Spark API

Tao Ruangyam, ING Analytics - Frankfurt Hub

Istanbul 2020

What will be covered in this session

- Intro to Spark API
- RDD
- DataFrame API
- Window functions
- Built-in functions
- UDFs

Introduction

col1	col2	col3
Row 1		
Row 2		
…		
Row N		

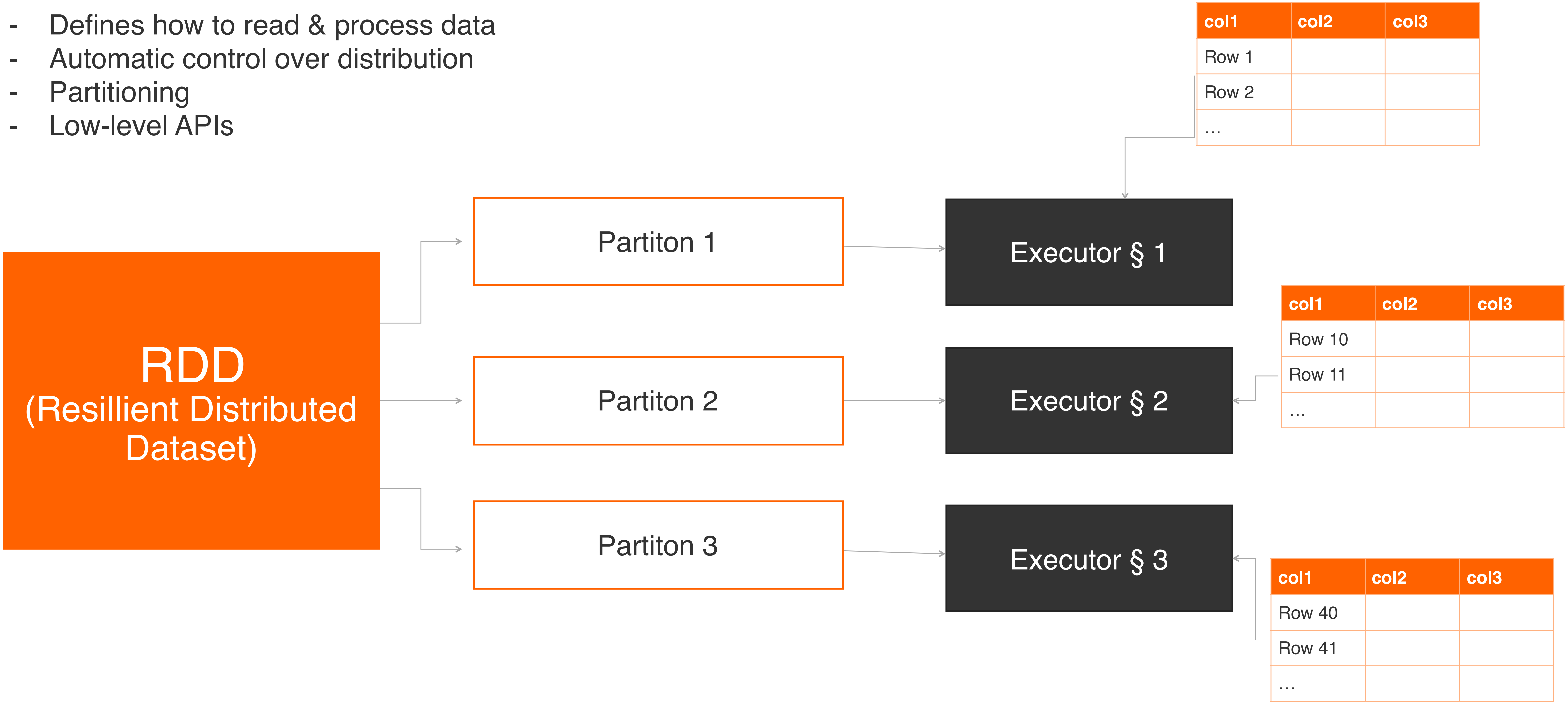
RDD
(Resillient Distributed Dataset)

DataFrame
(abstraction over RDD)

With Serialisation

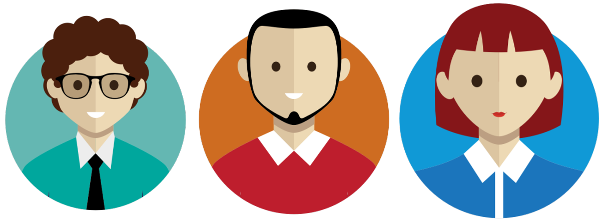
RDD

- Defines how to read & process data
- Automatic control over distribution
- Partitioning
- Low-level APIs



RDD

What coders see



API functions

- map()
- filter()
- cache()
- count()

- …

RDD
(Resillient Distributed Dataset)

What it tells Driver

- **How the data is created**
  - From beginning
  - Until the output
- **How the data is distributed**
  - How to partition
  - Which executor has it

RDD

Create an RDD

```
rdd = spark.sparkContext.parallelize([
    ('Red', 22, 3500),
    ('Green', 70, 5500),
    ('Blue', 15, 9500)
])

rdd = spark.sparkContext \
    .textFile('/path/to/file.csv') \
    .map(lambda line: line.split(','))
```

API overview

RDD
(Resillient Distributed Dataset)

DataFrame
(abstraction over RDD)

With Serialisation

DataSet
(Statically Typed)

With Serialisation

Supported by Python

Only Scala

DataFrame API

- Higher-level abstraction over **RDD**
- **No typesafe checking** at compilation (Dataset has)
- Fully compatible with **SparkSQL** API
- **Serialised!**

DataFrame API

Create a DataFrame

```
data = spark.read.csv('/path/to/file.csv')

data = spark.read.parquet('/path/to/file.parquet')

data = spark.sql('SELECT a,b,min(c) over (partition by a) FROM hivetable1')

data = spark.read \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "localhost:9092") \
        .option("subscribe", "topic1") \
        .load()
```

DataFrame API

Filter, add columns, do some aggregation

```python
from pyspark.sql.functions import *

result = data.filter((col('a') < 10) | (col('a').isNull()))

result = data.join(data2, "col-to-join", "inner")

result = data.withColumn("newCol", explode("array-column"))
```

Note that, Spark is lazy evaluated

Code will not be executed right away

```
from pyspark.sql.functions import *

result = data \
  .filter((col('a') < 10) | (col('a').isNull())) \
  .join(data2, "col-to-join", "inner")

exploded = result.withColumn("newCol", explode("array-column"))

..
..
..
..


exploded.show(25)

n = exploded.count()

exploded.write.save('path/to/file.parquet')
```

Not executed yet

Not executed yet

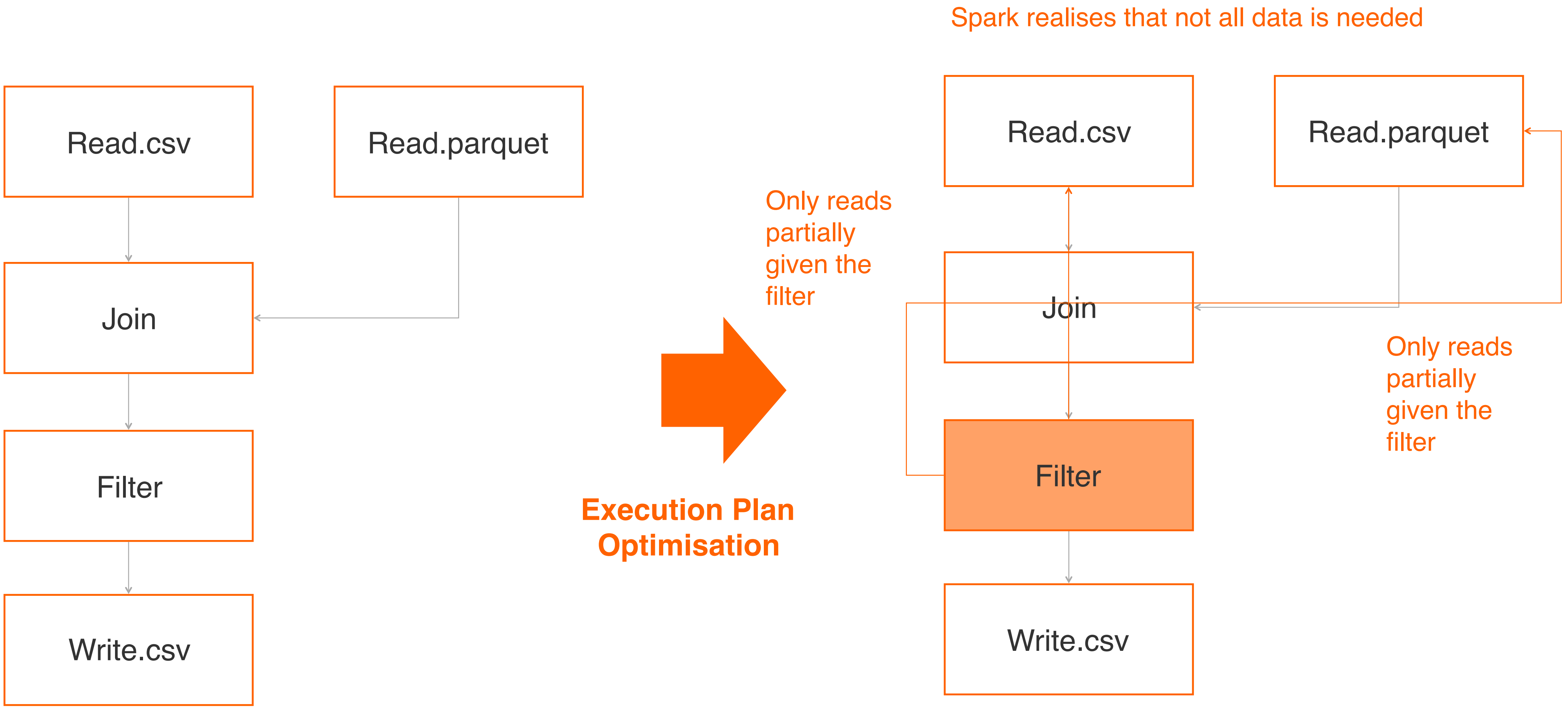
Executed!

