UNIVERSITÀ DEGLI STUDI DI PADOVA

J. Pazzini
PADOVA UNIVERSITY

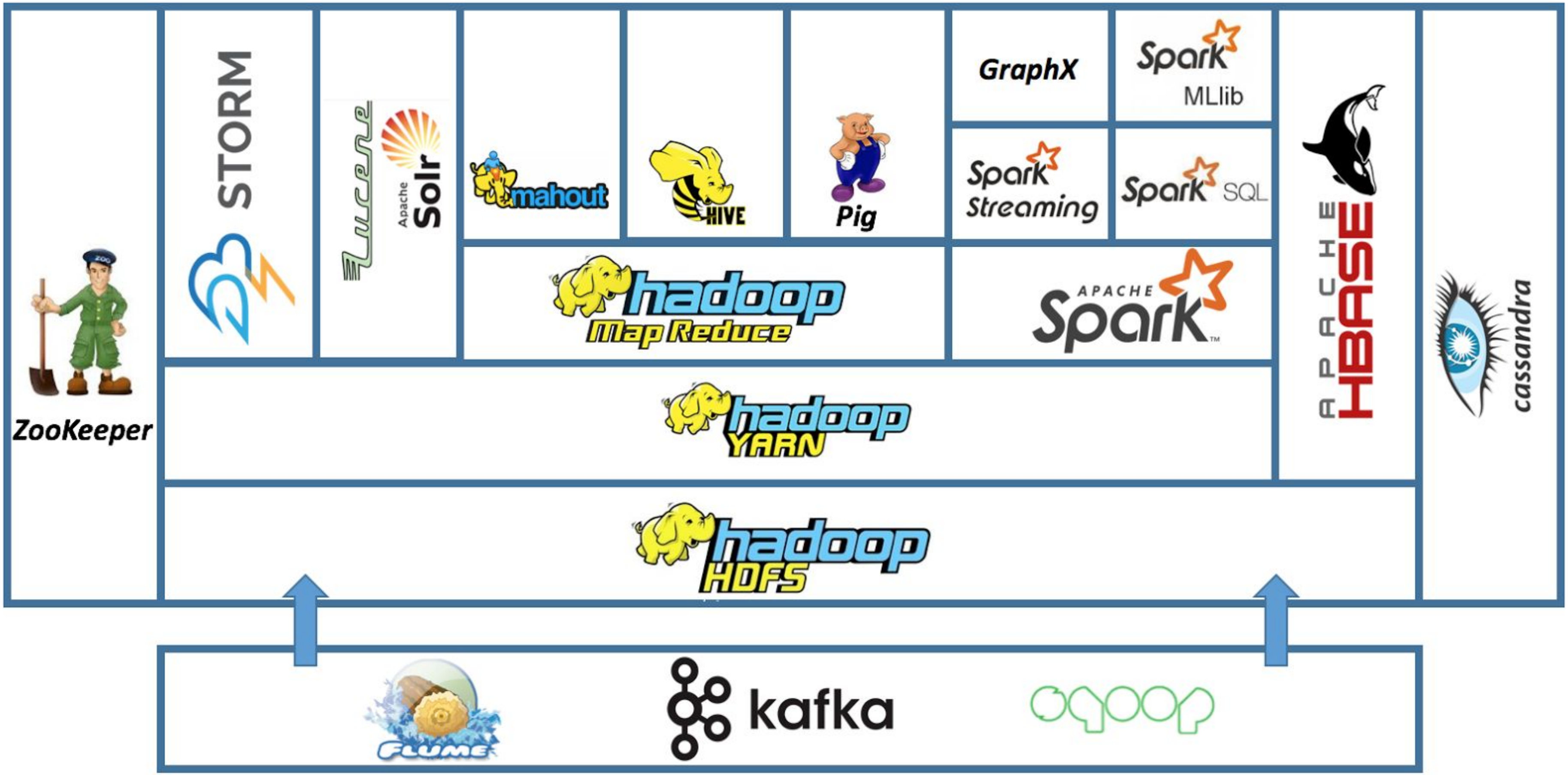
11 - SPARK

Management and Analysis of Physics Datasets - Module B

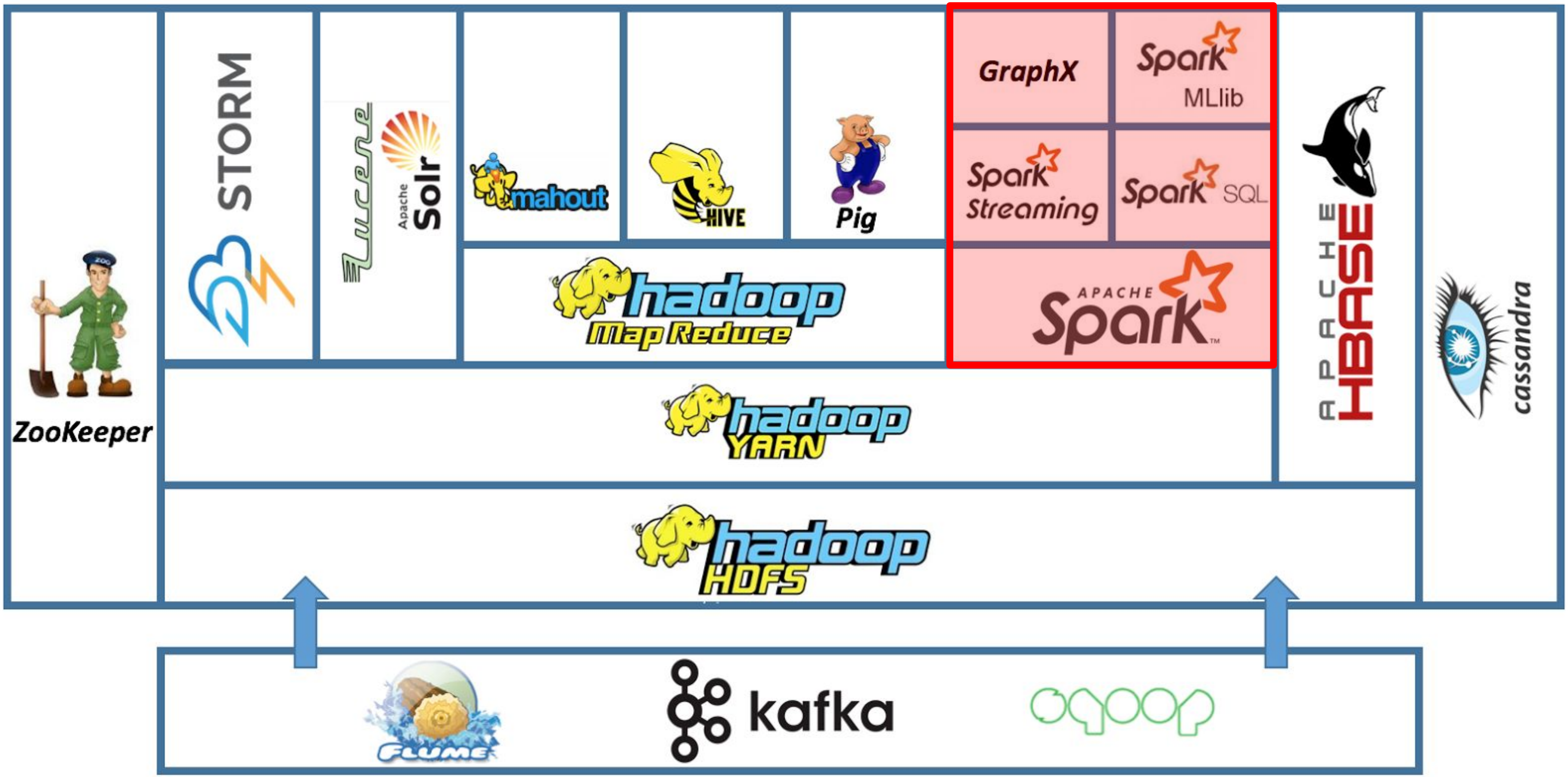
Physics of Data

A.A. 2023/2024

A BIGDATA ECOSYSTEM



A BIGDATA ECOSYSTEM



SPARK

Open source (Apache) cluster computing framework for data analytics, originally developed at UC Berkeley AMPLab in 2009

