.preferIPv6Addresses=false -XX:IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang.invoke=ALL-UNNAMED --add-opens=java.base/java.lang.reflect=ALL-UNNAMED'}]
```

Where to use PySpark

LSC cluster



```
giorgiodelzanno — user_lsc_78@it:~/RDDLab — ssh user_lsc_78@130.251.61.97 — 80x24

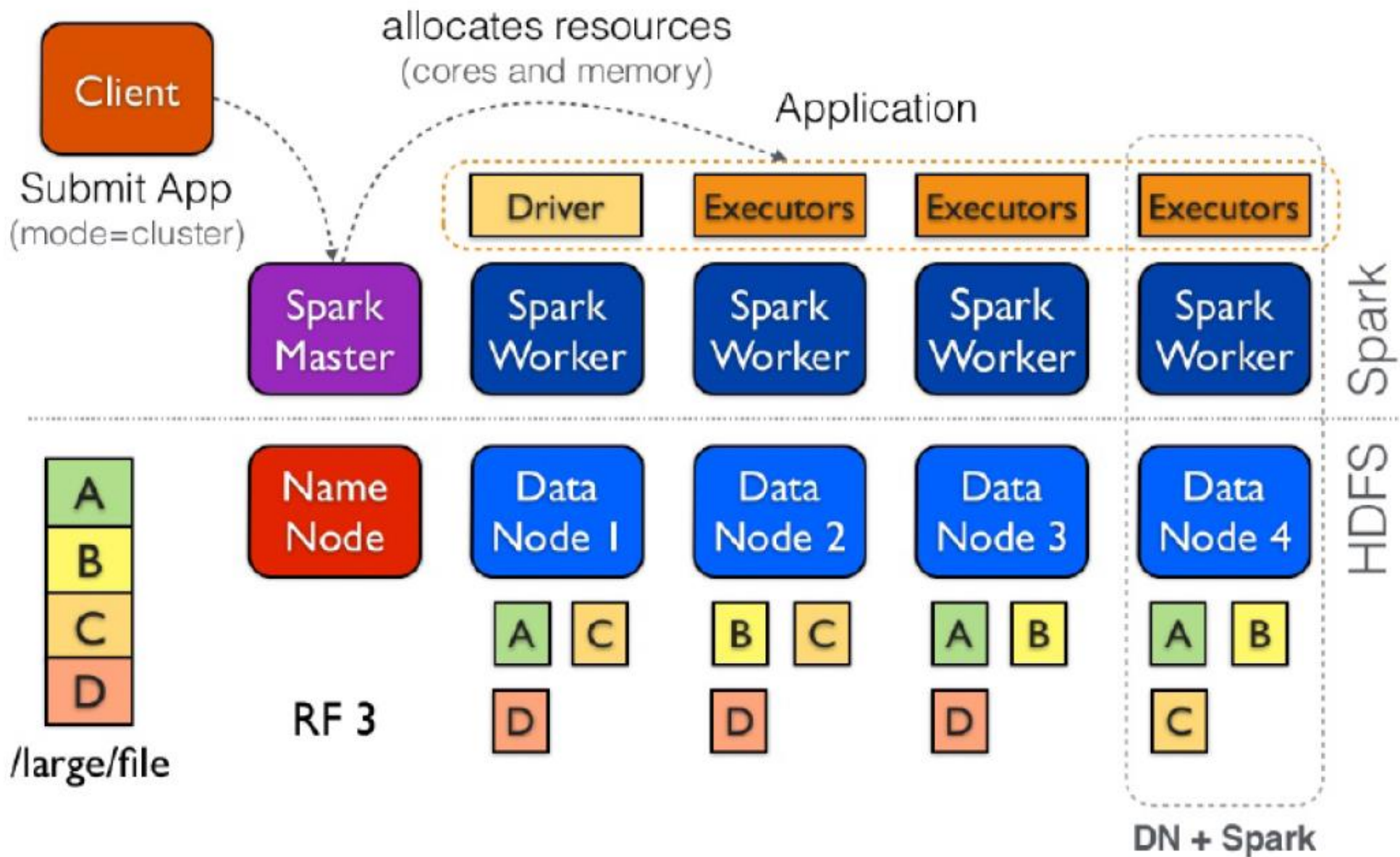
giorgiodelzanno@gmac ~ % ssh user_lsc_78@130.251.61.97
user_lsc_78@130.251.61.97's password:
Last failed login: Mon Nov 20 09:15:03 CET 2023 from host-87-14-124-175.retail.t
elecomitalia.it on ssh:notty
There was 1 failed login attempt since the last successful login.