Executed!

Executed!

Why lazy? Why not actively evaluate?

Process can be more optimisable with lazy evaluation

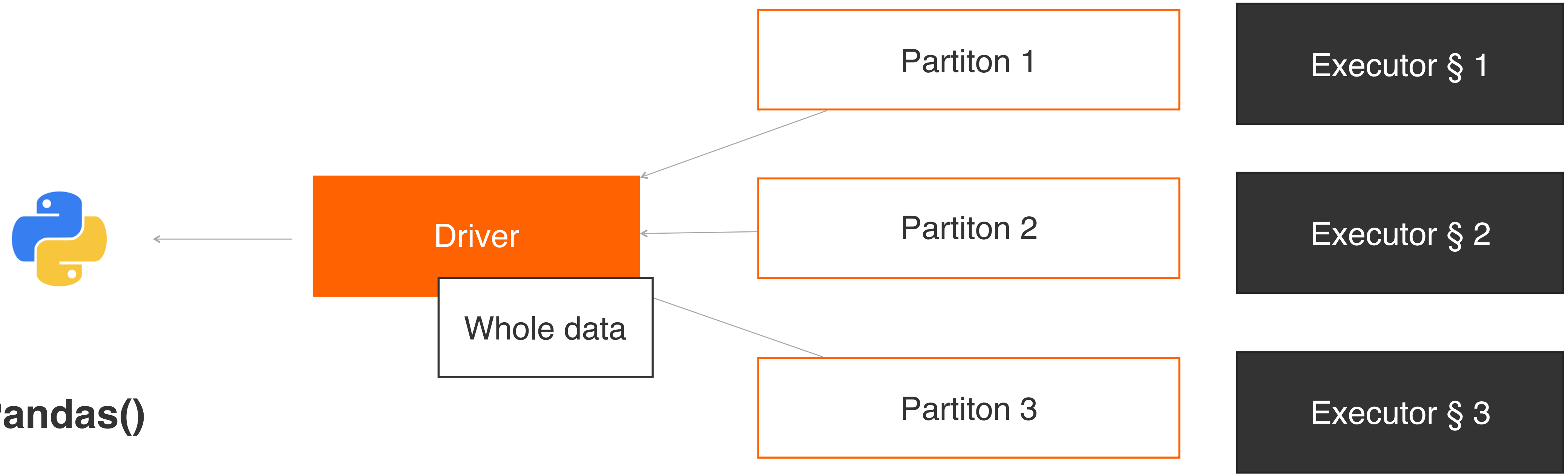


RDD API is accessible from DataFrame

```
df = spark.read.parquet('path/to/file.parquet')

out = df.rdd.map(lambda row: "Features: {}".format(row.feature))

kv = df.rdd.keyBy(lambda row: row.a)
```

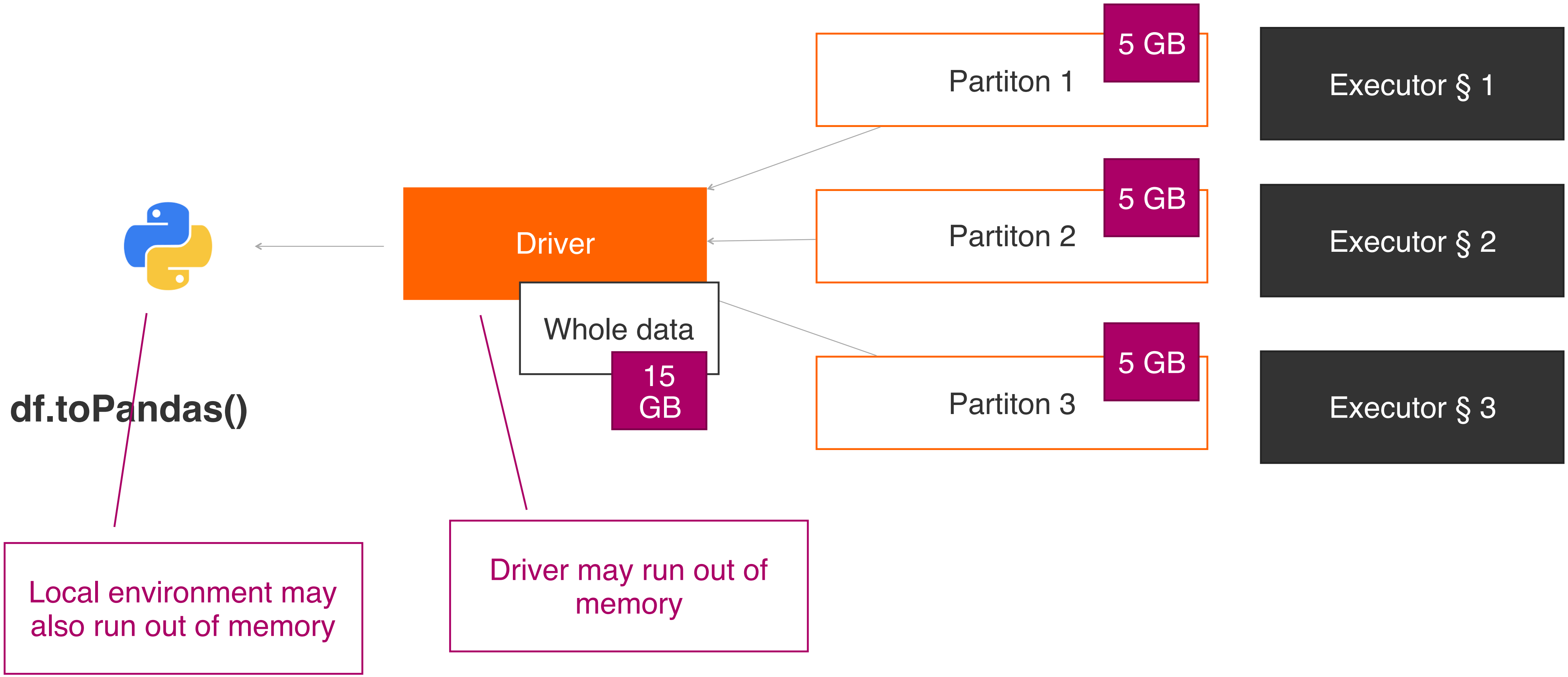
Since DataFrame is just an abstraction over RDD,
DF.rdd exposes the RDD inside

Ways to transfer data to Pandas



df.toPandas()

But this can cause out-of-memory



Pandas to Spark

```
import numpy as np
import pandas as pd

# Enable Arrow-based columnar data transfers
spark.conf.set("spark.sql.execution.arrow.enabled", "true")

# Generate a pandas DataFrame
pdf = pd.DataFrame(np.random.rand(100, 3))

# Create a Spark DataFrame from a pandas DataFrame using Arrow
df = spark.createDataFrame(pdf)
```