Difficult MapReduce programming of complex/iterative jobs (e.g. k-means), or requires to be split into several smaller MapReduce tasks

Processing is limited by the I/O operations of MapReduce functions to the disks

Suitable for batch-only processing (processing of data “at-rest”)

SPARK

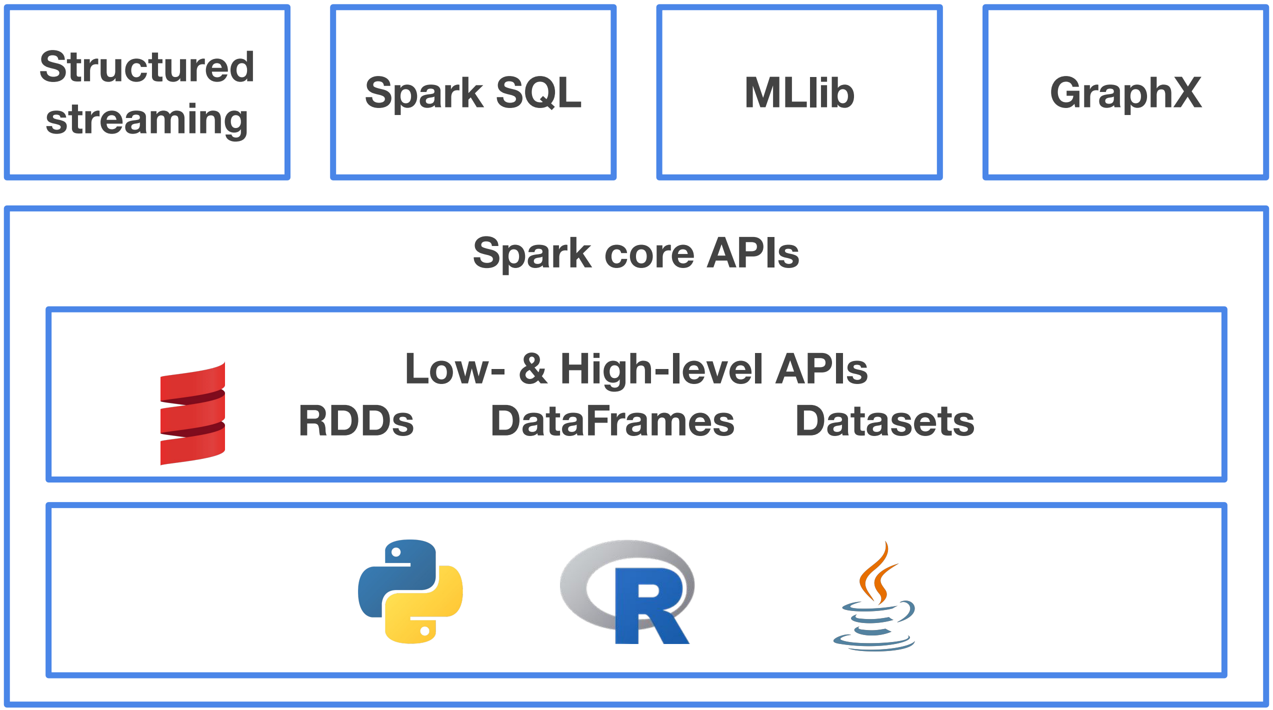
Open source (Apache) cluster computing framework for data analytics, originally developed at UC Berkeley AMPLab in 2009

Difficult MapReduce programming of complex/iterative jobs (e.g. k-means), or requires to be split into several smaller MapReduce tasks	→	Not (strictly/necessarily) relying on MapReduce
Processing is limited by the I/O operations of MapReduce functions to the disks	→	Supporting in-memory processing
Suitable for batch-only processing (processing of data “at-rest”)	→	Allowing both batch and real-time streaming processing

SPARK ECOSYSTEM

Spark is developed and written in Scala, while it includes also APIs for Java, Python and R languages (however, performances will be optimal only using Scala)

Spark provides a broad set of APIs for Analytics, Structured data analysis, Machine Learning, Graph computation, …

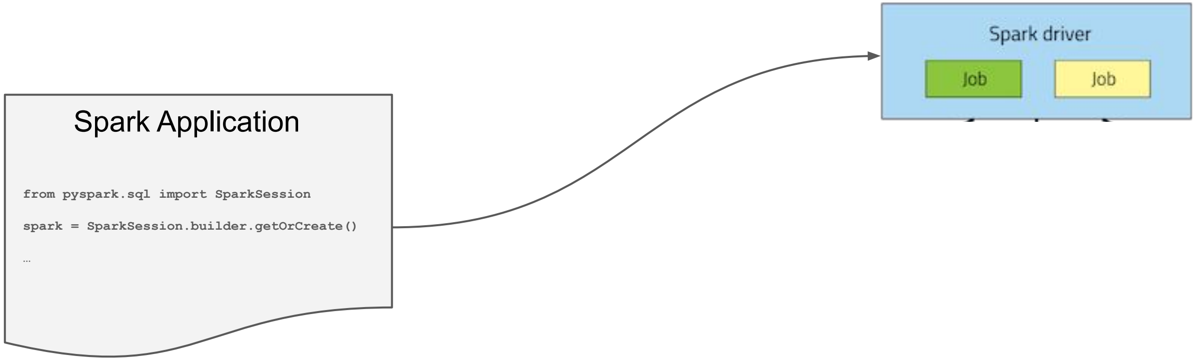


SPARK CLUSTER ARCHITECTURE

Spark deals with applications using a Master/Slave architecture

Once a **Spark Application** is submitted, a **Driver** (application's driver process) is created

→ The Driver process is responsible for converting a user application into smaller execution units (tasks), and for distributing and scheduling tasks across the executors



