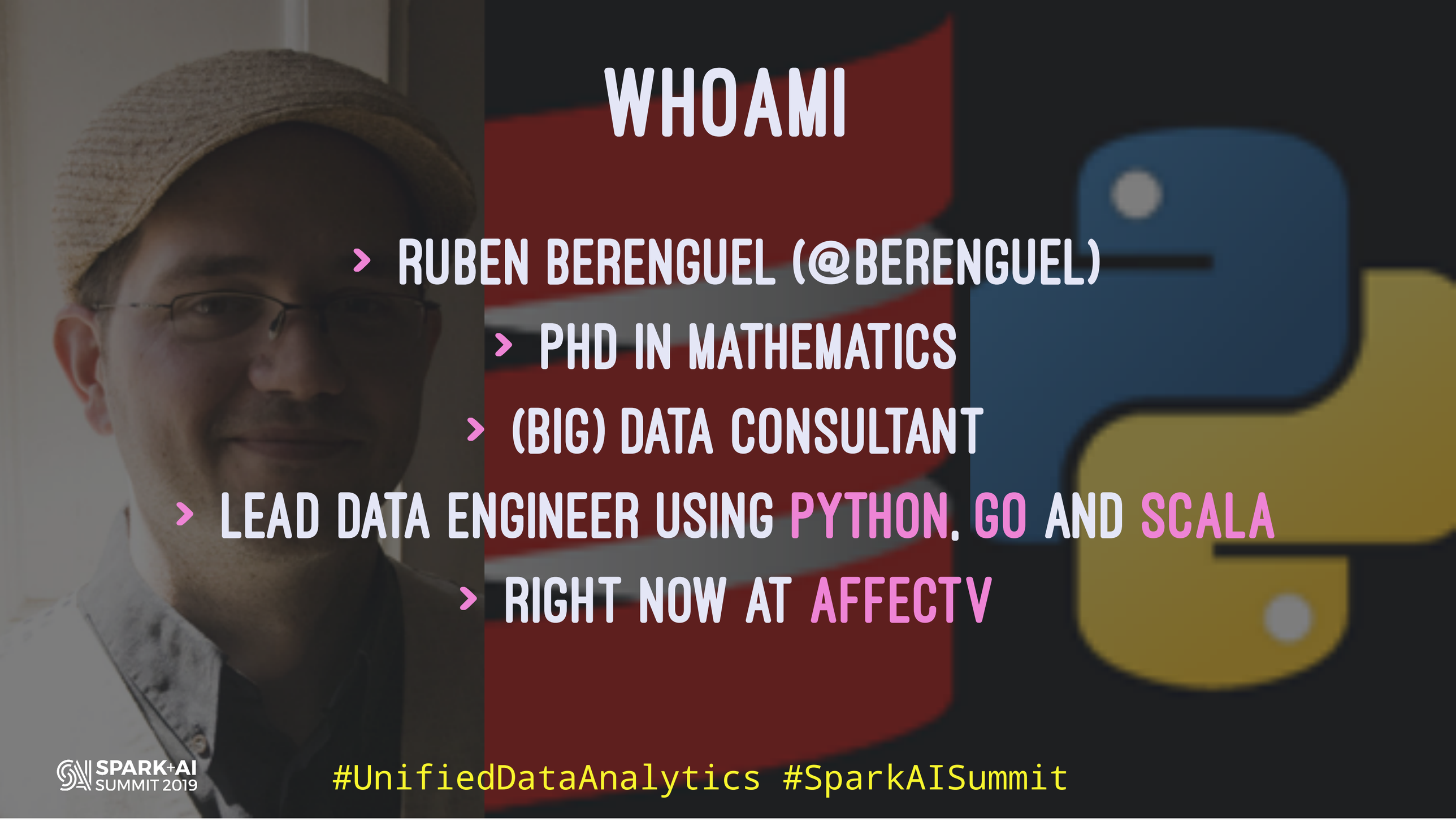
Test footnote, hello

I will start with what is `Pandas`, what is `Arrow` and how they relate. Then explain a bit what is `Spark` and how it works (I'll try to be fast here) and then how `PySpark` works. Finally, I'll cover why `Arrow` speeds up processes.

Test footnote 2, hello there



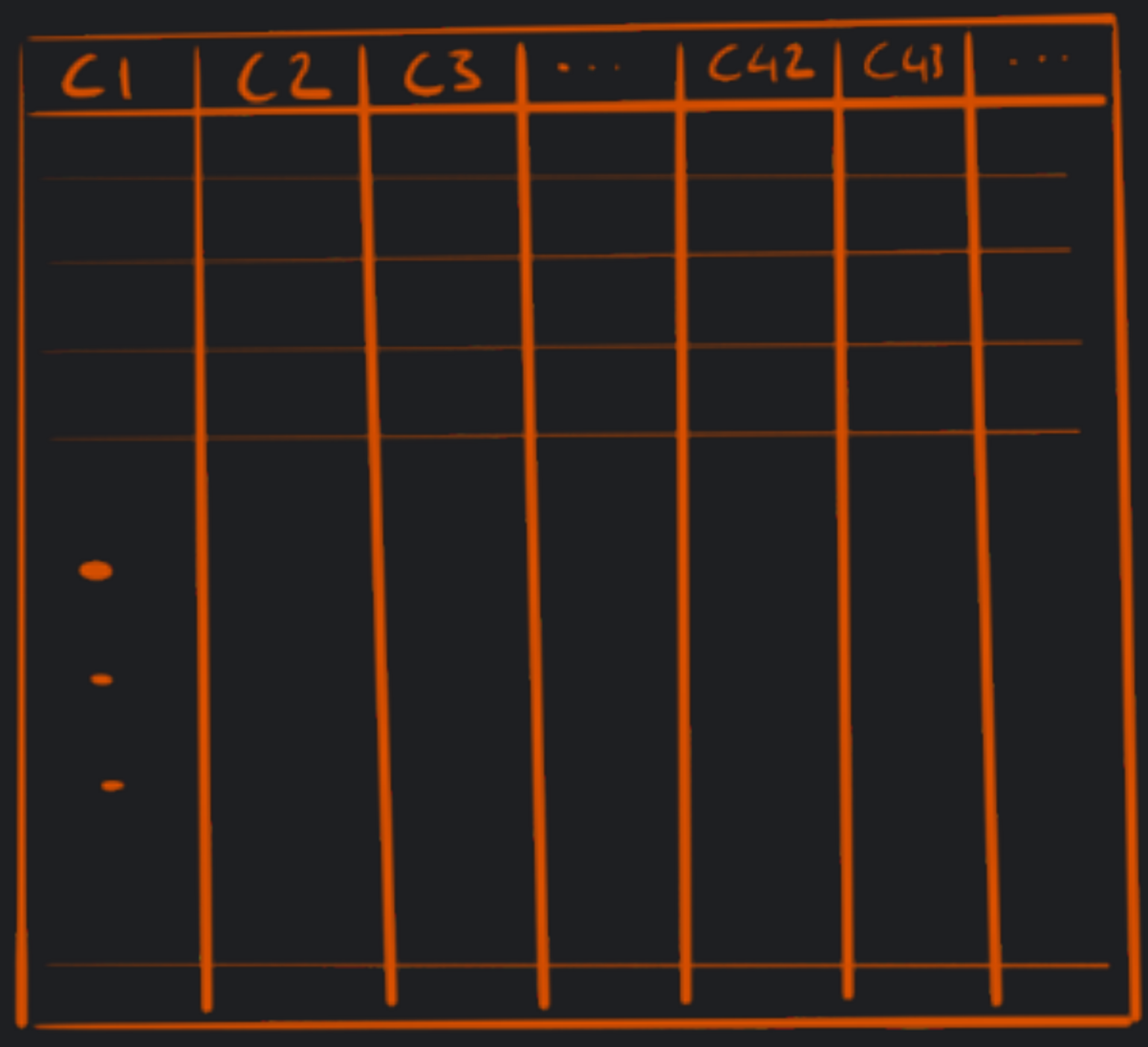
WHOAMI

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

WHAT IS PANDAS?

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

(third bullet: Have you ever asked yourself why "Columnar database" is so trendy?)

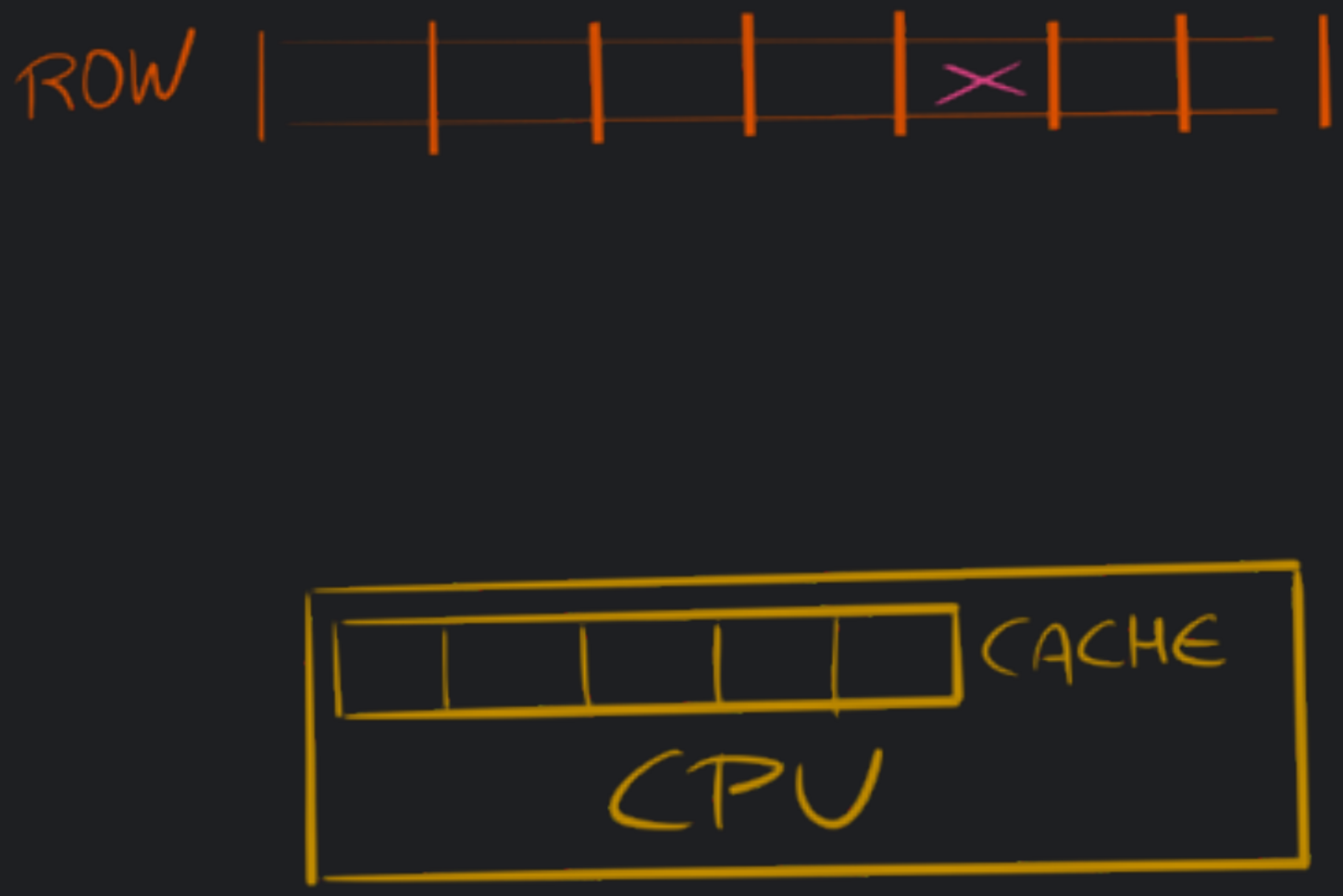


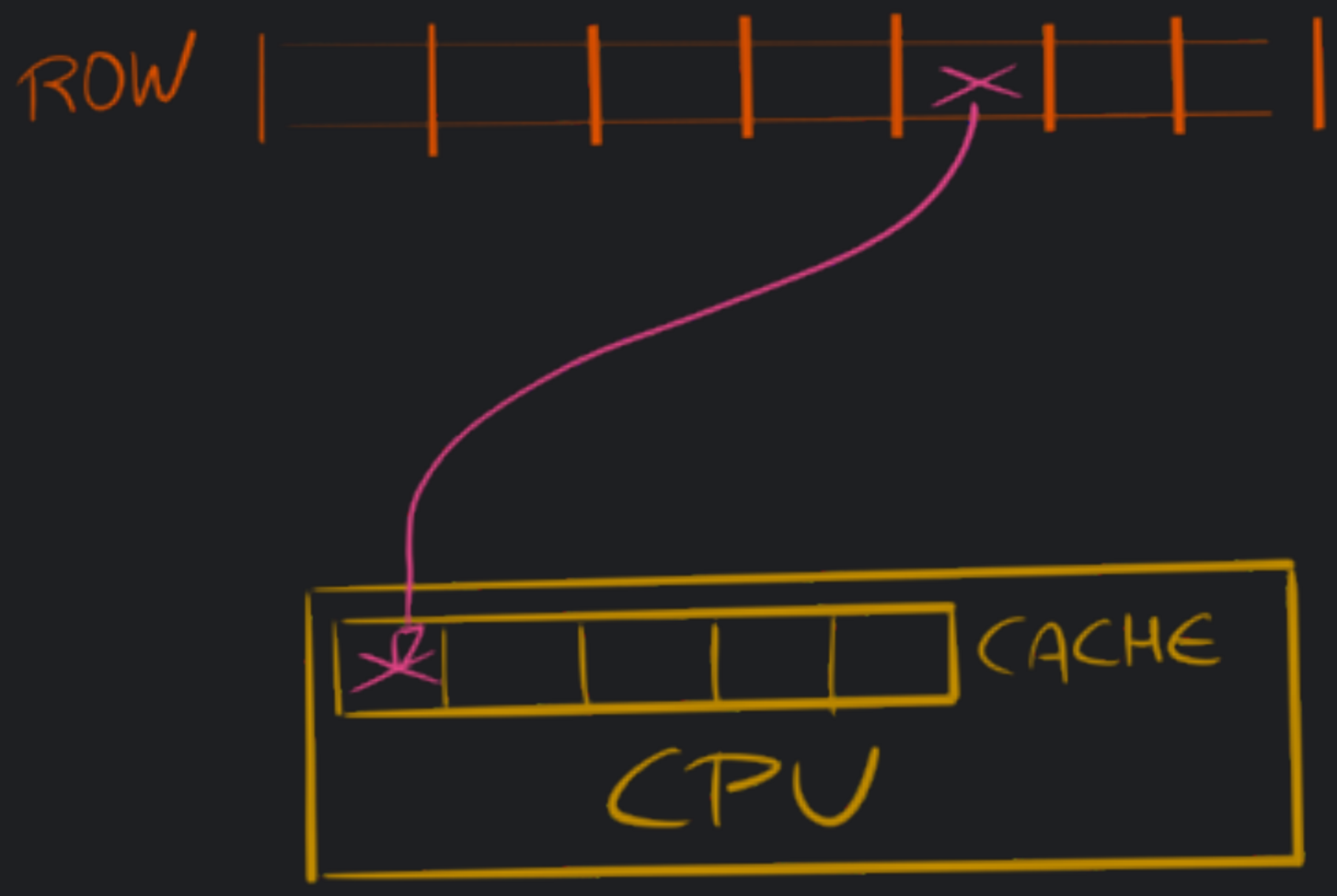
So, we have a table with many columns

And want the sum of one

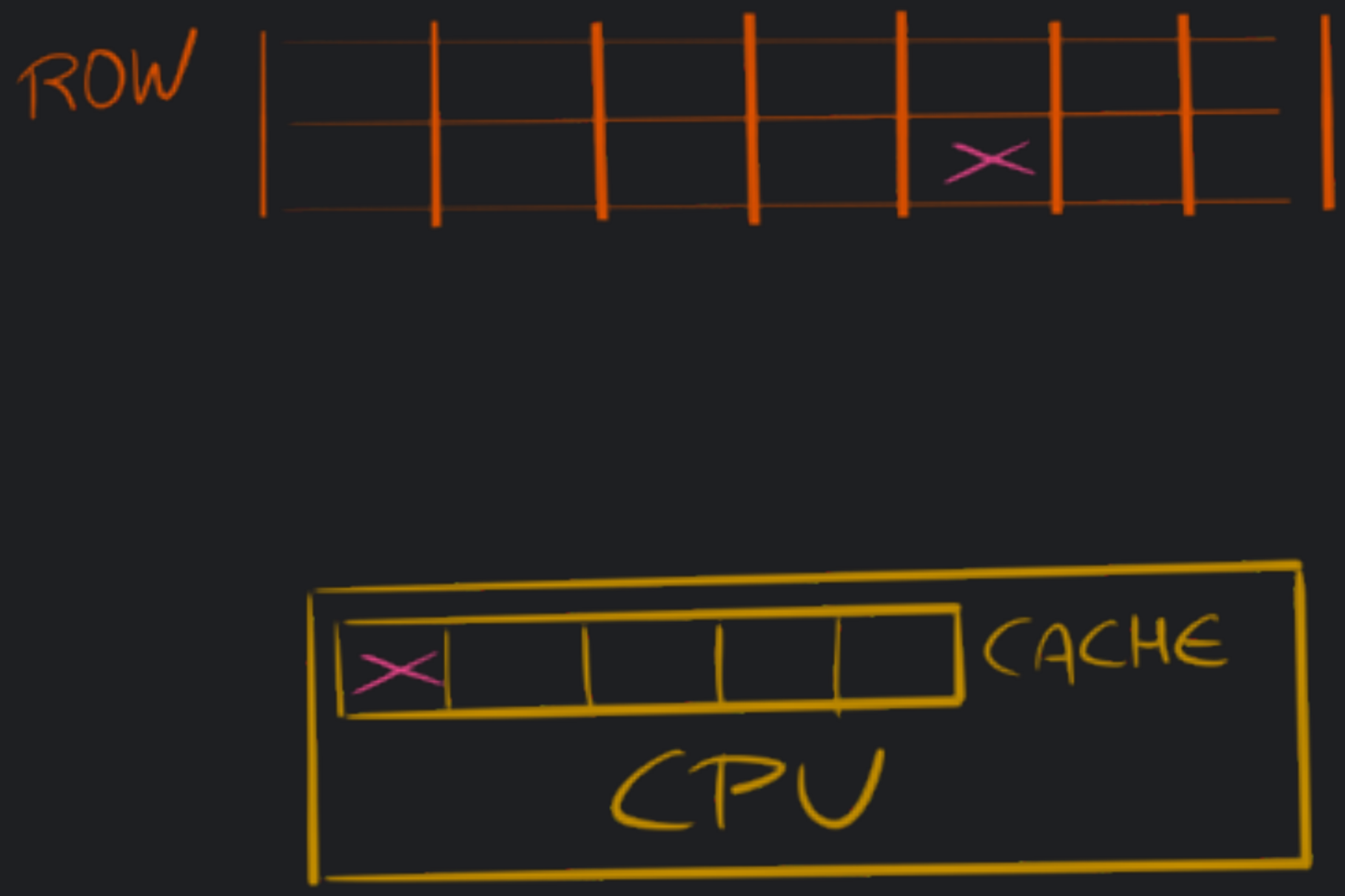
Let's isolate it at the level of *one row* and *one CPU*

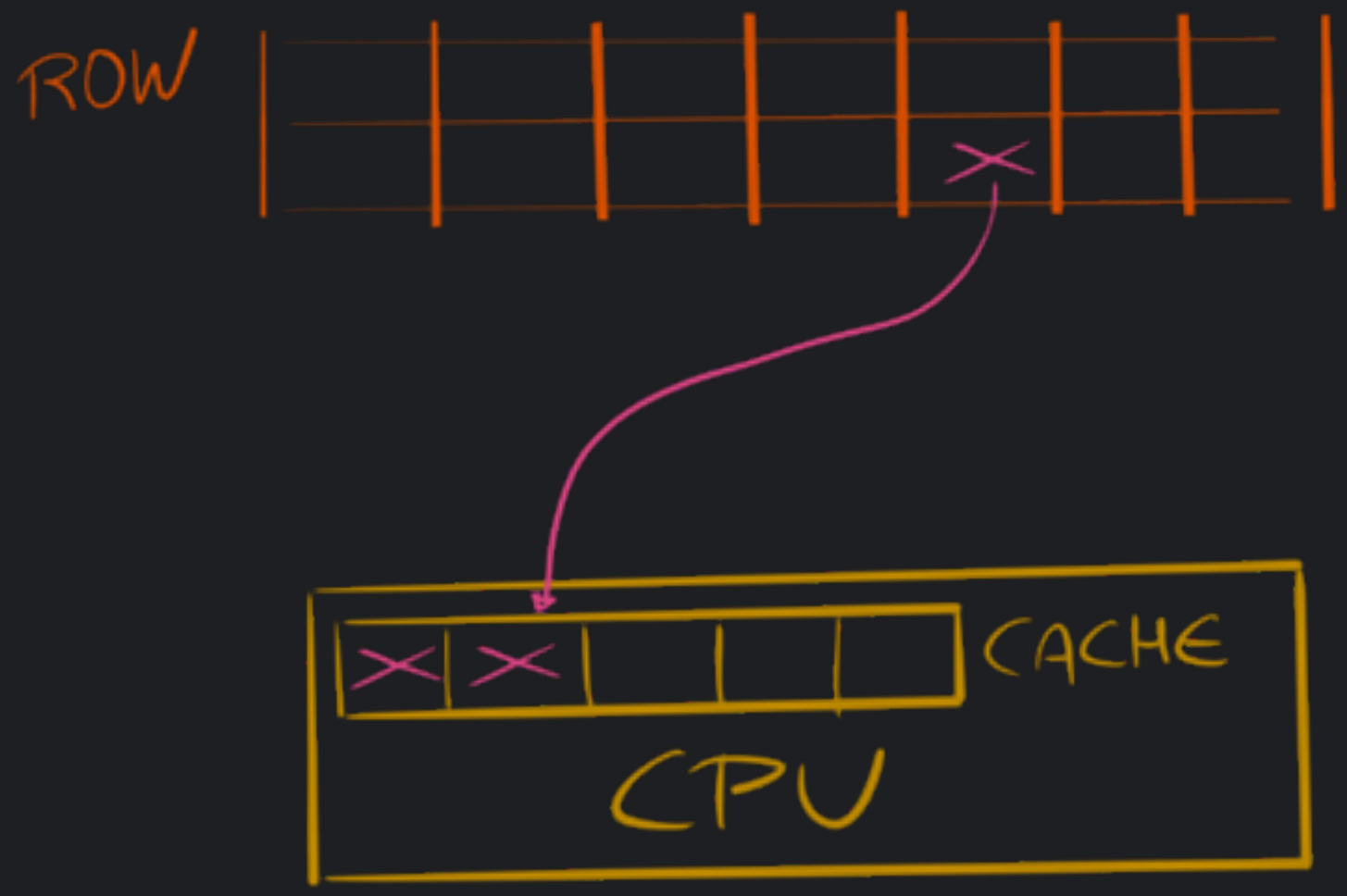
Moving data from memory (or disk) to a cache line can be done in batches. If we have the data ordered by rows in memory or disk...


ROW
CACHE
CPU

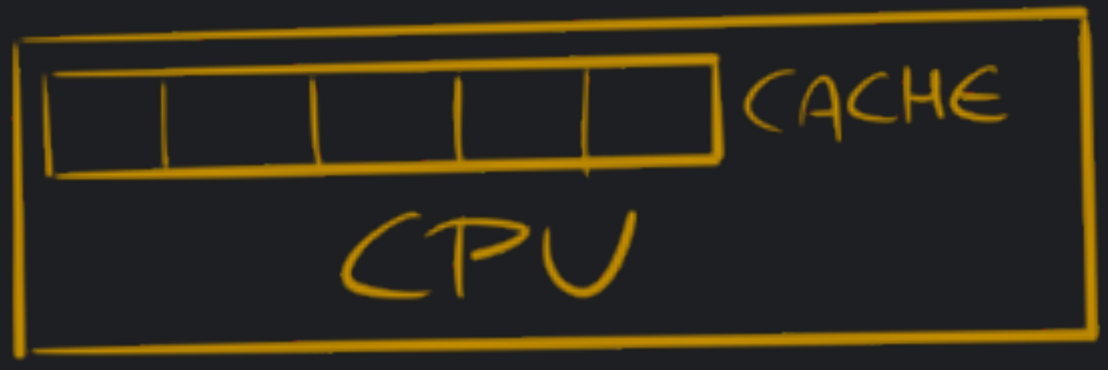


We move items one by one

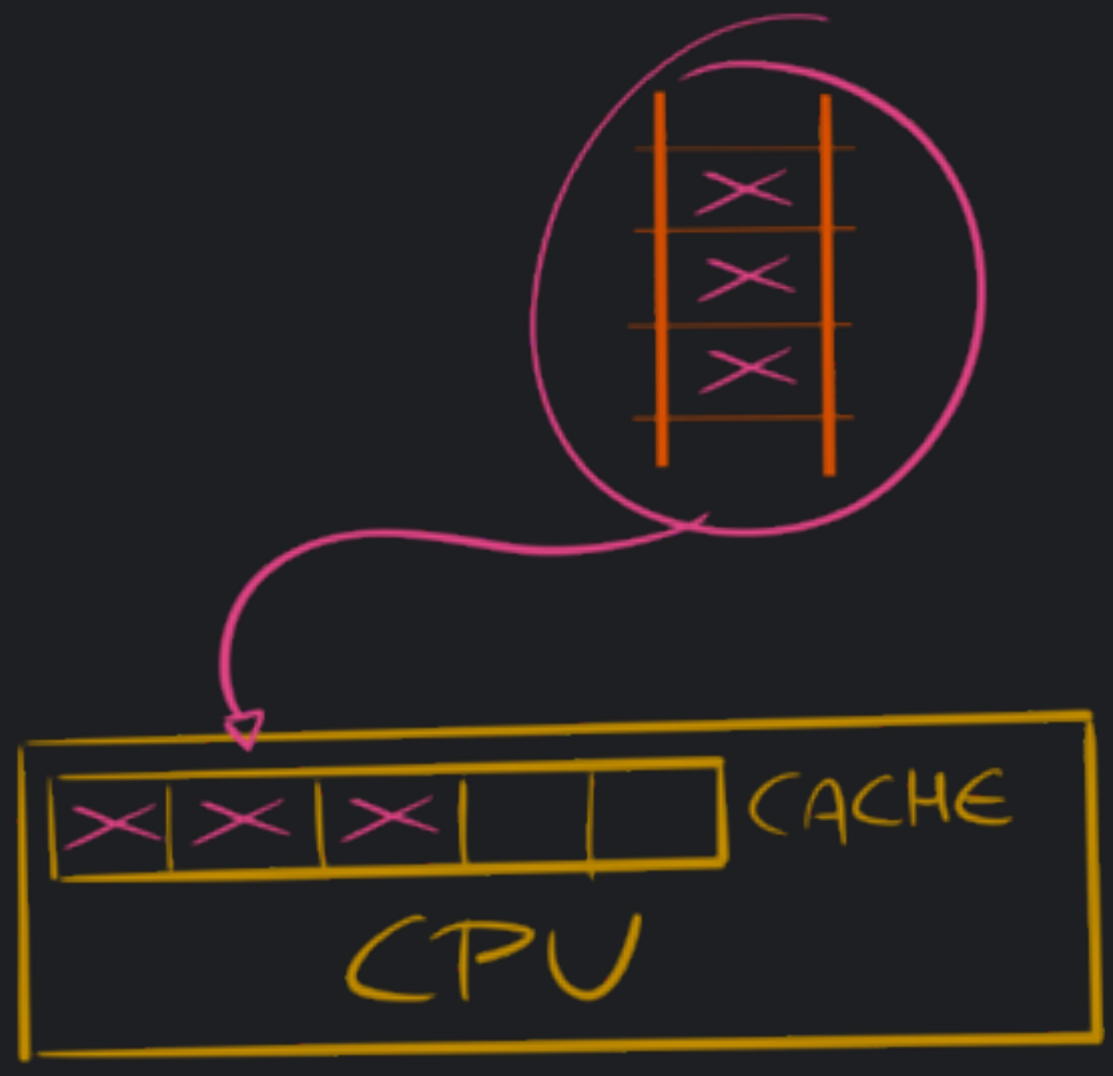

ROW
CACHE
CPU



Looks slow



If data is stored by column, we can load the whole column and operate straight with it with less memory fetching

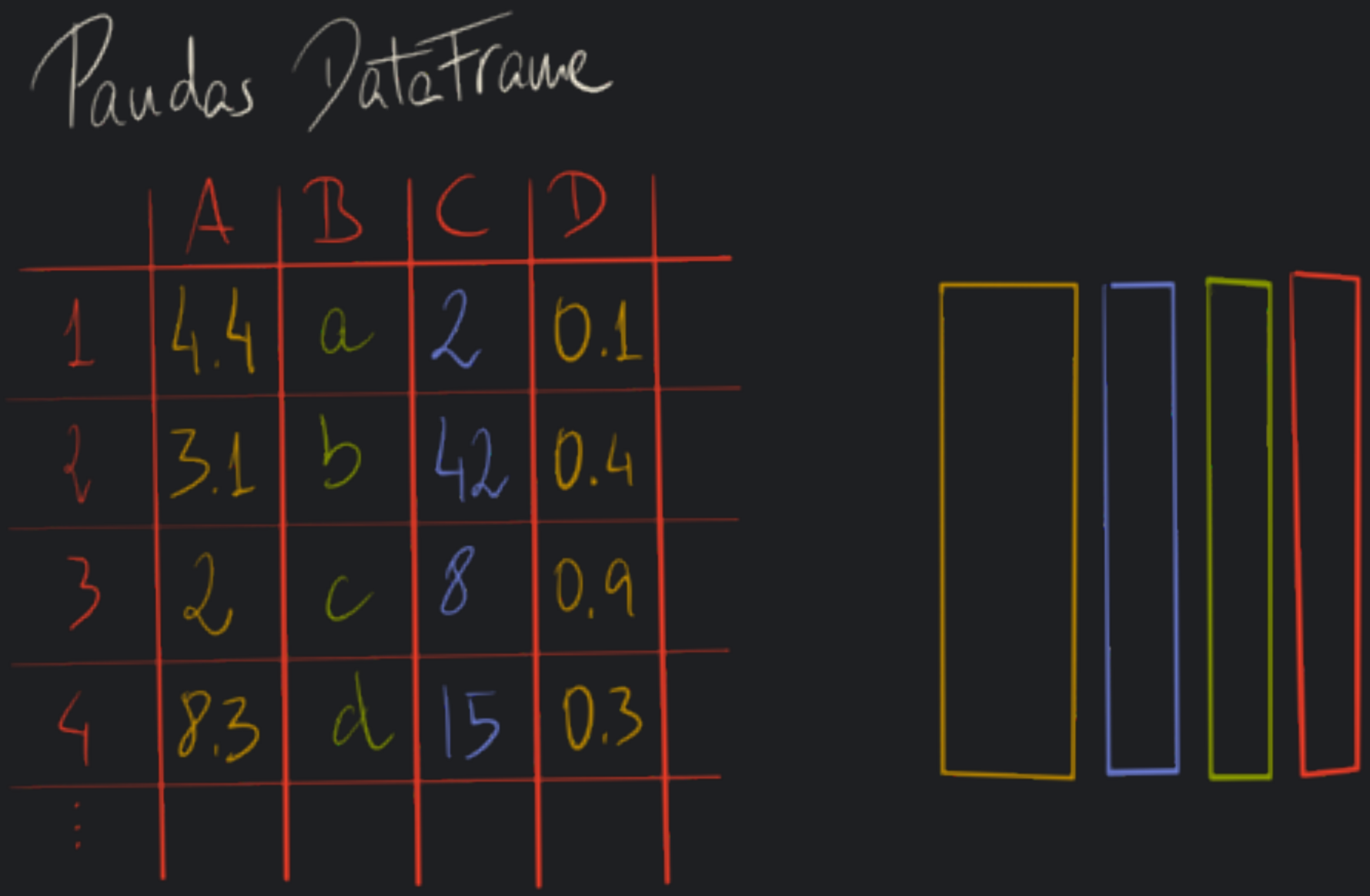

CACHE
CPU

HOW DOES PANDAS MANAGE COLUMNAR DATA?

