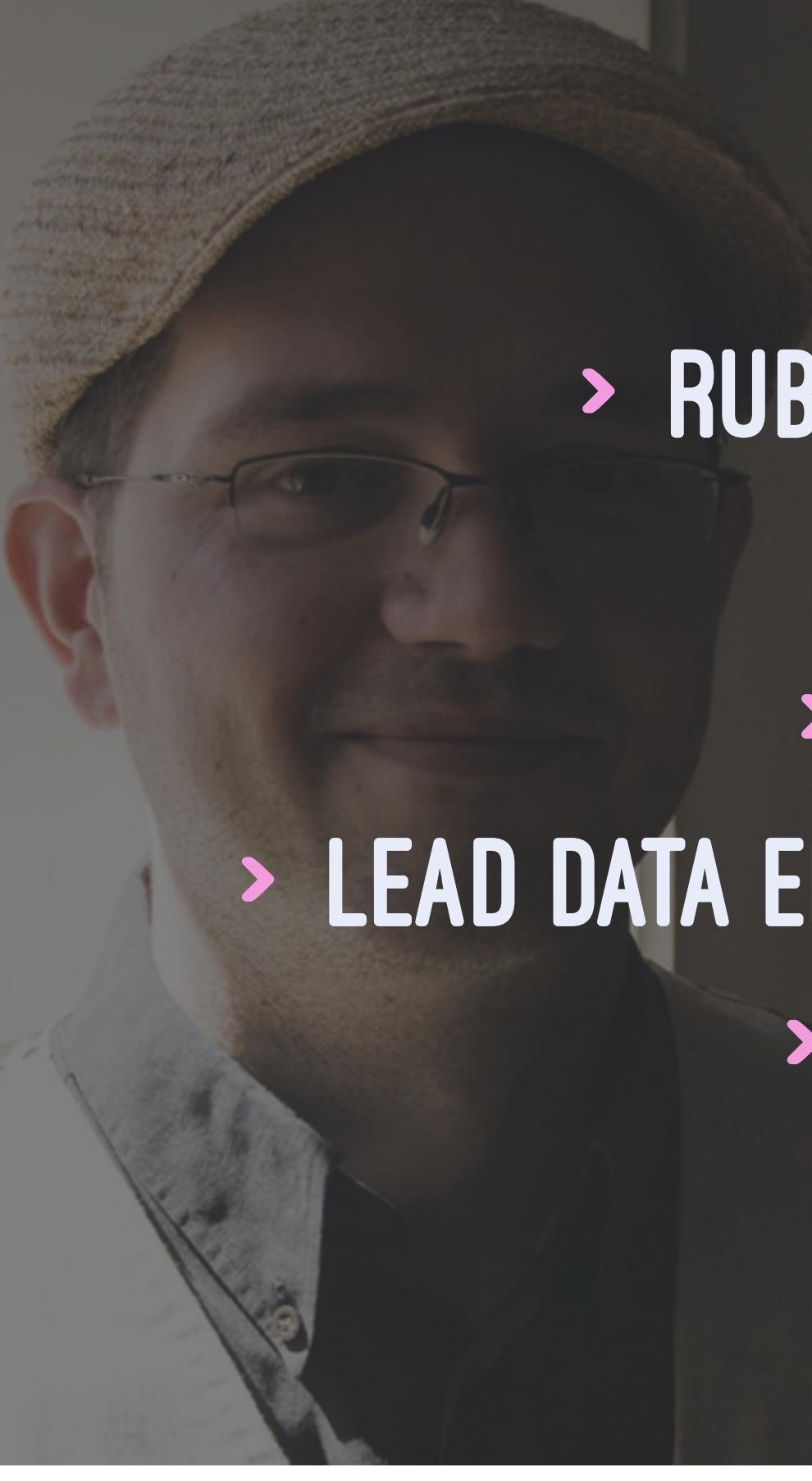


# SPEEDING UP PYSPARK WITH ARROW





# WHOAMI

- > RUBEN BERENGUEL (@BERENGUEL)
- > PHD IN MATHEMATICS
- > (BIG) DATA CONSULTANT
- > LEAD DATA ENGINEER USING PYTHON, GO AND SCALA
- > RIGHT NOW AT AFFECTV

# WHAT IS PANDAS?

## WHAT IS PANDAS?

> PYTHON DATA ANALYSIS LIBRARY

## WHAT IS PANDAS?

- › PYTHON DATA ANALYSIS LIBRARY
- › USED EVERYWHERE DATA AND PYTHON APPEAR IN JOB OFFERS

## WHAT IS PANDAS?

- > PYTHON DATA ANALYSIS LIBRARY
- > USED EVERYWHERE DATA AND PYTHON APPEAR IN JOB OFFERS
- > EFFICIENT (IS COLUMNAR AND HAS A C AND CYTHON BACKEND)

Have you ever asked yourself  
why "Columnar database" is  
so trendy?

SUM

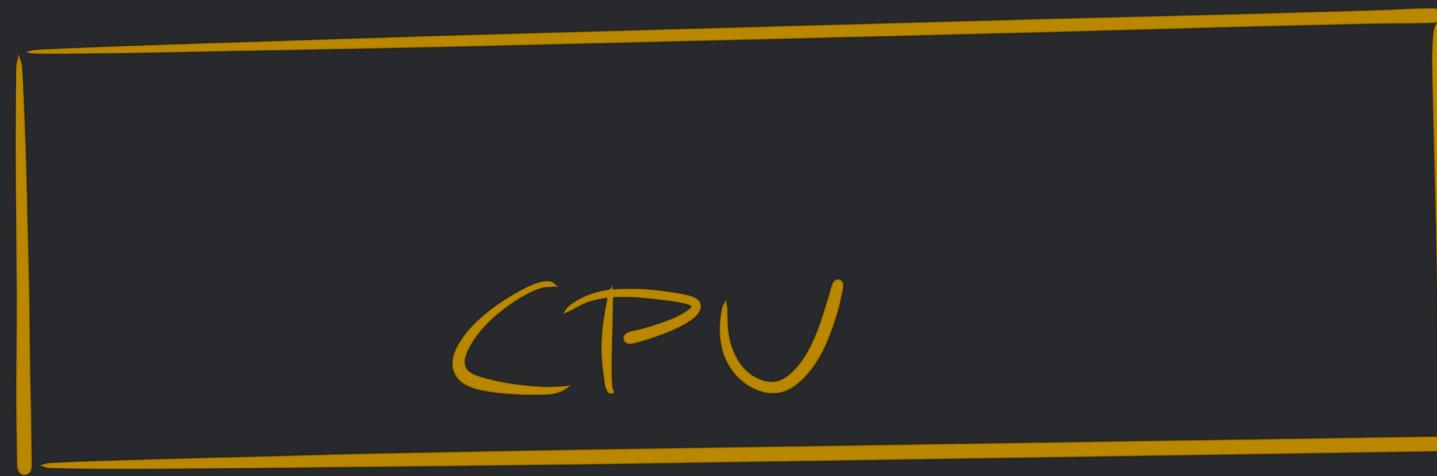


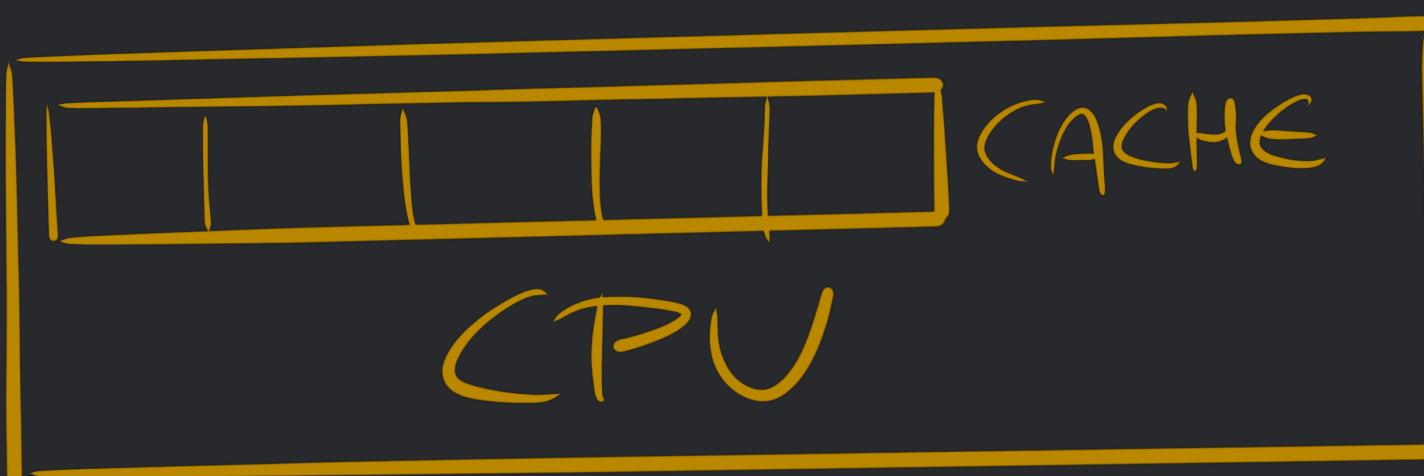
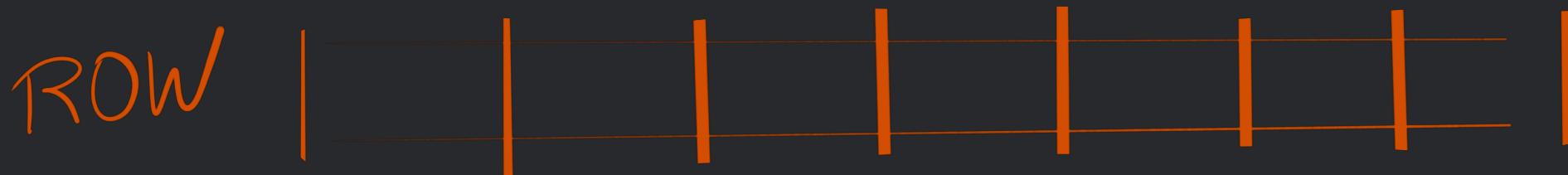
$c_1$	$c_2$	$c_3$	$\dots$	$c_{42}$	$c_{43}$	$\dots$
•						
•						
•						

ROW



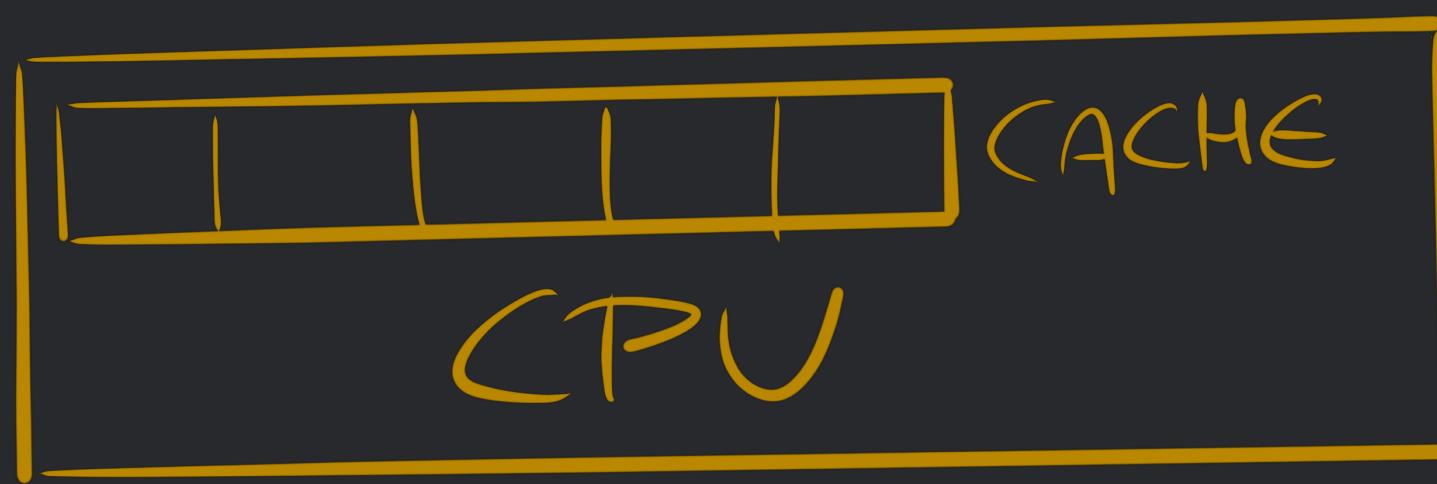
CPU

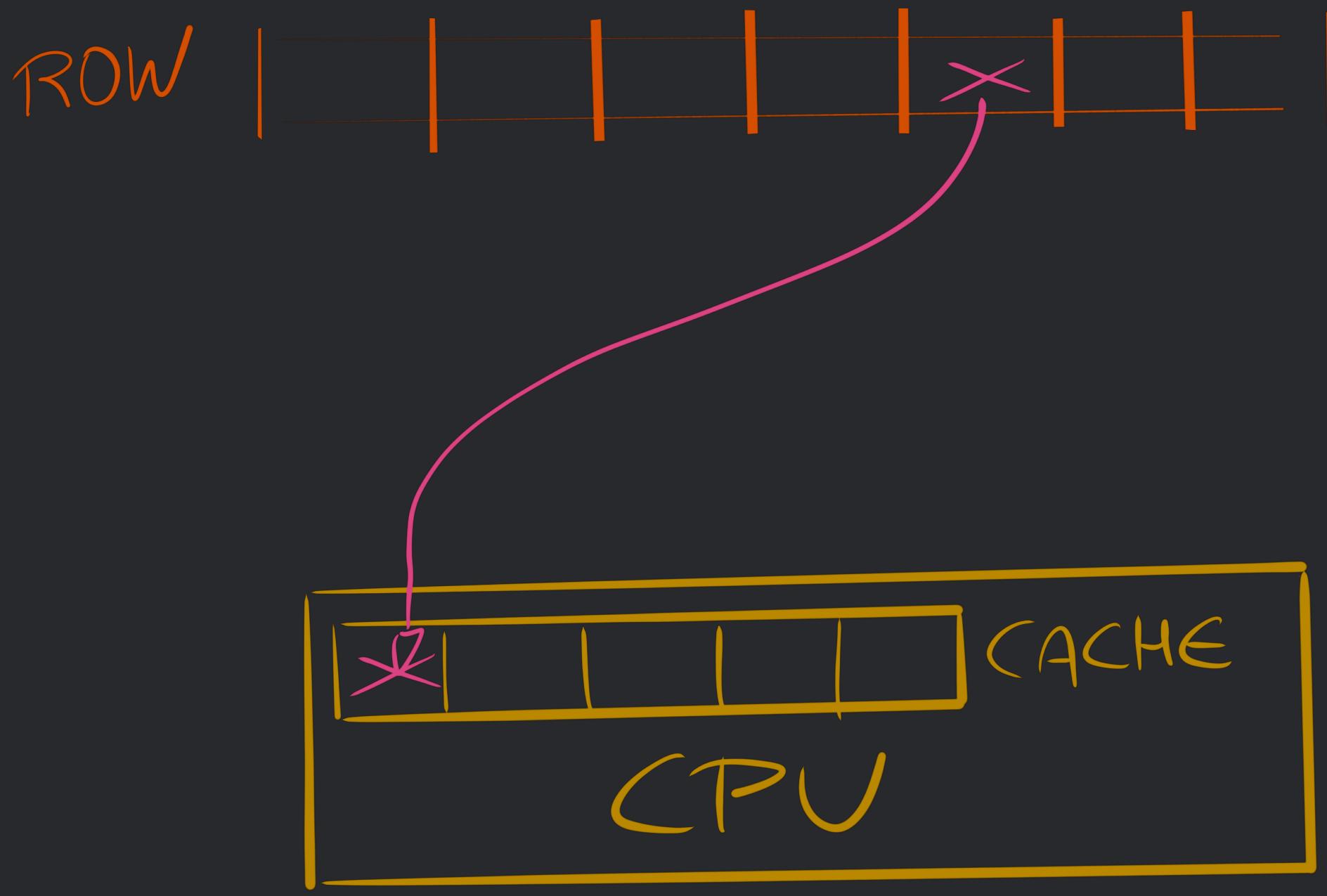




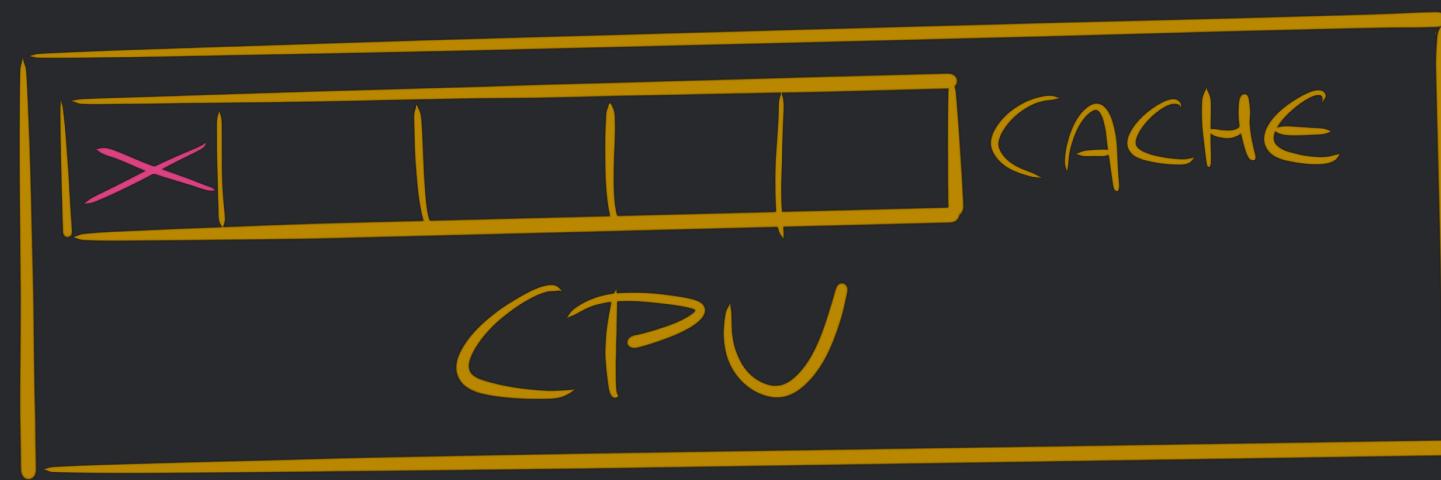
Moving data from memory to a cache line can be done in batches. If data is stored by column, we can load the whole column and operate straight with it with less memory fetching

ROW

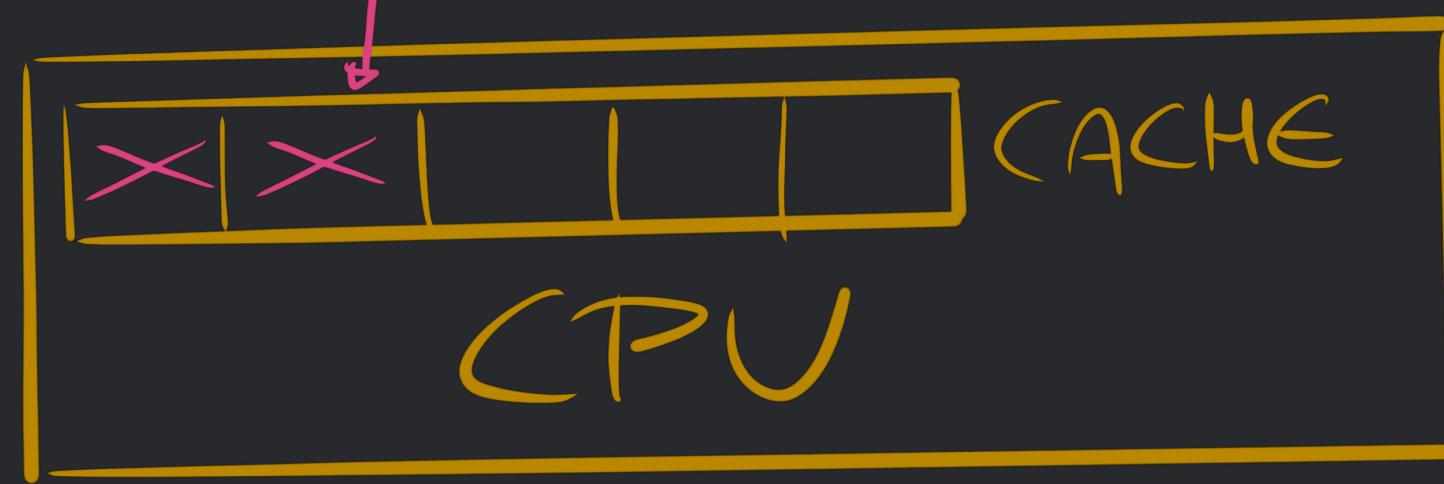


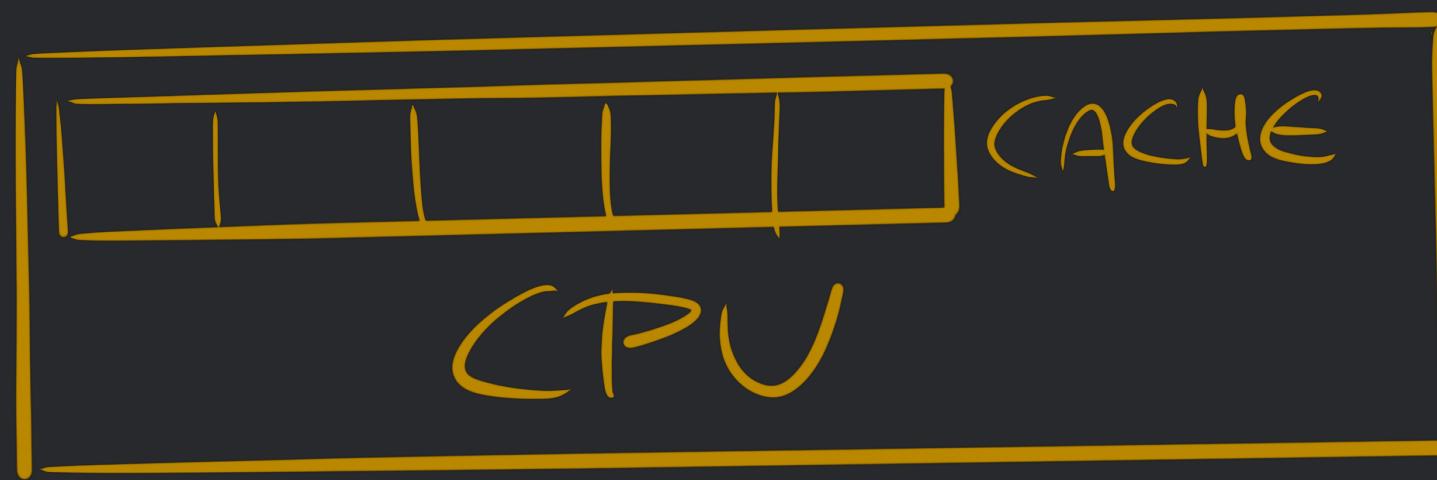


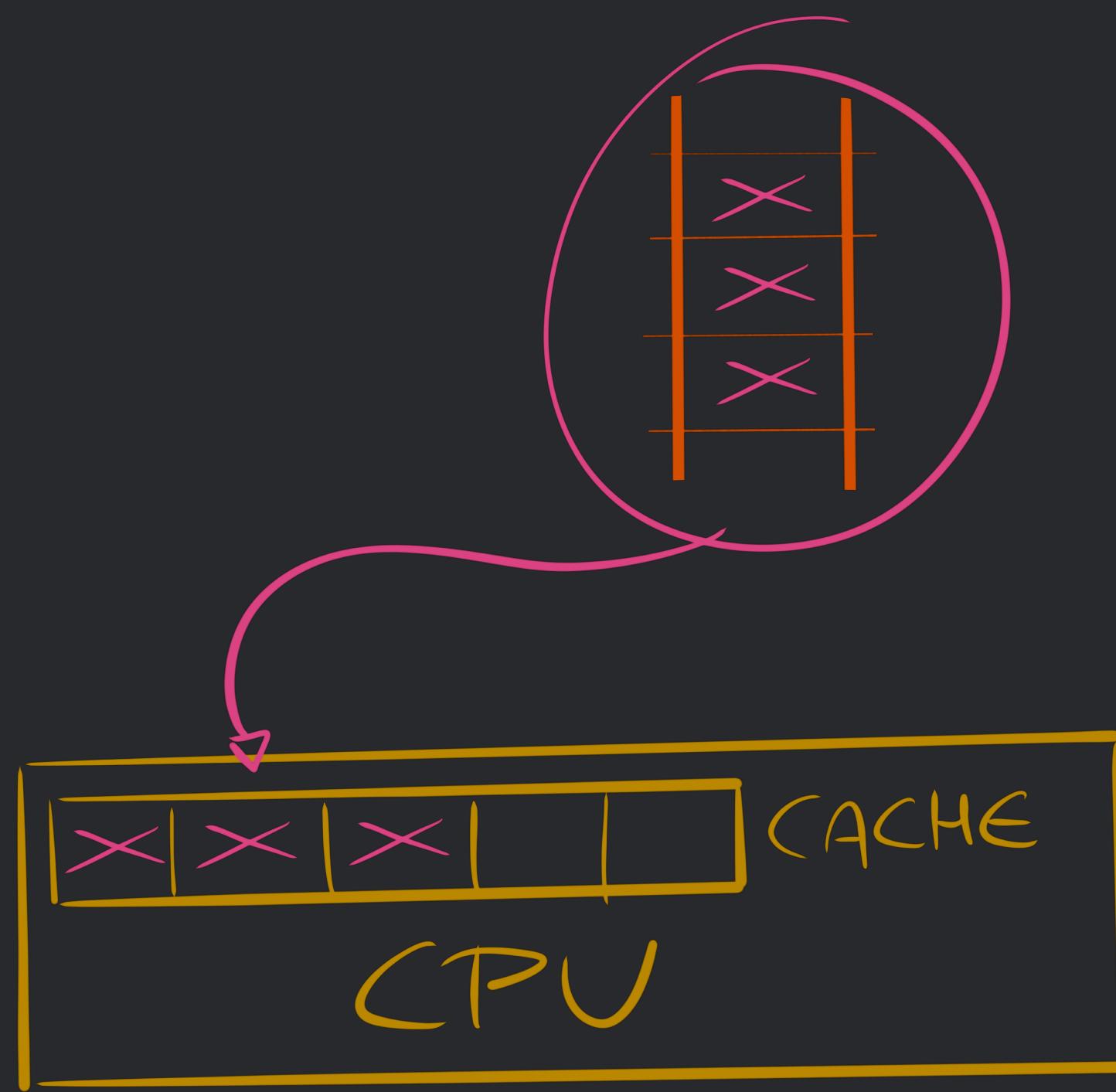
ROW



ROW







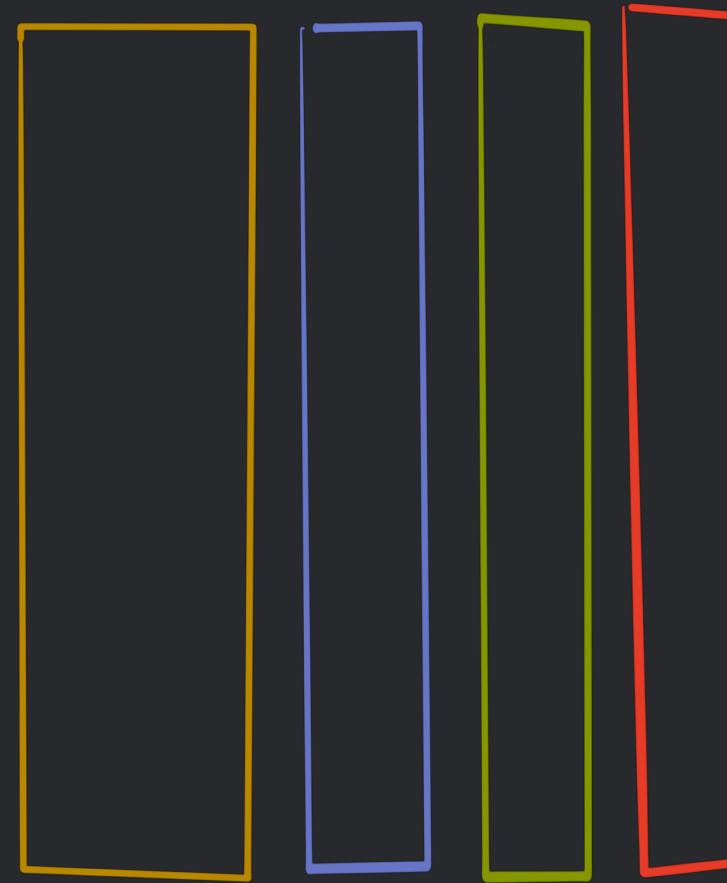
# HOW DOES PANDAS MANAGE COLUMNAR DATA?