SPARK CLUSTER ARCHITECTURE

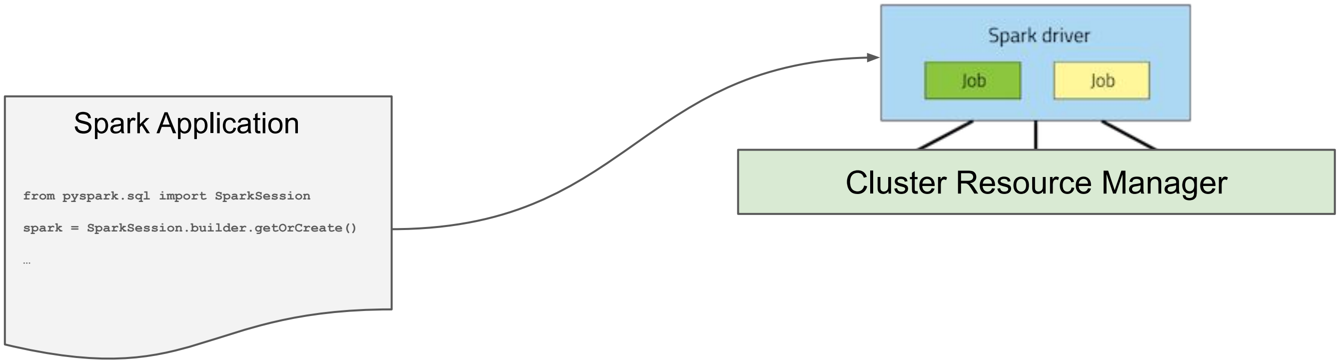
Spark deals with applications using a Master/Slave architecture

Once a **Spark Application** is submitted, a **Driver** (application's driver process) is created

→ The Driver process is responsible for converting a user application into smaller execution units (tasks), and for distributing and scheduling tasks across the executors

The Spark Driver creates a **Spark Context** for each individual Spark Application

→ Allows the Application to access the Cluster resources with the help of **Resource Manager**



Spark deals with applications using a Master/Slave architecture

Once a **Spark Application** is submitted, a **Driver** (application's driver process) is created

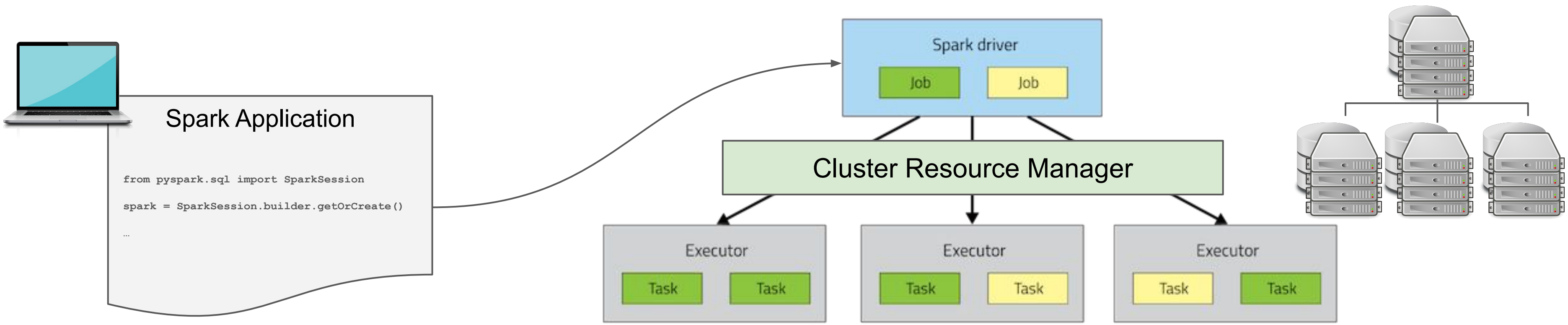
→ The Driver process is responsible for converting a user application into smaller execution units (tasks), and for distributing and scheduling tasks across the executors

The Spark Driver creates a **Spark Context** for each individual Spark Application

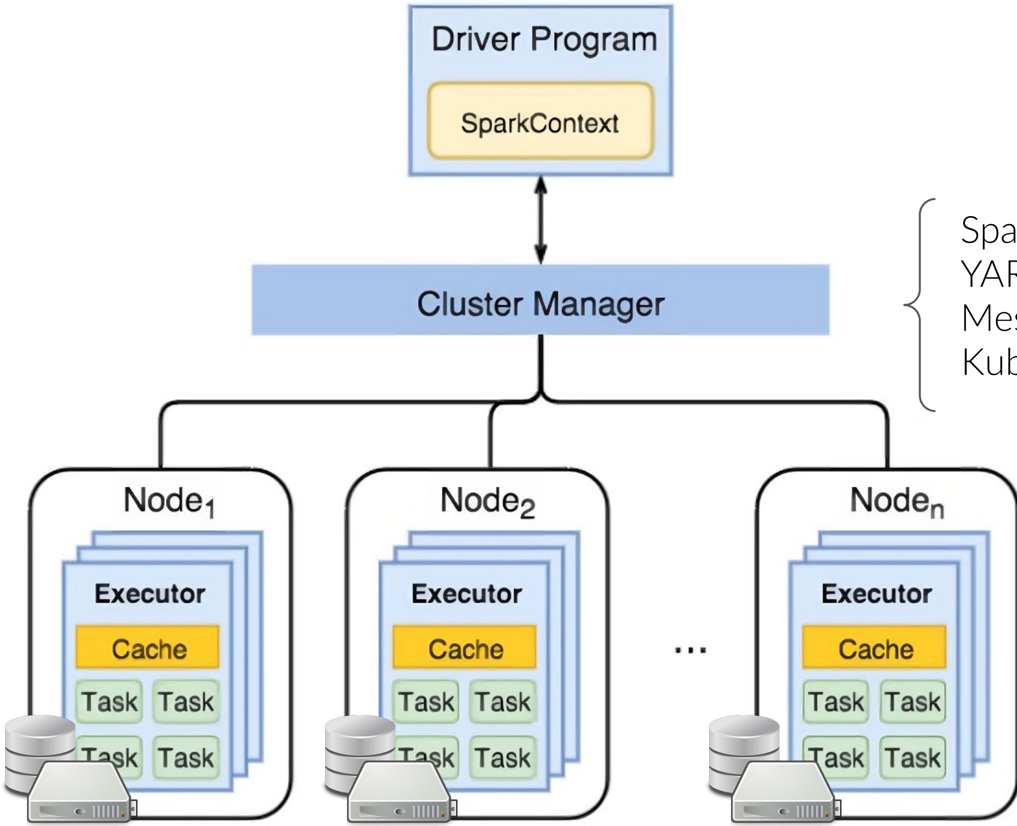
→ Allows the Application to access the Cluster resources with the help of **Resource Manager**

Together with the Driver, a set of **Executors** are created on the clusters' nodes:

→ Each Executor is responsible for executing the tasks assigned by the Driver, and for reporting the state of the computation back to the driver node