Last login: Mon Nov 20 09:14:27 2023 from host-87-14-124-175.retail.telecomitali
a.it
[user_lsc_78@it ~]$ cd RDDLab/
[user_lsc_78@it RDDLab]$ ls
complete.py  genre.csv  lab.py      network.csv  simple.py.save
g1.csv       ll.csv     listenings.csv  simple.py    tmpnull
[user_lsc_78@it RDDLab]$ spark-submit --master local[*] simple.py
2023-11-20 09:15:21,824 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
2023-11-20 09:15:22,995 INFO spark.SparkContext: Running Spark version 3.0.1
2023-11-20 09:15:23,055 INFO resource.ResourceUtils: =====
=====
2023-11-20 09:15:23,056 INFO resource.ResourceUtils: Resources for spark.driver:
2023-11-20 09:15:23,057 INFO resource.ResourceUtils: =====
=====
```

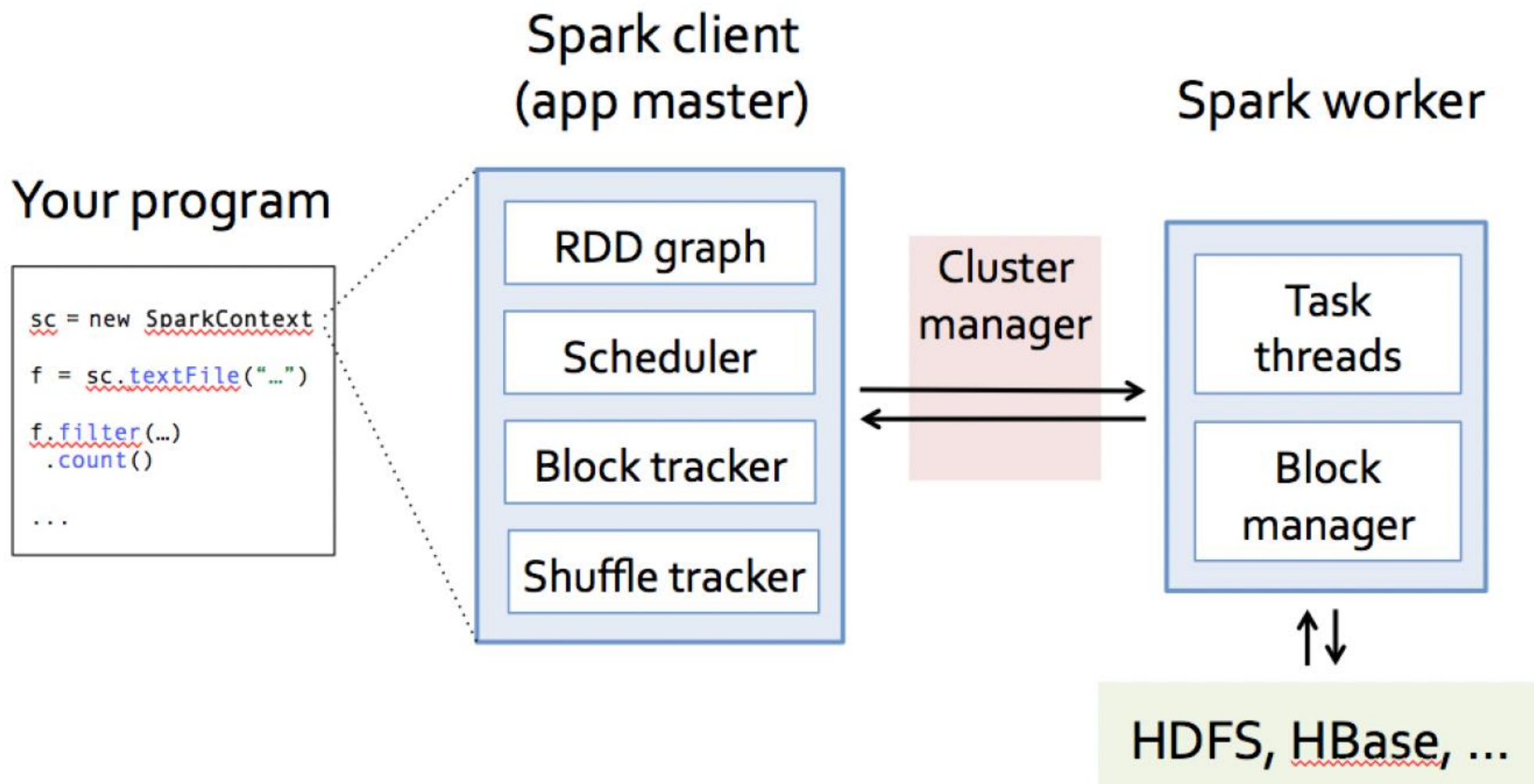
LSC Cluster frontend (i7) + 10 machines (i5) + HDFS

```
export SPARK_WORKER_CORES=4
export SPARK_MASTER_HOST=192.168.0.20
export SPARK_LOCAL_IP=192.168.0.20
export HADOOP_CONF_DIR="/usr/local/hadoop/etc/hadoop"
export YARN_CONF_DIR="/usr/local/hadoop/etc/hadoop"
export PYSPARK_PYTHON="/usr/bin/python3"
export PYSPARK_DRIVER_PYTHON="/usr/bin/python3"
```