# Pandas DataFrame

	A	B	C	D
1	4.4	a	2	0.1
2	3.1	b	42	0.4
3	2	c	8	0.9
4	8.3	d	15	0.3
:				

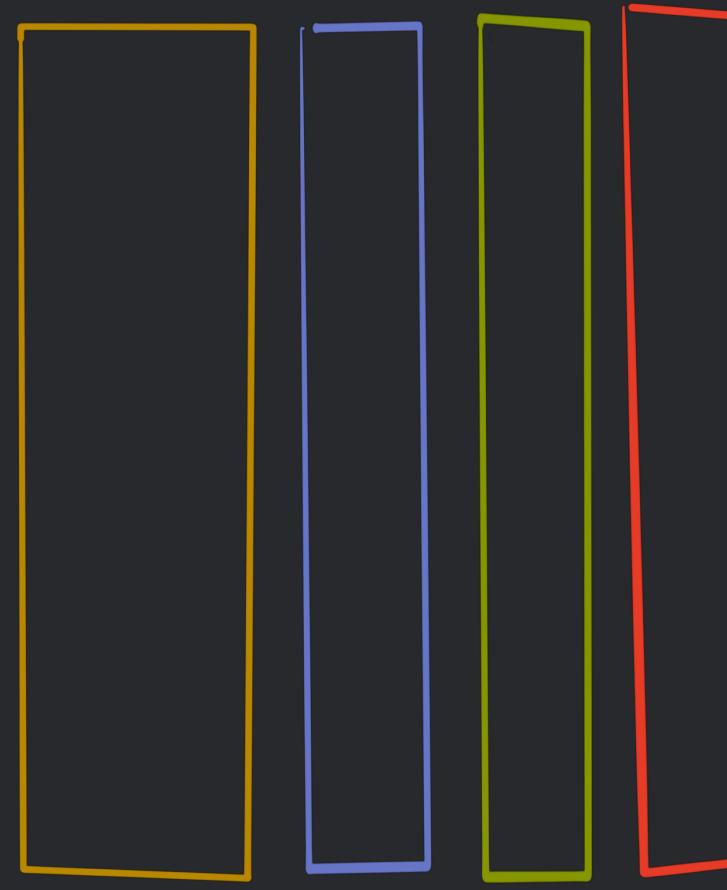
# Pandas DataFrame

	A	B	C	D
1	4.4	a	2	0.1
2	3.1	b	42	0.4
3	2	c	8	0.9
4	8.3	d	15	0.3
:				



# Pandas DataFrame

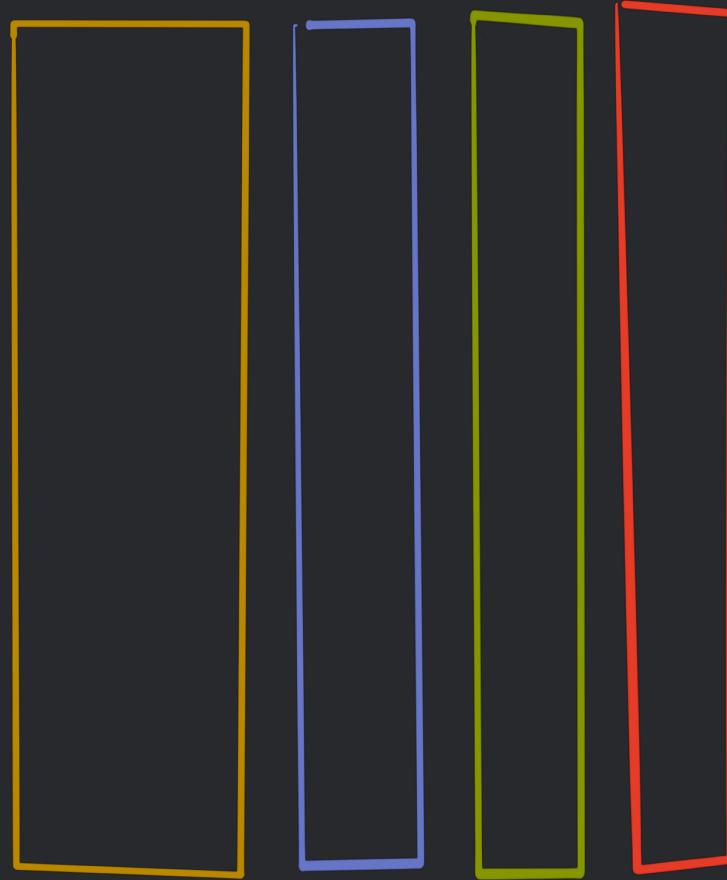
	A	B	C	D	
1	4.4	a	2	0.1	
2	3.1	b	42	0.4	
3	2	c	8	0.9	
4	8.3	d	15	0.3	
:					



INDEX  
SECTION

Pandas DataFrame

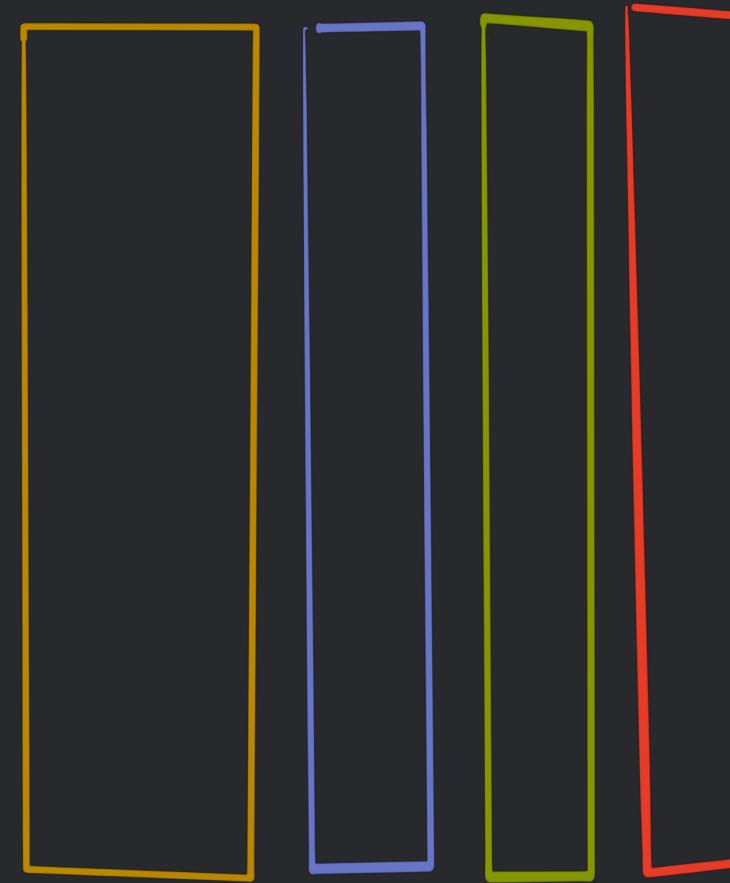
	A	B	C	D
1	4.4	a	2	0.1
2	3.1	b	42	0.4
3	2	c	8	0.9
4	8.3	d	15	0.3
:				



$2 \times N$  FloatBlock

Pandas DataFrame

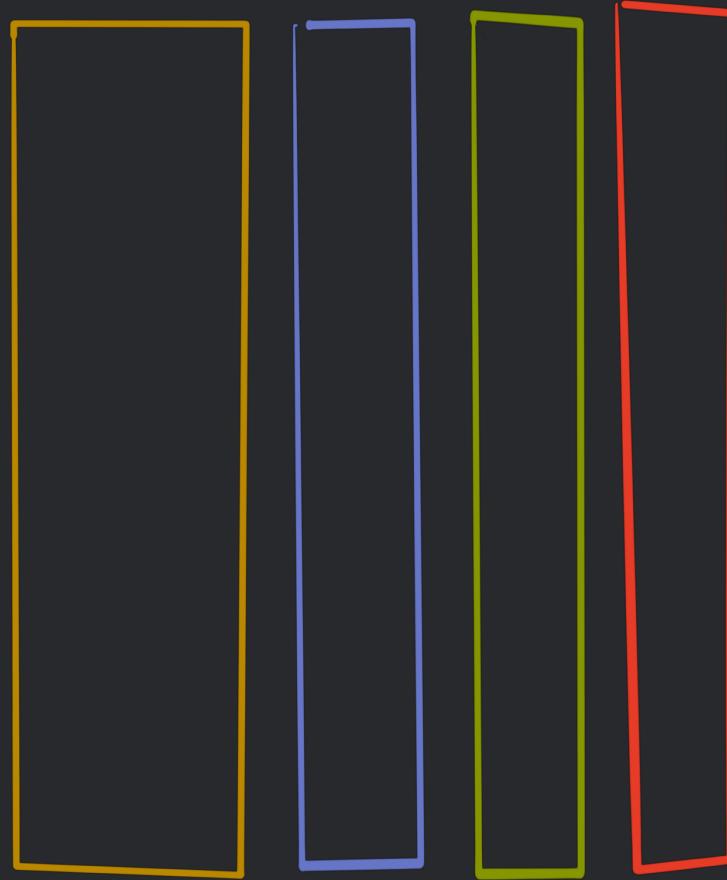
	A	B	C	D
1	4.4	a	2	0.1
2	3.1	b	42	0.4
3	2	c	8	0.9
4	8.3	d	15	0.3
:				



1xN ObjectBlock

# Pandas DataFrame

	A	B	C	D
1	4.4	a	2	0.1
2	3.1	b	42	0.4
3	2	c	8	0.9
4	8.3	d	15	0.3
:				

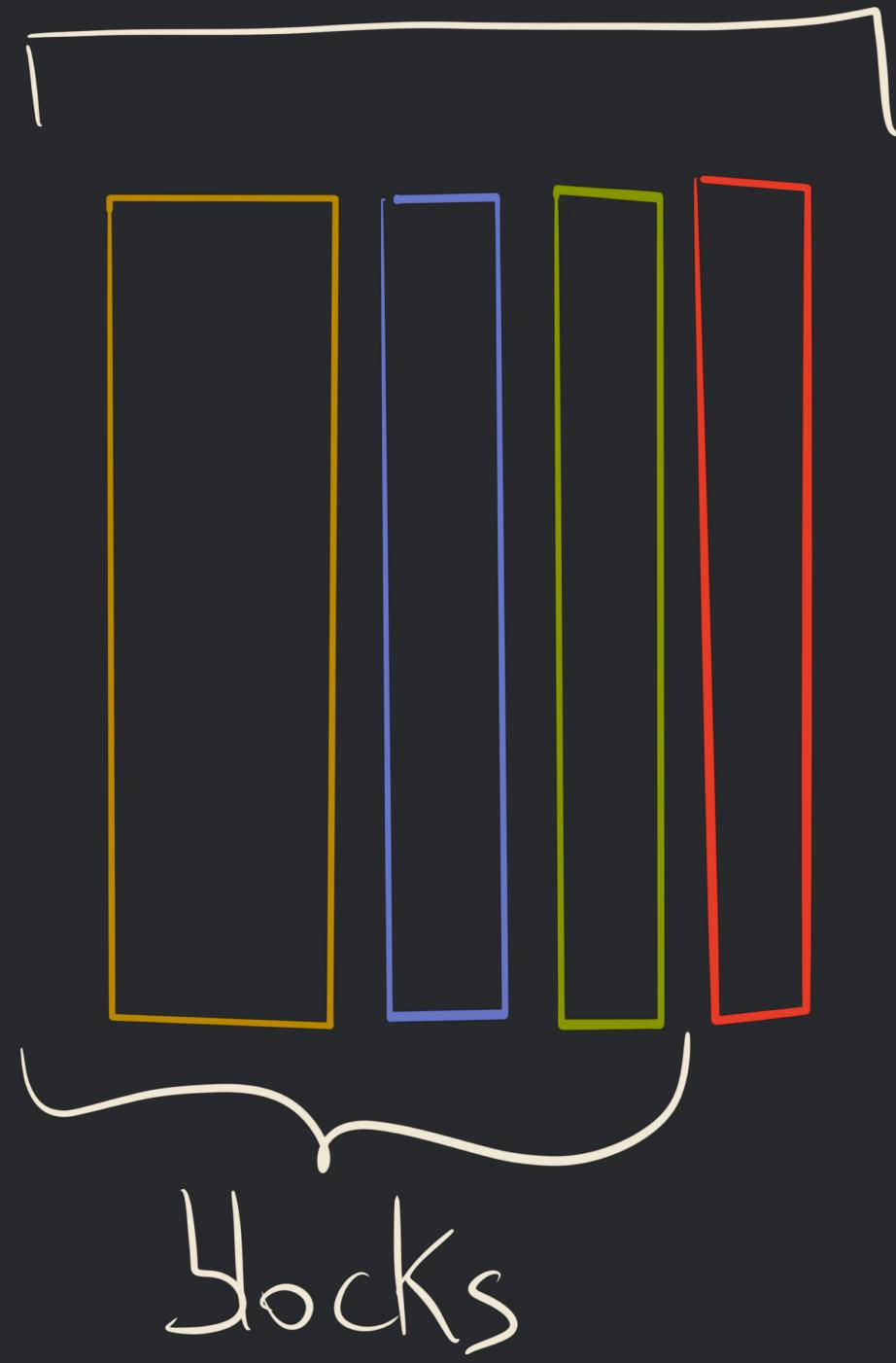


1xN IntBlock

Pandas DataFrame

	A	B	C	D
1	4.4	a	2	0.1
2	3.1	b	42	0.4
3	2	c	8	0.9
4	8.3	d	15	0.3
:				

BlockManager



# WHAT IS ARROW?

# WHAT IS ARROW?

- › CROSS-LANGUAGE IN-MEMORY COLUMNAR FORMAT LIBRARY

# WHAT IS ARROW?

- › CROSS-LANGUAGE IN-MEMORY COLUMNAR FORMAT LIBRARY
  - › OPTIMISED FOR EFFICIENCY ACROSS LANGUAGES

# WHAT IS ARROW?

- › CROSS-LANGUAGE IN-MEMORY COLUMNAR FORMAT LIBRARY
  - › OPTIMISED FOR EFFICIENCY ACROSS LANGUAGES
  - › INTEGRATES SEAMLESSLY WITH PANDAS

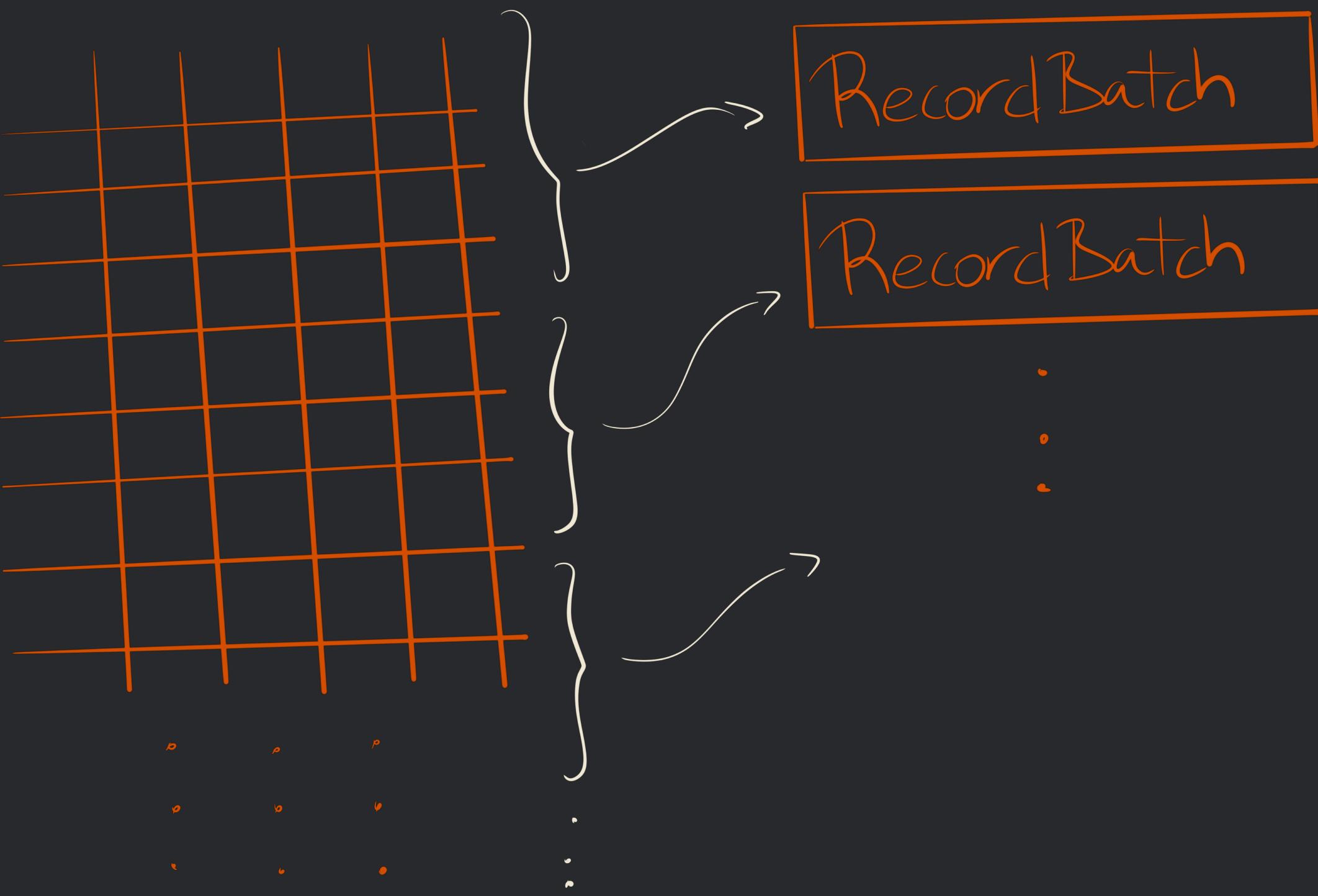
# HOW DOES ARROW MANAGE COLUMNAR DATA?

Table


P P P  
P P P  
P P P

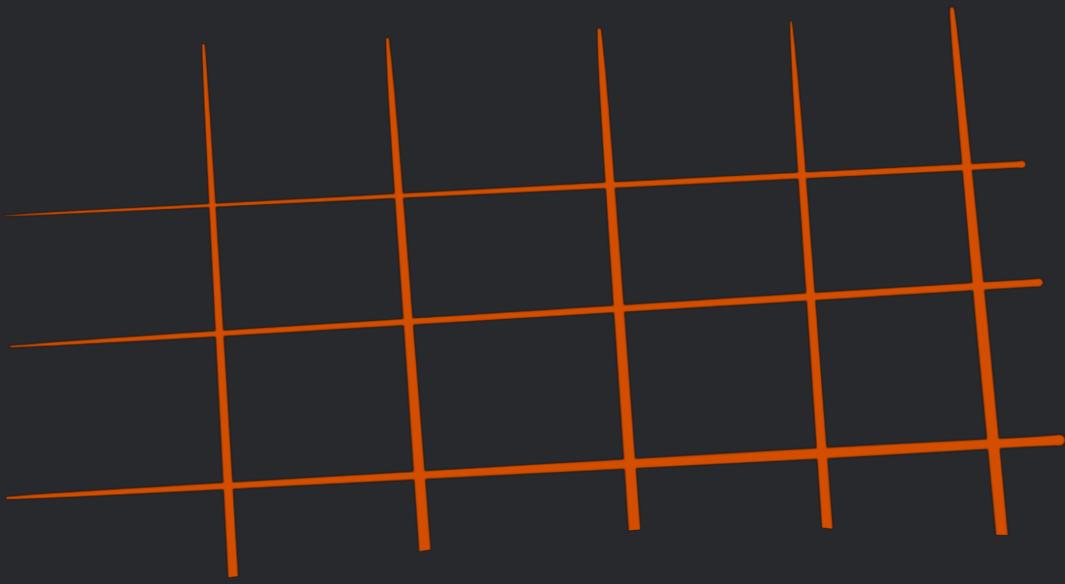
Everything starts with a  
Table structure

Table

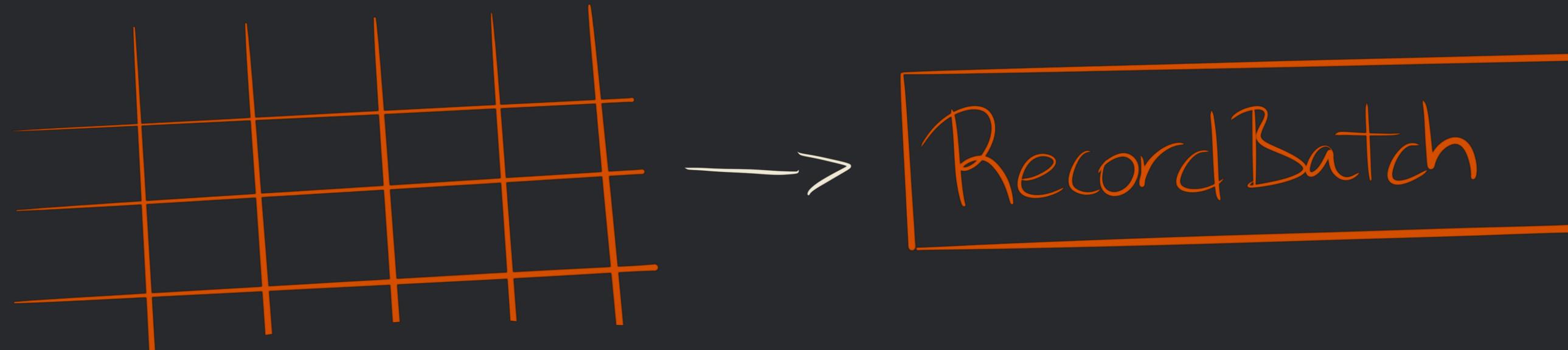


It is formed of RecordBatches, which contain a certain amount of rows. This makes it a streamable format

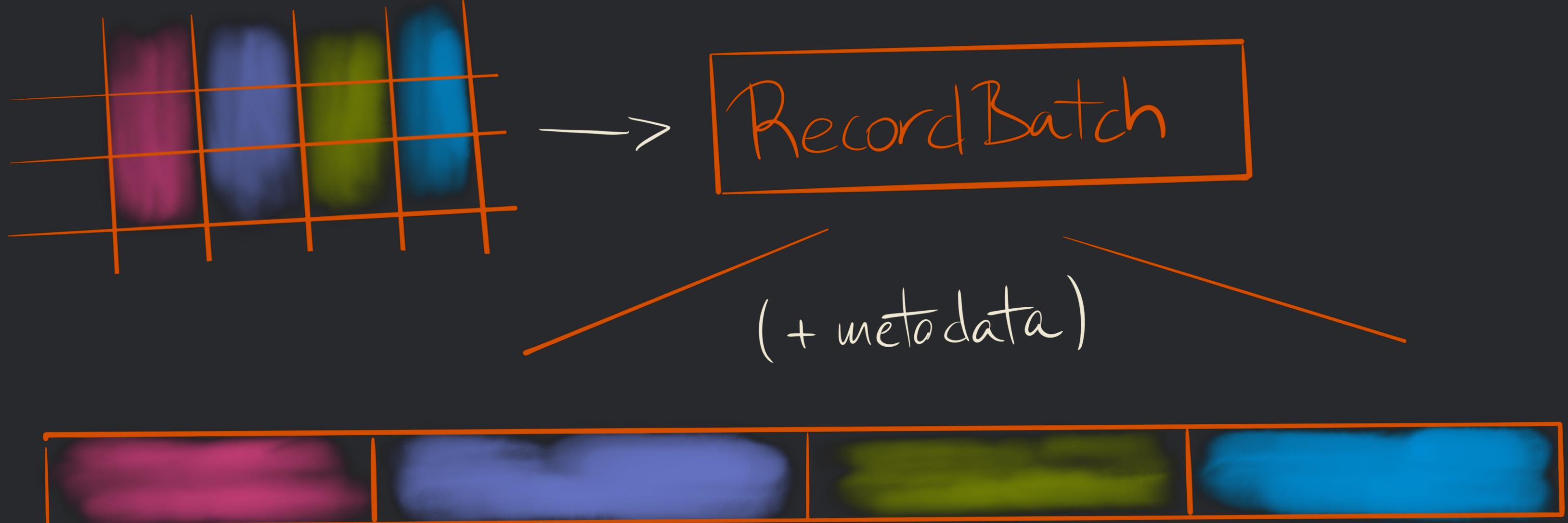
Some rows



Some rows



Some rows



Internally, RecordBatches have a columnar layout. Each RecordBatch contains some additional metadata.

