Apache Spark Dataframes Part 1

Source: Spark in Action, 2nd Edition
https://www.manning.com/books/spark-in-action-second-edition?query=spark%20in%20action

Spark: The Definitive Guide
https://learning-oreilly-com.proxy.library.nyu.edu/library/view/spark-the-definitive/9781491912201/

https://github.com/databricks/Spark-The-Definitive-Guide

NYU

Where to Look for APIs

https://spark.apache.org/docs/latest/

`DataFrame (Dataset) Methods`

This is actually a bit of a trick because a DataFrame is just a Dataset of **Row** types, so you'll actually end up looking at the Dataset methods

`Column Methods`

They hold a variety of general column-related methods like alias or contains.

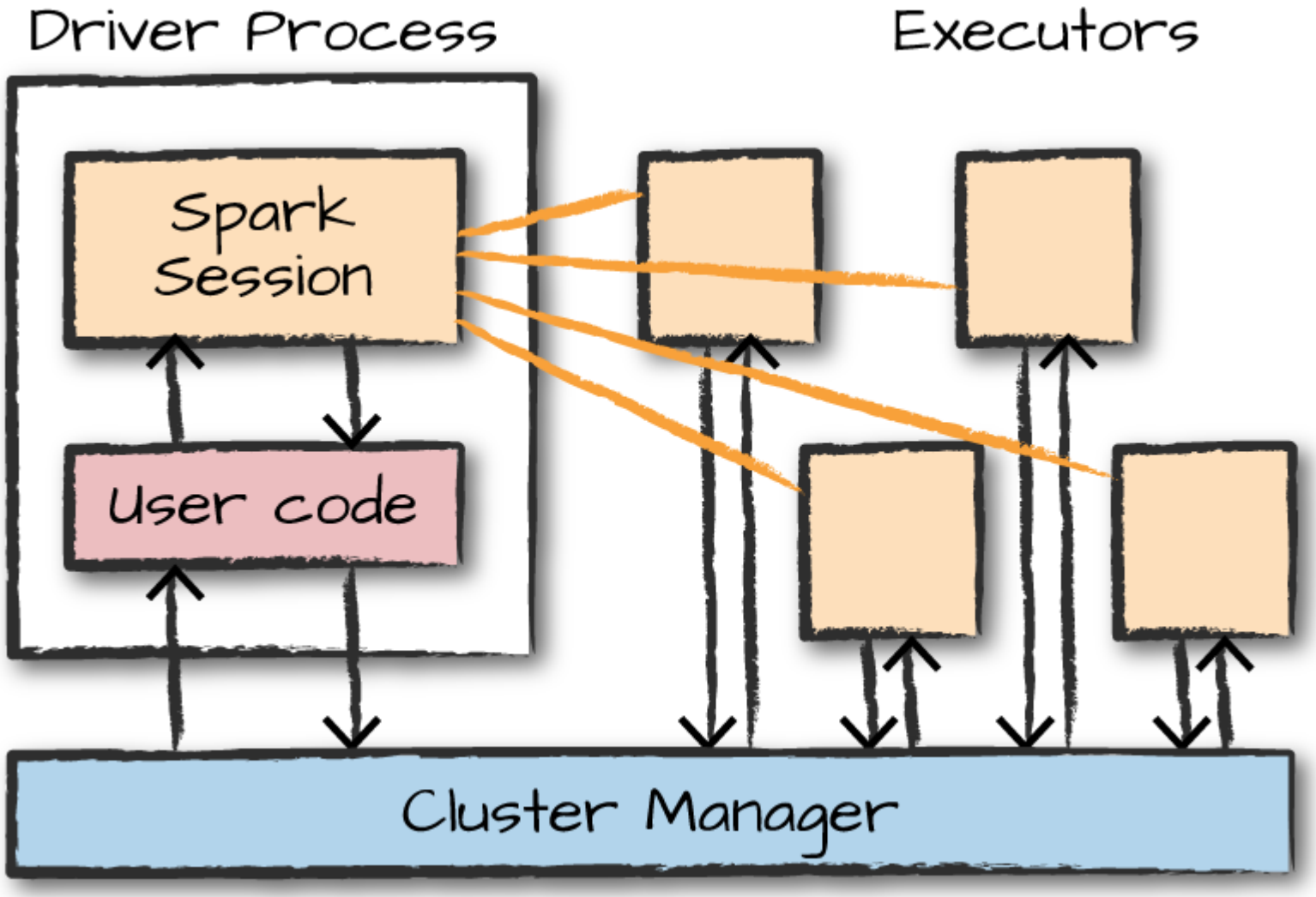
`org.apache.spark.sql.functions`

Contains a variety of functions for a range of different data types. Often, you'll see the entire package imported because they are used so frequently.