Arrow will transfer local Pandas dataframe to the Spark instance

Pandas to Spark

Pandas DF

Driver

Whole data

Partiton 1

Partiton 2

Partiton 3

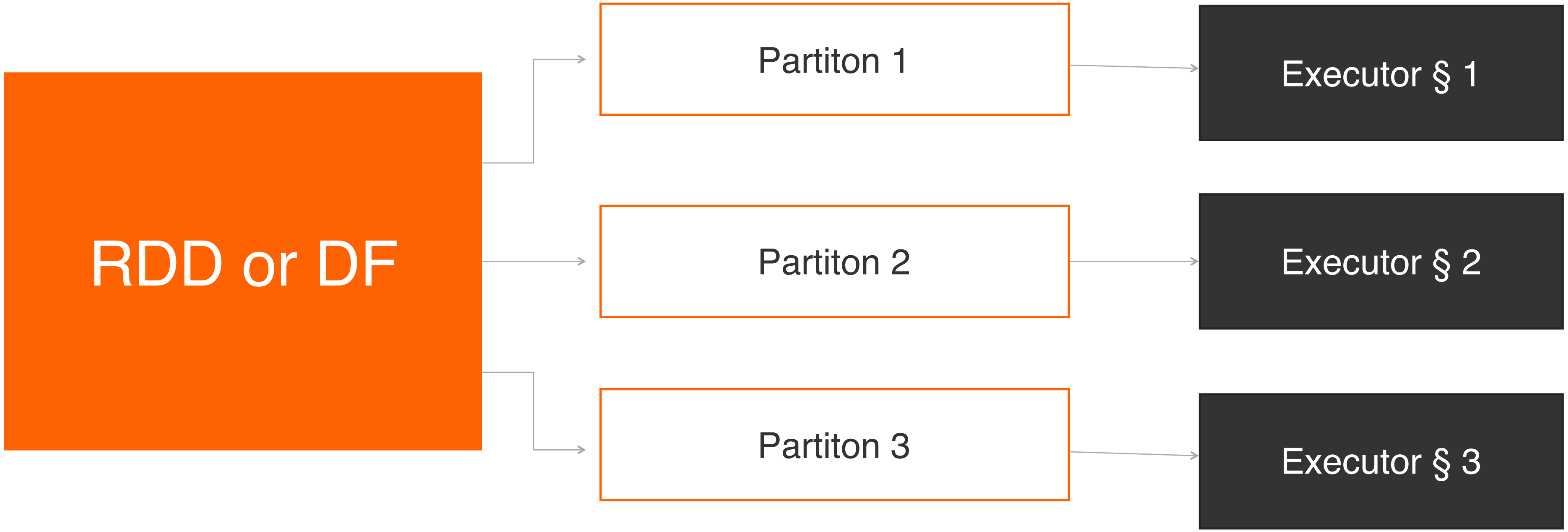
Executor § 1

Executor § 2

Executor § 3

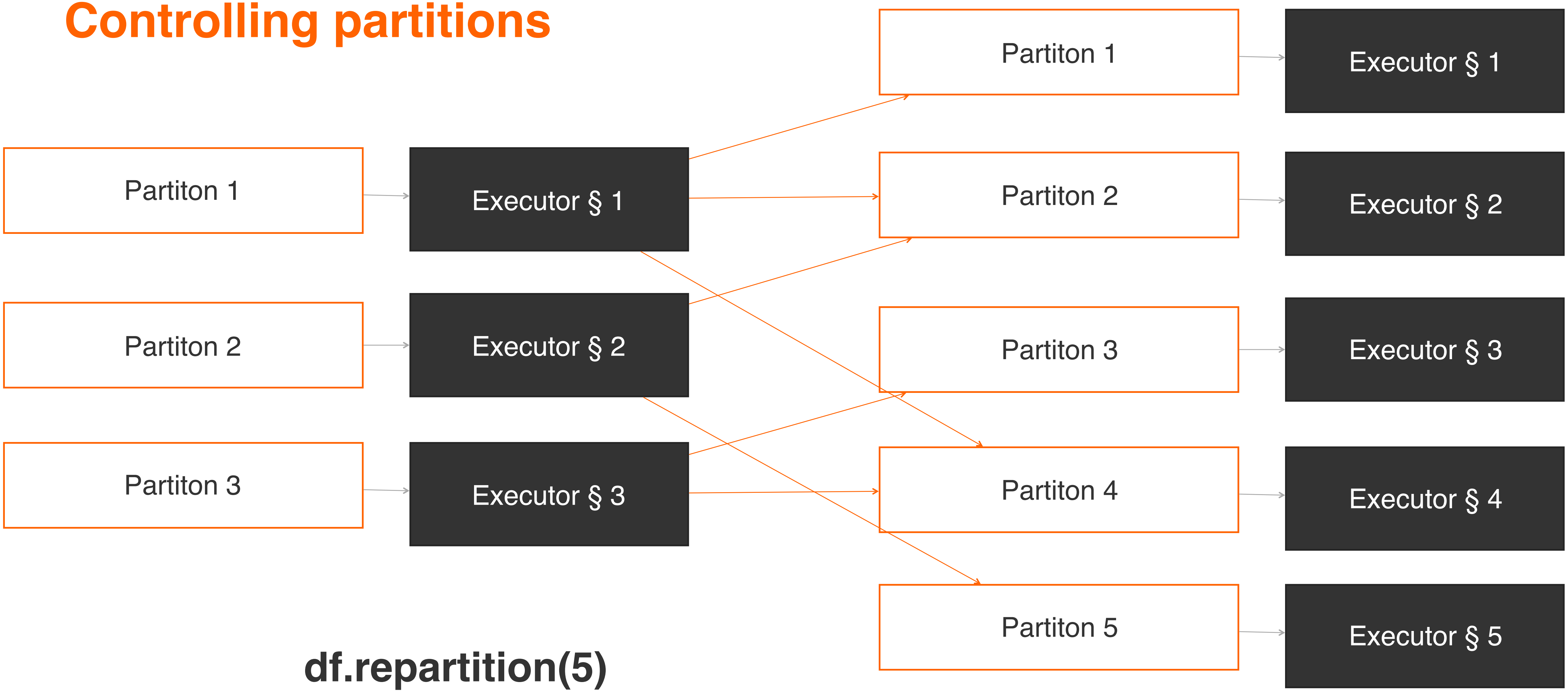
spark.createDataFrame(pandaDF)

Controlling partitions



spark.default.parallelism (number of partitions after join or reduceByKey)

Controlling partitions

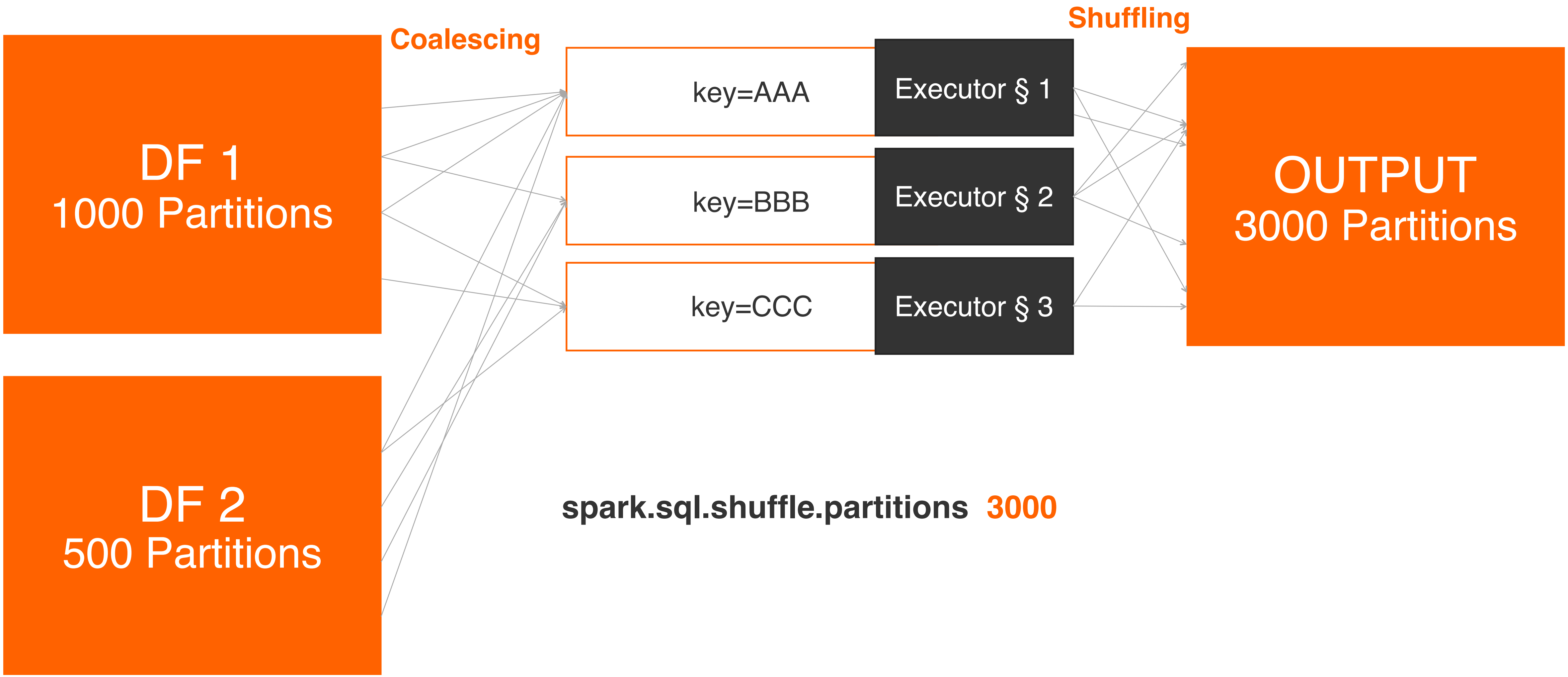


df.repartition(5)