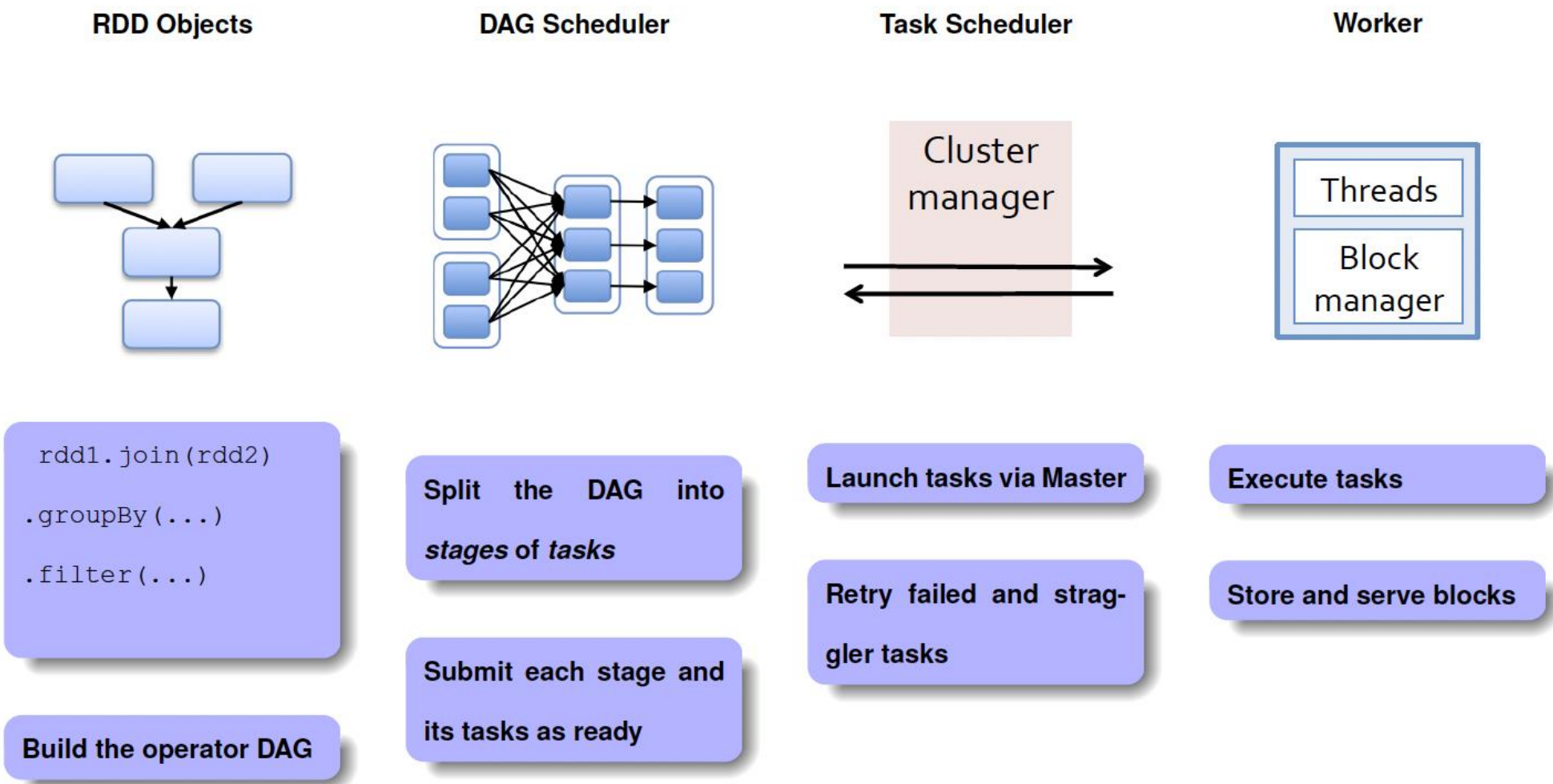
```
# A Spark Worker will be started on each of the machines listed below.
```

```
it.unige.dibris.lsccluster.node0
it.unige.dibris.lsccluster.node1
it.unige.dibris.lsccluster.node2
it.unige.dibris.lsccluster.node3
it.unige.dibris.lsccluster.node4
it.unige.dibris.lsccluster.node5
it.unige.dibris.lsccluster.node6
it.unige.dibris.lsccluster.node7
it.unige.dibris.lsccluster.node8
it.unige.dibris.lsccluster.node9
```



Lifetime of a Job in Spark



Application model for scheduling

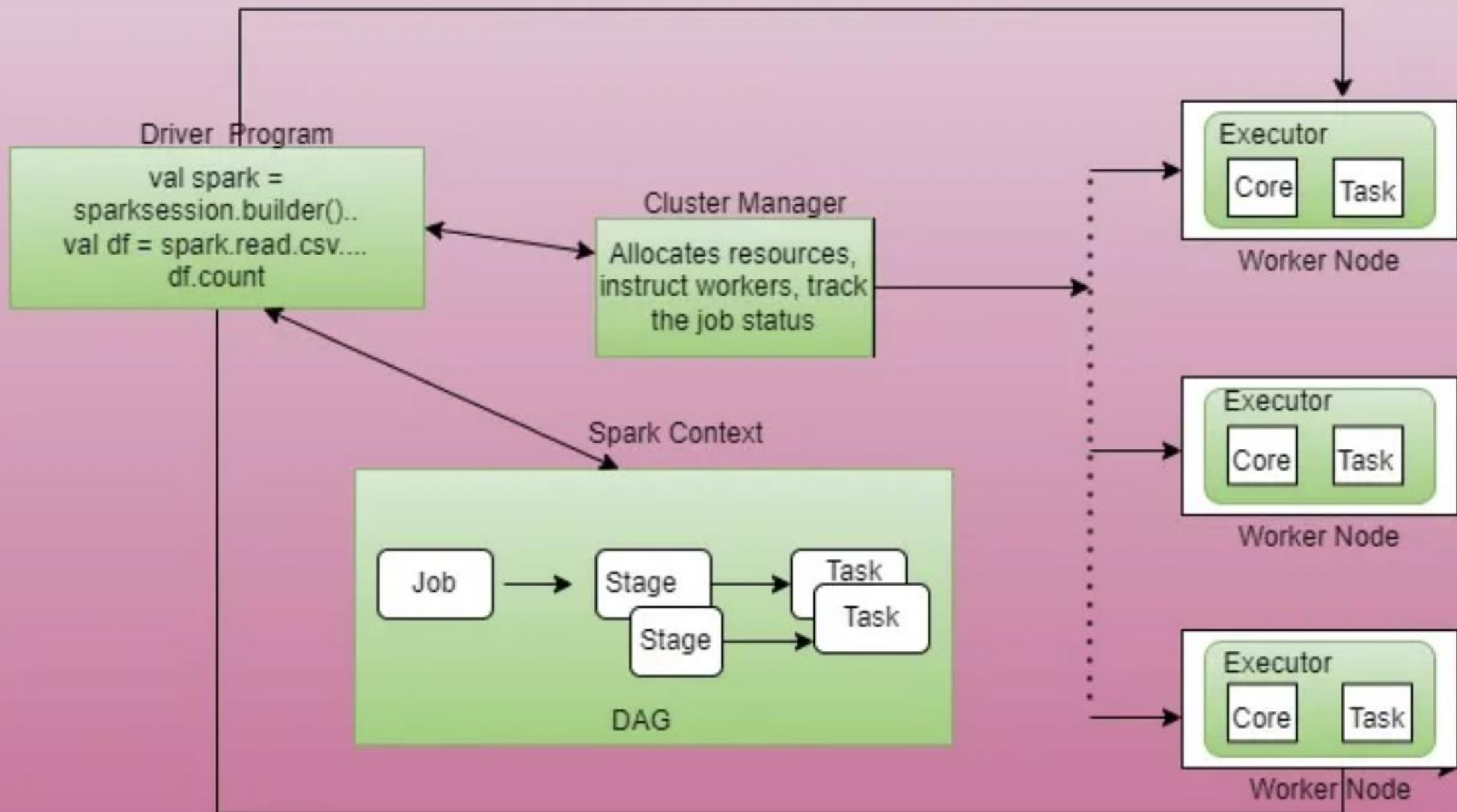
Application: Driver code that represents the DAG

Job: Subset of application triggered for execution by an “action” in the DAG

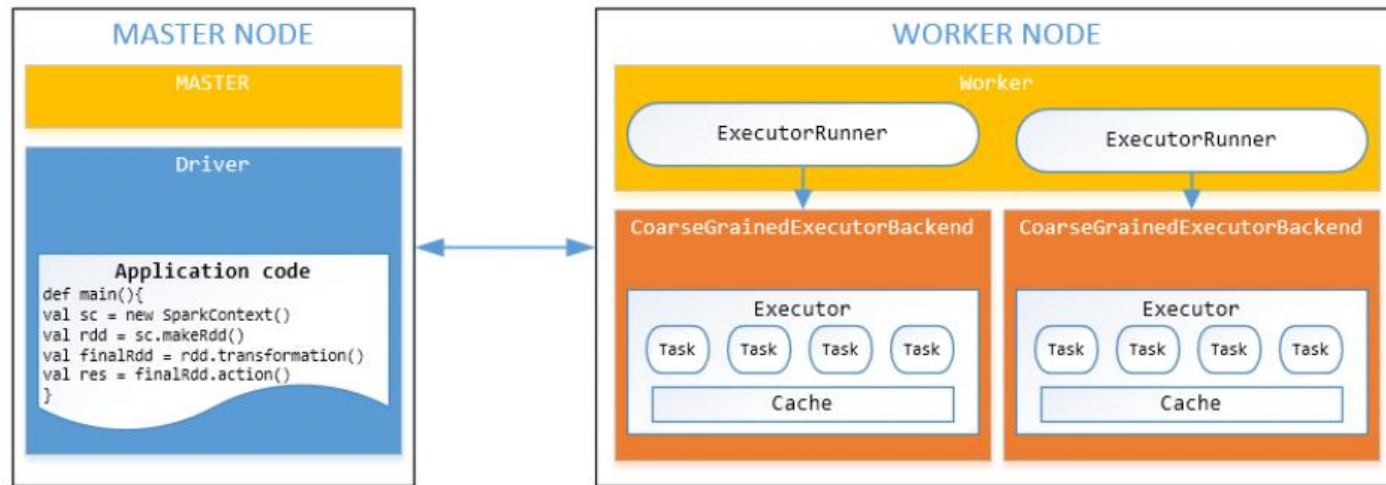
Stage: Job sub-divided into stages that have dependencies with each other

Task: Unit of work in a stage that is scheduled on a worker