SPARK CLUSTER ARCHITECTURE

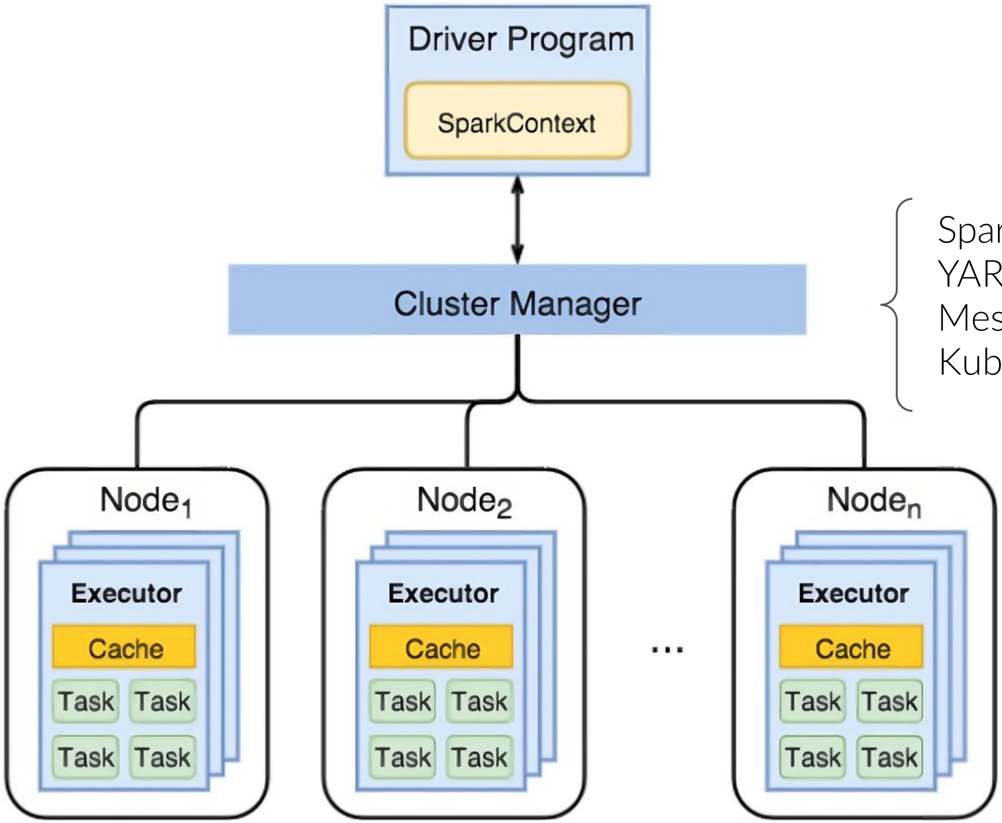


Spark standalone cluster manager
YARN
Mesos
Kubernetes

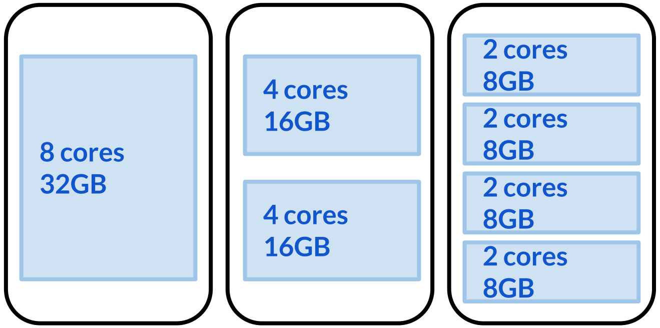
The Driver and Executors live in Java Virtual Machines (JVMs) within the **worker nodes**

- 1 executor cannot span multiple nodes although one node may contain several executors

SPARK CLUSTER ARCHITECTURE



Spark standalone cluster manager
YARN
Mesos
Kubernetes



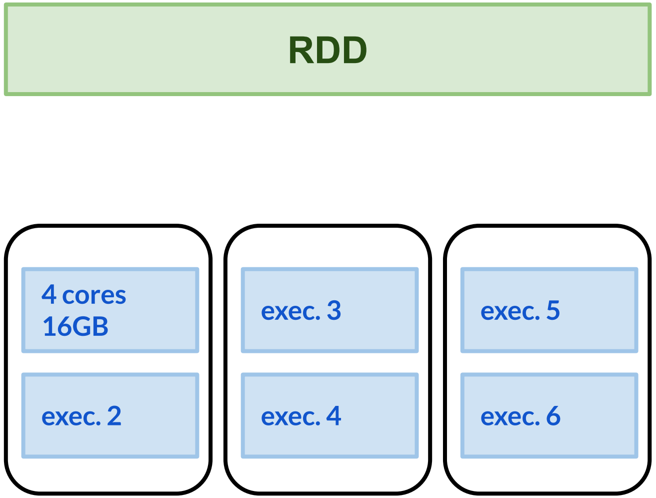
The Driver and Executors live in Java Virtual Machines (JVMs) within the **worker nodes**

- 1 executor cannot span multiple nodes although one node may contain several executors

RDD

Resilient Distributed Dataset → Spark low-level data abstraction that describes an immutable, partitioned collection of records that can be operated on in parallel

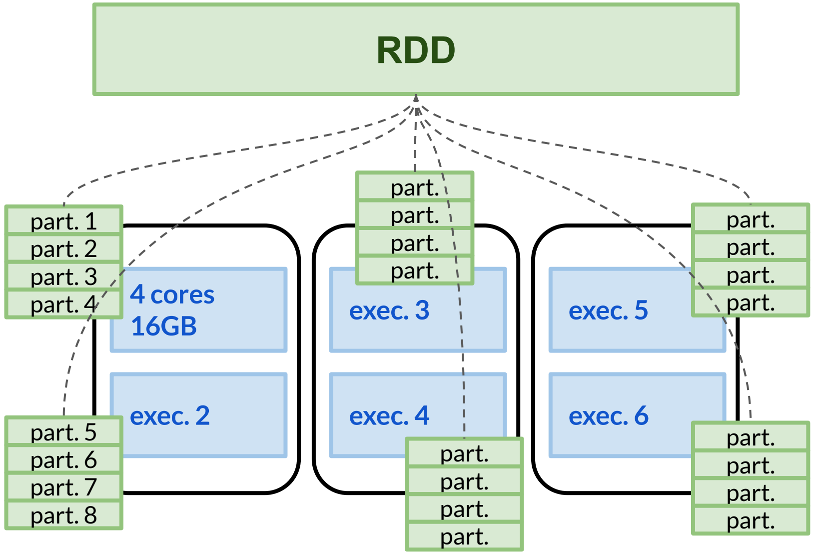
- The data is stored as a **collection of Java/Python objects (unstructured)**
- Split into **partitions, hosted within the executors (→ data locality)**
- Spark can keep an **RDD loaded in-memory** throughout the life of an application
- RDDs are **immutable**: no changes are applied to the RDD across the application
- **Fault-tolerant**: the RDD contains all the dependencies to recover from a partition loss



RDD

Resilient Distributed Dataset → Spark low-level data abstraction that describes an immutable, partitioned collection of records that can be operated on in parallel

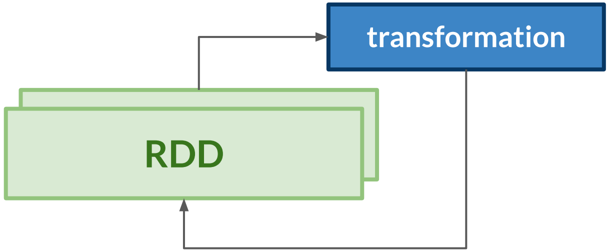
- The data is stored as a **collection of Java/Python objects (unstructured)**
- Split into **partitions, hosted within the executors (→ data locality)**
- Spark can keep an **RDD loaded in-memory** throughout the life of an application
- RDDs are **immutable**: no changes are applied to the RDD across the application
- **Fault-tolerant**: the RDD contains all the dependencies to recover from a partition loss