Datasets for this lecture

HPC JupyterHub, shared directory

The Spark Session



```
Java: SparkSession spark = SparkSession
        .builder().appName("")
        .master("local").getOrCreate();

Scala: val spark = SparkSession
        .builder().appName("")
        .master("local").getOrCreate();

Python: spark = SparkSession
        .builder().appName("")
        .master("local").getOrCreate();
```

The Spark Session

```
spark
```

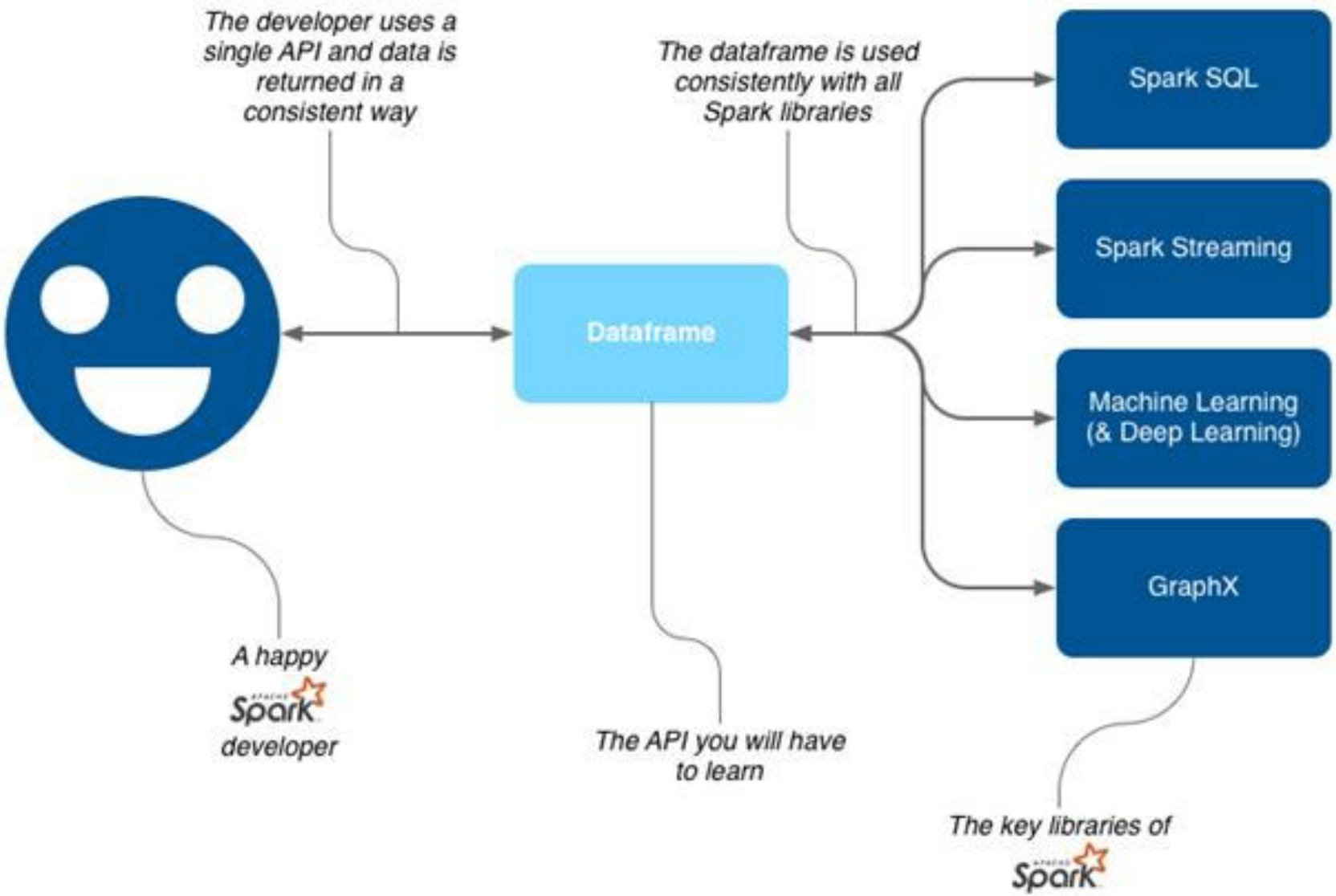
In Scala, you should see something like the following:

```
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.Spark
```

In Python you'll see something like this:

```
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>
```