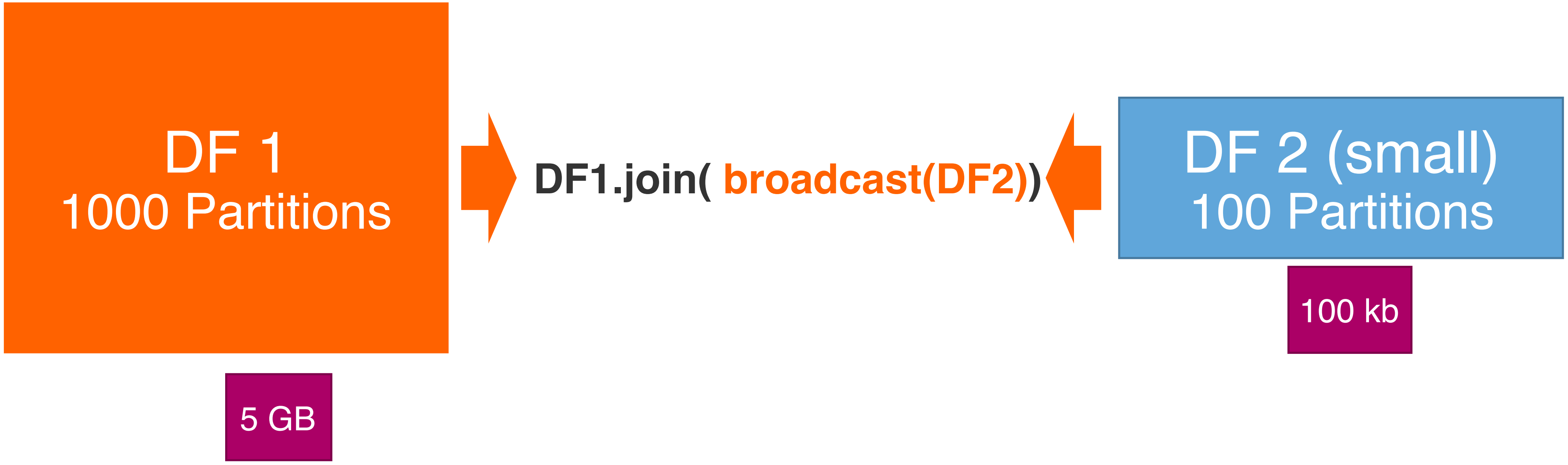
HashJoin

Regardless of prior number of partitions,
Spark **reshuffles** them by **joining keys**

HashJoin

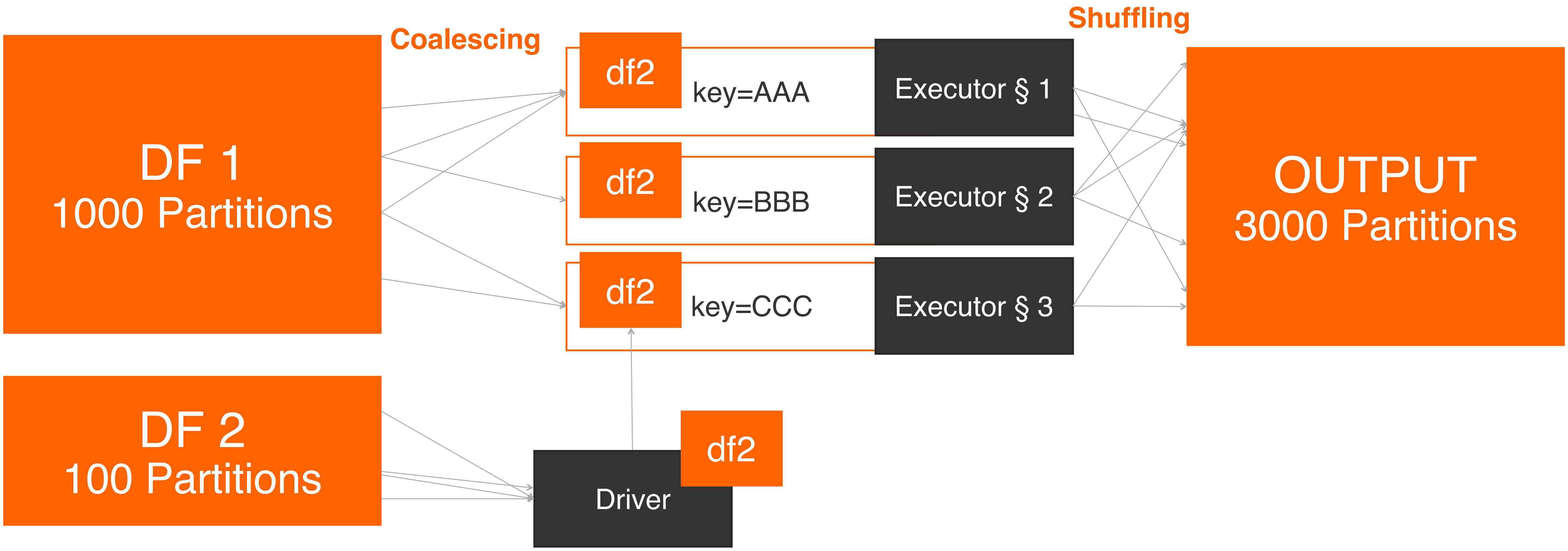

Coalescing
Shuffling
DF 1
1000 Partitions
DF 2
500 Partitions
key=AAA
key=BBB
key=CCC
Executor § 1
Executor § 2
Executor § 3
OUTPUT
3000 Partitions
spark.sql.shuffle.partitions 3000

BroadcastHashJoin



Still works the same way with **HashJoin**
but all executors **get the whole broadcasted** dataframe

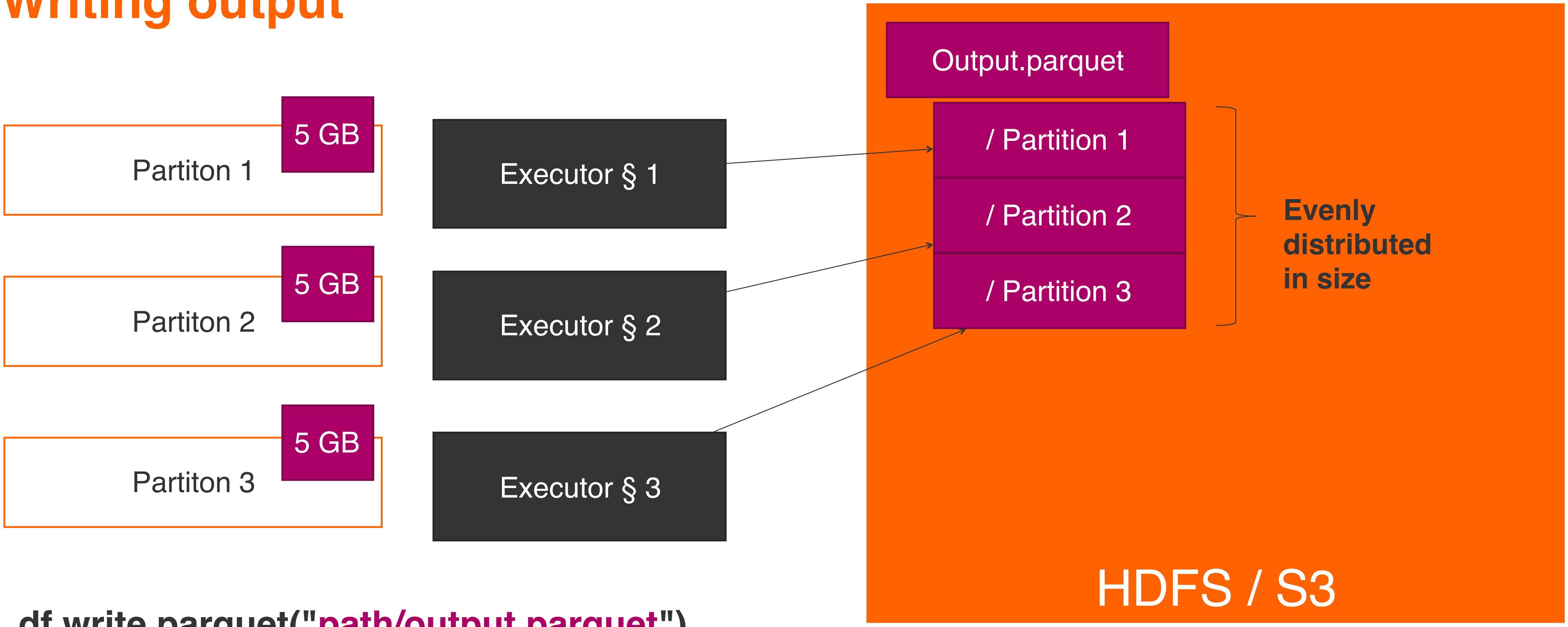
HashJoin


Coalescing
Shuffling
DF 1
1000 Partitions
DF 2
100 Partitions
df2
key=AAA
df2
key=BBB
df2
key=CCC
Executor § 1
Executor § 2
Executor § 3
OUTPUT
3000 Partitions
Driver
df2