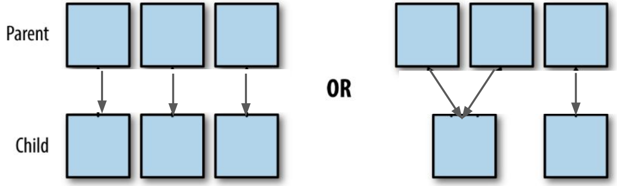


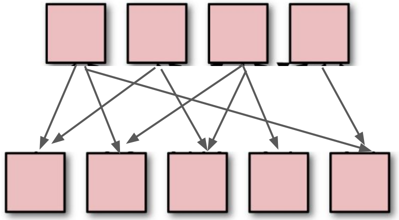
TRANSFORMATIONS & ACTIONS

TRANSFORMATIONS ⇒ *operations that act on the RDD and produce a “new” RDD*

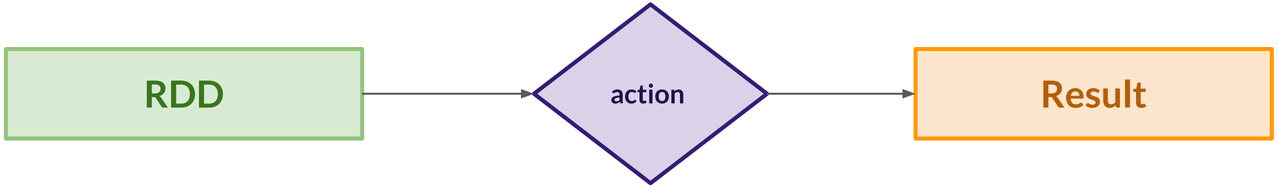
- **narrow** dependencies → each input partition will contribute to only one output partition
- **wide** dependencies → input partitions contributing to many output partitions (shuffling)







ACTIONS ⇒ *operations that return a value as the result of a computation on an RDD*



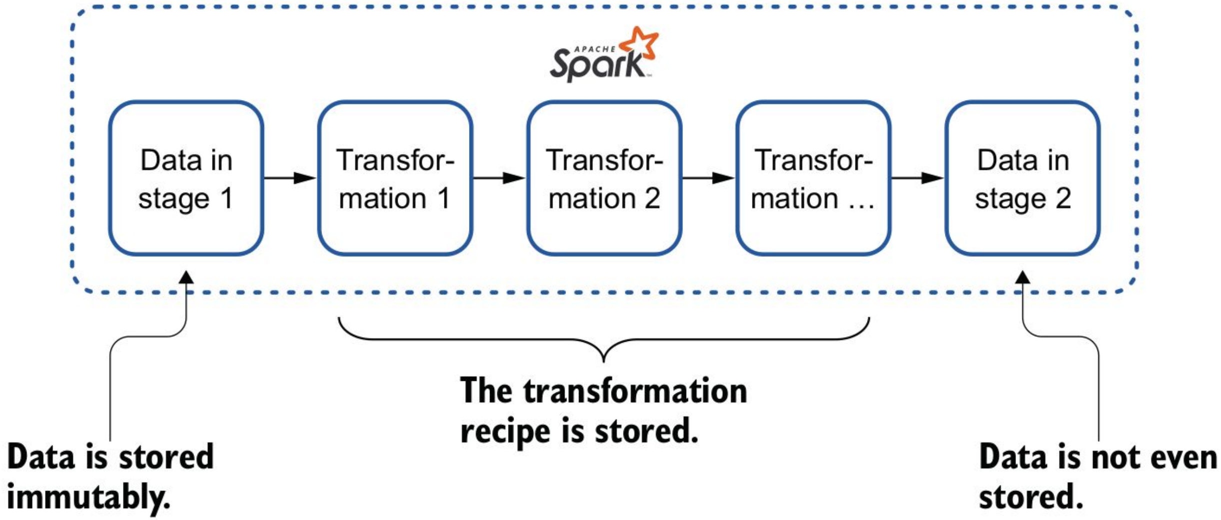
LAZINESS

Transformations are **LAZY** → a transformation execution will not start until an action is triggered

- Instead of applying the transformation to all the data stored in all partitions at the same time Spark keeps the data in sync over the nodes and share only the transformation "recipe"

- This minimizes the driver-executors calls

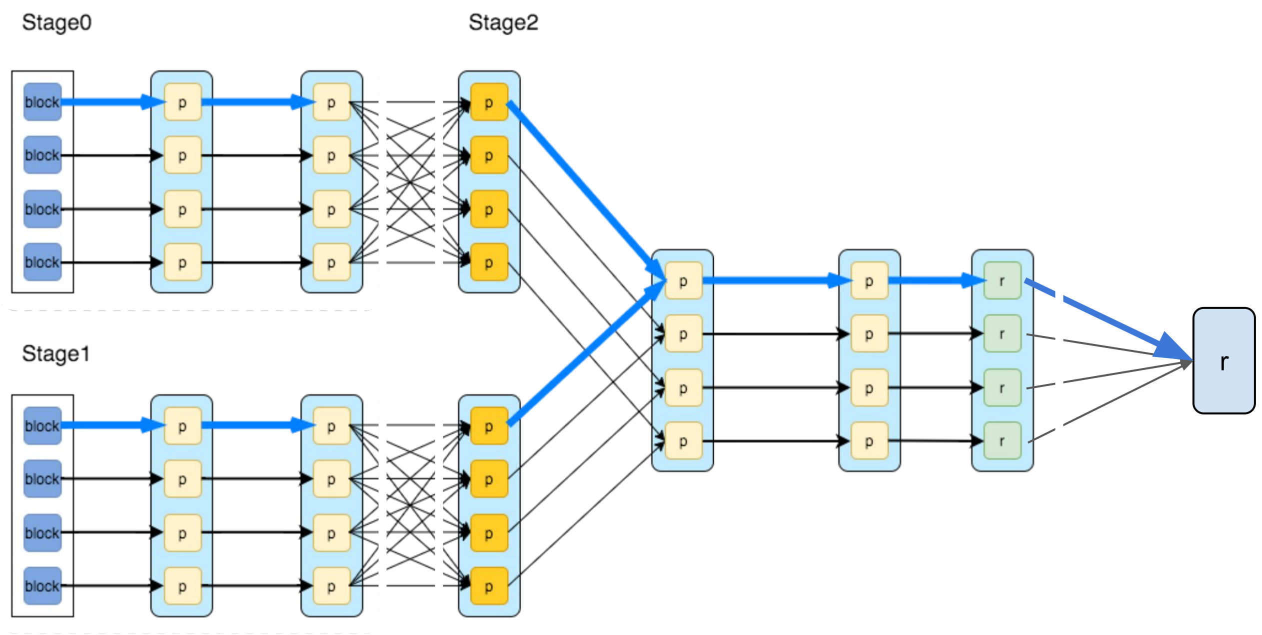
- And allows the optimization of the tasks dispatched to the executors prior its execution



DAG SCHEDULER

Spark optimizes the execution of all operations within a job by representing them using a **DAG (Directed Acyclic Graph)**

