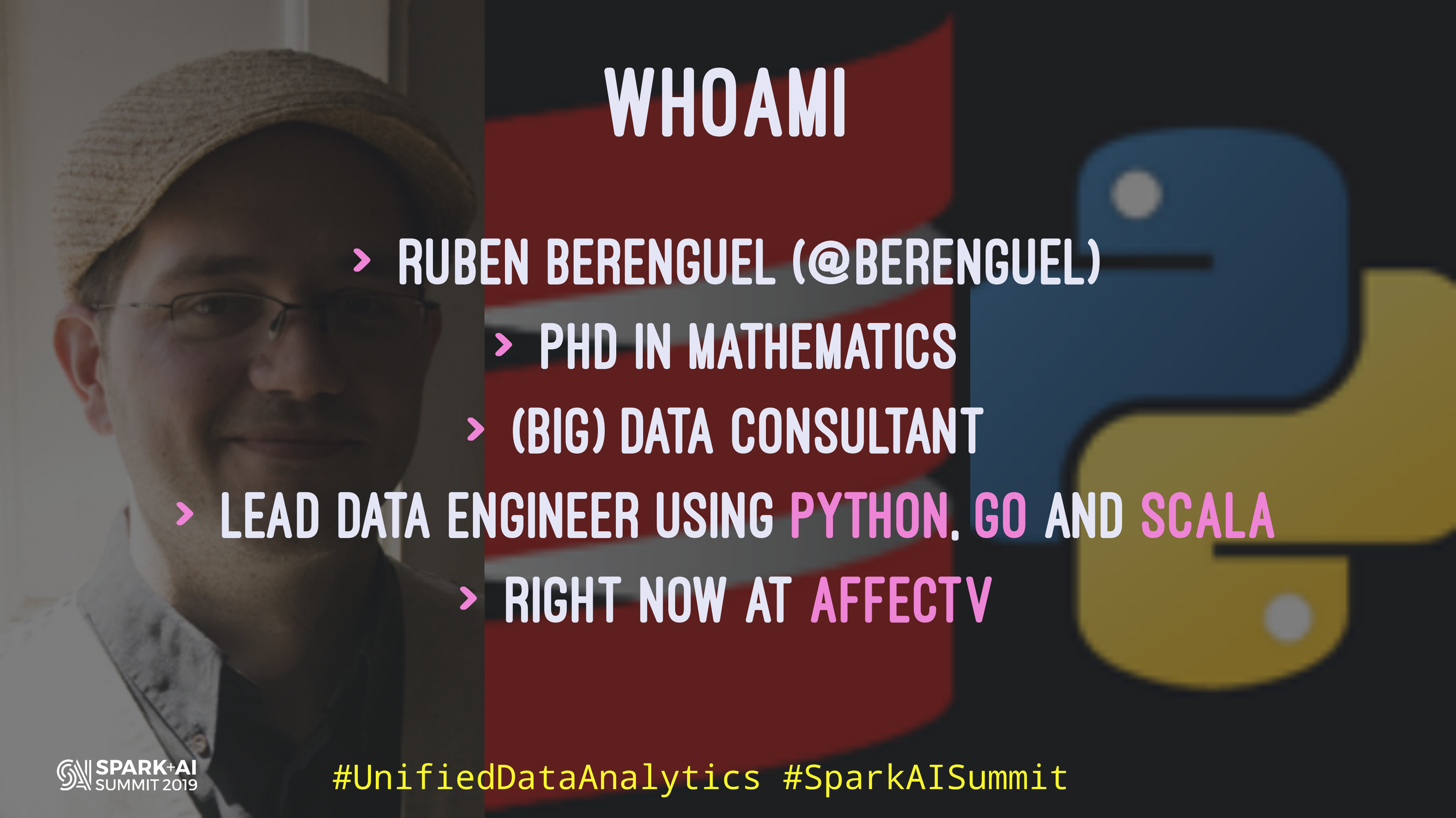

SPARK+AI
SUMMIT 2019
Build.
Unify.
Scale.
WIFI SSID:Spark+AISummit | Password: UnifiedDataAnalytics

Internals of Speeding up PySpark with Arrow

Ruben Berenguel, Consultant

#UnifiedDataAnalytics #SparkAISummit

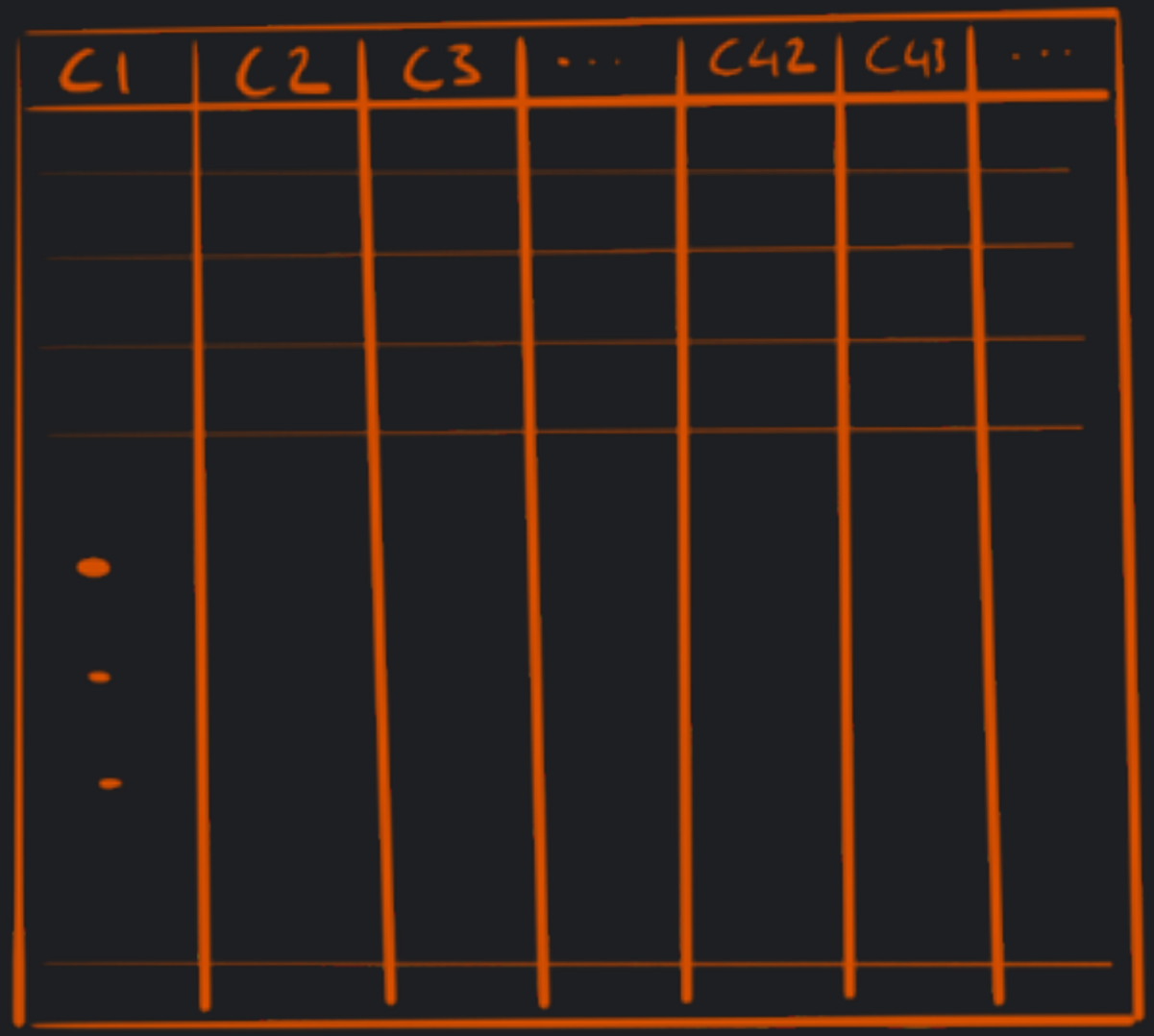


WHOAMI

- 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)


C1
C2
C3
...
C42
C43
...


SUM
C1
C2
C3
...
C42
C43
...


ROW

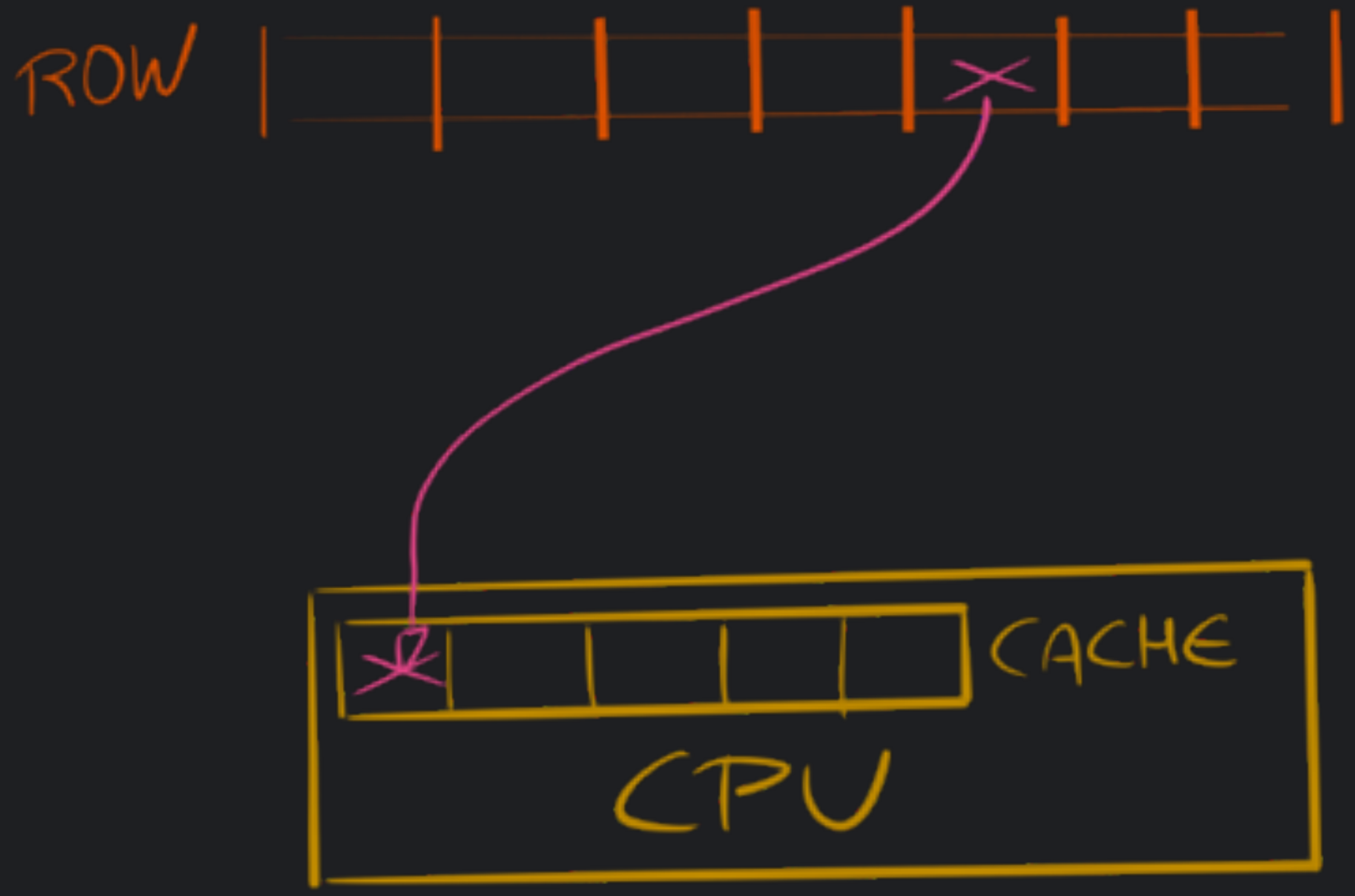

CPU


ROW


CACHE
CPU

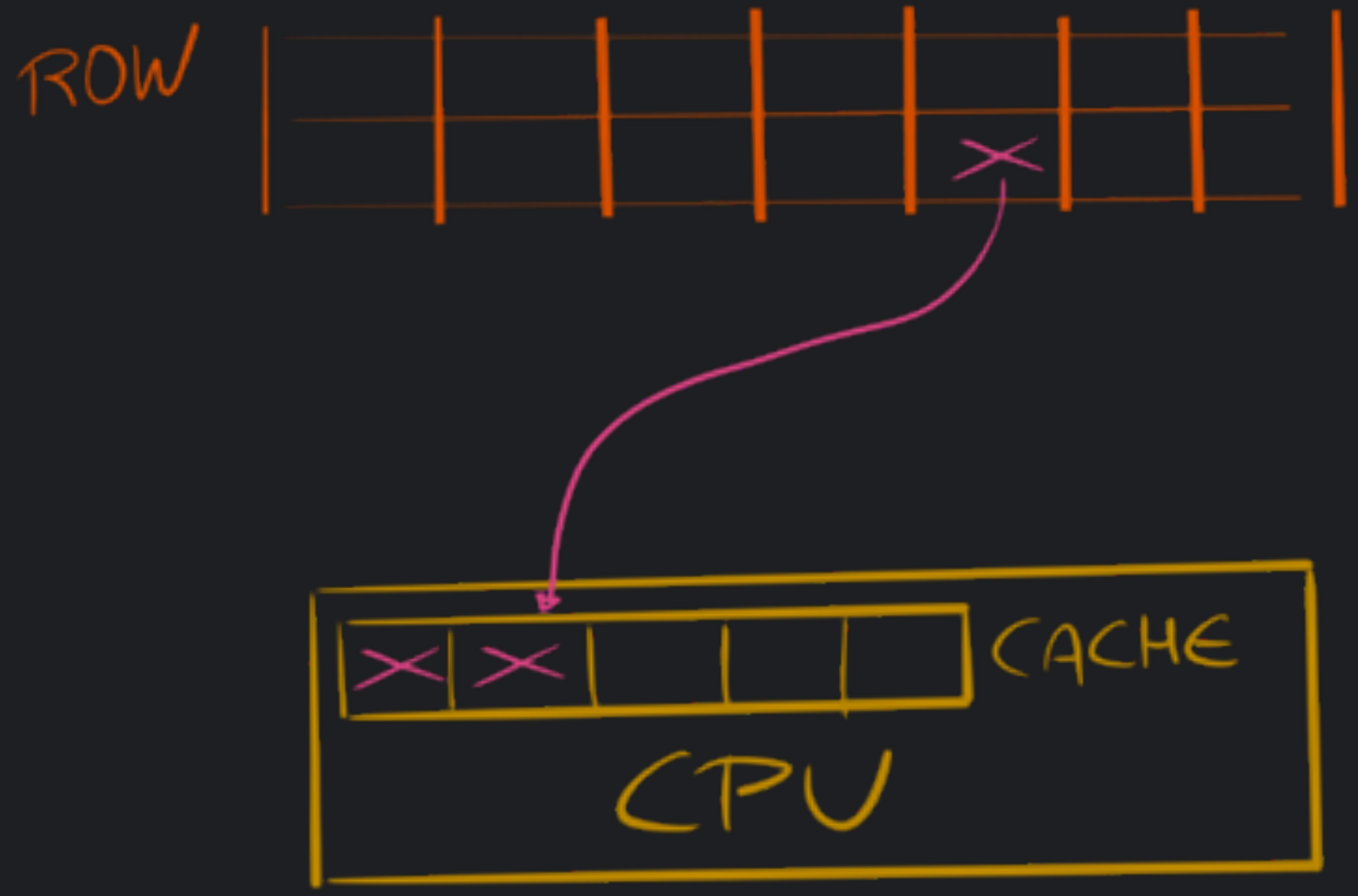

ROW

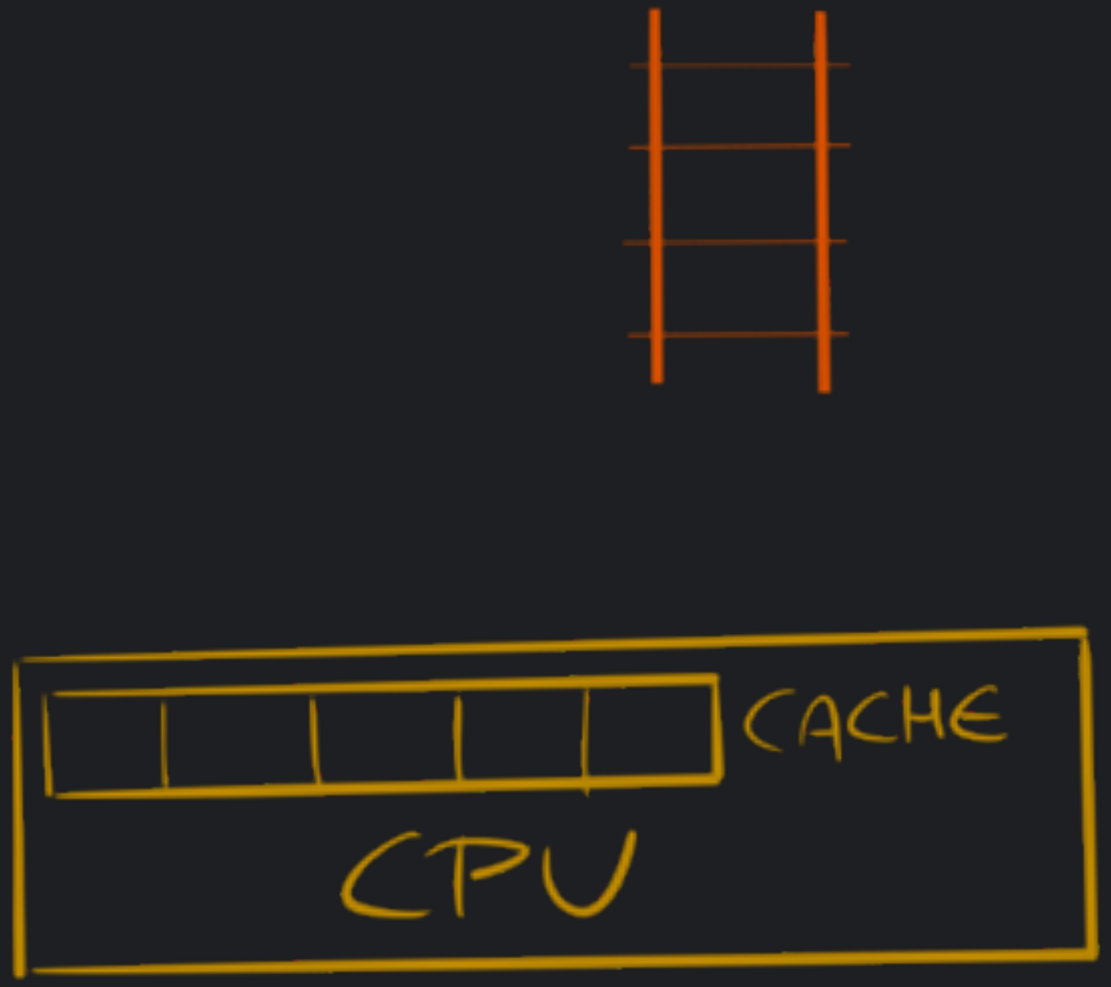

CACHE
CPU

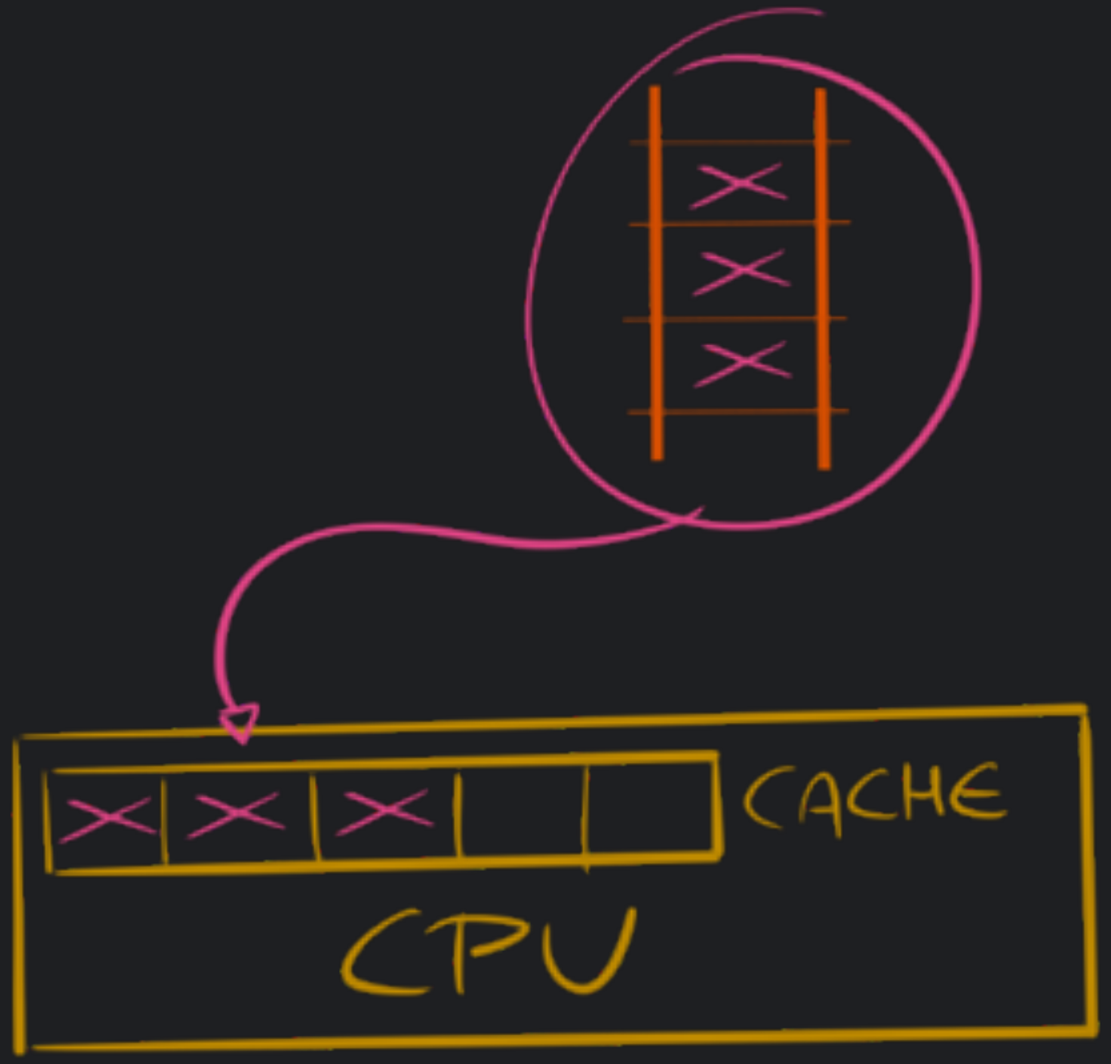

ROW
CACHE
CPU


ROW


CACHE
CPU


ROW
CACHE
CPU


CACHE
CPU


CACHE
CPU

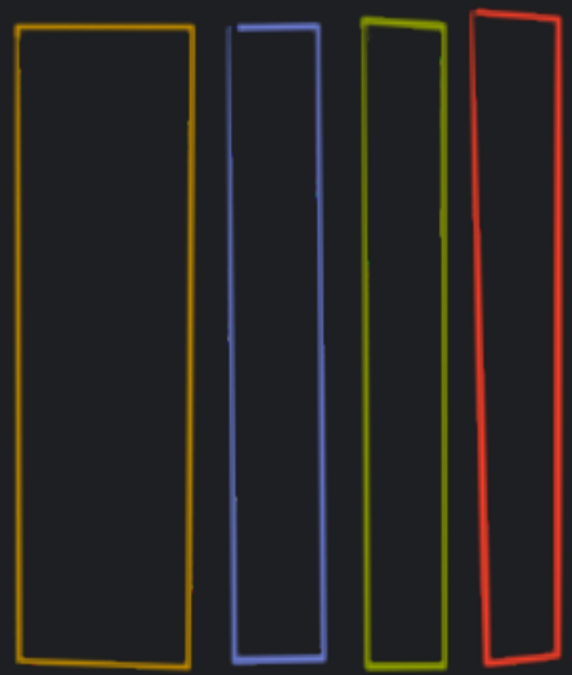
HOW DOES PANDAS MANAGE COLUMNAR DATA?