Writing output

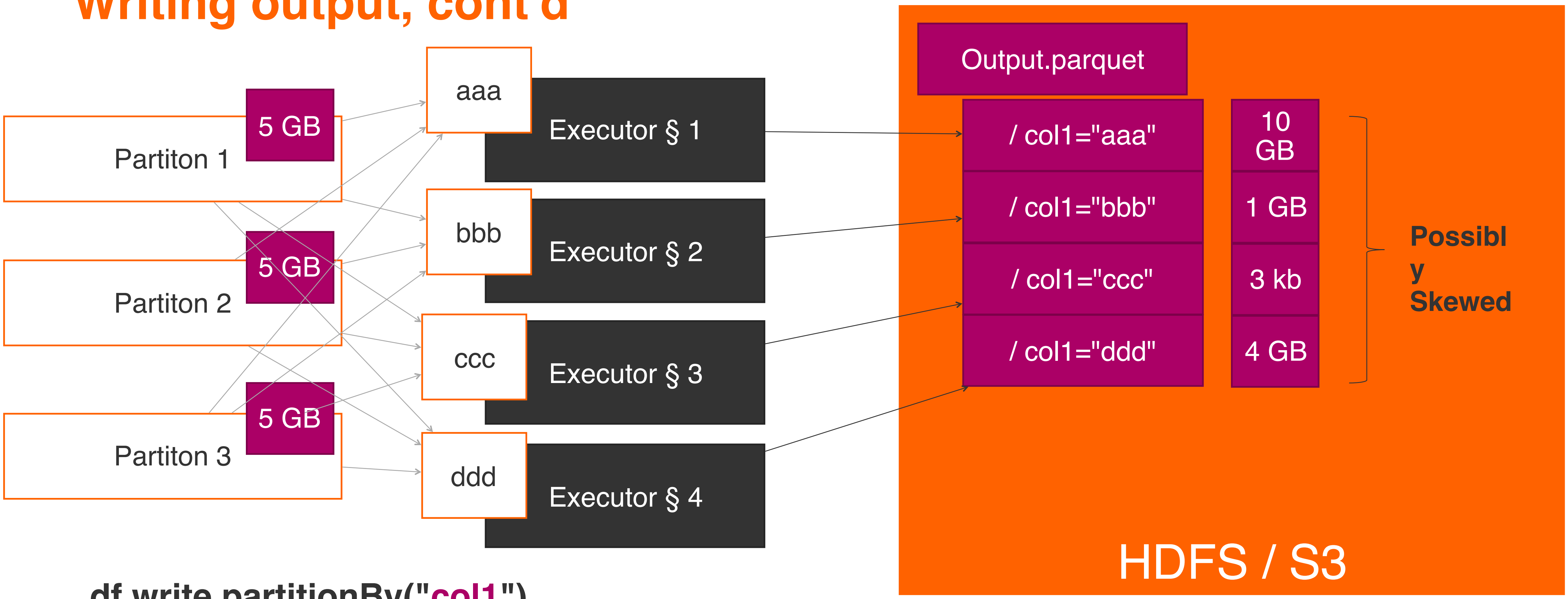
```
df.write.option("header","true").csv("path/file.csv")

df.write.partitonBy("col1").parquet("path/file.parquet")

df.write.mode("append").parquet("path/file.parquet")

df.write.json("path/file.json")

df.write.saveAsTable("schema.hive_table_name")
```

Writing output



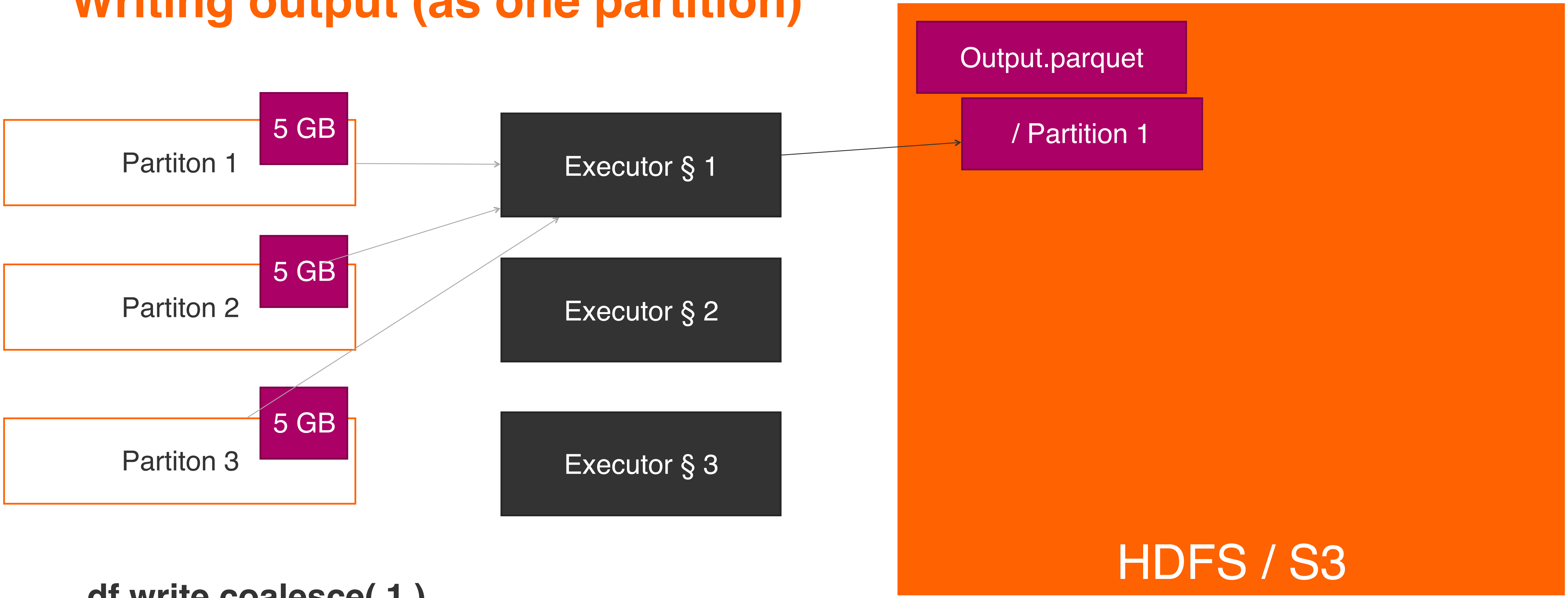
df.write.parquet("path/output.parquet")

Writing output, cont'd




```
df.write.partitionBy("col1")
        .parquet("path/output.parquet")
```

Writing output (as one partition)

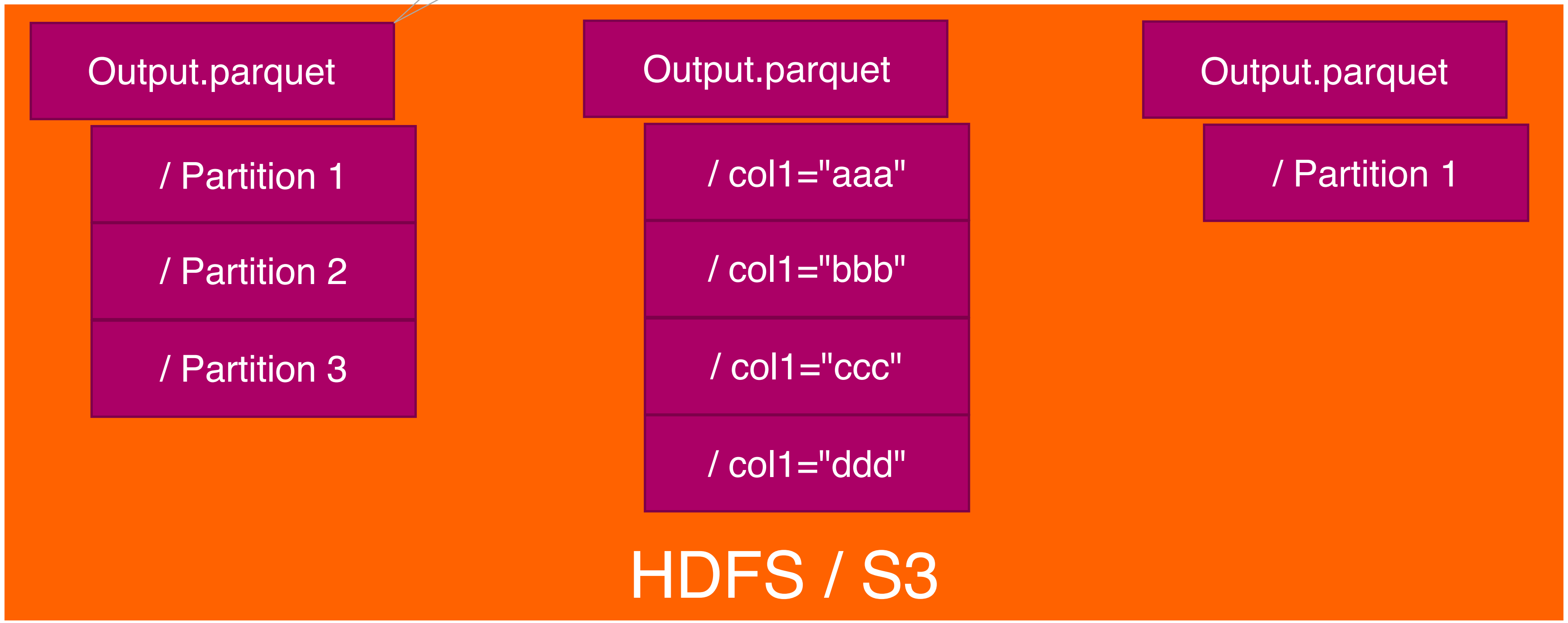



```
df.write.coalesce( 1 )
        .parquet("path/output.parquet")
```