# Arrow Table



In the end, an Arrow Table is formed of a set of RecordBatches



Basically, from **Pandas** to **Arrow**  
you build RecordBatches out of  
data  
consumed from a BlockManager,  
in reverse, you build a  
BlockManager with data  
consumed from RecordBatches.



› **ARROW USES** RecordBatches

Basically, from **Pandas** to **Arrow**  
you build RecordBatches out of  
data  
consumed from a **BlockManager**,  
in reverse, you build a  
**BlockManager** with data  
consumed from RecordBatches.



- > **ARROW USES** RecordBatches
- > **PANDAS USES BLOCKS HANDLED BY A BlockManager**

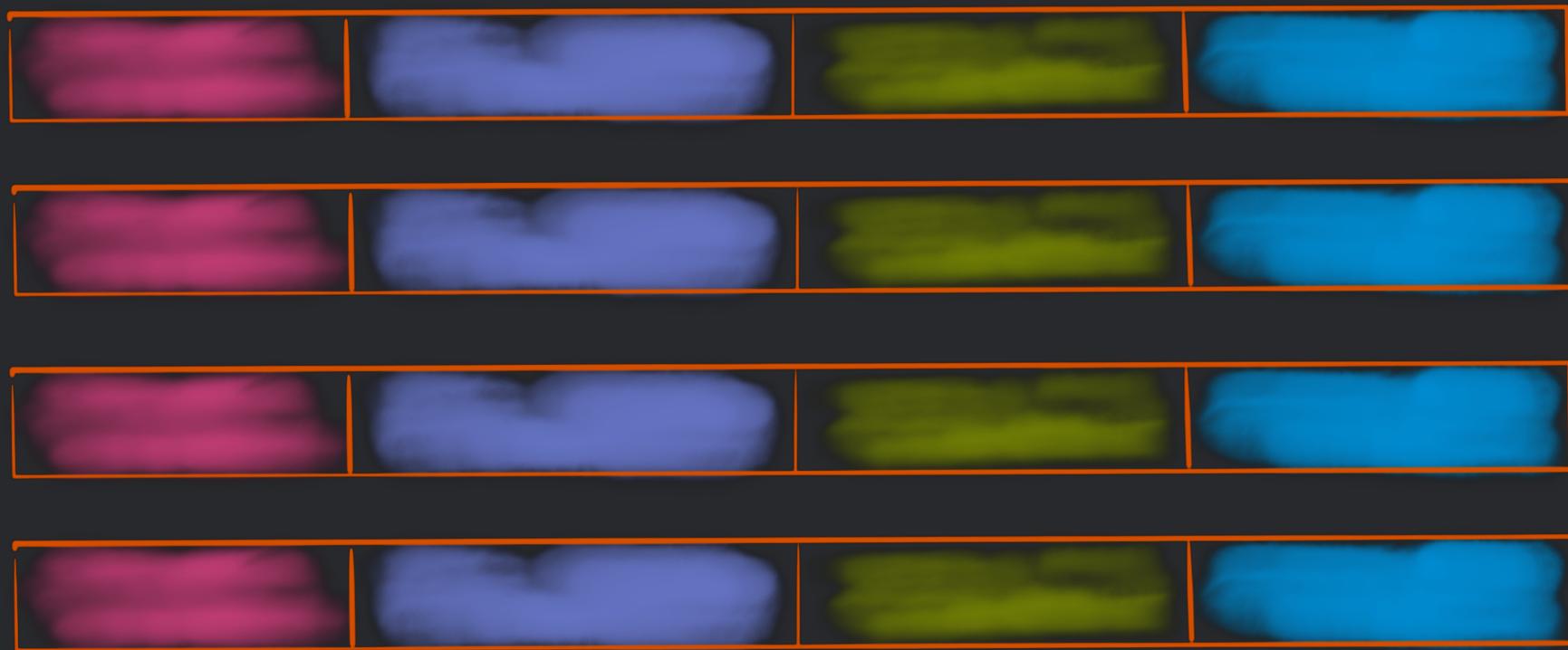
Basically, from **Pandas** to **Arrow**  
you build RecordBatches out of  
data  
consumed from a BlockManager,  
in reverse, you build a  
BlockManager with data  
consumed from RecordBatches.



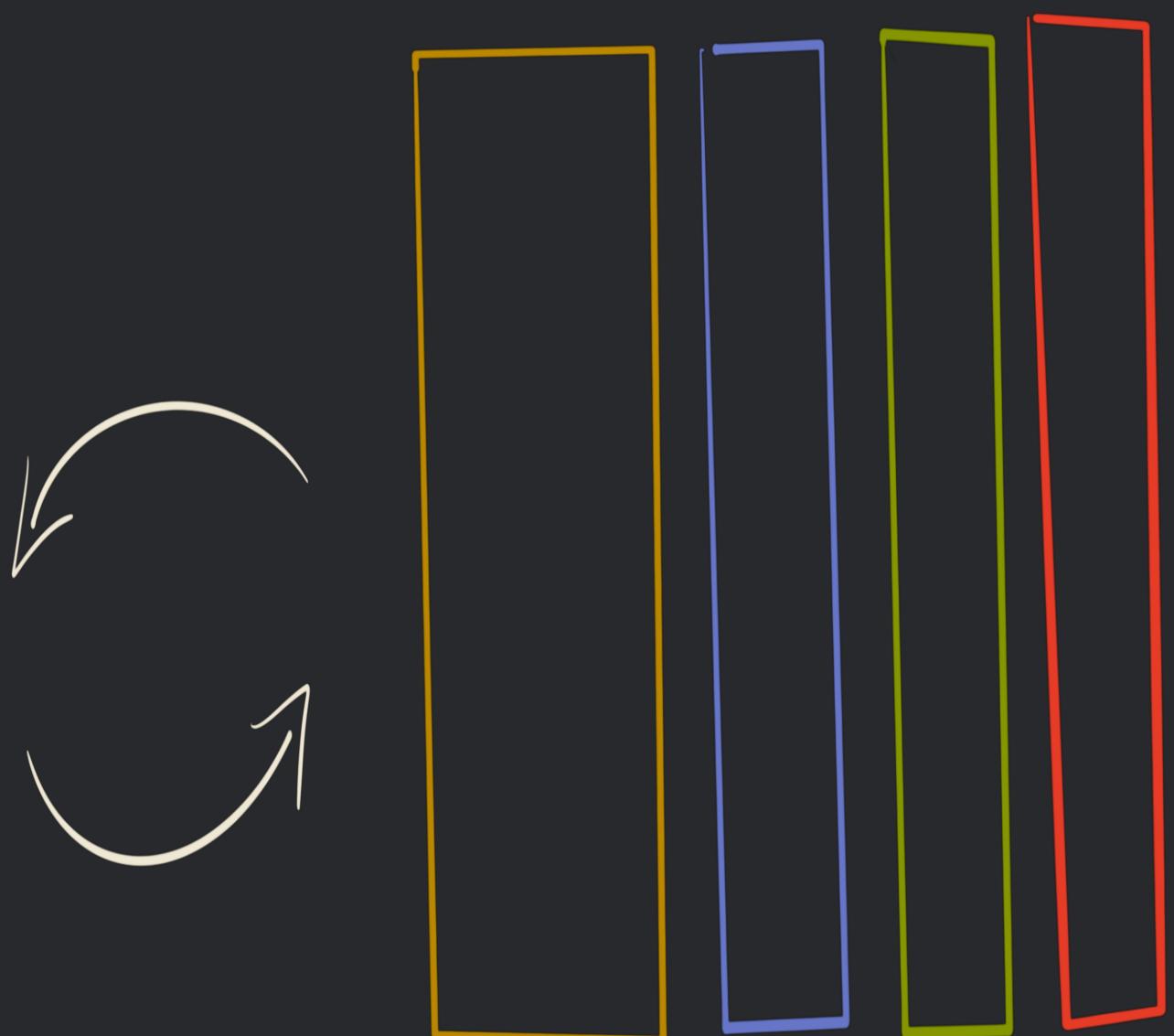
- > ARROW USES RecordBatches
- > PANDAS USES BLOCKS HANDLED BY A BlockManager
- > YOU CAN CONVERT AN ARROW Table INTO A PANDAS DataFrame EASILY

Basically, from Pandas to Arrow  
you build RecordBatches out of  
data  
consumed from a BlockManager,  
in reverse, you build a  
BlockManager with data  
consumed from RecordBatches.

# Arrow Table



# Pandas BlockManager



The internal layouts are similar enough that transforming one into the other is close to being zero-copy. Since most of the code for this step is written in C and Cython, it is *very fast*. Note that Pandas is already storing data in a columnar way: *Arrow just offers an unified way to be able to share the same data representation among languages.* Thanks to Marc Garcia for pointing out this should be made more clear here

# WHAT IS SPARK?

# WHAT IS SPARK?

- > DISTRIBUTED COMPUTATION FRAMEWORK

# WHAT IS SPARK?

- > DISTRIBUTED COMPUTATION FRAMEWORK
- > OPEN SOURCE

# WHAT IS SPARK?

- > DISTRIBUTED COMPUTATION FRAMEWORK
  - > OPEN SOURCE
  - > EASY TO USE

# WHAT IS SPARK?

- > DISTRIBUTED COMPUTATION FRAMEWORK
  - > OPEN SOURCE
  - > EASY TO USE
- > SCALES HORIZONTALLY AND VERTICALLY

HOW DOES  
SPARK WORK?

# SPARK USUALLY SITS ON TOP OF A CLUSTER MANAGER



Cluster Manager

This can be standalone, YARN,  
Mesos or in the bleeding edge,  
Kubernetes (using the  
Kubernetes scheduler)



AND A  
**DISTRIBUTED  
STORAGE**

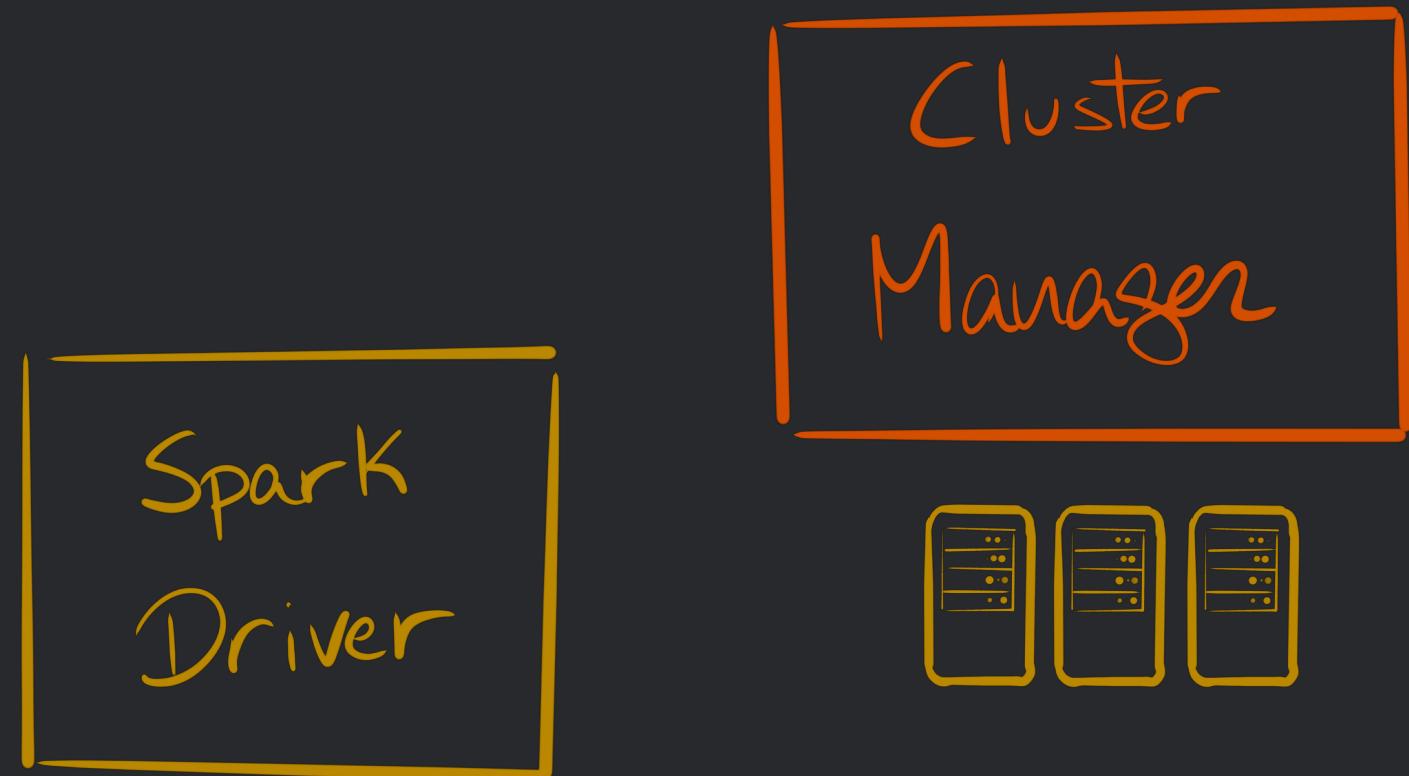
Cluster Manager

Distributed Storage



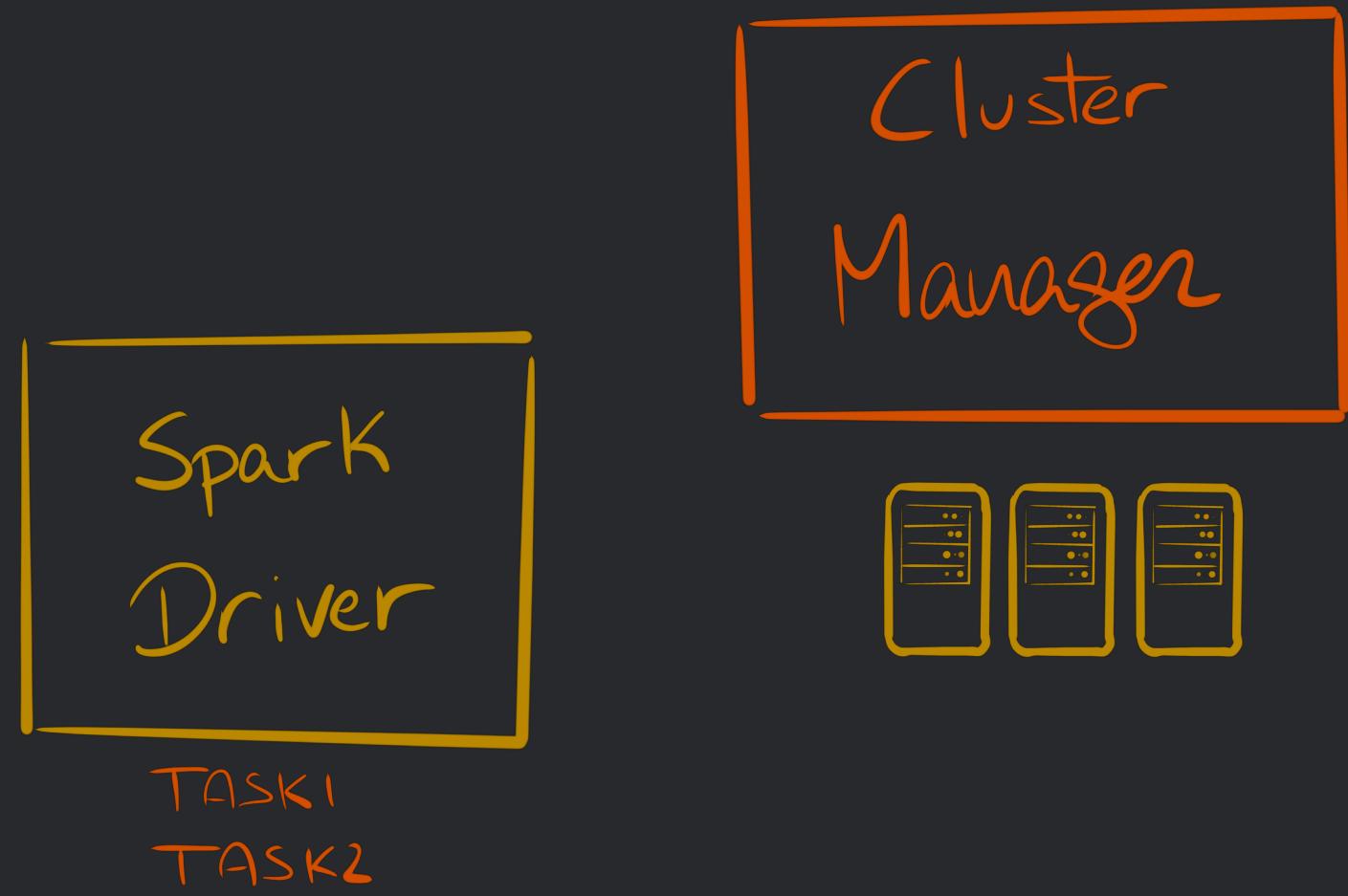
A SPARK PROGRAM  
RUNS IN THE DRIVER

THE DRIVER REQUESTS  
RESOURCES FROM THE  
CLUSTER MANAGER TO  
RUN TASKS

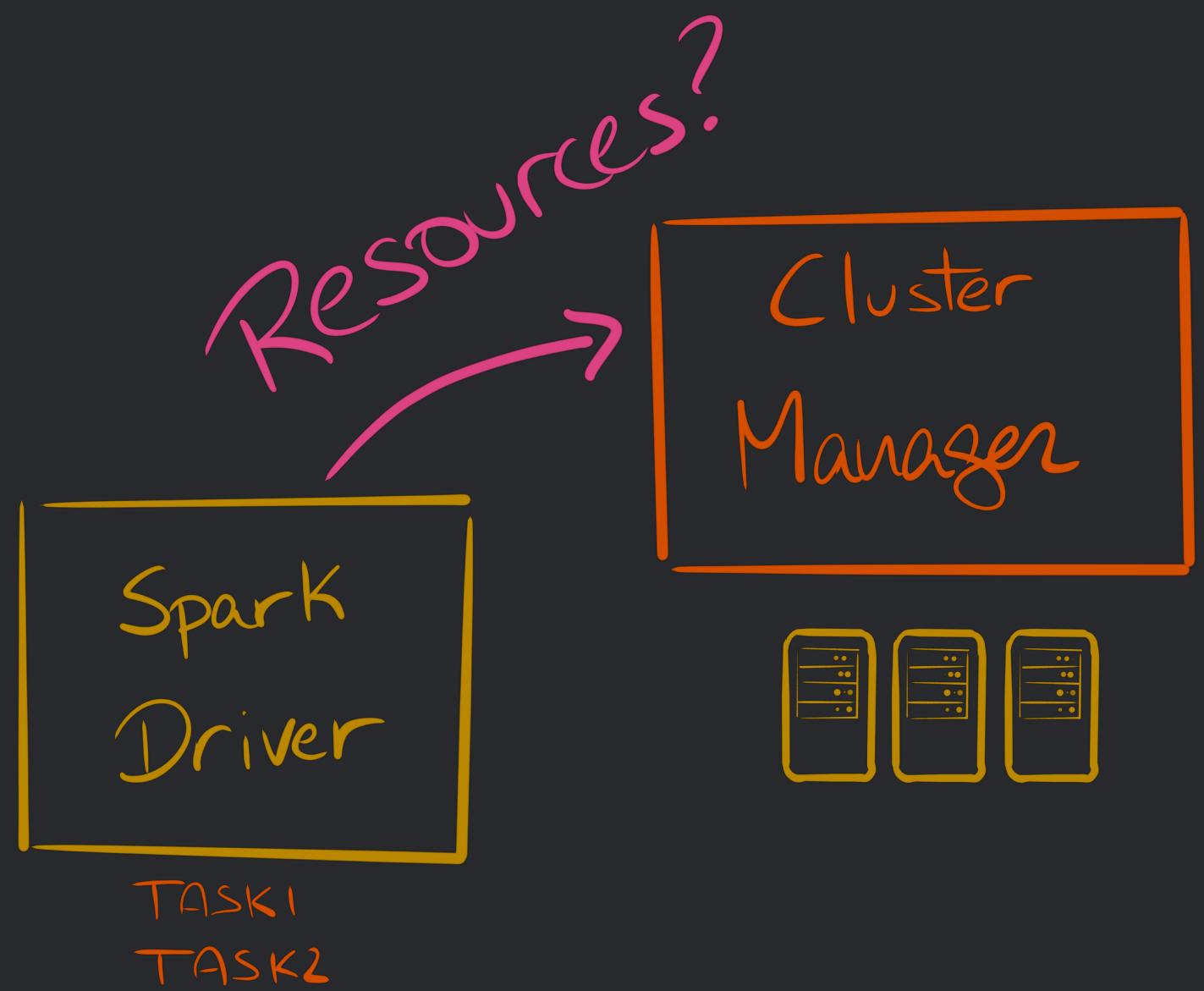


We usually don't need to worry  
about what the executors do  
(unless they blow up)

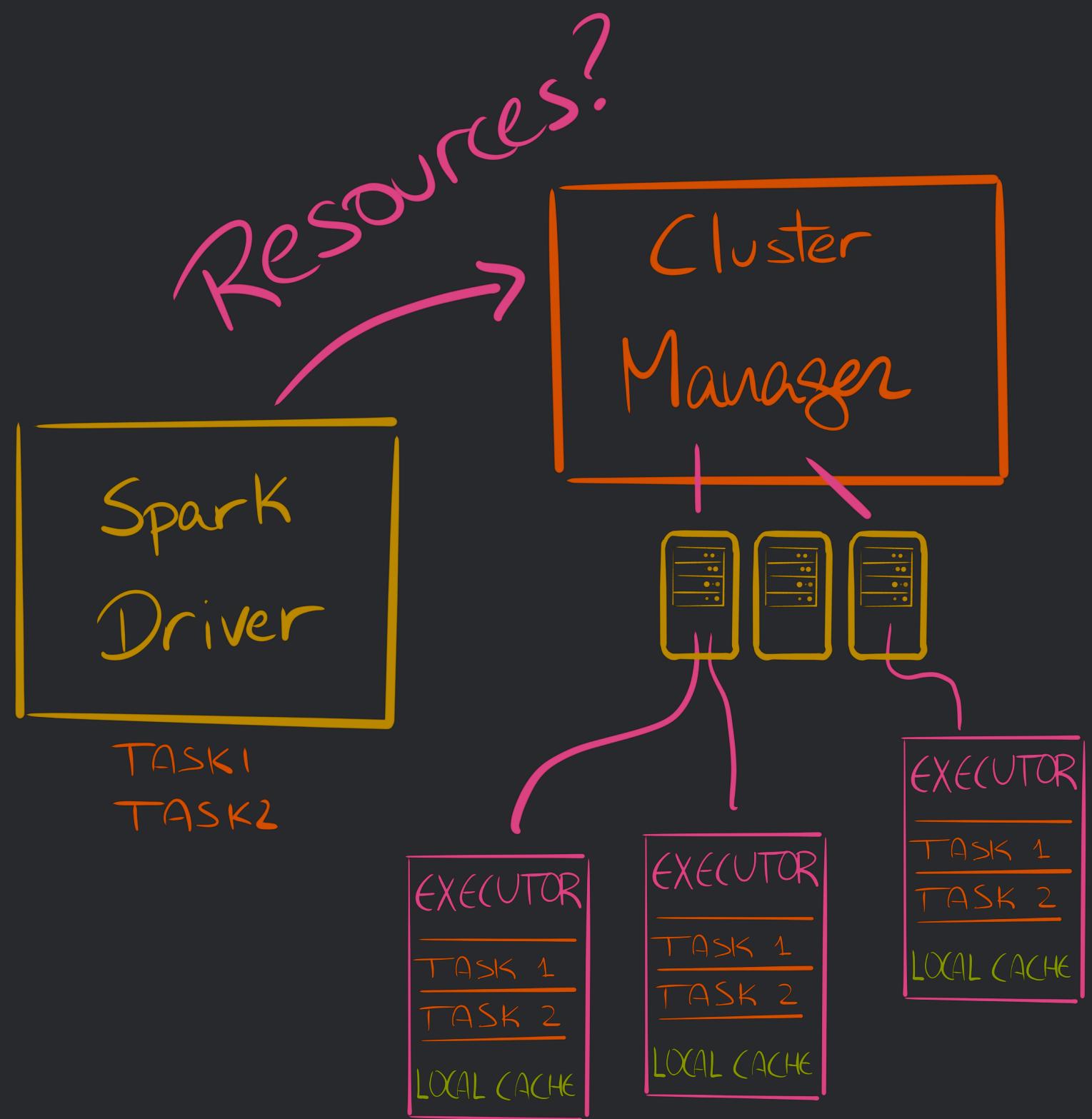
THE DRIVER REQUESTS  
RESOURCES FROM THE  
CLUSTER MANAGER TO  
RUN TASKS



THE DRIVER REQUESTS  
RESOURCES FROM THE  
CLUSTER MANAGER TO  
RUN TASKS



THE DRIVER REQUESTS  
RESOURCES FROM THE  
CLUSTER MANAGER TO  
RUN TASKS



This is very nice, but what is the magic that lets us compute things on several machines, and is machine-failure safe?