Sends the serialized result(code+data)



Worker Nodes and Executors



- **Worker nodes are machines that run executors**
 - ▶ Host one or multiple `Workers`
 - ▶ One JVM (= 1 UNIX process) per `Worker`
 - ▶ Each `Worker` can spawn one or more `Executors`
- **Executors run tasks, used by 1 application, for whole lifetime**
 - ▶ Run in child JVM (= 1 UNIX process)
 - ▶ Execute one or more task using threads in a `ThreadPool`

Spark's scheduler property

1. FIFO

By default, Spark's scheduler runs jobs in FIFO fashion. Each job is divided into `stages` (e.g. map and reduce phases), and the `first` job gets priority on all available resources while its stages have tasks to launch, then the `second` job gets priority, etc. If the jobs at the head of the queue don't need to use the whole cluster, later jobs can start to run right away, but if the jobs at the head of the queue are large, then later jobs may be delayed significantly.

E.g. in test phase


2. FAIR

The fair scheduler also supports grouping jobs into pools and setting different scheduling options (e.g. weight) for each pool. This can be useful to create a `high-priority` pool for more important jobs, for example, or to group the jobs of each user together and give users equal shares regardless of how many concurrent jobs they have instead of giving jobs equal shares. This approach is modeled after the *Hadoop* Fair Scheduler.

E.g. in production phase

Without any intervention, newly submitted jobs go into a default pool, but jobs' pools can be set by adding the `spark.scheduler.pool` "local property" to the SparkContext in the thread that's submitting them.

Spark WebUI running at http://localhost:4040

3.5.0

Jobs

Stages

Storage

Environment

Executors

PySparkShell application UI

Spark Jobs (?)