Read the partitioned data

```
df.read.parquet("path/Output.parquet")
```

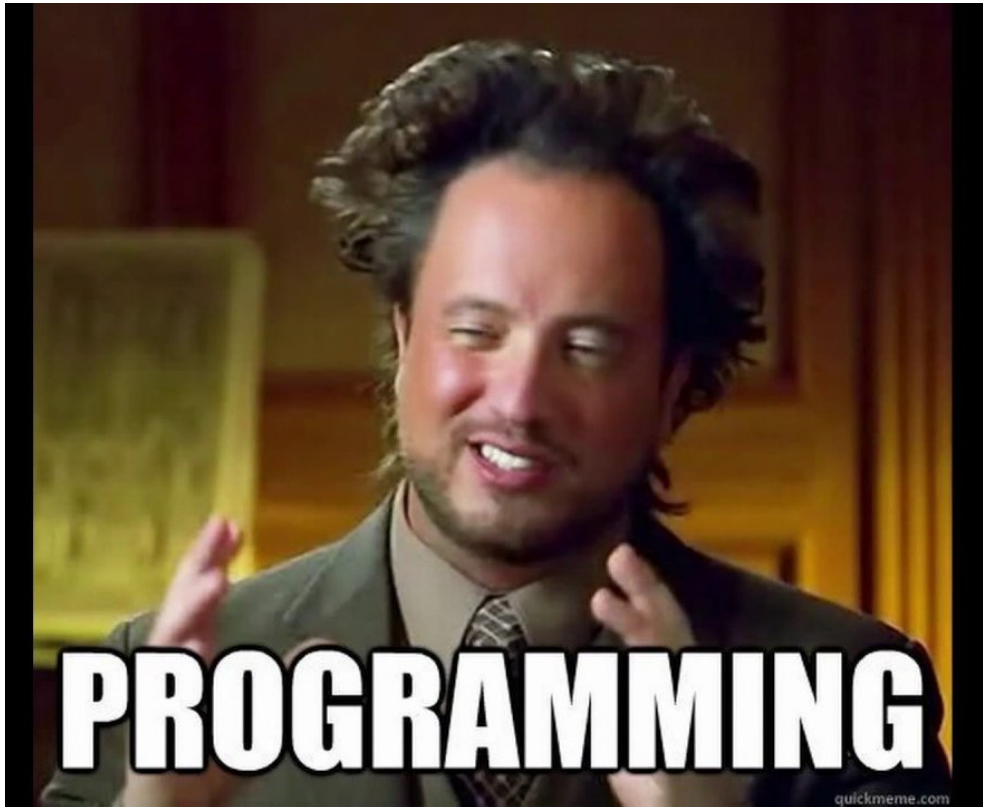
Spark always overrides number of partitions



Choosing the right data types

	Parquet	CSV	JSON	Avro
Optimised for	File size, SELECT	Readability	Portability	Schema changes
Read speed	**Fastest**	Slow	**Slowest**	Fast
Write speed	Slow	**Fastest**	**Slowest**	Fast
Partitonable	Yes	Yes	**No**	Yes
File size	**Smallest**	Large	**Largest**	Small
Spark needs to read the whole file	**No**	**Yes**	**Yes**	**No**

Let's try out the API



https://github.com/tao-pr/spark-workshop

Built-in Functions

Column type

```
col("name")
lit(value)
```

Functions
min()
explode()
rank()
…
collect_set()
udf()

Column type

Built-in Functions

cover	num
Red,Blue,Black	5
Red,Green	10
Green,White	15
Yellow	30

cover (str)	num	colours (Array[str])
Red,Blue,Black	5	[Red,Blue,Black]
Red,Green	10	[Red,Green]
Green,White	15	[Green,White]
Yellow	30	[Yellow]

```
dfOut = df.withColumn("colours", split(col("cover"), lit("\\,")) )
```

Built-in Functions

product	loss
light	3.0
light	4.0
fridge	8.5
fridge	2.5
fridge	1.5
powersupply	0.1

→

	double	double
product	**loss**	**logloss**
light	3.0	1.09861229
light	4.0	1.38629436
fridge	8.5	2.14006616
fridge	2.5	0.91629073
fridge	1.5	0.40546511
powersupply	0.1	-2.30258509

```
dfOut = df.withColumn("logloss", log(col("loss")) )
```

Built-in Functions

product	loss
light	3.0
light	4.0
fridge	8.5
fridge	2.5
fridge	1.5
powersupply	0.1

product	loss	logloss	minlogloss
light	3.0	1.09861229	-2.30258509
light	4.0	1.38629436	-2.30258509
fridge	8.5	2.14006616	-2.30258509
fridge	2.5	0.91629073	-2.30258509
fridge	1.5	0.40546511	-2.30258509
powersupply	0.1	-2.30258509	-2.30258509

```
dfOut = df.withColumn("logloss", log(col("loss")) )
          .withColumn("minlogloss", min(col("logloss")) )
```

Window functions!

product	loss
light	3.0
light	4.0
fridge	8.5
fridge	2.5
fridge	1.5
powersupply	0.1

product	loss	logloss	minlogloss
light	3.0	1.09861229	1.09861229
light	4.0	1.38629436	1.09861229
fridge	8.5	2.14006616	0.40546511
fridge	2.5	0.91629073	0.40546511
fridge	1.5	0.40546511	0.40546511
powersupply	0.1	-2.30258509	-2.30258509

```
from pyspark.sql.window import Window

w     = Window.partitionBy("product")
dfOut = df.withColumn("logloss", log(col("loss")) )
          .withColumn("minlogloss", min(col("logloss")).over(w) )
```


ING

Window functions!

product	loss
light	3.0
light	4.0
fridge	8.5
fridge	2.5
fridge	1.5
powersupply	0.1

product	loss	logloss	rank
light	3.0	1.09861229	1
light	4.0	1.38629436	2
fridge	8.5	2.14006616	3
fridge	2.5	0.91629073	2
fridge	1.5	0.40546511	1
powersupply	0.1	-2.30258509	1

```
from pyspark.sql.window import Window

w     = Window.partitionBy("product").orderBy("loss")
dfOut = df.withColumn("logloss", log(col("loss")) )
          .withColumn("rank", rank(col("logloss")).over(w) )
```

Explode array column

product	prices
A	[15,25,35]
B	[]
C	[20]
D	[1,3,5,6]

	Array [double]	double
product	prices	price
A	[15,25,35]	15
A	[15,25,35]	25
A	[15,25,35]	35
C	[20]	20
D	[1,3,5,6]	1
D	[1,3,5,6]	3
D	[1,3,5,6]	5
D	[1,3,5,6]	6

```
dfOut = df.withColumn("price", explode(col("prices")) )
```

Conditions

product	prices
A	[15,25,35]
B	[]
C	[20]
D	[1,3,5,6]

	double	str
product	price	
A	15	A or B
A	25	A or B
A	35	A or B
B	Null	A or B
C	20	
D	1	
D	3	
D	5	
D	6	

```
dfOut = df.select(col("product"),
                  explode(col("prices")).alias("price"),
                  when(col("product").isin(["A","B"]), lit("A or
                  B")).otherwise(lit("")))
```

Using built-in funcitons anywhere

+	Example
Select	**df.select**("TARGET", **explode**(**col**("a")))
WithColumn	**df.withColumn**("TARGET", **explode**(**col**("a")))
Filter	**df.filter**(**abs**(**col**("v") > 50)
Aggregation	**df.groupBy**(**floor**(**col**("v")).**agg**(**sum**("w"))

UDF (User-defined function)

Make a function distributable across nodes

```
myfunc = udf( func )
```

```
.withColumn("a", myfunc(….))

.select(myfunc(….))
```

```
def func(arg1, arg2)

func = lambda arg1, arg2 : ….
```