# THE MAIN BUILDING BLOCK IS THE RDD: **RESILIENT DISTRIBUTED DATASET**

RDDs are defined in the Scala core. In Python and R we end up interacting with the underlying JVM objects

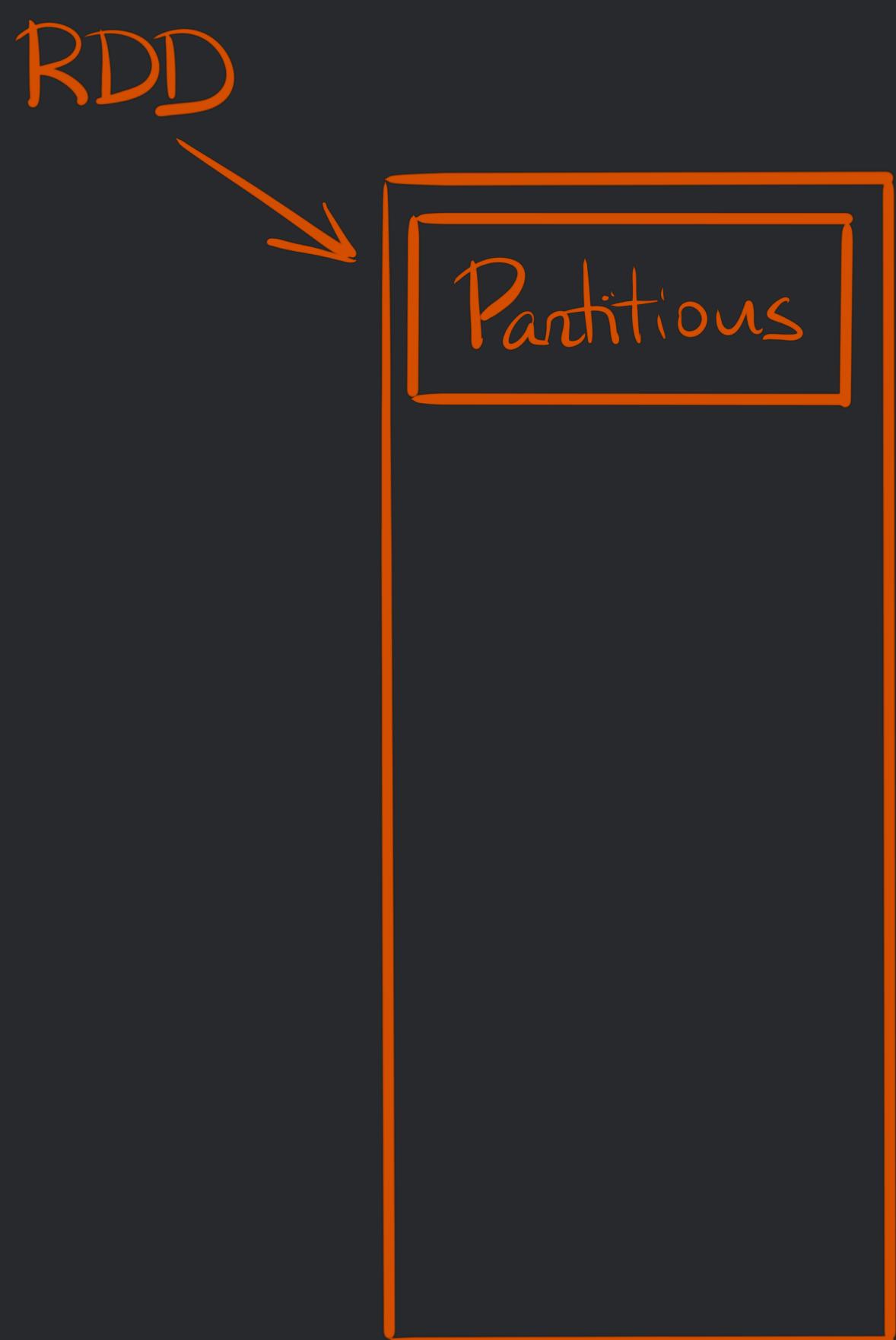


Happy RDD

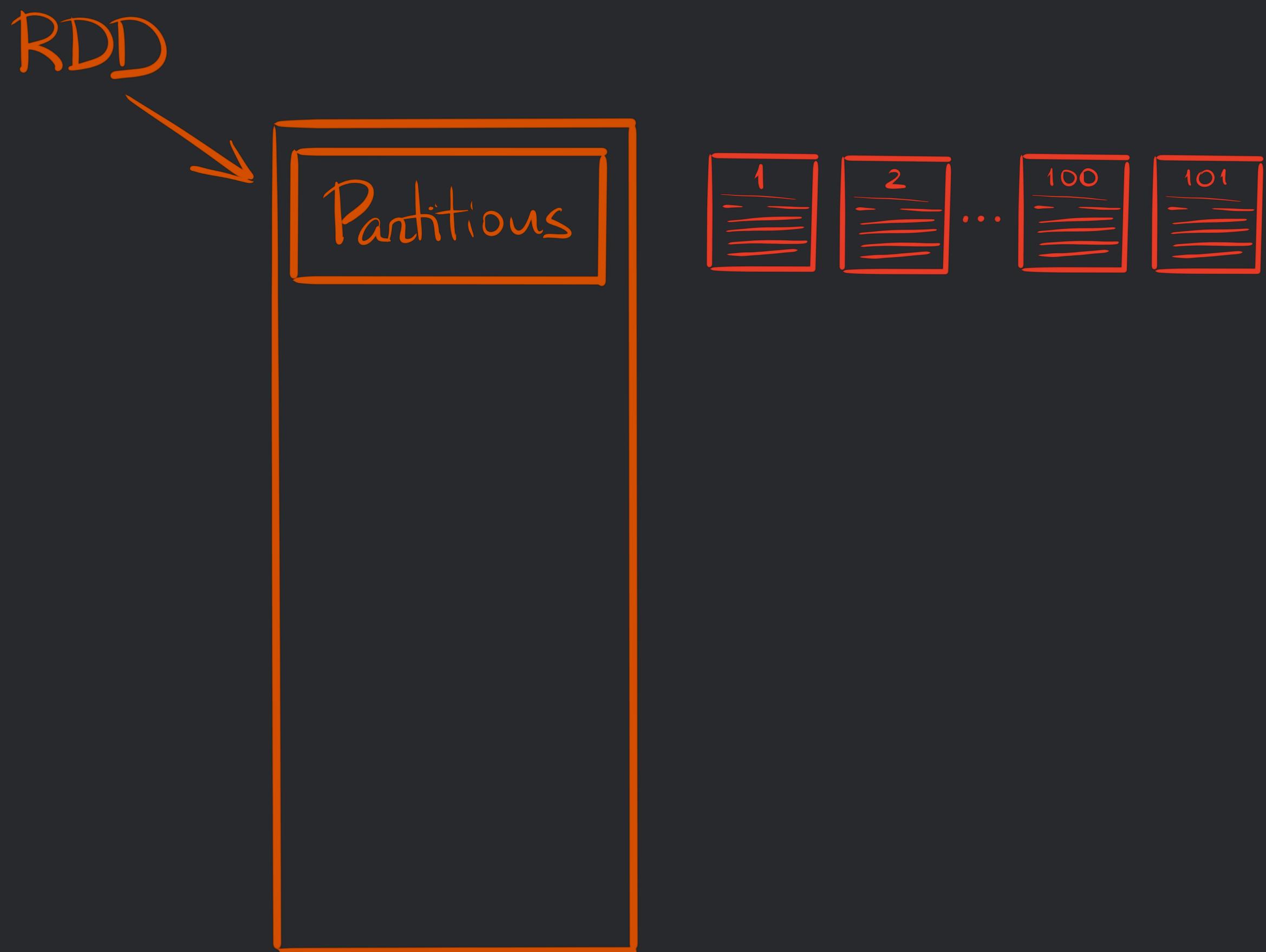
RDD



Happy RDD



Partitions define how the data  
is *partitioned* across machines

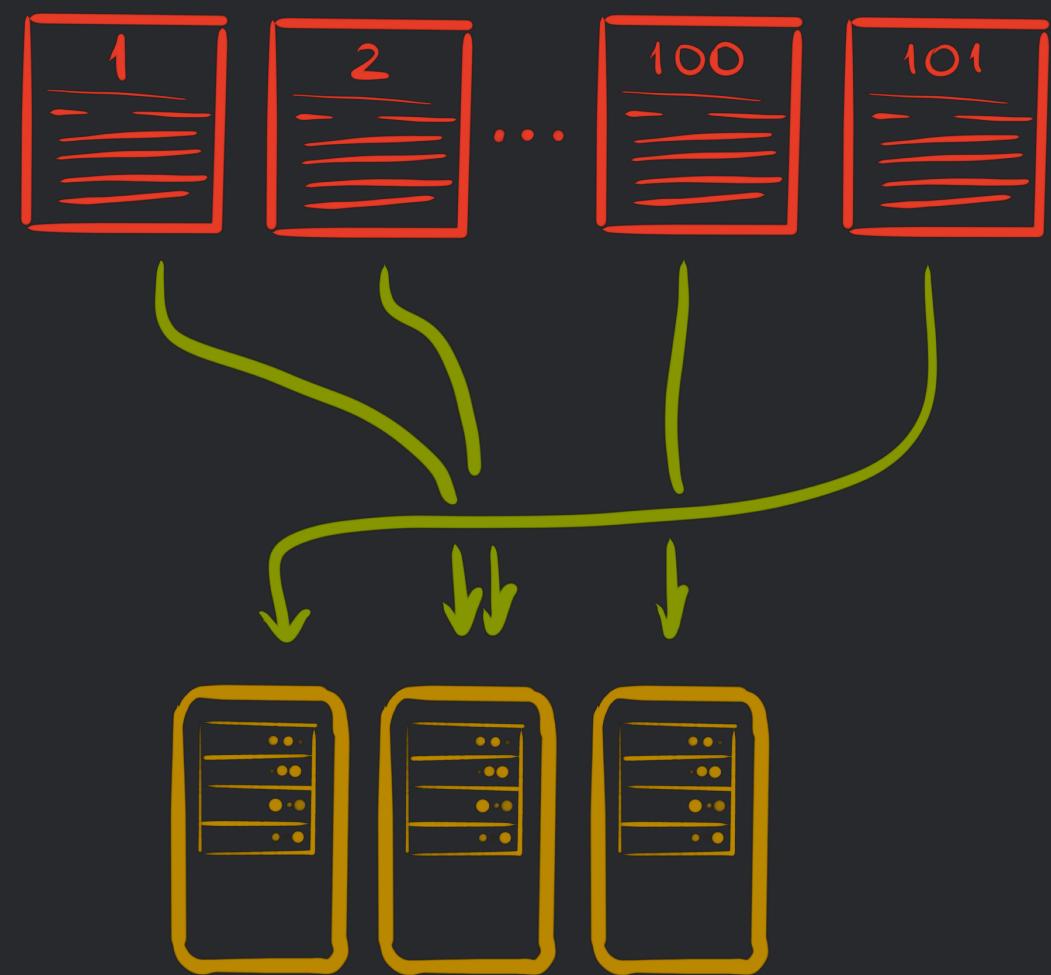


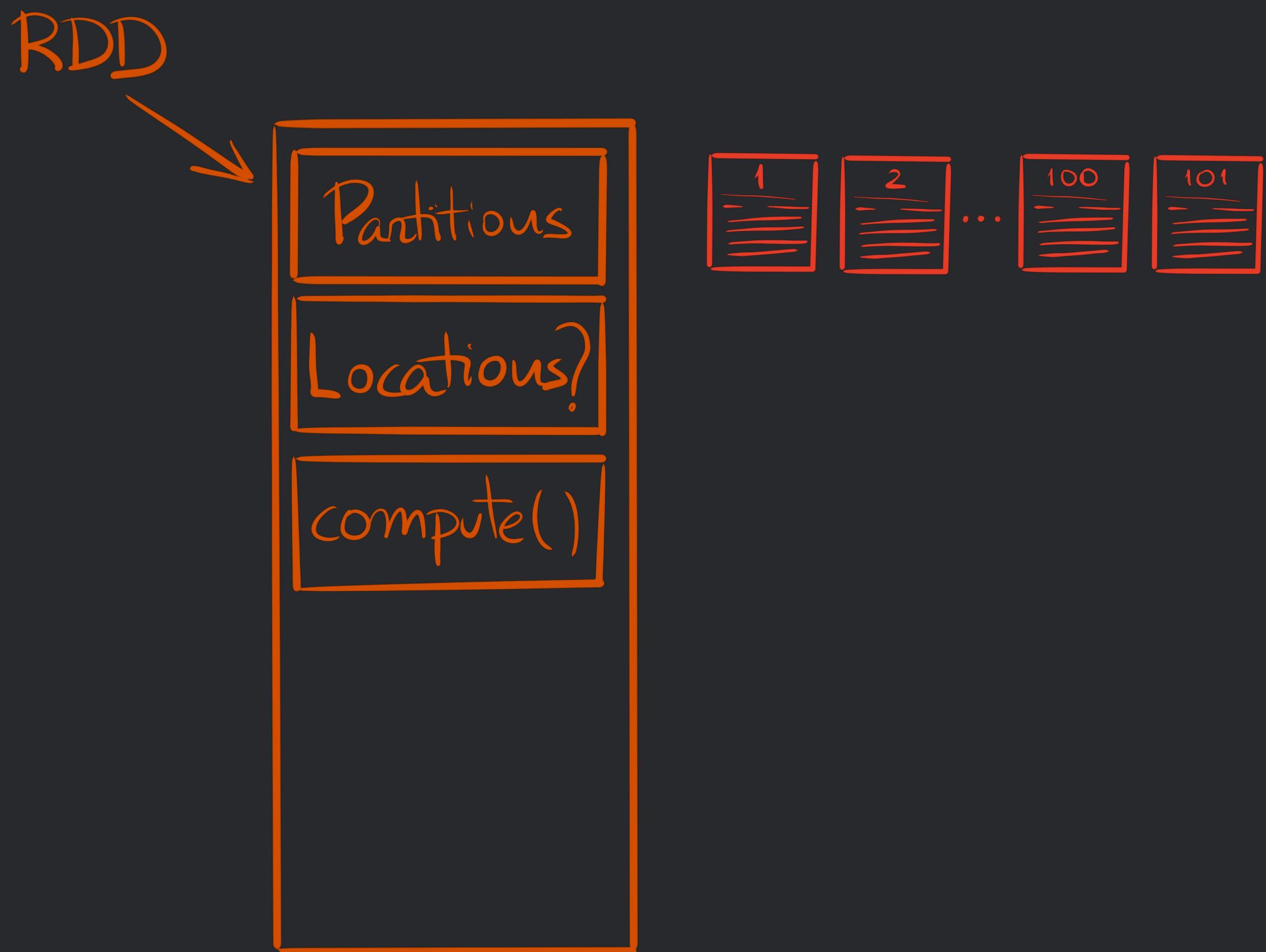
Locations  
(preferredLocations)  
are optional, and allow for fine  
grained  
execution to avoid shuffling  
data across the cluster

RDD



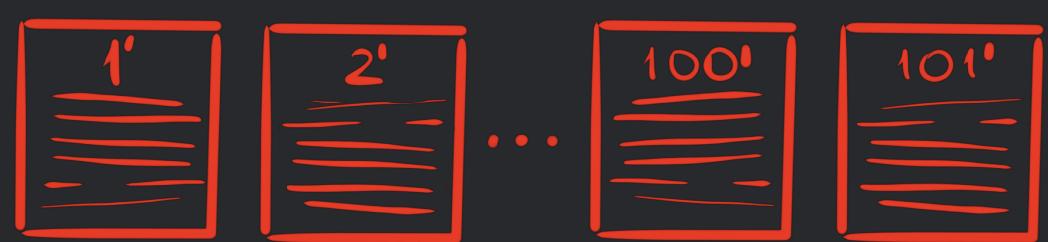
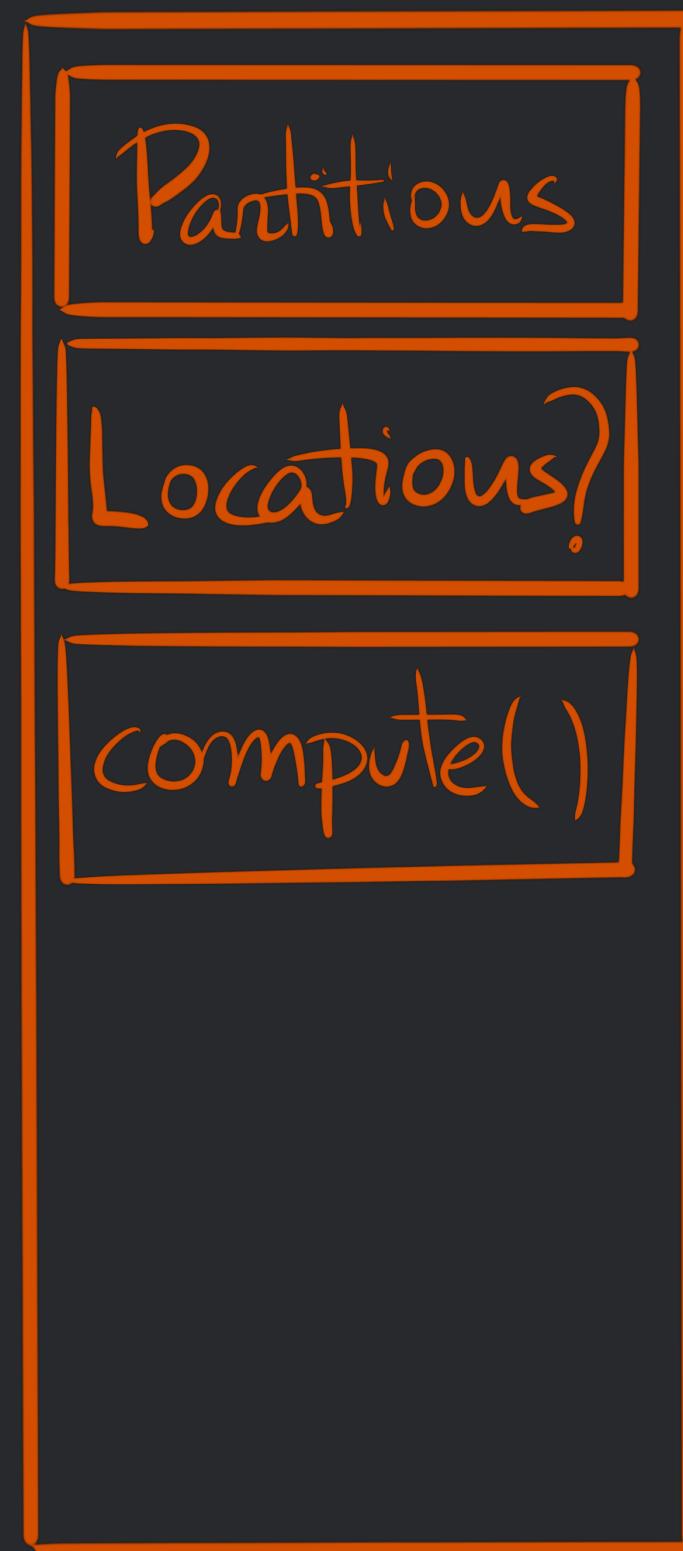
RDD

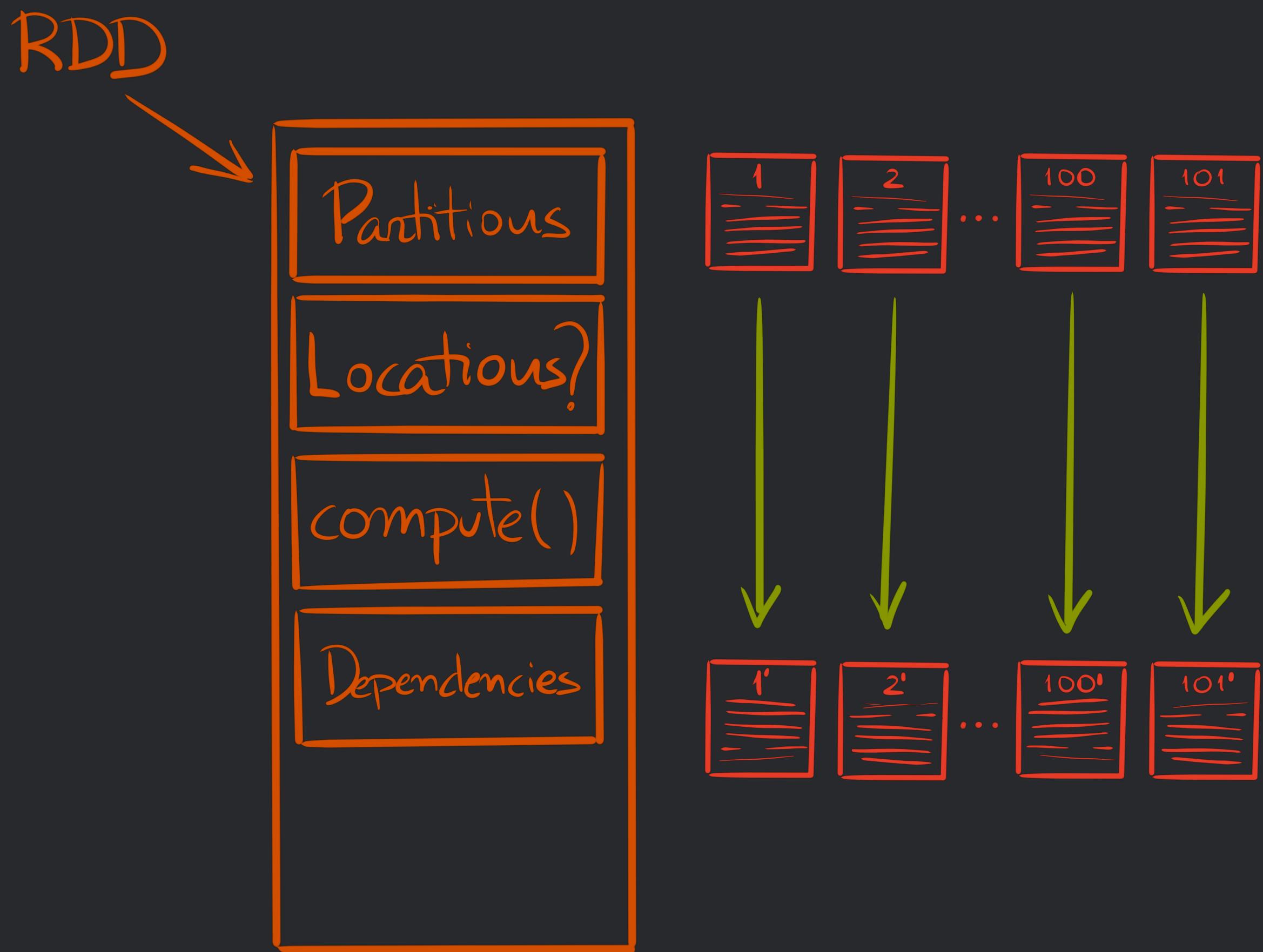




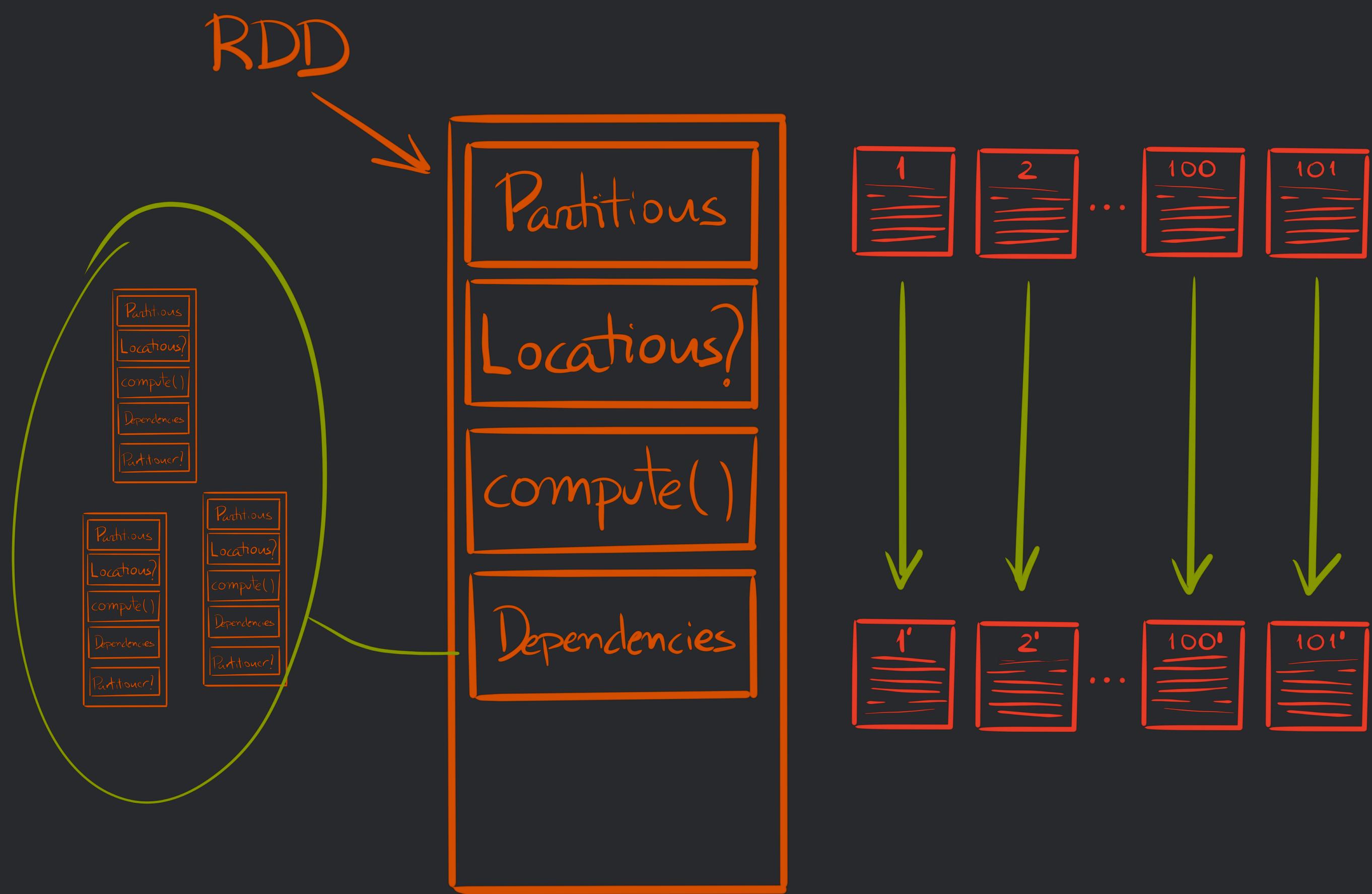
Compute is evaluated in the executors, each executor has access to its assigned partitions. The result of compute is a new RDD, with different data