User: giorgiodelzanno
Total Uptime: 6,6 min
Scheduling Mode: FIFO
Completed Jobs: 1

▼ Event Timeline

☐ Enable zooming

Executors

Added


Removed

Jobs

Succeeded

Failed

Running



	08:10	08:11	08:12	08:13	08:14
Mon 20 November					

▼ Completed Jobs (1)

Page: 1

1 Pages. Jump to 1. Show 100 items in a page. Go

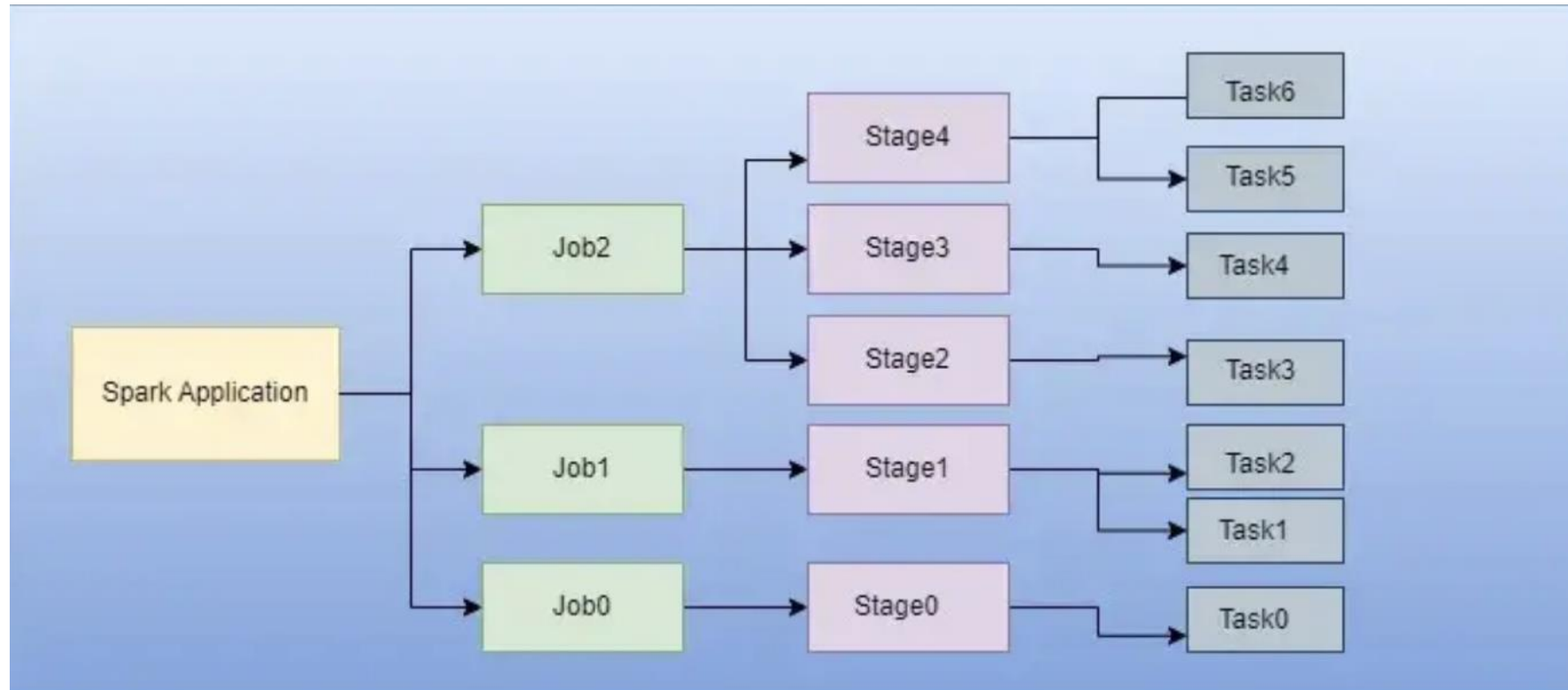
Job Id ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at /var/folders/2m/8c0190kj4n93sz7fj15kn4hm0000gn/T/ipykernel_29328/29041998... collect at	2023/11/20 08:14:49	0,5 s	1/1	8/8

Stage and task

A Stage is a collection of tasks that share the same shuffle dependencies, meaning that they must exchange data with one another during execution.

When a Spark job is submitted, it is broken down into stages based on the operations defined in the code.

Each stage is composed of one or more tasks that can be executed in parallel across multiple nodes in a cluster. Stages are executed sequentially, with the output of one stage becoming the input to the next stage.



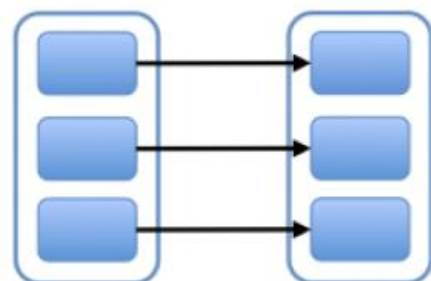
Narrow/wide stages

A typical Spark job consists of multiple stages. Each stage is a sequence of transformations and actions on the input data. When a Spark job is submitted, Spark evaluates the execution plan and divides the job into multiple stages based on the dependencies between the transformations. Spark executes each stage in parallel, where each stage can have multiple tasks running on different nodes in the cluster.

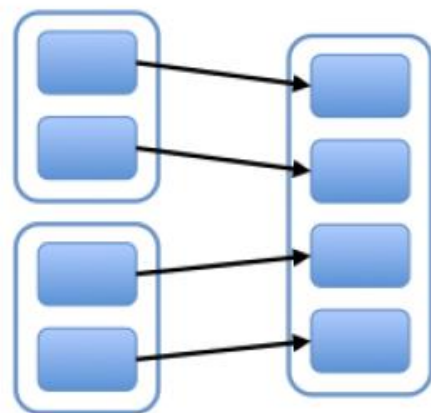
Narrow Stages: Narrow stages are stages where the data does not need to be shuffled. Each task in a narrow stage operates on a subset of the partitions of its parent RDD. Narrow stages are executed in parallel and can be pipelined.