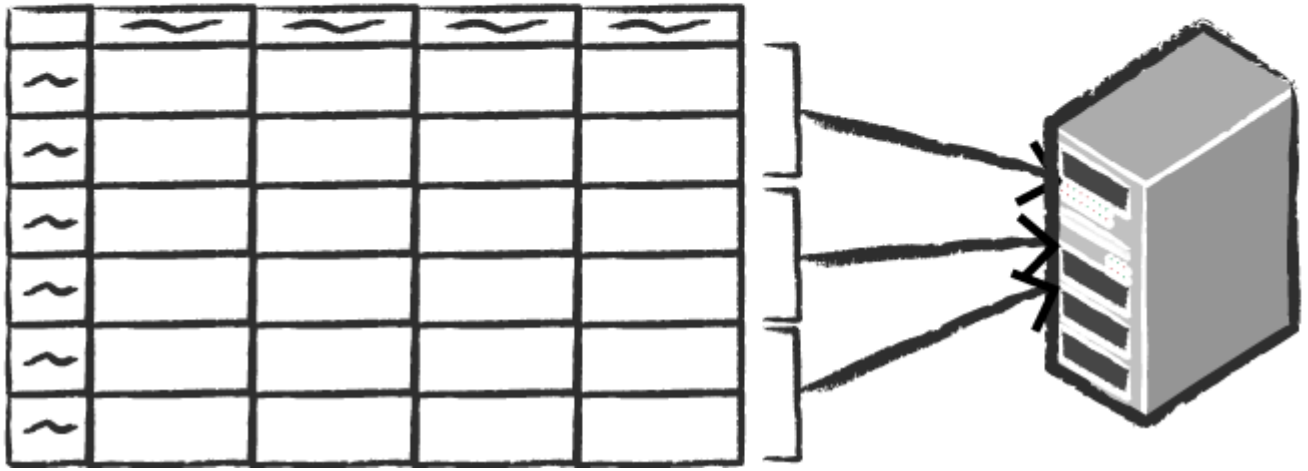
Dataframe


The developer uses a single API and data is returned in a consistent way
The dataframe is used consistently with all Spark libraries
Spark SQL
Spark Streaming
Machine Learning (& Deep Learning)
GraphX
Dataframe
A happy Spark developer
The API you will have to learn
The key libraries of Spark

Dataframe

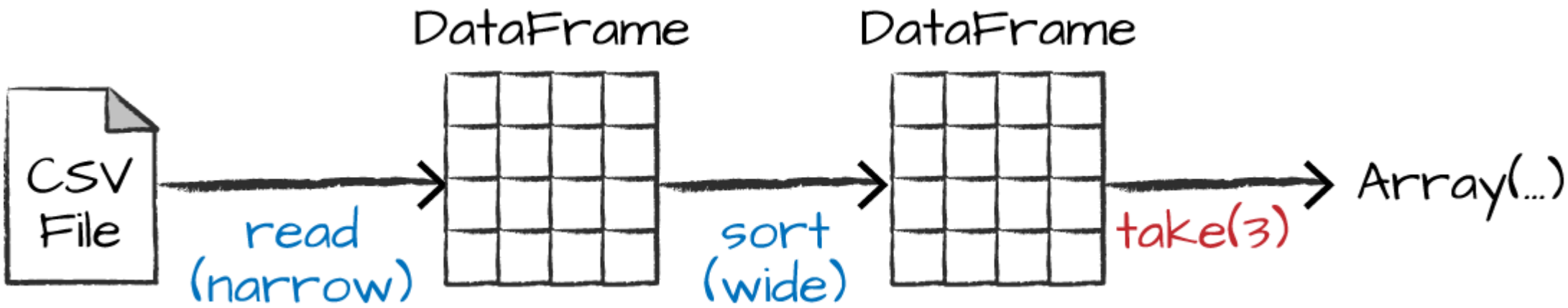
Spreadsheet on
a single machine

Table or Data Frame
partitioned across servers
in a data center



```
# in Python
flightData2015 = spark\
     .read\
     .option("inferSchema", "true")\
     .option("header", "true")\
     .csv("shared/spark-guide/data/flight-
data/csv/2015-summary.csv")
```

```
flightData2015.sort("count")
```




```
flightData2015.sort("count").explain()
```