Spark SQL

Tao Ruangyam, ING Analytics - Frankfurt Hub

Istanbul 2020

Equivalent DataFrame functions on SQL

Interchangeability between **DataFrameAPI** and **SQL command**

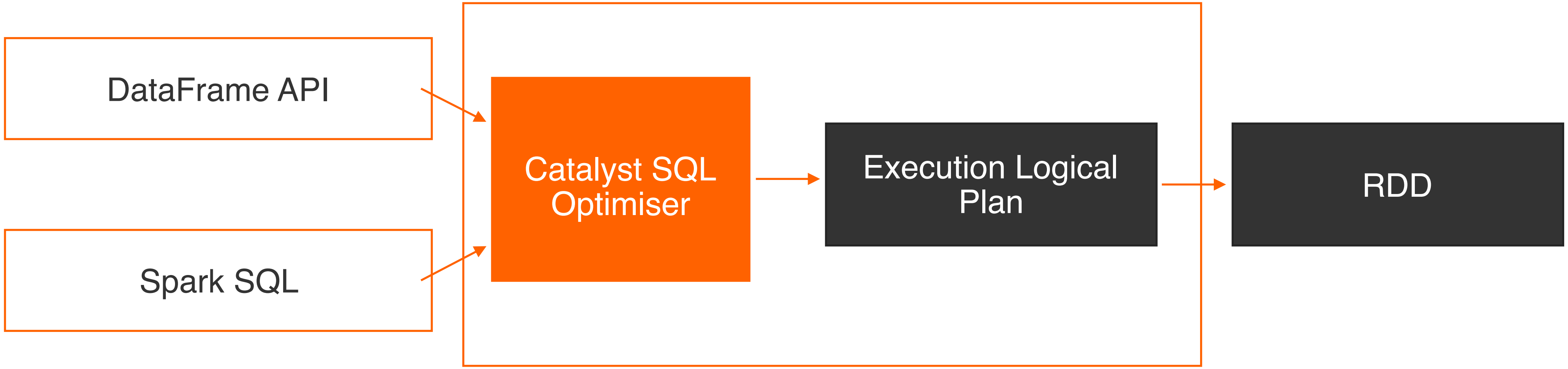
```
df.where(col("price")<3000)\
  .groupBy("grade").agg(
    count(lit(1)).alias("units"),
    avg("size").alias("avgsize")
  ).show(3)

+-----+-----+-----------------+
|grade|units|          avgsize|
+-----+-----+-----------------+
|    B|  113|85.77658681742913|
|    C|  122|83.27286338415303|
|    A|   16| 98.3440670967102|
+-----+-----+-----------------+

spark.sql("""SELECT grade, sum(1) as units, avg(size) as size
    FROM inventory WHERE price<3000
    GROUP BY grade"""
    ).show(3)
```

DataFrame API (SparkSQL)

Equivalent SQL command

Relation between DataFrame API and SQL engine

Both are compiled into the same form: **Physical execution plan**



Typically users do not have to worry about this box

But the logical plan can also be customisable (Scala)

Instead of writing DataFrame API, we can write SQL

Both **DataFrame API** and **SQL** are compiled into **Physical execution plan**

```
>>> spark.sql("SELECT type, min(price) as low, max(price) as high FROM inventory GROUP BY type").explain()

== Physical Plan ==
*(6) HashAggregate(keys=[type#1667], functions=[min(price#1710), max(price#1710)])
+- Exchange hashpartitioning(type#1667, 200)
   +- *(5) HashAggregate(keys=[type#1667], functions=[partial_min(price#1710), partial_max(price#1710)])
      +- *(5) Project [type#1667, price#1710]
         +- *(5) Sort [_nondeterministic#1774 ASC NULLS FIRST], true, 0
            +- Exchange rangepartitioning(_nondeterministic#1774 ASC NULLS FIRST, 200)
               +- *(4) Project [type#1667, pythonUDF0#5951 AS price#1710, rand(8934628709428066776) AS _nonde
                  +- BatchEvalPython [<lambda>(avgprice#1670L, stdprice#1671L)], [avgprice#1670L, stdprice#16
                     +- *(3) Project [avgprice#1670L, stdprice#1671L, type#1667]
                        +- Generate explode(size#1686), [type#1667, avgprice#1670L, stdprice#1671L], false, [
                           +- *(2) Project [type#1667, avgprice#1670L, stdprice#1671L, pythonUDF0#5950 AS siz
                              +- BatchEvalPython [<lambda>(avgsize#1672L, stdsize#1673L, qty#1669L)], [avgpri
                                 +- *(1) Project [avgprice#1670L, avgsize#1672L, qty#1669L, stdprice#1671L, s
                                    +- Scan ExistingRDD[type#1667,grade#1668,qty#1669L,avgprice#1670L,stdpric
```

What is SparkSQL?

Both **DataFrame API** and **SQL** are compiled into **Physical execution plan**

```
>>> df.groupBy("type").agg(min("price").alias("low"), max("price").alias("high")).explain()

== Physical Plan ==
*(6) HashAggregate(keys=[type#1667], functions=[min(price#1710), max(price#1710)])
+- Exchange hashpartitioning(type#1667, 200)
   +- *(5) HashAggregate(keys=[type#1667], functions=[partial_min(price#1710), partial_max(price#1710)])
      +- *(5) Project [type#1667, price#1710]
         +- *(5) Sort [_nondeterministic#1774 ASC NULLS FIRST], true, 0
            +- Exchange rangepartitioning(_nondeterministic#1774 ASC NULLS FIRST, 200)
               +- *(4) Project [type#1667, pythonUDF0#5971 AS price#1710, rand(8934628709428066776) AS _nonde
                  +- BatchEvalPython [<lambda>(avgprice#1670L, stdprice#1671L)], [avgprice#1670L, stdprice#16
                     +- *(3) Project [avgprice#1670L, stdprice#1671L, type#1667]
                        +- Generate explode(size#1686), [type#1667, avgprice#1670L, stdprice#1671L], false,
                           +- *(2) Project [type#1667, avgprice#1670L, stdprice#1671L, pythonUDF0#5970 AS si
                              +- BatchEvalPython [<lambda>(avgsize#1672L, stdsize#1673L, qty#1669L)], [avgpri
                                 +- *(1) Project [avgprice#1670L, avgsize#1672L, qty#1669L, stdprice#1671L, s
                                    +- Scan ExistingRDD[type#1667,grade#1668,qty#1669L,avgprice#1670L,stdpric
```


ING

Built-in functions in SQL

All Spark built-in functions can also work with SQL!

```
spark.sql("SELECT a,min(b),collect_set(c) FROM table WHERE c>0 GROUP BY a")

df.where(col("c")>0).groupBy("a").agg(min(col("b")), collect_set(col("c")))
```

Built-in functions in SQL

SQL trick with **explode**

```
+-----+------------+
|grade|         sub|
+-----+------------+
|    A|[A1, A2, A3]|
|    B|    [B1, B2]|
|    C|    [C1, C3]|
|    D|          []|
+-----+------------+


>>> spark.sql("select grade, explode(sub) from vec").show(10)
+-----+---+
|grade|col|
+-----+---+
|    A| A1|
|    A| A2|
|    A| A3|
|    B| B1|
|    B| B2|
|    C| C1|
|    C| C3|
+-----+---+
```

Exploded array will **vanish** because explode does not reserve null

Built-in functions in SQL

```
+-----+------------+
|grade|         sub|
+-----+------------+
|    A|[A1, A2, A3]|
|    B|    [B1, B2]|
|    C|    [C1, C3]|
|    D|          []|
+-----+------------+


>>> spark.sql("select grade, s from vec lateral view outer explode(sub) as s").show(10)
+-----+----+
|grade|   s|
+-----+----+
|    A|  A1|
|    A|  A2|
|    A|  A3|
|    B|  B1|
|    B|  B2|
|    C|  C1|
|    C|  C3|
|    D|null|
+-----+----+
```

Reserve null with **lateral view outer**

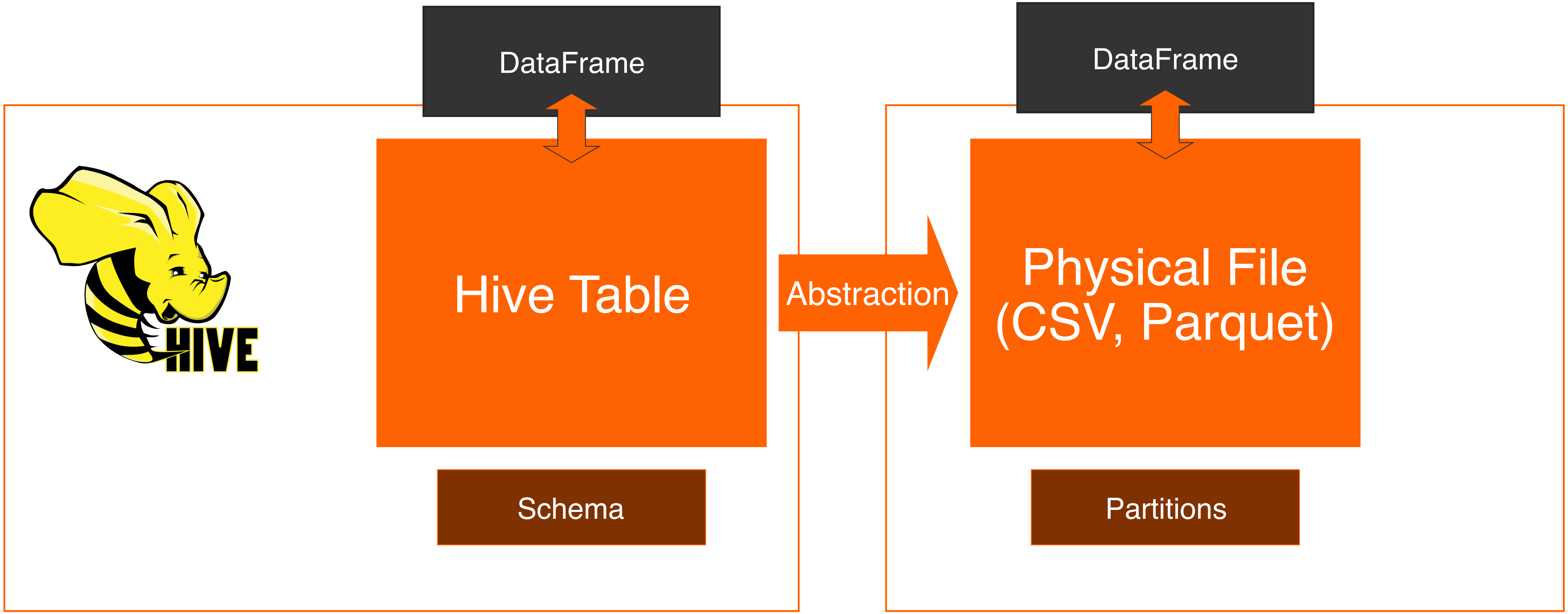
More complex SQL is also supported

```
+---------+-----+------+---------+---------+---------+---------+
|     type|grade|seller|    price|     size|      lat|      lng|
+---------+-----+------+---------+---------+---------+---------+
|Apartment|    A|    24|4262.7754|92.717896|53.396652|35.058865|
|Apartment|    B|    29| 2935.234| 71.07544|  56.5649|41.392433|
|       WG|    B|     6|2136.9932|111.69808| 24.81349|14.496414|
|       WG|    C|    17| 988.5911|43.284058|24.123186| 38.55362|
...
|Apartment|    C|    14| 1091.334|30.537722|  17.6535|30.582376|
+---------+-----+------+---------+---------+---------+---------+


SELECT type, grade, price, COALESCE(
    CASE WHEN size<30 THEN "small" ELSE NULL END,
    CASE WHEN size<80 THEN "medium" ELSE NULL END,
    CASE WHEN size<100 THEN "large" ELSE "xlarge" END) AS size from df


+---------+-----+----------+------+
|     type|grade|     price|  size|
+---------+-----+----------+------+
|Apartment|    A| 3271.7893| large|
|Apartment|    B| 3043.0178|medium|
|       WG|    B|  2311.949|xlarge|
|       WG|    C|  832.8406|medium|
|Apartment|    B| 3210.3003|medium|
|Apartment|    B| 3756.9866| large|
...
|    House|    A|  8788.338|xlarge|
|    House|    C| 3782.6099|xlarge|
|       WG|    B| 1800.9767| large|
|Apartment|    C| 790.08777| small|
```