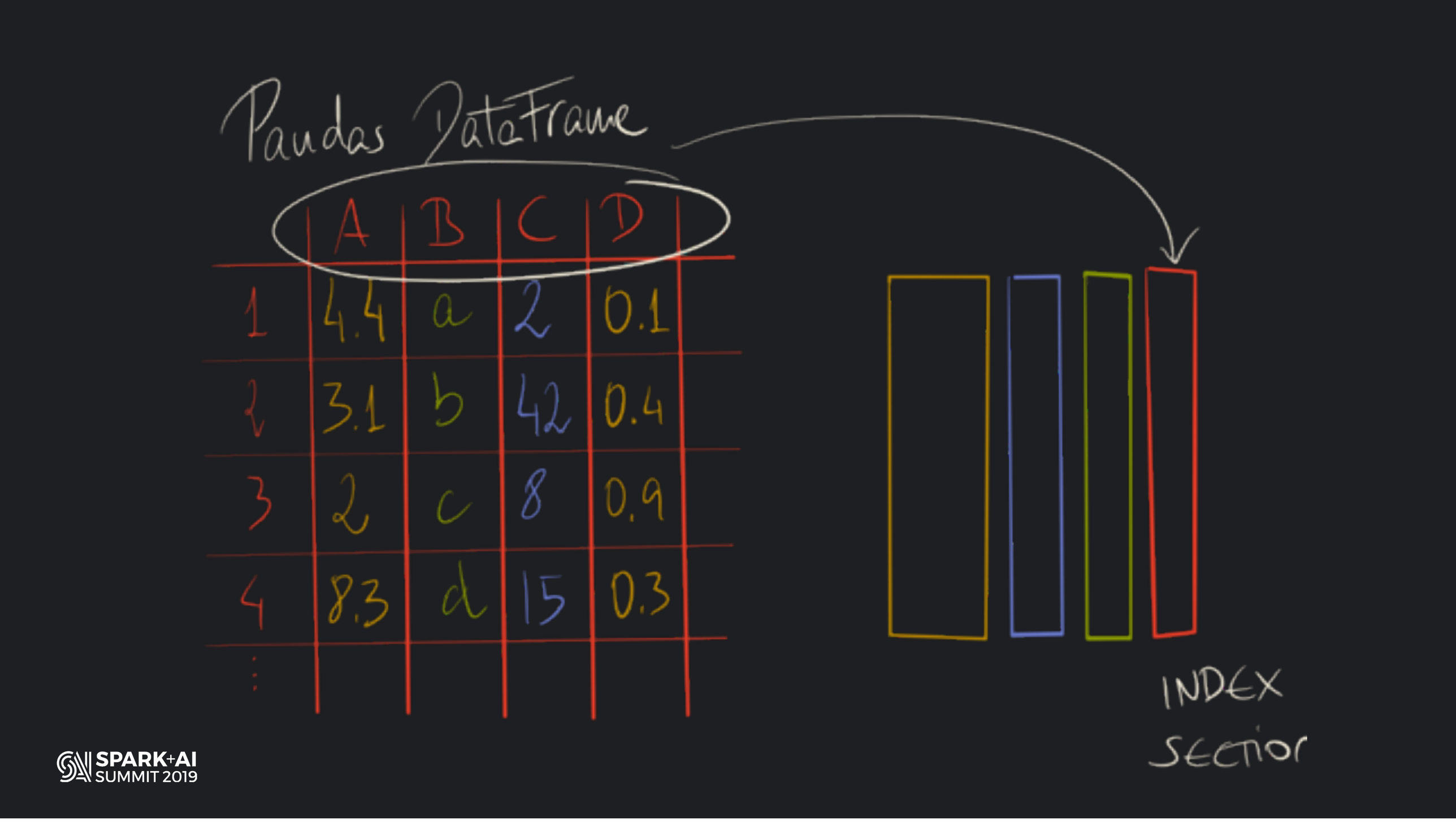
Pandas DataFrame

	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	
⋮					

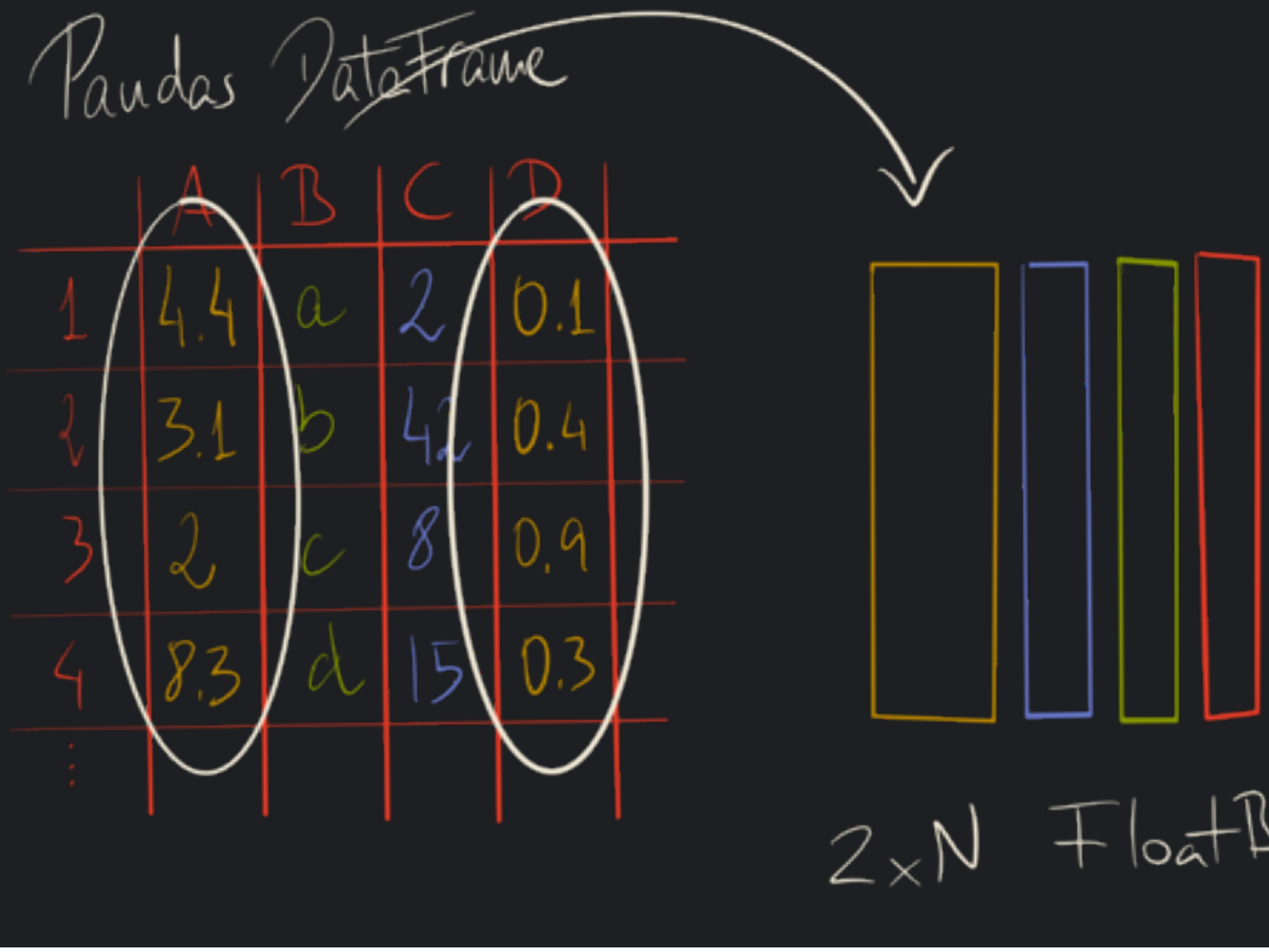
Imagine we have this table, with some columns of different types



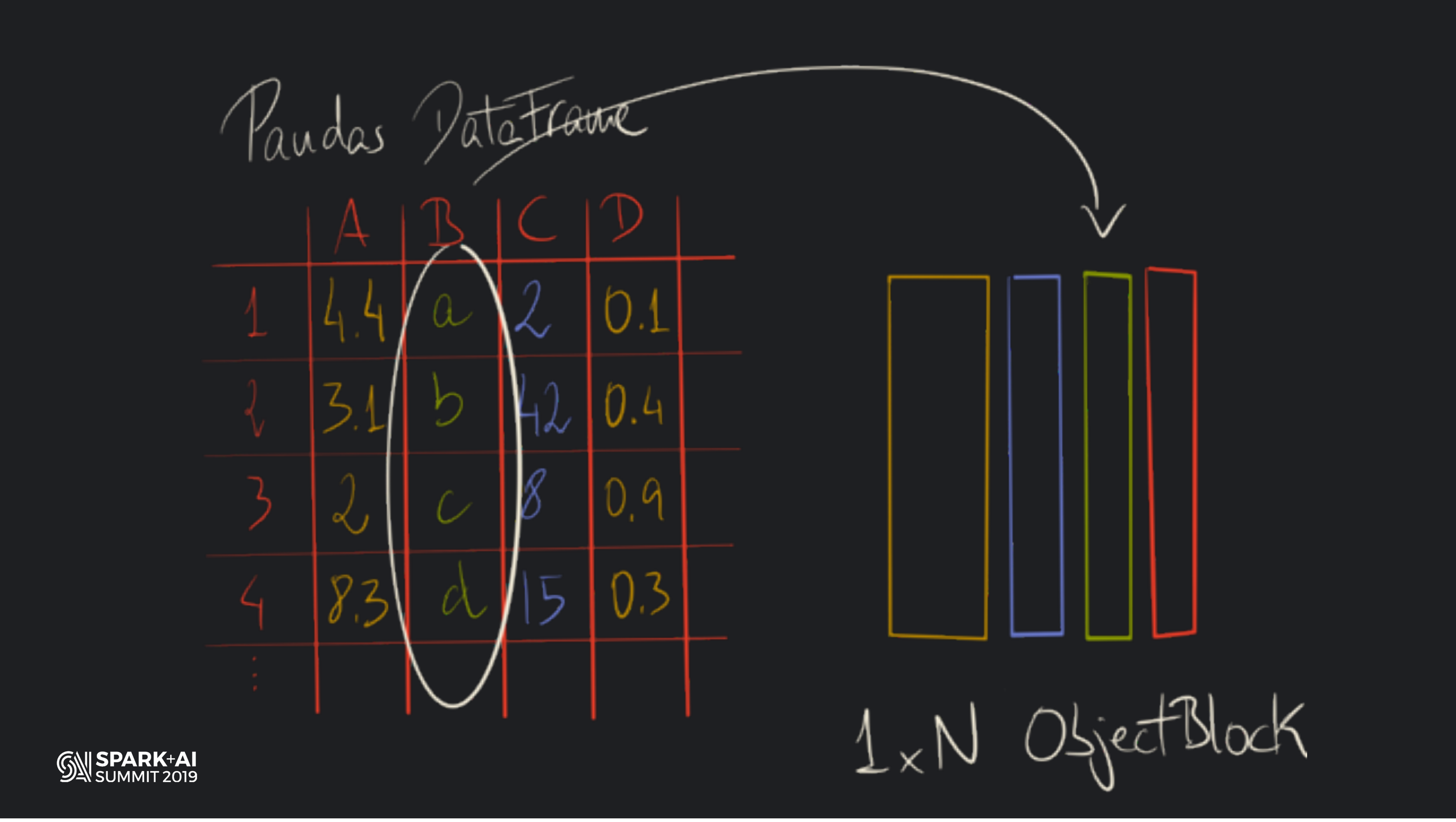
Internally, Pandas handles columns *together by type*



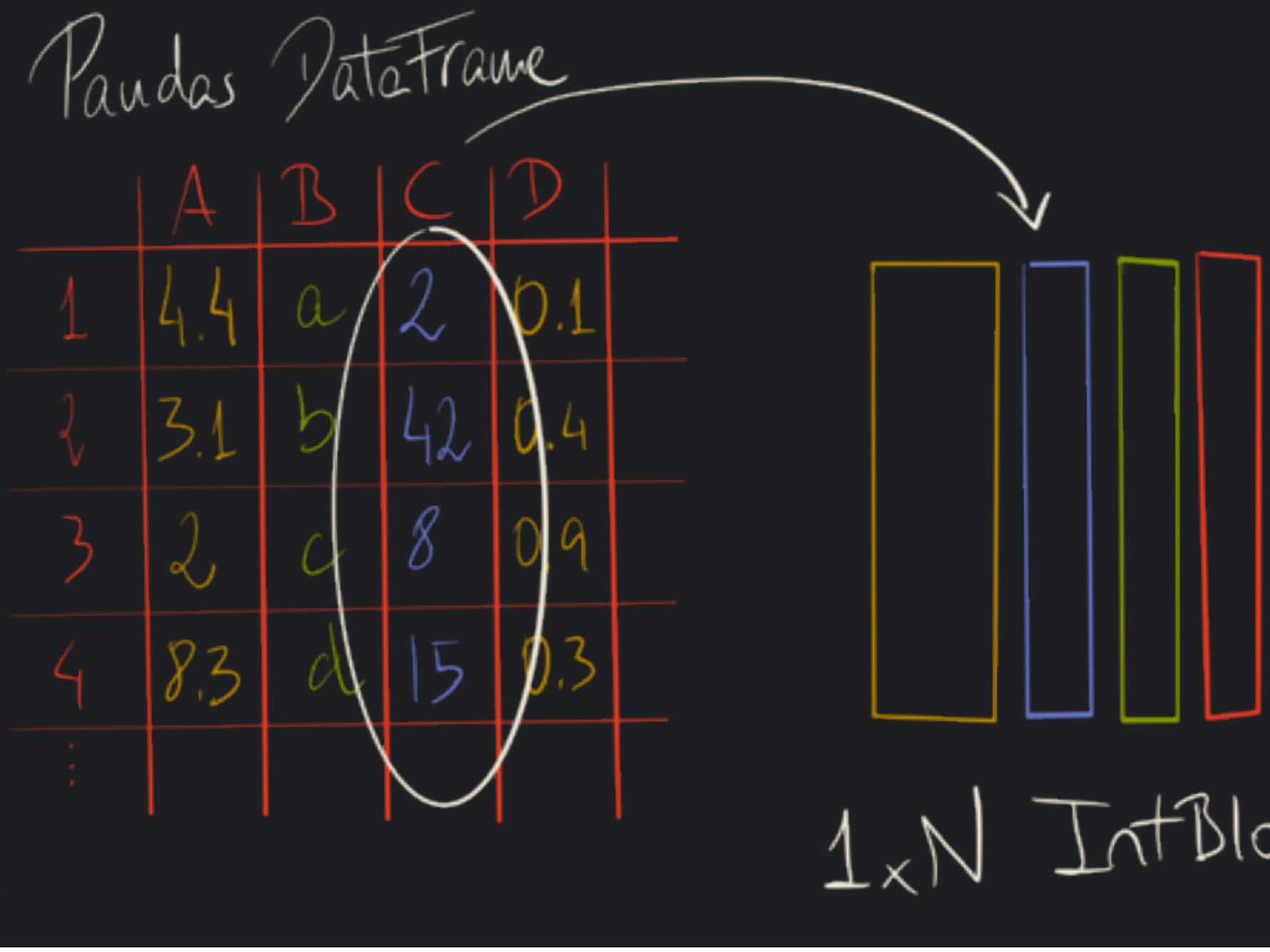
So, we keep the index separate



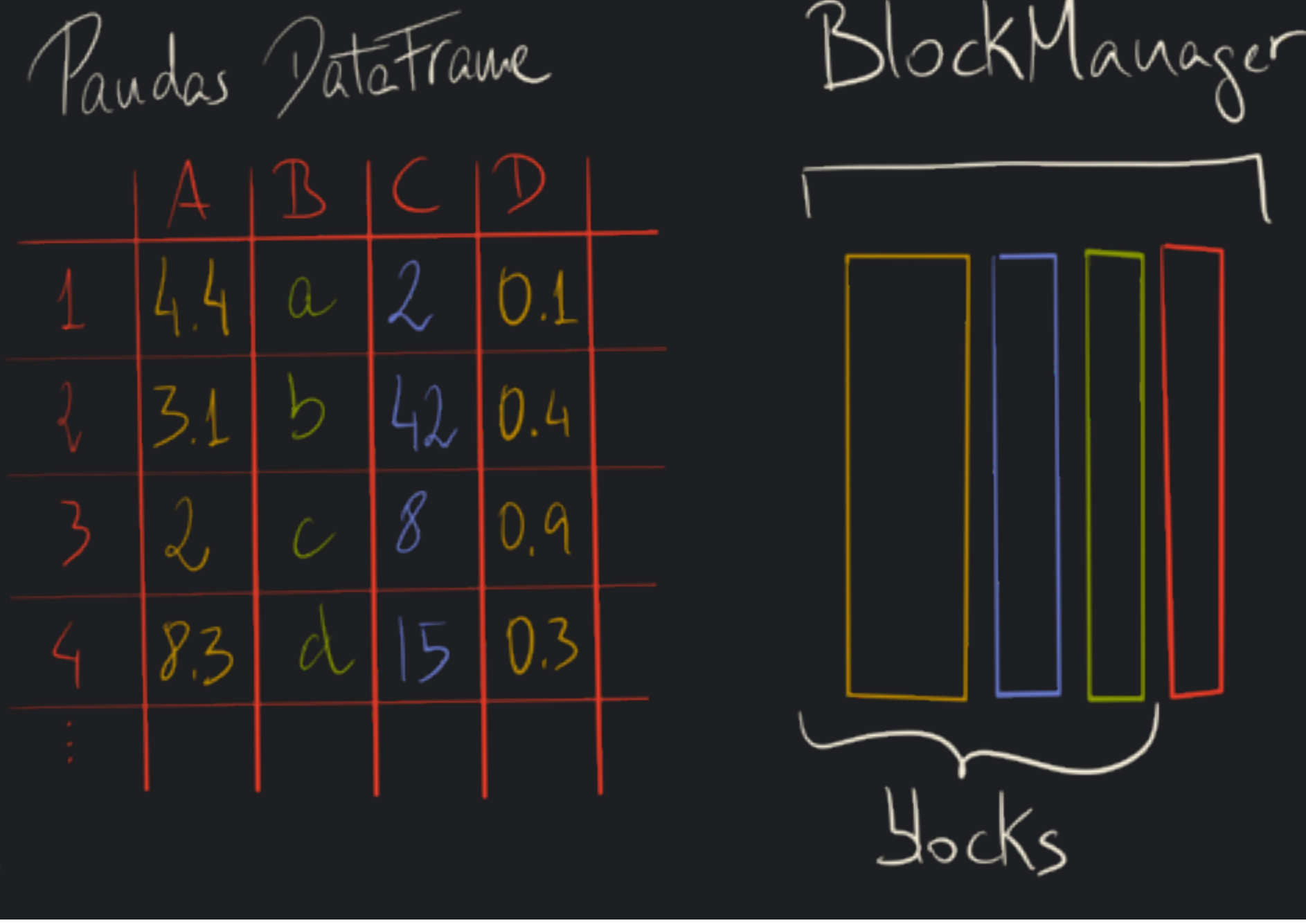
All the floats "together"



Strings and binary stuff in the same place



And integers collected



A `BlockManager` structure handles these `Blocks`

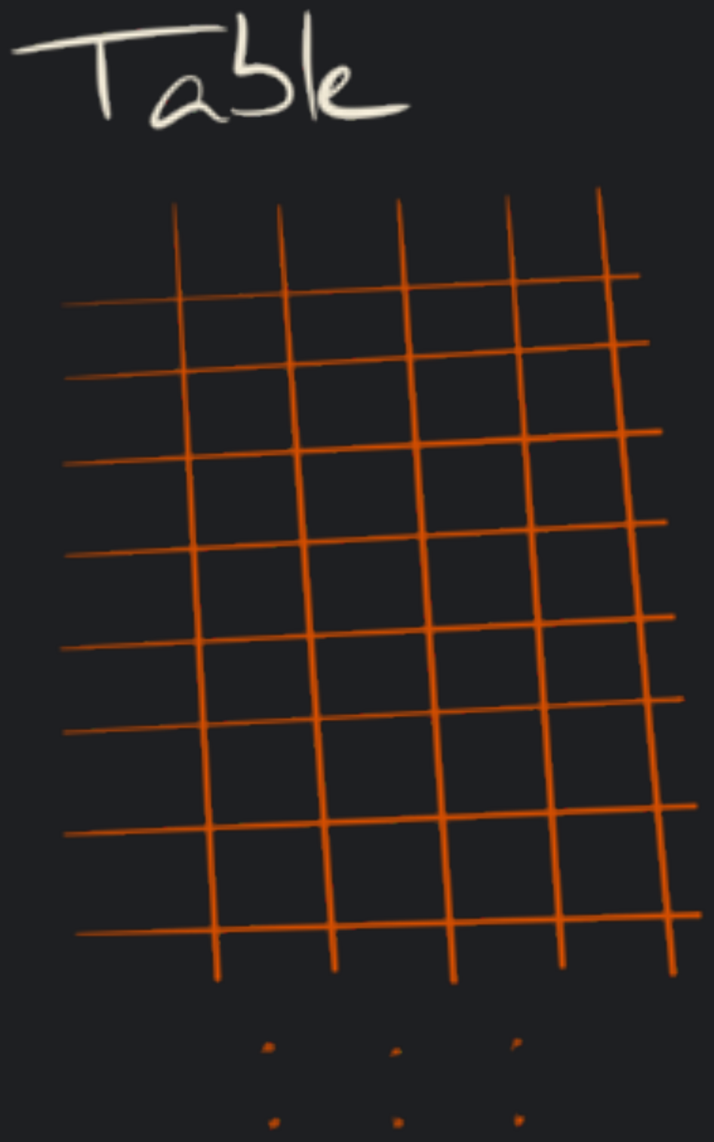
WHAT IS ARROW?

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

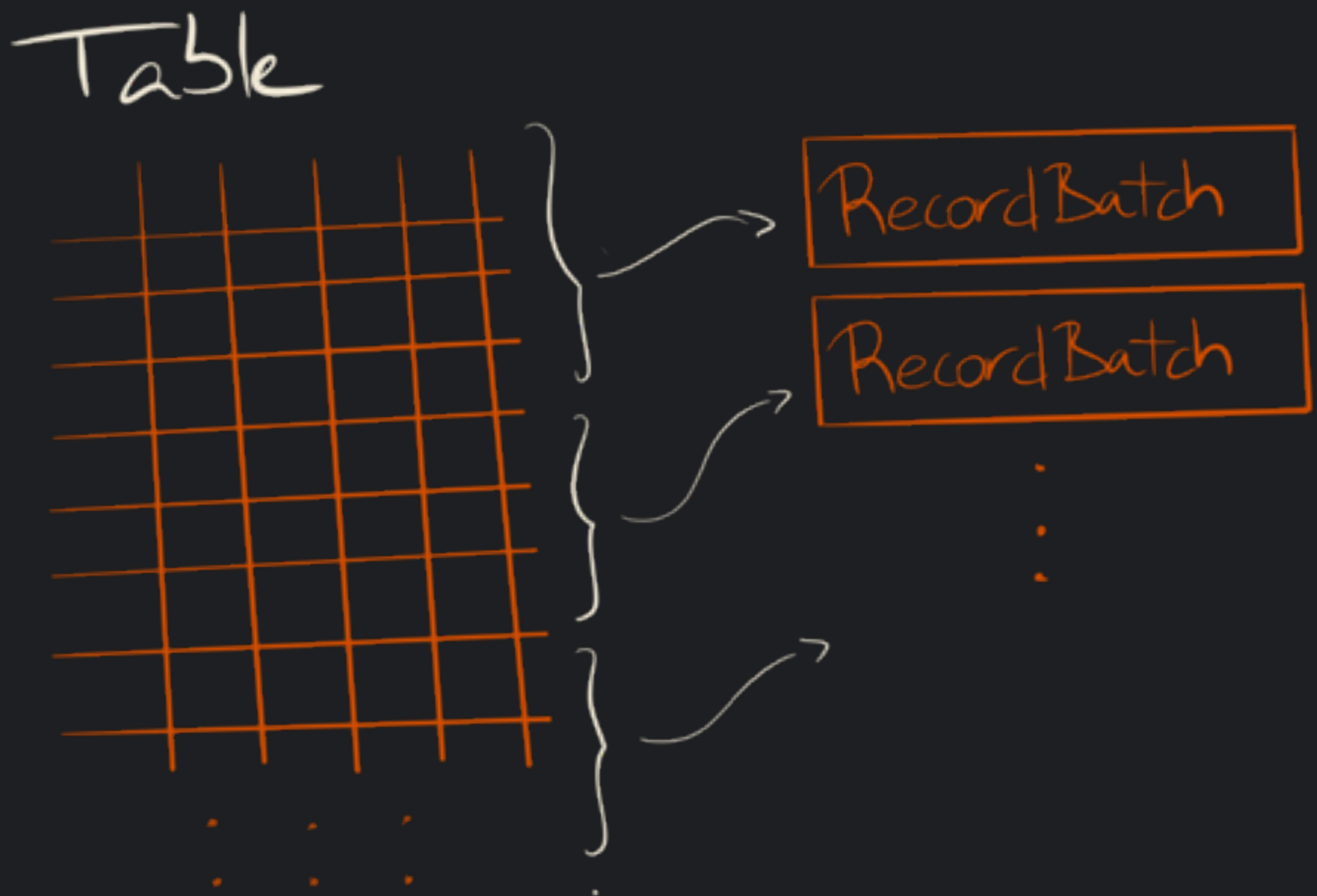
Some projects using Arrow internally are
Pandas,
Spark,
Parquet, Dask,
Ray, a POC of data processing in Rust with Arrow, Ballista, PyJava a Java/Scala connector for Python using Arrow to interchange data

HOW DOES ARROW MANAGE COLUMNAR DATA?

I will show a simplification here

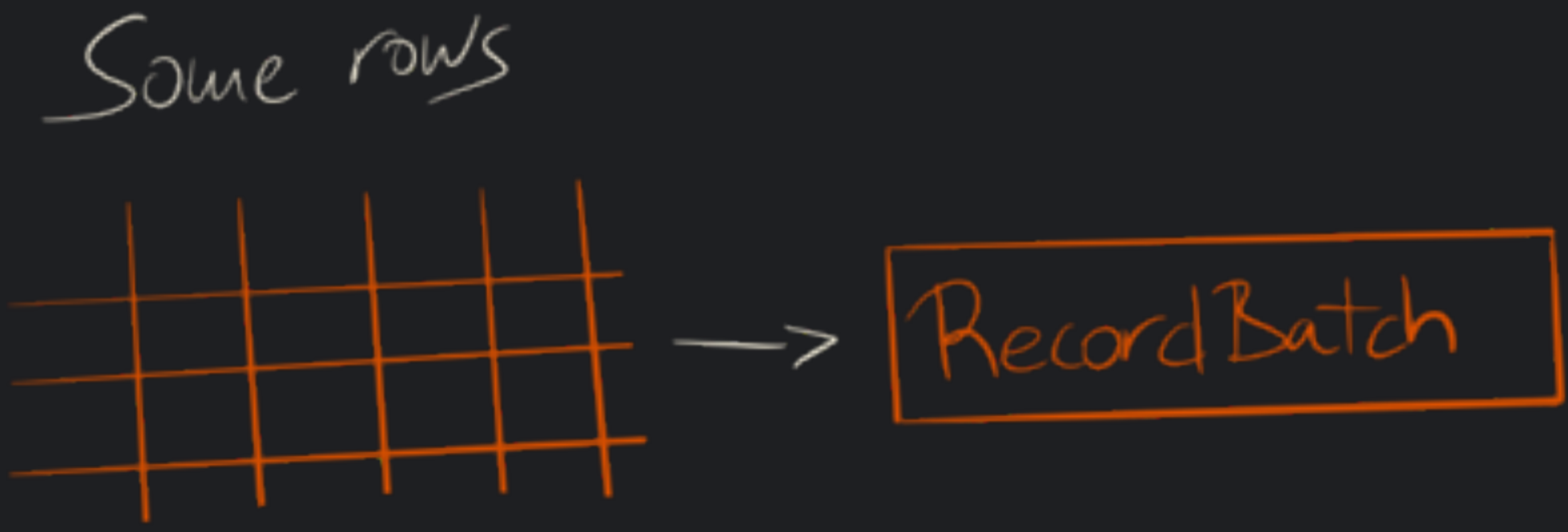


Everything starts with a `Table` structure

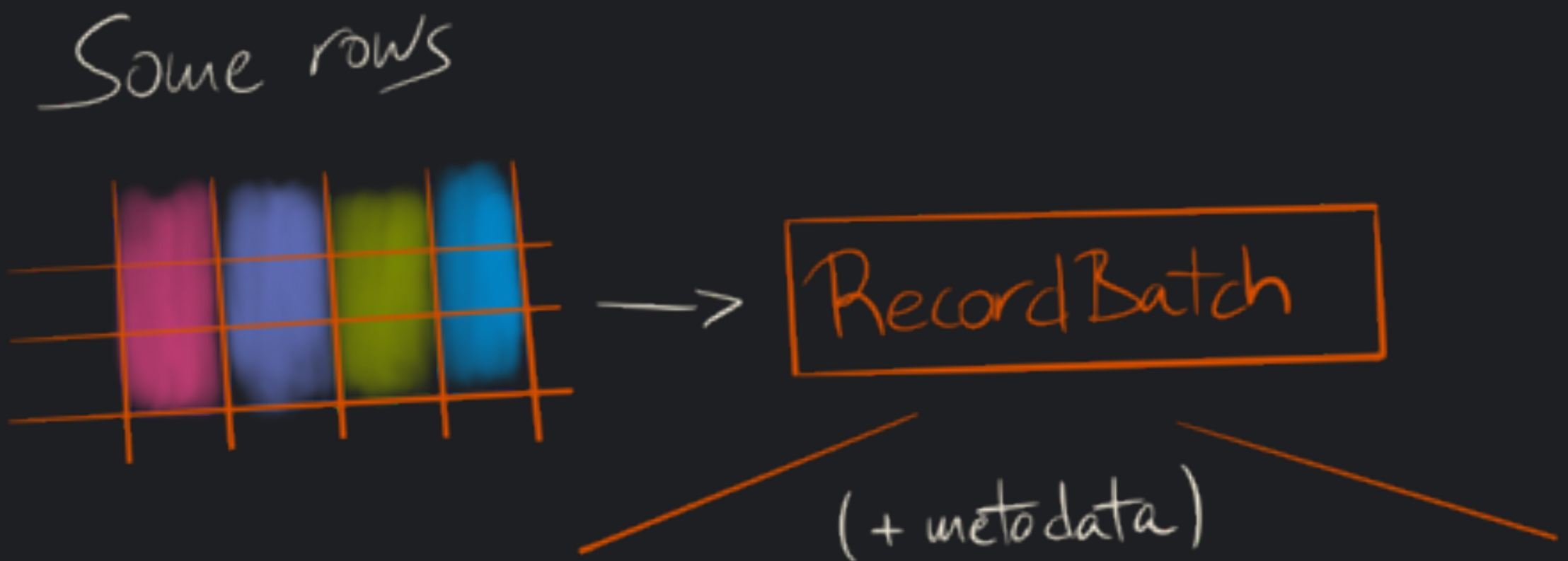


It is formed of `RecordBatches`, which contain a certain amount of rows. This makes it a streamable format