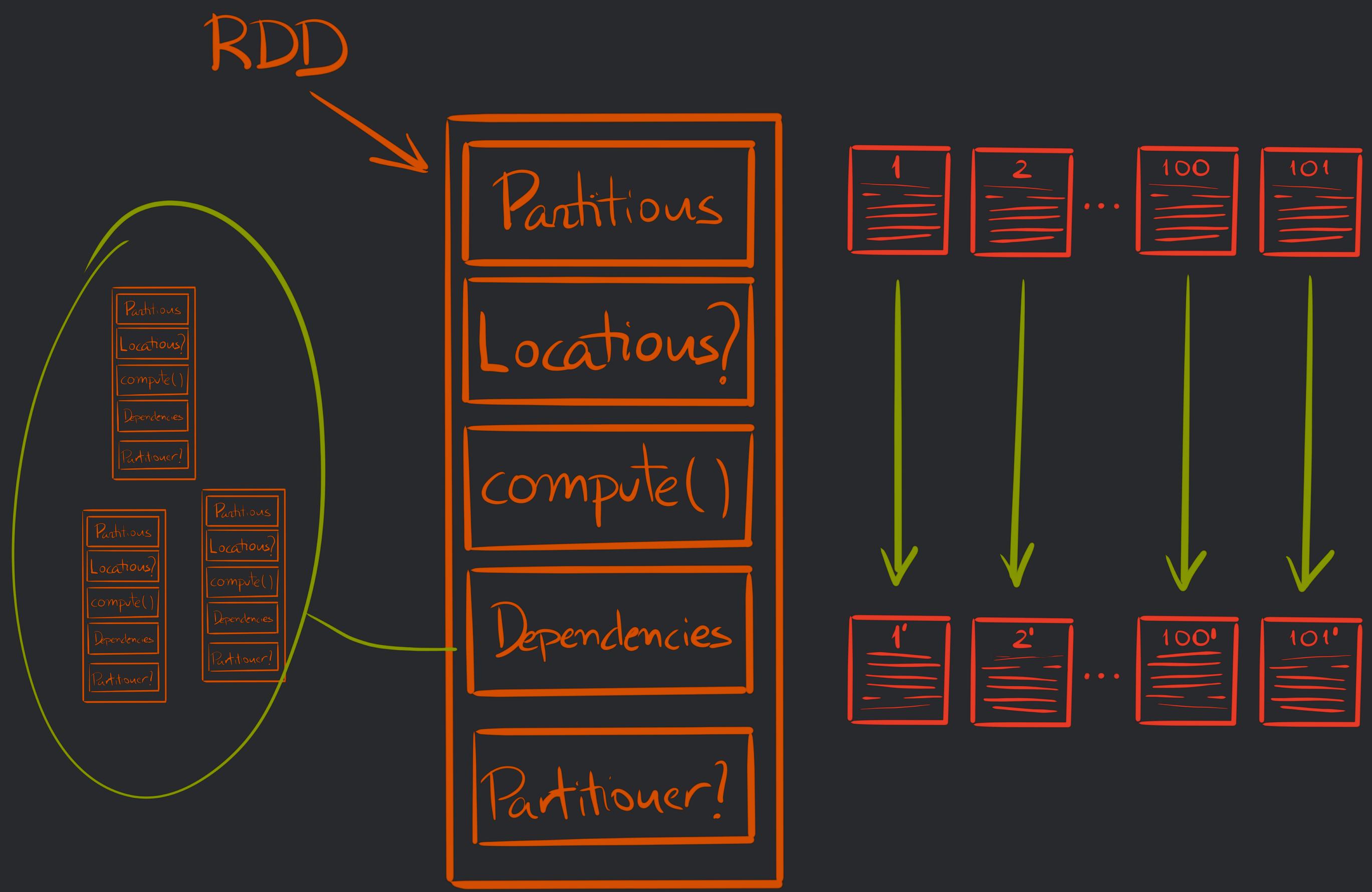
RDD





Dependencies are also called *lineage*. Each **RDD** (and by extension, each partition) has a specific way to be computed. If for some reason one is lost (say, a machine dies), **Spark** knows how to recompute *only* that





The partitioner needs to be a  
1-1 function from keys/  
whatever to the data, to  
allow recomputing

# PYSPARK

PYSPARK OFFERS A  
PYTHON API TO THE SCALA  
CORE OF SPARK

# IT USES THE PY4J BRIDGE

Each object in PySpark comes bundled with a JVM gateway (started when the Python driver starts). Python methods then act on the internal JVM object by passing serialised messages to the Py4J gateway

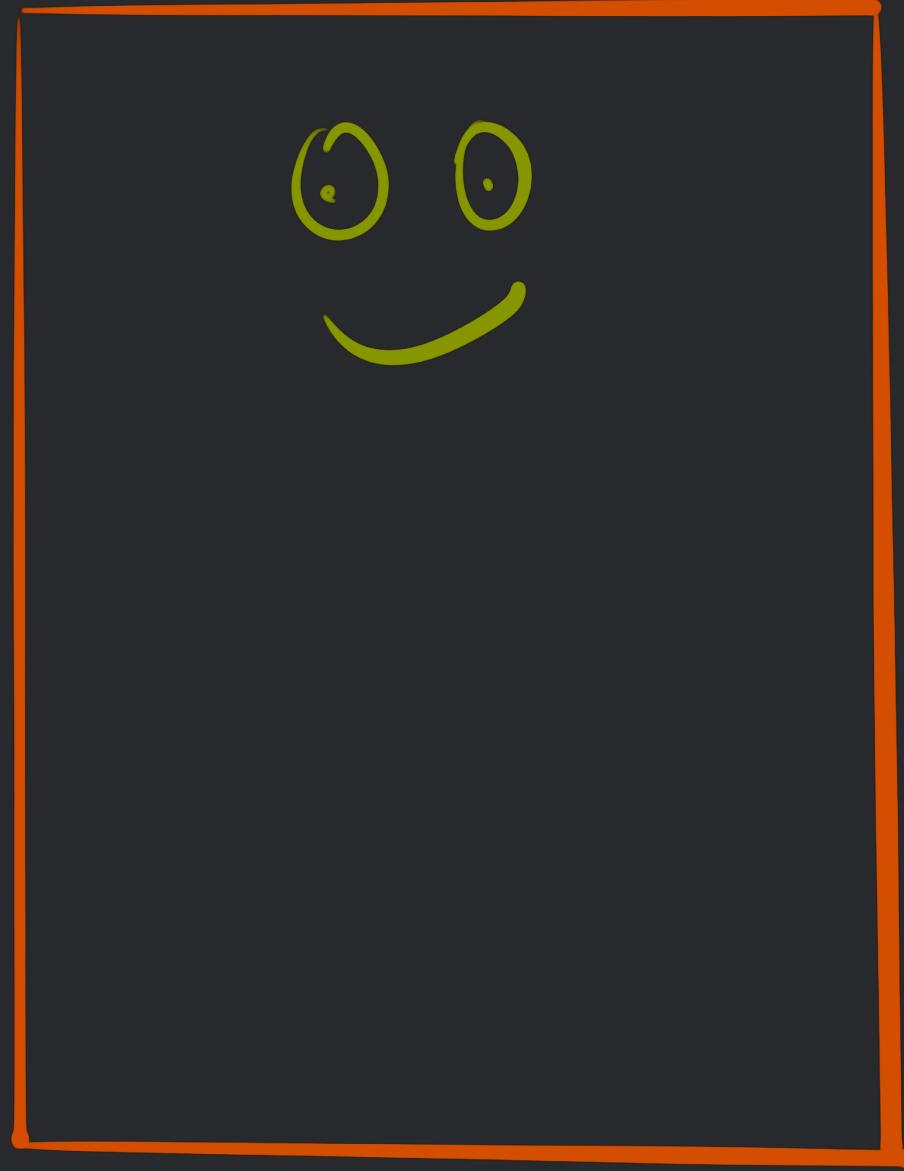
```
# Connect to the gateway
gateway = JavaGateway(
    gateway_parameters=GatewayParameters(
        port=gateway_port,
        auth_token=gateway_secret,
        auto_convert=True))

# Import the classes used by PySpark
java_import(gateway.jvm, "org.apache.spark.SparkConf")
java_import(gateway.jvm, "org.apache.spark.api.java.*")
java_import(gateway.jvm, "org.apache.spark.api.python.*")
.

.

.

return gateway
```



# Happy RDD in Python

Python Land



RDD in  
Python Land



Some `_jrdd` appears. Each "Spark" related object has some internal relationship with an equivalent JVM object. An RDD, for instance, has a `_jrdd`, which refers to how the RDD was created in the JVM. By extension, if in Python you create an RDD from a file, for instance, this will call the JVM method to do so, and the resulting Python object will have a `_jrdd` pointing to that.

Python Land



M  
is for  
murder

Just like M is for Murder...

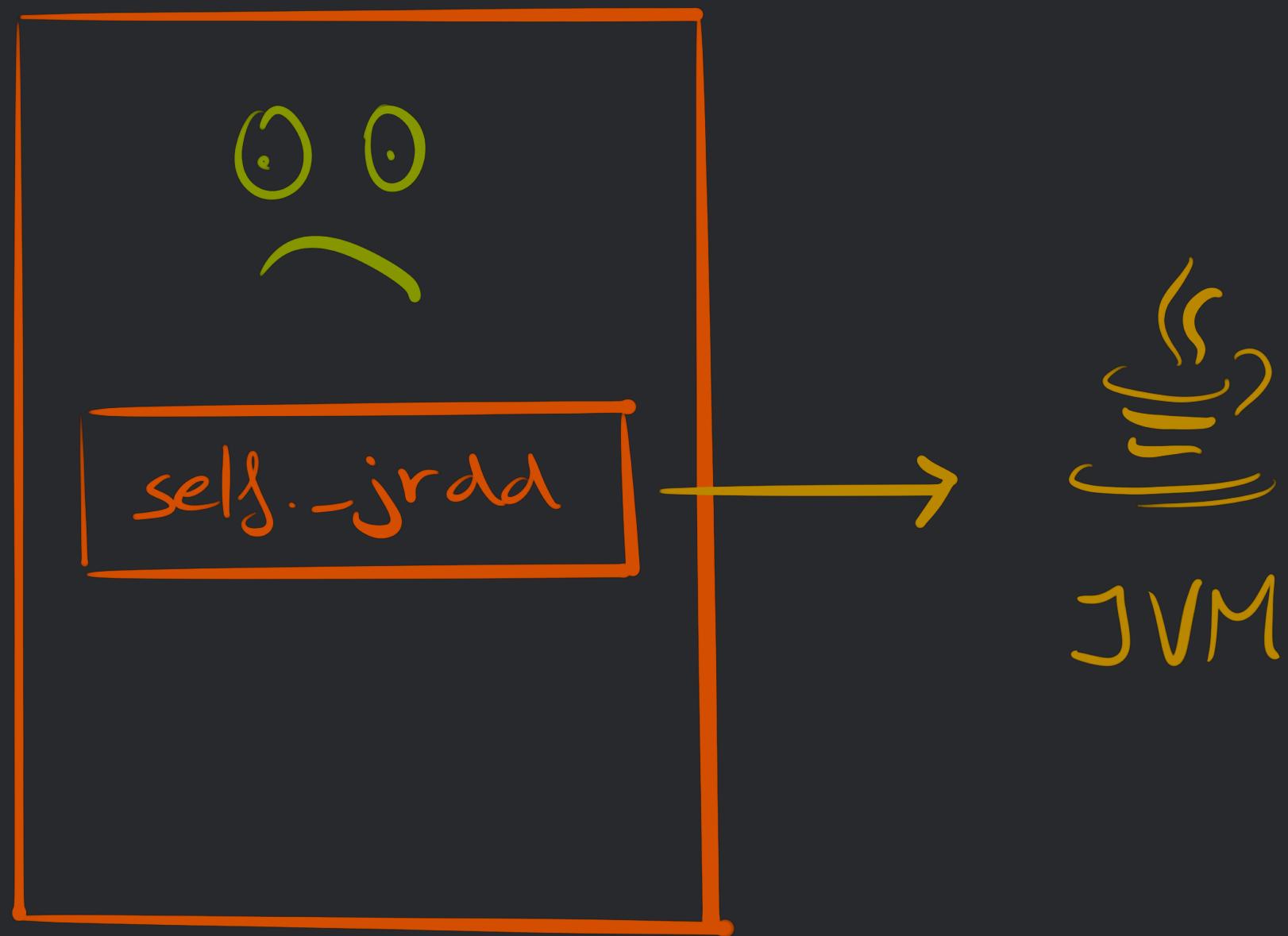
*Python Land*



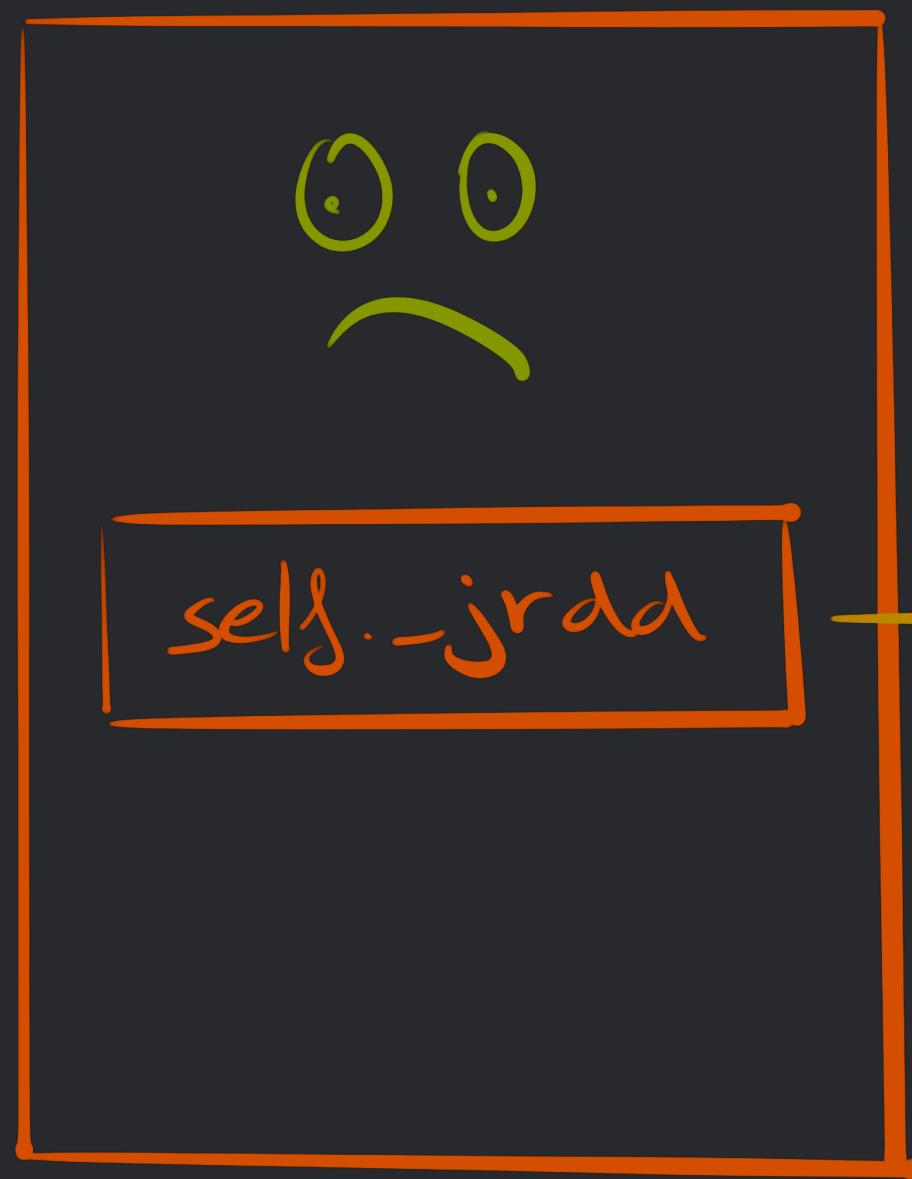
*j*  
is for  
java

J is for Java. There are a *lot* of properties prefixed with *j* in PySpark, to denote they refer to a JVM object/property/class

*Python RDD in Land*



PyRDD in  
Python Land



Py4J bridge  
JVM

# THE MAIN ENTRYPONTS ARE RDD AND PipelinedRDD(RDD)

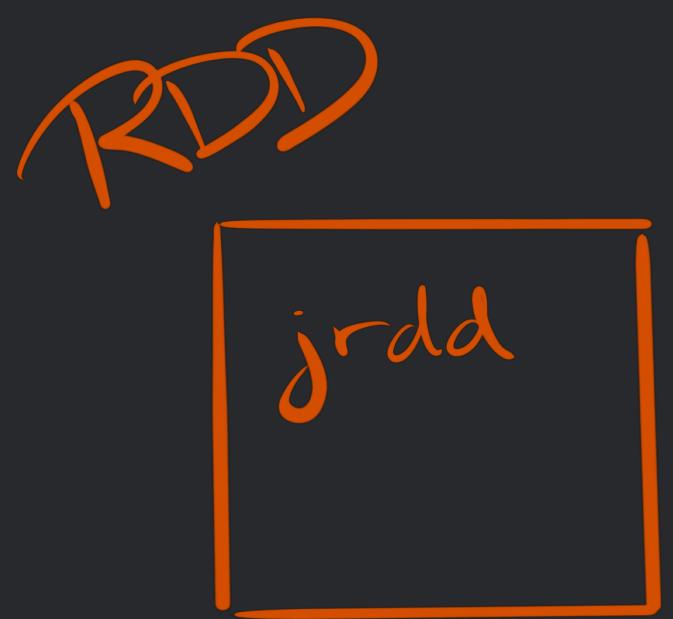
In Python land. The first exposes the API of Scala RDDs (by interacting with the JVM connected to the RDD), the second defines how to apply a Python function to an RDD. For instance, all map methods on RDDs defer to the mapPartitionsWithIndex method, which builds a PipelinedRDD wrapping the Python function to map, and then does some other things I'll explain later

PipelinedRDD  
BUILDS IN THE JVM A  
PythonRDD





# Angry RDD



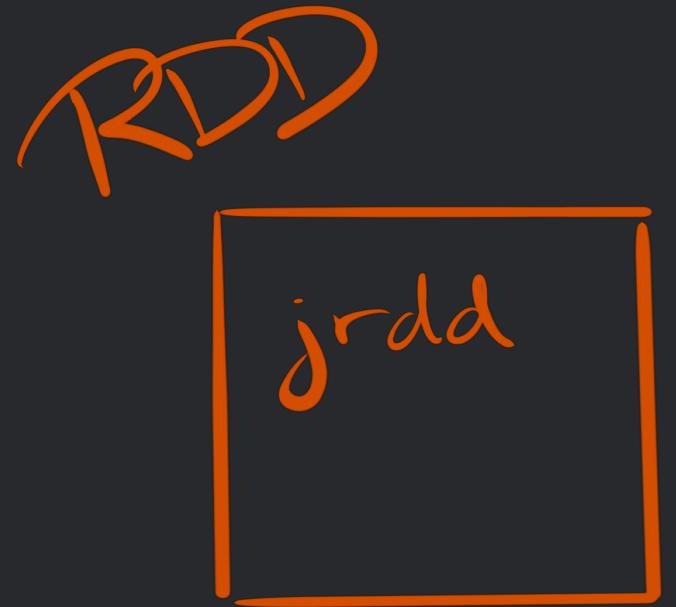
With its jrdd (so, it's in  
Python land)

RDD



MAP(f)

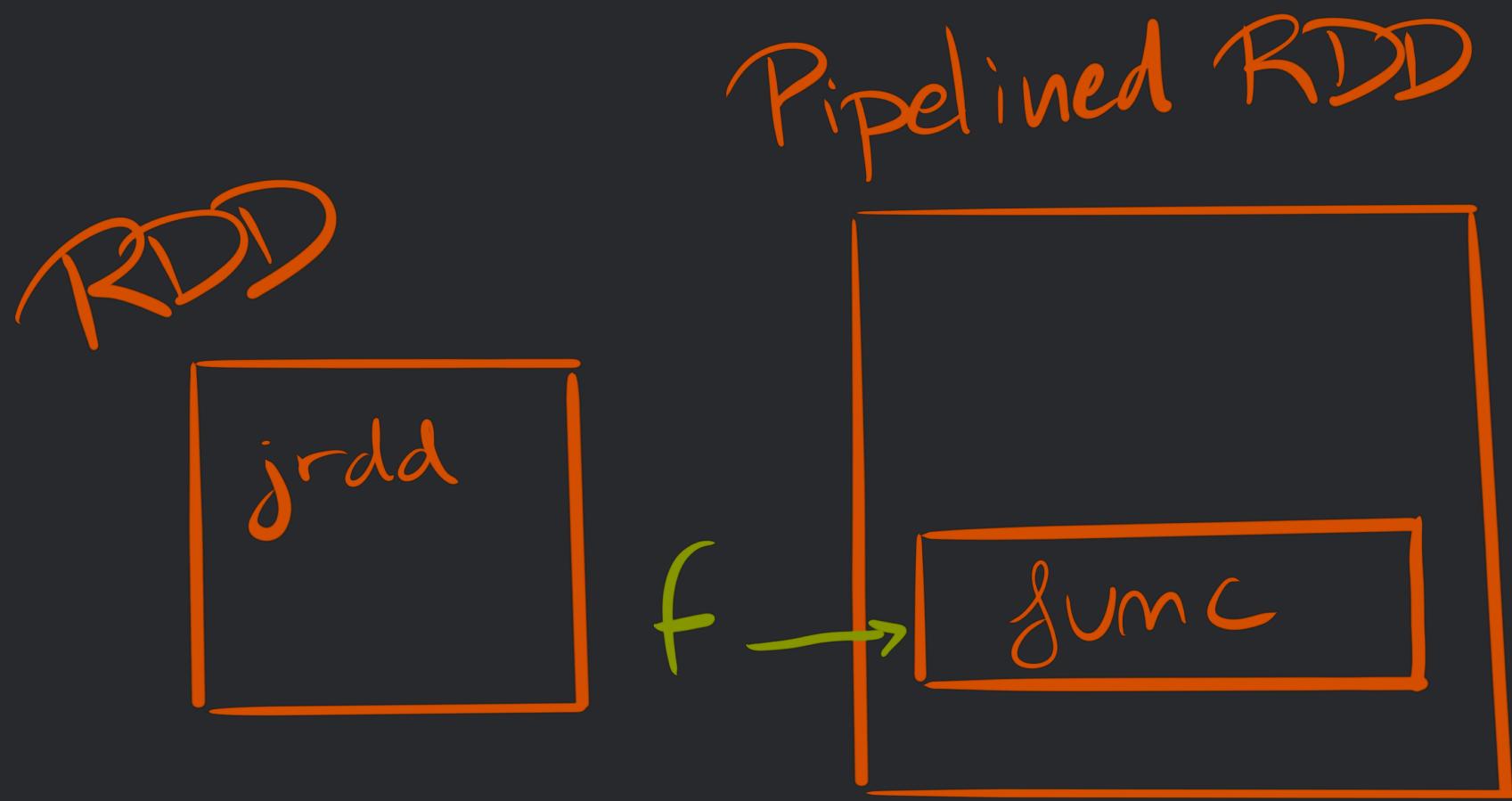
Let's map over this RDD!