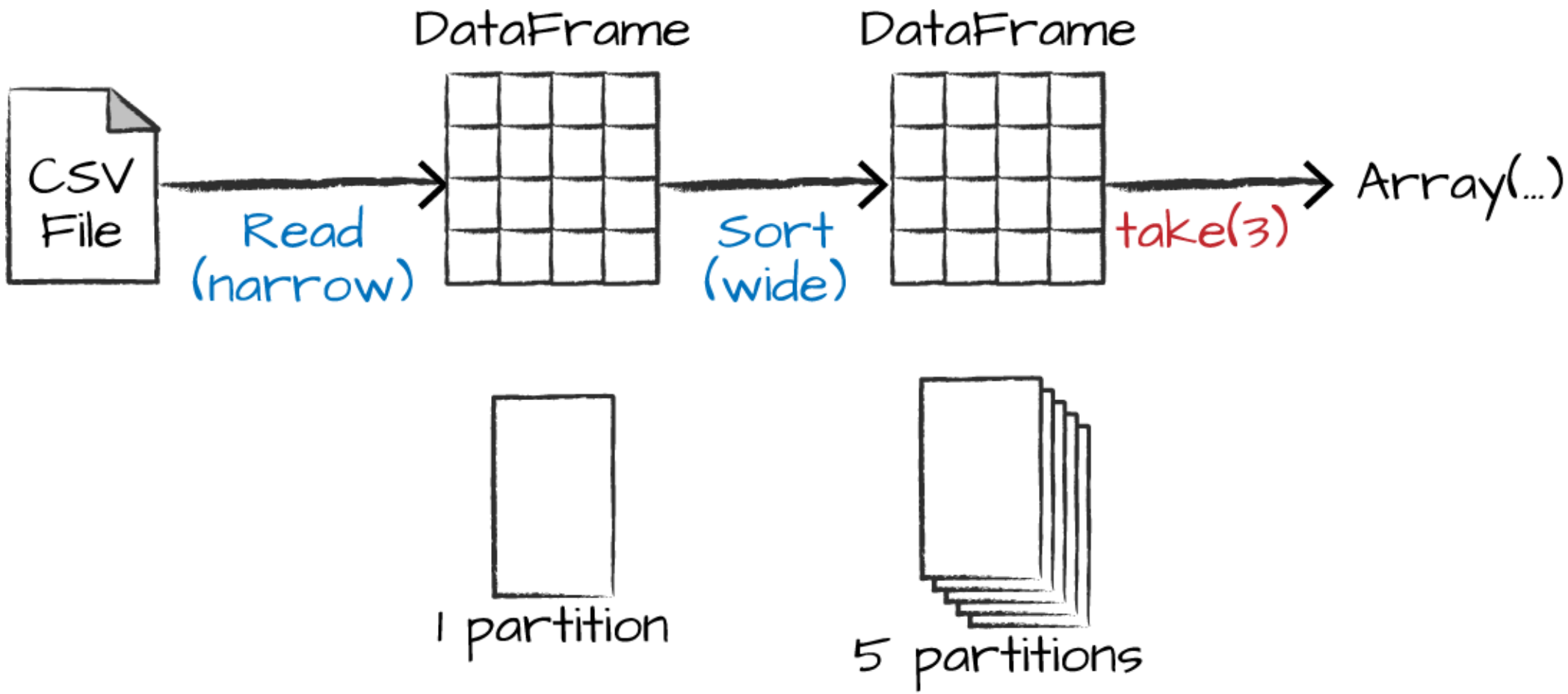
Partitions

- To allow every executor to perform work in parallel, Spark breaks up the data into chunks called *partitions*.

- A partition is a collection of rows that sit on one physical machine in your cluster

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
```

```
flightData2015.sort("count").take(3)
```



DataFrames and SQL

You can make any DataFrame into a table or view with one simple method call

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

```
# in Python

sqlWay = spark.sql(""" SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015 GROUP BY DEST_COUNTRY_NAME """)

dataFrameWay = flightData2015\
      .groupBy("DEST_COUNTRY_NAME")\
      .count()