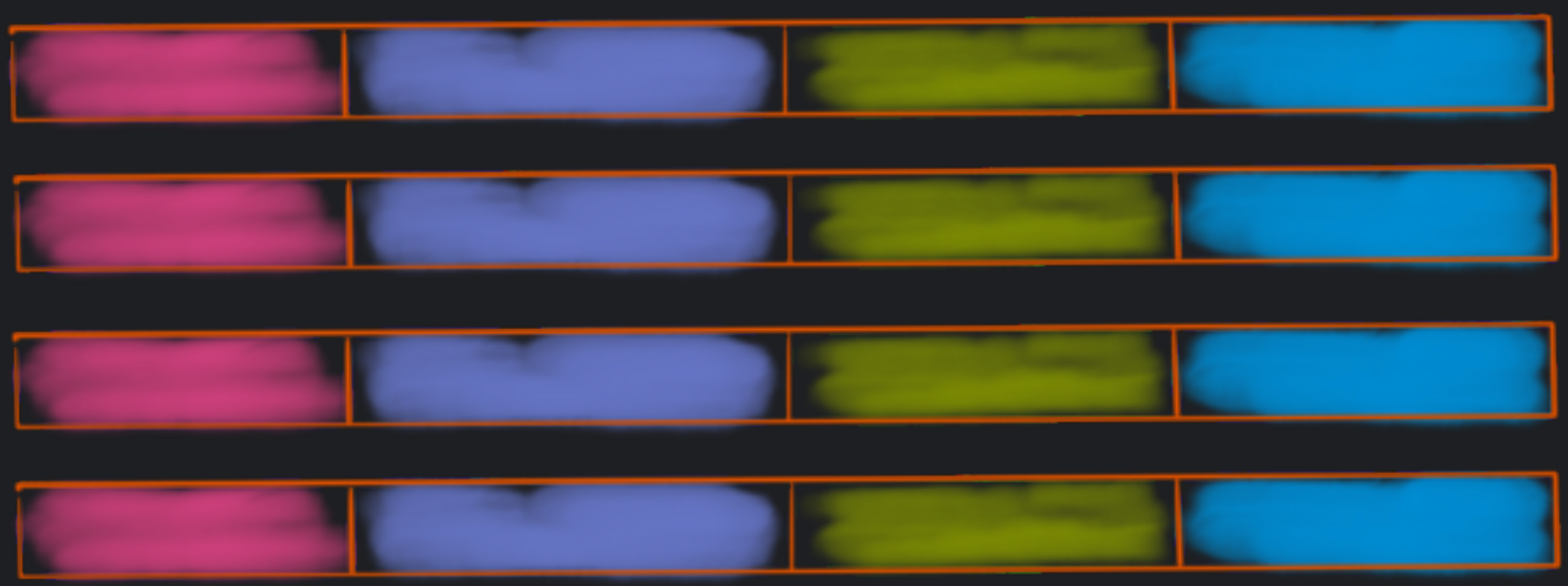
So, we have rows



And together form a `RecordBatch`



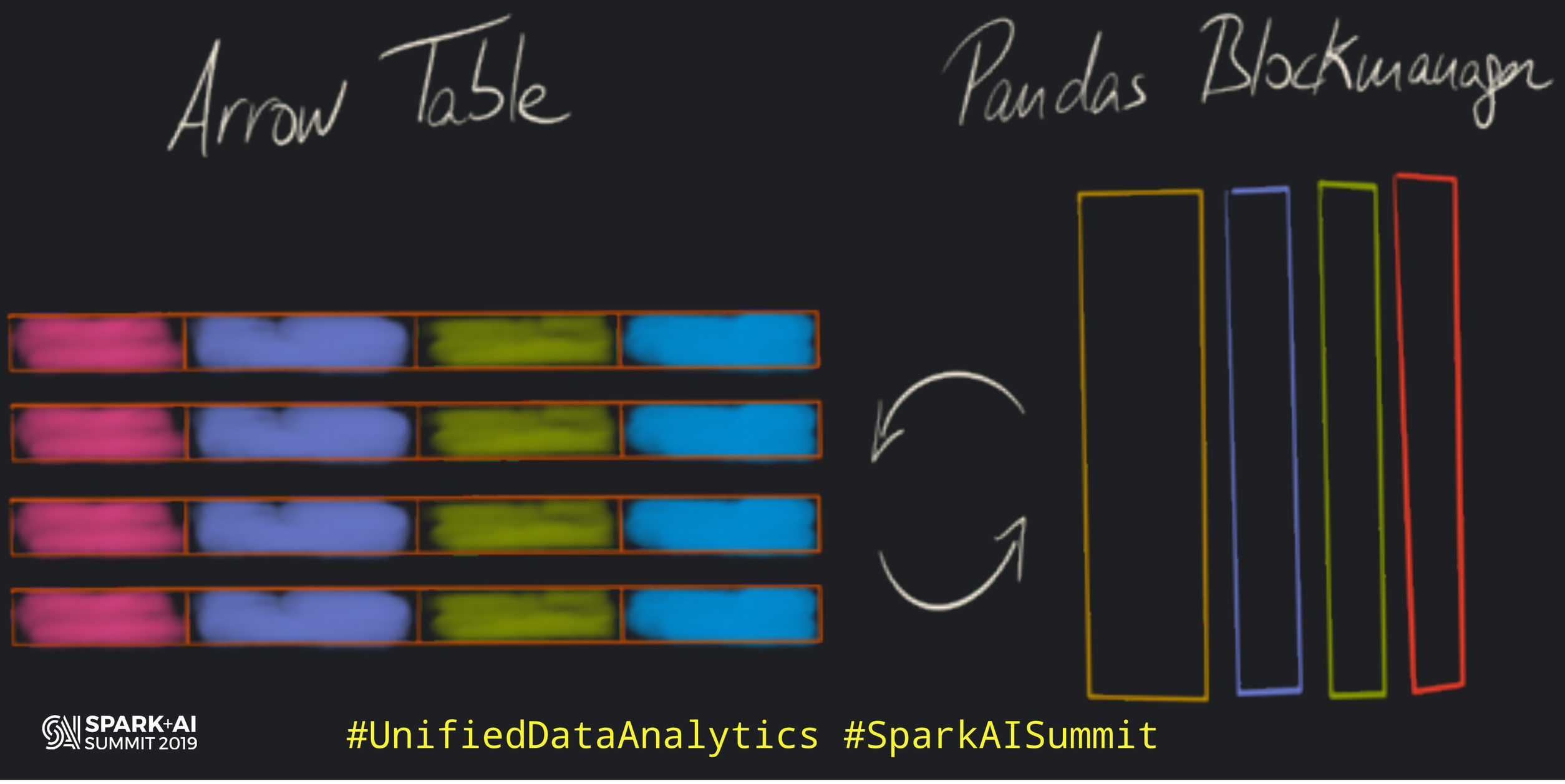
Internally, `RecordBatches` have a columnar layout. Each `RecordBatch` contains some additional metadata.



In the end, you can think of an Arrow `Table` is formed of a set of
`RecordBatches` for this presentation. It's actually a collection of *chunked*
arrays, where each array is formed of different chunked pieces column-wise.
RecordBatches have the same length (so, when you create a table from a set of
`RecordBatches` all the chunks are "the same"). Thanks Uwe L. Korn
(xhochy) for his pointer to this.

- 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`.



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?

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

HOW DOES SPARK WORK?

SPARK USUALLY RUNS ON TOP OF A CLUSTER MANAGER

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

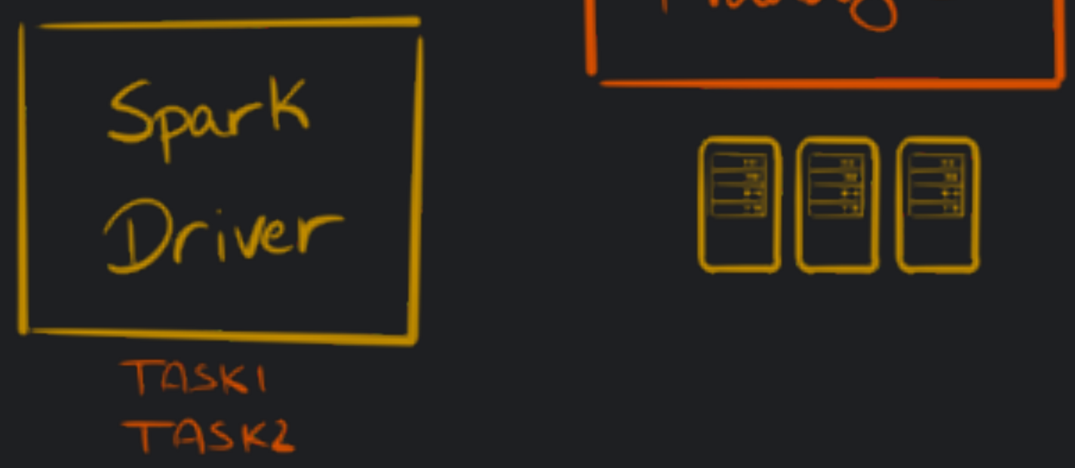
AND A DISTRIBUTED STORAGE

A SPARK PROGRAM RUNS IN THE DRIVER

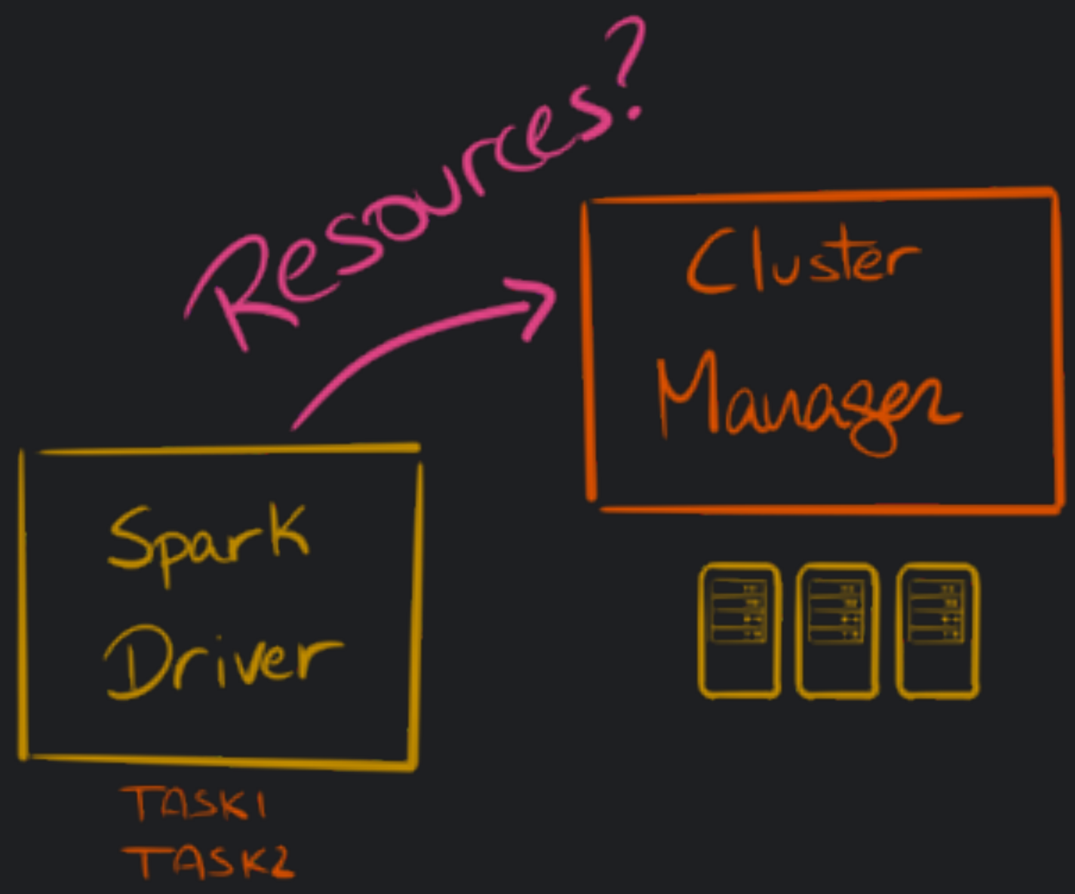
THE DRIVER REQUESTS RESOURCES FROM THE CLUSTER MANAGER TO RUN TASKS

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

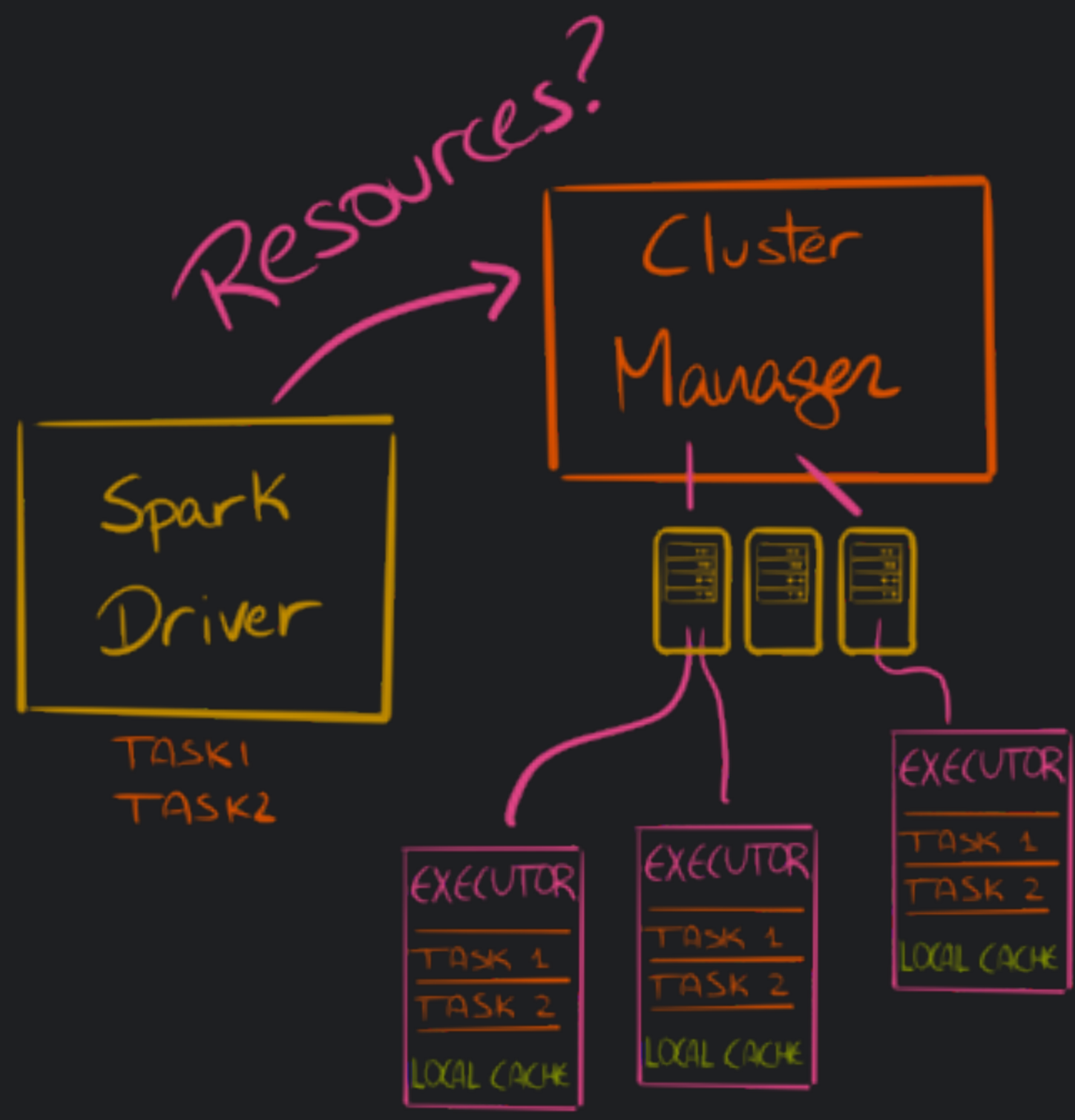
THE DRIVER REQUESTS RESOURCES FROM THE CLUSTER MANAGER TO RUN TASKS



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

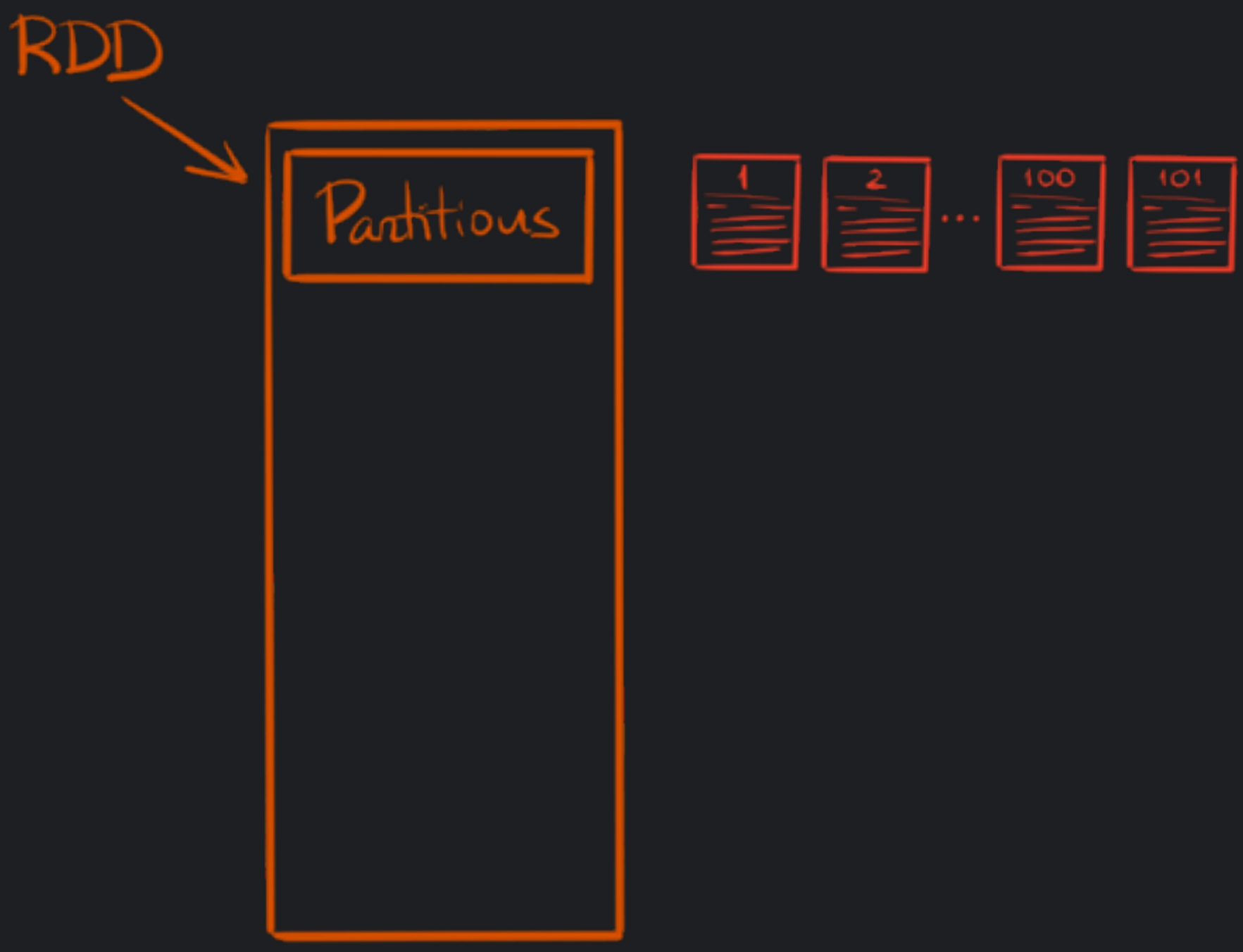
RDDs are defined in the Scala core. In Python and R we end up interacting with the underlying JVM objects

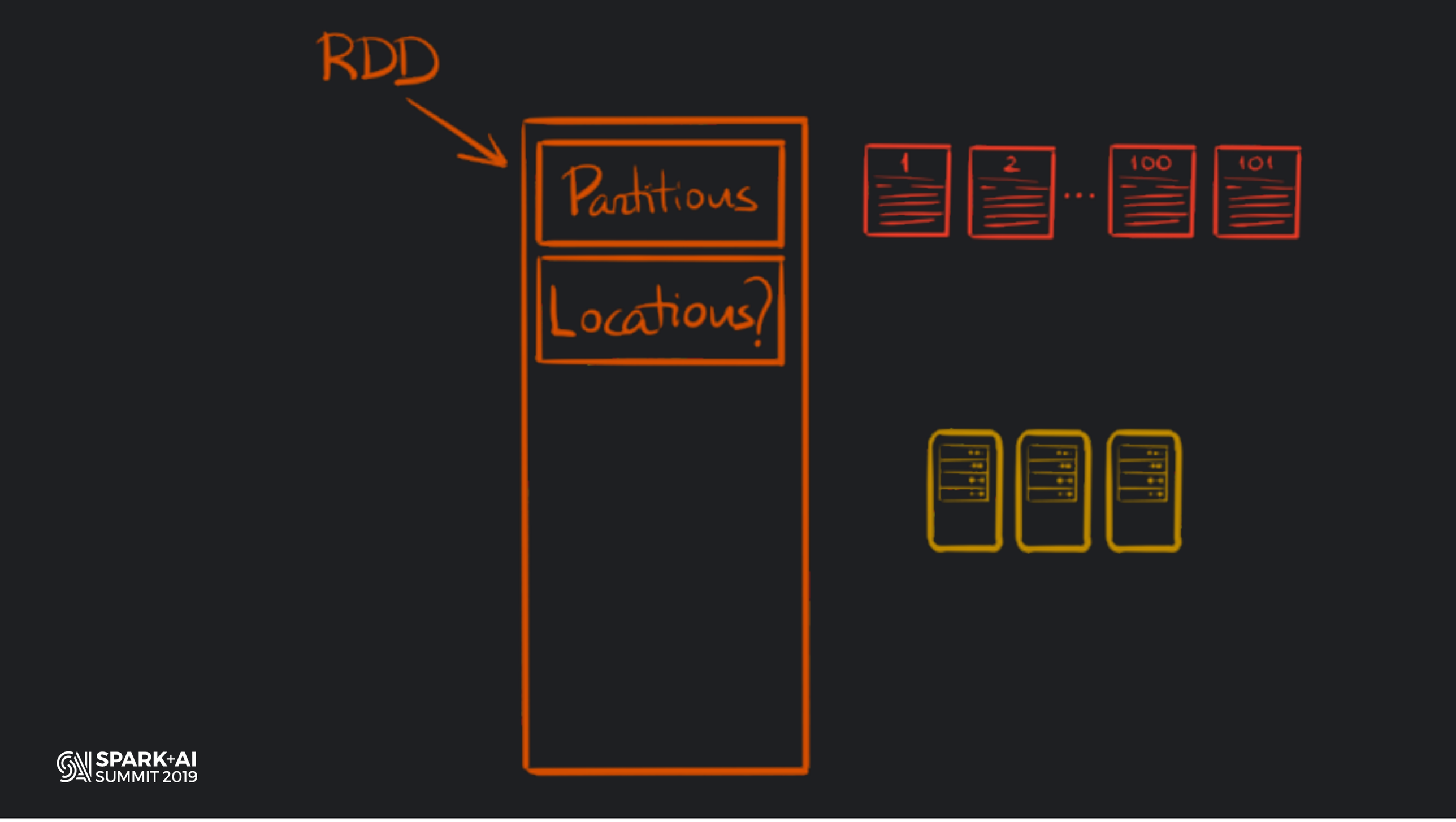
Happy RDD

SPARK+AI SUMMIT 2019

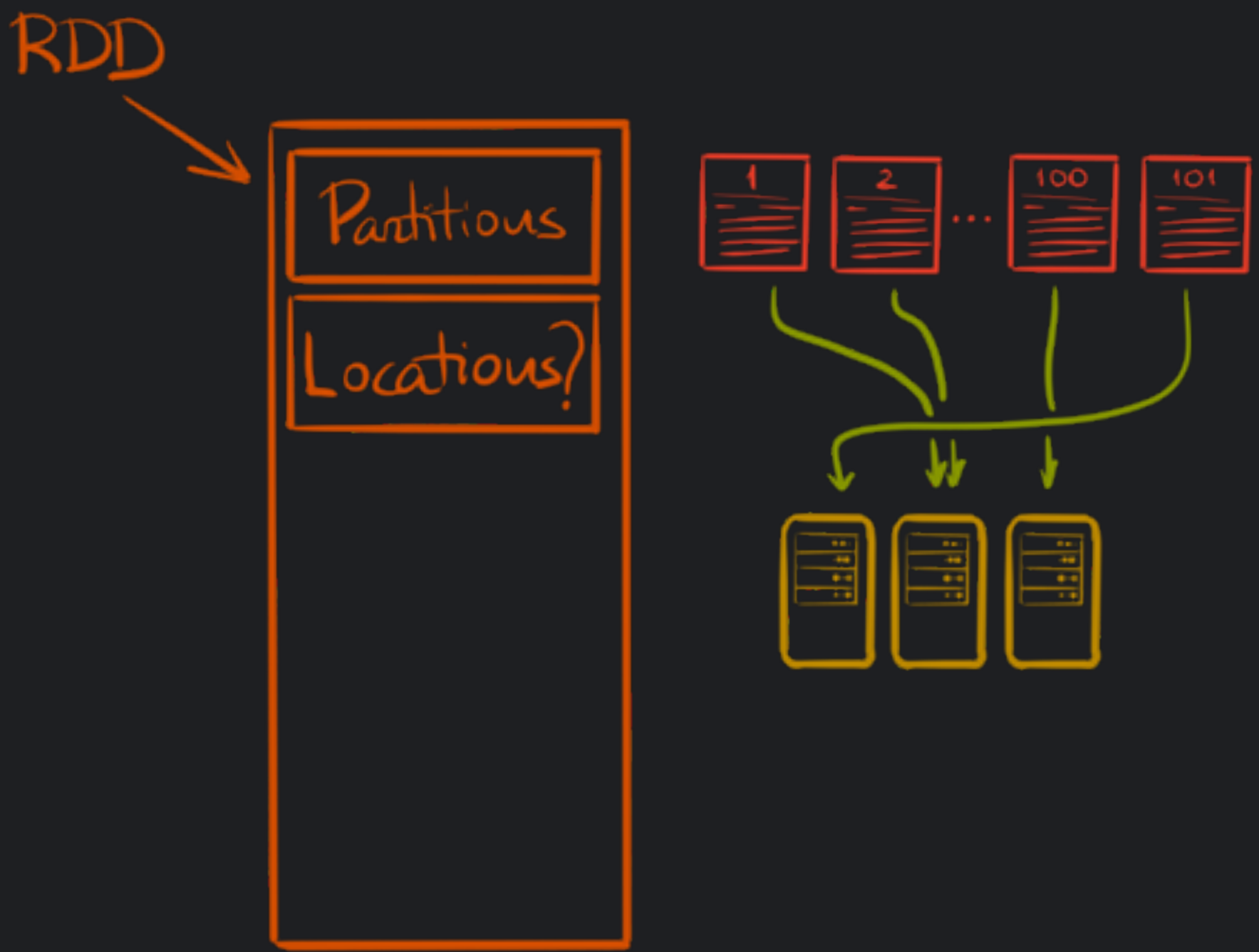
Happy RDD. A RDD needs 5 items to be considered complete (2 of them optional)

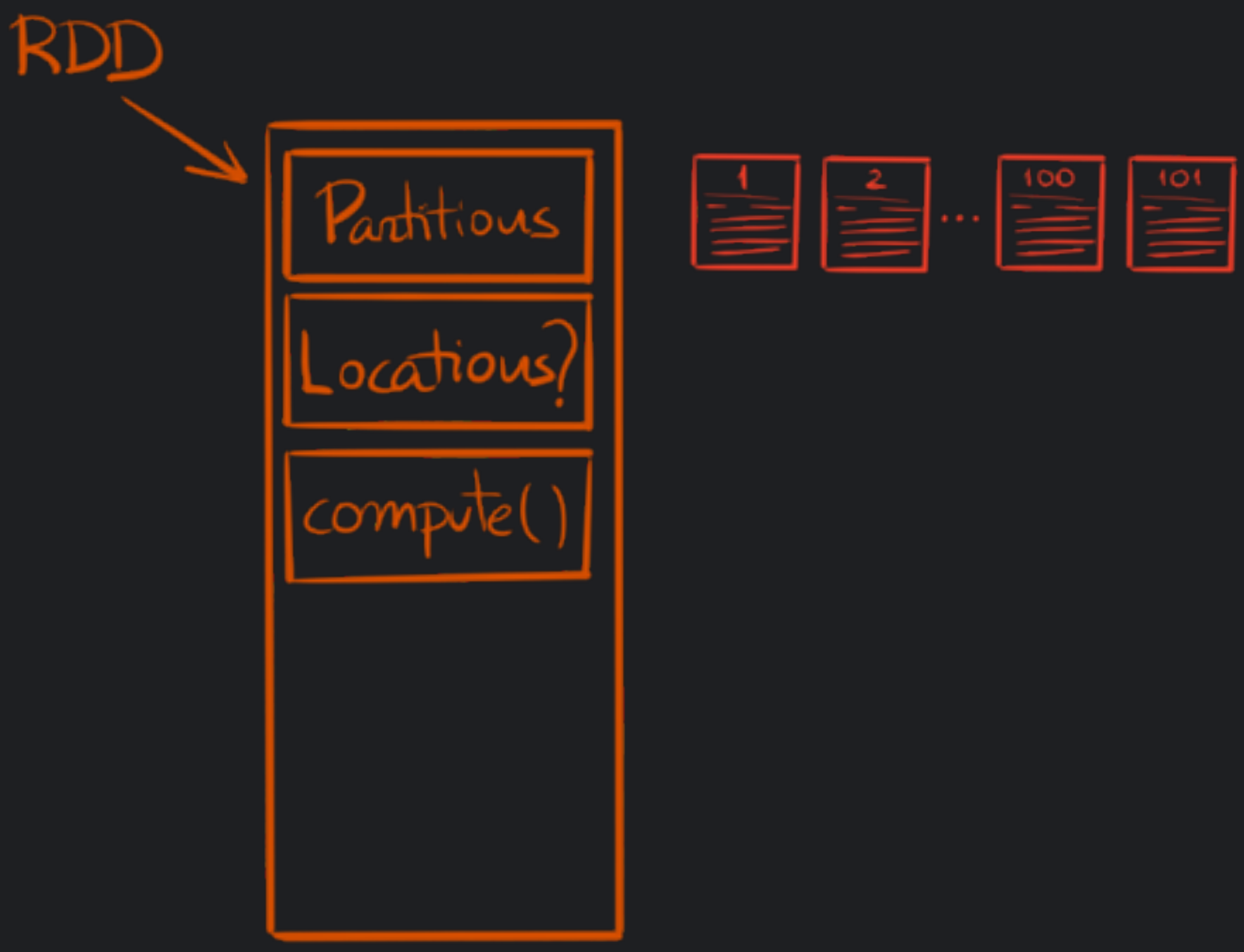
Partitions define how the data is *partitioned* across machines