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	
⋮					

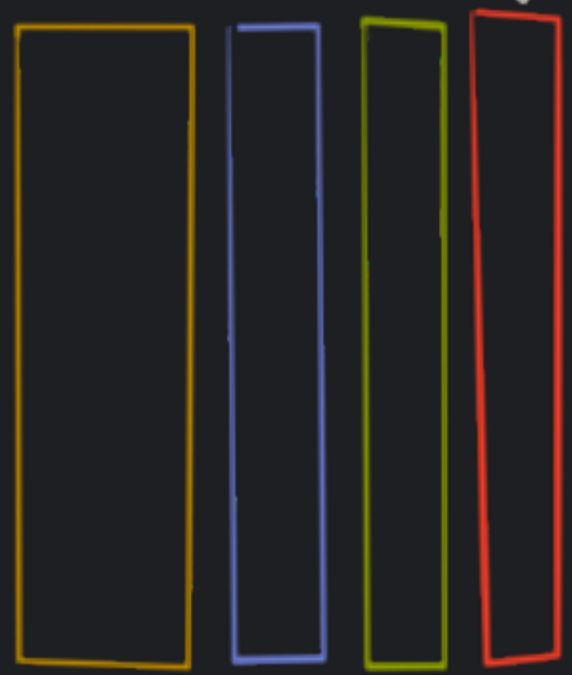
Pandas DataFrame

	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
⋮				



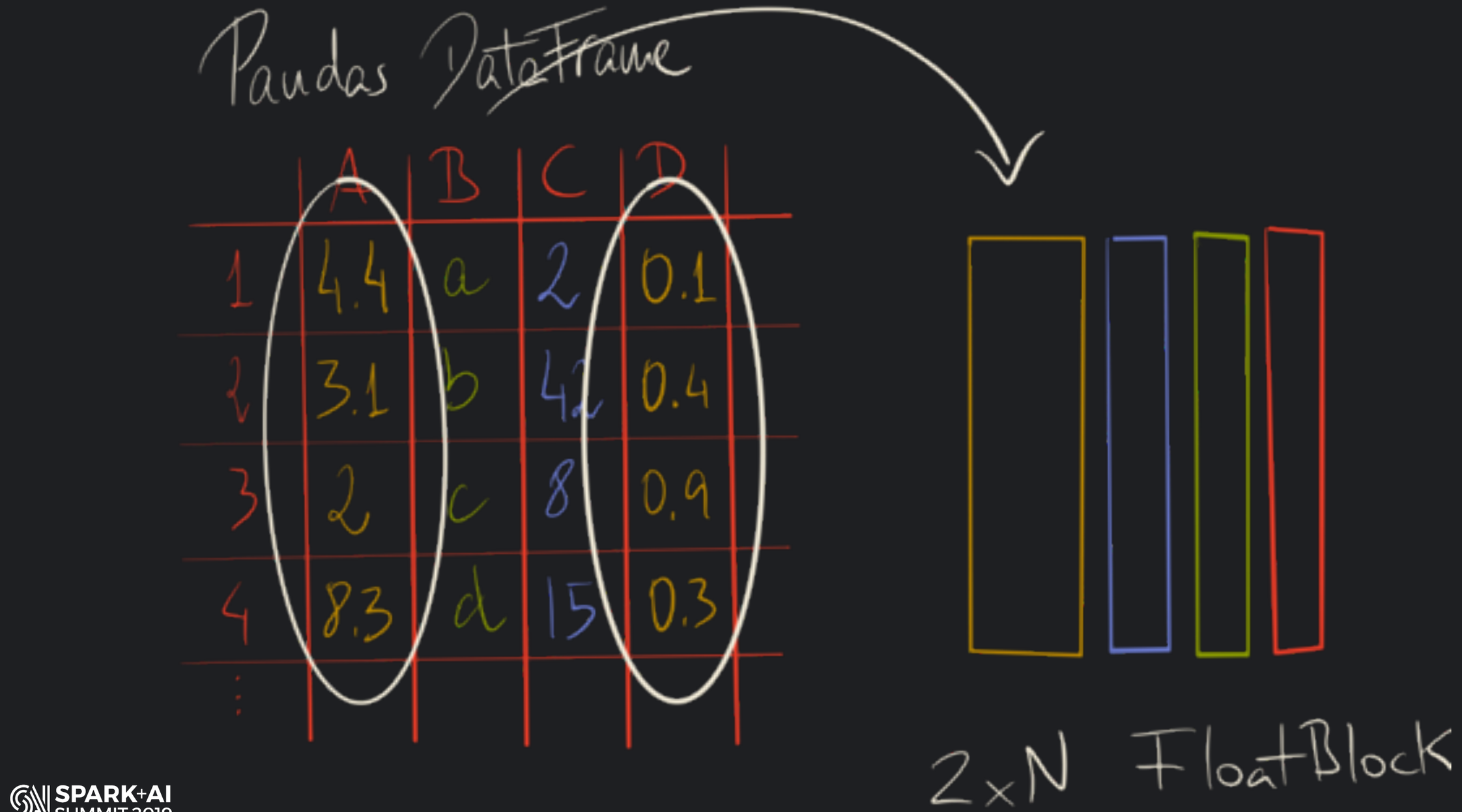
Pandas DataFrame

	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
⋮				



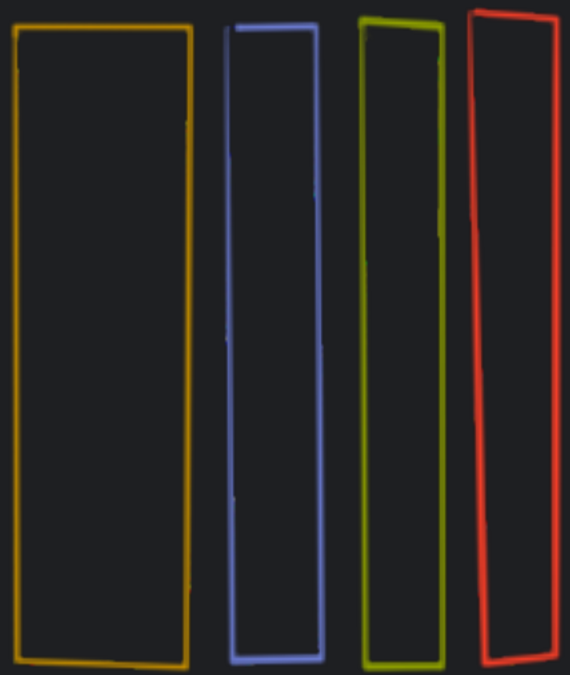
INDEX

SECTION

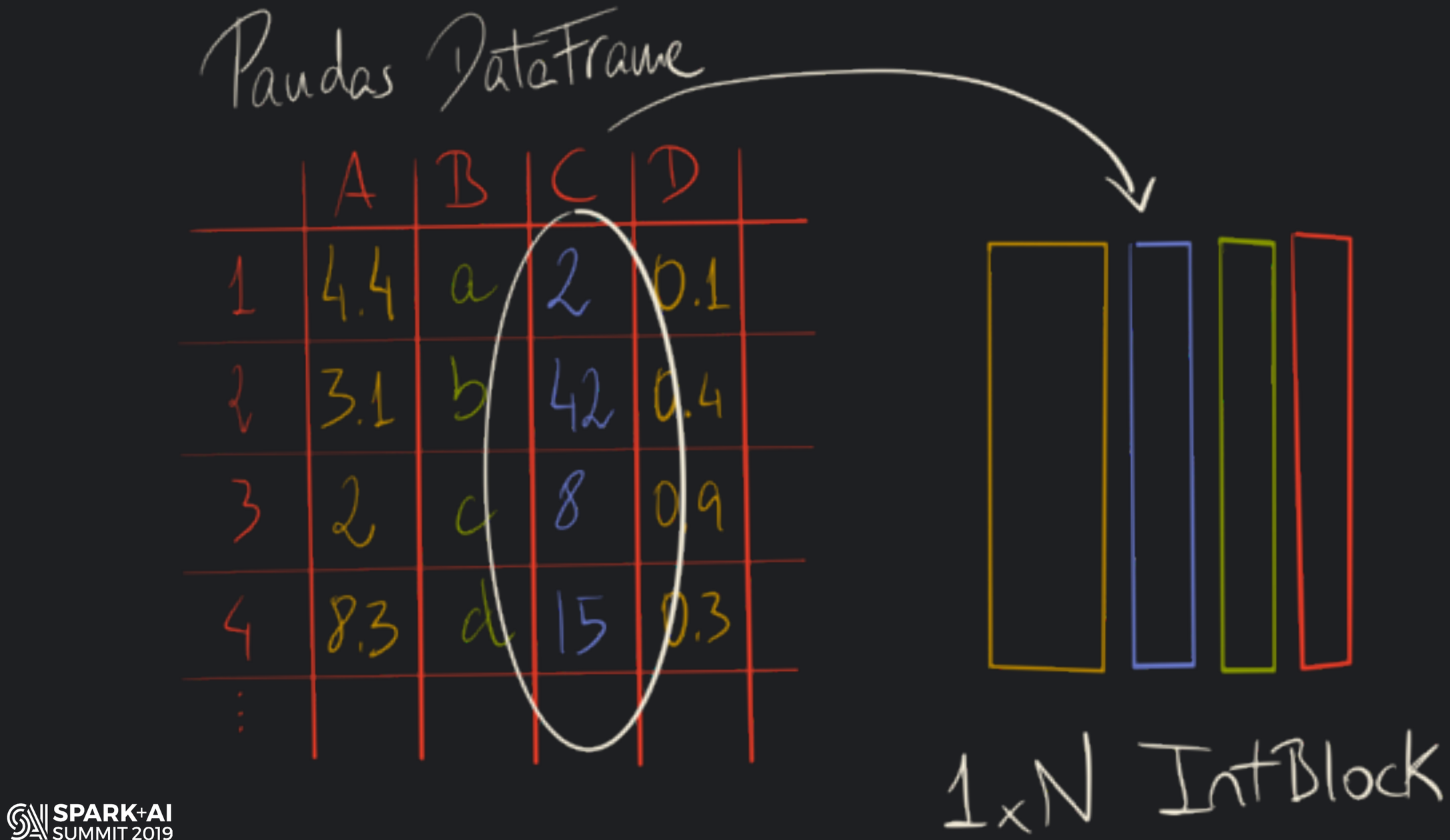

Pandas DataFrame
A
B
C
D
1
4.4
a
2
0.1
2
3.1
b
4.2
0.4
3
2
c
8
0.9
4
8.3
d
15
0.3
2 x N FloatBlock

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
⋮				



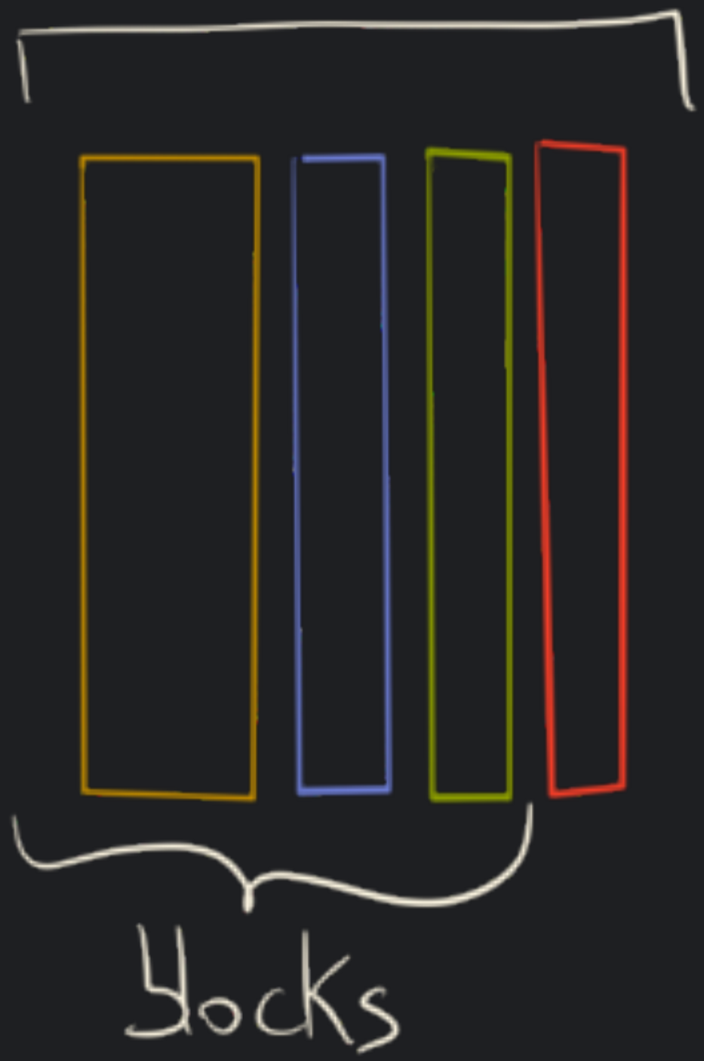
1 x N ObjectBlock


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
1 x N 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?

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

HOW DOES ARROW MANAGE COLUMNAR DATA?

Table

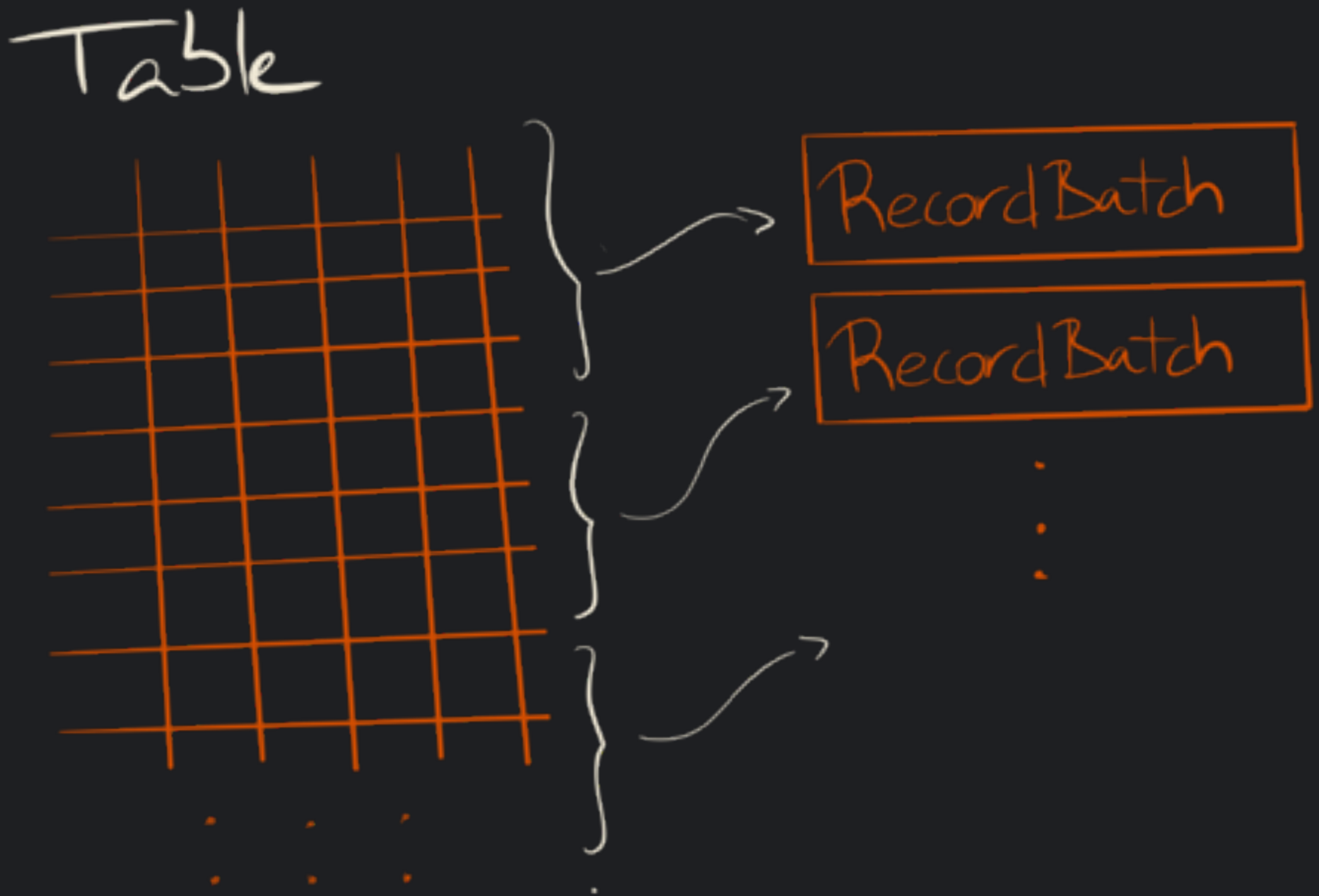
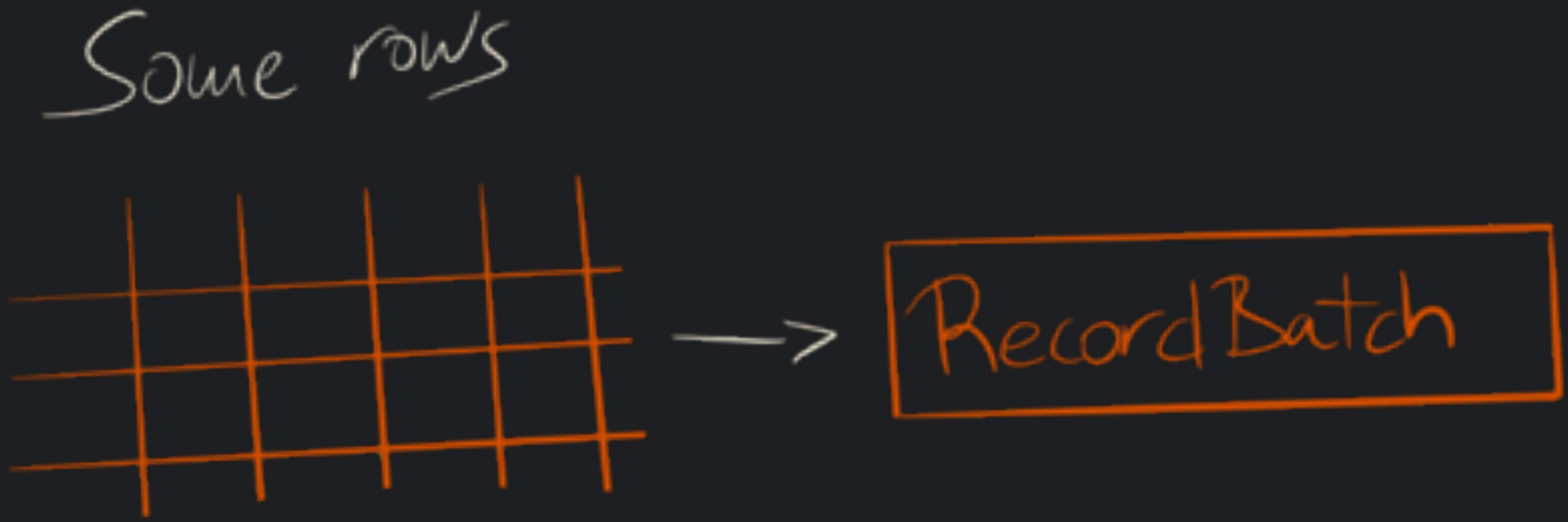
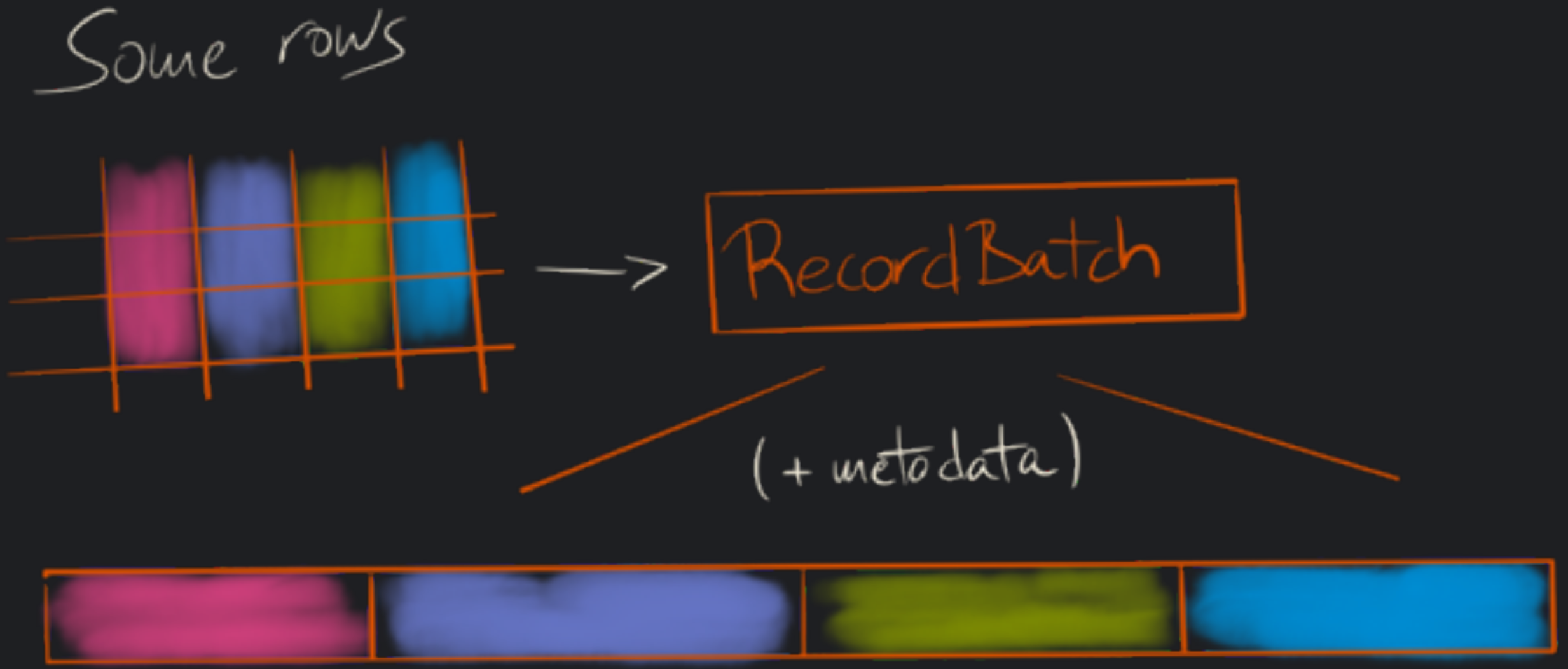


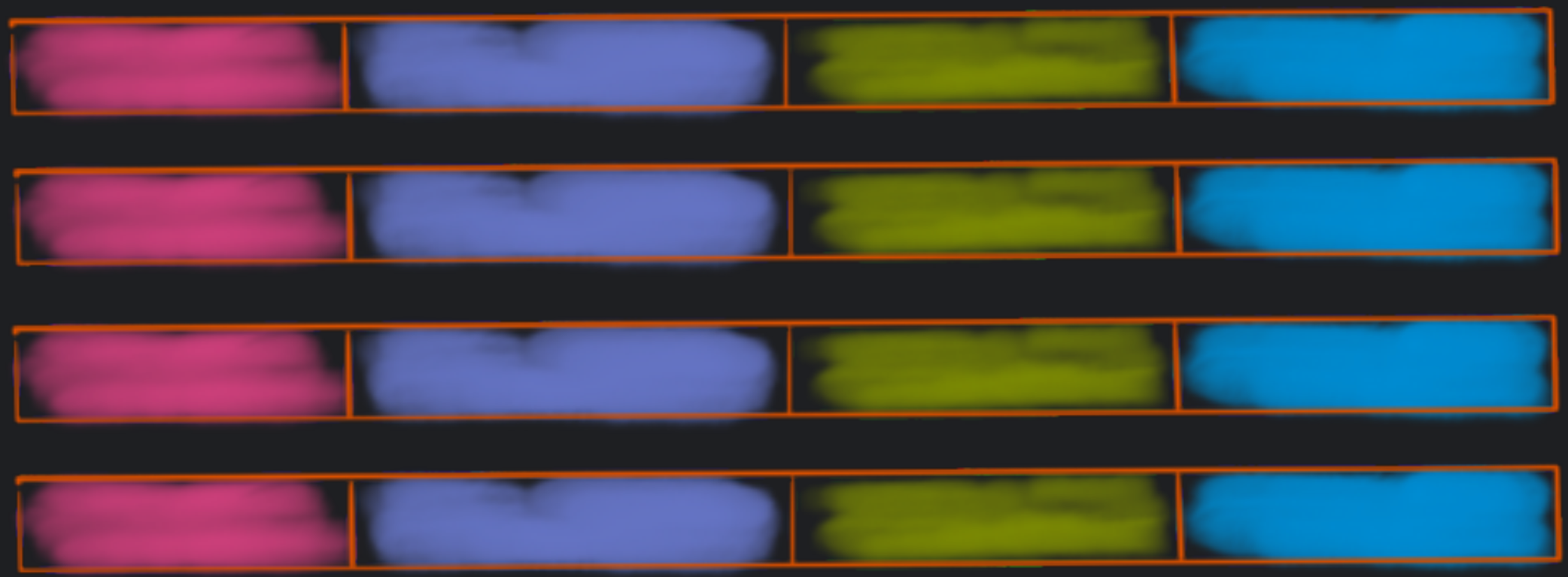
Table
RecordBatch
RecordBatch

Some rows


Some rows
RecordBatch


Some rows
RecordBatch
(+ metodata)

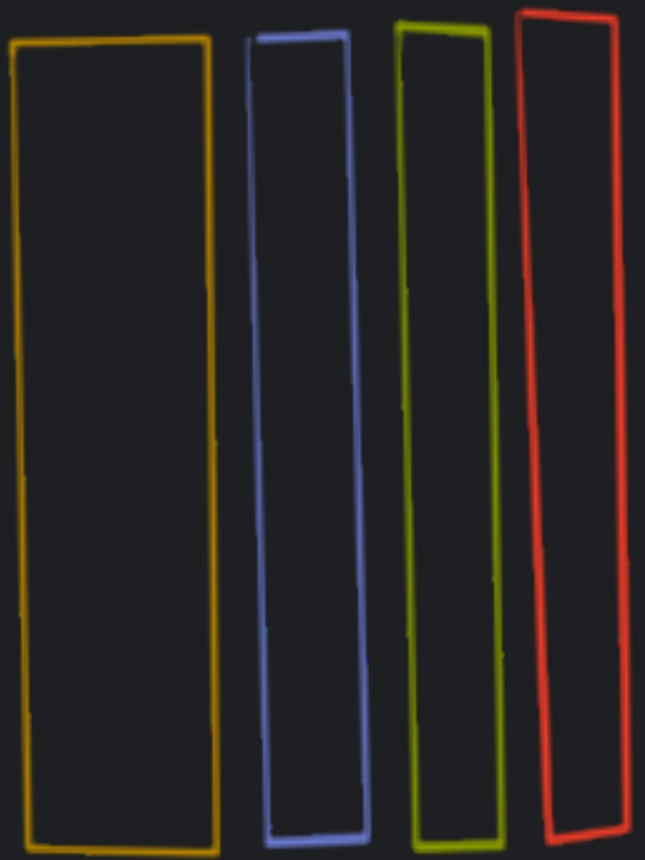
Arrow Table



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

Arrow Table

Pandas BlockManager



WHAT IS SPARK?

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

HOW DOES SPARK WORK?