sqlWay.explain()
dataFrameWay.explain()
```

```
spark.sql("SELECT max(count) from flight_data_2015").take(1)
```

```
# in Python

from pyspark.sql.functions import max

flightData2015.select(max("count")).take(1)
```

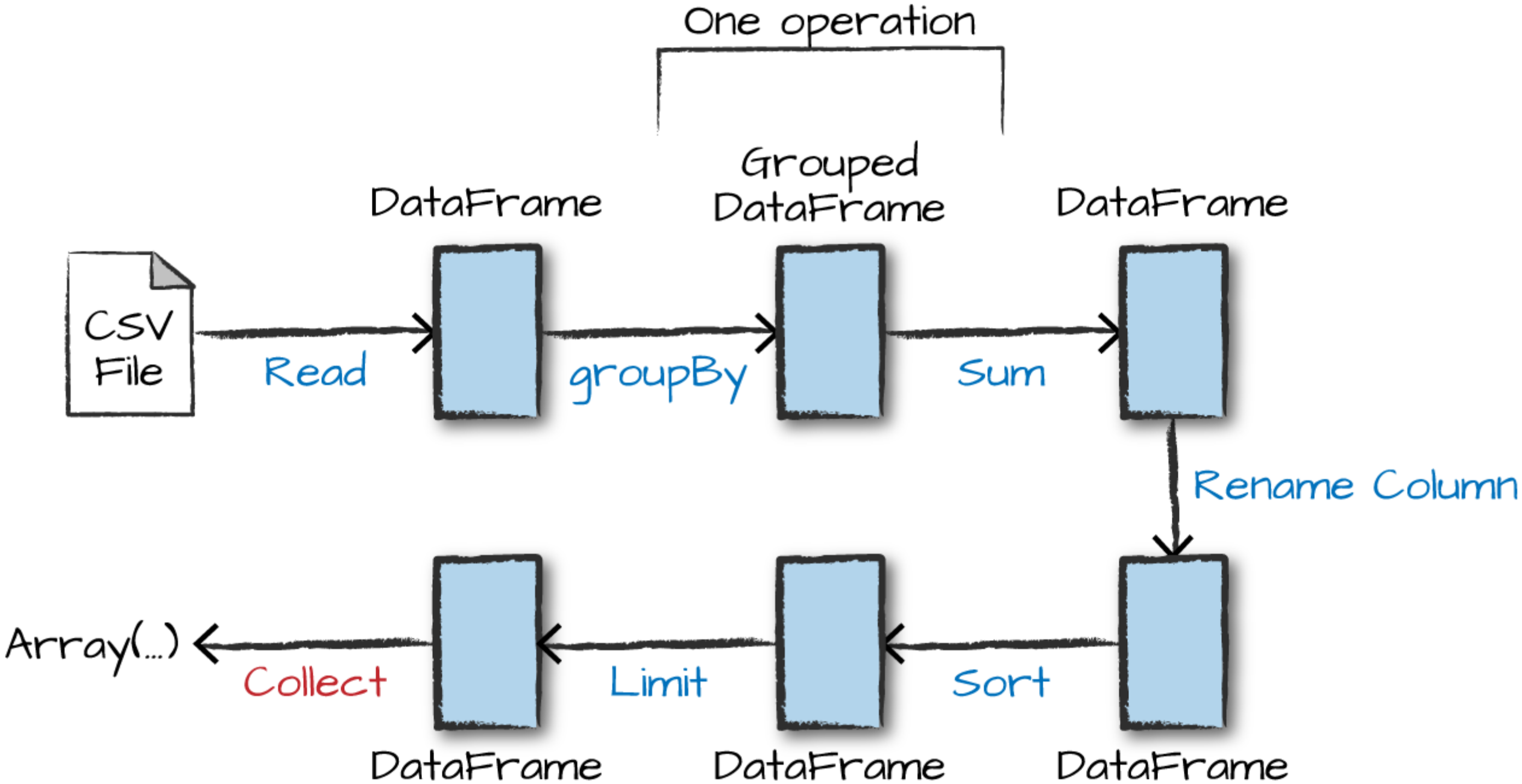
```
# in Python