RDD
Partitions
1
2
100
101

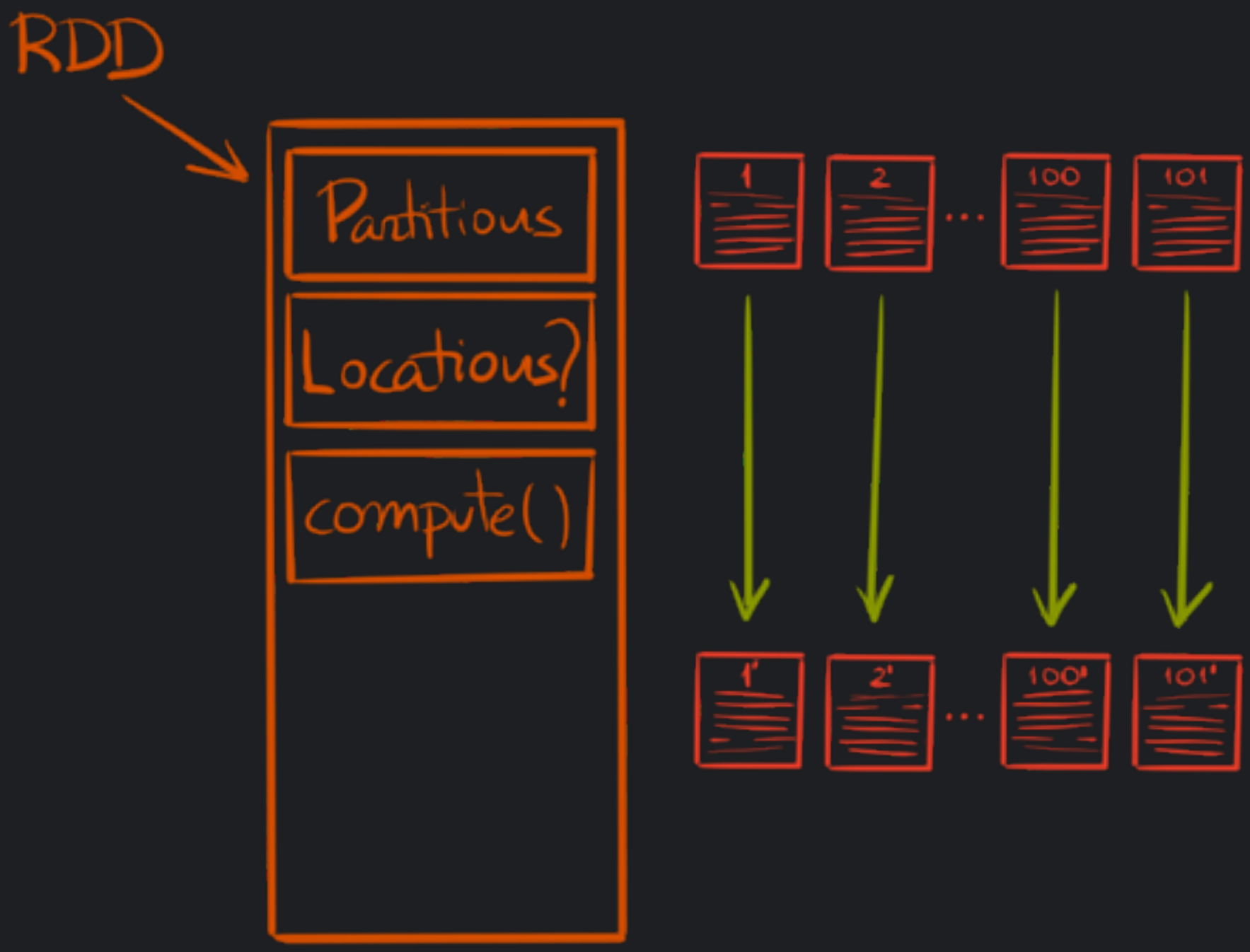


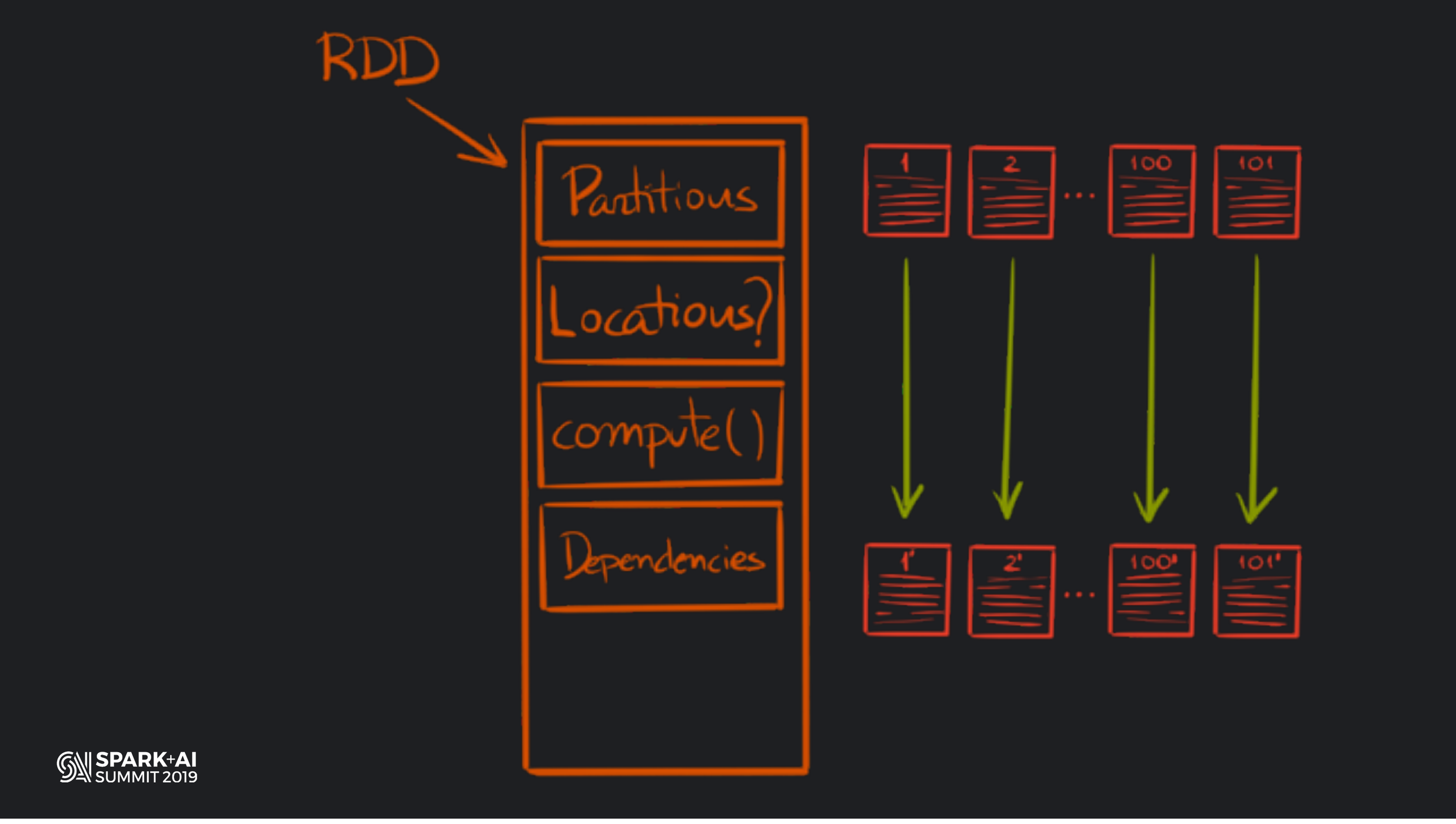
Locations (`preferredLocations`) are optional, and allow for fine grained execution to avoid shuffling data across the cluster


RDD
Partitions
Locations?
1
2
100
101

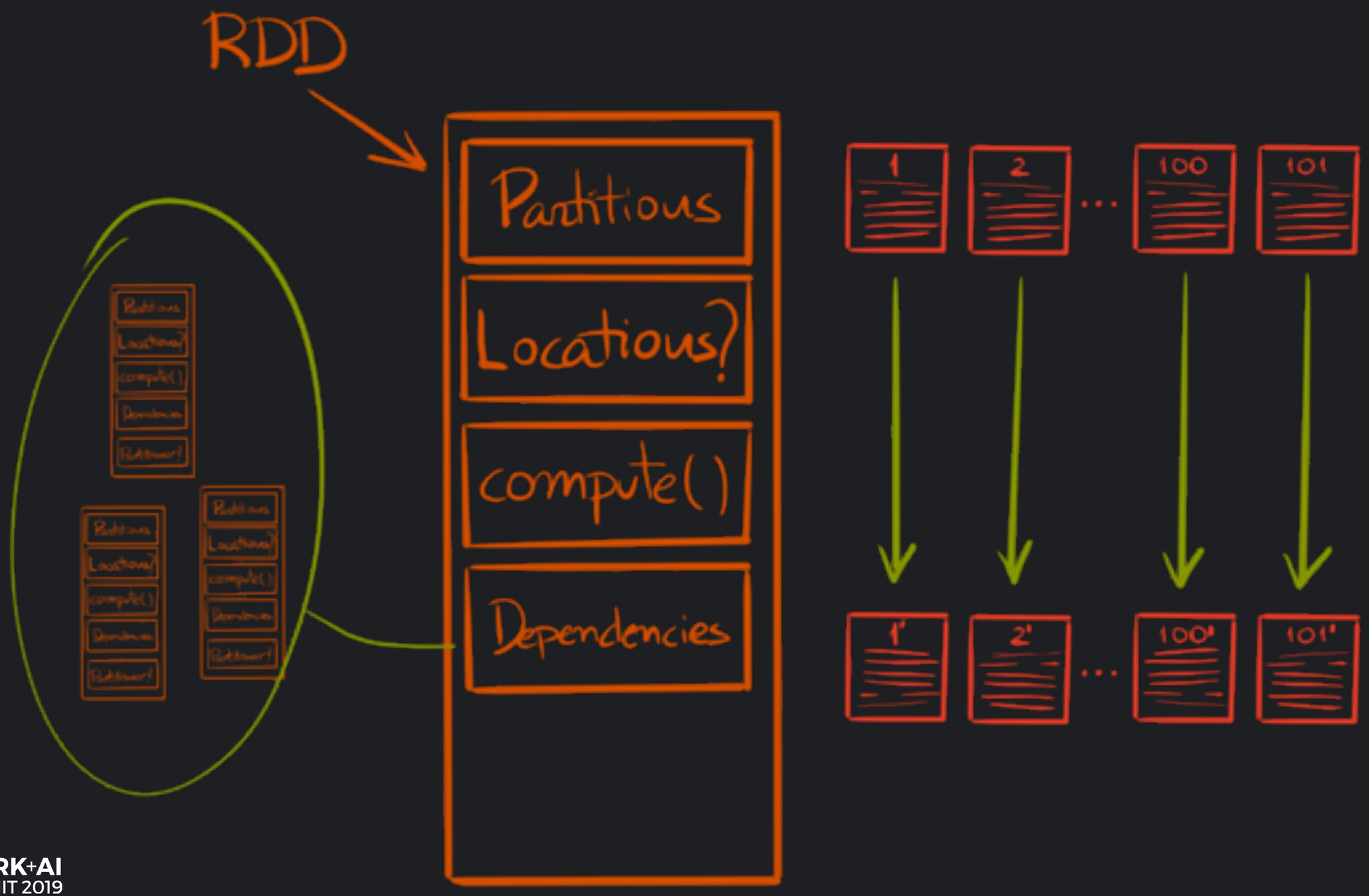


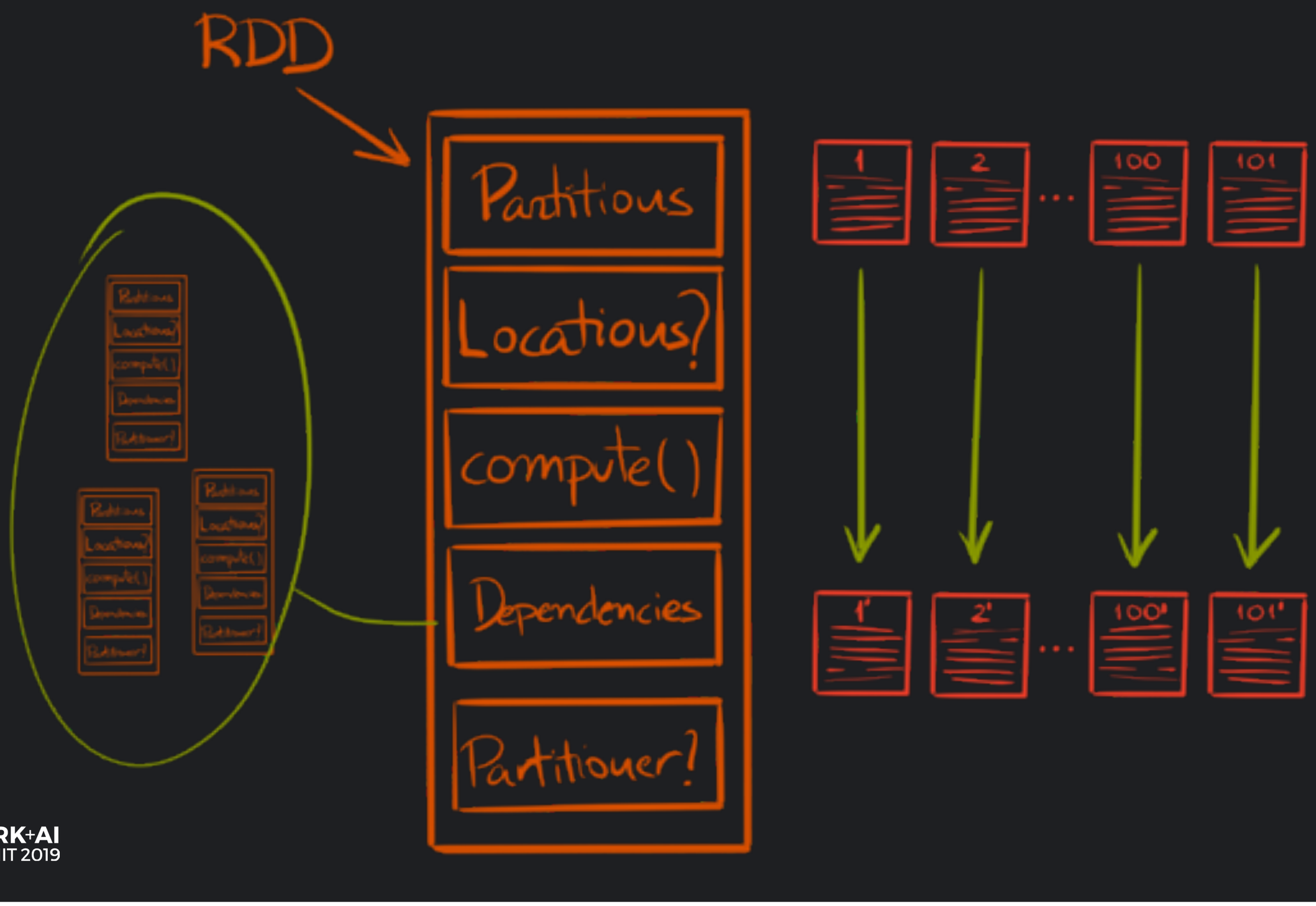
Compute is evaluated in the executors, each executor has access to its assigned partitions. The result of compute is a new RDD, with different data


RDD
Partitions
Locations?
compute()
1
2
100
101
1'
2'
100'
101'



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


RDD
Partitions
Locations?
compute()
Dependencies
1
2
100
101
1'
2'
100'
101'



The partitioner needs to be a 1-1 function from keys/whatever to the data, to allow recomputing. It is as well optional (Spark will default to something sensible then)

PYSPARK

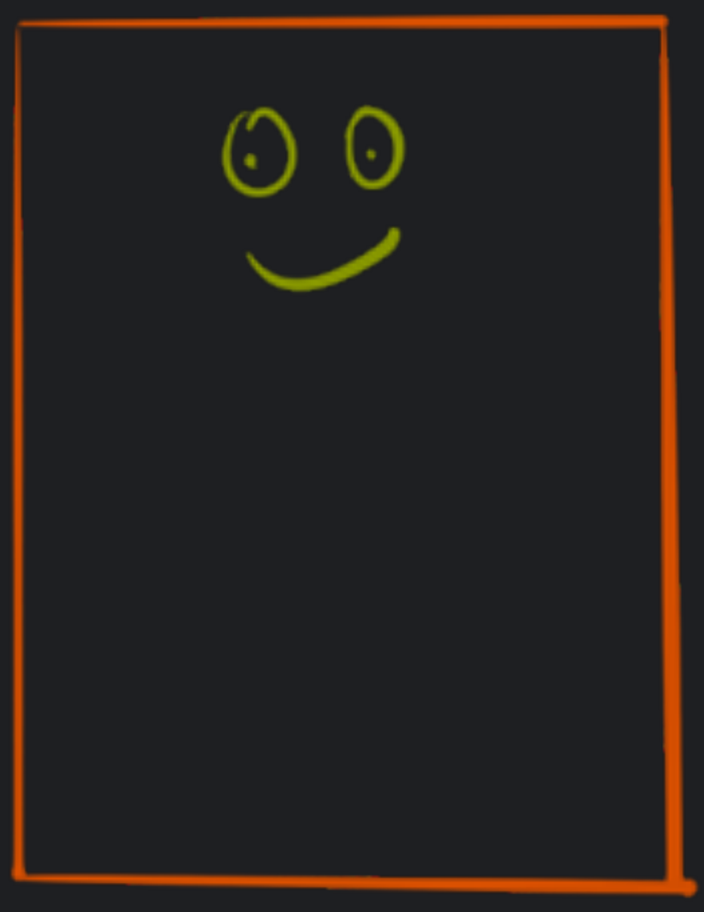
I will use PySpark to refer to the Python API on top of Spark, and say Scala Spark or just Spark for the Scala API