Wide Stages: Wide stages are stages where the data needs to be shuffled across the nodes in the cluster. This is because each task in a wide stage operates on all the partitions of its parent RDD. Wide stages involve a data shuffle and are typically more expensive than narrow stages.

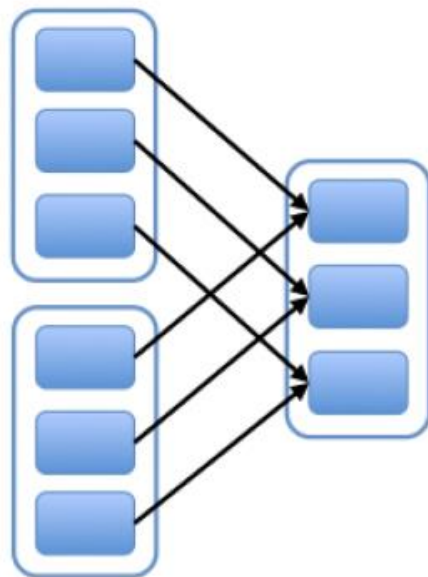
Narrow dependencies



map, filter

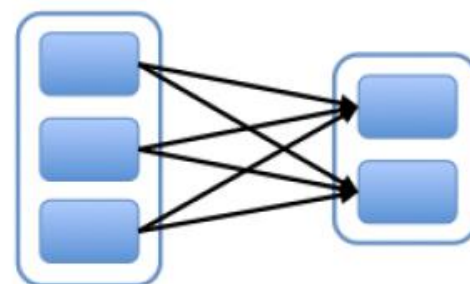


union

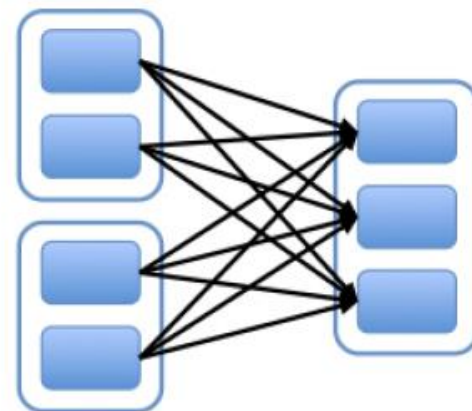


join with
co-partitioned
inputs

Wide dependencies



groupByKey

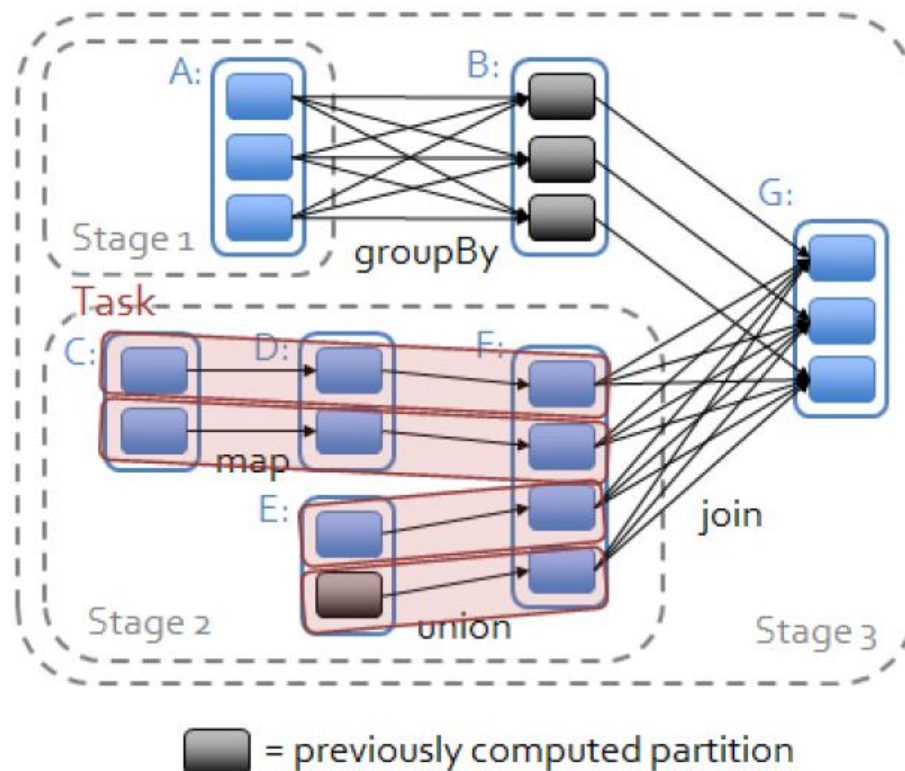


join with inputs not
co-partitioned

Dependency Types: Optimizations

- **Benefits of Lazy evaluation**