maxSql = spark.sql(""" SELECT DEST_COUNTRY_NAME, sum(count)
      AS destination_total FROM flight_data_2015
      GROUP BY DEST_COUNTRY_NAME
      ORDER BY sum(count) DESC LIMIT 5 """)
maxSql.show()
```

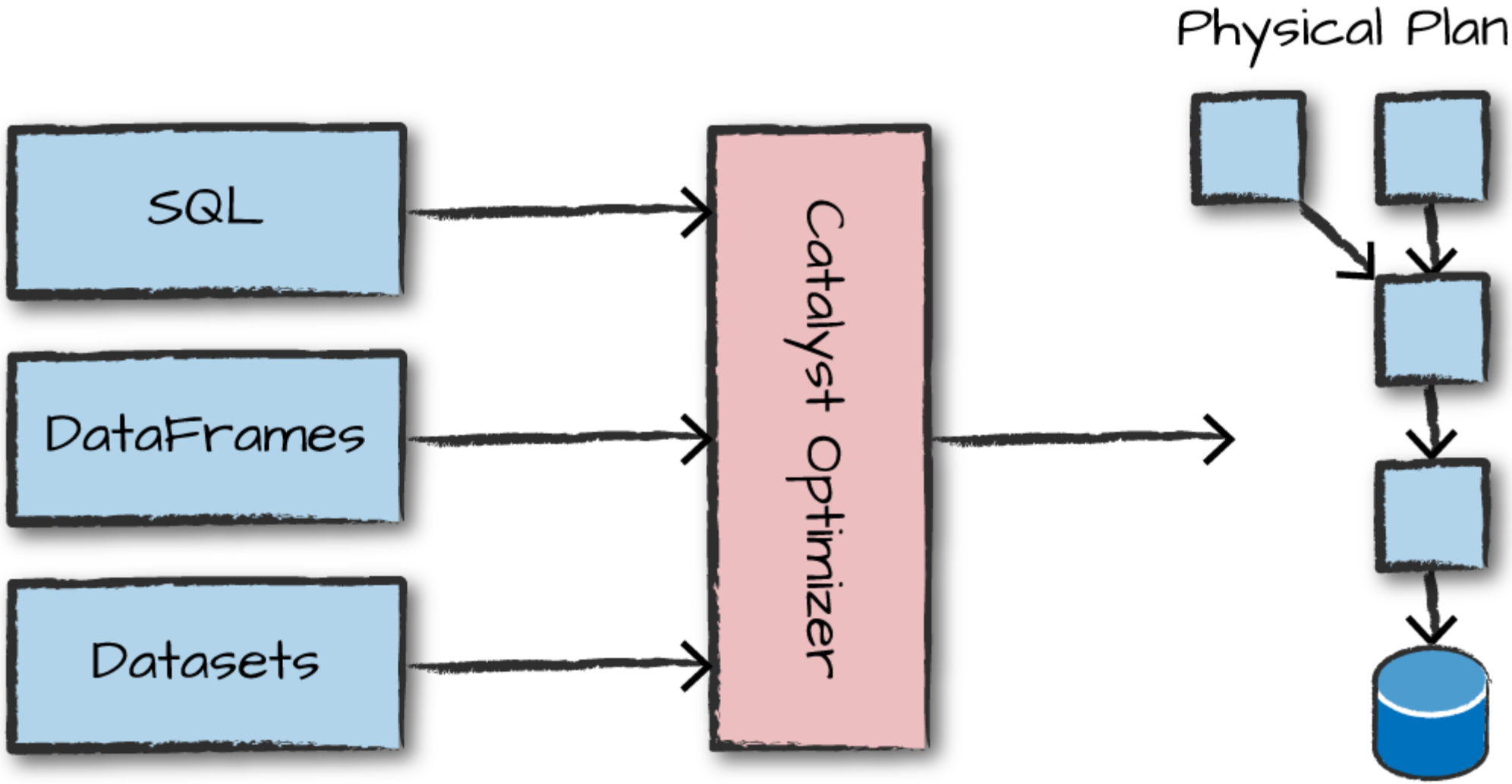
```
from pyspark.sql.functions import desc
flightData2015\
      .groupBy("DEST_COUNTRY_NAME")\
      .sum("count")\
      .withColumnRenamed("sum(count)", "destination_total")\
      .sort(desc("destination_total"))\
      .limit(5)\
      .show()
```

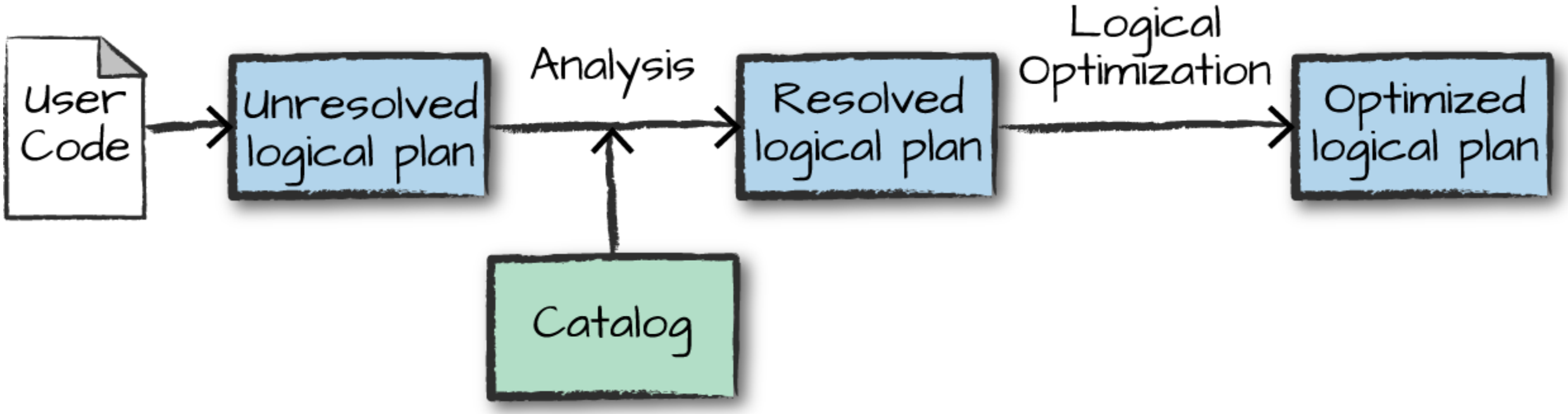
Transformations

Lazy Evaluation

Actions


One operation
DataFrame
Grouped DataFrame
DataFrame
CSV File
Read
groupBy
Sum
Rename Column
Array(...)
Collect
Limit
Sort
DataFrame
DataFrame
DataFrame


SQL
DataFrames
Datasets
Catalyst Optimizer
Physical Plan


User Code
Unresolved logical plan
Analysis
Catalog
Resolved logical plan
Logical Optimization
Optimized logical plan

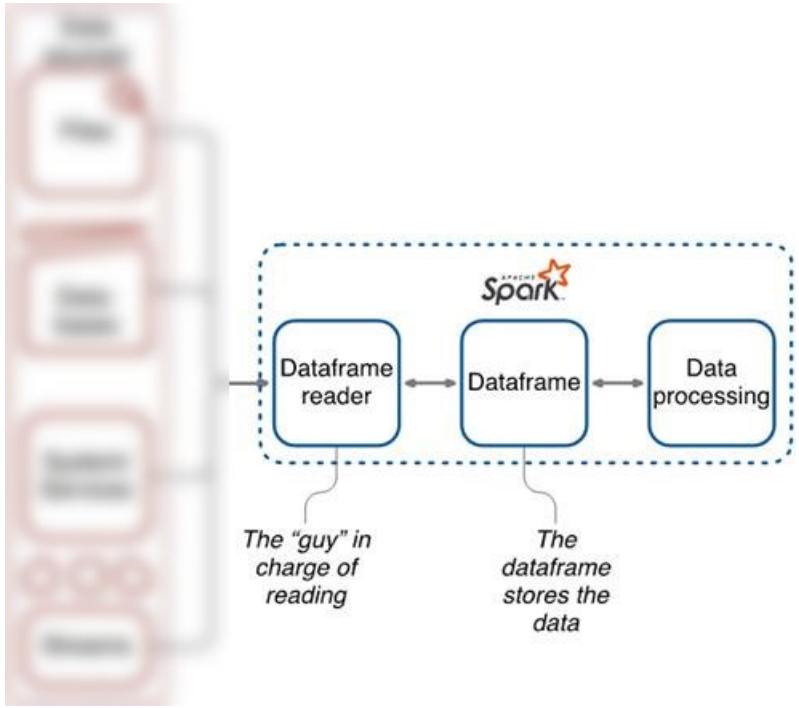
Actions

- collect()
- count()
- describe()
- foreach()
- foreachPartition()
- head()

Transformations

- map()
- coaslesce()
- distinct()
- filter(), where()
- drop()
- withColumn(), withColumnRenamed()
- groupByKey() *
- orderBy()
- join()
- select()

Dataframe



org.apache.spark.sql
Dataset

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset

`spark.Dataset[Row]`

- Actions
- Functions
- Transformations

Creating DataFrames