PYSPARK OFFERS A PYTHON API TO THE SCALA CORE OF SPARK

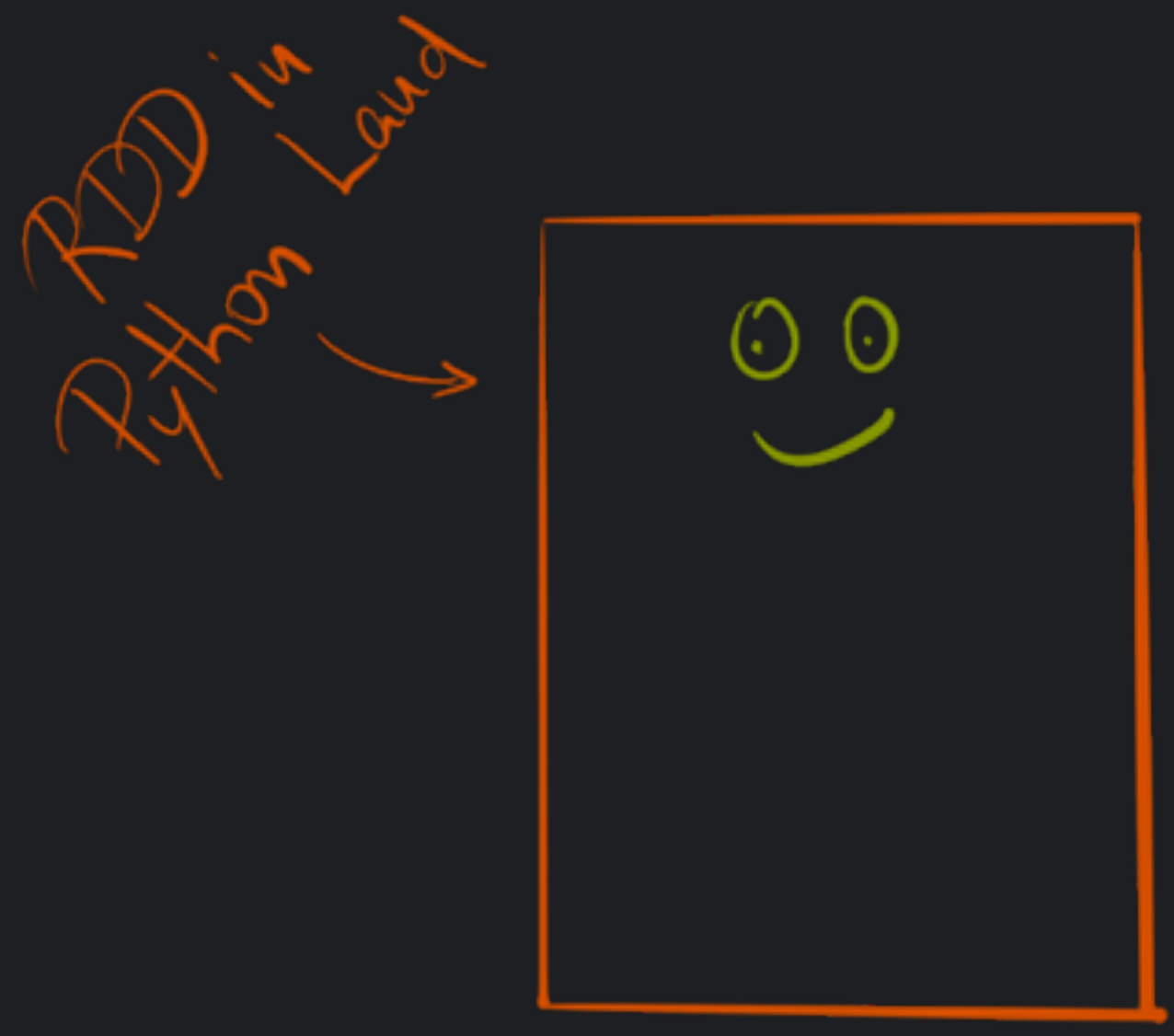
IT USES THE PY4J BRIDGE

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
```

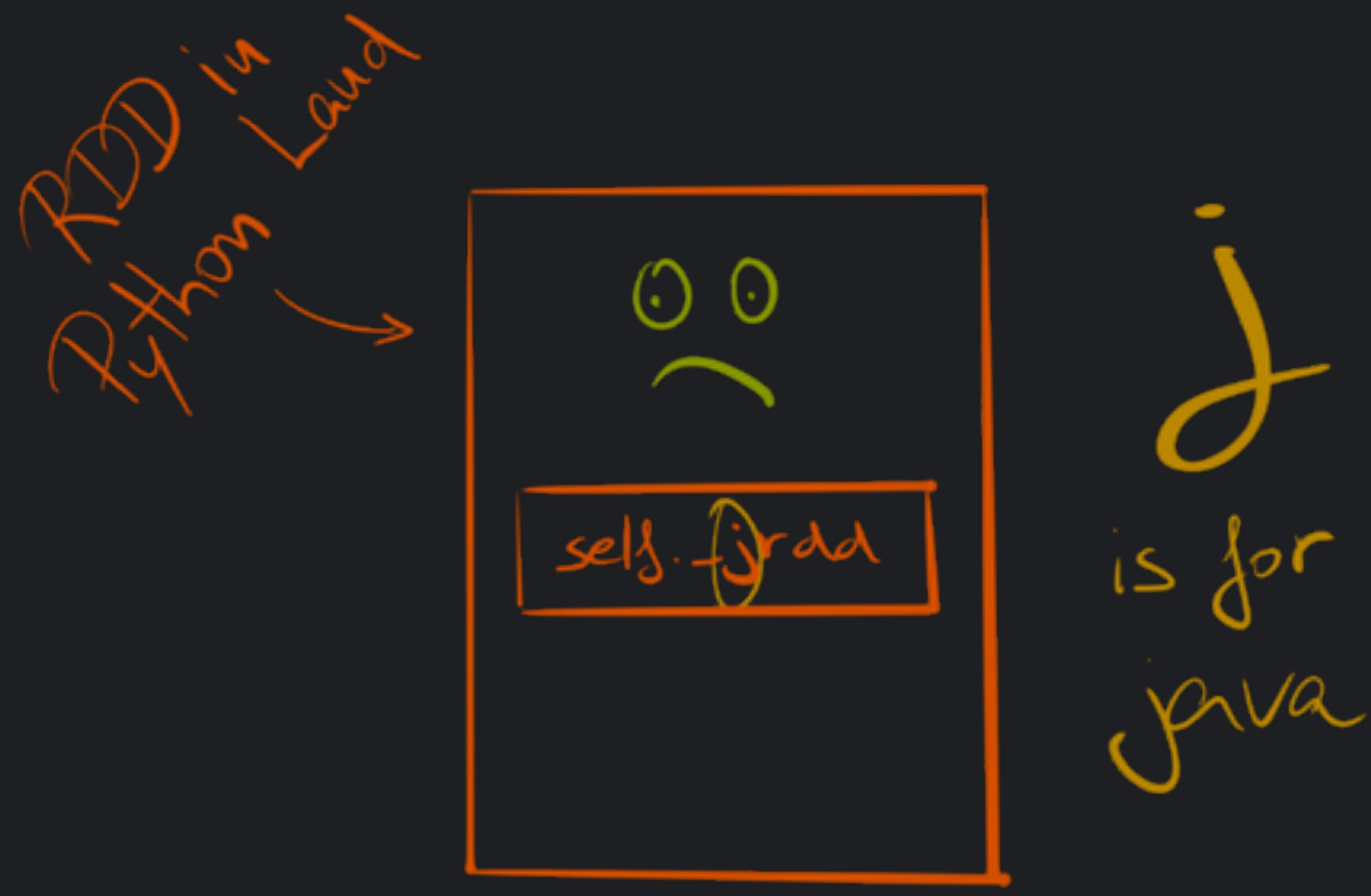


Happy RDD in Python

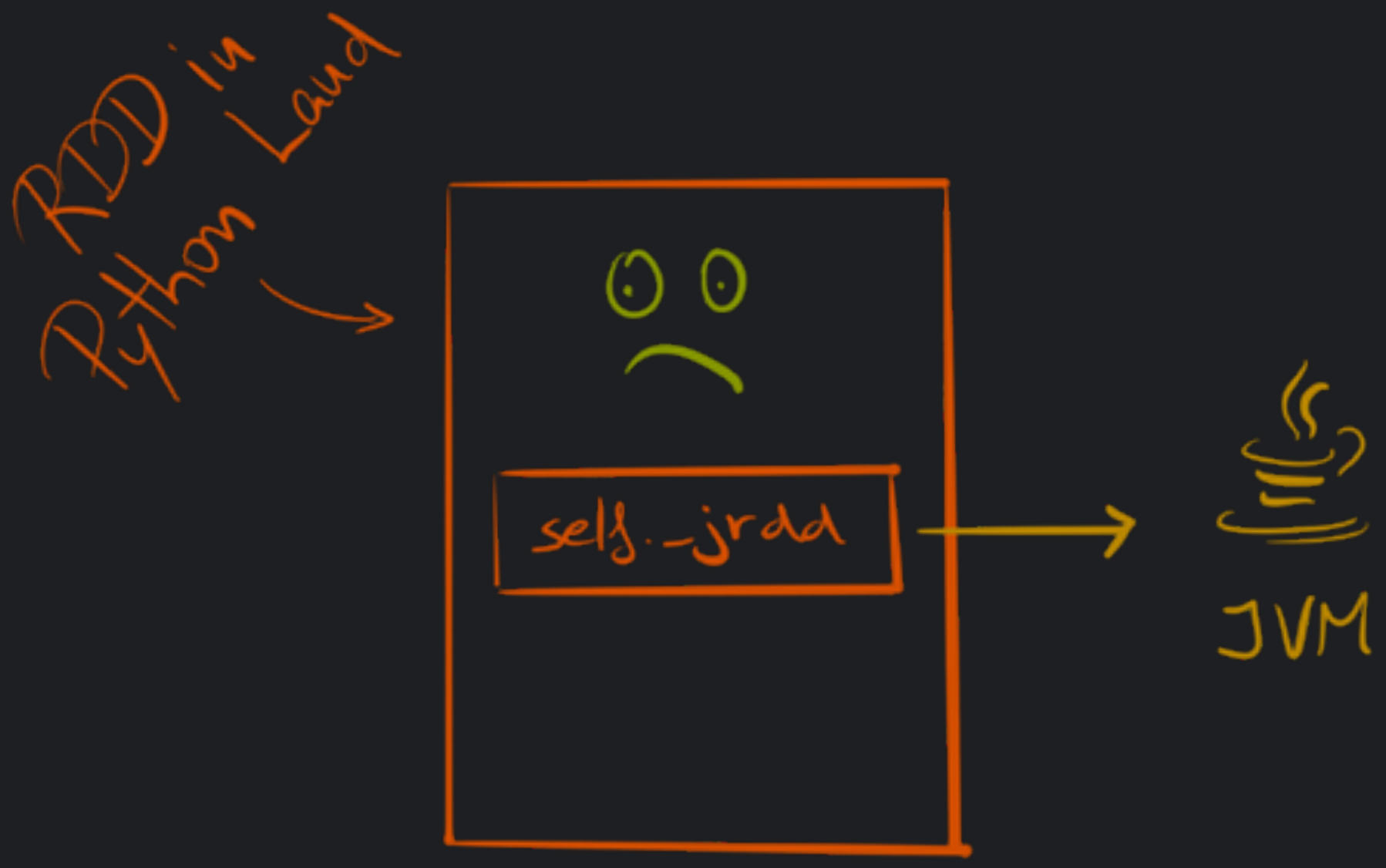

RDD in
Python Land

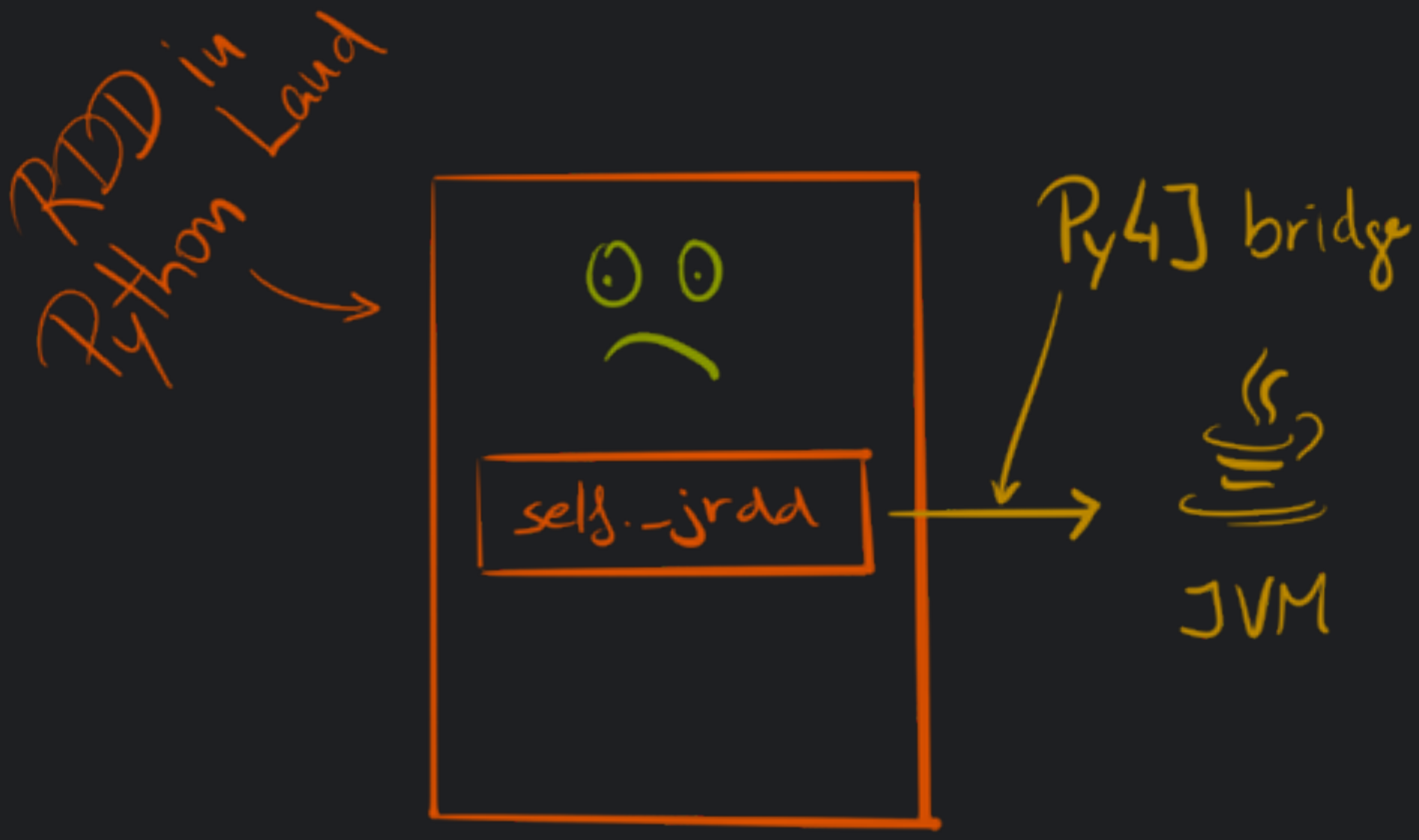
Some `_jrdd` appears. Each "Spark" related object has some internal relationship with an equivalent JVM object. An RDD, for instance, has a `_jrdd`, which refers to how the RDD was created in the JVM. By extension, if in Python you create an RDD from a file, for instance, this will call the JVM method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

Just like M is for Murder...



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


RDD in Python Land
self._jrdd
JVM


RDD in Python Land
self._jrdd
Py4J bridge
JVM

THE MAIN ENTRYPOINTS ARE RDD AND PipelinedRDD(RDD)

In Python land. The first exposes the API of Scala RDDs (by interacting with
the JVM connected to the underlying 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

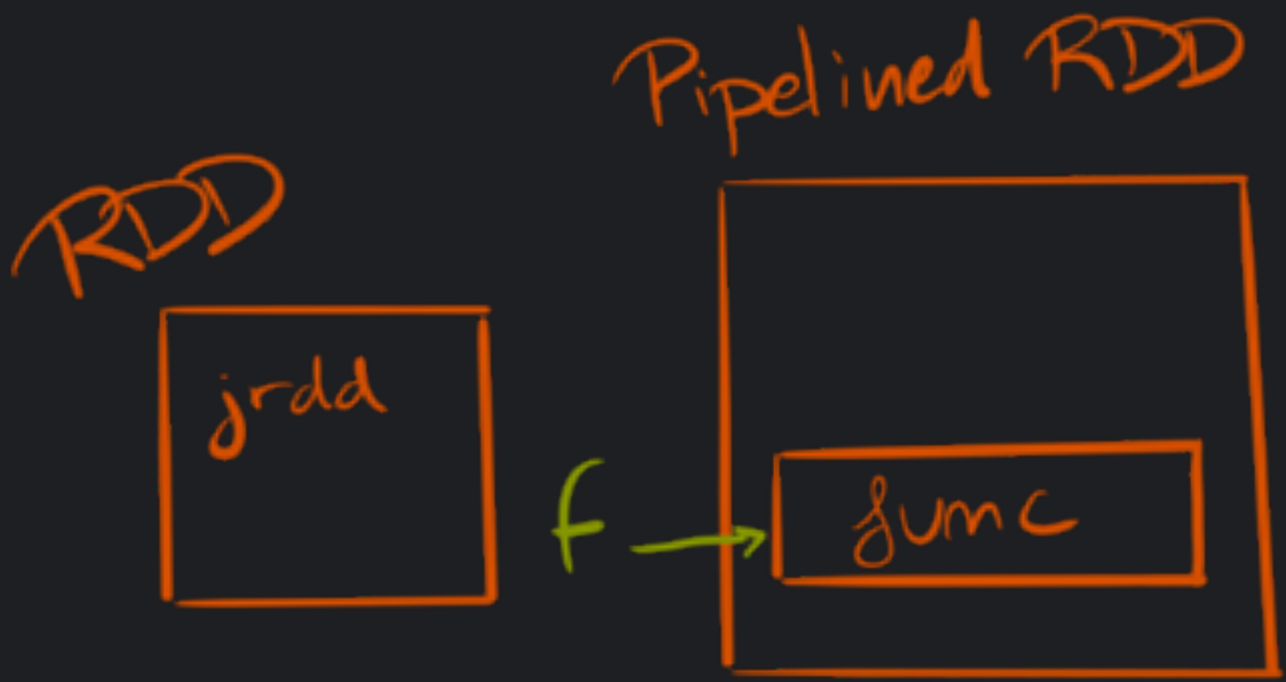
PipelinedRDD
BUILDS IN THE JVM A
PythonRDD

Angry RDD

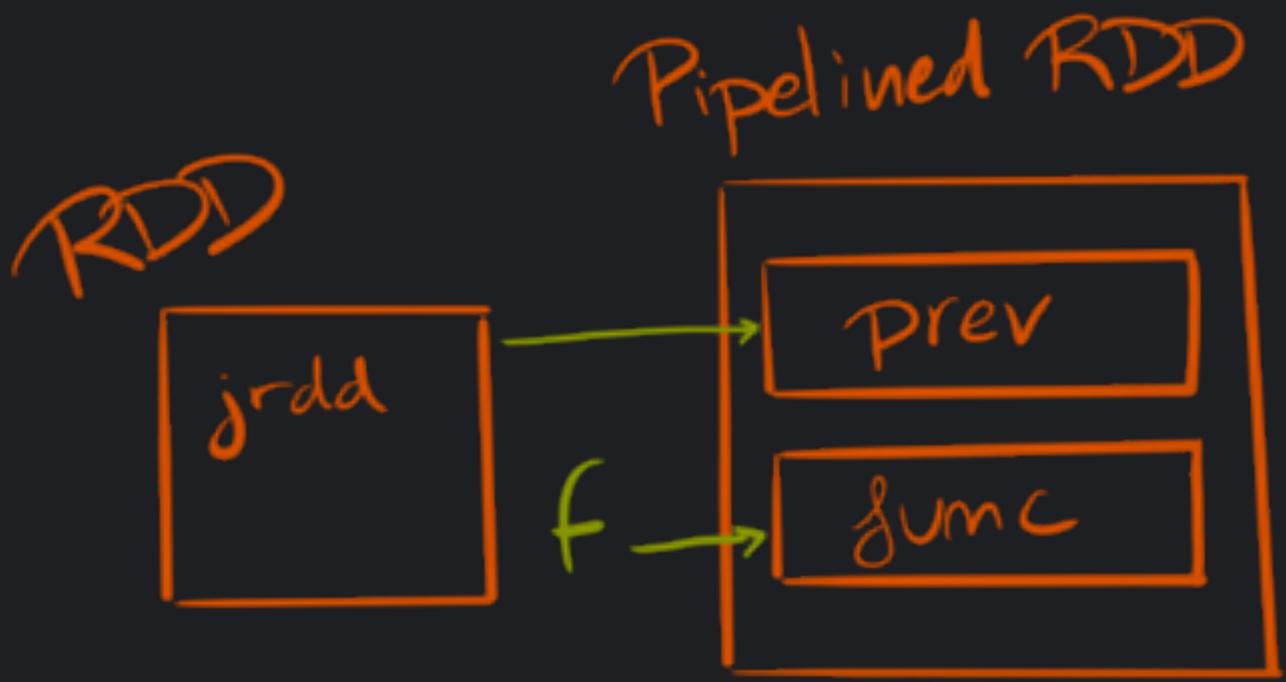
With its `jrdd` (so, it's in Python land)

Let's map over this RDD

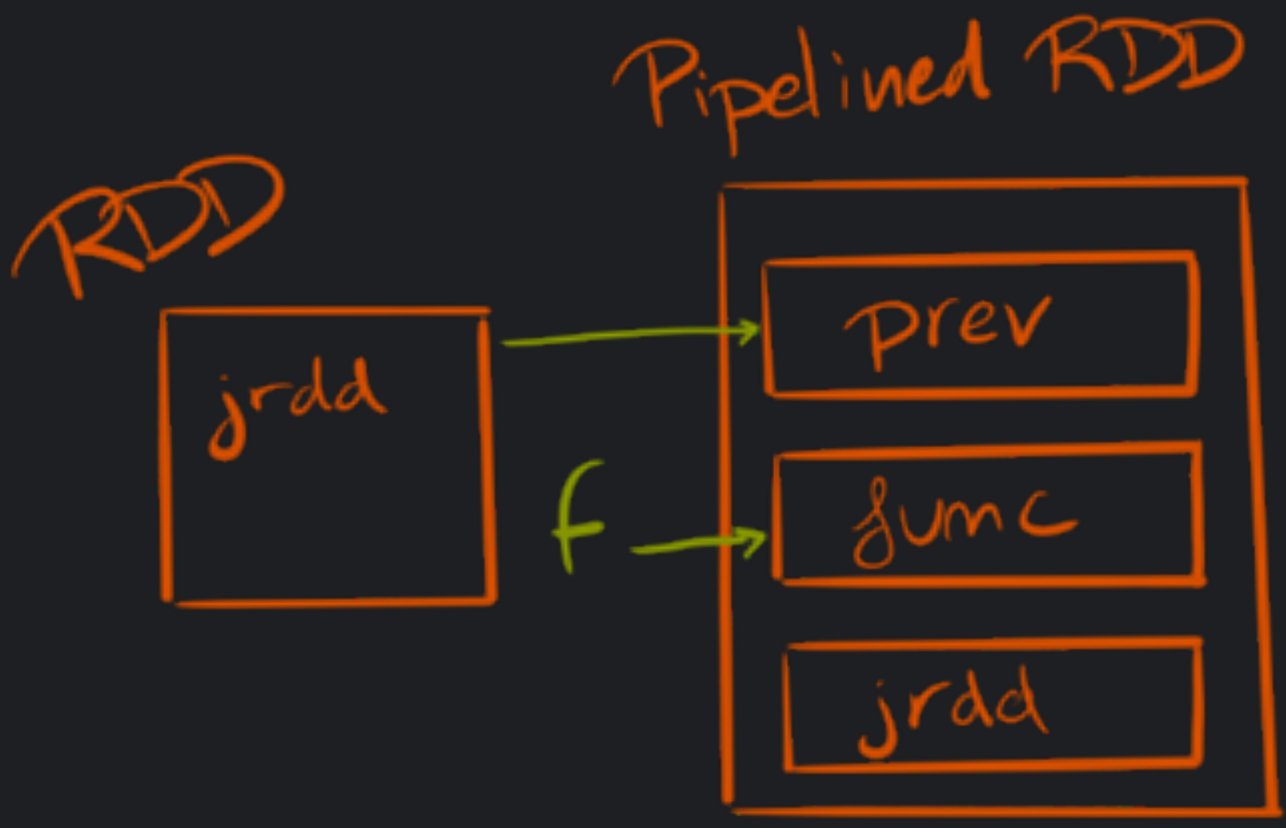
RDD's `map` will create a `PipelinedRDD`



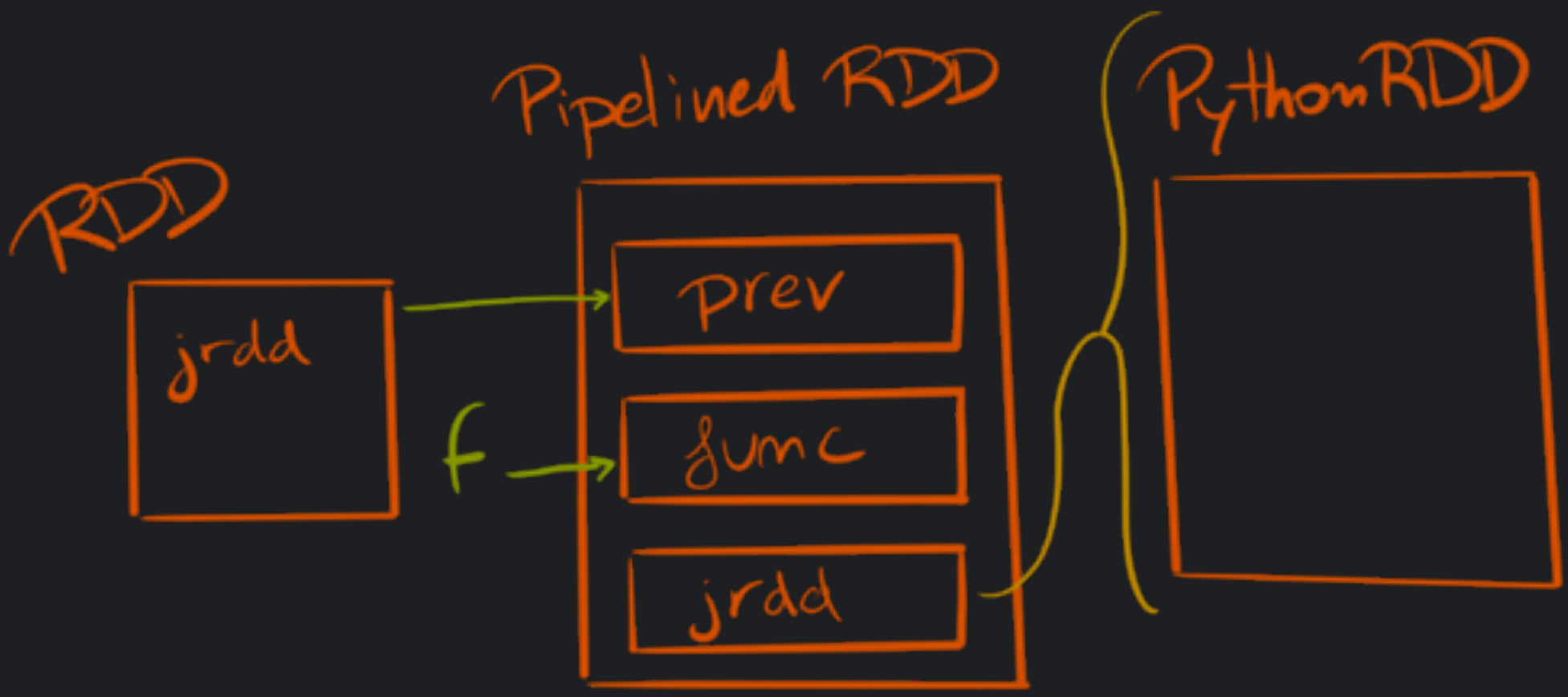
Will put the function in the `func` field



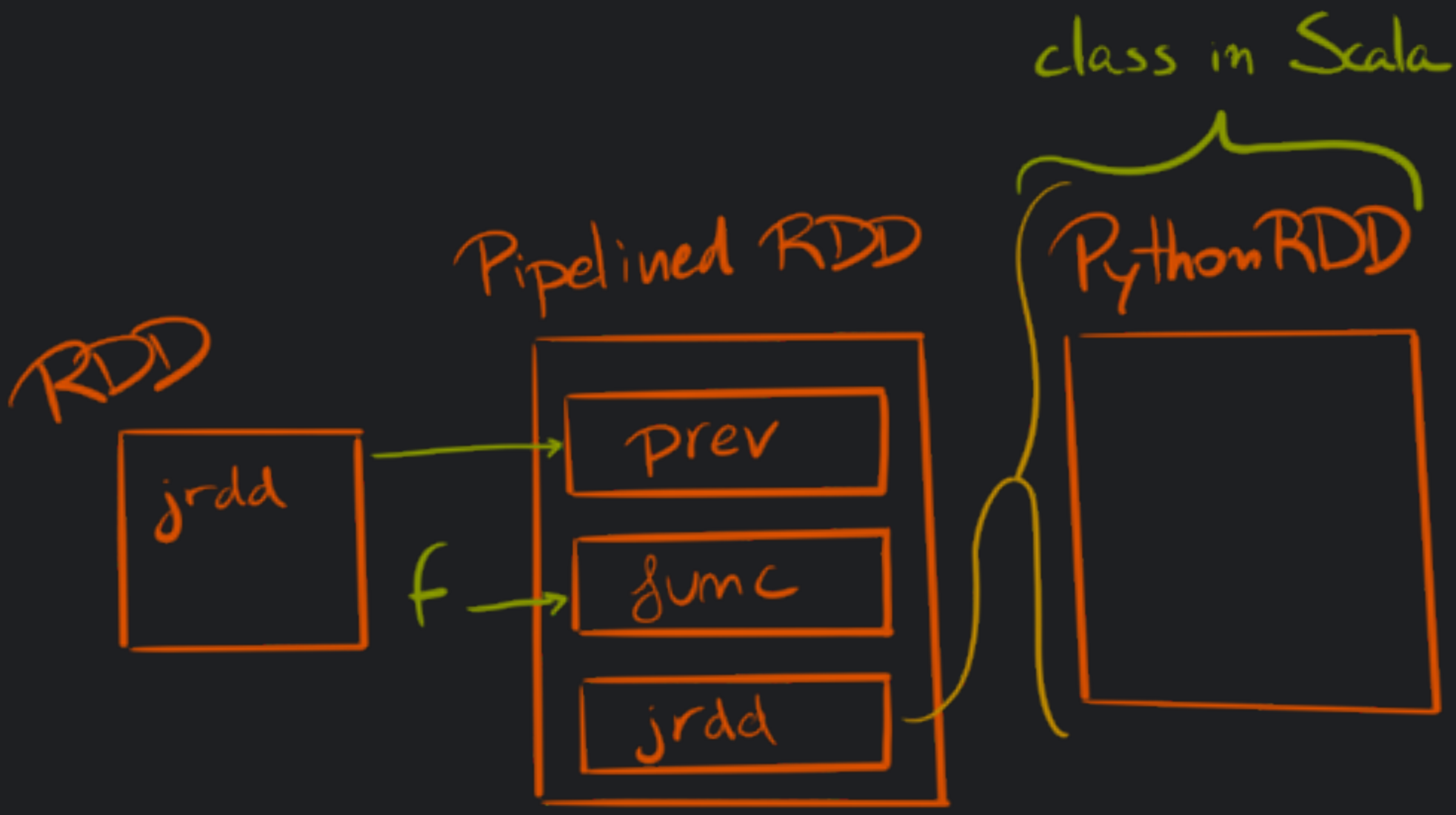
And will point `prev` to the RDD. Missing anything?

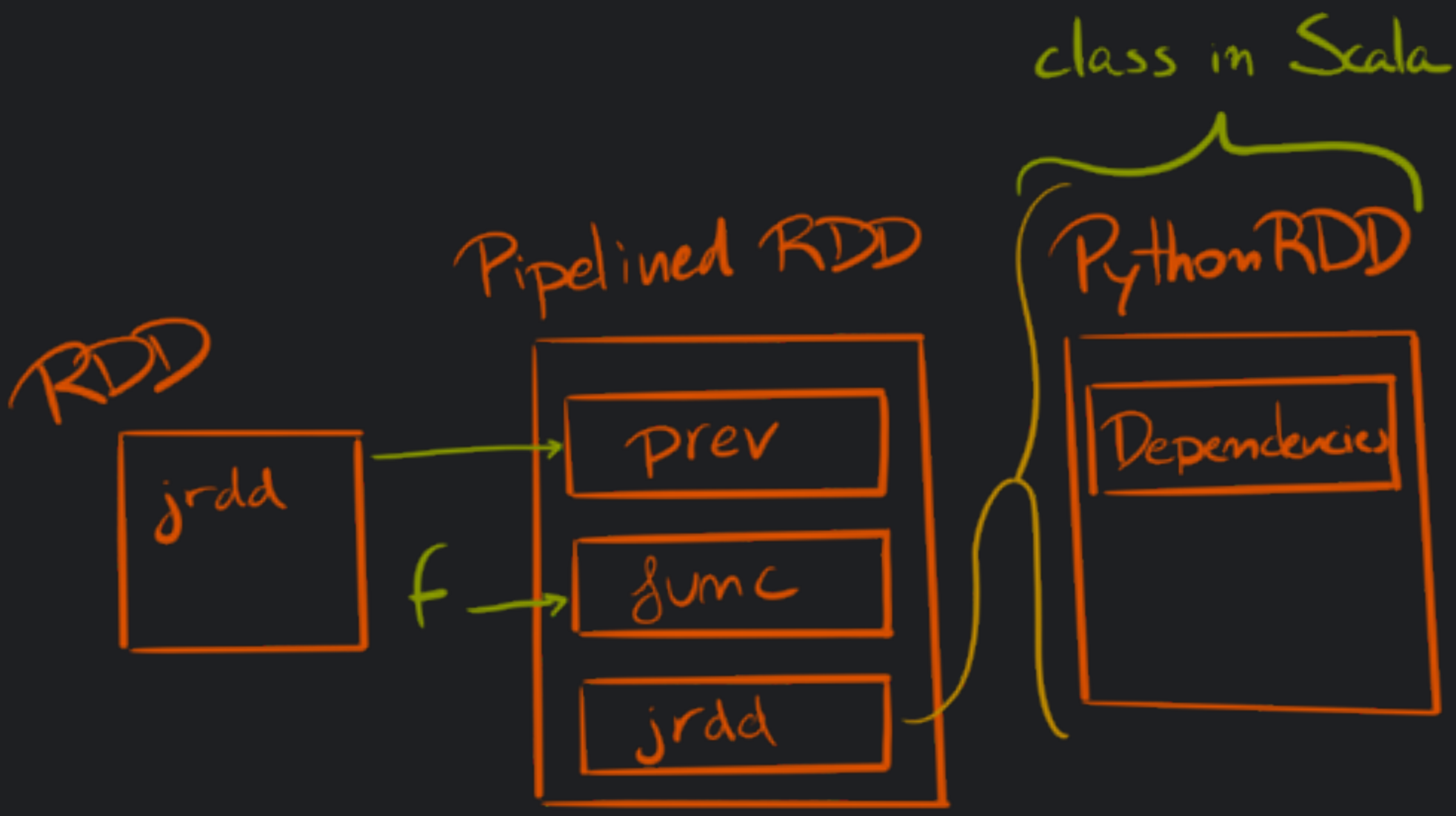


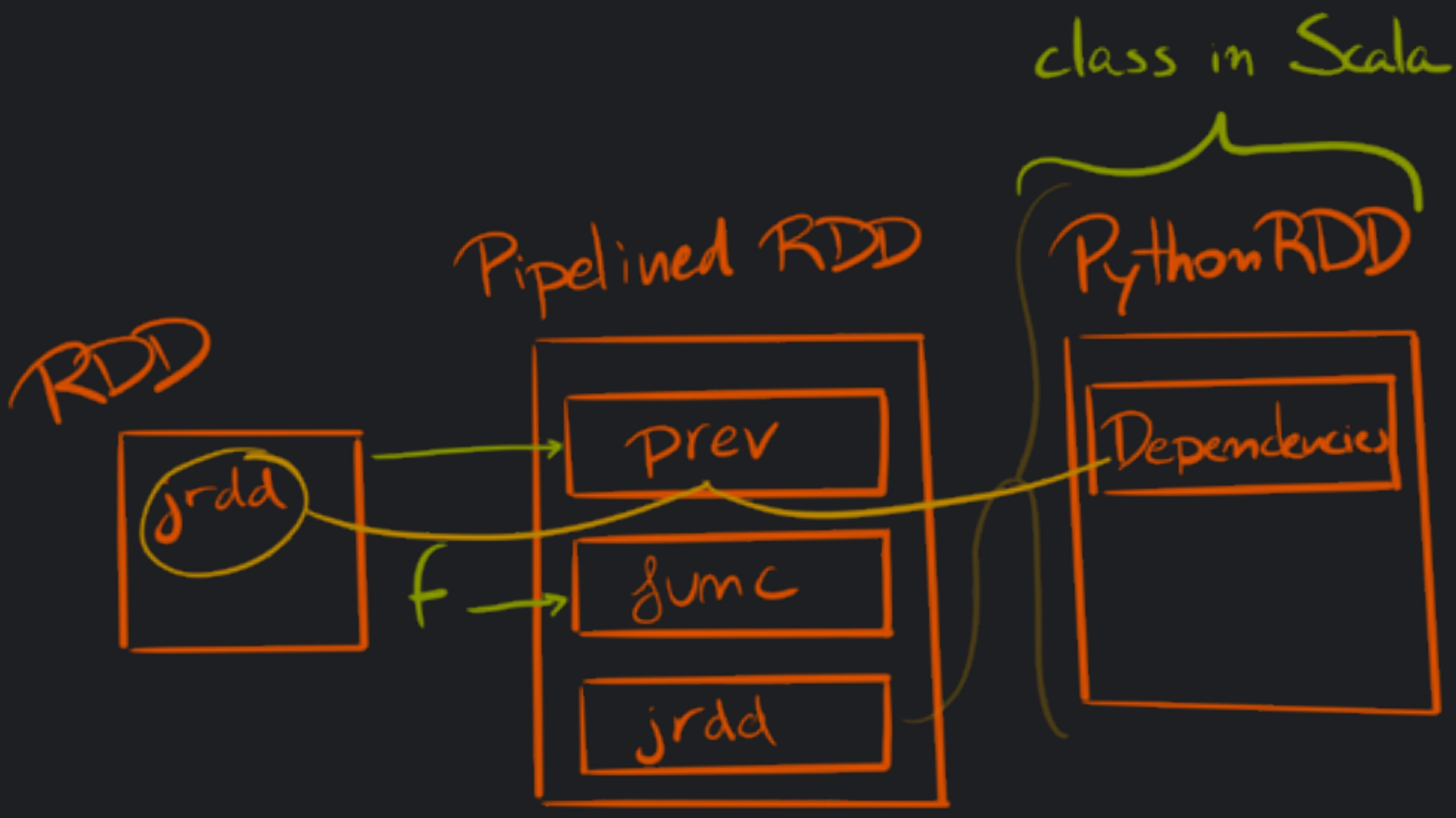
And now, what is the _jrdd of the PipelinedRDD?



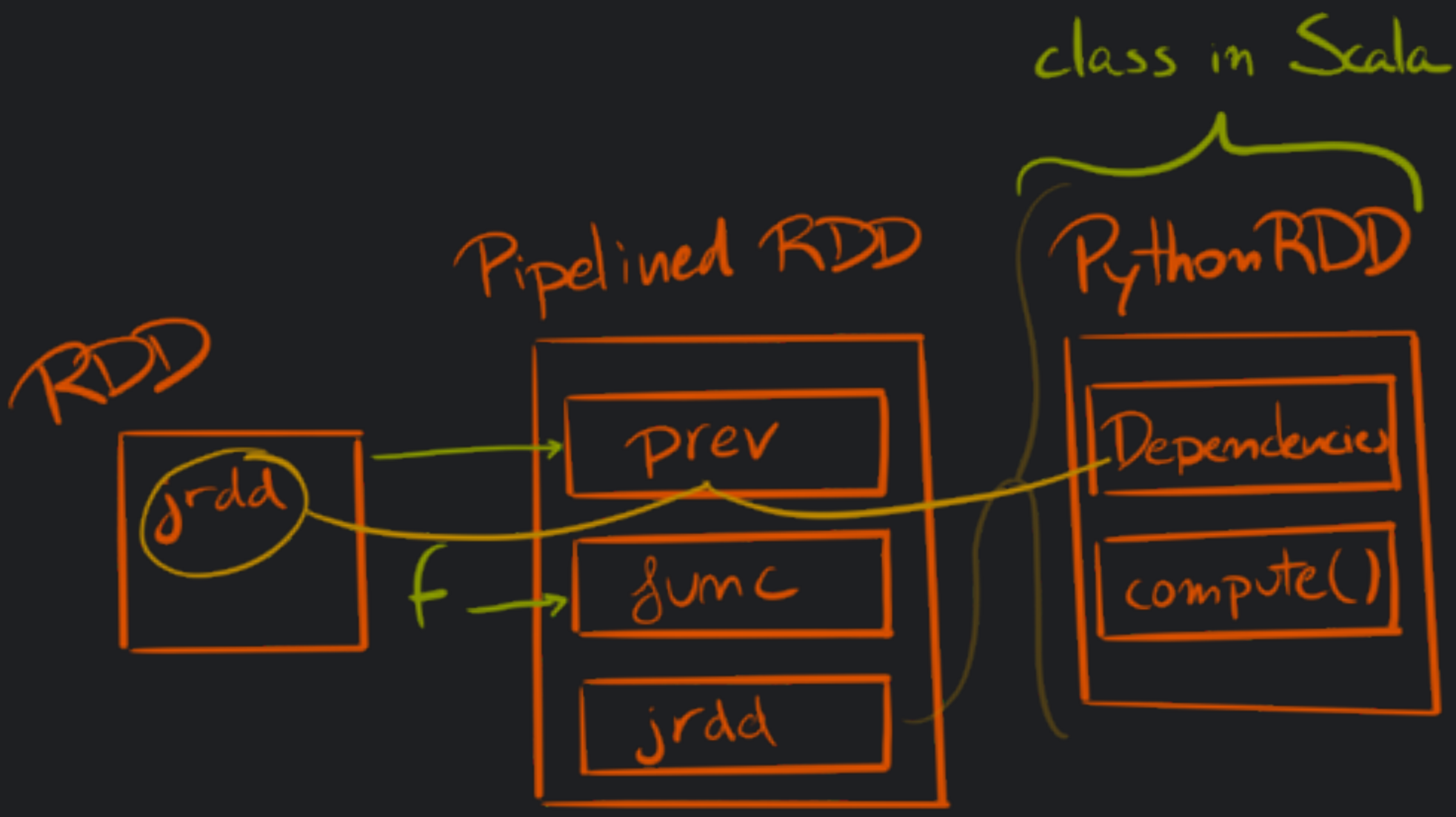
It's a PythonRDD (built via Py4J of course, this is in the Scala code), which is a sub-class of RDD

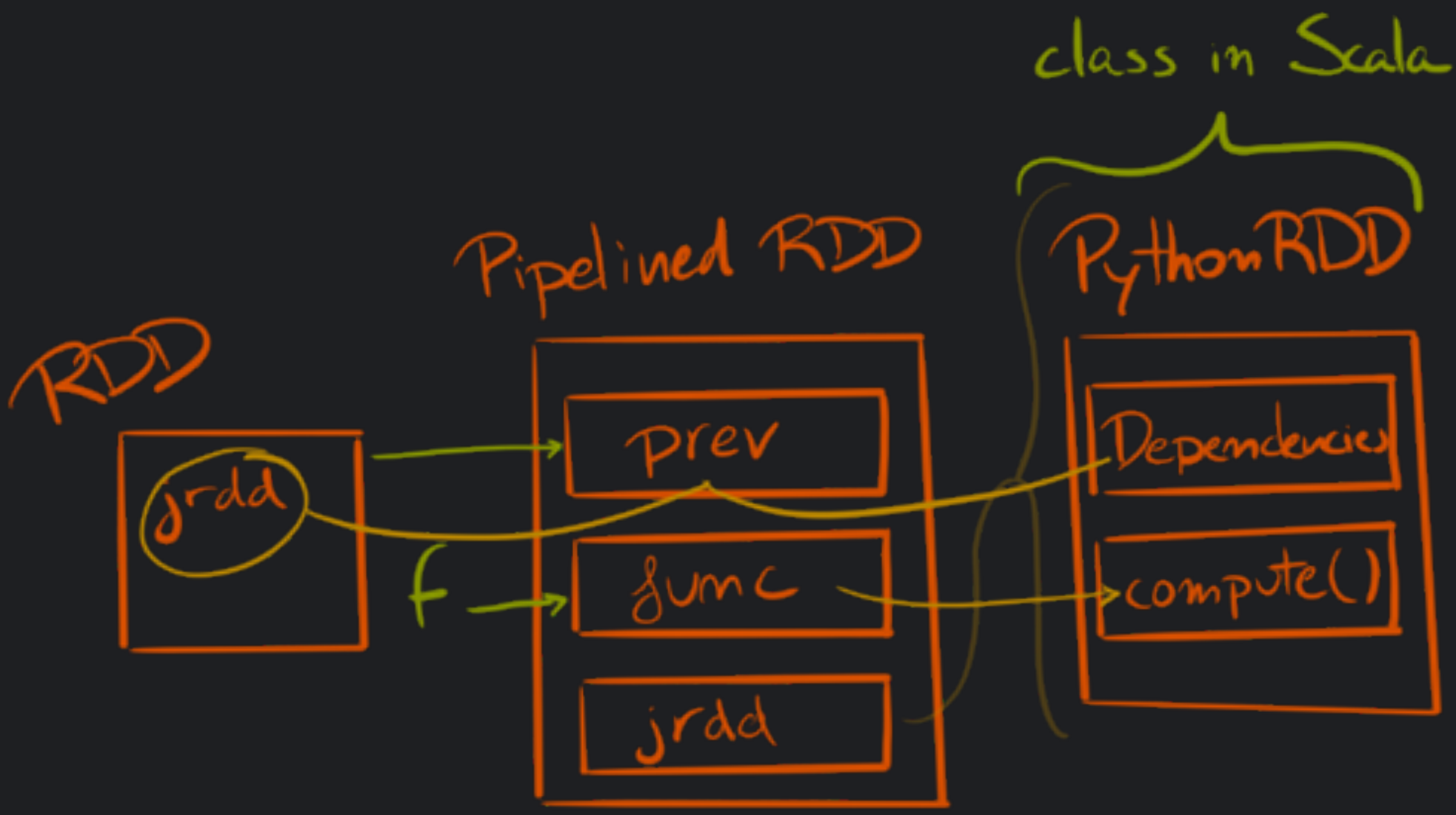

class in Scala
Pipelined RDD
PythonRDD
RDD
jrdd
prev
f
func
jrdd


class in Scala
Pipelined RDD
PythonRDD
RDD
jrdd
prev
f
func
jrdd
Dependencies



The dependencies of this RDD will point to the original _jrdd


class in Scala
RDD
jrdd
f
Pipelined RDD
prev
func
jrdd
PythonRDD
Dependencies
compute()



The compute method will wrap the function defined in func

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

compute

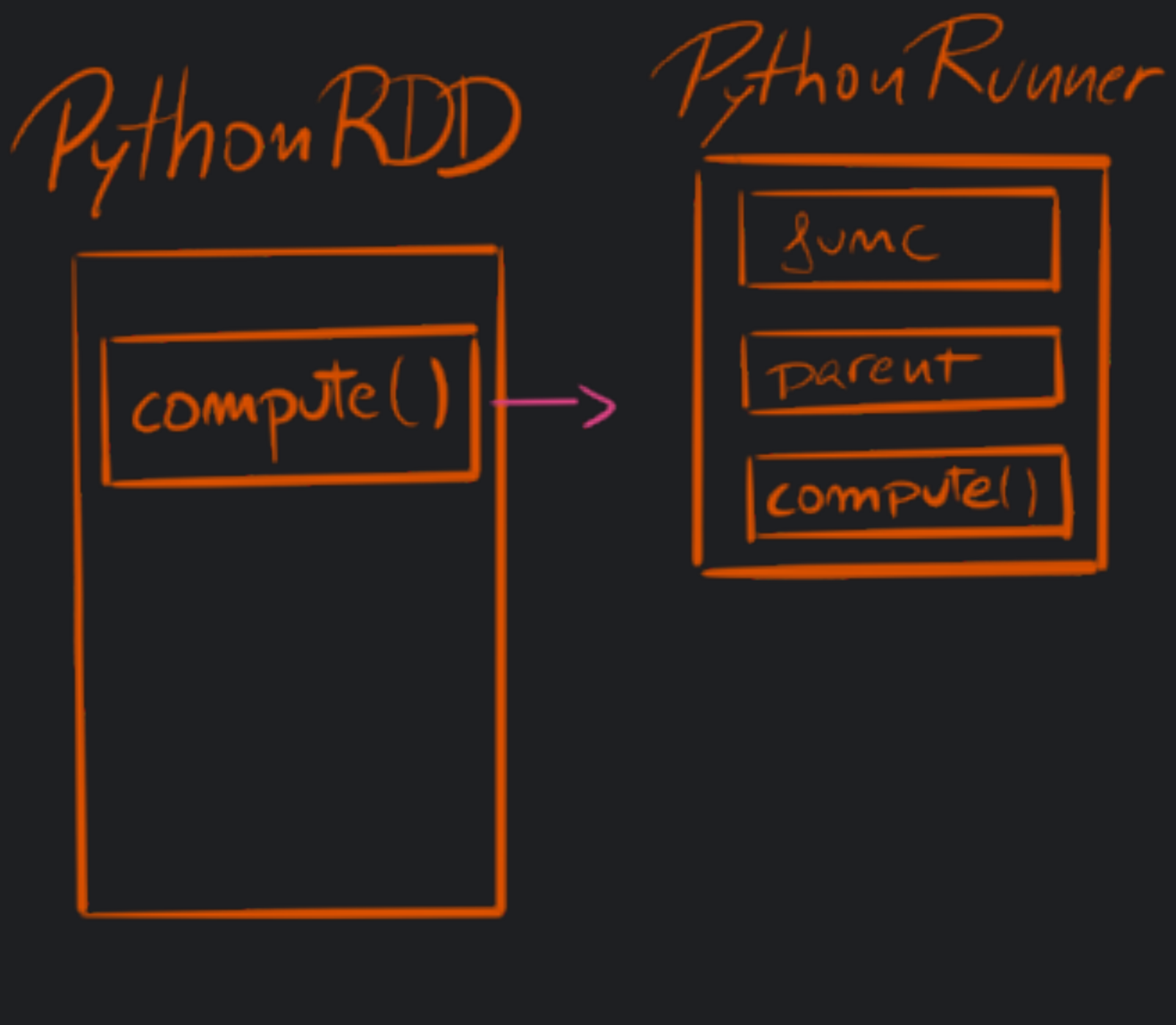
IS RUN ON EACH EXECUTOR AND STARTS A PYTHON WORKER VIA PythonRunner

SPARK+AI SUMMIT 2019