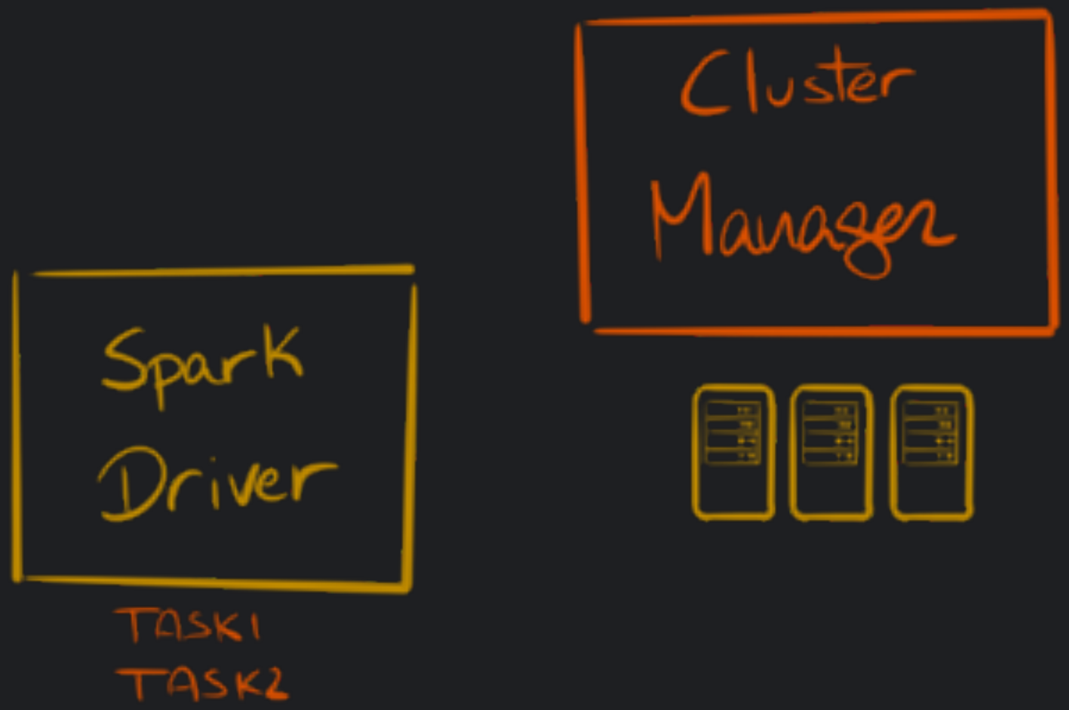
SPARK USUALLY RUNS ON TOP OF A CLUSTER MANAGER

AND A DISTRIBUTED STORAGE

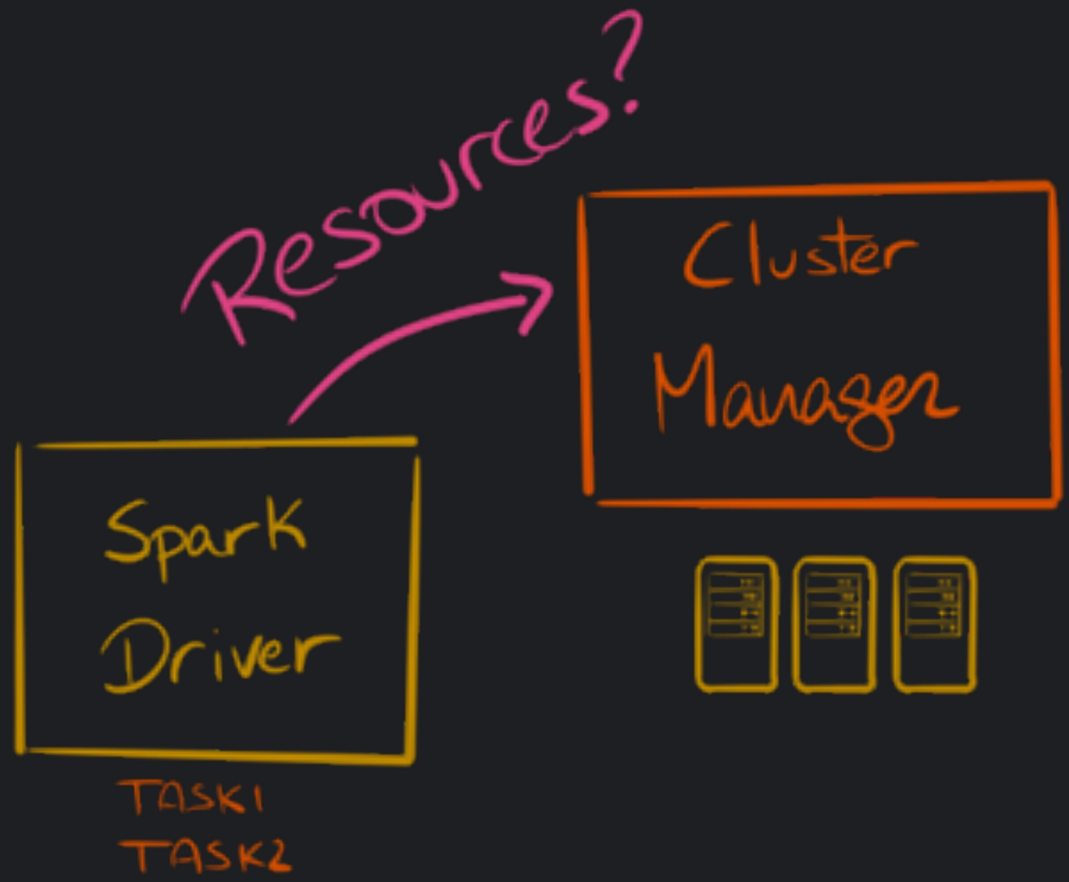
A SPARK PROGRAM RUNS IN THE DRIVER

THE DRIVER REQUESTS RESOURCES FROM THE CLUSTER MANAGER TO RUN TASKS

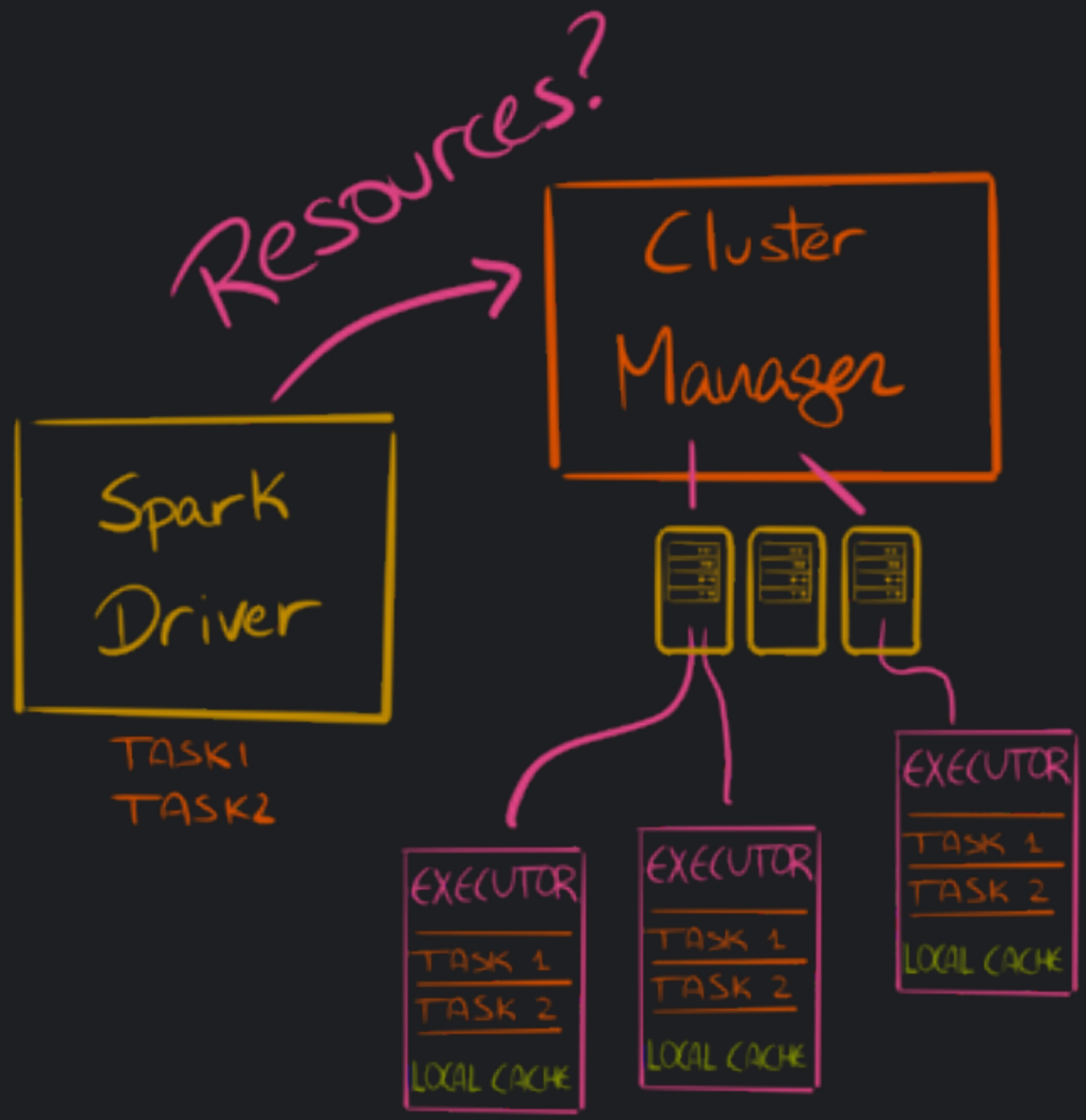
THE DRIVER REQUESTS RESOURCES FROM THE CLUSTER MANAGER TO RUN TASKS



THE DRIVER REQUESTS RESOURCES FROM THE CLUSTER MANAGER TO RUN TASKS



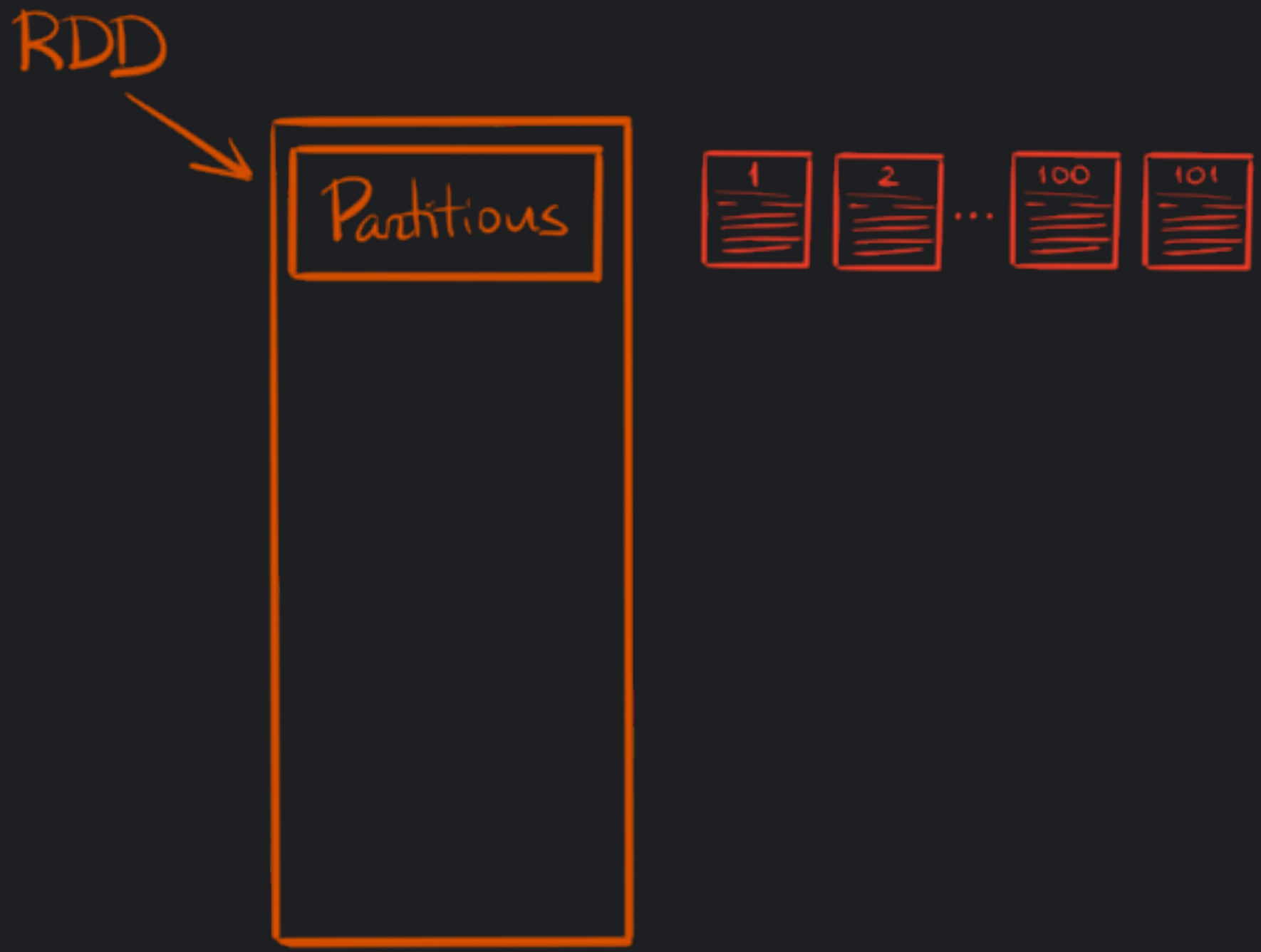
THE DRIVER REQUESTS RESOURCES FROM THE CLUSTER MANAGER TO RUN TASKS

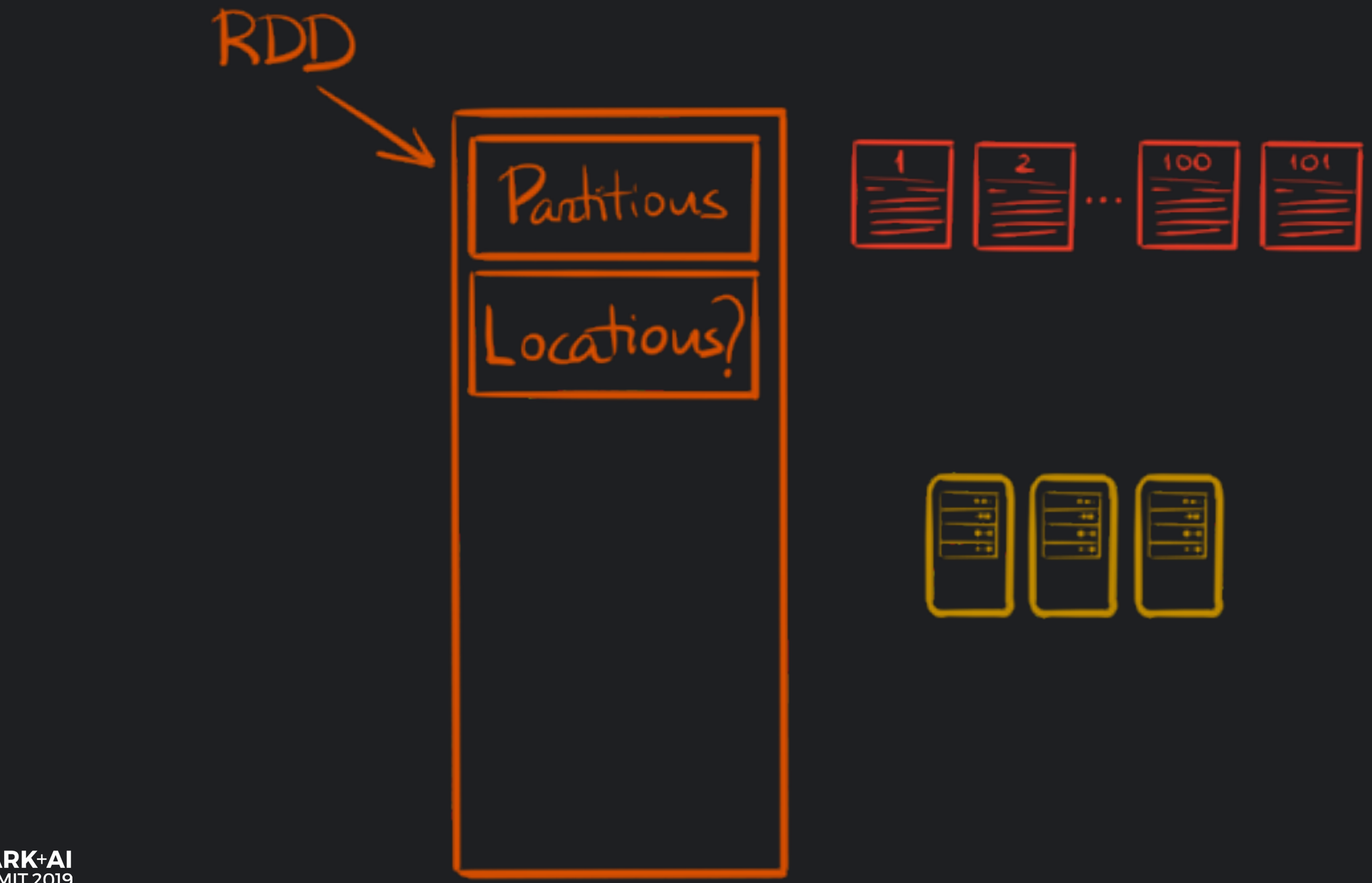


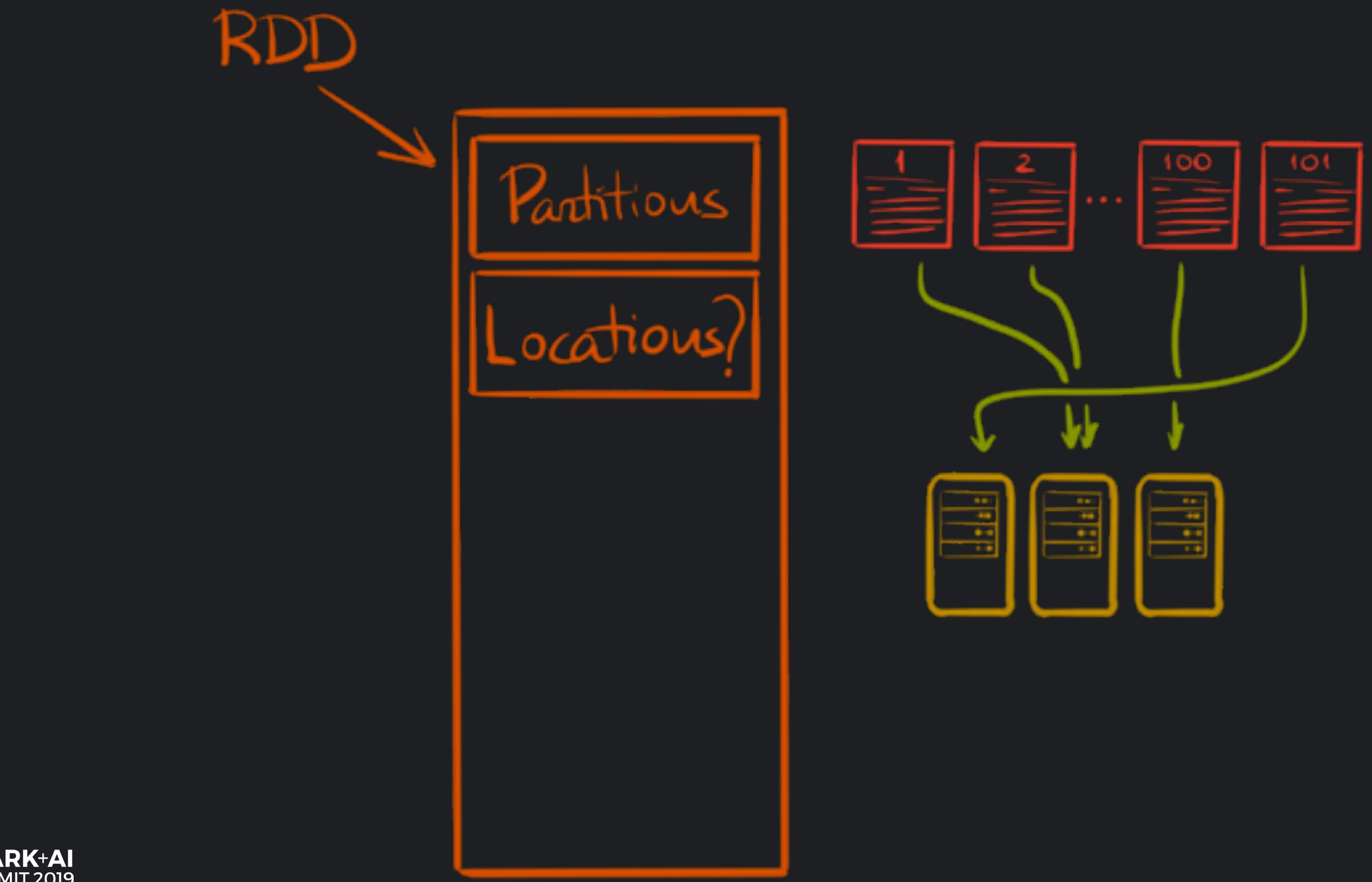
THE MAIN BUILDING BLOCK IS THE RDD: RESILIENT DISTRIBUTED DATASET

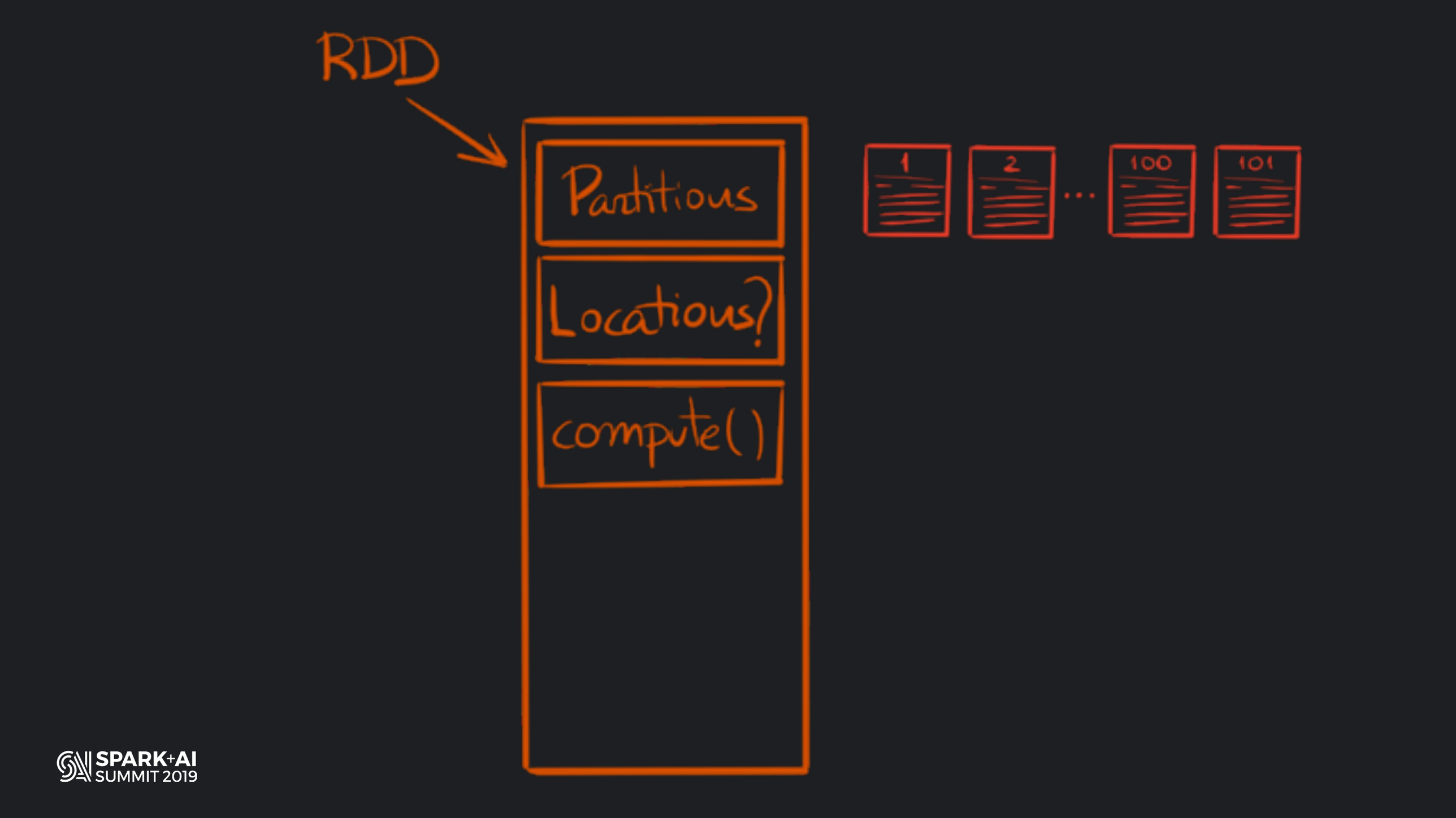

RDD

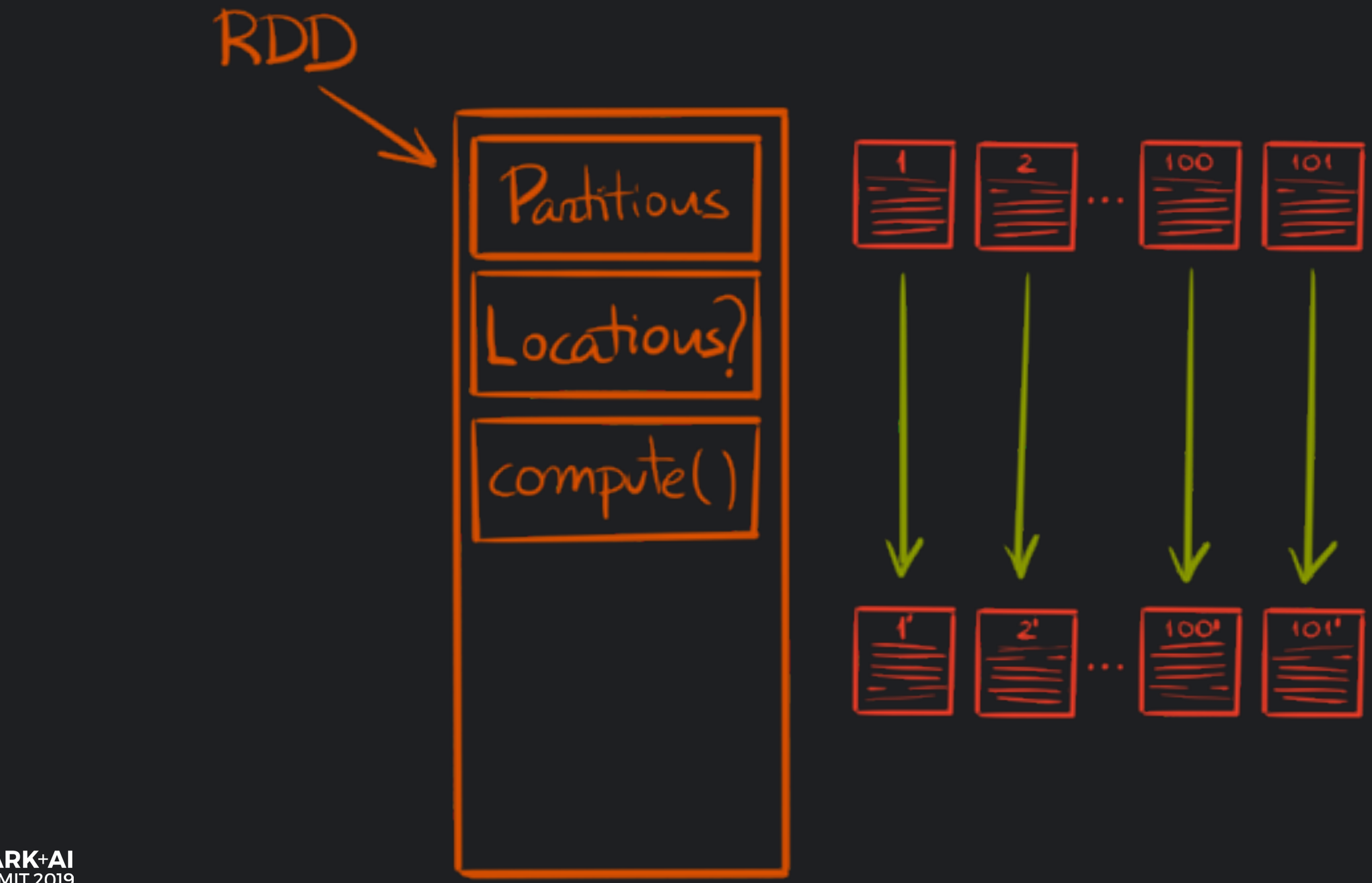

RDD
Partitions

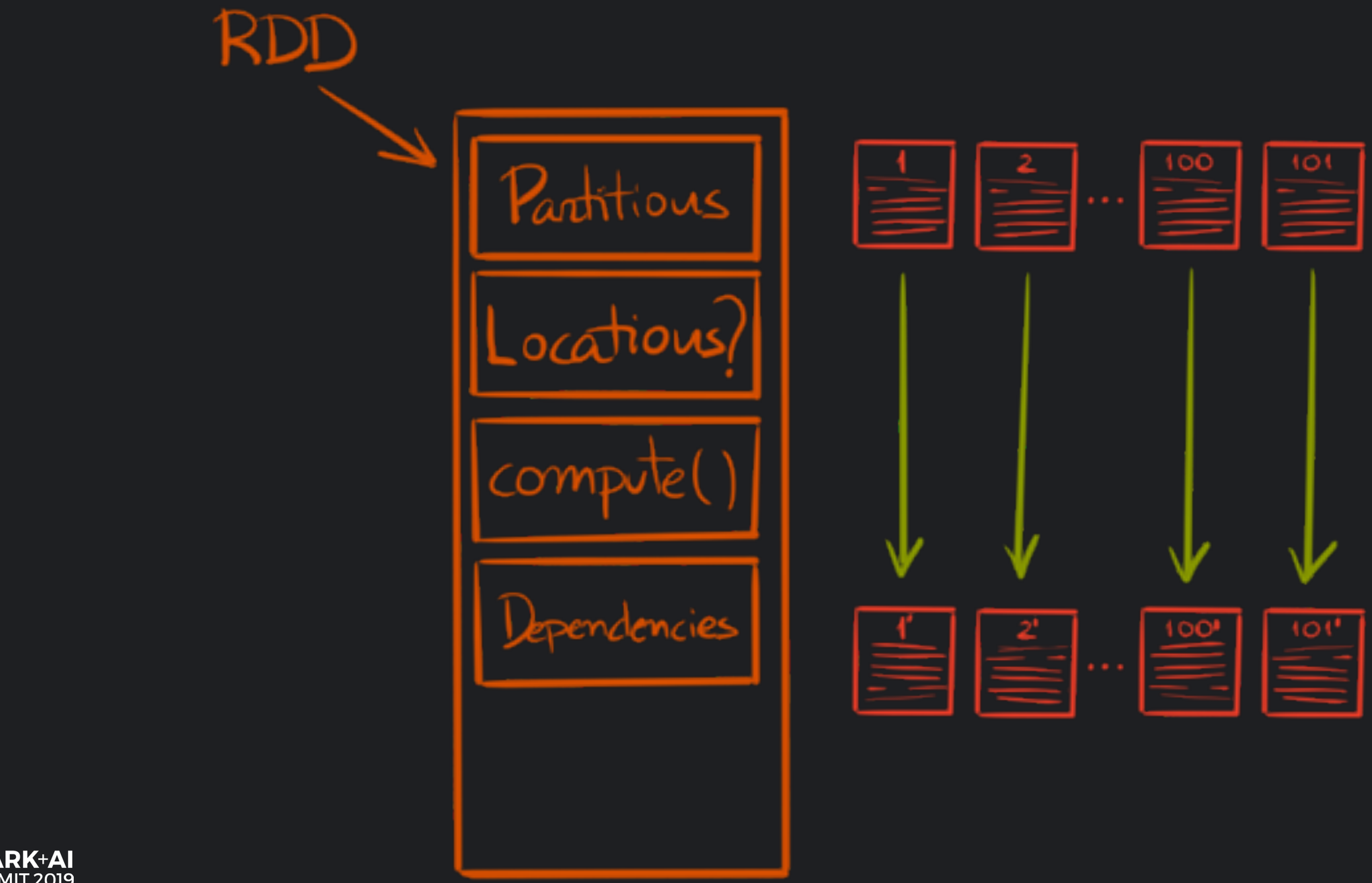

RDD
Partitions
1
2
100
101

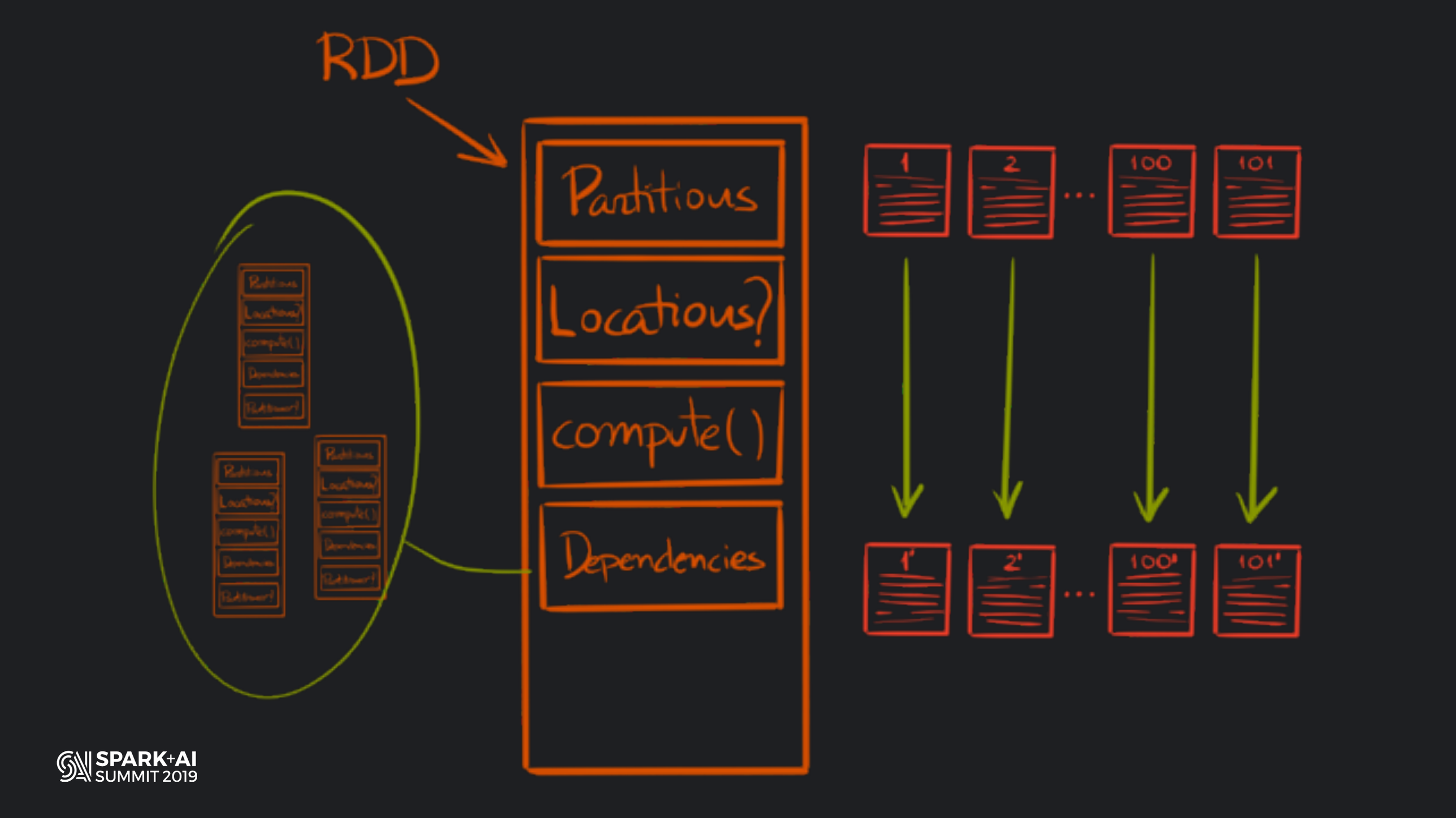