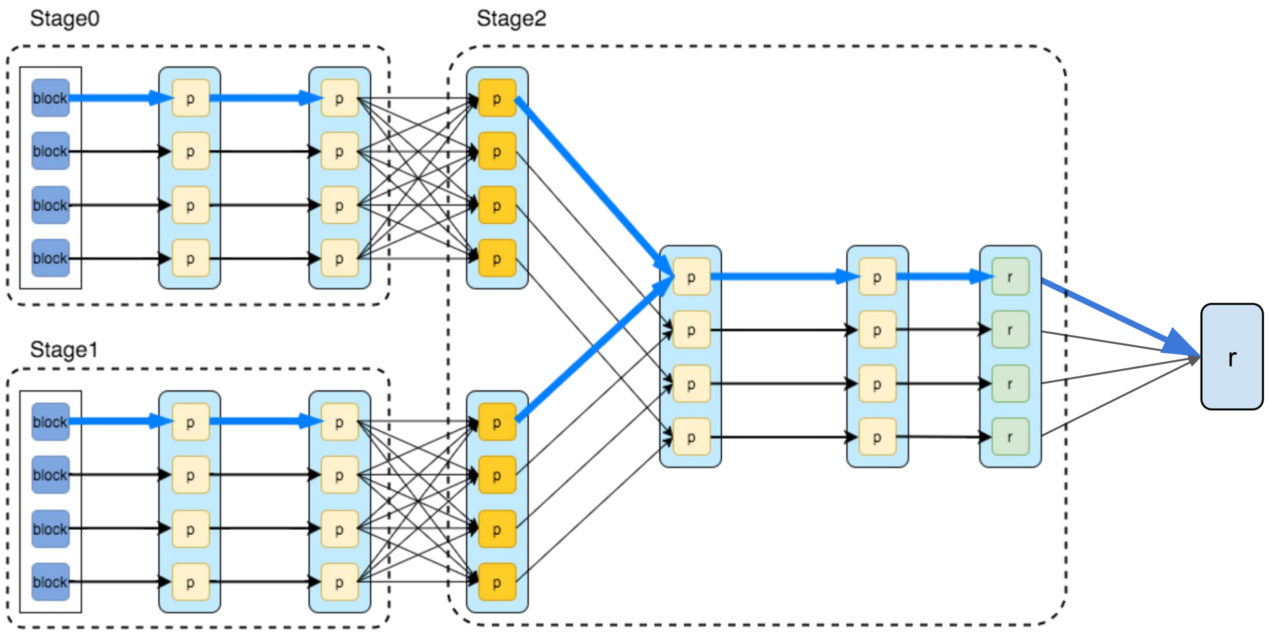
- The Spark **DAG scheduler organizes tasks** that can be performed without exchanging data across partitions (i.e. narrow transformations) **into stages**
- Each **operation within a stage is pipelined**
- All tasks are then dispatched by the driver to the executors

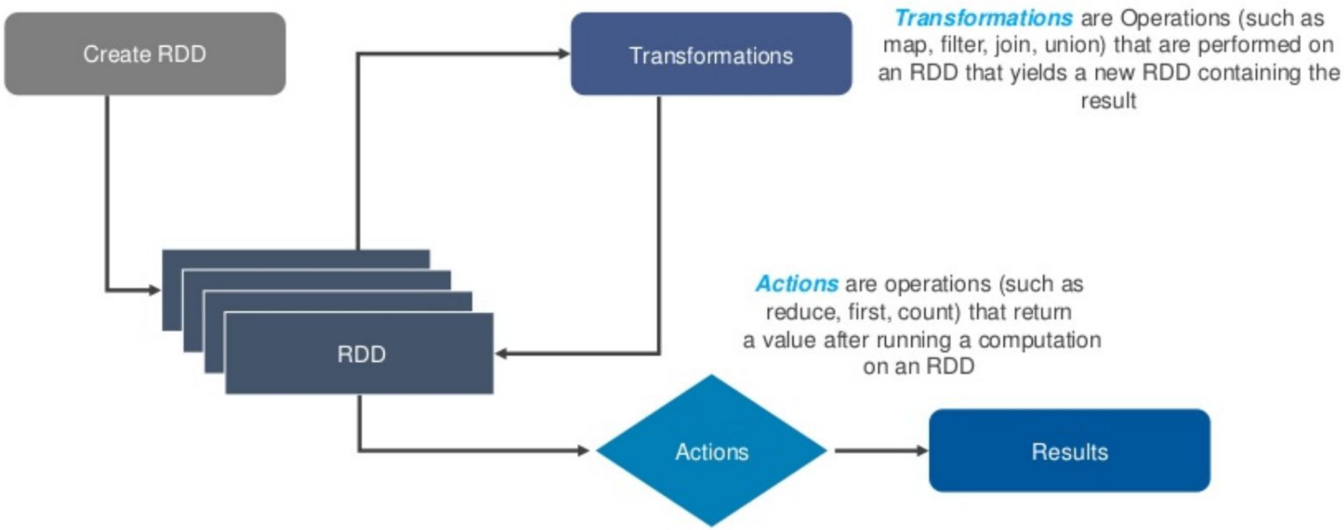


DAG SCHEDULER

Spark optimizes the execution of all operations within a job by representing them using a **DAG (Directed Acyclic Graph)**

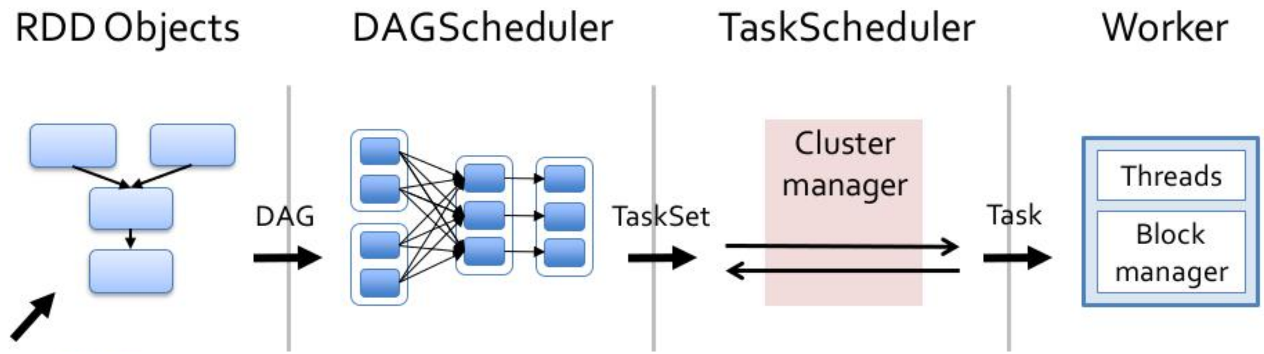
- The Spark **DAG scheduler organizes tasks** that can be performed without exchanging data across partitions (i.e. narrow transformations) **into stages**
- Each **operation within a stage is pipelined**
- All tasks are then dispatched by the driver to the executors