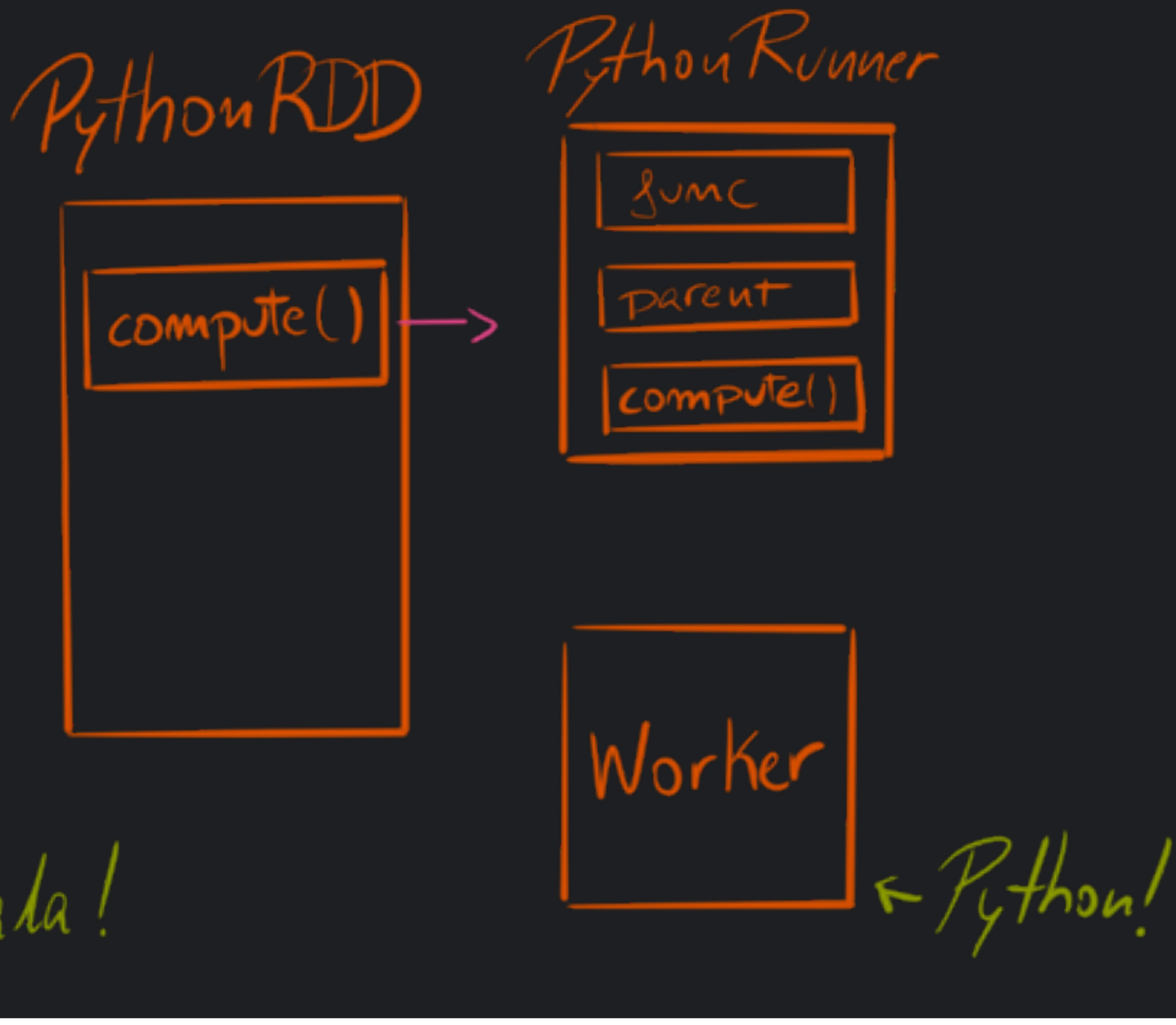
It will send all includes, broadcasts, etc through the stream. And actually the order of the data sent is important

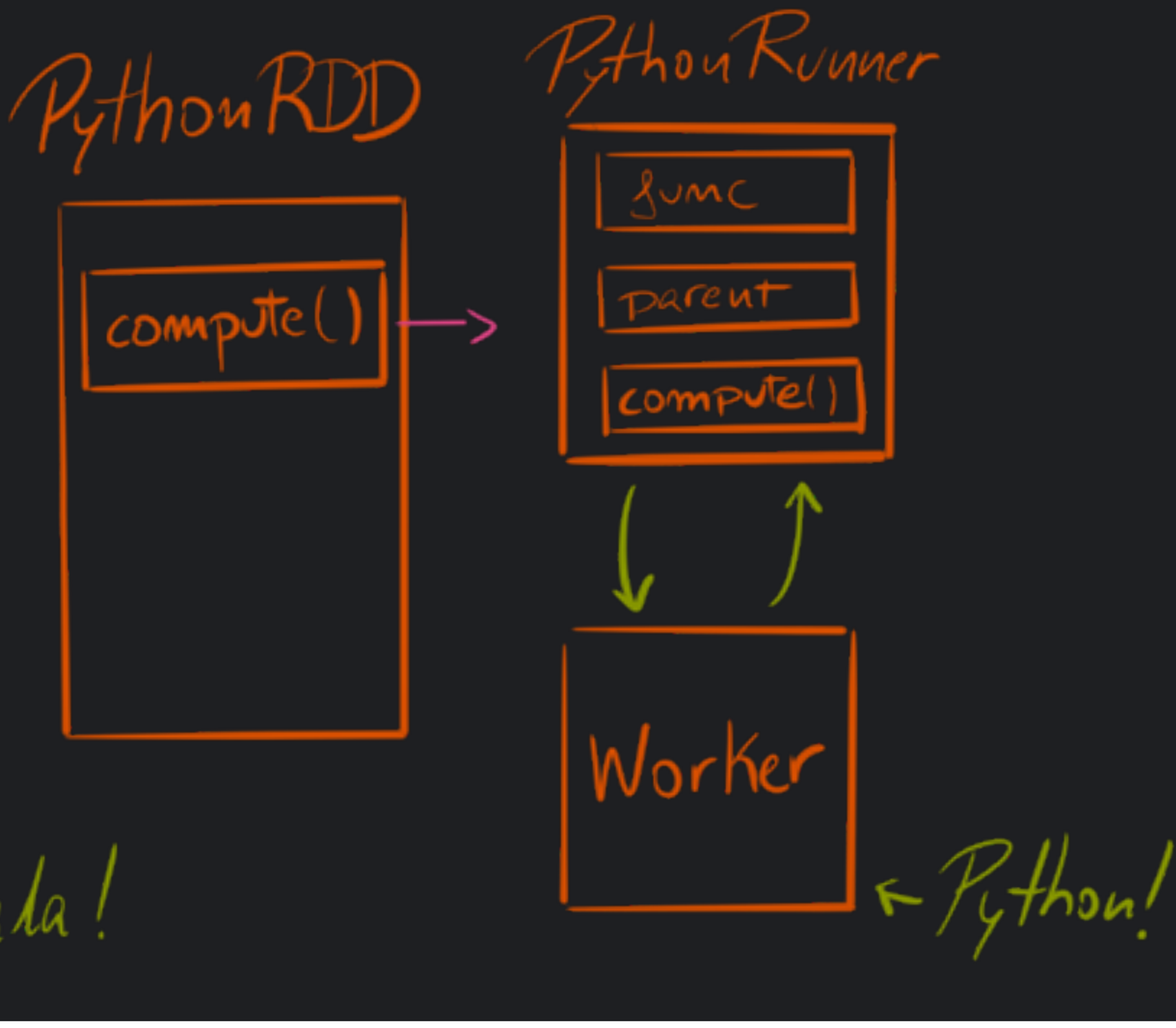
PythonRDD

Scala!



Scala!


PythonRDD
compute()
PythonRunner
func
parent
compute()
Worker
Scala!
Python!


PythonRDD
compute()
PythonRunner
func
parent
compute()
Worker
Scala!
Python!

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

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

SPARK+AI SUMMIT 2019

The Catalyst optimiser. There's also a code generator in there (using Janino to compile Java code in real time, how that works and why is a matter for another presentation...) Catalyst prunes trees

DEPENDING ON THE FUNCTION, THE OPTIMISER WILL CHOOSE

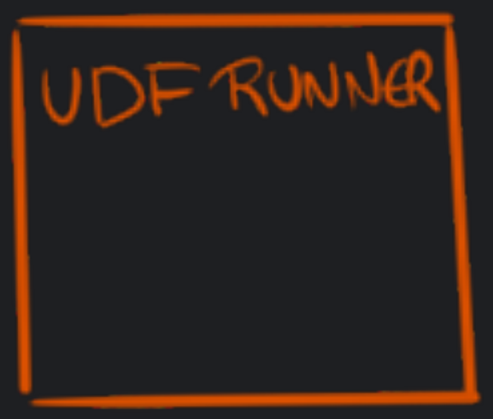
PythonUDFRunner

OR

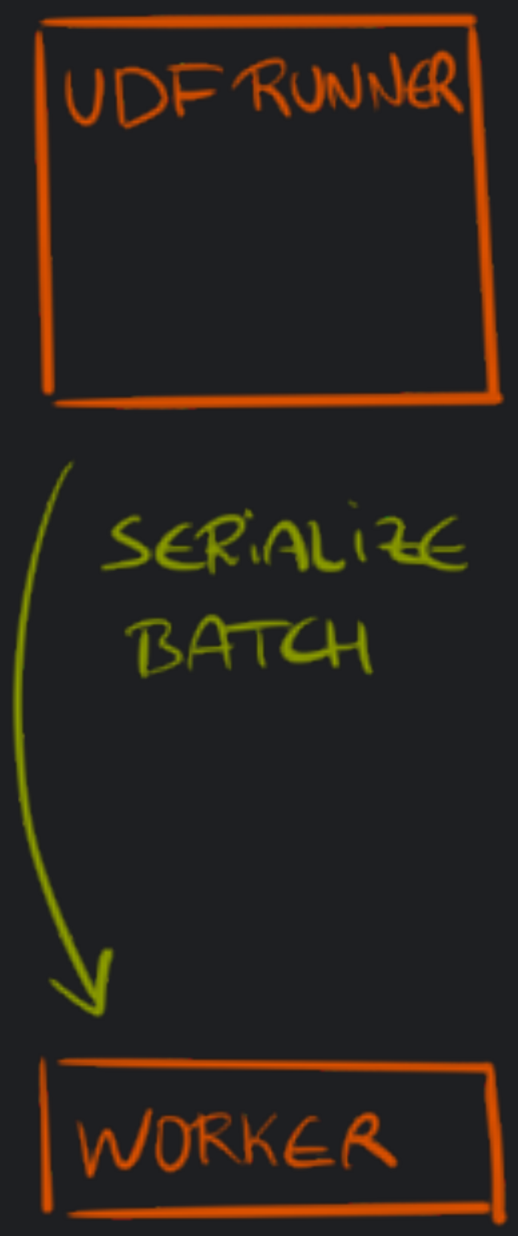
PythonArrowRunner

(BOTH EXTEND PythonRunner)

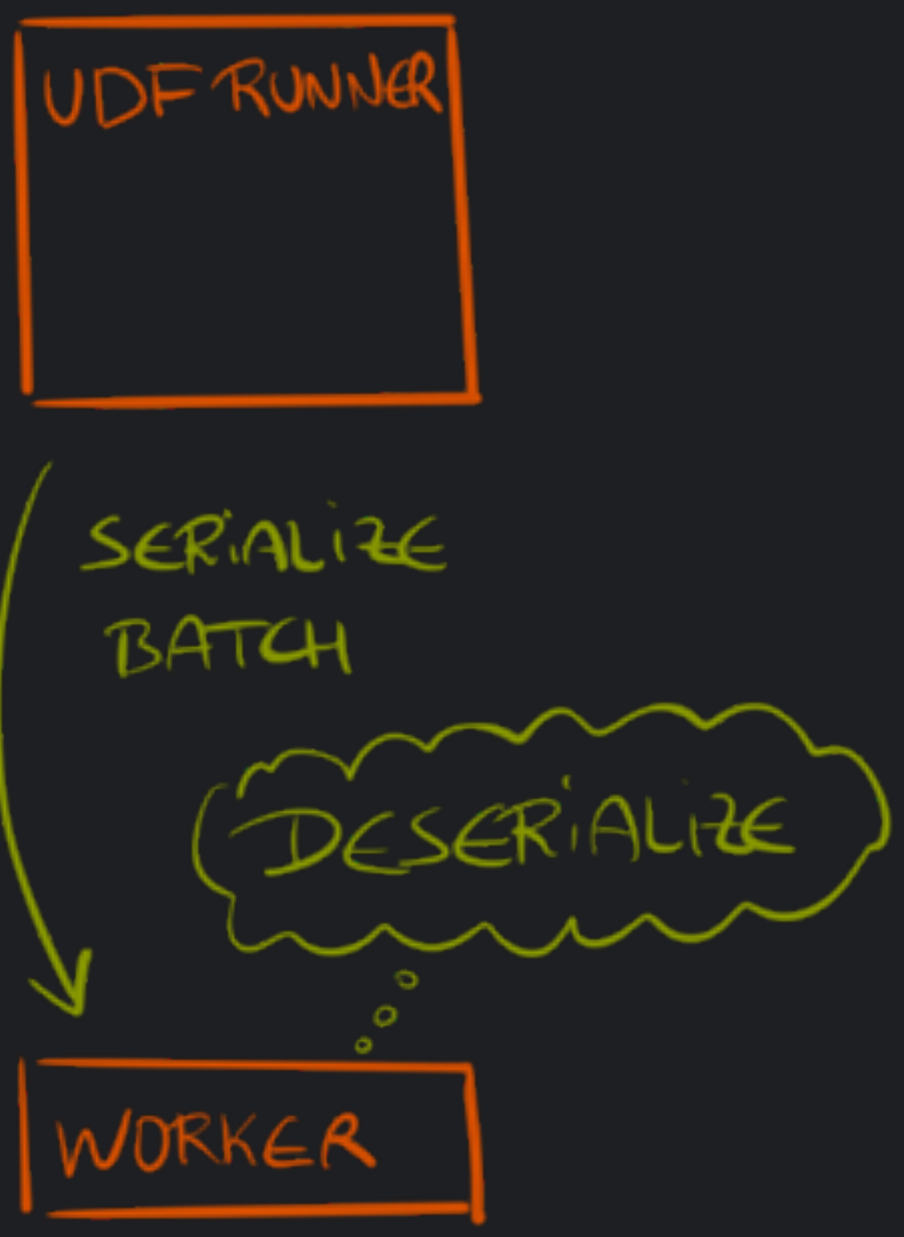
For the physical execution node in the plan


UDF RUNNER

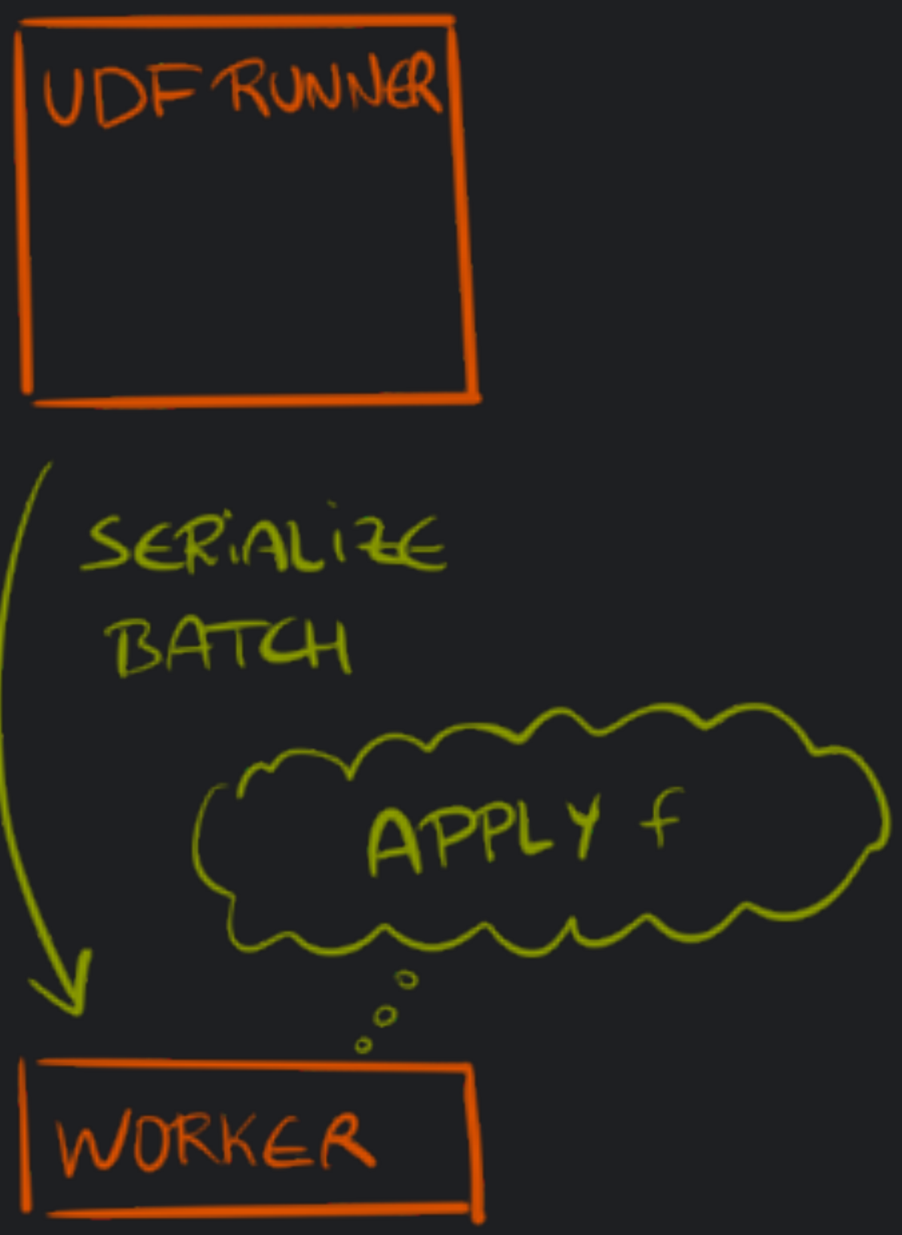

WORKER



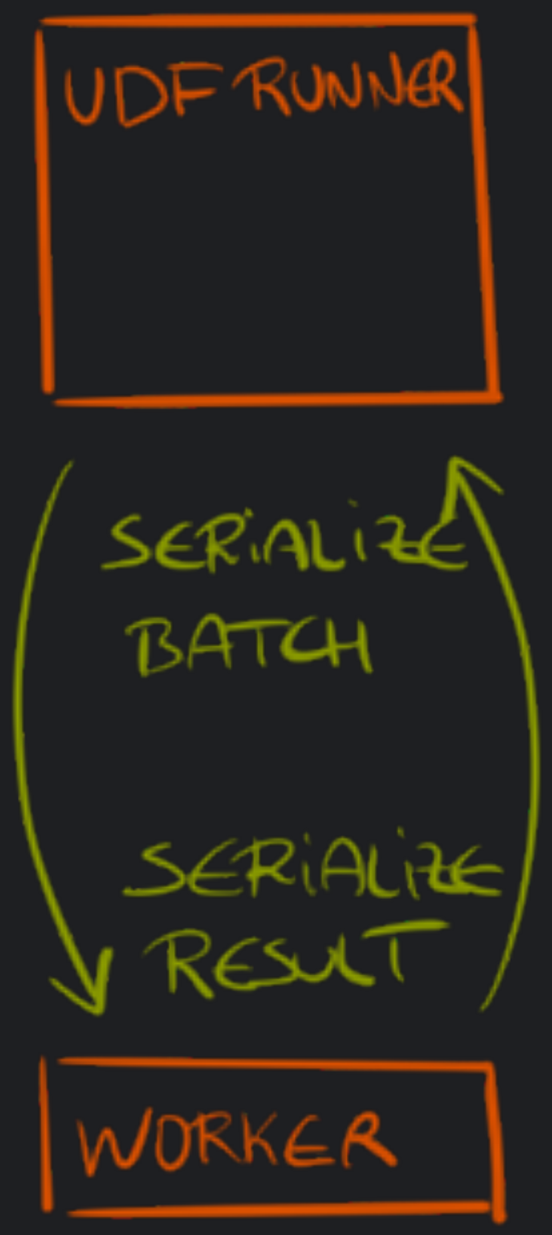
In the stream information about environment, length of data, versions, etc will be sent first. Then data will be serialized in batches (of 100 rows I think) using a Pickle implementation in Java (net.razorvine.pickle.{Pickler, Unpickler})



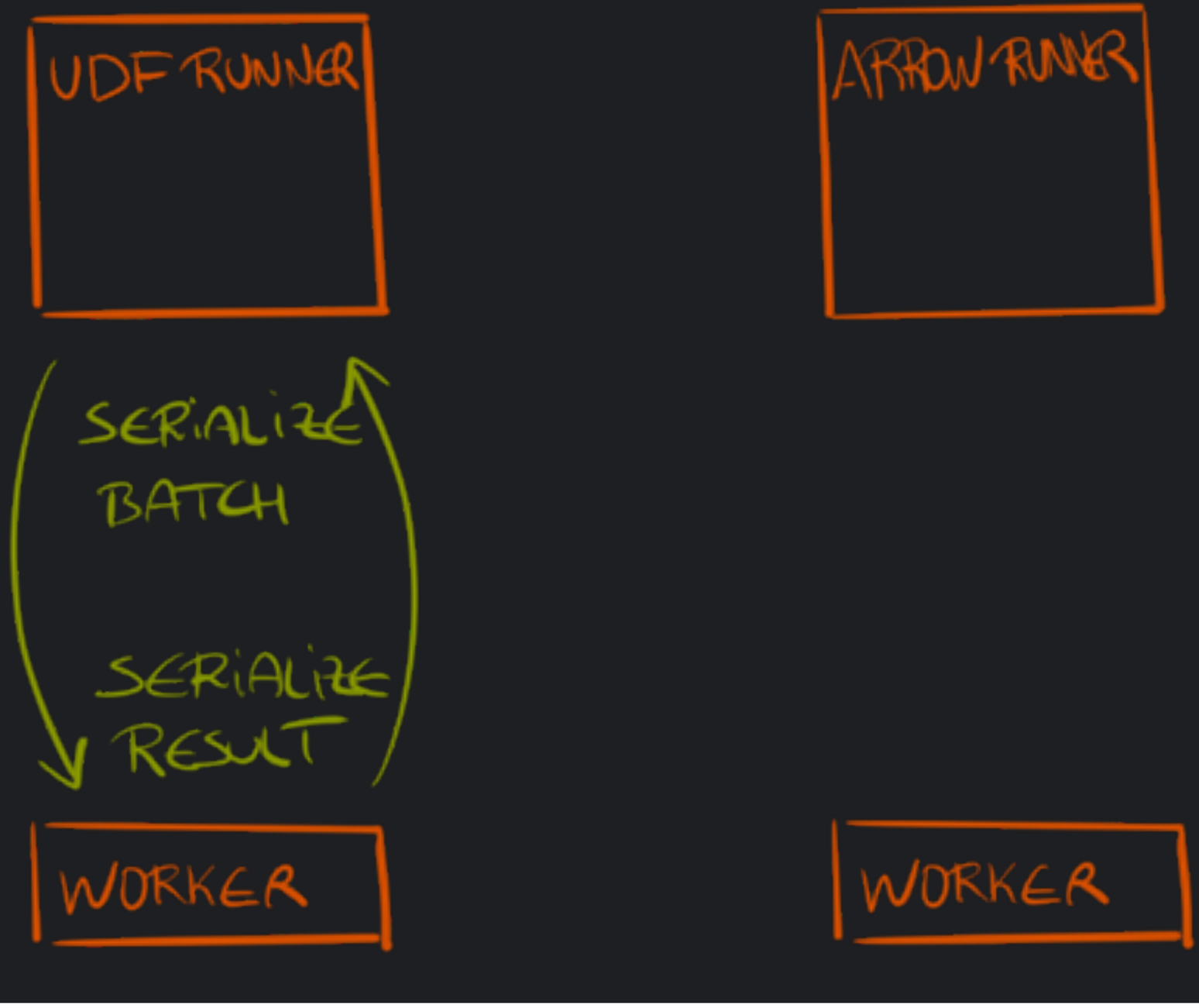
In Python land, after loading all the configuration and startup stuff, the data is unpickled



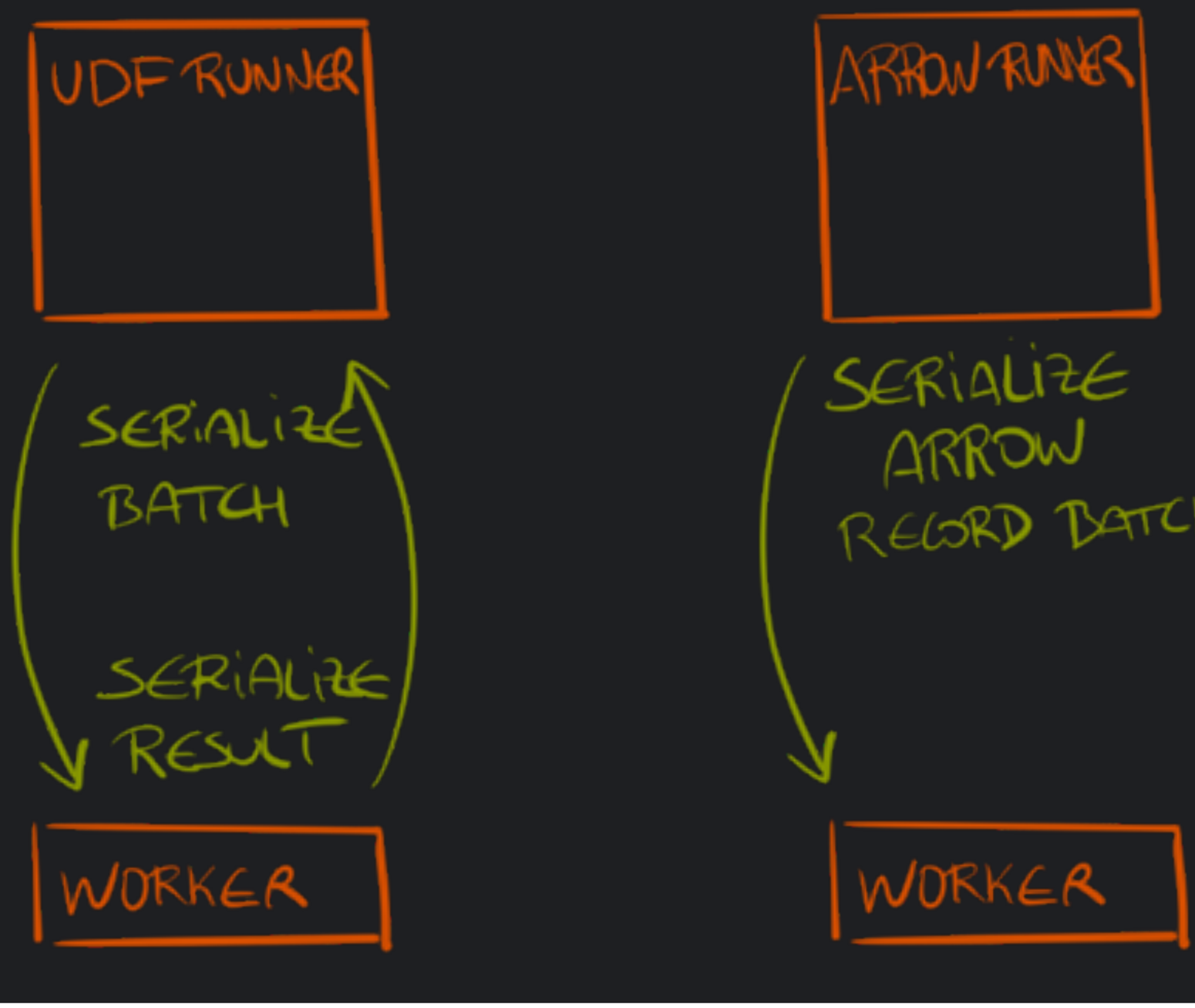
The stream is loaded as an iterator, and the function is applied to each item in the batch



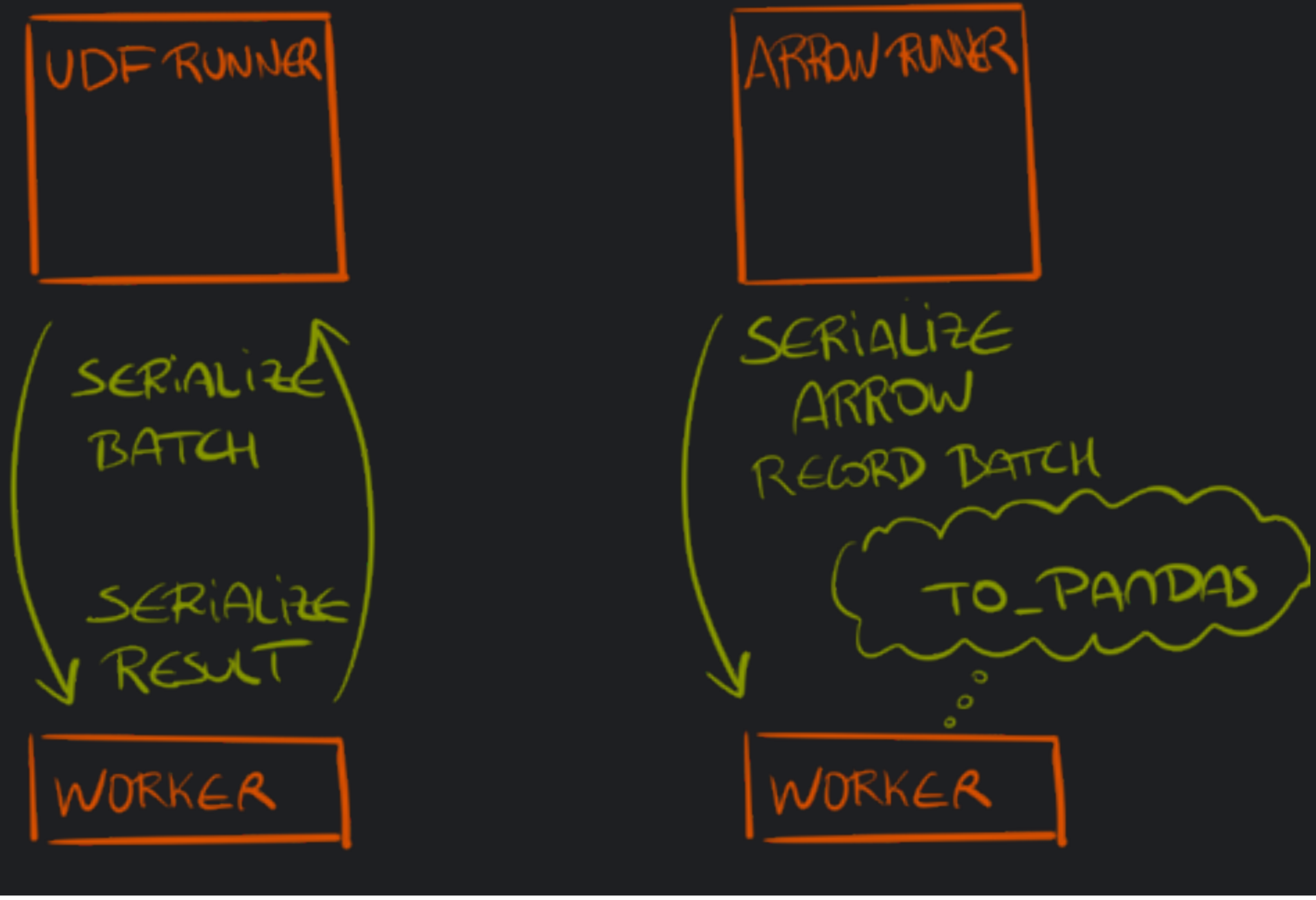
Then the data is serialized back to send to the JVM



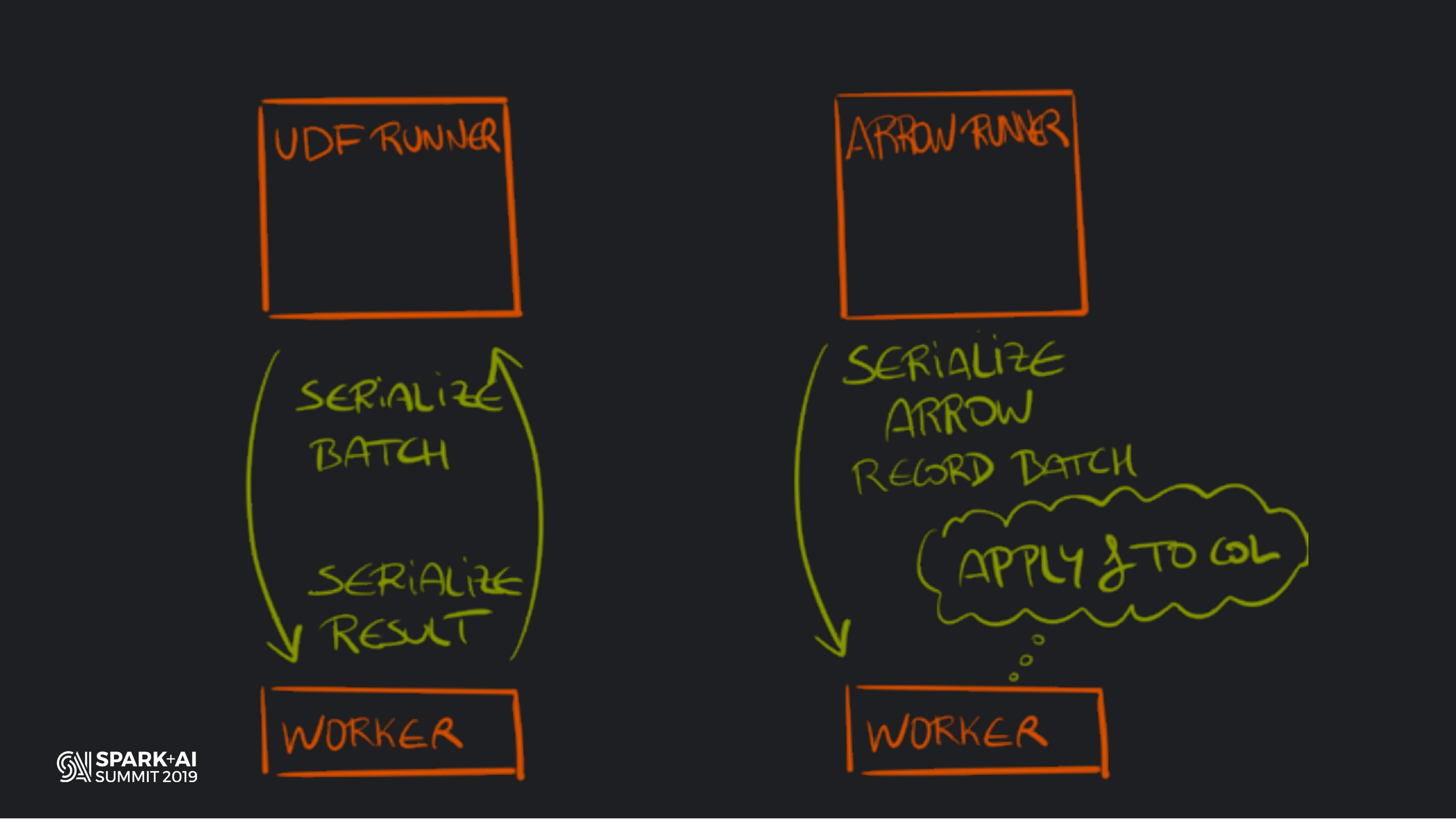
Arrow works a bit different, but not much



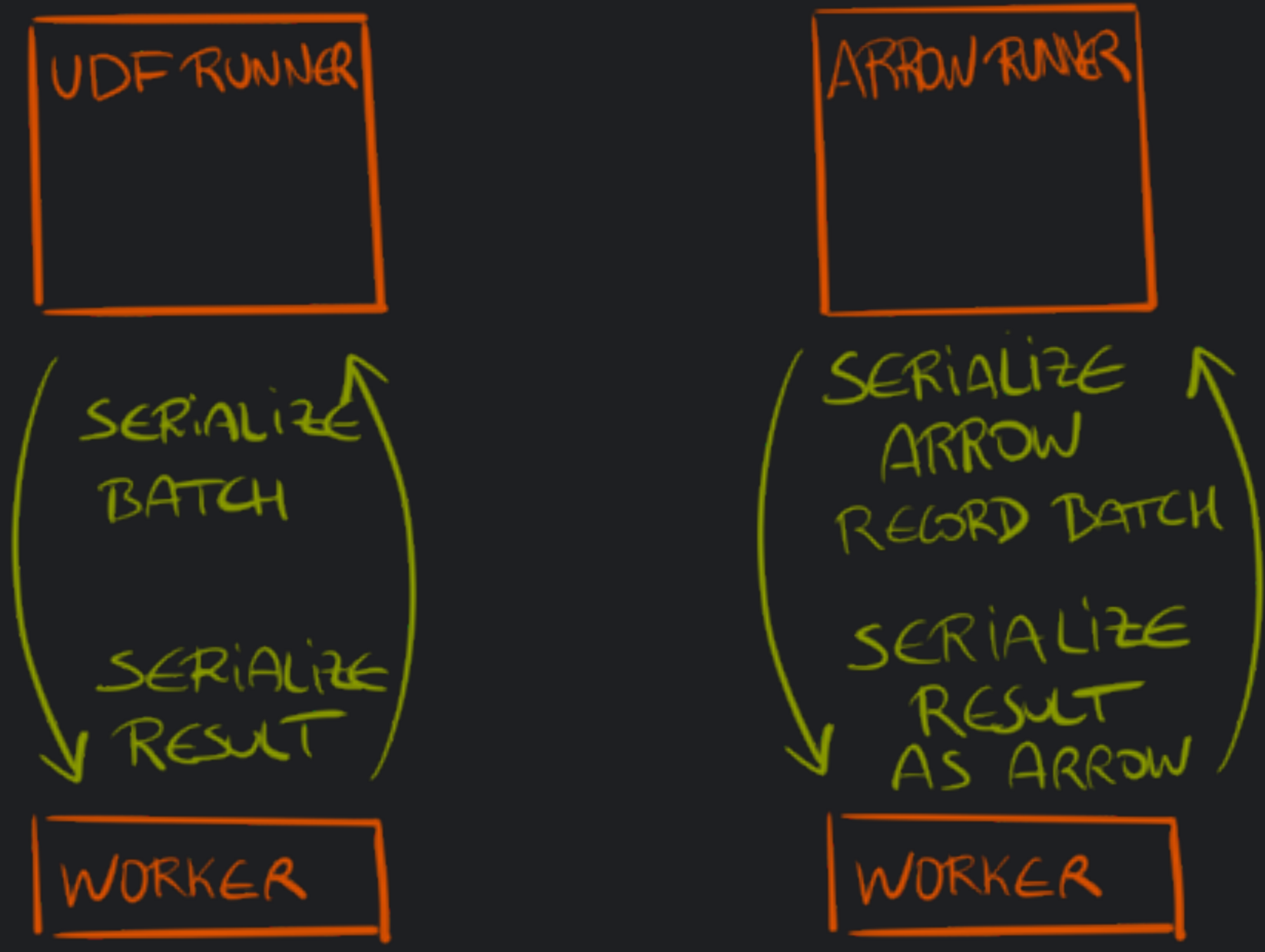
Data is sent as Arrow RecordBatch-es



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. There are other columnar solutions you can plug into Spark directly for
working only in Scala, but they also work as data stores as well (see Apache
Kudu and partially, Apache
Ignite). There is also ongoing work for
exposing to the user an API of columnar data in Spark in this SPIP. This would make
working with pure columnar data (for instance, Parquet) faster for advanced
users.



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

QUICK EXAMPLES

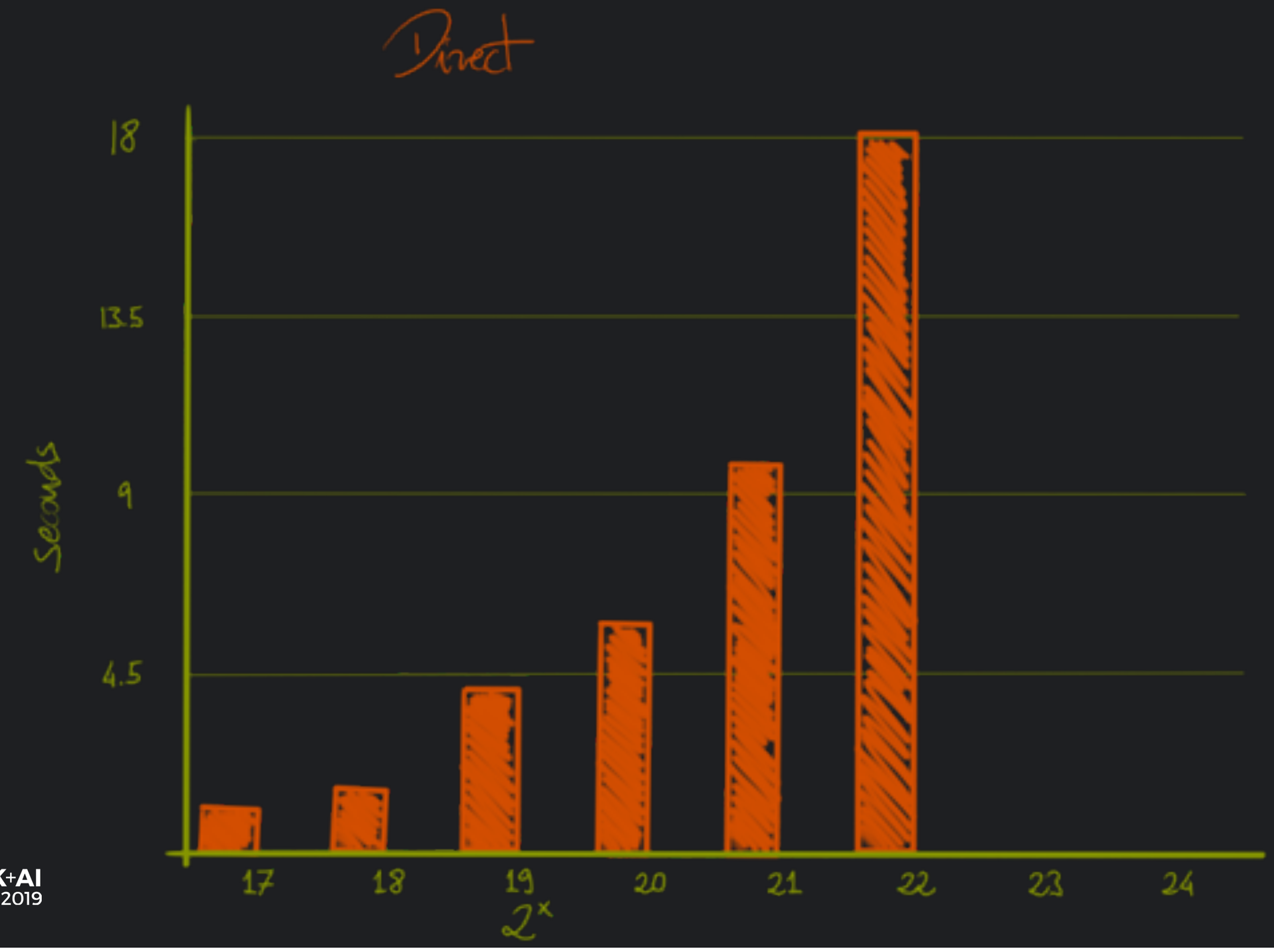
THE BASICS: toPandas