- ▶ The DAG Scheduler optimizes *Stages* and *Tasks* before submitting them to the Task Scheduler
- ▶ **Piplining** narrow dependencies within a Stage
- ▶ **Join plan selection** based on partitioning
- ▶ **Cache reuse**

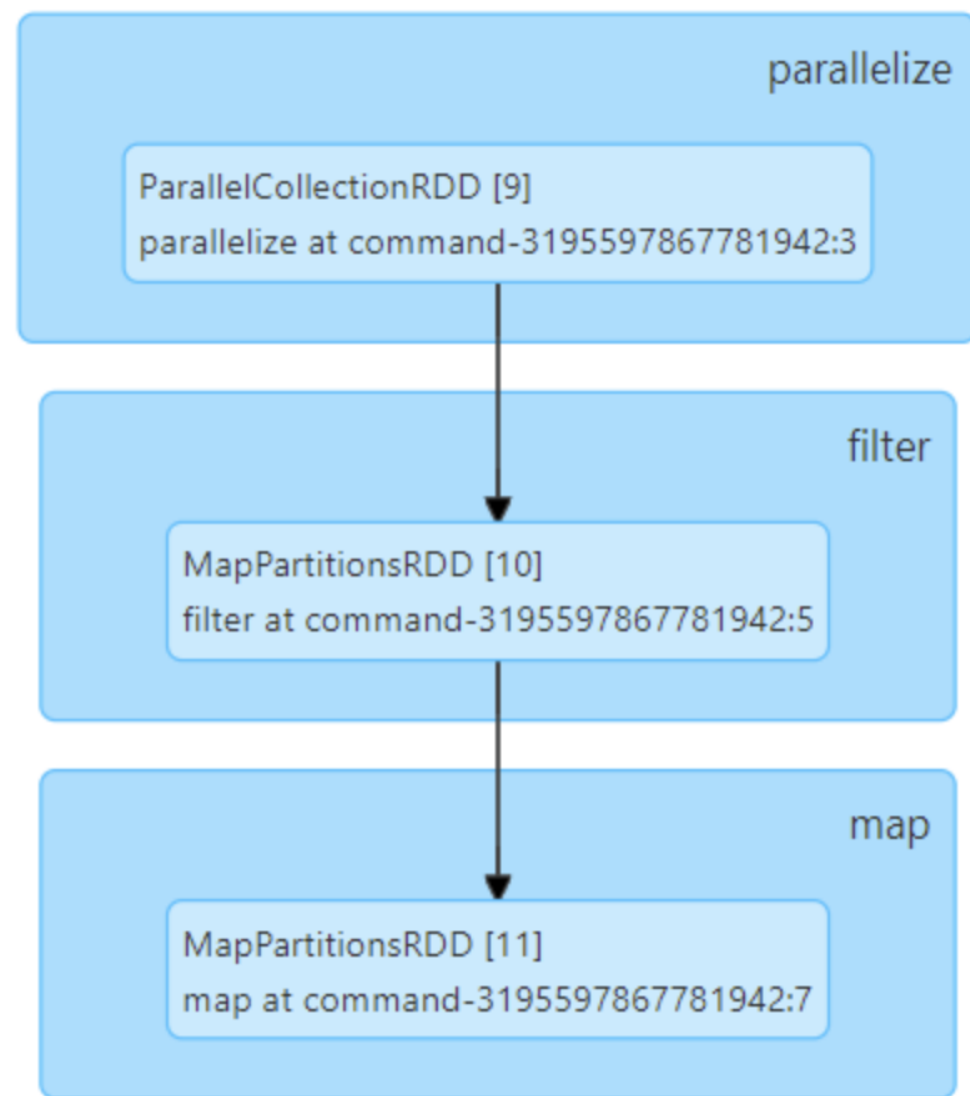


```
# Create RDD
val data = spark.sparkContext.parallelize(Seq(1, 2, 3, 4, 5, 6))

# RDD filter
val filtered = data.filter(_ % 2 == 0)

# map()
val mapped = filtered.map(_ * 2)

# Collect
val result = mapped.collect()
```



```
val sc = spark.sparkContext
val rdd1 = sc.parallelize(Seq(("a",55),("b",56),("c",57)))
val rdd2 = sc.parallelize(Seq(("a",60),("b",65),("d",61)))
val joinrdd = rdd1.cartesian(rdd2)
joinrdd.saveAsTextFile("/path/to/output")
```