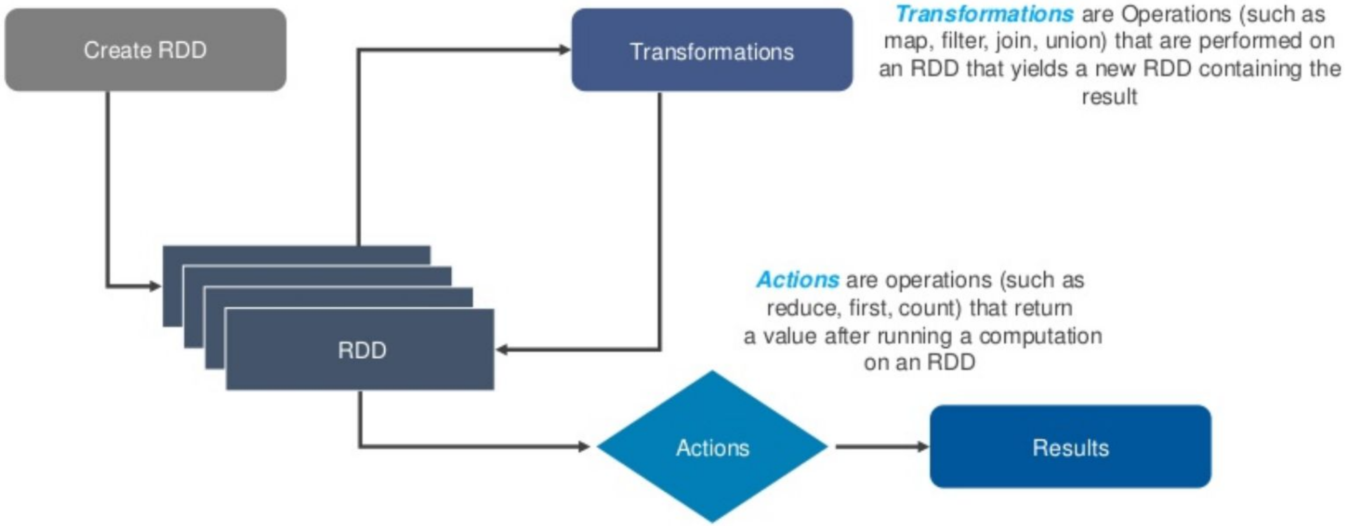




Logical flow

Implementation, Scheduling, and Execution





Logical flow

DATAFRAME

The RDD is intended as a low-level API, mostly suited for *semi-structured and unstructured datasets*

The high-level pySpark API for structured datasets is the SparkSQL **DataFrame**

- **Similar features to the DataFrame in Pandas**
- **Offers a SQL interface for queries**
- **Distributed, and resilient in nature, as RDDs**

```
>>> rdd.take(10)
['model,mpg,cyl,disp,hp,drat,wt,qsec,vs,am,gear,carb',
'Mazda RX4,21,6,160,110,3.9,2.62,16.46,0,1,4,4', 'Mazda RX4
Wag,21,6,160,110,3.9,2.875,17.02,0,1,4,4', 'Datsun
710,22.8,4,108,93,3.85,2.32,18.61,1,1,4,1', 'Hornet 4
Drive,21.4,6,258,110,3.08,3.215,19.44,1,0,3,1', 'Hornet
Sportabout,18.7,8,360,175,3.15,3.44,17.02,0,0,3,2',
'Valiant,18.1,6,225,105,2.76,3.46,20.22,1,0,3,1', 'Duster
360,14.3,8,360,245,3.21,3.57,15.84,0,0,3,4', 'Merc
240D,24.4,4,146.7,62,3.69,3.19,20,1,0,4,2', 'Merc
230,22.8,4,140.8,95,3.92,3.15,22.9,1,0,4,2']
```

RDD

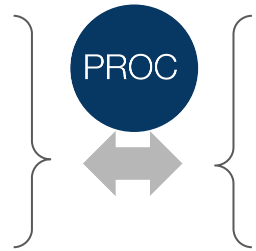
```
>>> dataframe.show(10)
+-----------------+----+---+-----+---+----+-----+-----+---+---+----+----+
|            model| mpg|cyl| disp| hp|drat|   wt| qsec| vs| am|gear|carb|
+-----------------+----+---+-----+---+----+-----+-----+---+---+----+----+
|        Mazda RX4|21.0|  6|160.0|110| 3.9| 2.62|16.46|  0|  1|   4|   4|
|    Mazda RX4 Wag|21.0|  6|160.0|110| 3.9|2.875|17.02|  0|  1|   4|   4|
|       Datsun 710|22.8|  4|108.0| 93|3.85| 2.32|18.61|  1|  1|   4|   1|
|   Hornet 4 Drive|21.4|  6|258.0|110|3.08|3.215|19.44|  1|  0|   3|   1|
|Hornet Sportabout|18.7|  8|360.0|175|3.15| 3.44|17.02|  0|  0|   3|   2|
|          Valiant|18.1|  6|225.0|105|2.76| 3.46|20.22|  1|  0|   3|   1|
|       Duster 360|14.3|  8|360.0|245|3.21| 3.57|15.84|  0|  0|   3|   4|
|        Merc 240D|24.4|  4|146.7| 62|3.69| 3.19| 20.0|  1|  0|   4|   2|
|         Merc 230|22.8|  4|140.8| 95|3.92| 3.15| 22.9|  1|  0|   4|   2|
|         Merc 280|19.2|  6|167.6|123|3.92| 3.44| 18.3|  1|  0|   4|   4|
+-----------------+----+---+-----+---+----+-----+-----+---+---+----+----+
```

DataFrame

REAL TIME ANALYTICS

Real time processing of unbounded data flows is a common use-case in a variety of applications

- Sensor readings from Physics detectors
- Readings from IoT devices
- Monitoring of economic transactions
- User-data from website / webapp



- Live data monitoring
- Rolling / windowed aggregation of data
- Triggering of alerts / fault detection
- Real-time ML

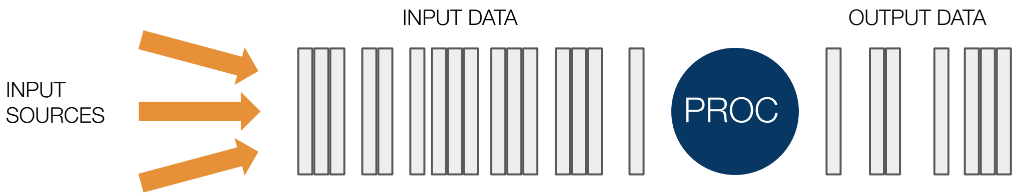
Both the **data input rate** and the **max affordable time for data processing** might severely differ depending on the use case

The complexities related to the distributed processing of streaming data are plenty:

- *Potentially out-of-order data*
- *(High) Data throughput*
- *Reliability issues (each event must be processed exactly once)*
- *Handling of load imbalance*
- …

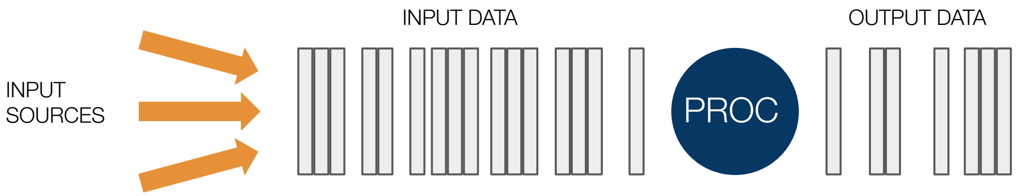
One-Element-at-a-Time

- a (fixed) pipeline of computations is implemented
- the system is continually listening the input sources
- a compute is triggered on any new record



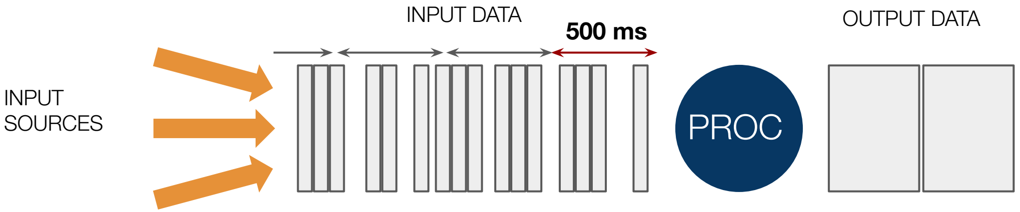
One-Element-at-a-Time

- a (fixed) pipeline of computations is implemented
- the system is continually listening the input sources
- a compute is triggered on any new record