```
# in Python

df = spark.read.\
     format("json")
     .load(“shared/spark-guide/data/flight-data/json/2015-
summary.json")

df.printSchema()

df.createOrReplaceTempView("dfTable")
```

Columns

```
# in Python

from pyspark.sql.functions import col, column
col("someColumnName")
column("someColumnName")
```

Explicit column references

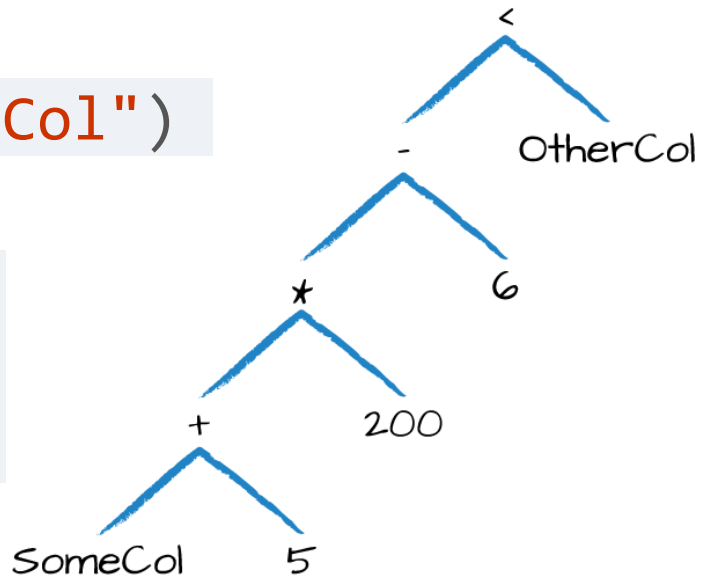
```
df.col("count")
```

Columns as expressions

`expr("someCol - 5")` is the same transformation as `col("someCol") - 5`, or even `expr("someCol") - 5`.

- Columns are just expressions.
- Columns and transformations of those columns compile to the same logical plan as parsed expressions.