You can use SQL at any complexity
Spark SQL can handle like other industry-grade RDBMS

SparkSQL supports Hive

Hive is a **relational database** engine for Hadoop



SparkSQL supports Hive

Hive table abstracts the access to physical files with SQL

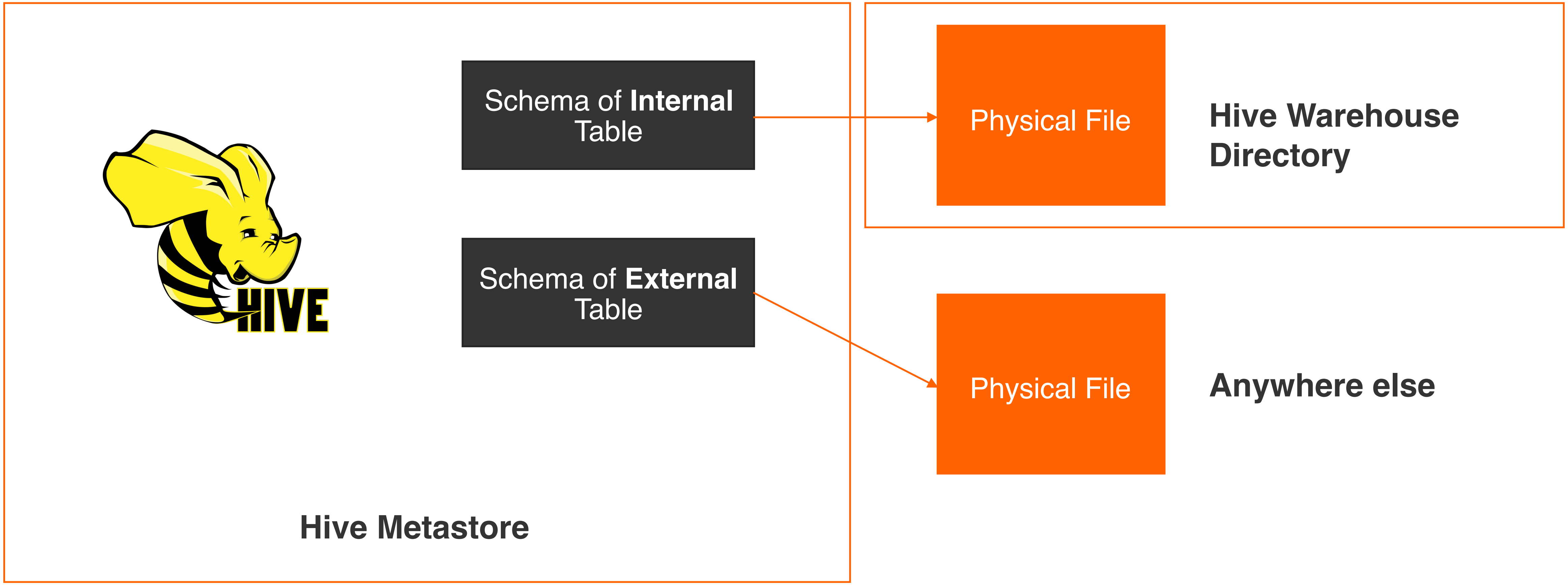
```
df = spark.read.csv("path/file.csv").filter(col("a") == 100).select("a","b")

df = spark.sql("SELECT a, b FROM schema.table WHERE a = 100")
```

To create a Hive table from an existing file

```
CREATE EXTERNAL TABLE IF NOT EXISTS schema.table(a INT, b STRING)
ROW FORMAT DELIMITED
FIELD TERMINATED BY ','
STORED AS CSV
LOCATION '/path/file.csv'
```

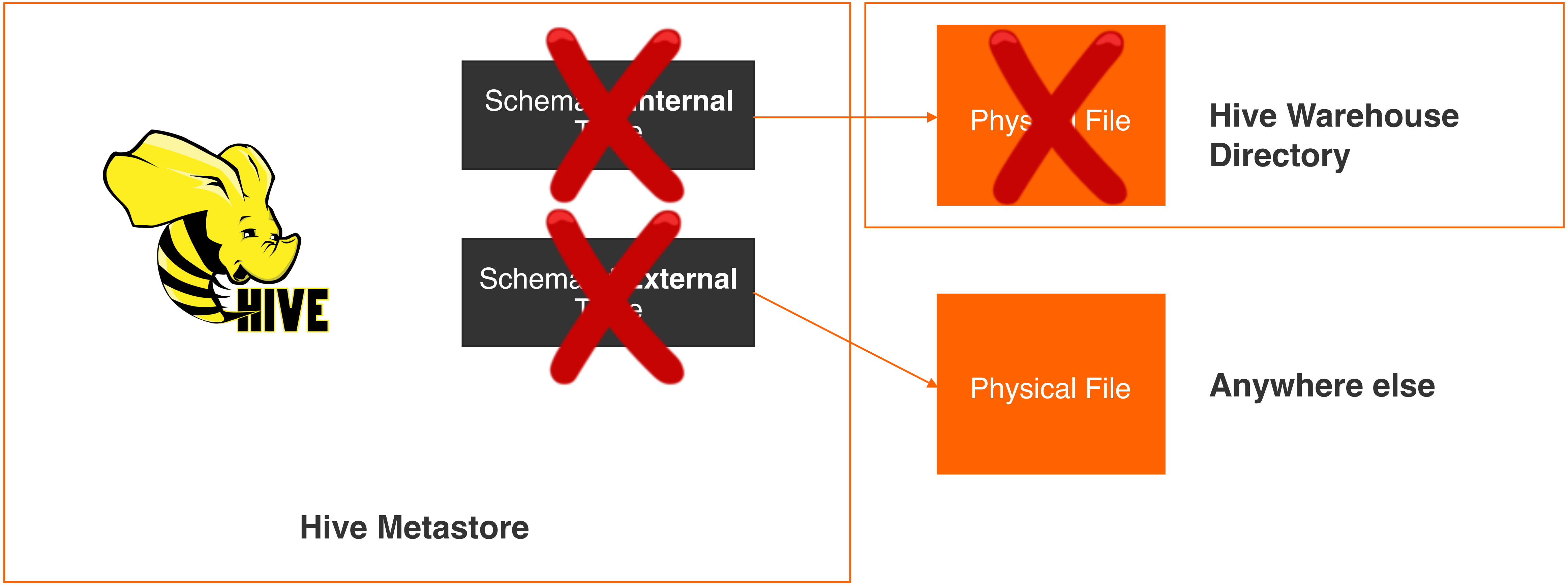
Hive External vs Internal table

External means outside of Hive’s Warehouse location



Hive External vs Internal table

Dropping external table also destroys the physical file



Hive vs Impala

Impala is upto 70x faster as it cache the direct pointer to the physical file

Schema

Hive Metastore

Physical File

Hive needs to do full scan based on partition

Schema

Impala Daemon process

Impala

Physical File part 1

Physical File part 2

Physical File part 3

Impala does not need to full scan. It already caches the mapping between table values and raw physical location