RDD
Partitions
Locations?
1
2
100
101

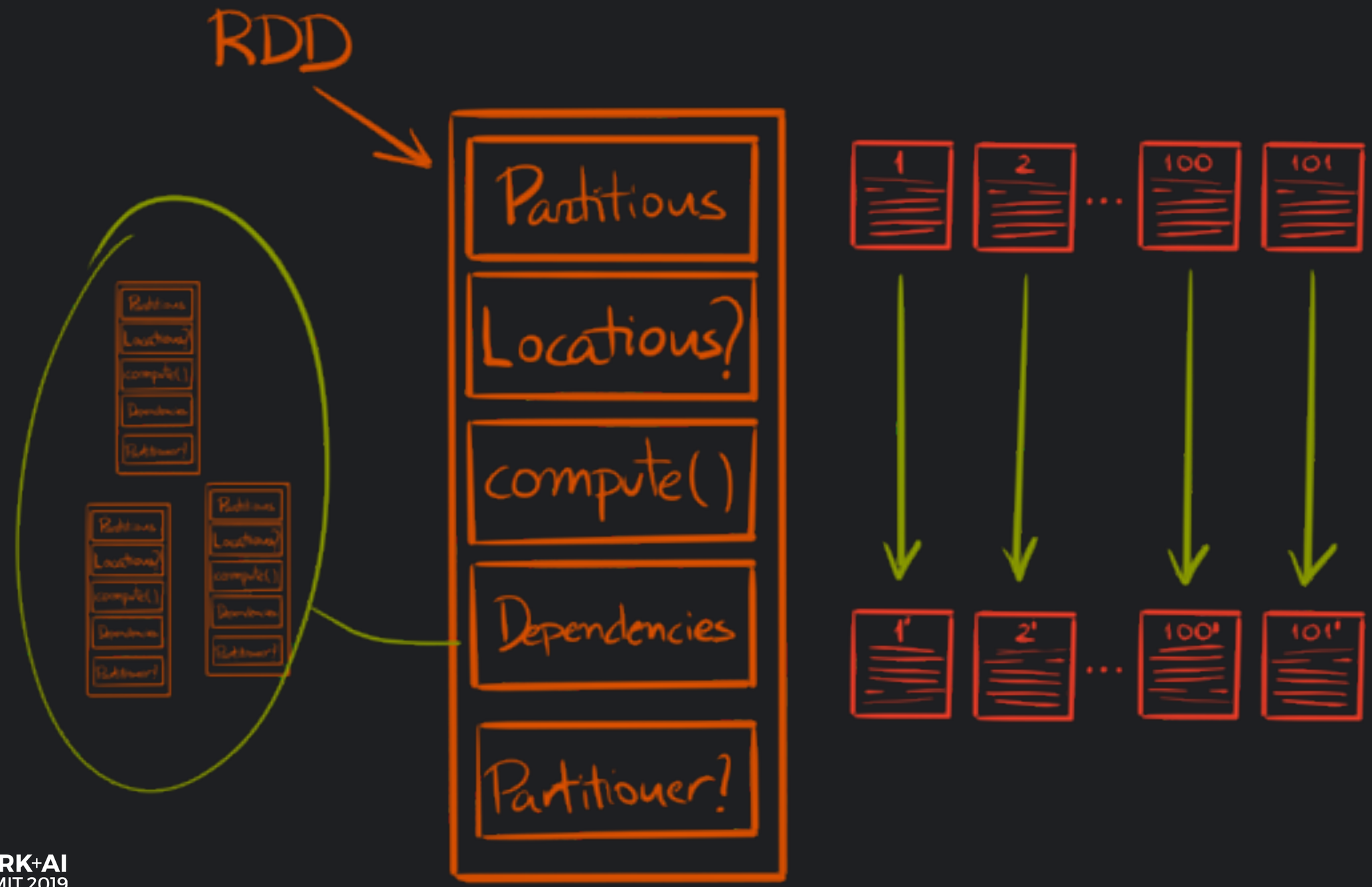

RDD
Partitions
Locations?
1
2
100
101


RDD
Partitions
Locations?
compute()
1
2
100
101


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


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


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

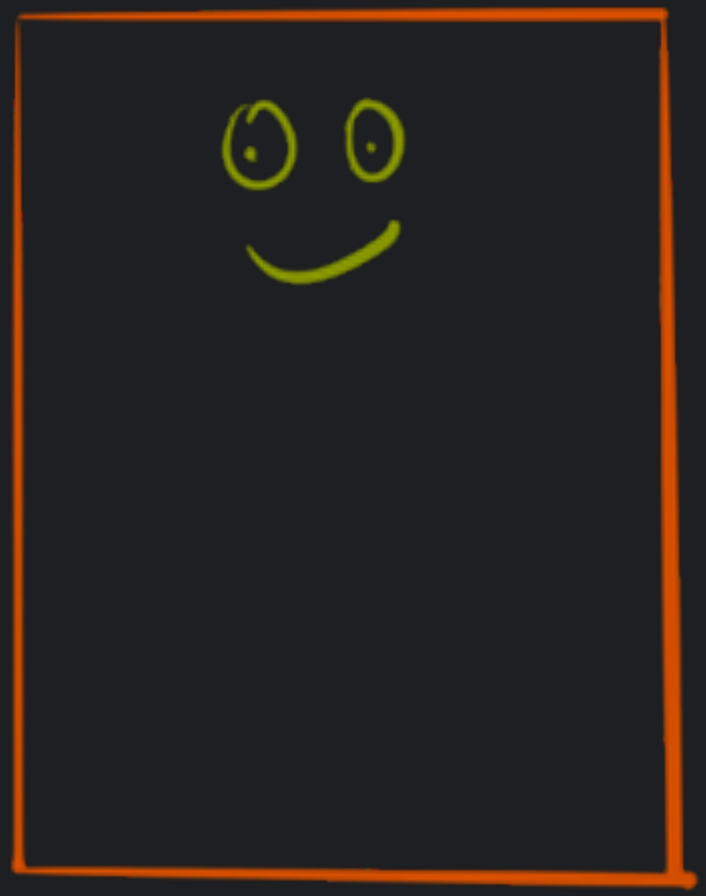

RDD
Partitions
Locations?
compute()
Dependencies
Partitioner?
1
2
100
101
1'
2'
100'
101'

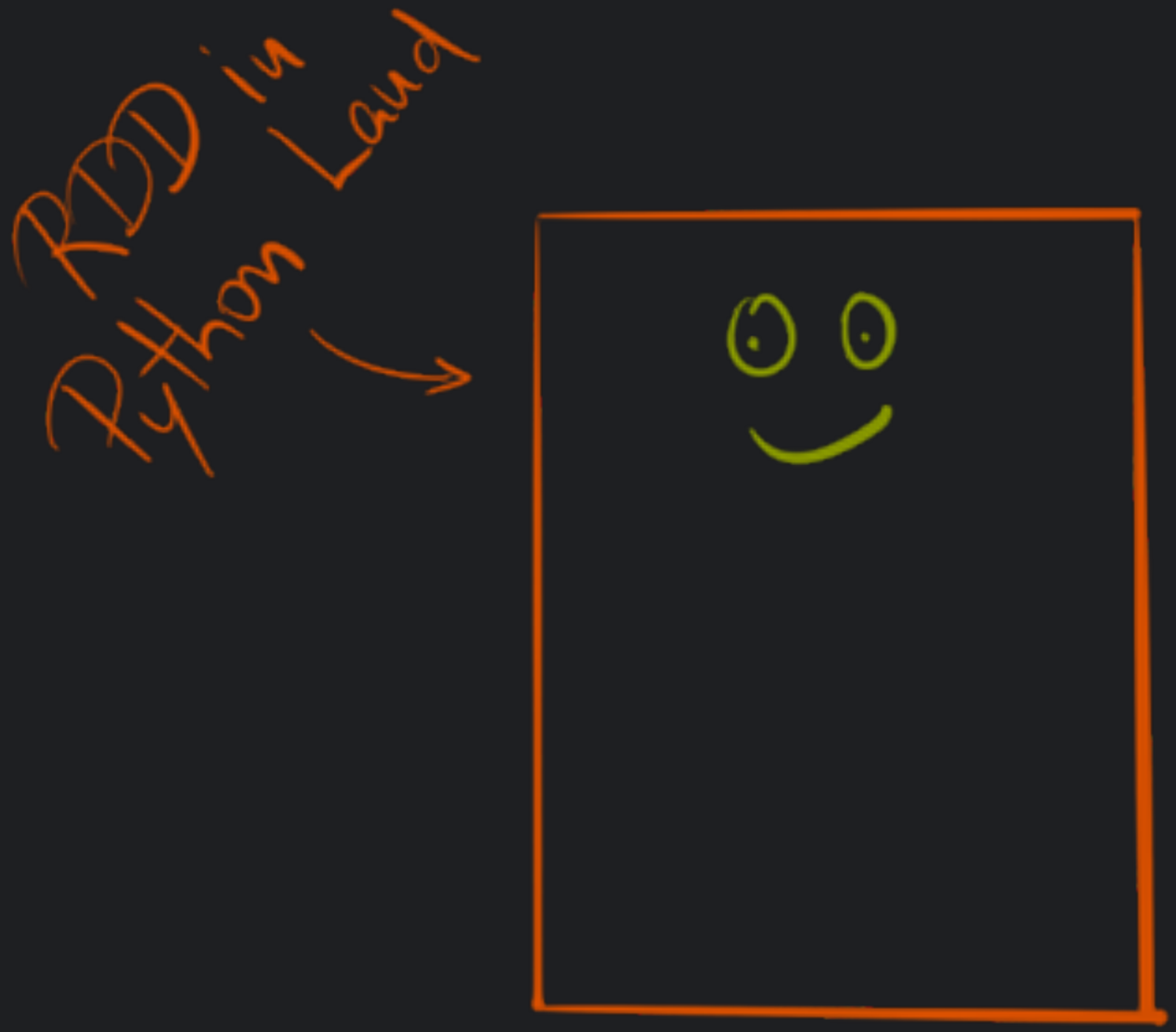
PYSPARK

PYSPARK OFFERS A PYTHON API TO THE SCALA CORE OF SPARK

IT USES THE PY4J BRIDGE

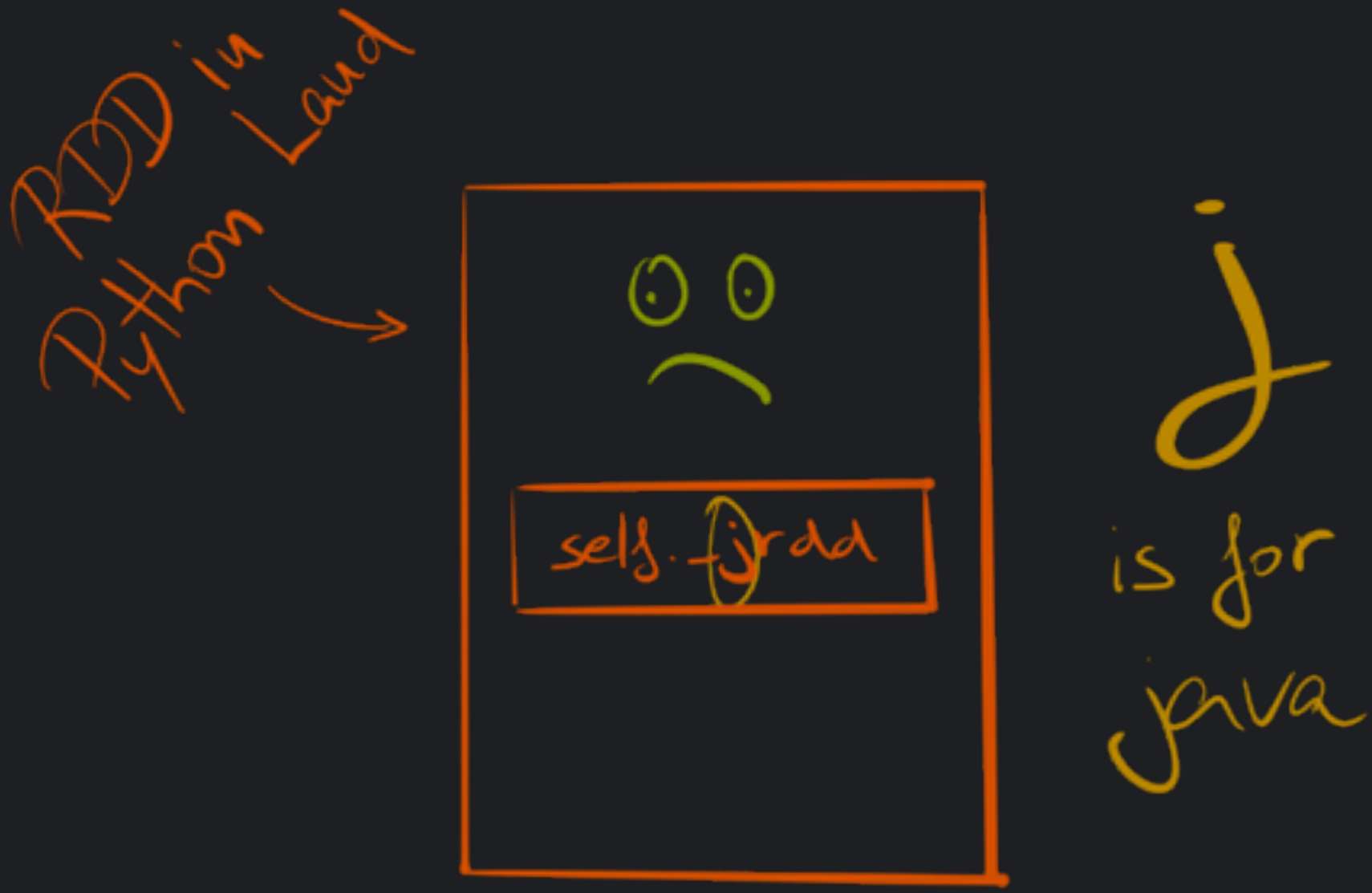
```
# Connect to the gateway
gateway = JavaGateway(
    gateway_parameters=GatewayParameters(
        port=gateway_port,
        auth_token=gateway_secret,
        auto_convert=True))

# 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
```