```
from pyspark.sql.functions import rand

df = spark.range(1 << 20).toDF("id").withColumn("x", rand())

spark.conf.set("spark.sql.execution.arrow.enabled", "false")
pandas_df = df.toPandas() #  we'll time this
```

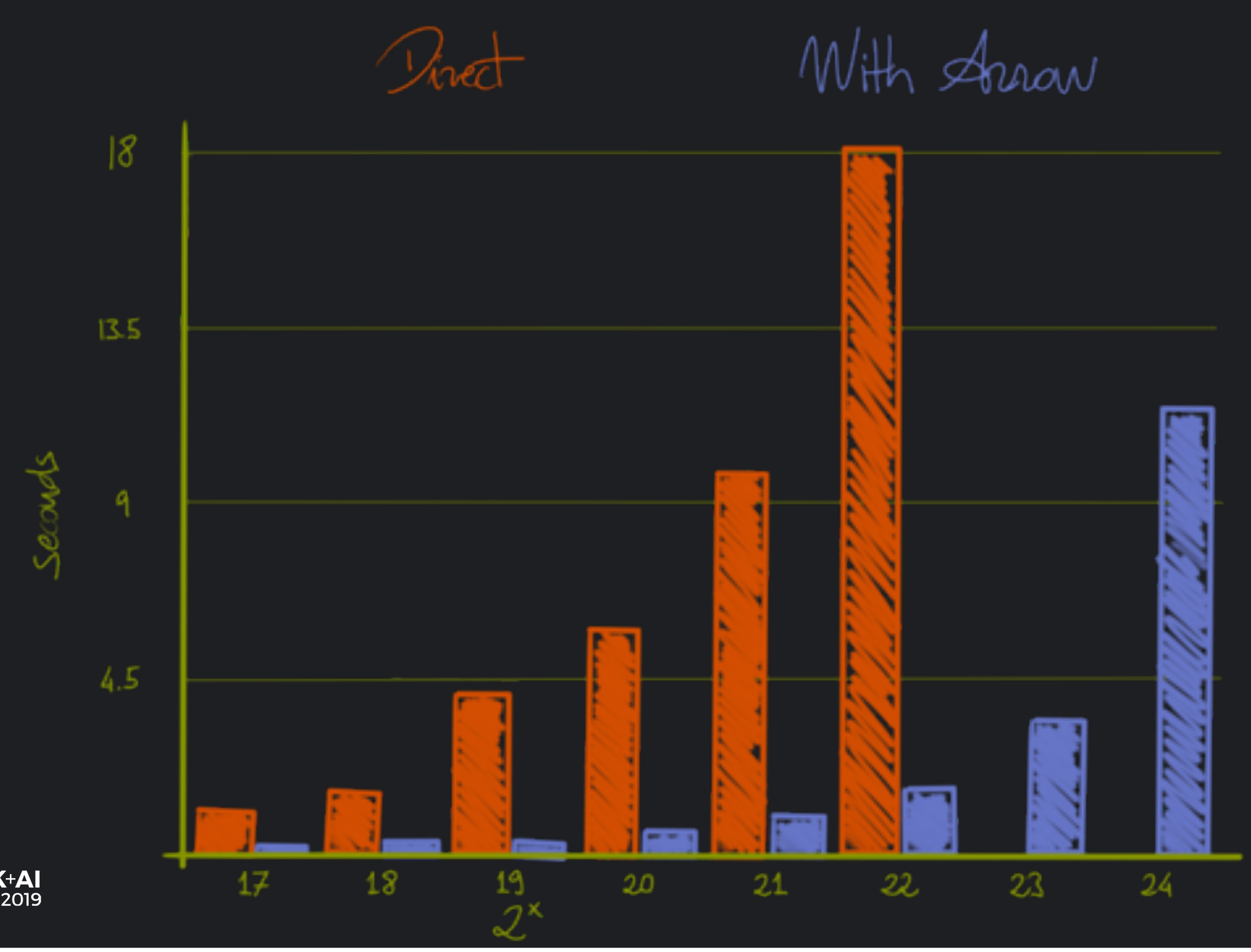

This comes directly from the speed comparisons by Bryan Cutler here

```
from pyspark.sql.functions import rand

df = spark.range(1 << 20).toDF("id").withColumn("x", rand())

spark.conf.set("spark.sql.execution.arrow.enabled", "false")
pandas_df = df.toPandas() #  we'll time this
```

```
from pyspark.sql.functions import rand

df = spark.range(1 << 20).toDF("id").withColumn("x", rand())

spark.conf.set("spark.sql.execution.arrow.enabled", "true")
pandas_df = df.toPandas() #  we'll time this
```



You can find the table at the end of the presentation. Direct conversion fails
for larger powers with either Py4j running out of memory or the driver running
out of heap with my local setup


Direct
With Arrow
Seconds
18
13.5
9
4.5
17
18
19
20
21
22
23
24
2^x

THE FUN: .groupBy

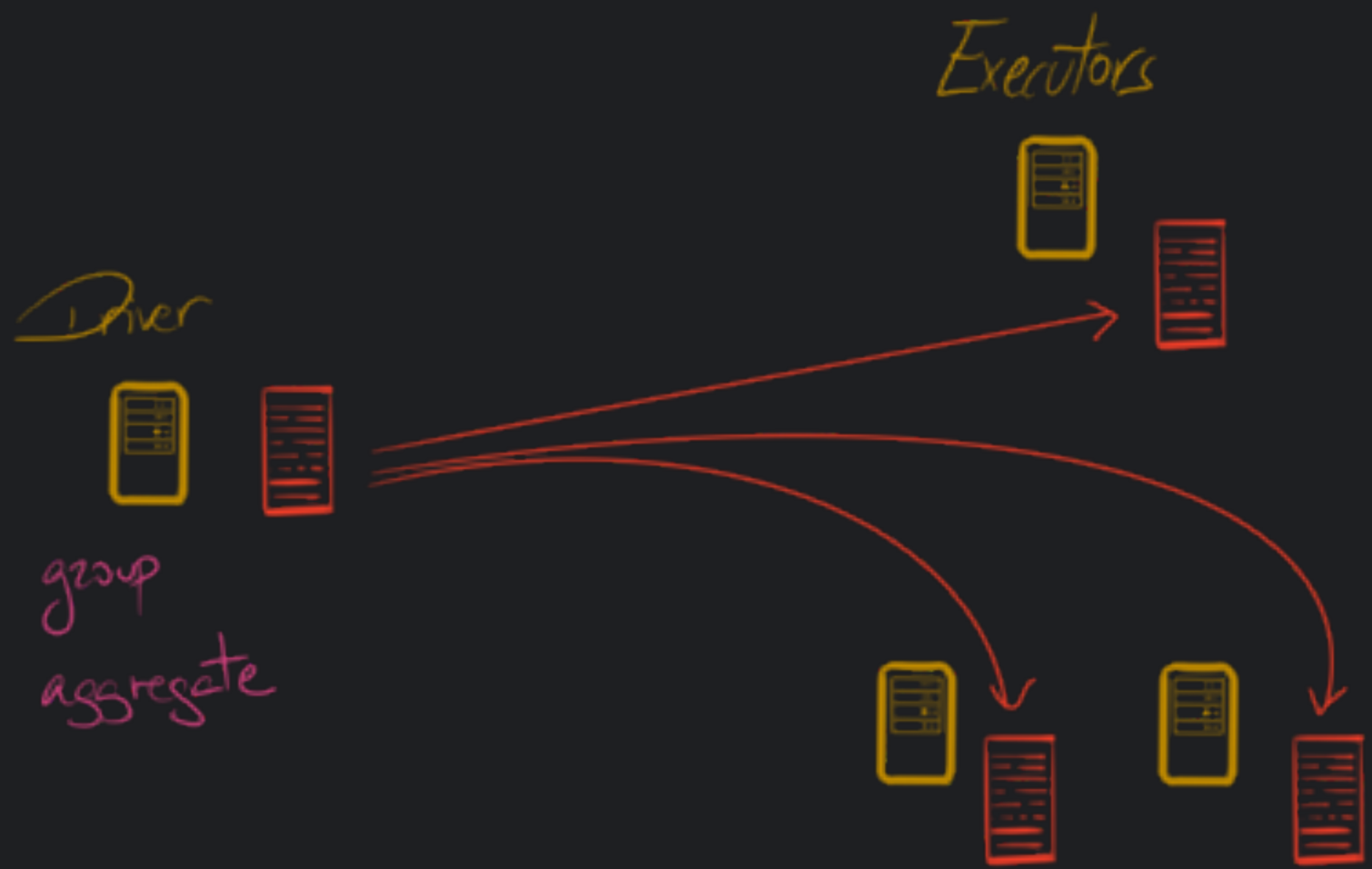
```
from pyspark.sql.functions import rand, randn, floor
from pyspark.sql.functions import pandas_udf, PandasUDFType

df = spark.range(20000000).toDF("row").drop("row") \
     .withColumn("id", floor(rand()*10000)).withColumn("spent", (randn()+3)*100)

@pandas_udf("id long, spent double", PandasUDFType.GROUPED_MAP)
def subtract_mean(pdf):
    spent = pdf.spent
    return pdf.assign(spent=spent - spent.mean())

df_to_pandas_arrow = df.groupby("id").apply(subtract_mean).toPandas()
```


We have a list of transactions and we want to see how they deviate from the mean for the corresponding user

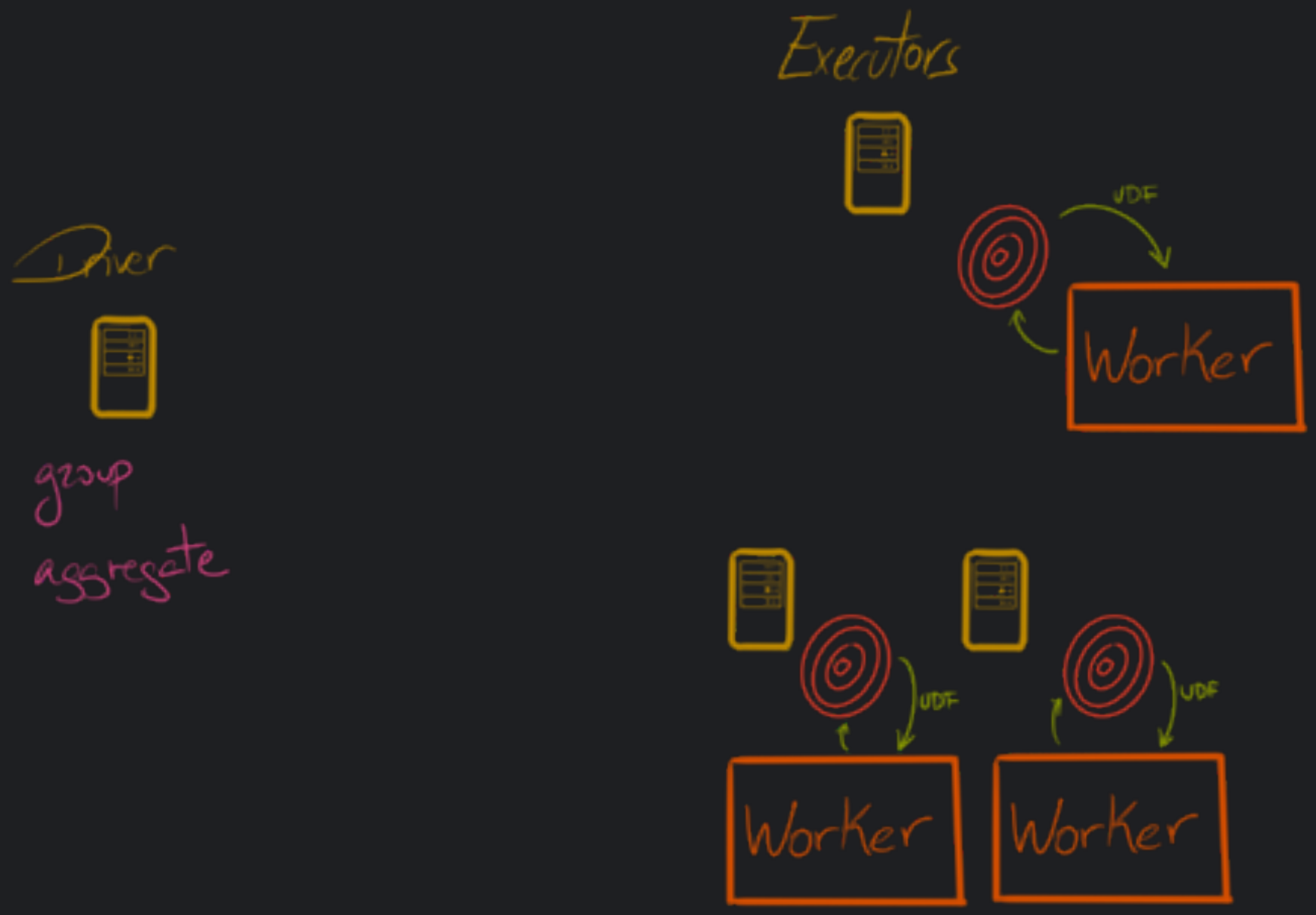
```python
from pyspark.sql.functions import rand, randn, floor
from pyspark.sql.functions import pandas_udf, PandasUDFType

df = spark.range(20000000).toDF("row").drop("row") \
     .withColumn("id", floor(rand()*10000)).withColumn("spent", (randn()+3)*100)