Micro-batching

- accumulate small batches of input data depending on a fixed "wall-time" (e.g. every 500 ms)
- then process each batch independently using a distributed collection of tasks

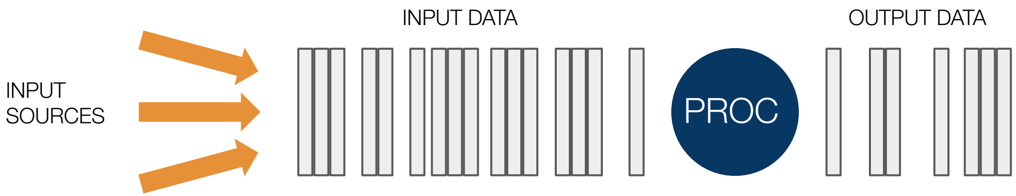


REAL TIME ANALYTICS

One-Element-at-a-Time

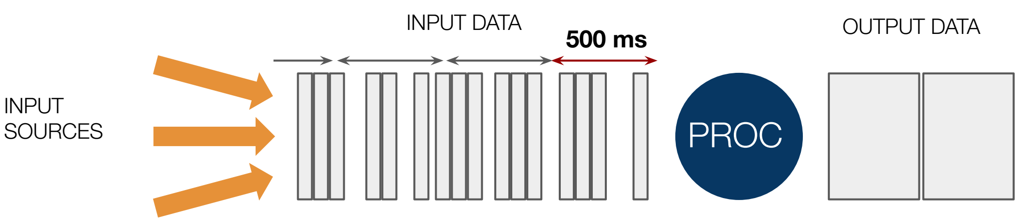
- a (fixed) pipeline of computations is implemented
- the system is continually listening the input sources
- a compute is triggered on any new record

▲ Low latency
▼ Limited maximum throughput
▼ Possible load balancing issues



Micro-batching

- accumulate small batches of input data depending on a fixed “wall-time” (e.g. every 500 ms)
- then process each batch independently using a distributed collection of tasks



▼ Latency limited by wall-time
▲ Higher maximum throughput
▲ Dynamic workload (load balancing)

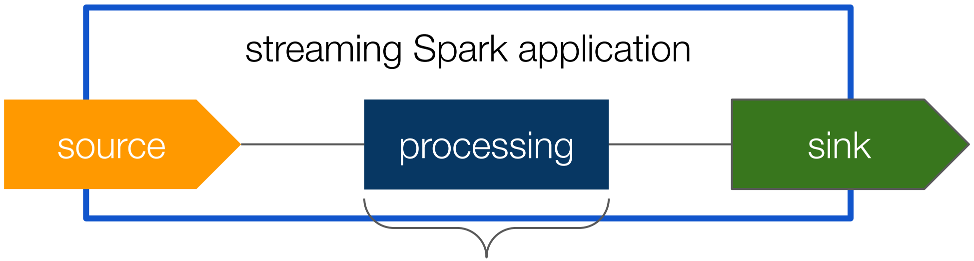
REAL TIME ANALYTICS IN SPARK

Processing of streaming data in Spark can be performed by means of low- and high-level APIs

- Spark Streaming (now phased out…)
- **Spark Structured Streaming**

Spark represents the input and output of the streaming data processing using the abstractions:

- **Source** → interface for connecting to streaming systems such as Kafka, Flume, Twitter, a TCP socket, …
- **Sink** → abstraction used to write the processes data stream outside Spark, such as to Kafka, files, DB, …



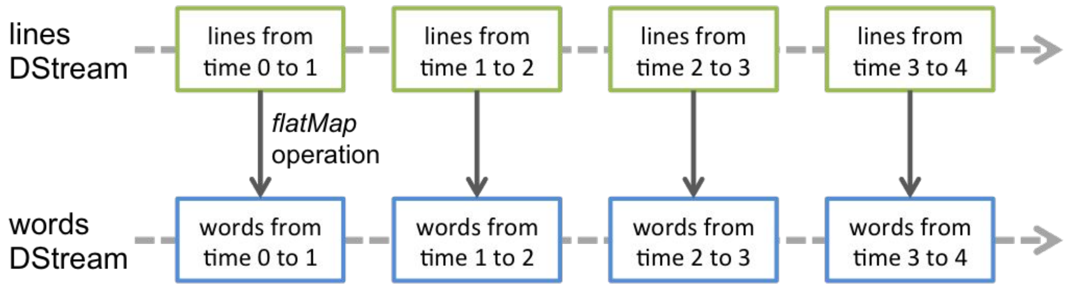
Same application as in the case of non-streaming

Spark Streaming

- Primarily intended for the low-level RDD, and now (from Spark 3.5) not supported anymore
- It provides the **DStream API** → streaming data gets divided into "chunks" as RDDs

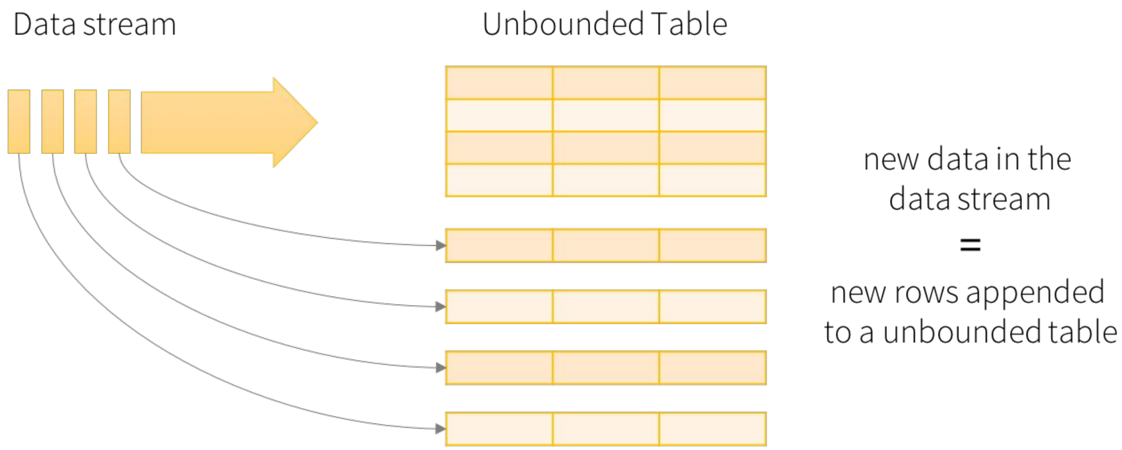
Spark Streaming API provides stream processing via **micro-batches** of a given size/duration

- input data is discretized into a DStream
- A DStream element can be created from input data or by applying operations on other DStreams
- *DStreams can be represented almost like as sequences of RDDs*



Spark Structured Streaming

- Primarily intended for the high-level structured DataFrame
- Embeds the streaming functionalities in the Spark SQL API



Data stream as an unbounded table

Spark Structured Streaming API blurs the line between micro-batching / OEaaT processing

- input data is assumed to have a defined schema
- new records can be seen as “rows”, and appended to an “unbounded table”
- all computation are seen as queries over a continuously updating table