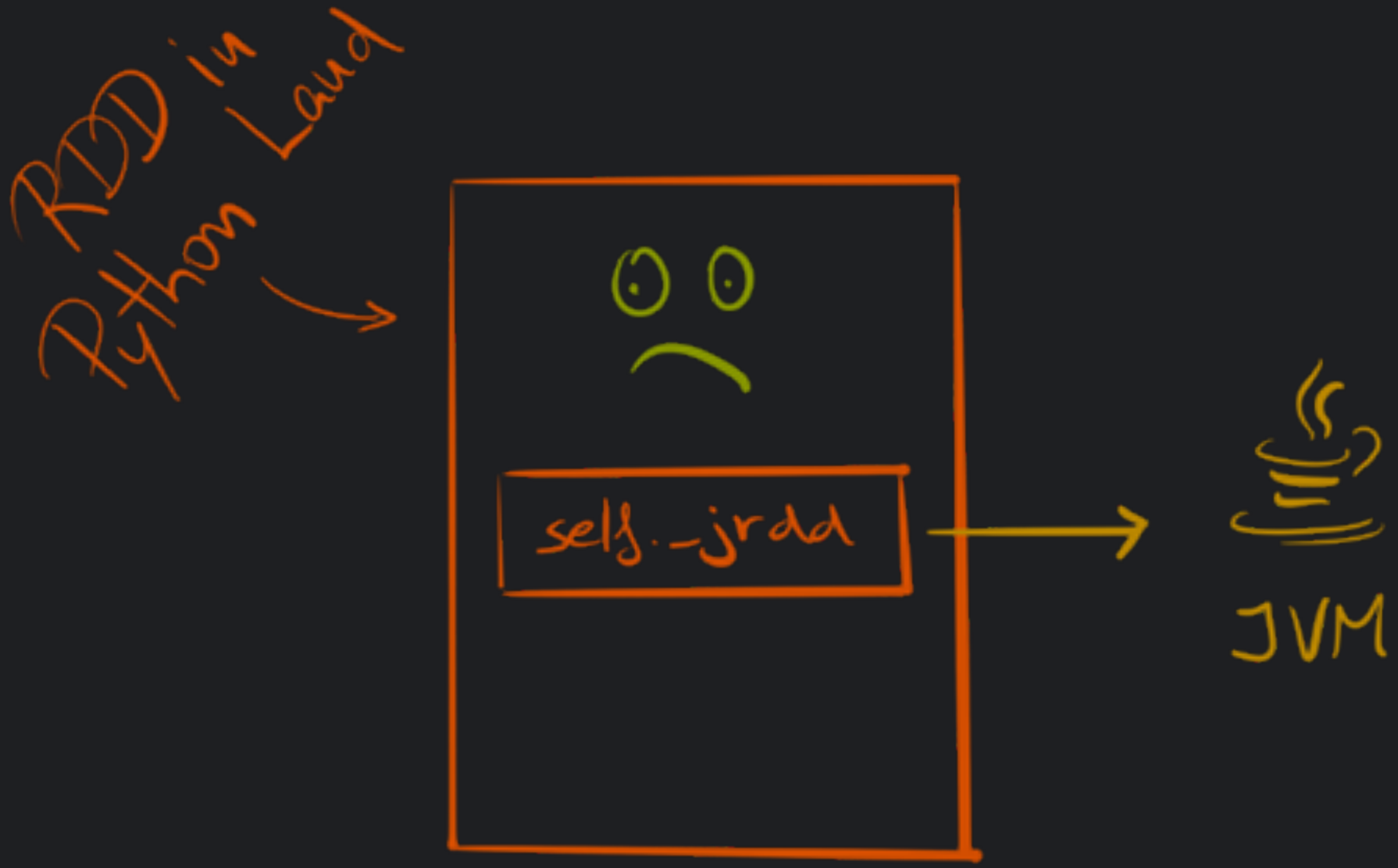


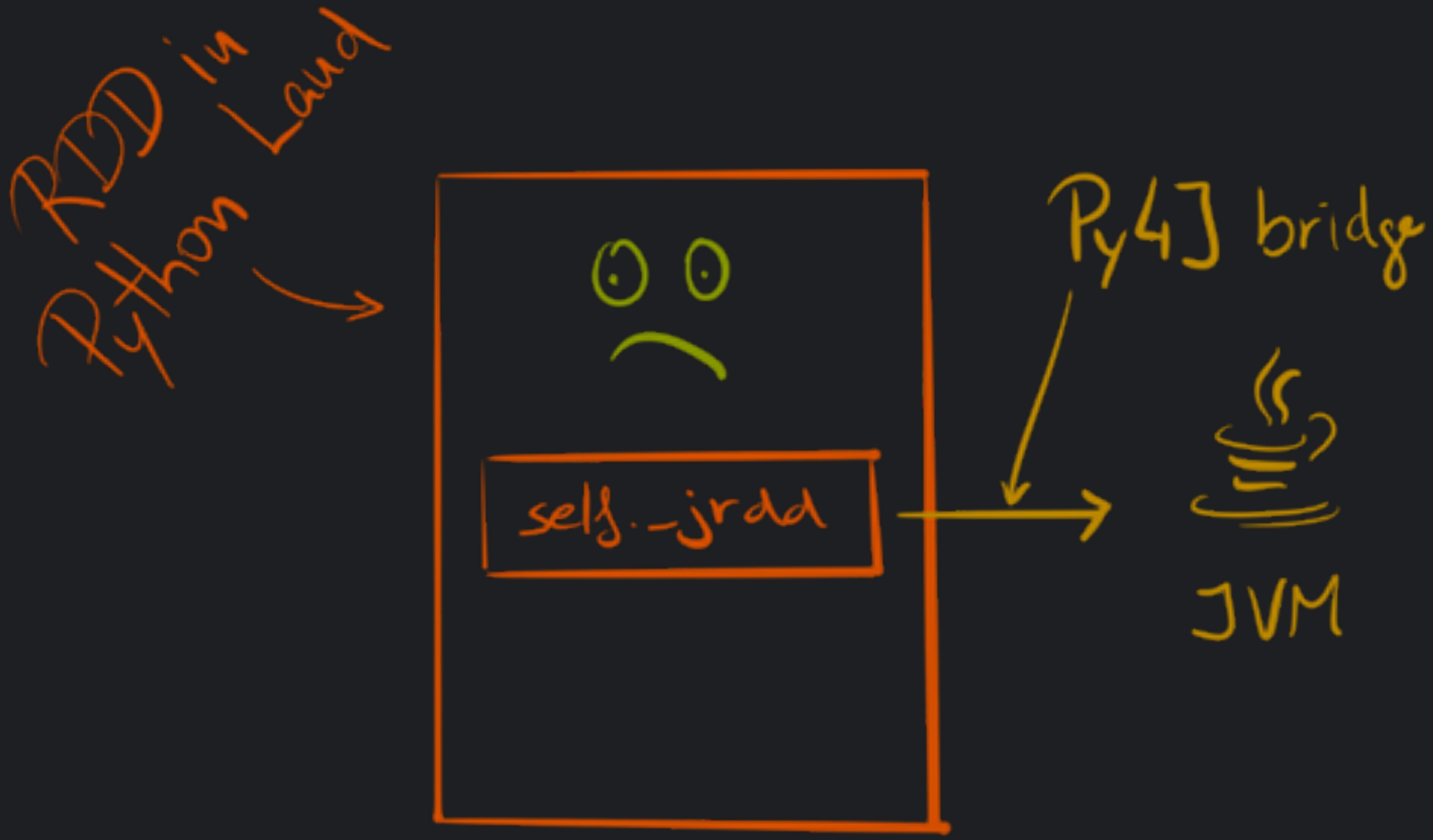

RDD in
Python Land


RDD in Python Land
self._jrdd


RDD in Python Land
self._jrdd
M is for murder


RDD in Python Land
self._jrdd
J is for Java


RDD in Python Land
self._jrdd
JVM

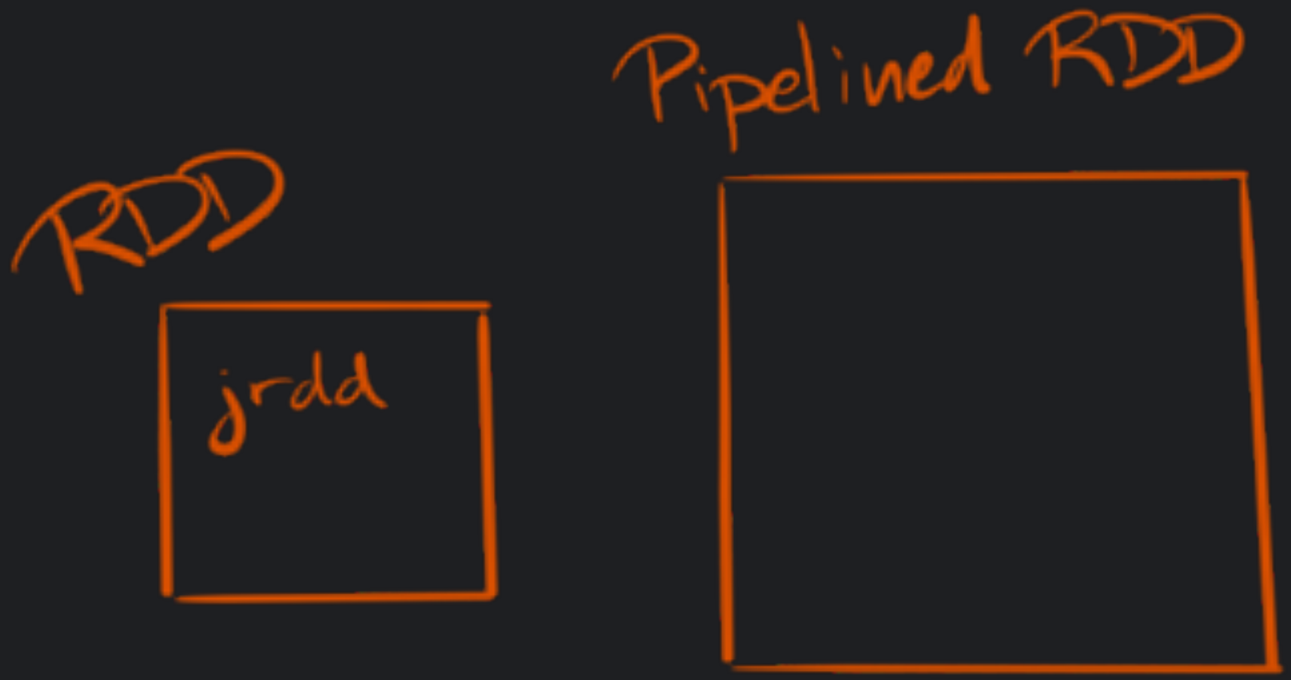

RDD in Python Land
self._jrdd
Py4J bridge
JVM

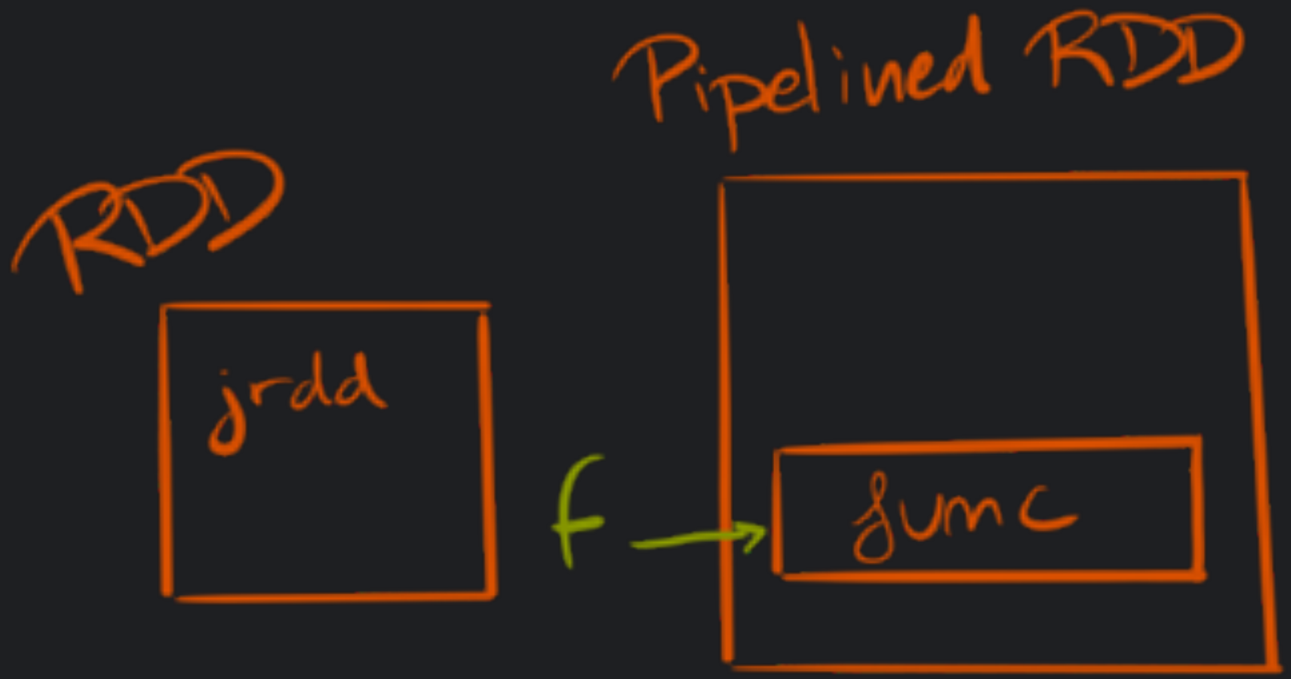
THE MAIN ENTRYPOINTS ARE RDD AND PipelinedRDD(RDD)

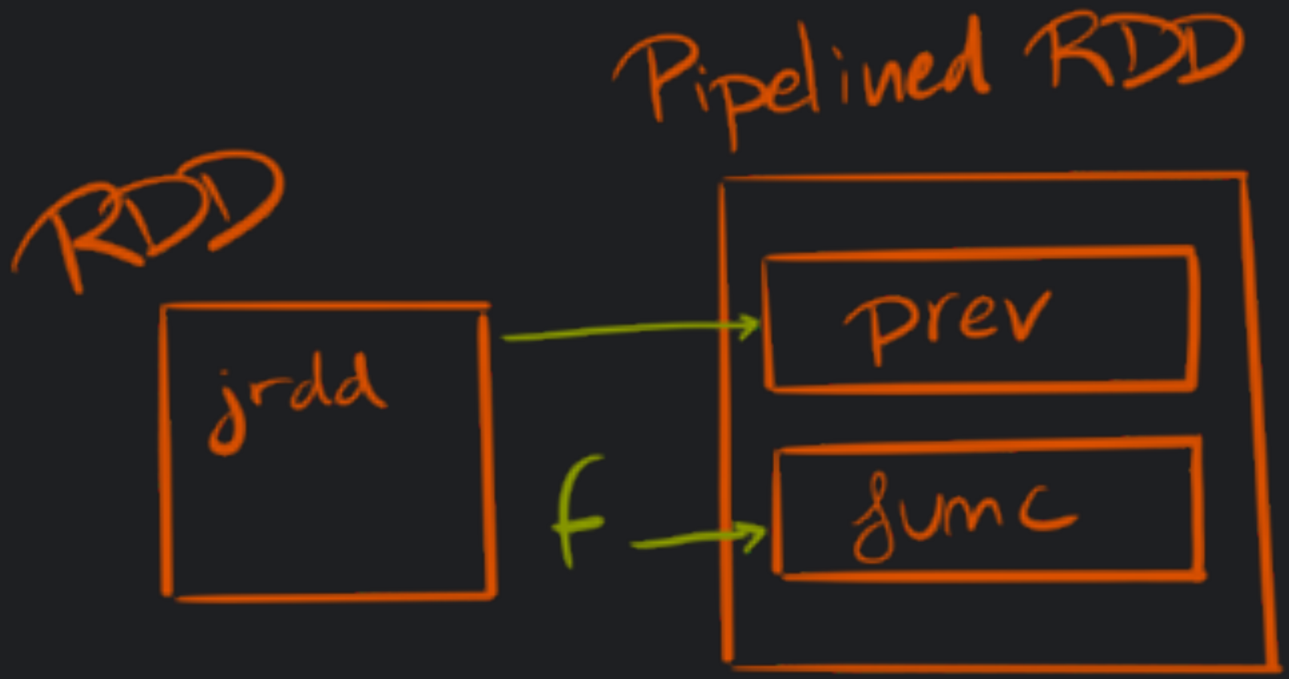
PipelinedRDD
BUILDS IN THE JVM A
PythonRDD

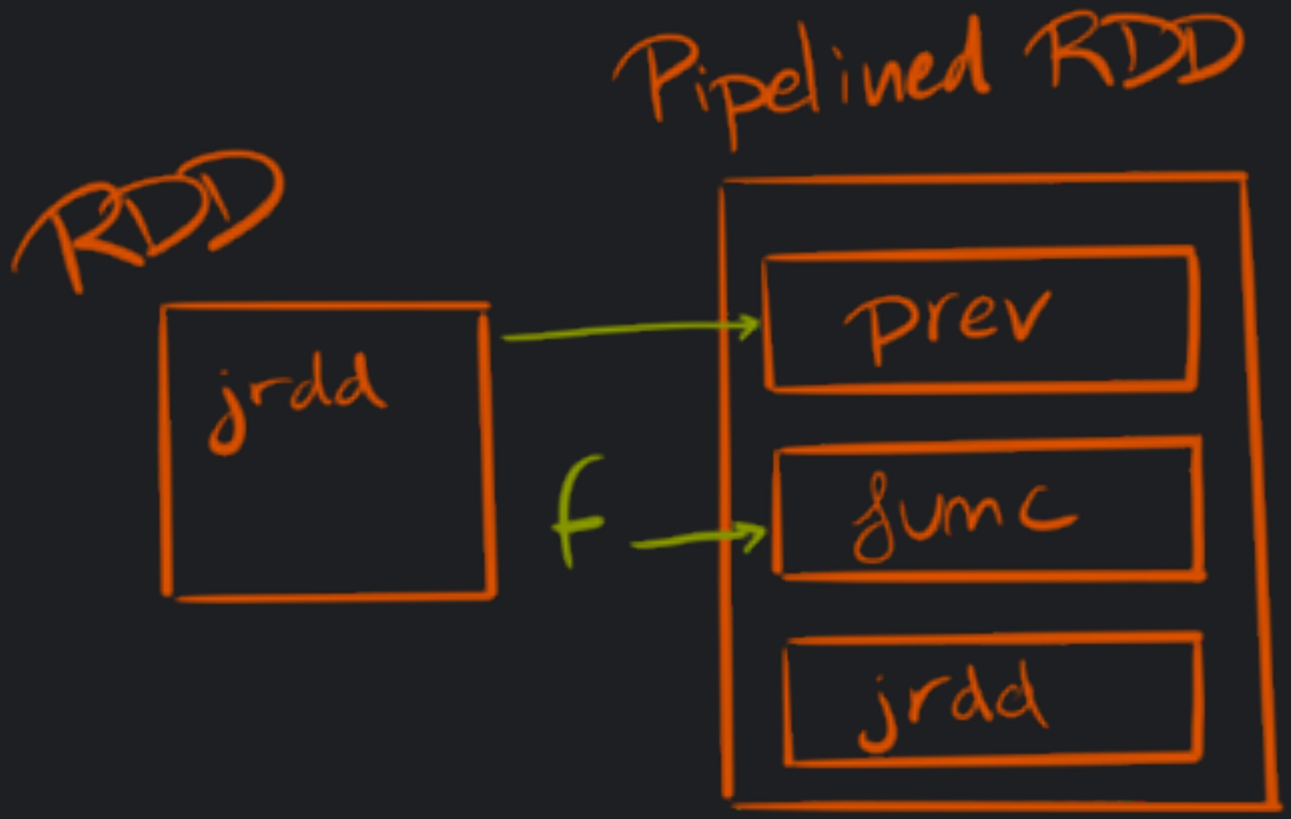

RDD
jrdd

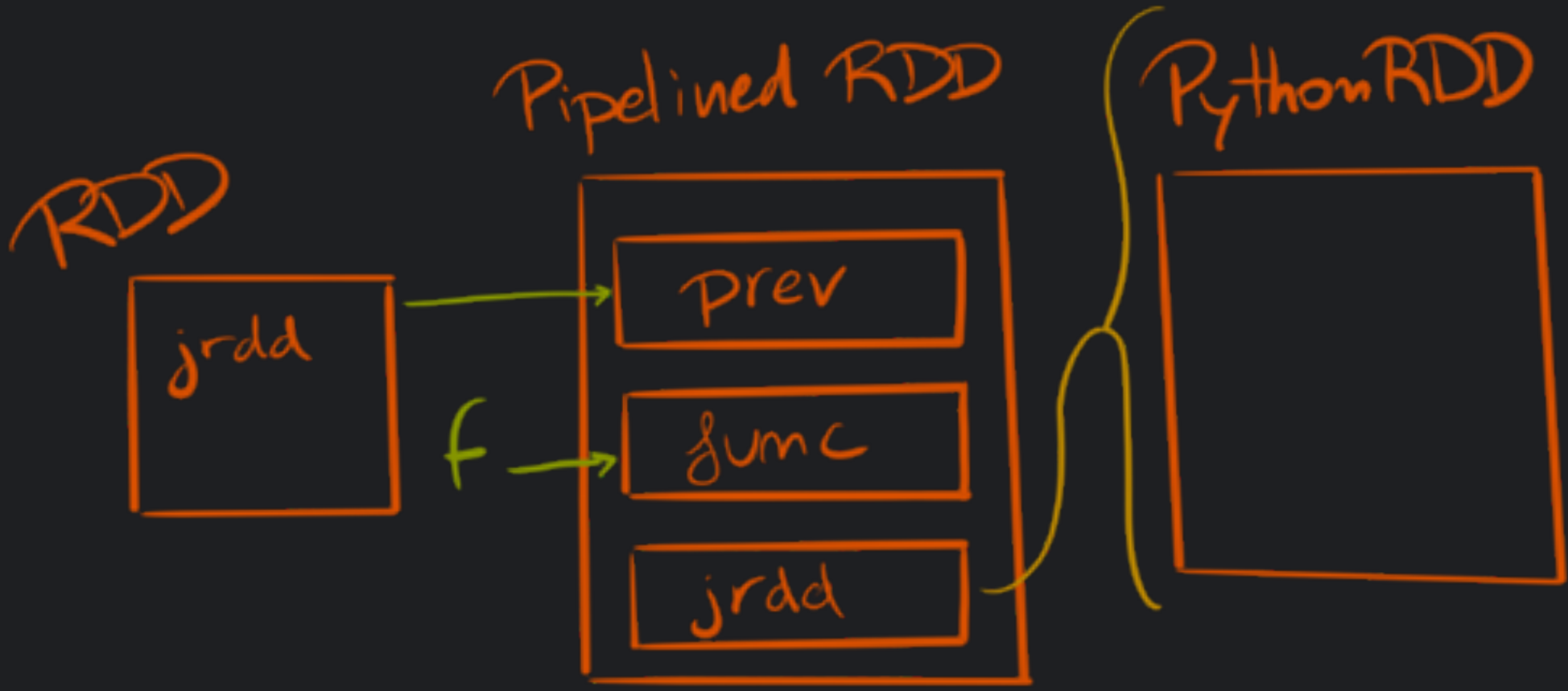

RDD
jrdd
MAP(f)