@pandas_udf("id long, spent double", PandasUDFType.GROUPED_MAP)
def subtract_mean(pdf):
    spent = pdf.spent
    return pdf.assign(spent=spent - spent.mean())

df_to_pandas_arrow = df.groupby("id").apply(subtract_mean).toPandas()
```


It's straightforward with a GROUPED_MAP UDF. This takes around 180s for 1<<25 records, 75s for 20M records

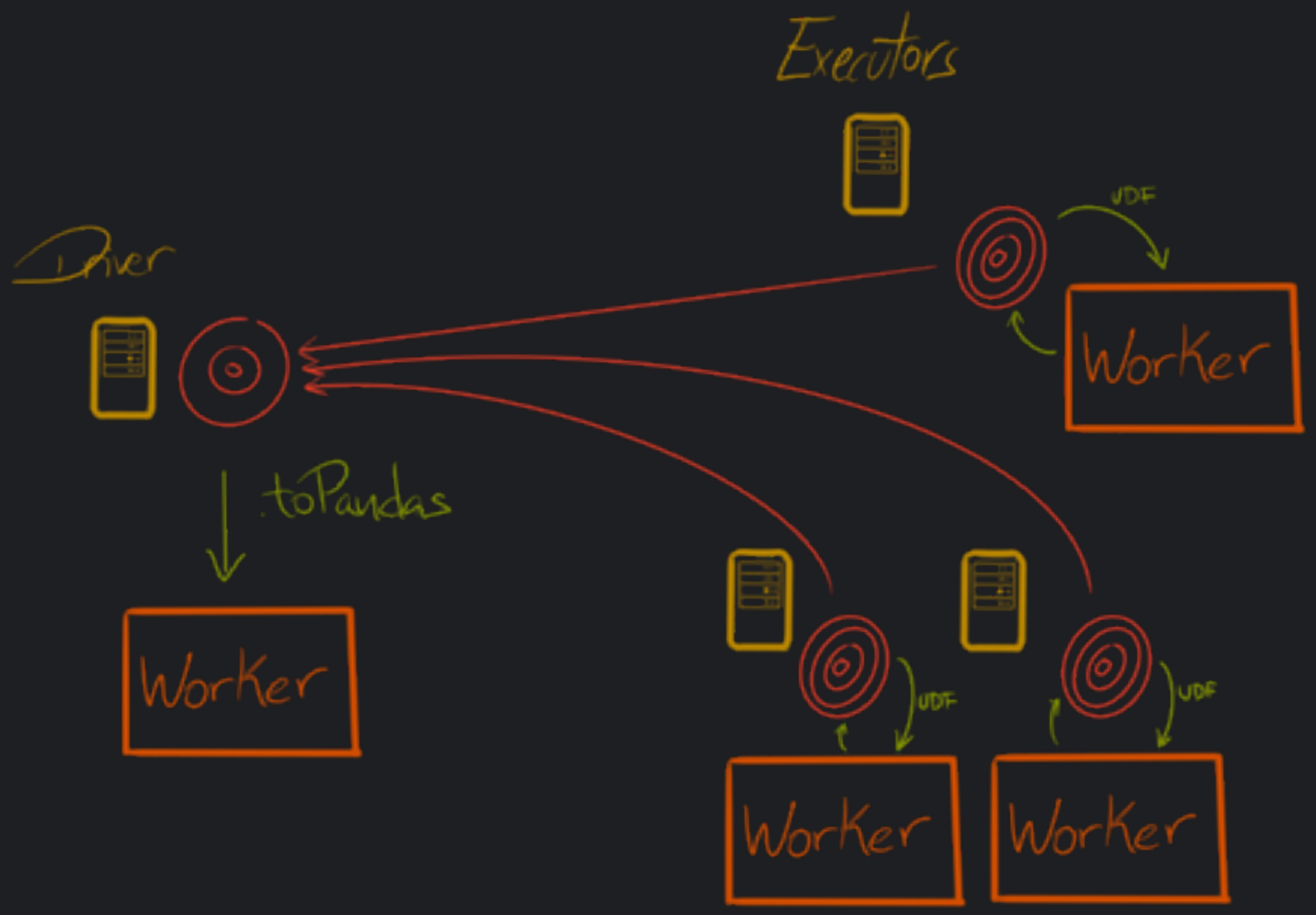
We have a dataset we want to perform some grouping operation



The dataset is shuffled as needed by the grouping columns



Note that each machine needs to have enough memory for the largest groups, but the work is being done in each machine via the UDF defined in Python, and data exchanged via Arrow



Finally only the required columns and data are passed to the driver, and we collect there `.toPandas`

BEFORE YOU MAY HAVE DONE SOMETHING LIKE..

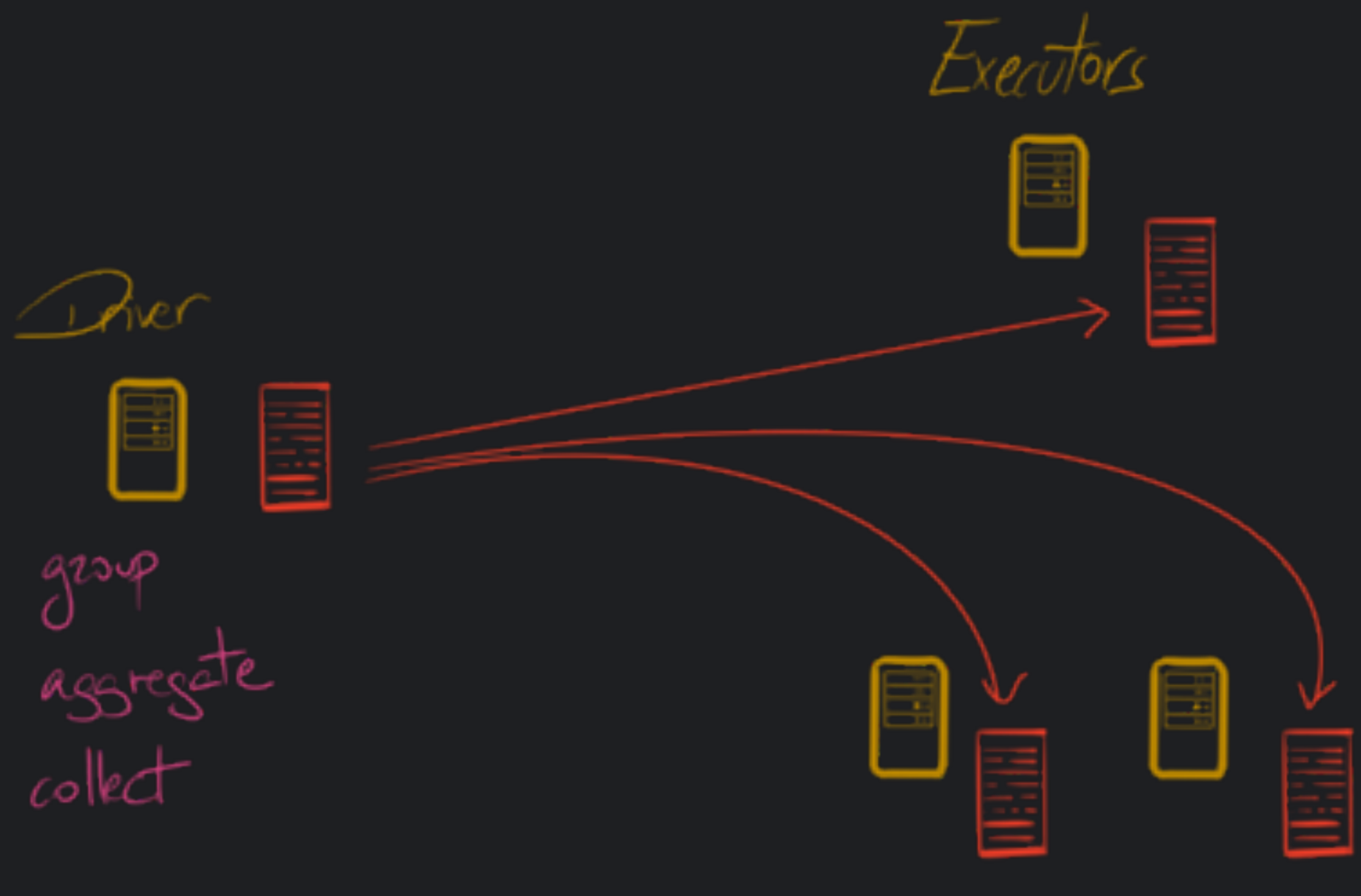
```
import numpy as np
from pyspark.sql.functions import collect_list

grouped = df2.groupby("id").agg(collect_list('spent').alias("spent_list"))
as_pandas = grouped.toPandas()
as_pandas["mean"] = as_pandas["spent_list"].apply(np.mean)
as_pandas["substracted"] = as_pandas["spent_list"].apply(np.array) - as_pandas["mean"]
df_to_pandas = as_pandas.drop(columns=["spent_list", "mean"]).explode("substracted")
```


Yikes

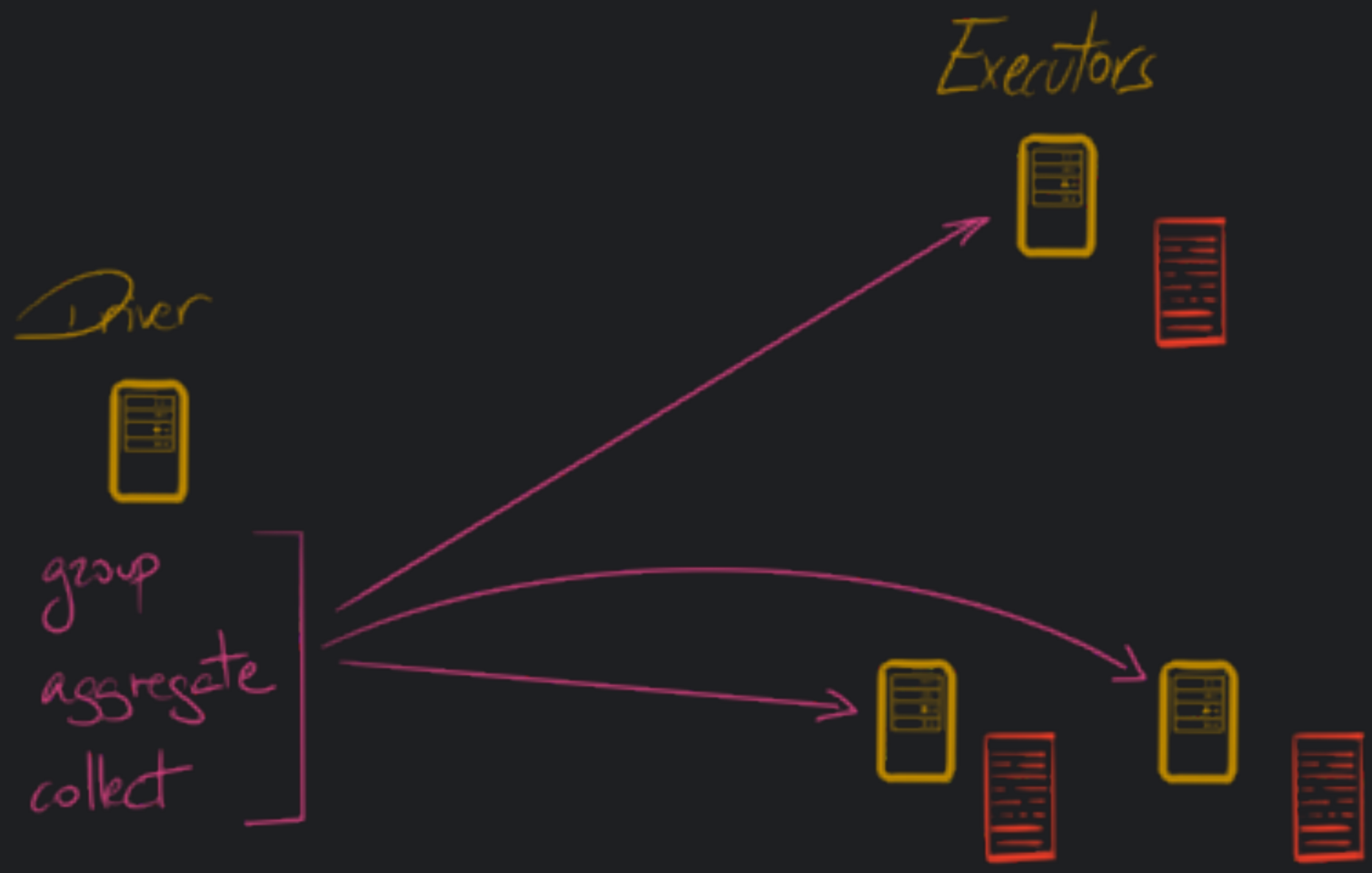
```
import numpy as np
from pyspark.sql.functions import collect_list

grouped = df2.groupby("id").agg(collect_list('spent').alias("spent_list"))
as_pandas = grouped.toPandas()
as_pandas["mean"] = as_pandas["spent_list"].apply(np.mean)
as_pandas["substracted"] = as_pandas["spent_list"].apply(np.array) - as_pandas["mean"]
df_to_pandas = as_pandas.drop(columns=["spent_list", "mean"]).explode("substracted")
```


This is painful now: we are sending *all* the grouped data to the driver. This fails for `1 << 25` with the driver dying and around `180s` for `20M` records.

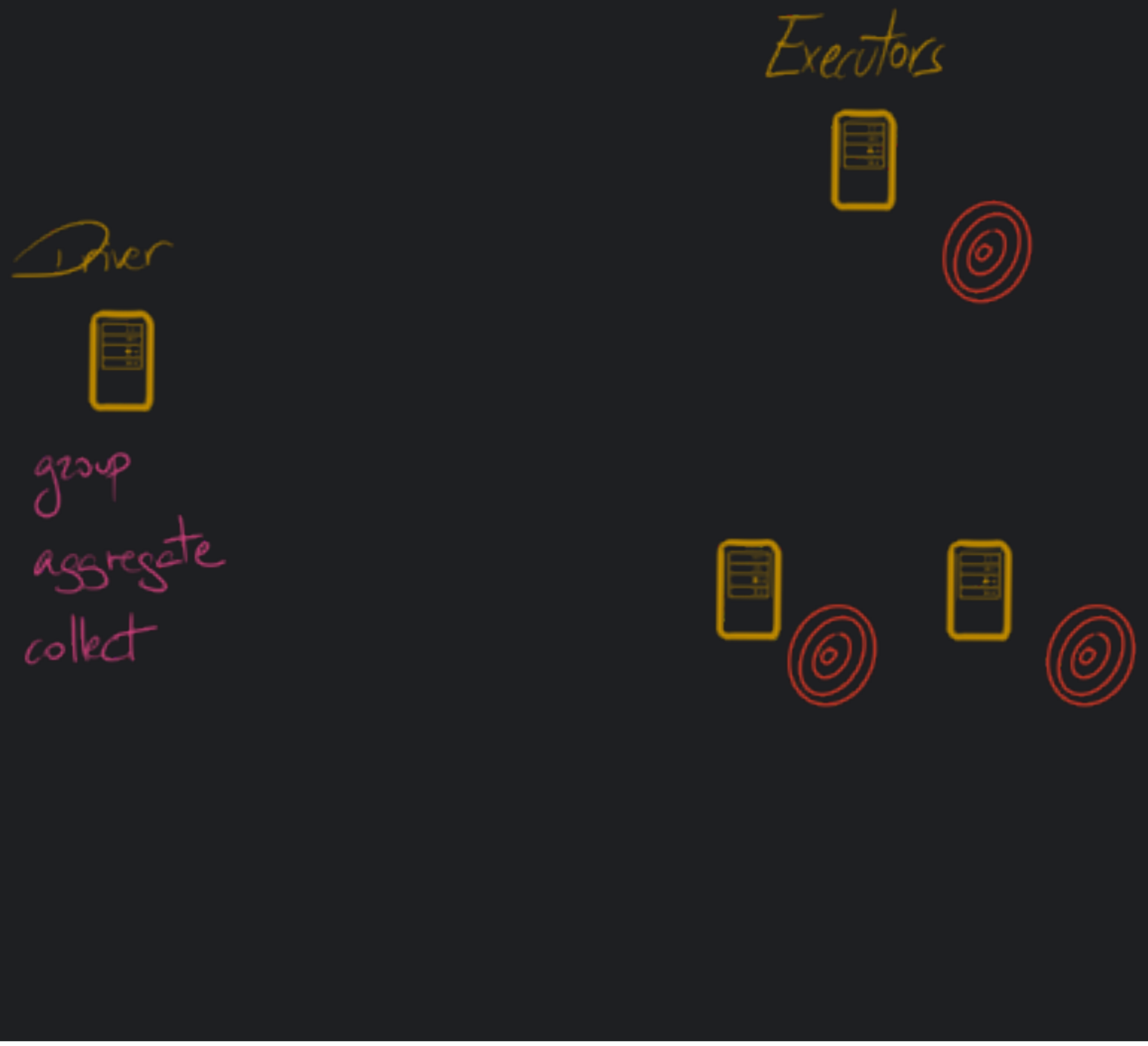
Note that there are other options that can do some or part of the work in the executors but that would get tricky *very* quick if you want to stay in Python land.



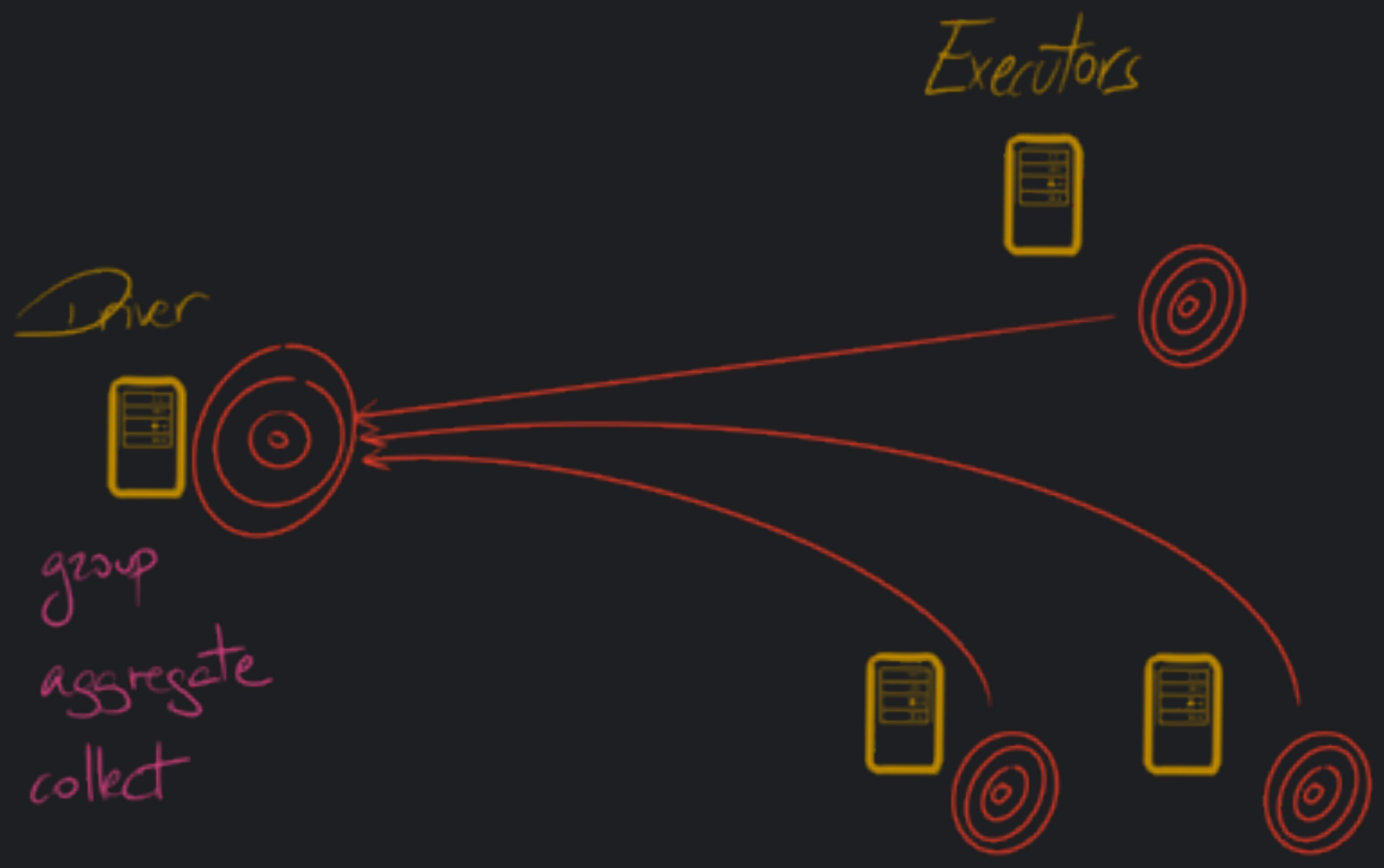
As before, some data gets distributed/shuffled



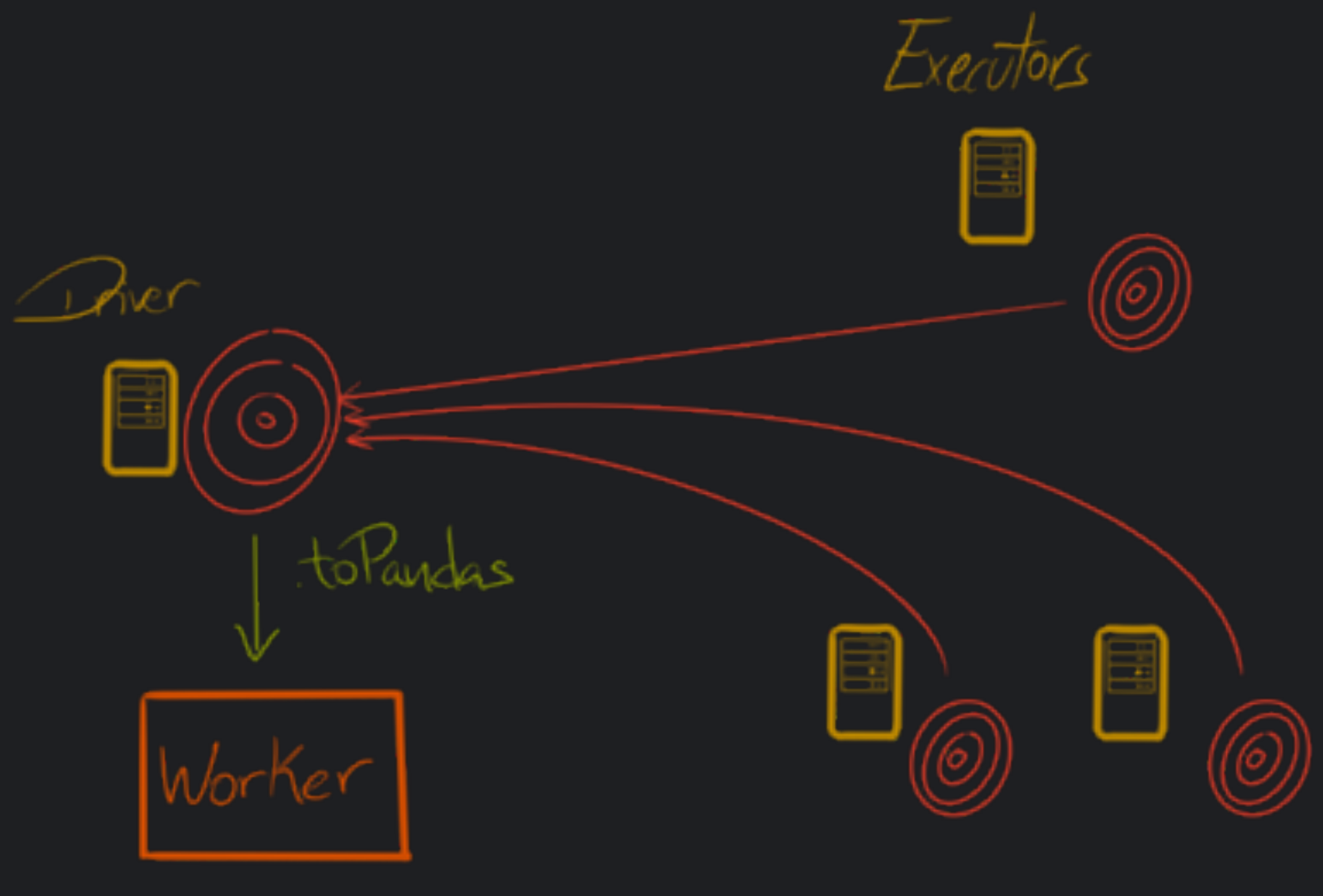
The operation goes to the executors



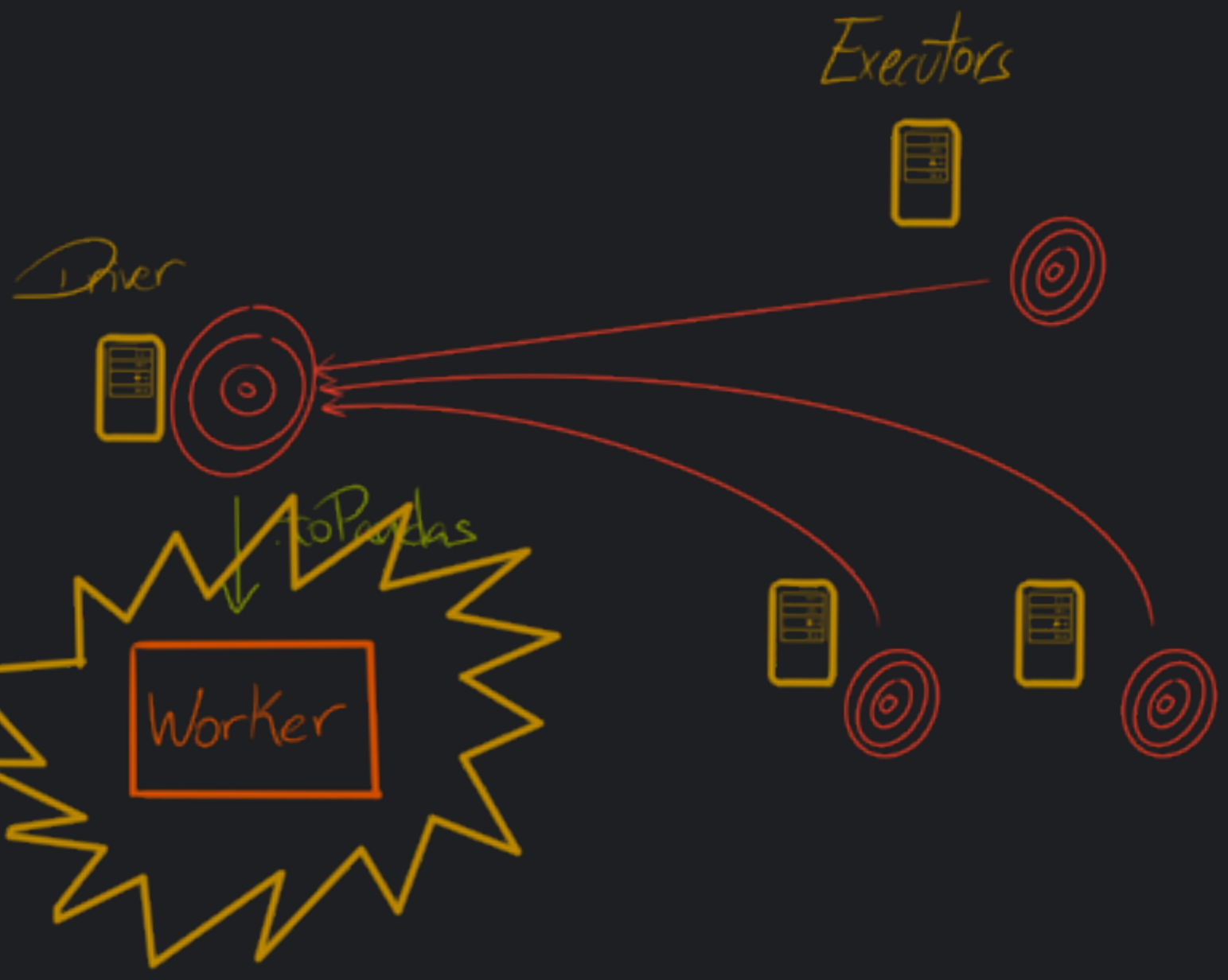
They group and aggregate with the not-fancy `collect_list` (when this is too large, fun times)



The `collect_list`-collected data is now sent to the driver, to finalise the operation in Python



All this data is slowly serialized into a Pandas dataframe



If this fits in memory in the driver, now the final work happens. That's a *big if*.

TLDR:

USE[1] ARROW AND PANDAS UDFS

[1] IN PYSPARK

RESOURCES

- SPARK DOCUMENTATION
- HIGH PERFORMANCE SPARK BY HOLDEN KARAU
- THE INTERNALS OF APACHE SPARK 2.4.2 BY JACEK LASKOWSKI
- SPARK'S GITHUB
- BECOME A CONTRIBUTOR

QUESTIONS?


THANKS!
SPARK+AI SUMMIT 2019
#UnifiedDataAnalytics #SparkAISummit

GET THE SLIDES FROM MY GITHUB:

github.com/rberenguel/

THE REPOSITORY IS

pyspark-arrow-pandas


SPARK+AI
SUMMIT 2019
Unify.
Scale.
Build.
WIFI SSID:Spark+AISummit | Password: UnifiedDataAnalytics

FURTHER REFERENCES

ARROW

ARROW'S HOME
ARROW'S GITHUB
ARROW SPEED BENCHMARKS
ARROW TO PANDAS CONVERSION BENCHMARKS
POST: STREAMING COLUMNAR DATA WITH APACHE ARROW
POST: WHY PANDAS USERS SHOULD BE EXCITED BY APACHE ARROW
CODE: ARROW-PANDAS COMPATIBILITY LAYER CODE
CODE: ARROW TABLE CODE
PYARROW IN-MEMORY DATA MODEL
BALLISTA: A POC DISTRIBUTED COMPUTE PLATFORM (RUST)
PYJAVA: POC ON JAVA/SCALA AND PYTHON DATA INTERCHANGE WITH ARROW

PANDAS

PANDAS' HOME
PANDAS' GITHUB
GUIDE: IDIOMATIC PANDAS
CODE: PANDAS INTERNALS
DESIGN: PANDAS INTERNALS
TALK: DEMYSTIFYING PANDAS' INTERNALS, BY MARC GARCIA
MEMORY LAYOUT OF MULTIDIMENSIONAL ARRAYS IN NUMPY

SPARK/PYSPARK

CODE: PYSPARK SERIALIZERS
JIRA: FIRST STEPS TO USING ARROW (ONLY IN THE PYSPARK DRIVER)
POST: SPEEDING UP PYSPARK WITH APACHE ARROW
ORIGINAL JIRA ISSUE: VECTORIZED UDFS IN SPARK
INITIAL DOC DRAFT
POST BY BRYAN CUTLER (LEADER FOR THE VEC UDFS PR)
POST: INTRODUCING PANDAS UDF FOR PYSPARK
CODE: ORG.APACHE.SPARK.SQL.VECTORIZED
POST BY BRYAN CUTLER: SPARK TOPANDAS() WITH ARROW, A DETAILED LOOK

PY4J

PY4J'S HOME
PY4J'S GITHUB
CODE: REFLECTION ENGINE

TABLE FOR `toPandas`

2^X	DIRECT (S)	WITH ARROW (S)	FACTOR
17	1.08	0.18	5.97
18	1.69	0.26	6.45
19	4.16	0.30	13.87
20	5.76	0.61	9.44
21	9.73	0.96	10.14
22	17.90	1.64	10.91
23	(OOM)	3.42	
24	(OOM)	11.40	

EOF