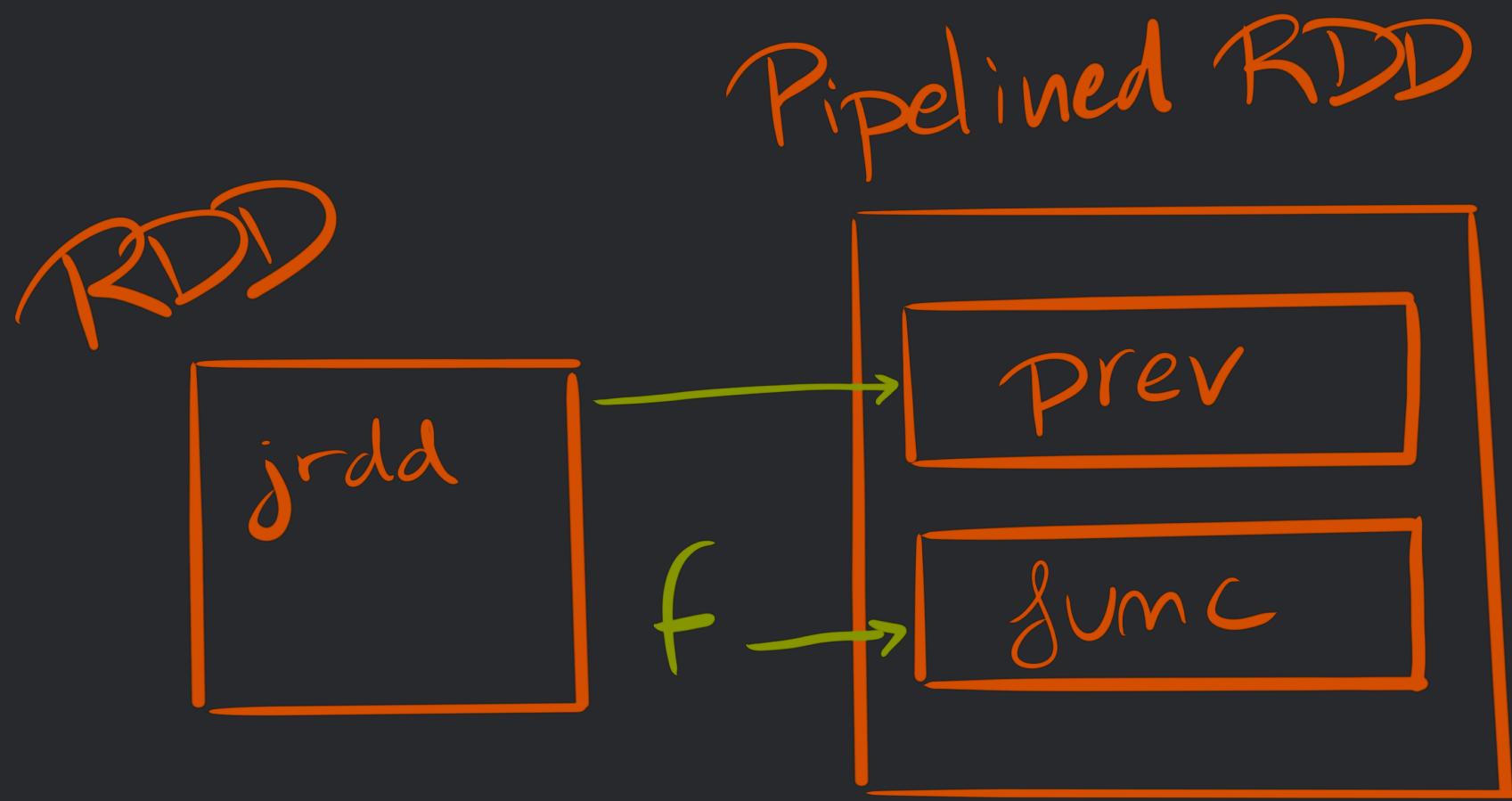
Pipelined RDD



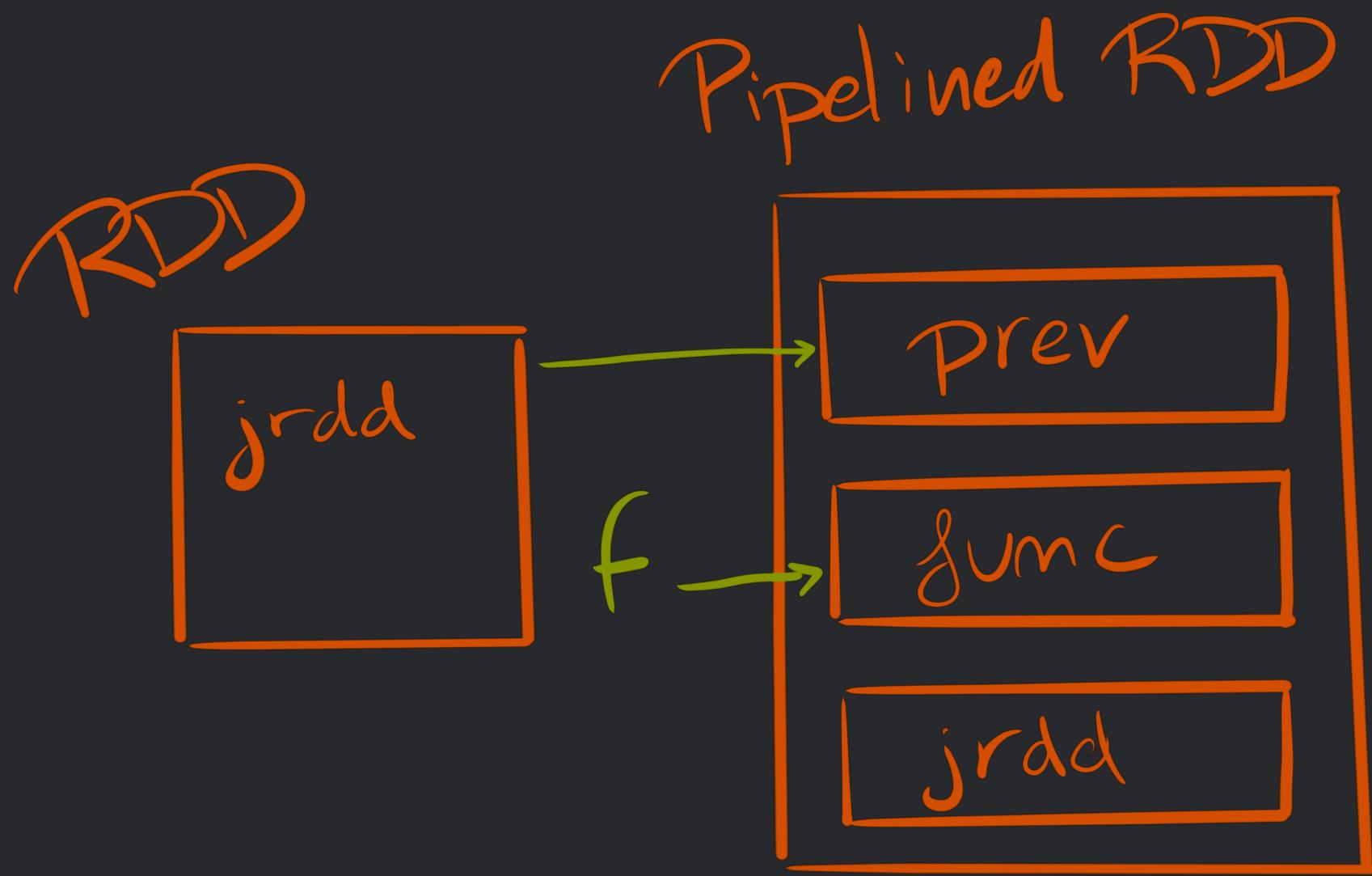
RDD's map will create a  
PipelinedRDD



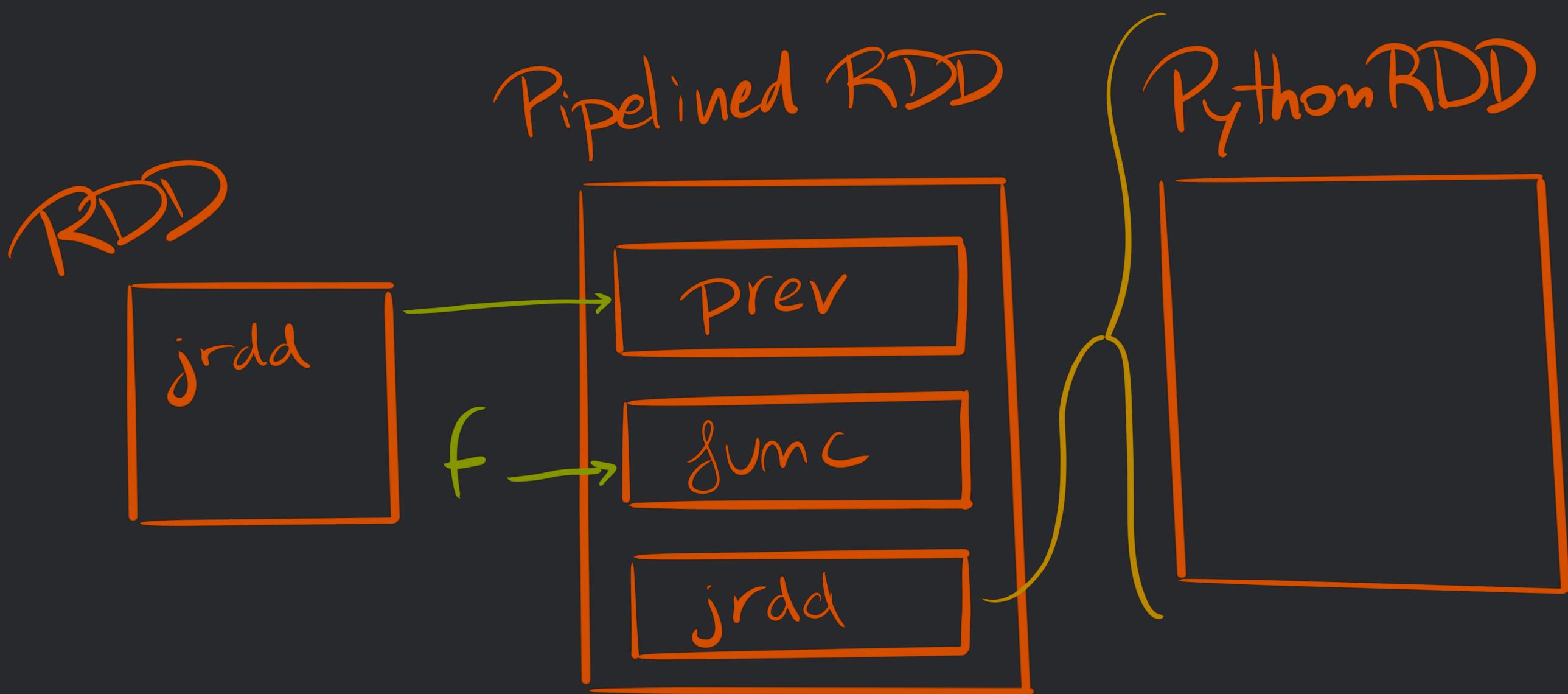
Will put the function in the  
func field



And will point prev to the  
RDD. Missing anything?



And now, what is the \_jrdd of the PipelinedRDD?



It's a **PythonRDD** (built via Py4J of course, this is in the Scala code), which is a sub-class of **RDD**

*RDD*

jRDD

Pipelined RDD

f

prev

func

jRDD

class in Scala

PythonRDD

*RDD*

jRDD

Pipelined RDD

f

prev

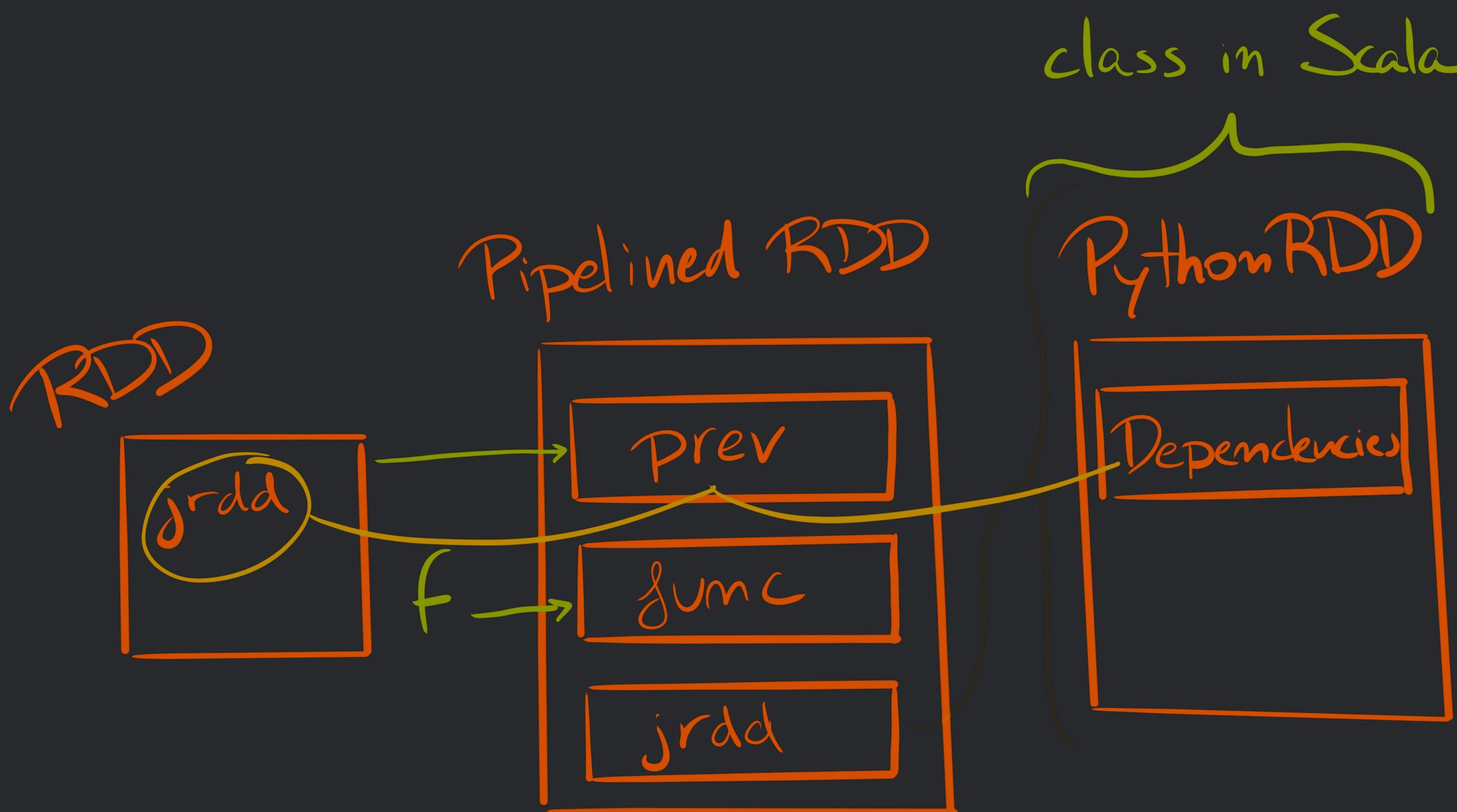
func

jRDD

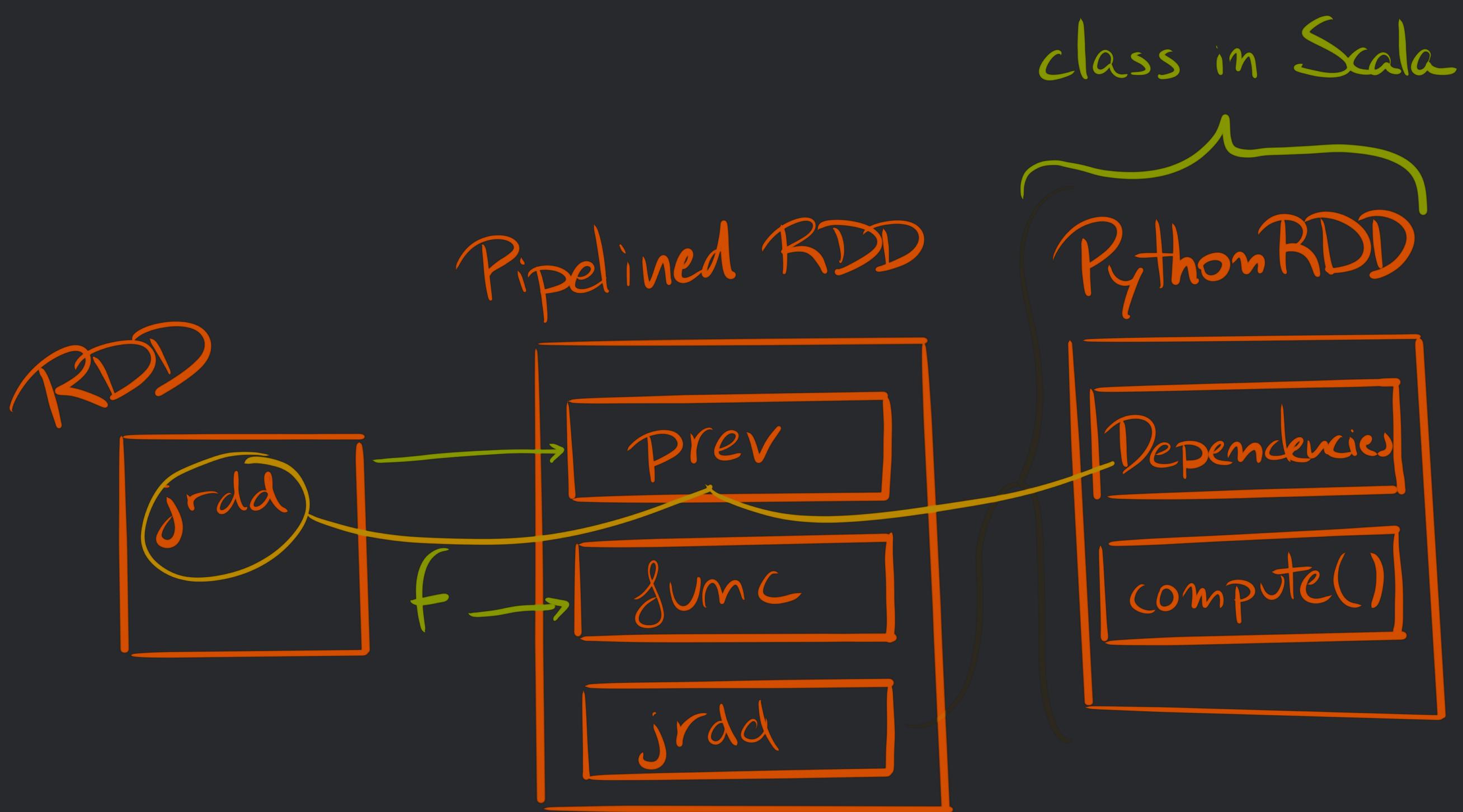
class in Scala

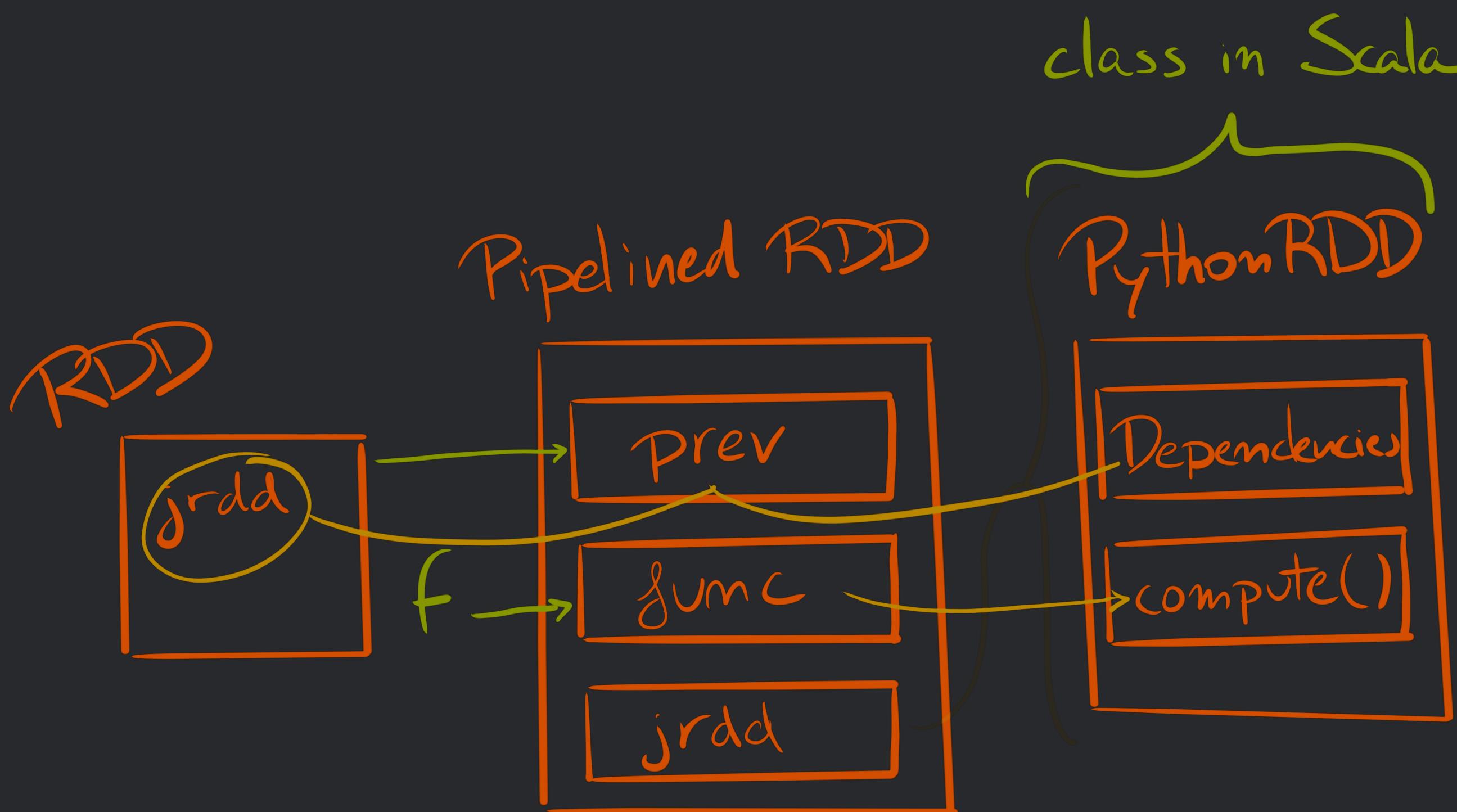
PythonRDD

Dependencies

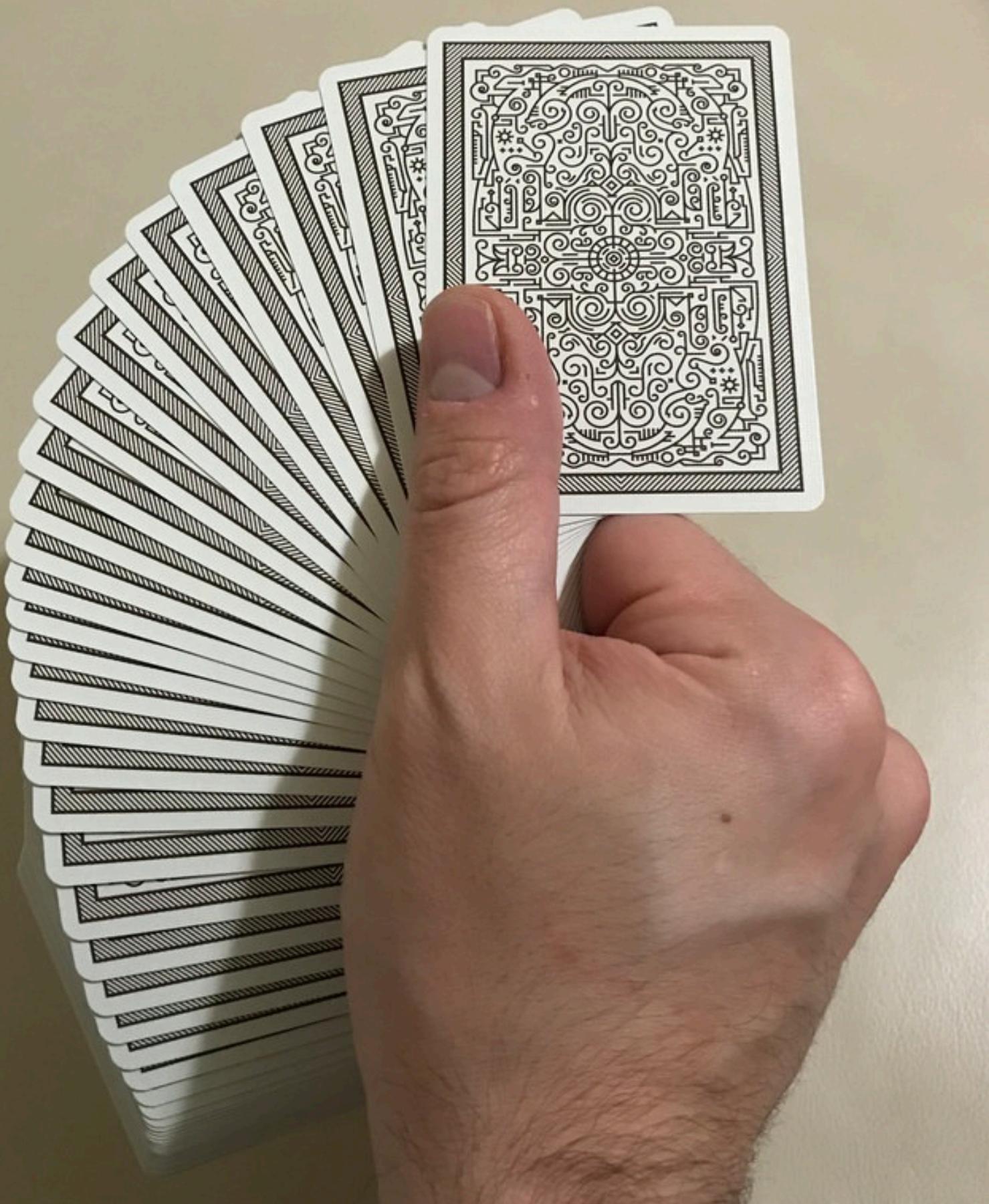


The dependencies of this RDD  
will point to the original `_jrdd`





The compute method will wrap the function defined in func



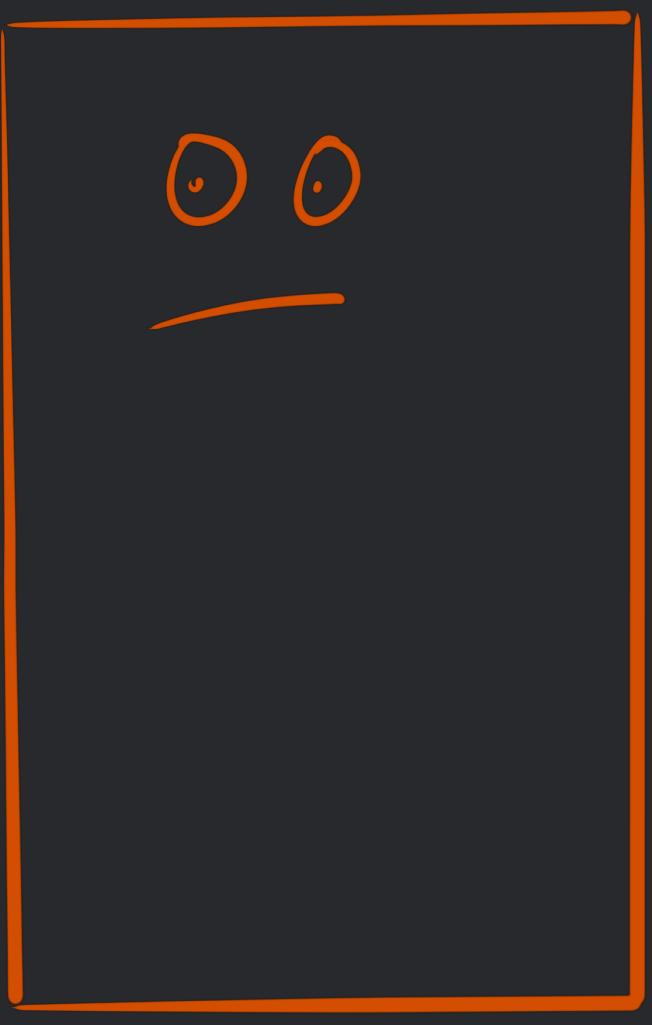
THE MAGIC IS  
IN  
compute

of PythonRDD It's where  
something gets eventually  
done to the RDD *in* Python

compute  
IS RUN ON EACH  
**EXECUTOR** AND STARTS  
A PYTHON **WORKER** VIA  
PythonRunner