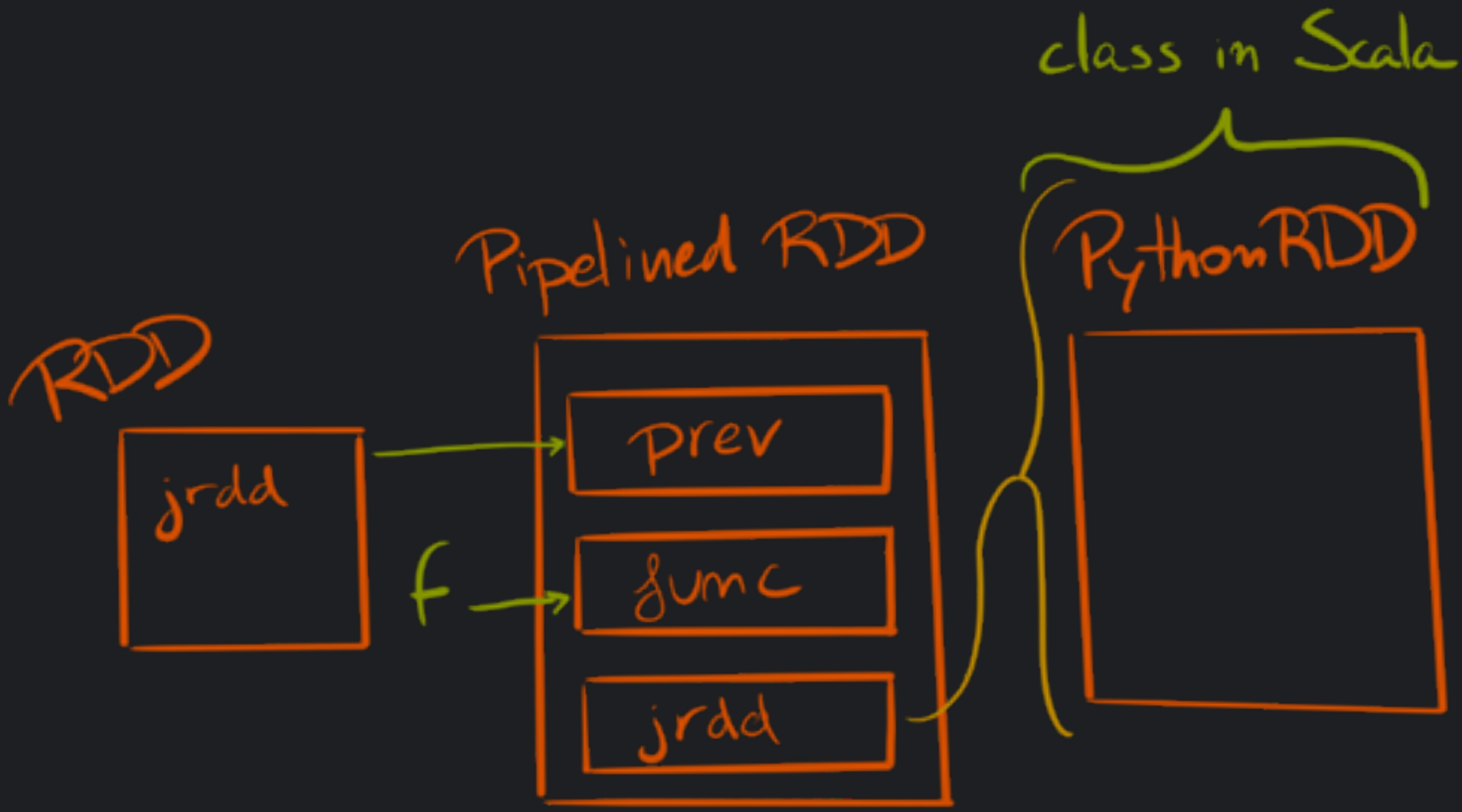

Pipelined RDD
RDD
jrdd

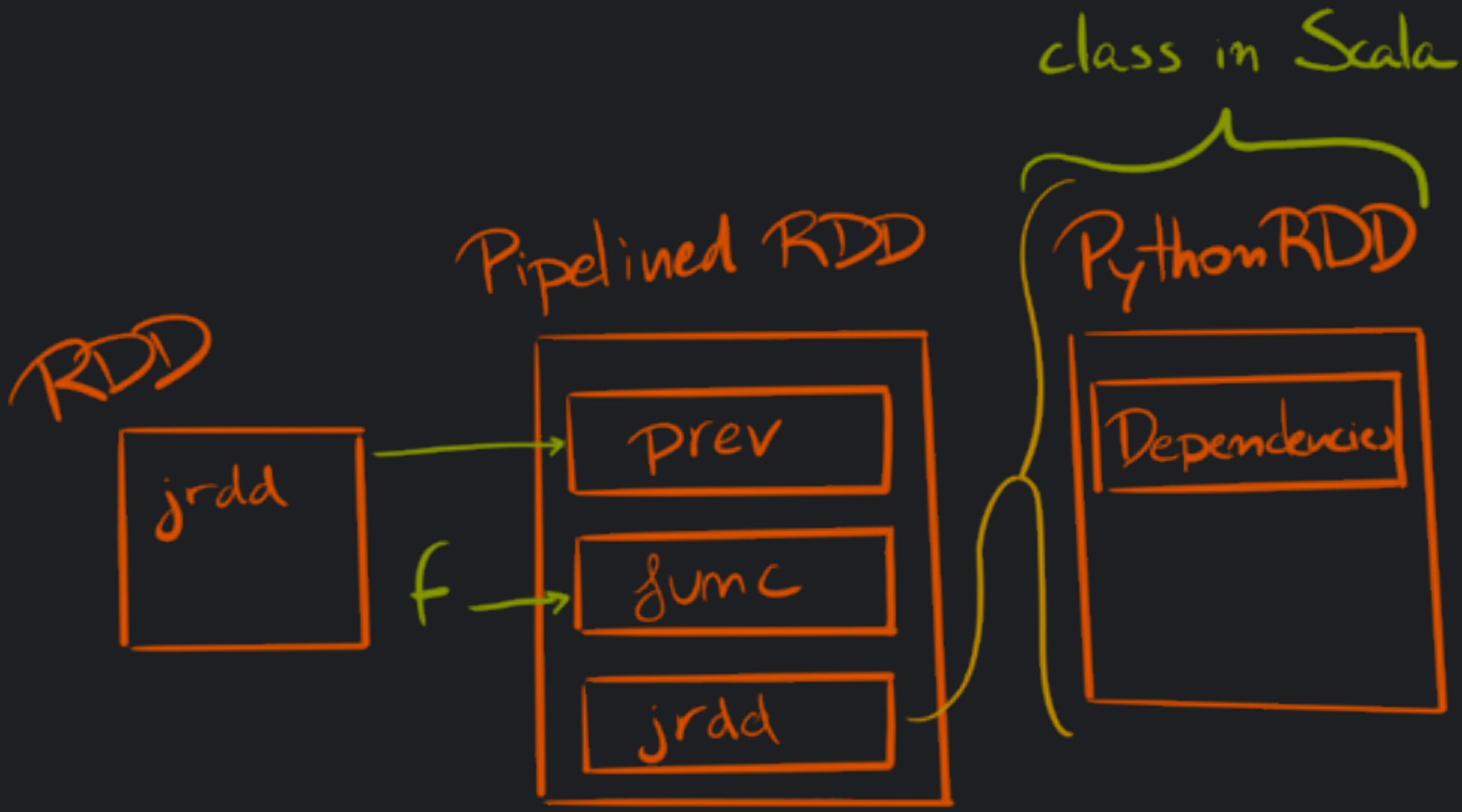

Pipelined RDD
RDD
jrdd
f
jumc

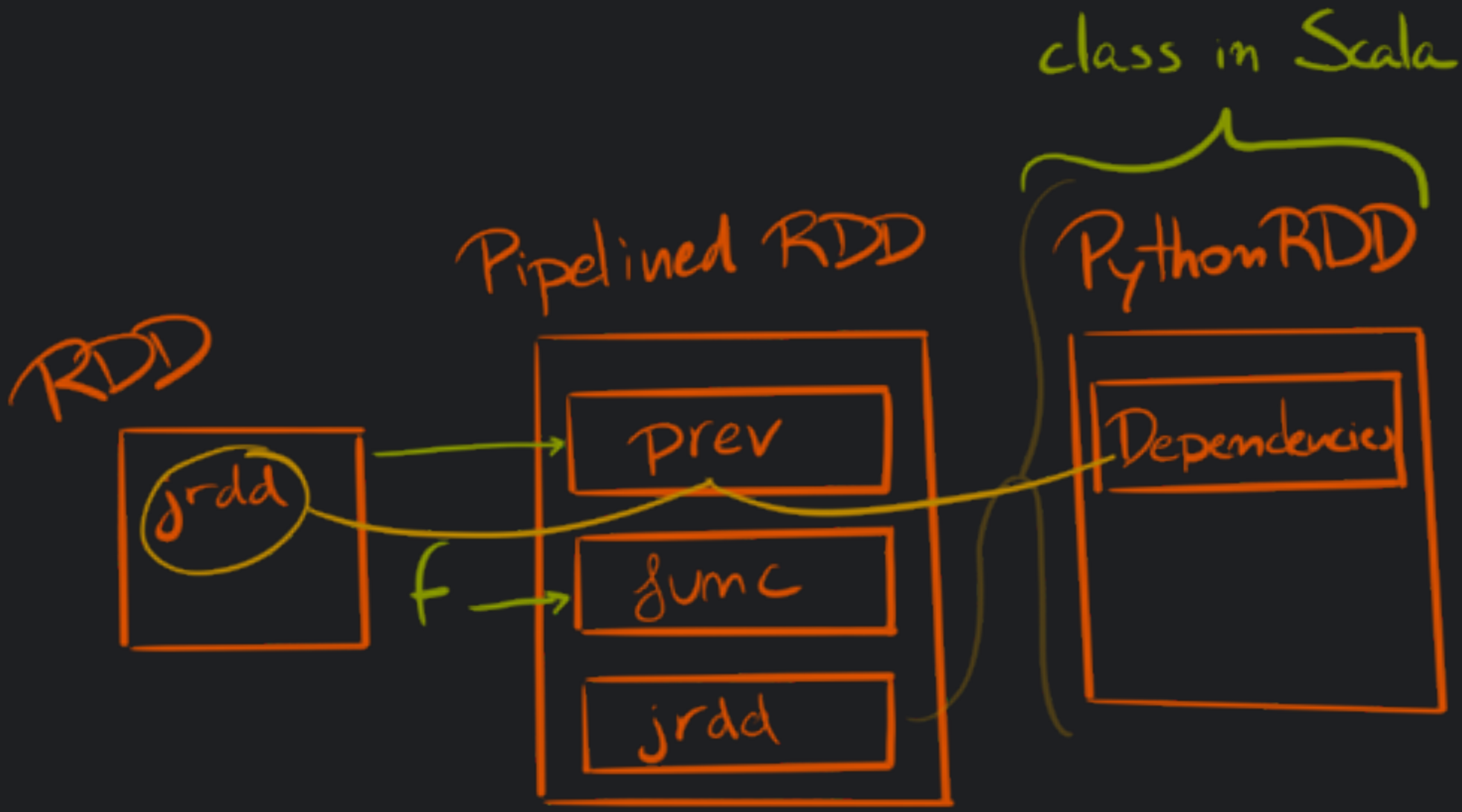

Pipelined RDD
RDD
jrdd
f
prev
func

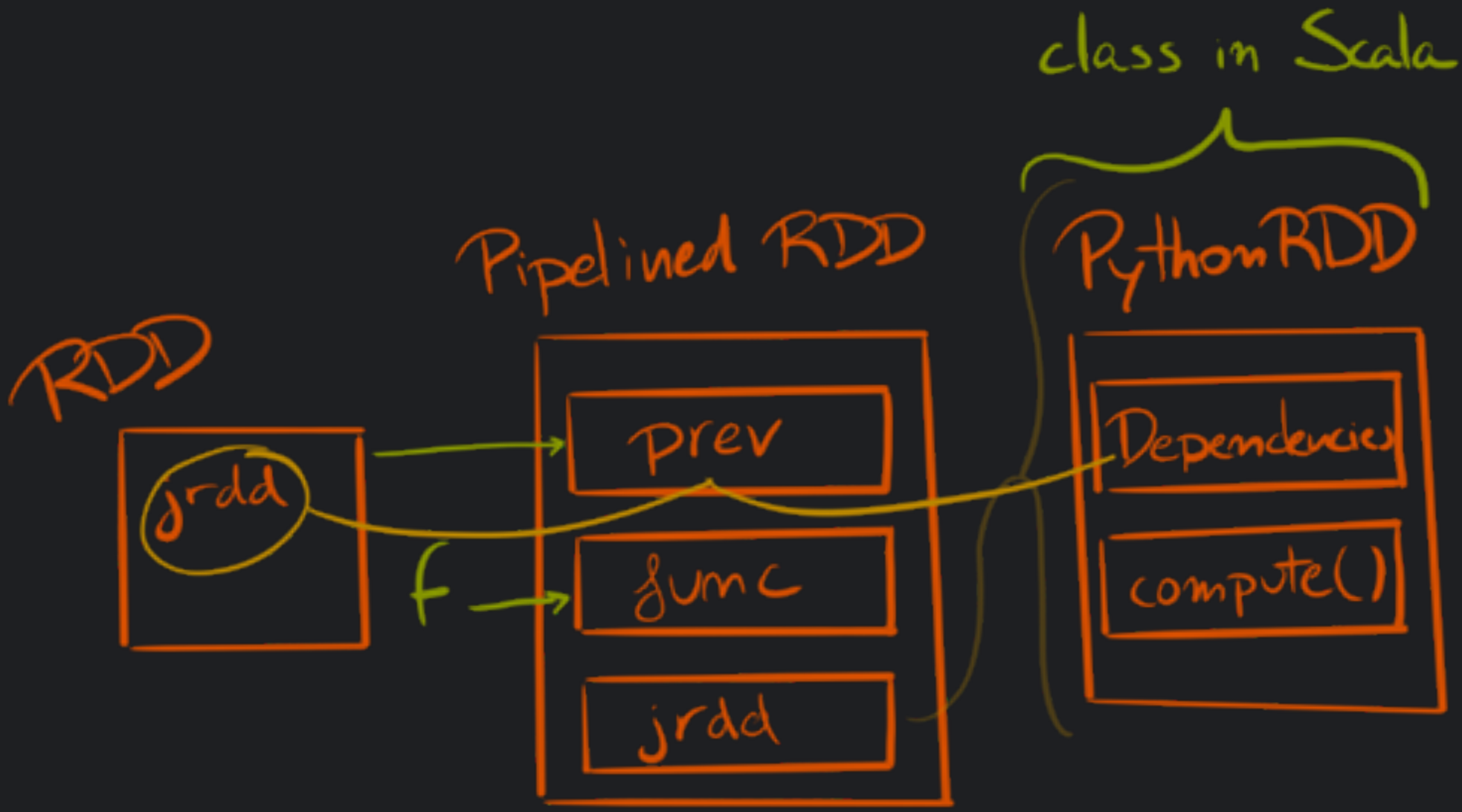

Pipelined RDD
RDD
jrdd
f
prev
func
jrdd

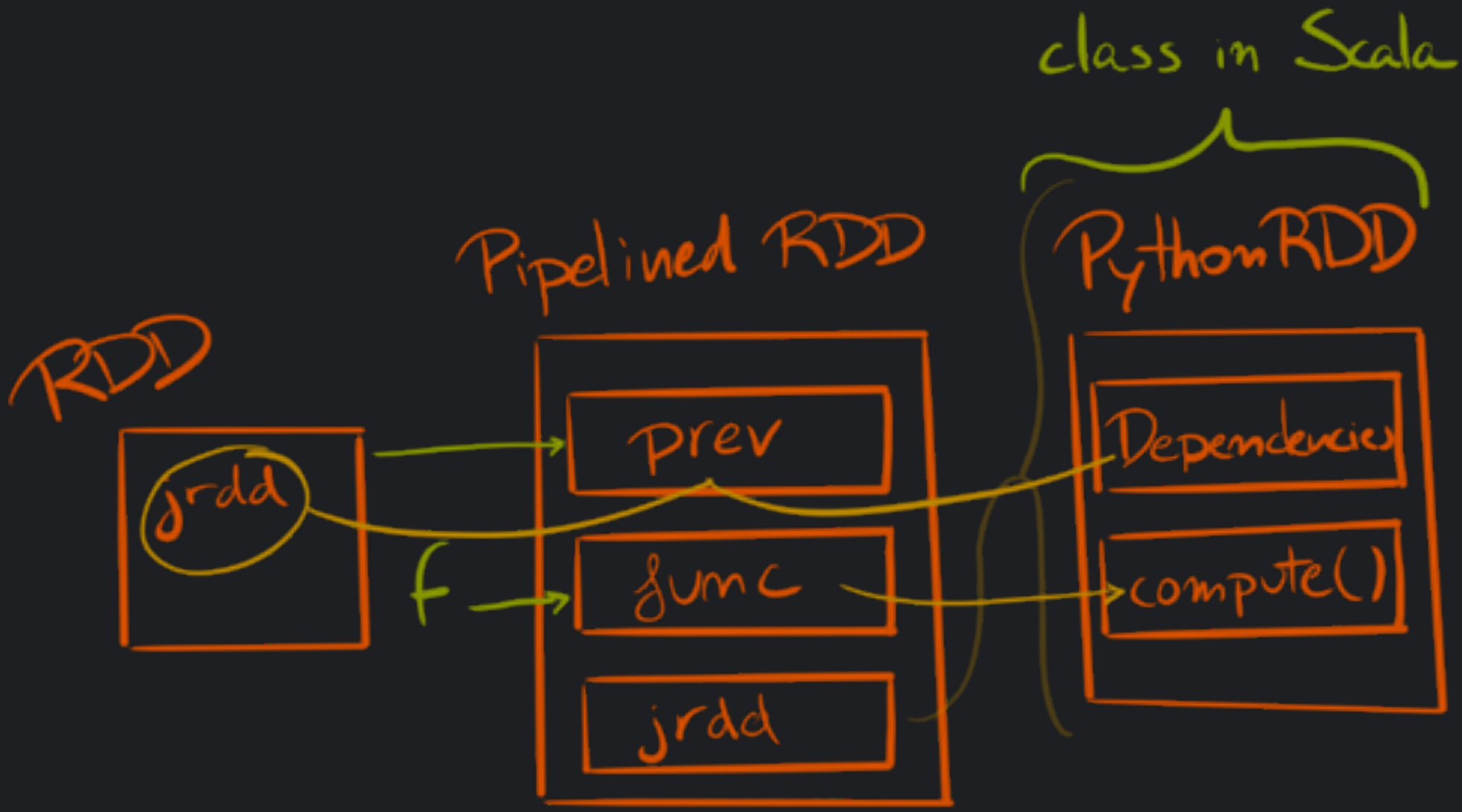

RDD
jrdd
f
Pipelined RDD
prev
func
jrdd
PythonRDD


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


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


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


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


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

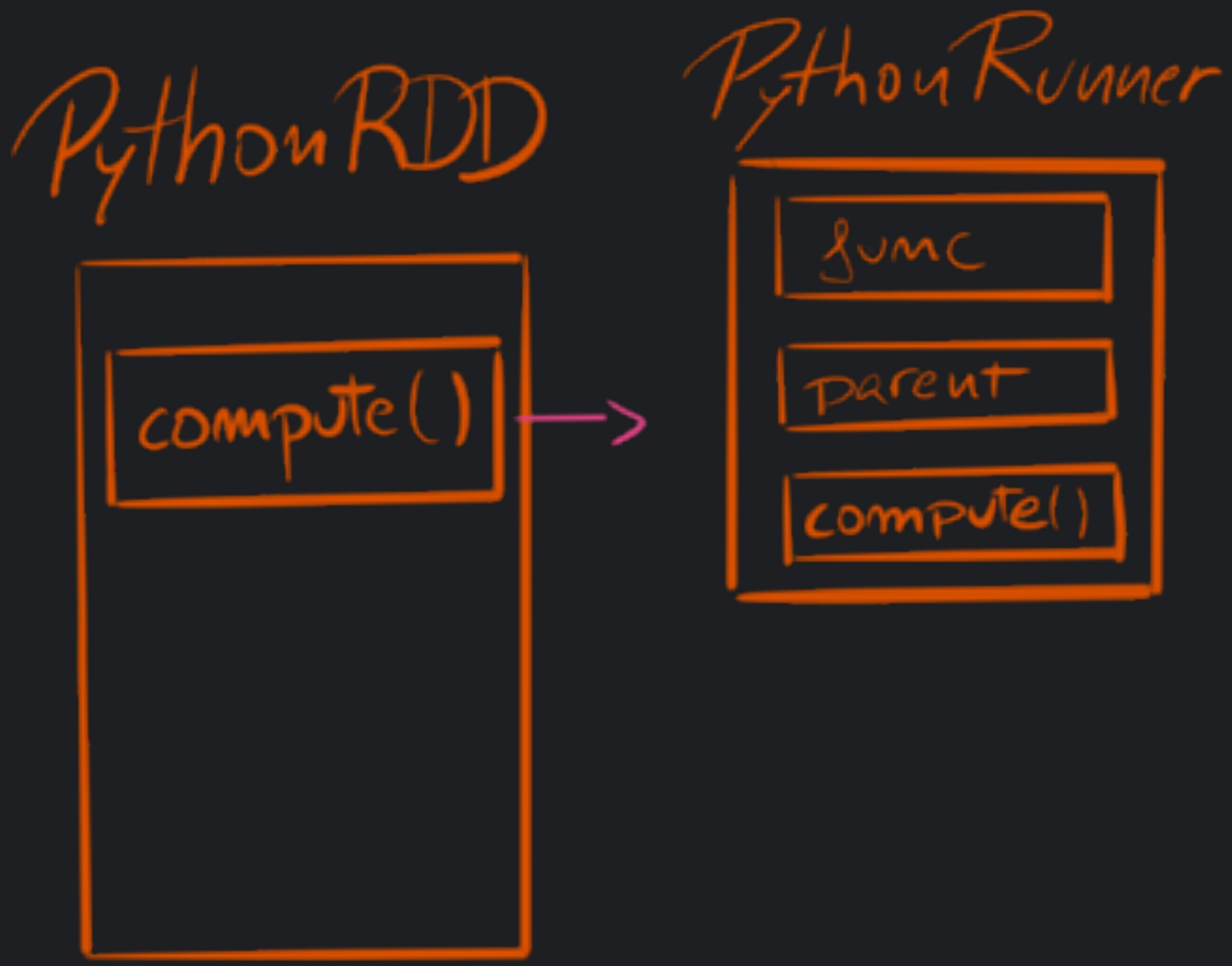
THE MAGIC IS IN compute

compute

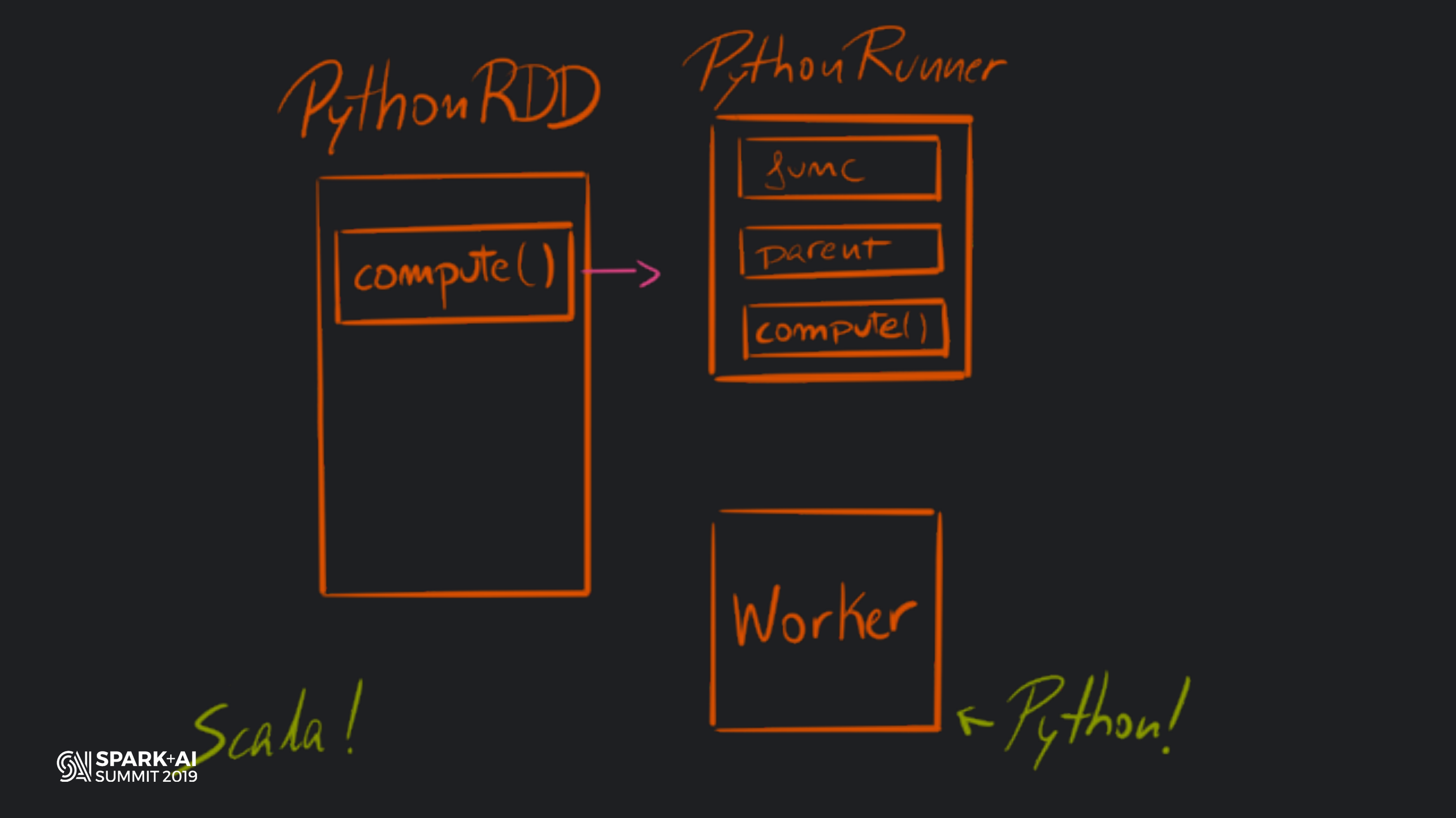
IS RUN ON EACH EXECUTOR AND STARTS A PYTHON WORKER VIA

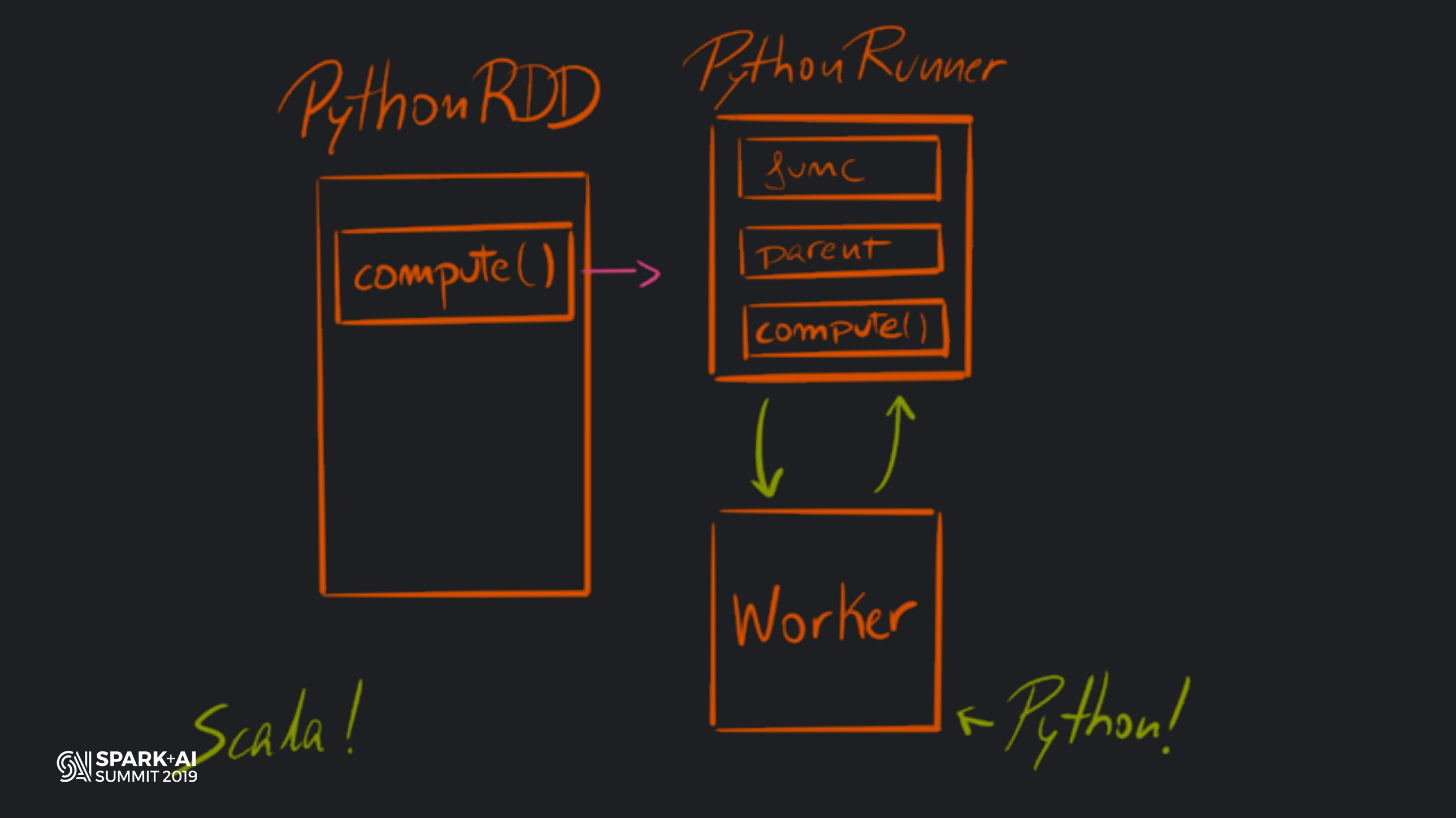
PythonRunner

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... WASN'T SPARK MAGICALLY OPTIMISING EVERYTHING?

YES. FOR SPARK DataFrame

SPARK WILL GENERATE A PLAN

(A DIRECTED ACYCLIC GRAPH)

TO COMPUTE THE RESULT

AND THE PLAN WILL BE OPTIMISED USING CATALYST

DEPENDING ON THE FUNCTION, THE OPTIMISER WILL CHOOSE

PythonUDFRunner

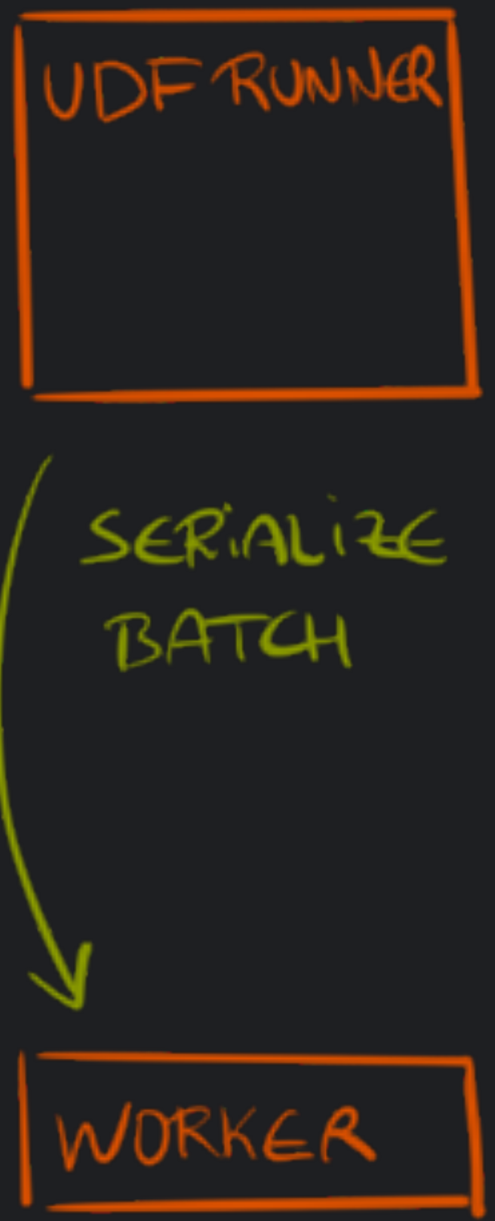
OR

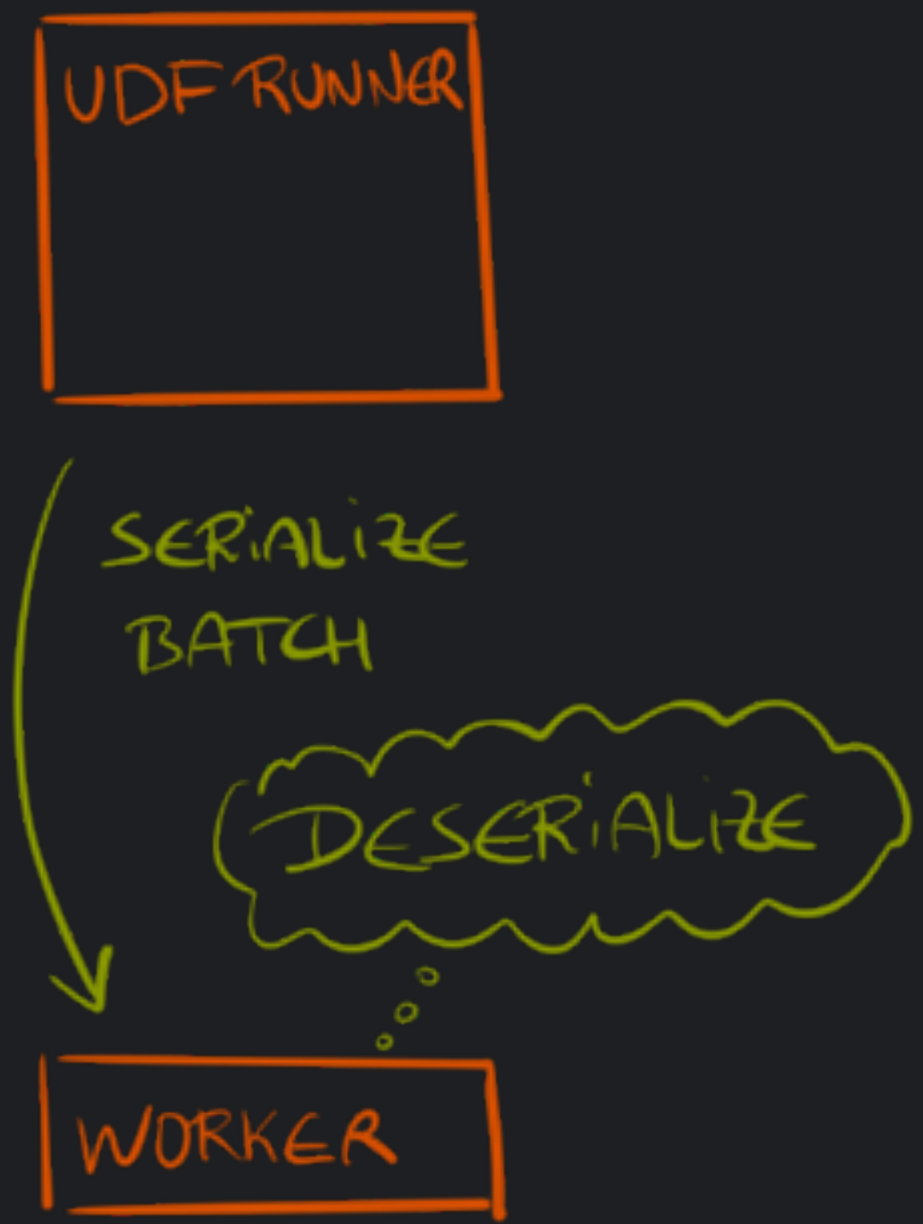
PythonArrowRunner

(BOTH EXTEND PythonRunner)