It will send all includes,  
broadcasts, etc through the  
stream. And actually  
the order of the data sent is  
important

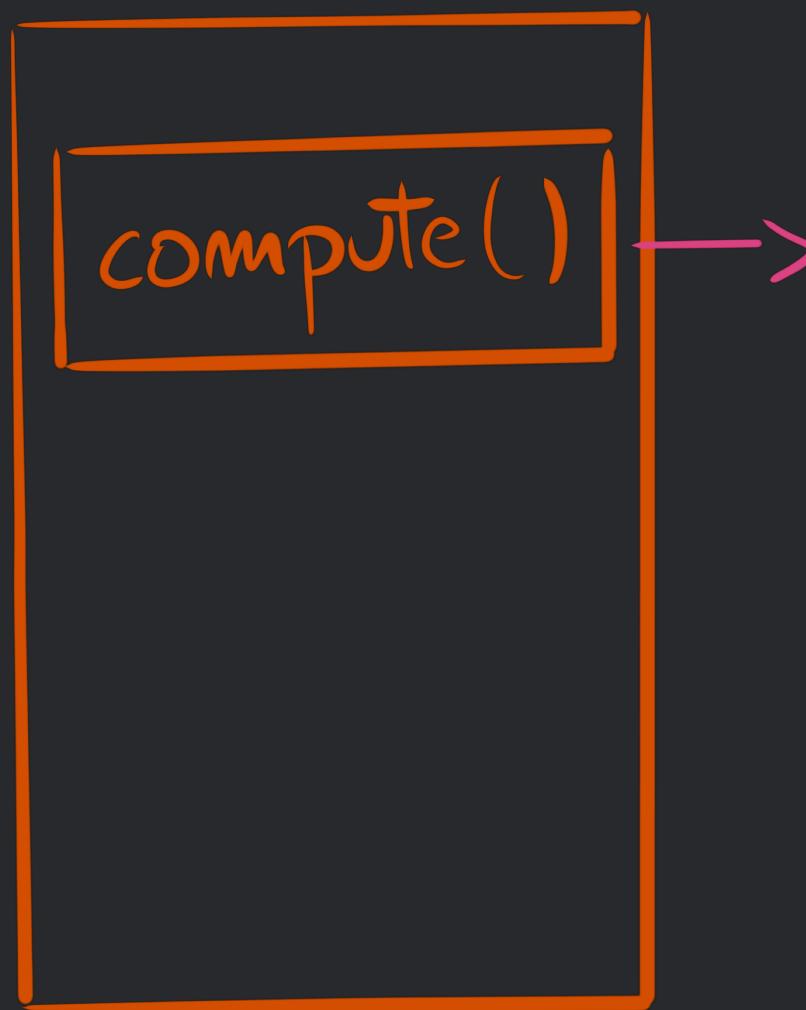


Python RDD

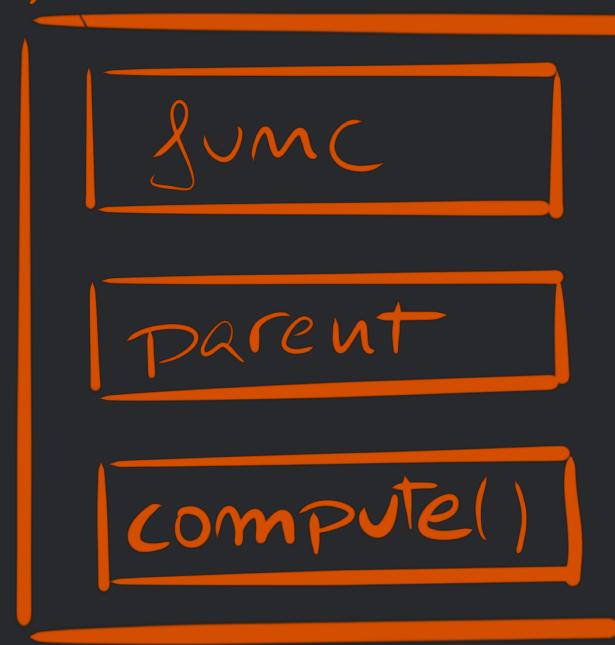


Scala!

PythonRDD

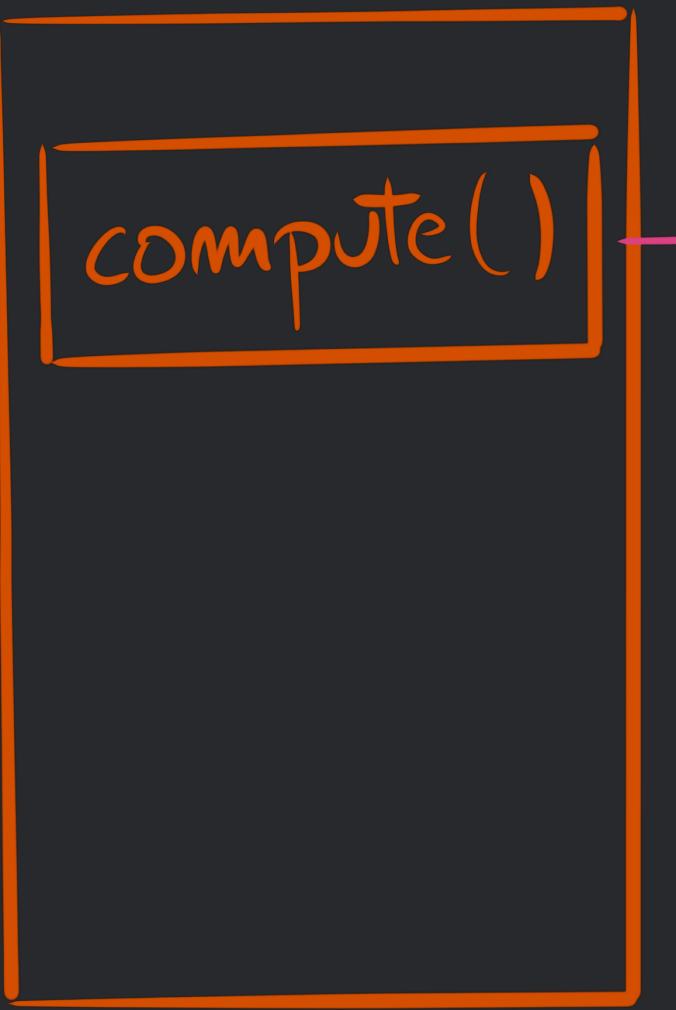


PythonRunner

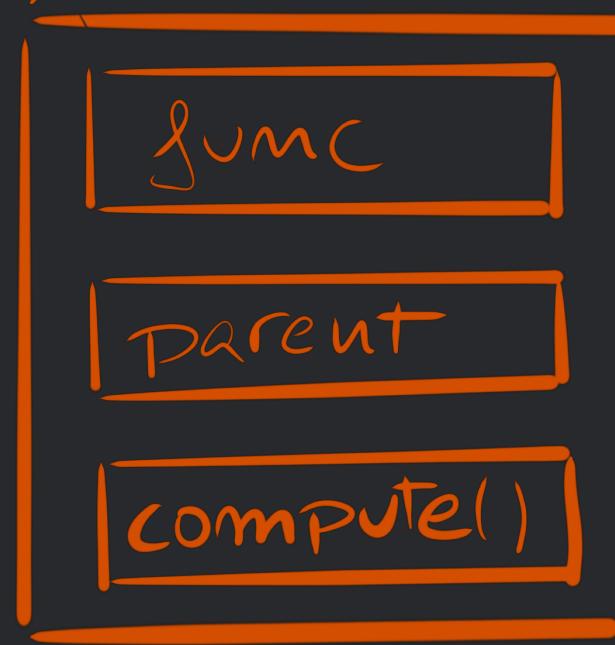


Scala!

PythonRDD



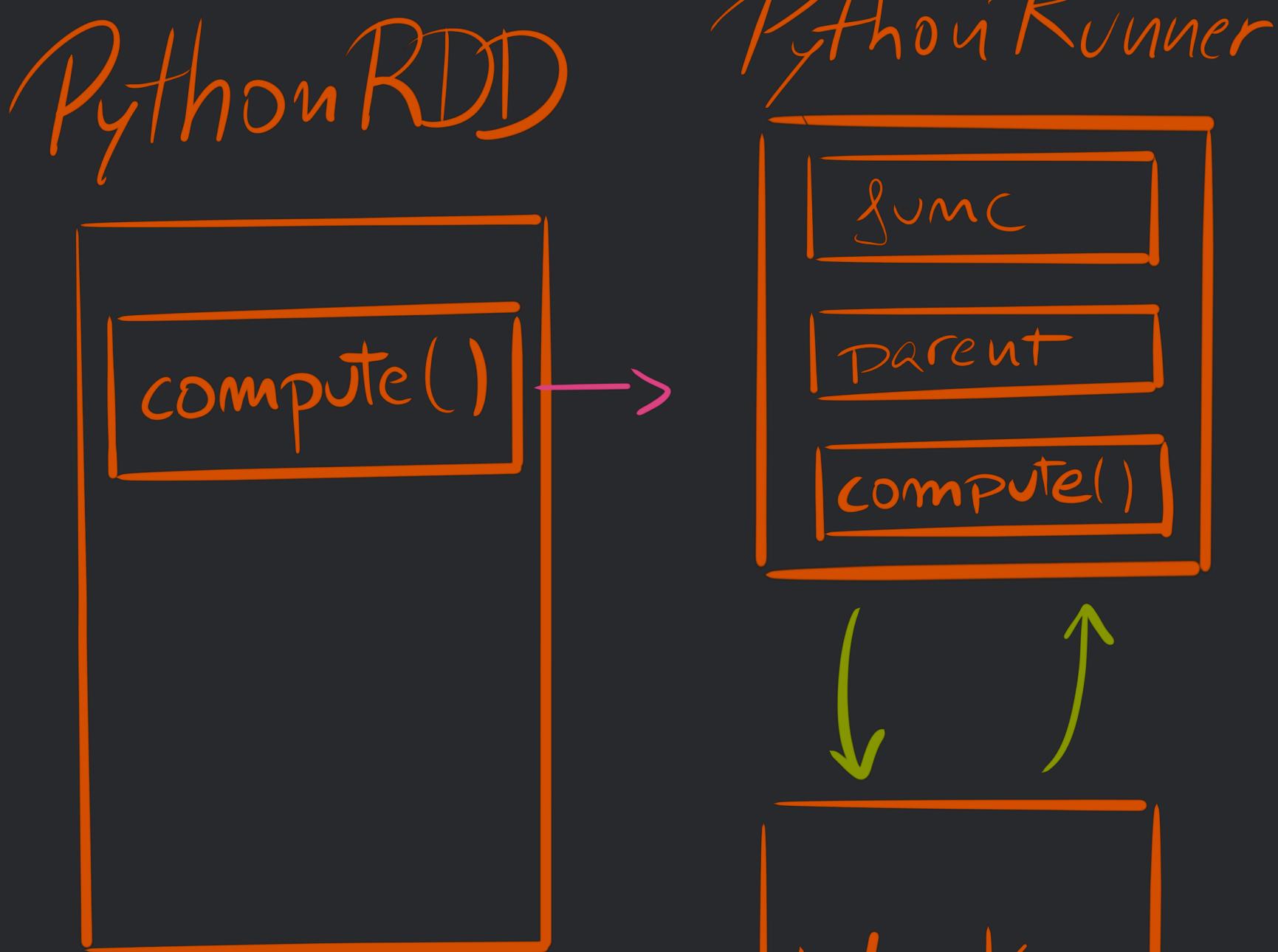
PythonRunner



Scala!



← Python!



Scala!

← Python!

# WORKERS ACT AS STANDALONE PROCESSORS OF STREAMS OF DATA

# WORKERS ACT AS STANDALONE PROCESSORS OF STREAMS OF DATA

- > CONNECTS BACK TO THE JVM THAT STARTED IT

# WORKERS ACT AS STANDALONE PROCESSORS OF STREAMS OF DATA

- > CONNECTS BACK TO THE JVM THAT STARTED IT
  - > LOAD INCLUDED PYTHON LIBRARIES

# WORKERS ACT AS STANDALONE PROCESSORS OF STREAMS OF DATA

- > CONNECTS BACK TO THE JVM THAT STARTED IT
  - > LOAD INCLUDED PYTHON LIBRARIES
- > DESERIALIZES THE PICKLED FUNCTION COMING FROM THE STREAM

# WORKERS ACT AS STANDALONE PROCESSORS OF STREAMS OF DATA

- > CONNECTS BACK TO THE JVM THAT STARTED IT
  - > LOAD INCLUDED PYTHON LIBRARIES
- > DESERIALIZES THE PICKLED FUNCTION COMING FROM THE STREAM
- > APPLIES THE FUNCTION TO THE DATA COMING FROM THE STREAM

# WORKERS ACT AS STANDALONE PROCESSORS OF STREAMS OF DATA

- > CONNECTS BACK TO THE JVM THAT STARTED IT
  - > LOAD INCLUDED PYTHON LIBRARIES
- > DESERIALIZES THE PICKLED FUNCTION COMING FROM THE STREAM
- > APPLIES THE FUNCTION TO THE DATA COMING FROM THE STREAM
  - > SENDS THE OUTPUT BACK

...

But, maybe you are missing  
something here, isn't Spark  
supposed to be pretty  
magic and plan and optimise  
stuff?

BUT... WASN'T SPARK  
MAGICALLY OPTIMISING  
EVERYTHING?

# YES, FOR SPARK DataFrame

You can think of DataFrames as RDDs which actually refer to tables. They have column names, and may have types for each column



SPARK WILL GENERATE  
A PLAN  
(A DIRECTED ACYCLIC GRAPH)  
TO COMPUTE THE  
RESULT

When you operate on  
DataFrames, plans are created  
magically. And actually it  
will generate a logical plan, an  
optimised logical plan, an  
execution plan...

AND THE PLAN WILL BE  
OPTIMISED USING  
**CATALYST**



The Catalyst optimiser.  
There's also a code generator  
in there (using Janino  
to compile Java code in real  
time) It prunes trees

DEPENDING ON THE FUNCTION, THE  
OPTIMISER WILL CHOOSE

PythonUDFRunner

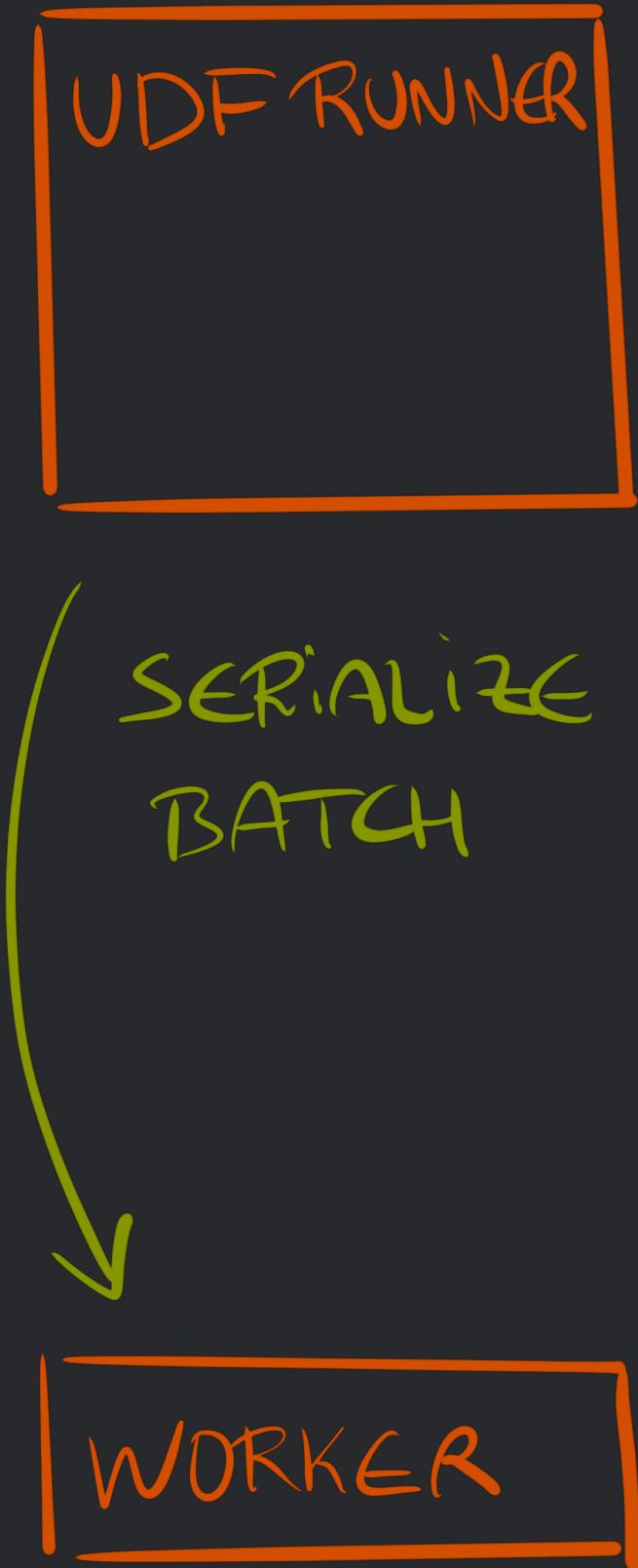
OR

PythonArrowRunner

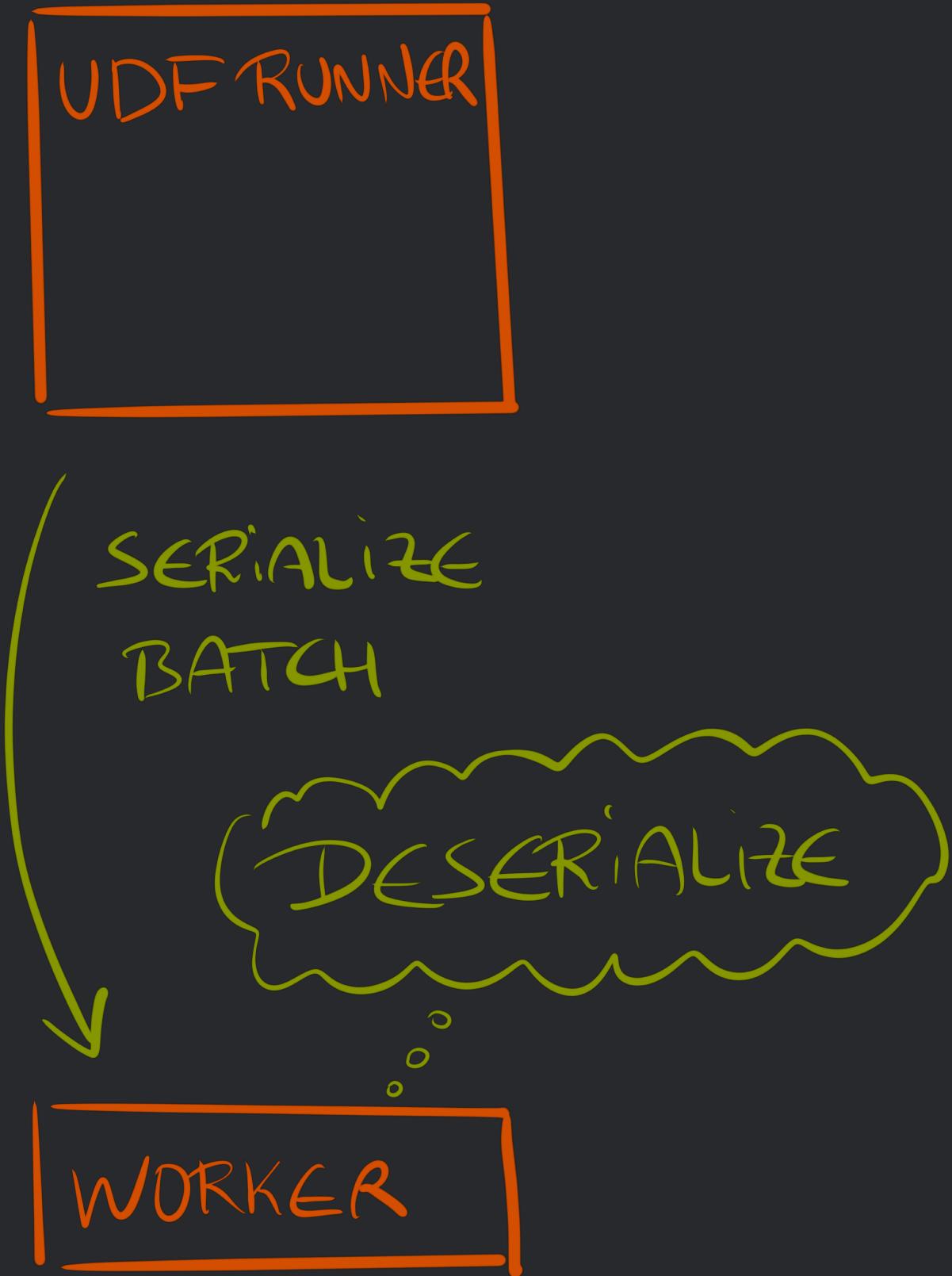
(BOTH EXTEND PythonRunner)

UDF RUNNER

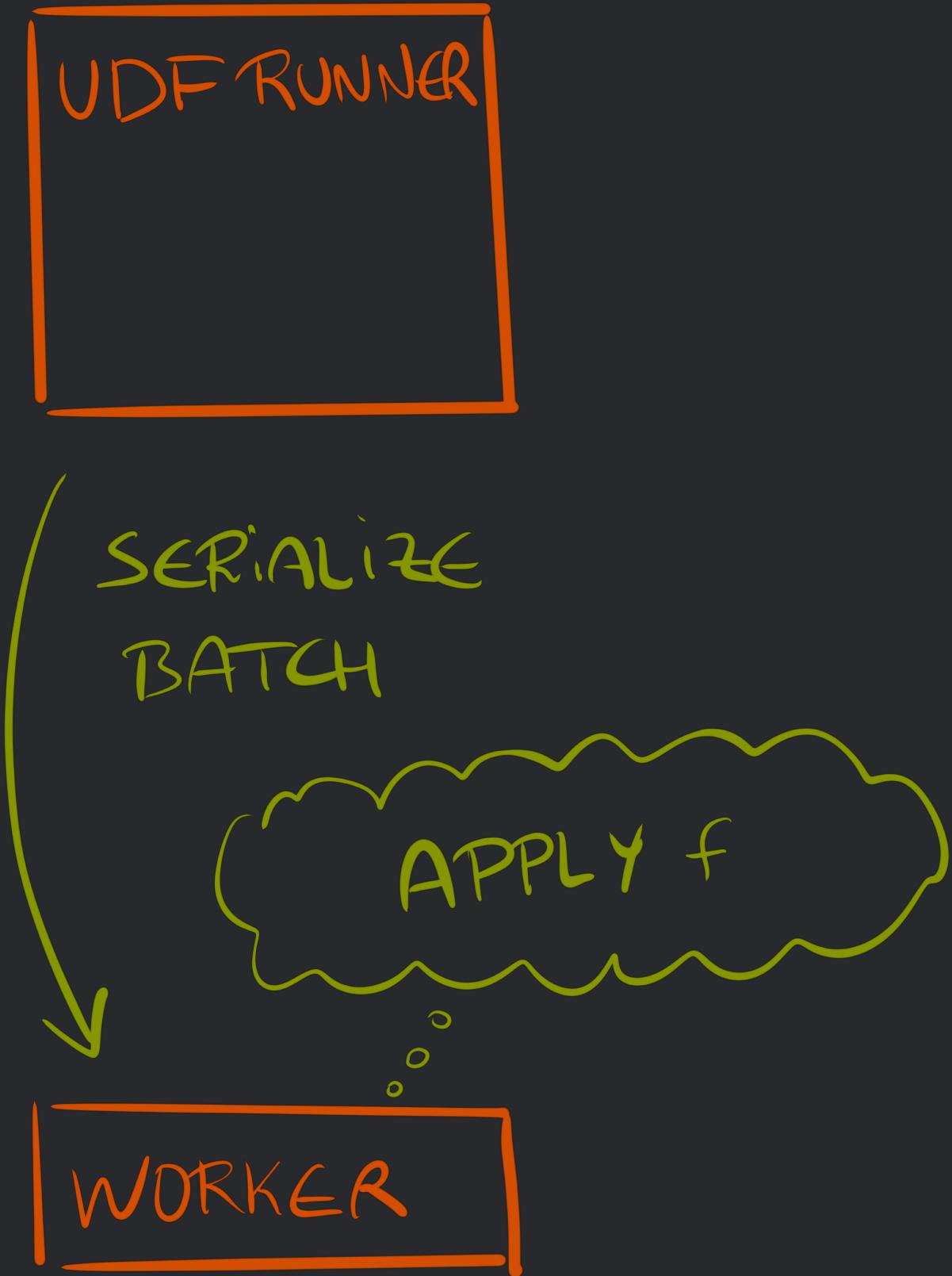
WORKER



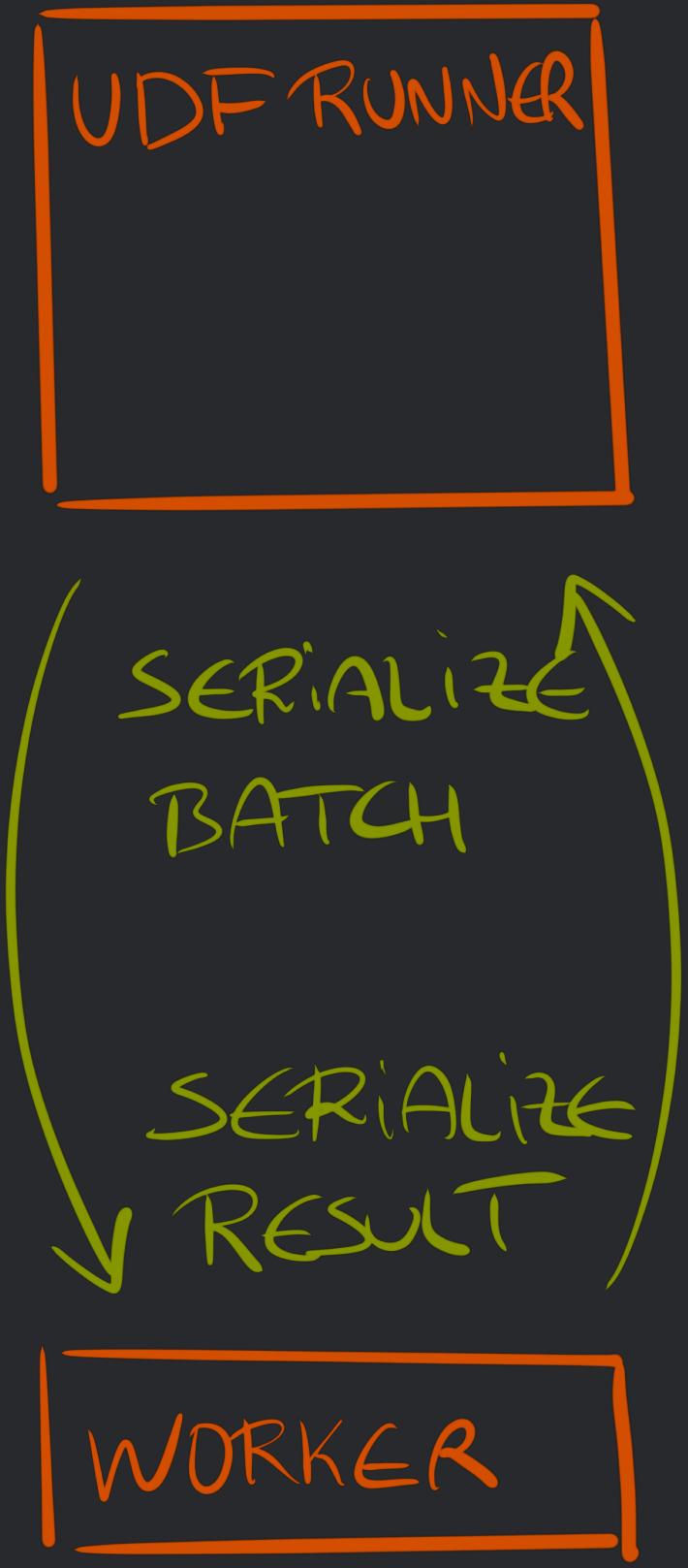
In the stream information about environment, length of data, versions, etc will be sent first. Then data will be serialized in batches (of 100 rows I think) using a Pickle implementation in Java (net.razorvine.pickle.{Pickler, Unpickler})



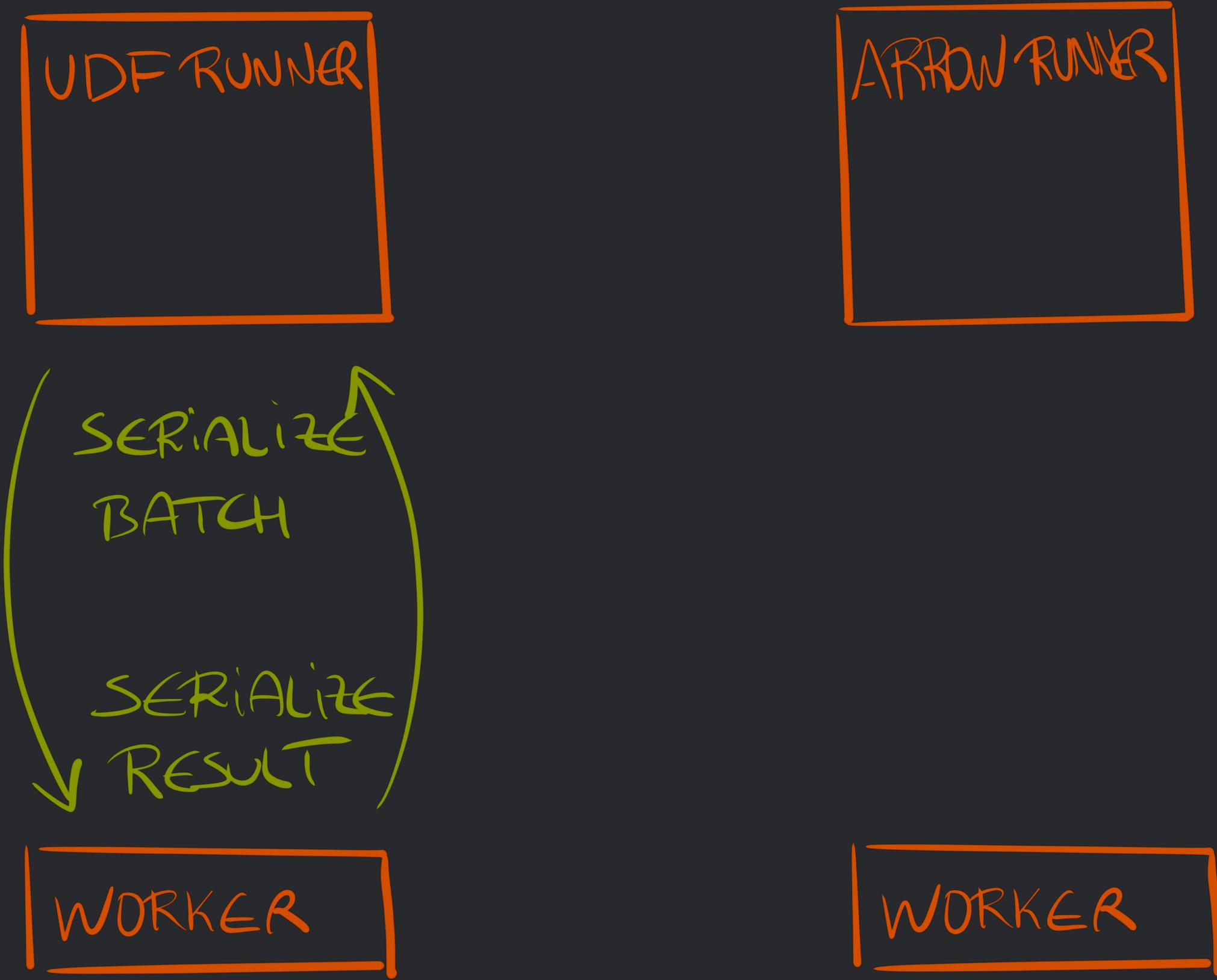
In Python land, after loading all the configuration and startup stuff, the data is unpickled



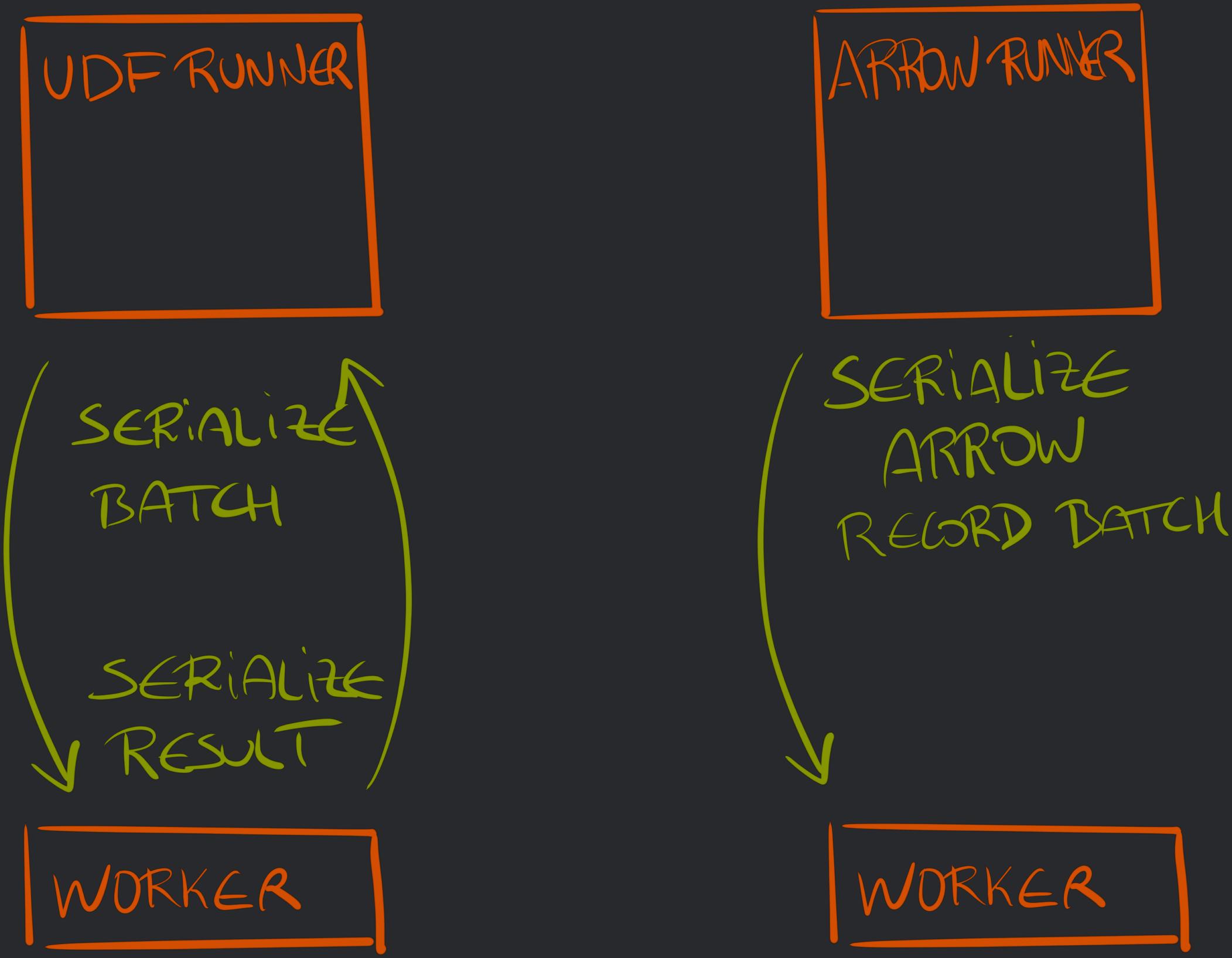
The stream is loaded as an iterator, and the function is applied to each item in the batch



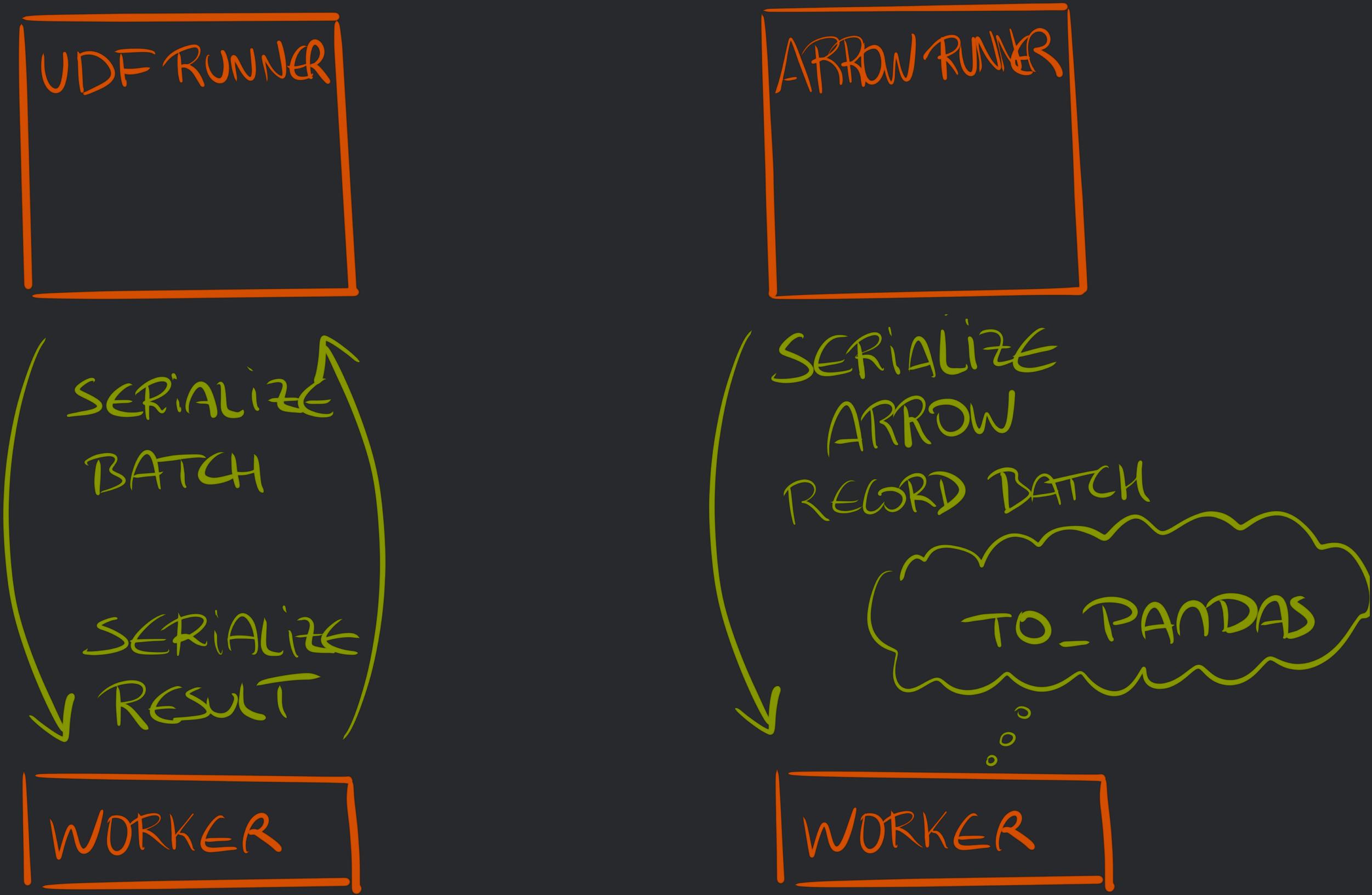
Then the data is serialized  
back to send to the JVM



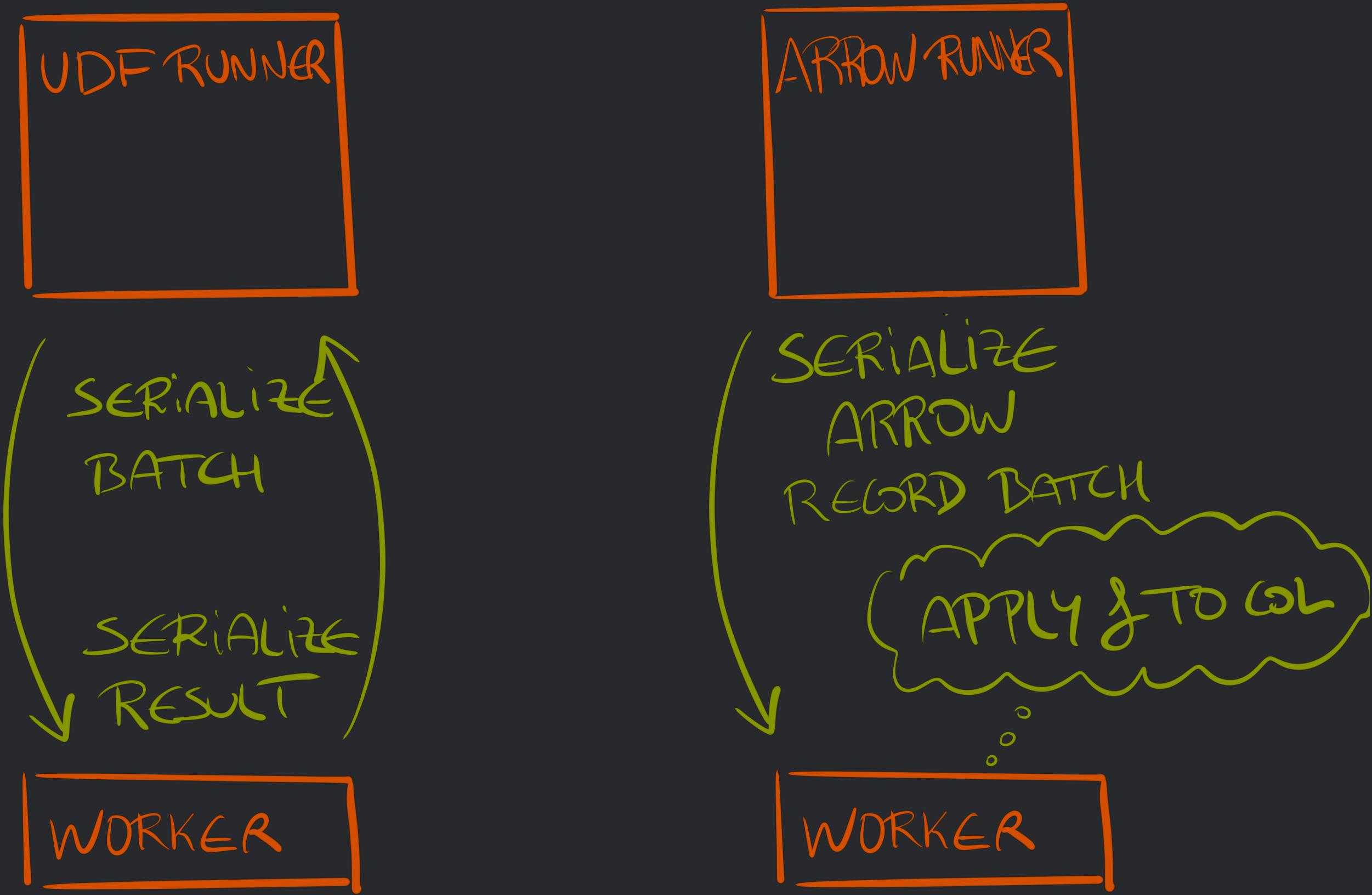
Arrow works a bit different, but  
not much



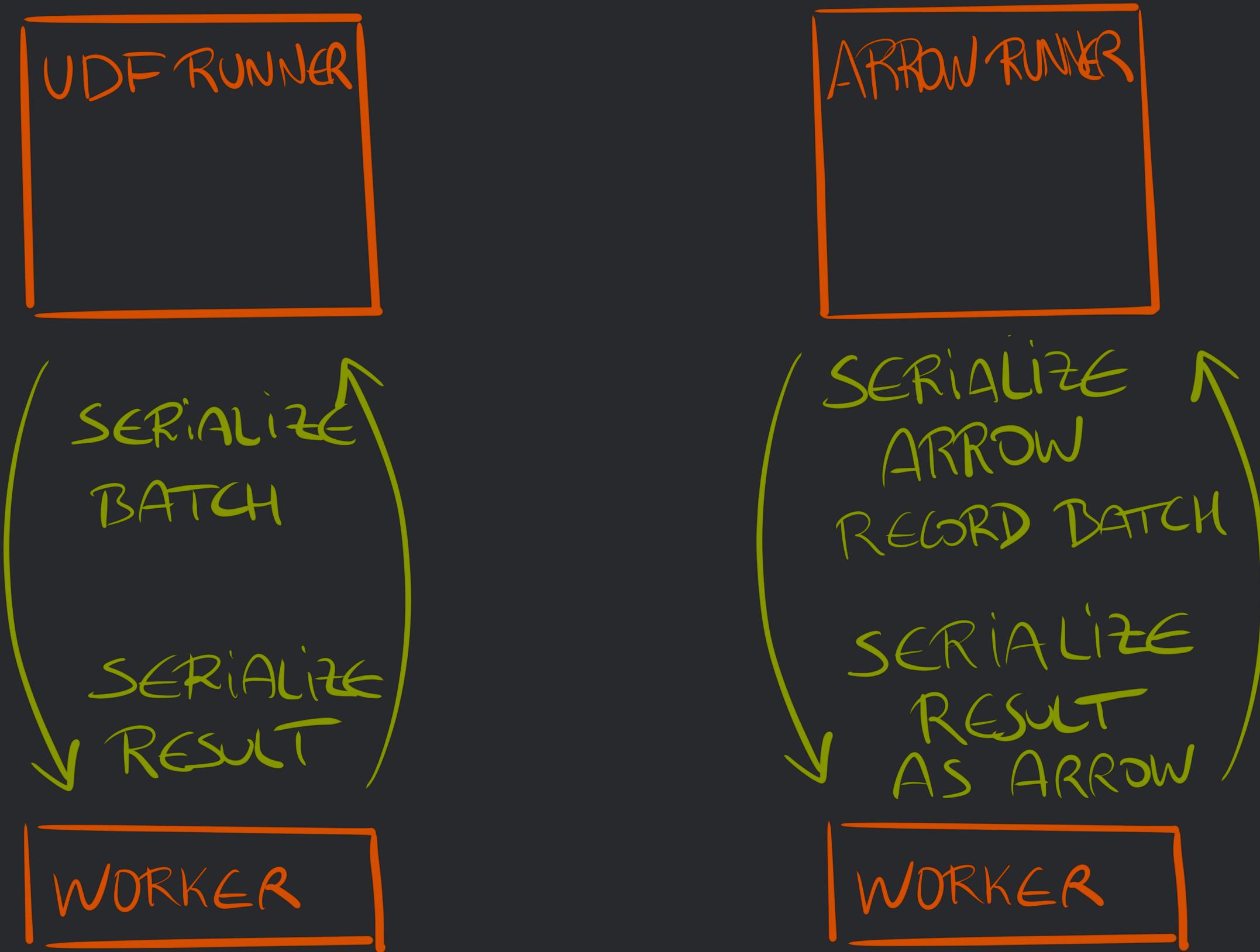
Data is sent as Arrow  
RecordBatch-es



Converting RecordBatches to a Pandas dataframe is essentially cost 0 (specially compared with pickling)



Applying a function to a column is super-fast, since it involves only running through all elements of the column. And the data has been loaded columnar already!



Data is sent back in Arrow columnar format, which again is pretty much cost-free in the Python side

**IF WE CAN DEFINE OUR FUNCTIONS  
USING PANDAS Series  
TRANSFORMATIONS WE CAN SPEED UP  
PYSPARK CODE FROM 3X TO 100X!**

See here: <https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html>

# RESOURCES

- > [SPARK DOCUMENTATION](#)
- > [HIGH PERFORMANCE SPARK BY HOLDEN KARAU](#)
- > [MASTERING APACHE SPARK 2.3 BY JACEK LASKOWSKI](#)
- > [SPARK'S GITHUB](#)
- > [BECOME A CONTRIBUTOR](#)

# QUESTIONS?



**THANKS!**

# FURTHER REFERENCES

# ARROW

[ARROW'S HOME](#)

[ARROW'S GITHUB](#)

[ARROW SPEED TESTS](#)

[ARROW TO PANDAS CONVERSION SPEED](#)

[STREAMING COLUMNAR DATA WITH APACHE ARROW](#)

[WHY PANDAS USERS SHOULD BE EXCITED BY APACHE ARROW](#)

[ARROW-PANDAS COMPATIBILITY LAYER CODE](#)

[ARROW TABLE CODE](#)

[PYARROW IN-MEMORY DATA MODEL](#)

# PANDAS

[PANDAS' HOME](#)

[PANDAS' GITHUB](#)

[IDIOMATIC PANDAS GUIDE](#)

[PANDAS INTERNALS CODE](#)

[PANDAS INTERNALS DESIGN](#)

[DEMYSTIFYING PANDAS' INTERNALS \(TALK BY MARC GARCIA\)](#)

[MEMORY LAYOUT OF MULTIDIMENSTIONAL ARRAYS \(NUMPY\)](#)

# SPARK/PYSPARK

PYSPARK SERIALIZERS CODE

FIRST STEPS TO USING ARROW (ONLY IN THE PYSPARK DRIVER)

SPEEDING UP PYSPARK WITH APACHE ARROW

ORIGINAL JIRA ISSUE: VECTORIZED UDFS IN SPARK

INITIAL DOC DRAFT

BLOG POST BY BRYAN CUTLER (LEADER FOR THE VEC UDFS PR)

INTRODUCING PANDAS UDF FOR PYSPARK

ORG.APACHE.SPARK.SQL.VECTORIZED

PY4J

PY4J'S HOME  
PY4J'S GITHUB  
REFLECTION ENGINE

**EOF**