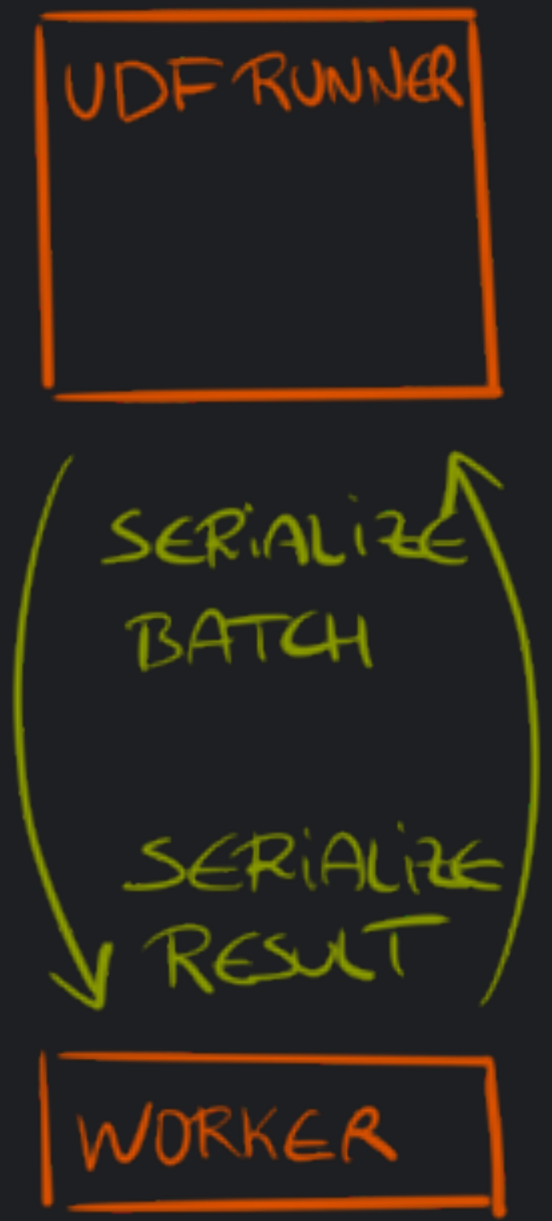

UDF RUNNER

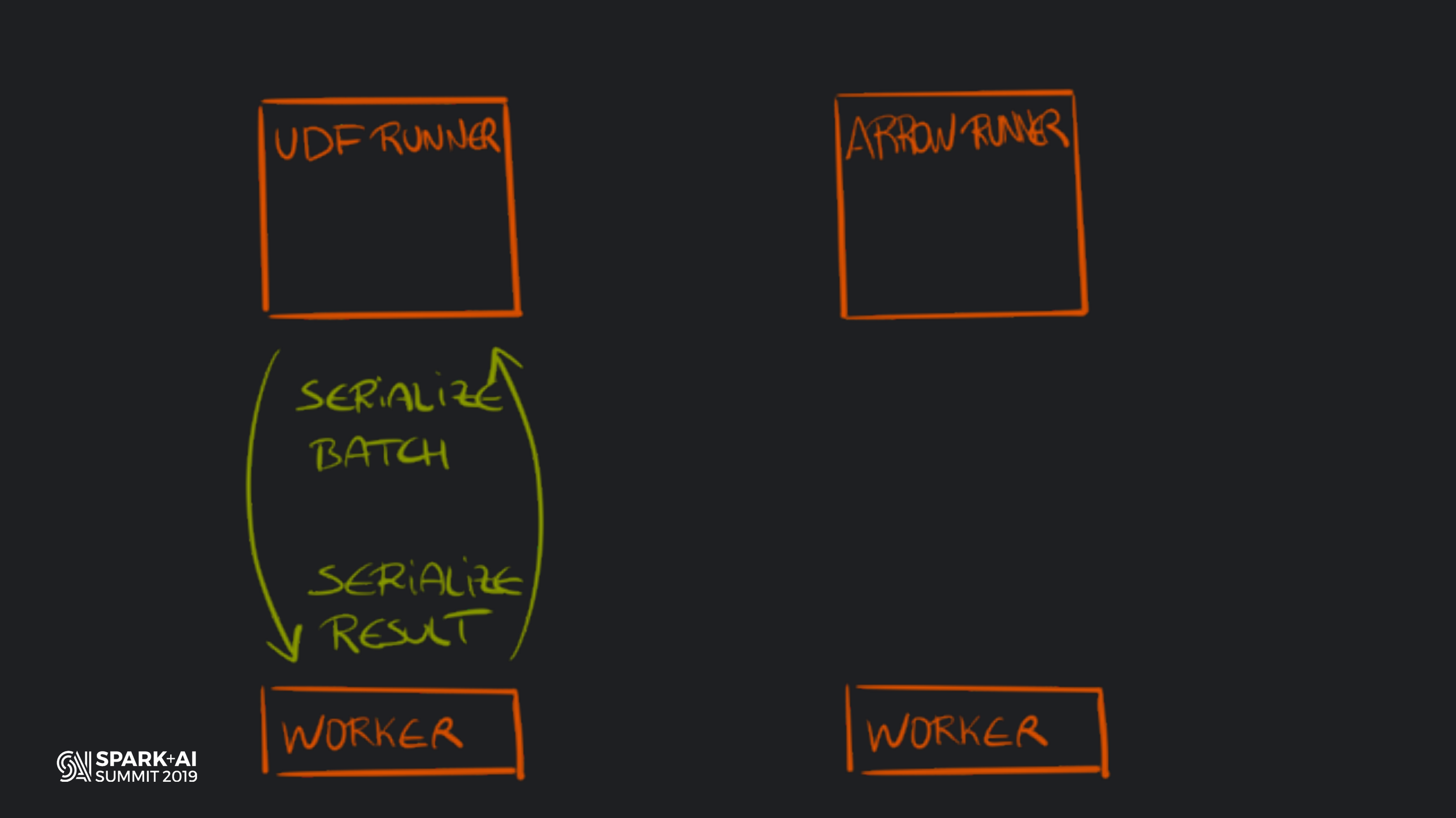

WORKER

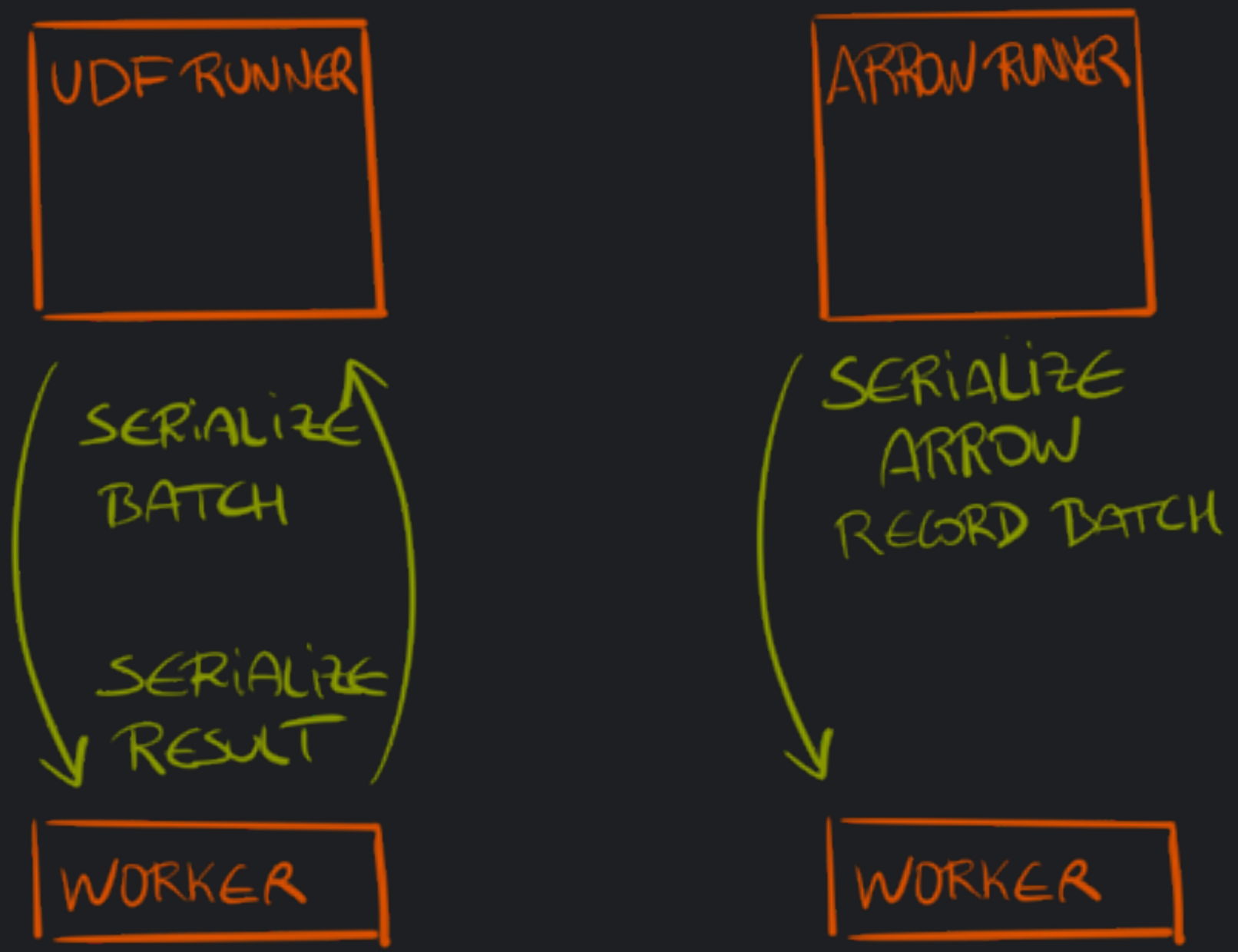

UDF RUNNER
SERIALIZE
BATCH
WORKER

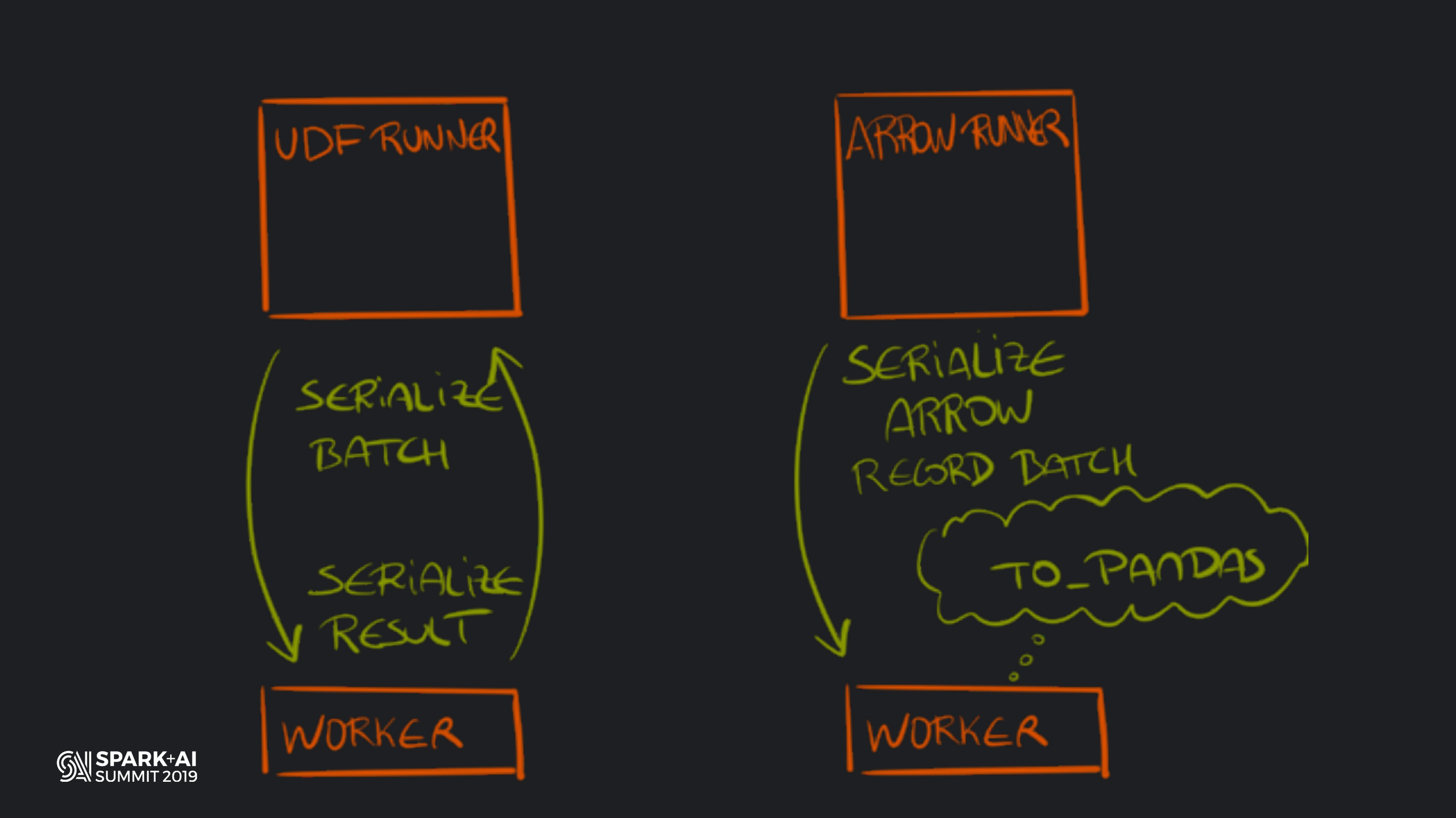

UDF RUNNER
SERIALIZE
BATCH
DESERIALIZE
WORKER

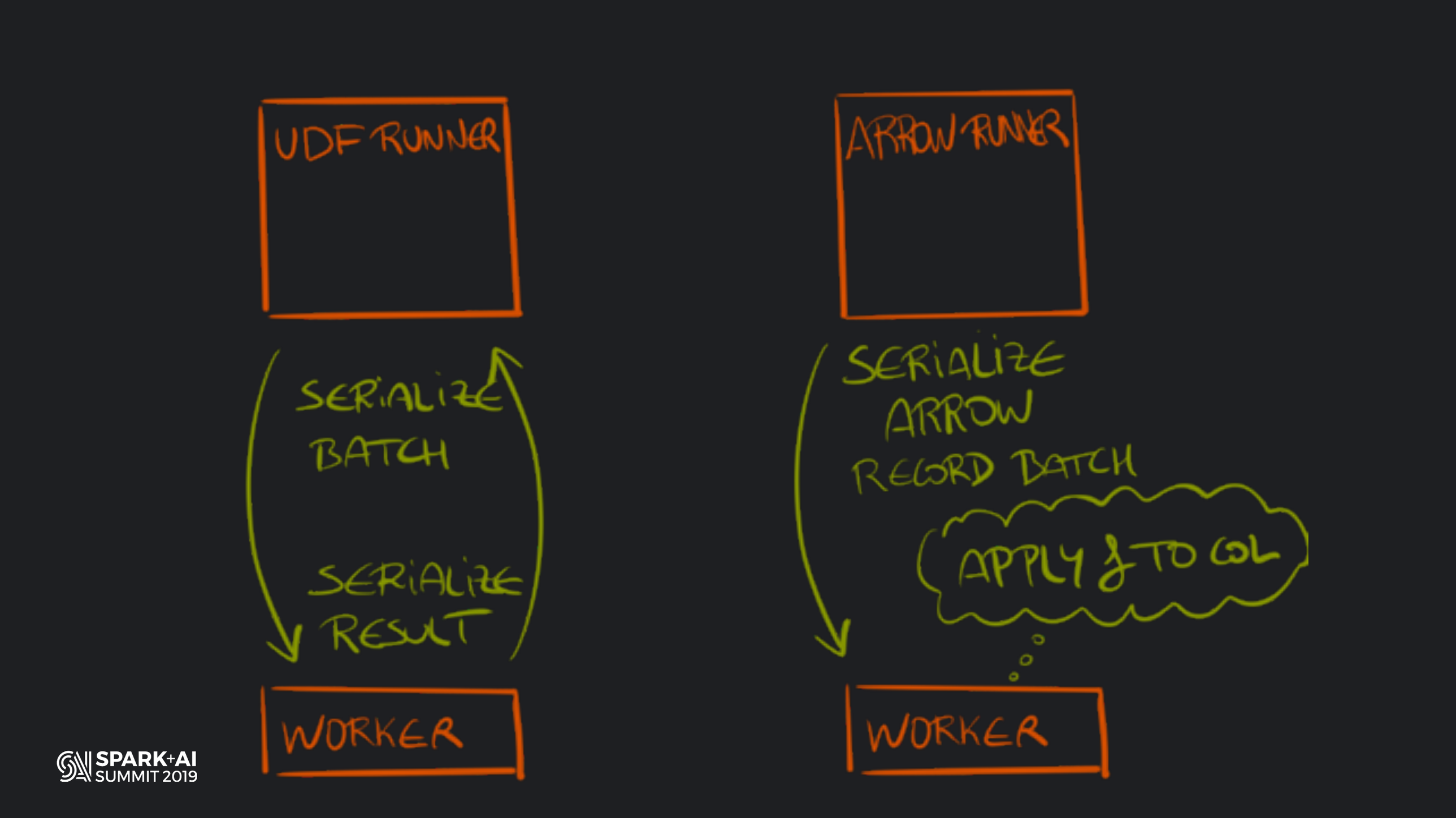

UDF RUNNER
SERIALIZE
BATCH
APPLY f
WORKER

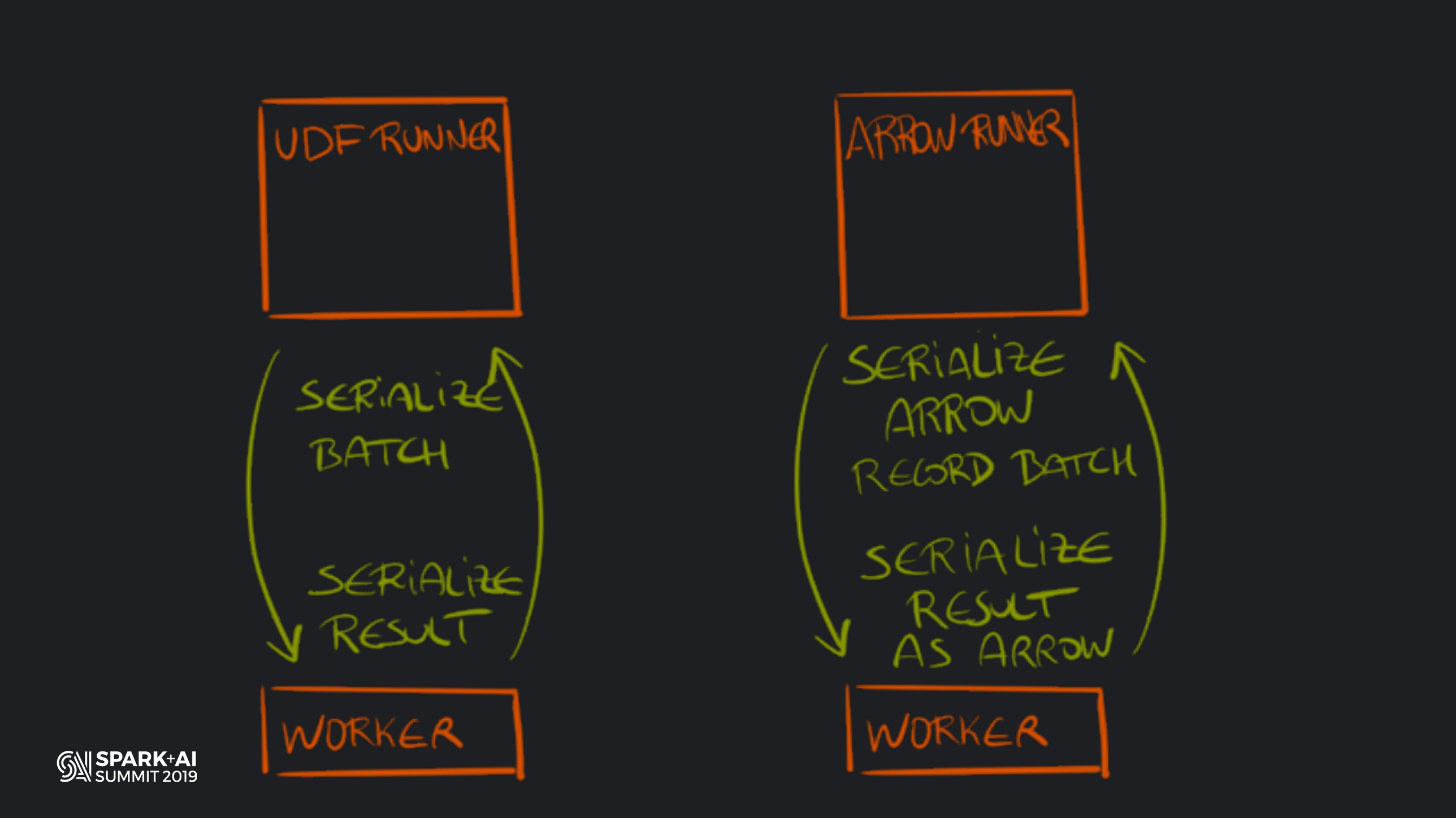

UDF RUNNER
SERIALIZE BATCH
SERIALIZE RESULT
WORKER


UDF RUNNER
ARROW RUNNER
SERIALIZE BATCH
SERIALIZE RESULT
WORKER
WORKER


UDF RUNNER
SERIALIZE BATCH
SERIALIZE RESULT
WORKER
ARROW RUNNER
SERIALIZE ARROW RECORD BATCH
WORKER


UDF RUNNER
SERIALIZE BATCH
SERIALIZE RESULT
WORKER
ARROW RUNNER
SERIALIZE ARROW RECORD BATCH
TO_PANDAS
WORKER


UDF RUNNER
SERIALIZE BATCH
SERIALIZE RESULT
WORKER
ARROW RUNNER
SERIALIZE ARROW RECORD BATCH
APPLY f TO COL
WORKER


UDF RUNNER
SERIALIZE BATCH
SERIALIZE RESULT
WORKER
ARROW RUNNER
SERIALIZE ARROW RECORD BATCH
SERIALIZE RESULT AS ARROW
WORKER

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

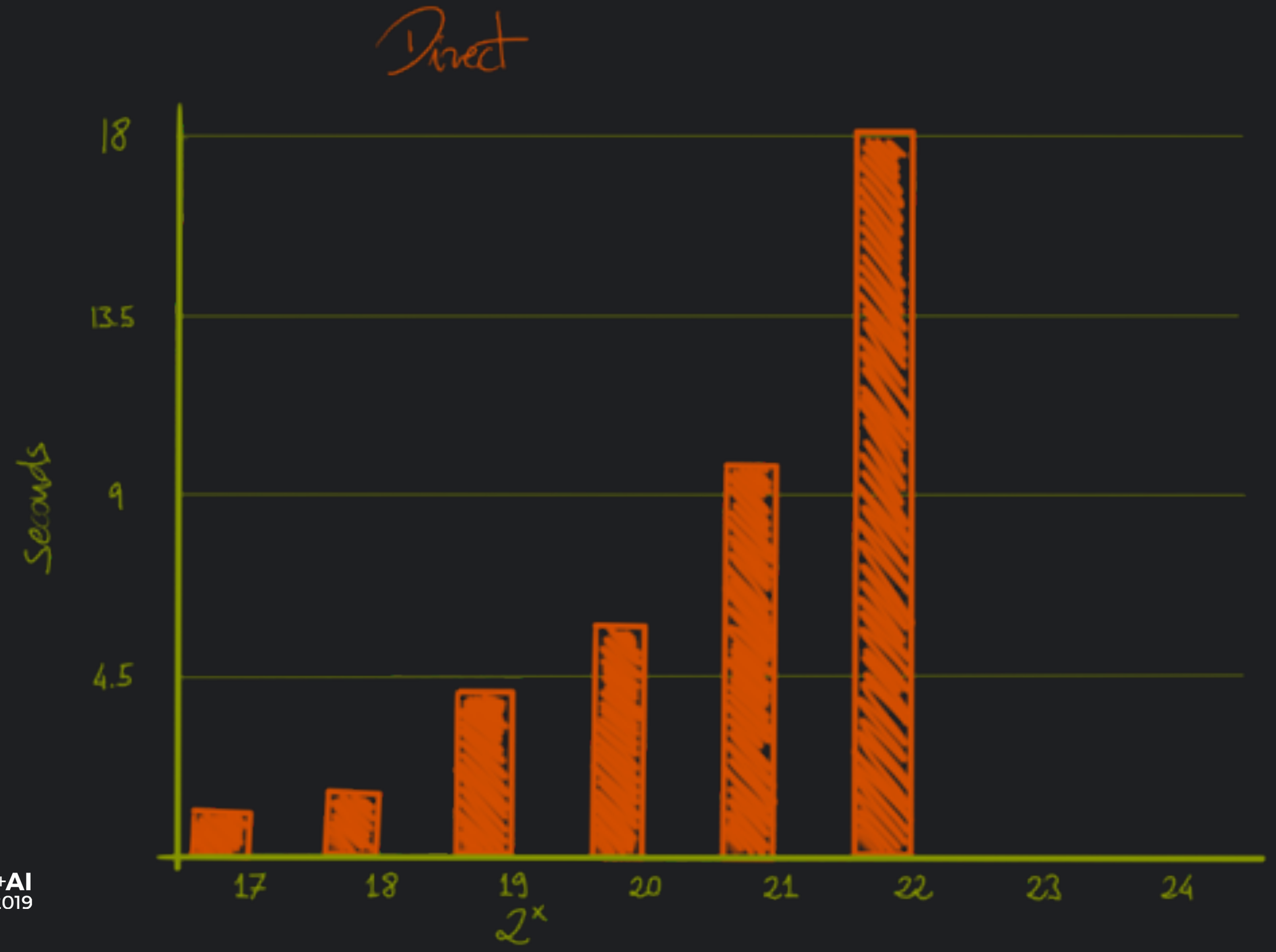
QUICK EXAMPLES

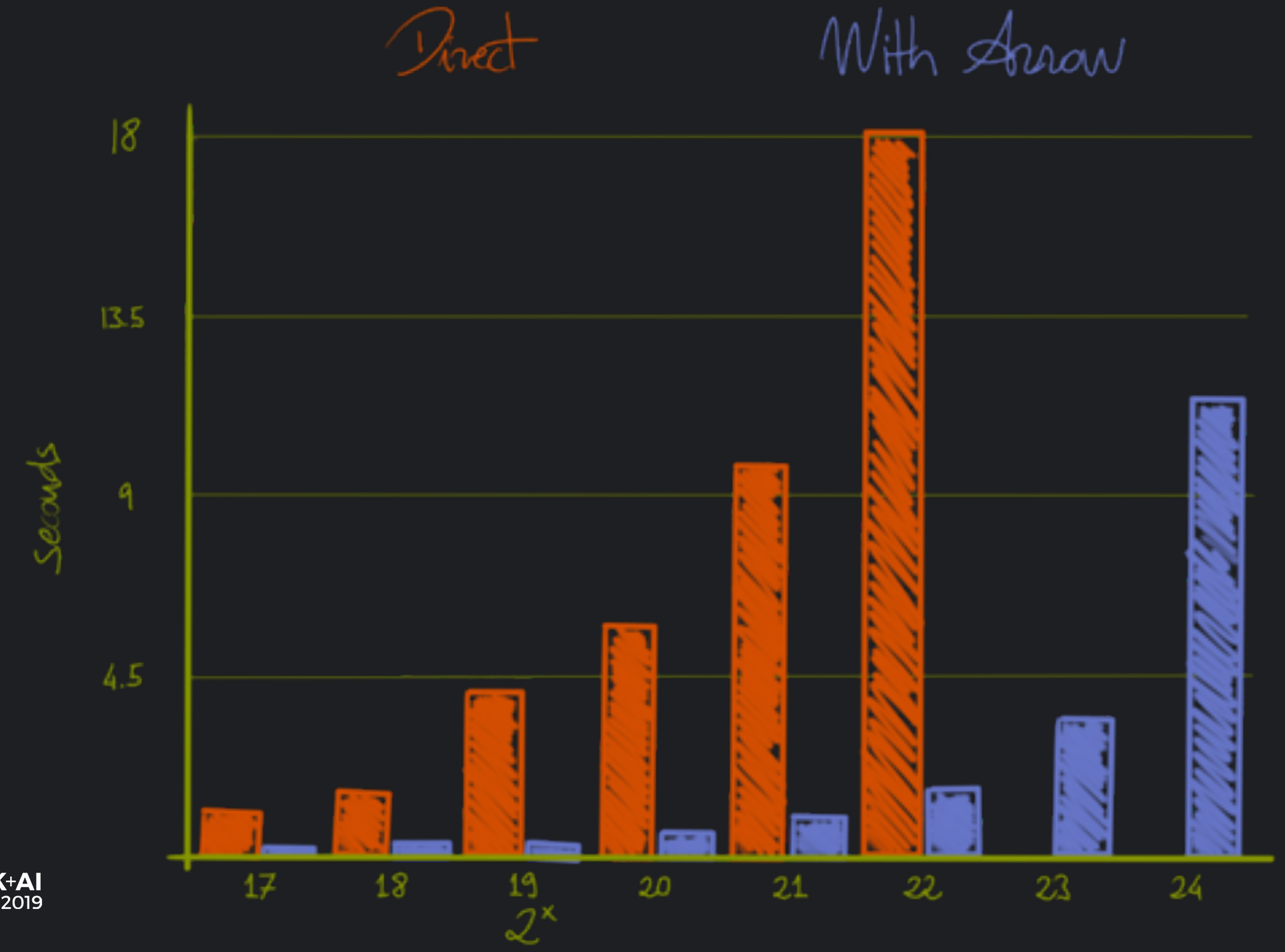
THE BASICS: toPandas

```
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
```

```
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
```


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

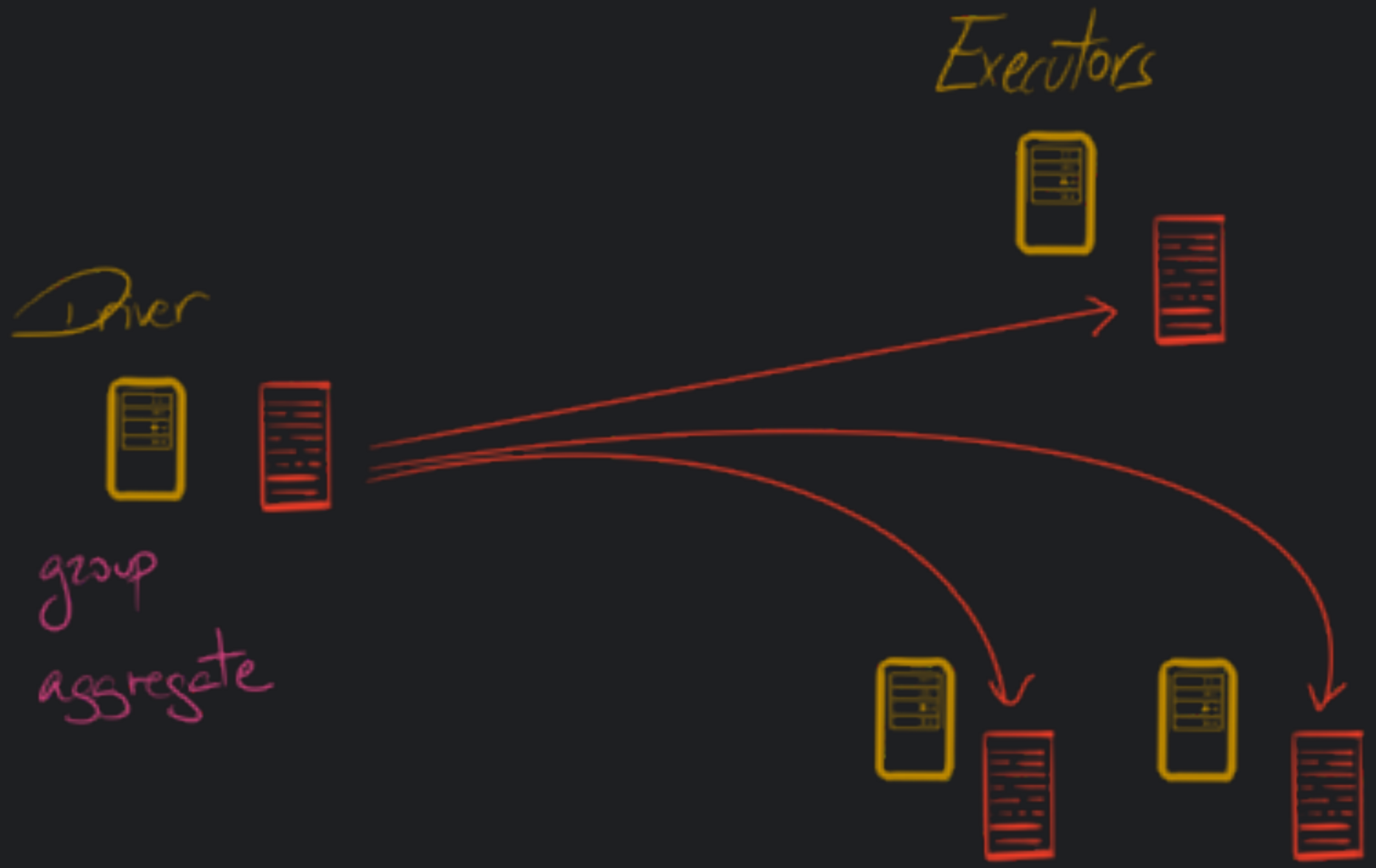

Direct
With Arrow
18
13.5
9
4.5
Seconds
17
18
19
20
21
22
23
24
2^x

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()
```

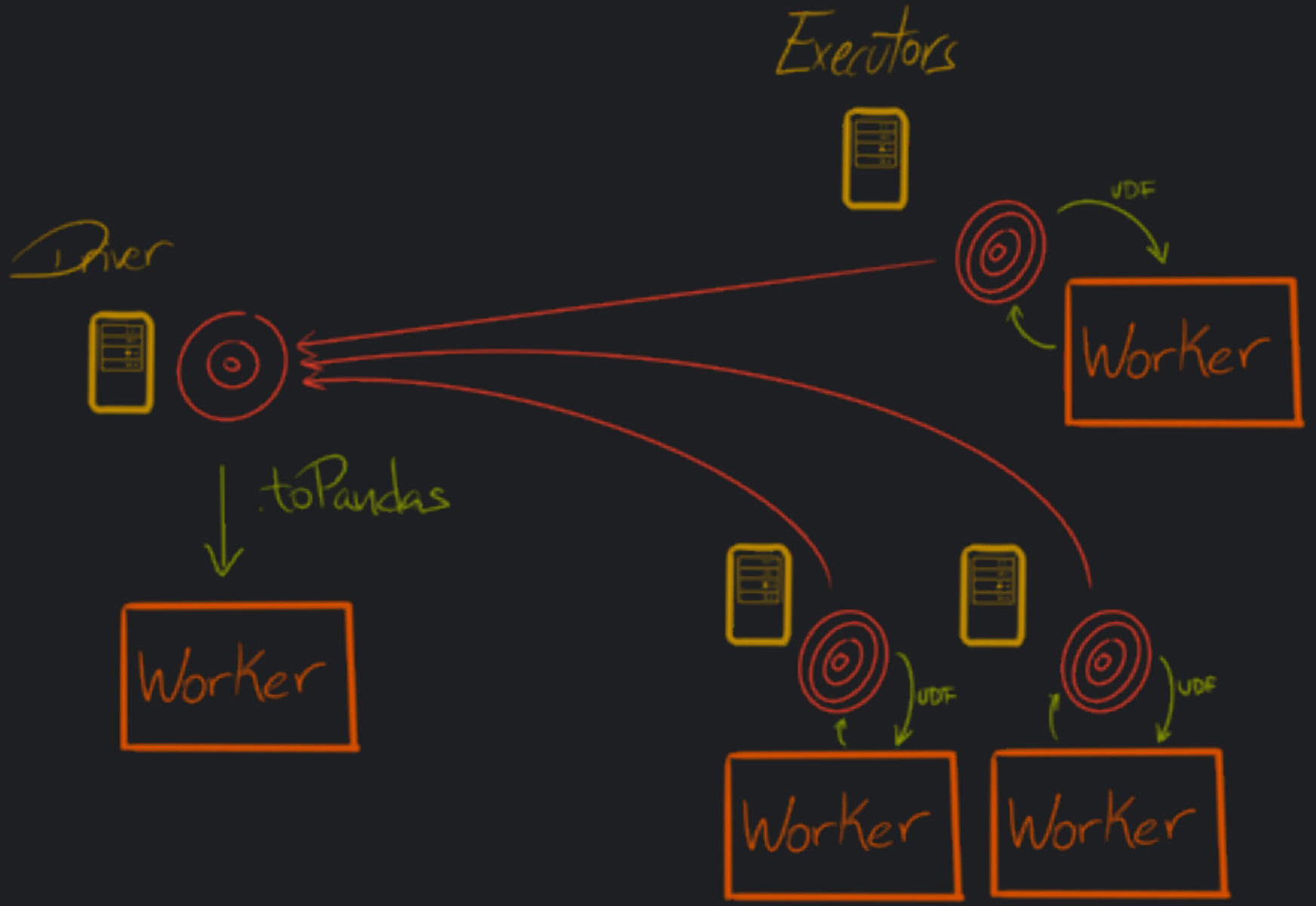
```
from pyspark.sql.functions import rand, randn, floor
from pyspark.sql.functions import pandas_udf, PandasUDFType

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()
```


Executors
Driver
group
aggregate


Executors
Driver
group
aggregate


Driver
group
aggregate

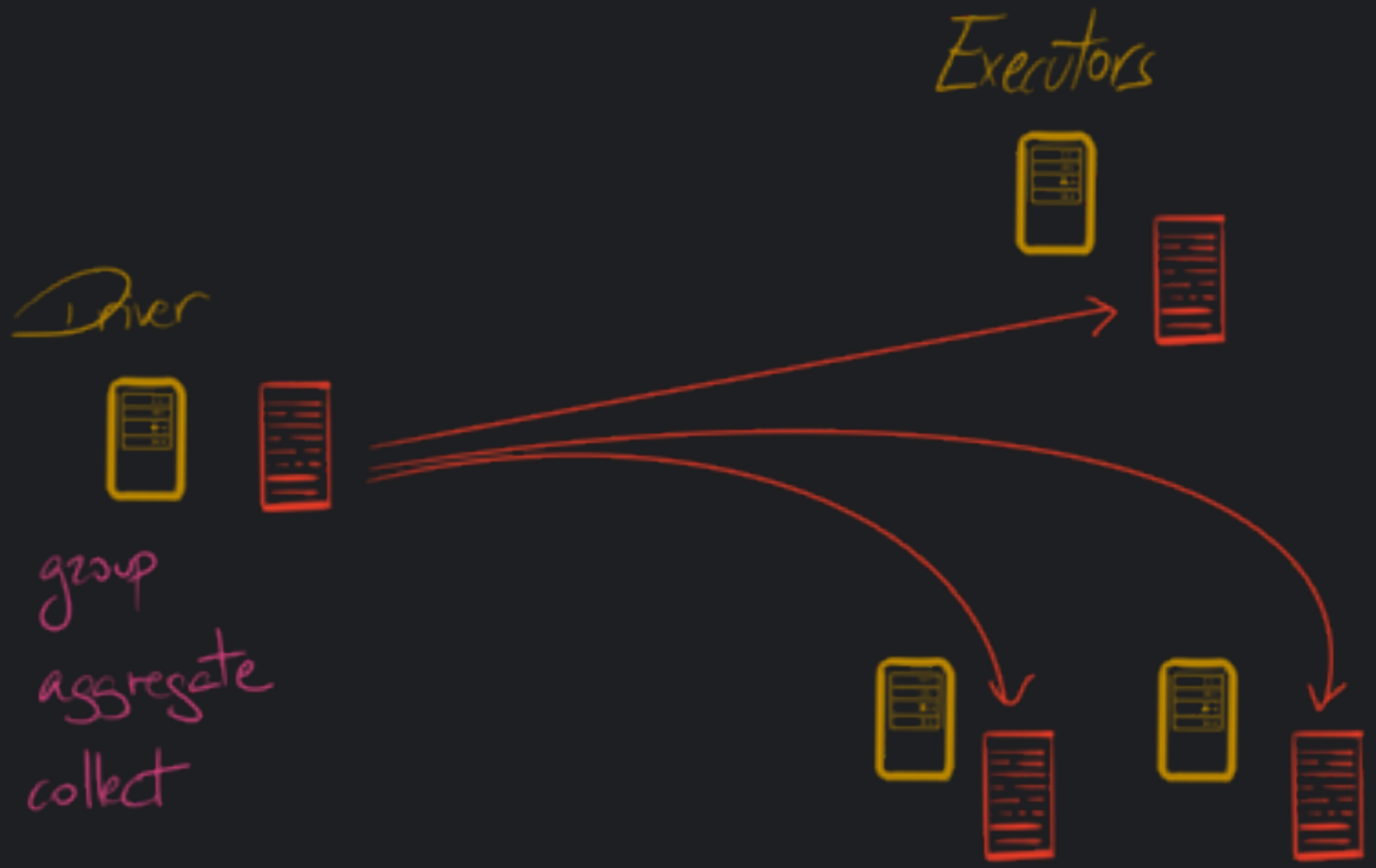

Executors
UDF
Worker
UDF
Worker
UDF
Worker

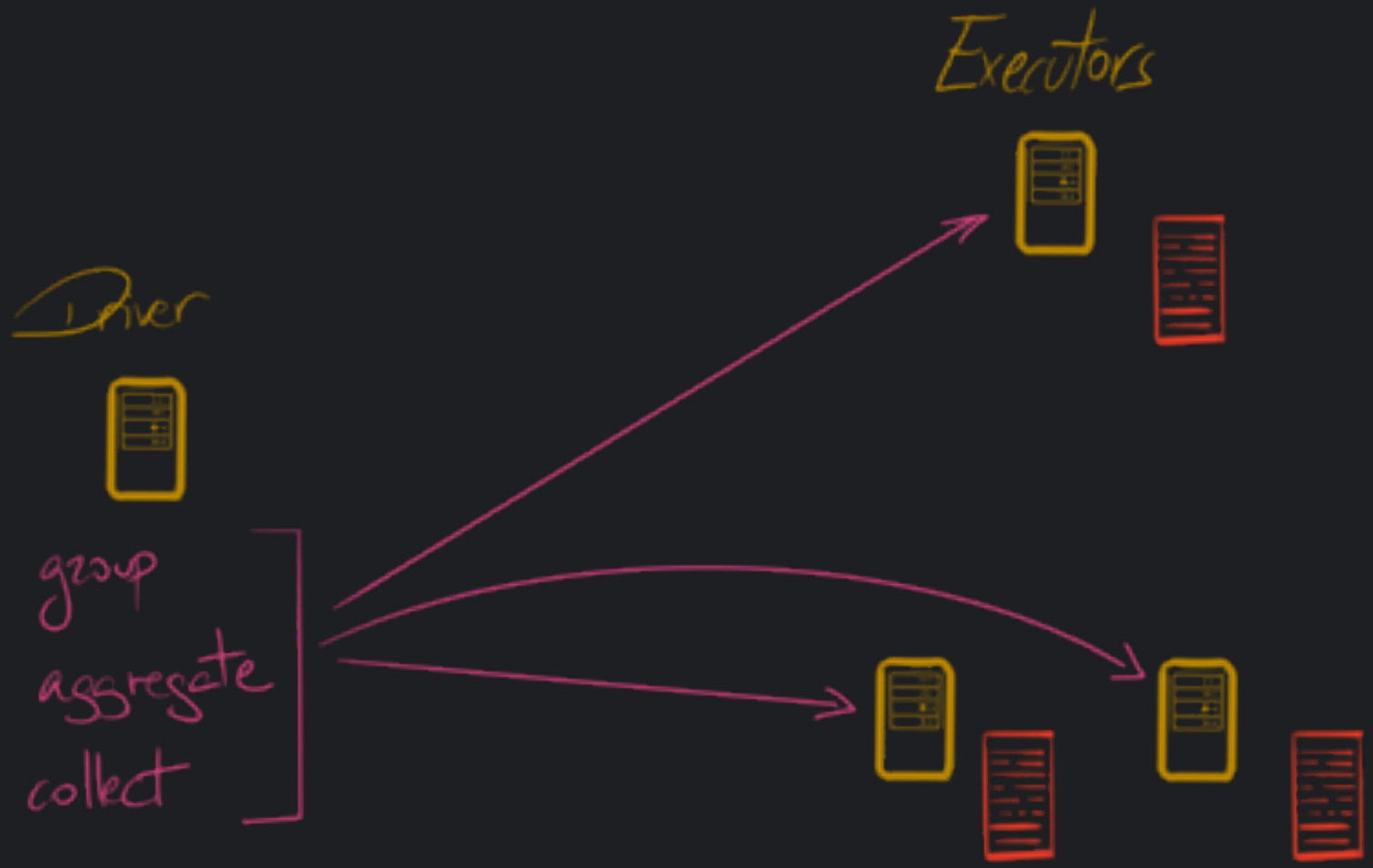

Executors
Driver
UDF
Worker
.toPandas
Worker
UDF
UDF
Worker
Worker

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")
```

```
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")
```


Executors
Driver
group
aggregate
collect


Executors
Driver
group
aggregate
collect

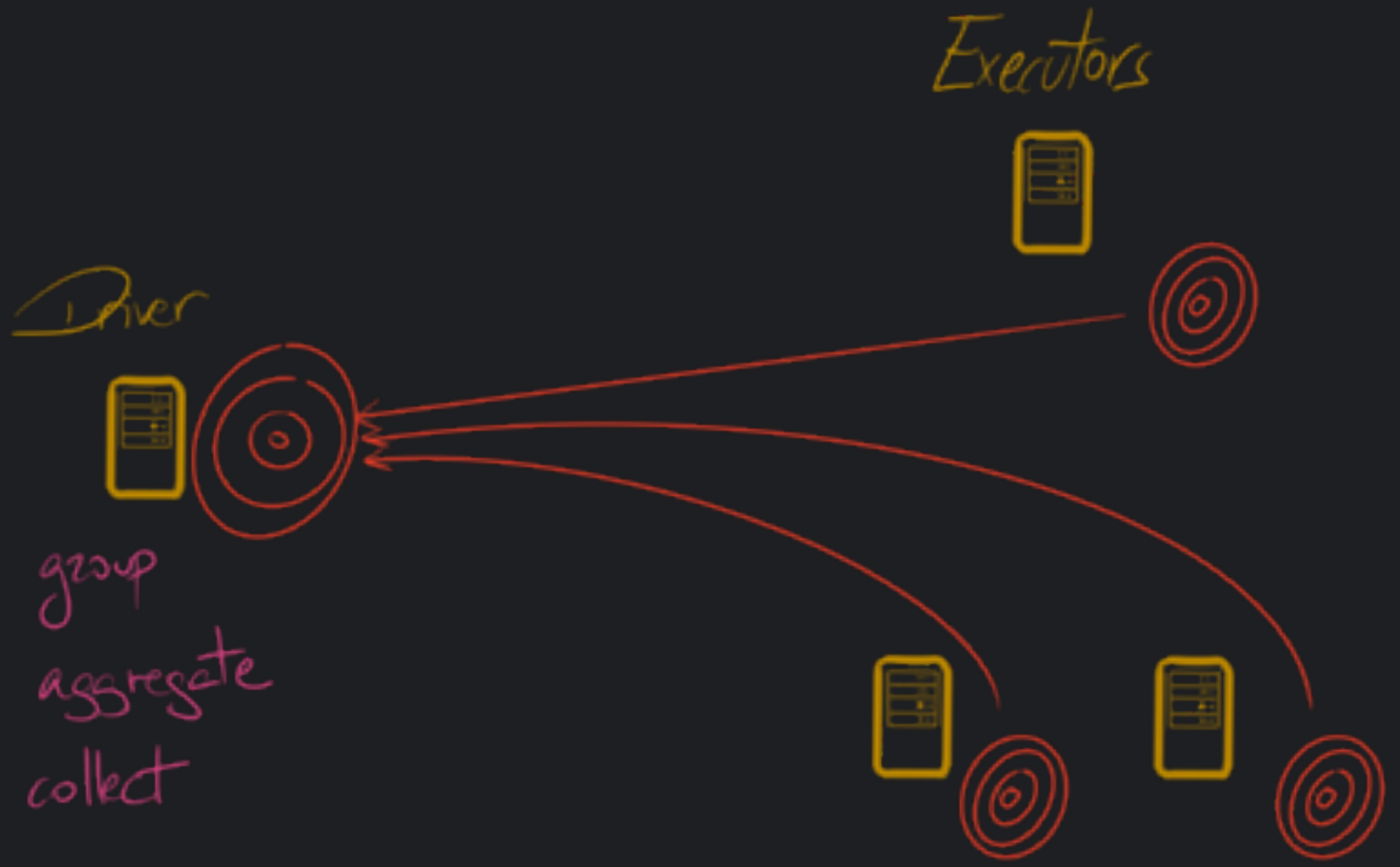
Executors

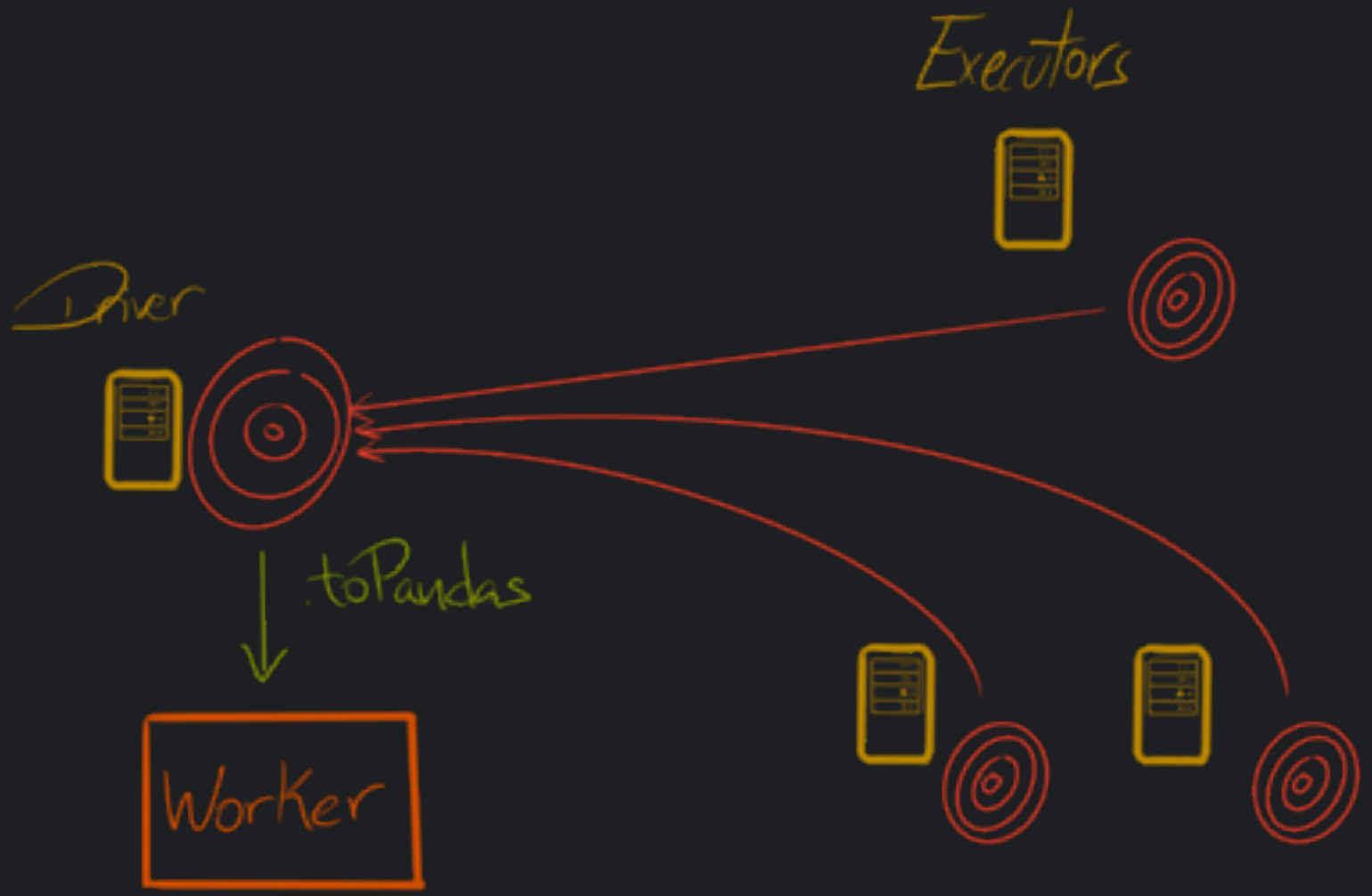
Driver

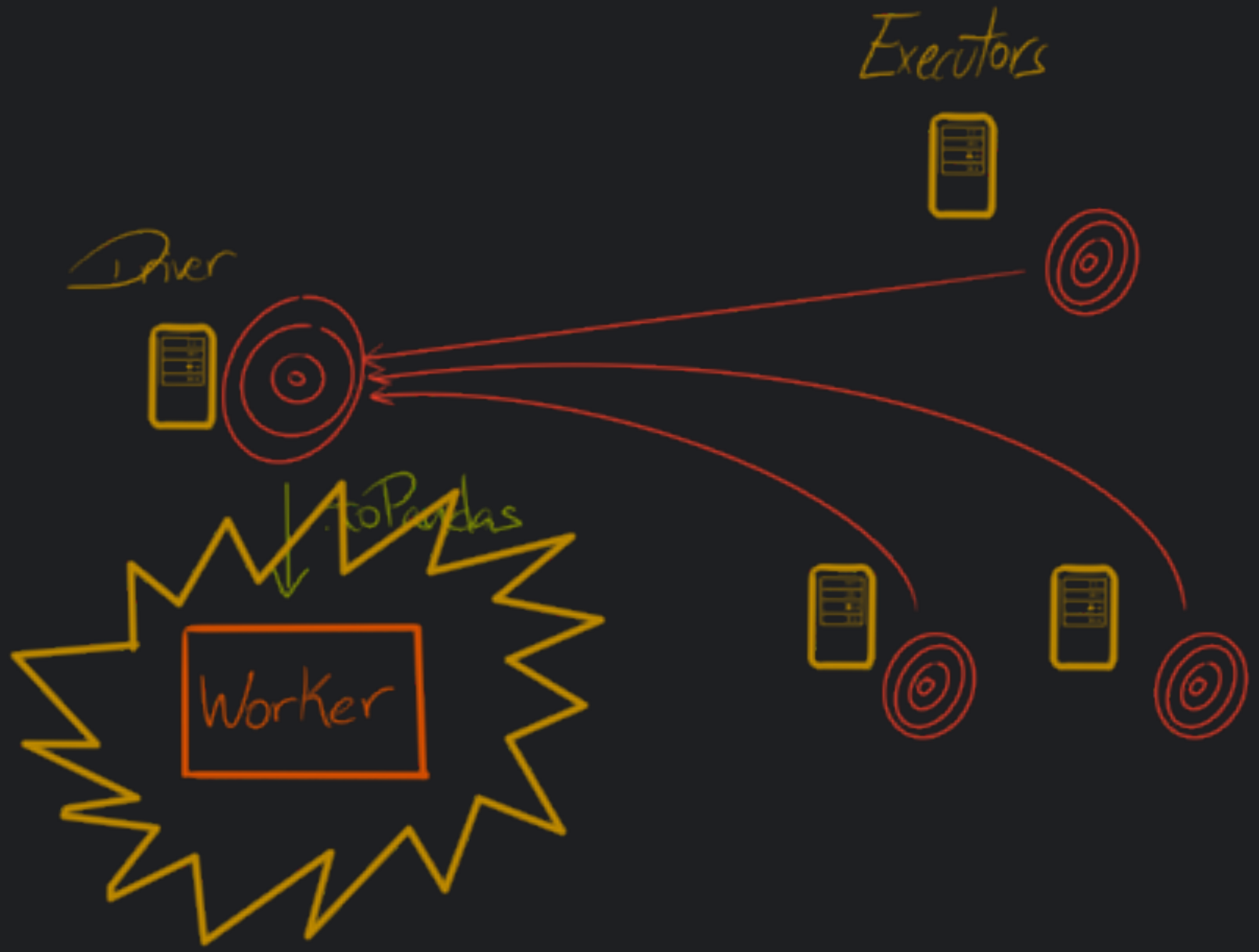
group

aggregate

collect


Executors
Driver
group
aggregate
collect


Executors
Driver
.toPandas
Worker


Executors
Driver
toPandas
Worker

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?

#UnifiedDataAnalytics #SparkAISummit

GET THE SLIDES FROM MY GITHUB:

github.com/rberenguel/

THE REPOSITORY IS

pyspark-arrow-pandas


SPARK+AI
SUMMIT 2019
Build.
Unify.
Scale.
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