```
(((col("someCol") + 5) * 200) - 6) < col("otherCol")
```




```
# in Python
from pyspark.sql.functions import expr
expr("(((someCol + 5) * 200) - 6) < otherCol")
```

Records and Rows

Each row in a DataFrame is a single record. Spark represents this record as an object of type **`Row`**

```
# in Python
from pyspark.sql import Row

myRow = Row("Hello", None, 1, False)
```

```
# in Python
myRow[0] myRow[2]
```

```python
# in Python

from pyspark.sql import Row
from pyspark.sql.types import StructField, StructType, StringType,
LongType

myManualSchema = StructType([
      StructField("some", StringType(), True),
      StructField("col", StringType(), True),
      StructField("names", LongType(), False) ])

myRow = Row("Hello", None, 1)
myDf = spark.createDataFrame([myRow], myManualSchema)
myDf.show()
```

select and selectExpr

```
-- in SQL SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME FROM
dfTable LIMIT 2
```

```
# in Python df.select("DEST_COUNTRY_NAME",
"ORIGIN_COUNTRY_NAME").show(2)
```

```
# in Python
from pyspark.sql.functions import expr, col, column
df.select( expr("DEST_COUNTRY_NAME"),
      col("DEST_COUNTRY_NAME"),
      column("DEST_COUNTRY_NAME"))\
      .show(2)
```

```
df.select(col("DEST_COUNTRY_NAME"), "DEST_COUNTRY_NAME")
```

Literals

```
# in Python

from pyspark.sql.functions import lit

df.select(expr("*"), lit(1).alias("One")).show(2)
```

Adding Columns

```
# in Python
df.withColumn("numberOne", lit(1)).show(2)
```

Renaming Columns

```
# in Python
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns
```

Changing a Column's Type (cast)

```
df.withColumn("count2", col("count").cast("long"))
```

Filtering Rows

```
df.filter(col("count") < 2).show(2)
df.where("count < 2").show(2)
```

```
# in Python
df.where(col("count") < 2)\
      .where(col("ORIGIN_COUNTRY_NAME") != "Croatia")\
      .show(2)
```

Getting Unique Rows

```
# in Python
df.select("ORIGIN_COUNTRY_NAME",
"DEST_COUNTRY_NAME").distinct().count()
```

```
# in Python
df.select("ORIGIN_COUNTRY_NAME").distinct().count()
```

Random Samples

```
# in Python
seed = 5 withReplacement = False fraction = 0.5
df.sample(withReplacement, fraction, seed).count()
```

```
# in Python
dataFrames = df.randomSplit([0.25, 0.75], seed)
dataFrames[0].count() > dataFrames[1].count() # False
```

Repartition and Coalesce

```
# in Python
df.rdd.getNumPartitions() # 1
```

```
# in Python
df.repartition(5)
```

If you know that you're going to be filtering by a certain column often, it can be worth repartitioning based on that column:

```
# in Python
df.repartition(col("DEST_COUNTRY_NAME"))
```

```
# in Python
df.repartition(5, col("DEST_COUNTRY_NAME"))
```

partitionBy()

Spark partitionBy() is a function of `pyspark.sql.DataFrameWriter` class which is used to partition based on one or multiple column values while writing DataFrame to Disk/File system.

```
$ ls -lrt zipcodes-state
total 24
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=AL'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=AZ'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=FL'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=NC'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=PR'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=TX'/
-rw-r--r-- 1 prabha 197121 0 Mar  4 21:18  _SUCCESS
```

Too Many Partitions Good?

- If you are a beginner, you would think too many partitions will boost the Spark Job Performance actually, it won't and it's overkill.
- Spark has to create one task per partition and most of the time goes into creating, scheduling, and managing the tasks then executing.

Collecting Rows to the Driver

There are times when you'll want to collect some of your data to the driver in order to manipulate it on your local machine.

```
# in Python

# take works with an Integer count
collectDF = df.limit(10) collectDF.take(5)
collectDF.show()

# this prints it out nicely
collectDF.show(5, False)
collectDF.collect()
```

WARNING

Any collection of data to the driver can be a very expensive operation! If you have a large dataset and call collect, you can crash the driver.

Spark.SQL Types

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.types.package

PySpark Documentation — PySpark 3.1.2 documentation

Types of Data

- Booleans
- Numbers
- Strings
- Dates and timestamps
- Handling null
- Complex types
- User-defined functions

https://spark.apache.org/docs/latest/

Spark Data Types

Literals

```
# in Python
from pyspark.sql.functions import lit
df.select(lit(5), lit("five"), lit(5.0))
```

Booleans

```
# in Python
from pyspark.sql.functions import col
df.where(col("InvoiceNo") != 536365)\
      .select("InvoiceNo", "Description")\
      .show(5, False)
```

```
# in Python
from pyspark.sql.functions import expr
df.withColumn("isExpensive", expr("NOT UnitPrice <= 250"))\
      .where("isExpensive")\
      .select("Description", "UnitPrice").show(5)
```

- To filter a `DataFrame`, you can also just specify a Boolean column

* shared/spark-guide/data/retail-data/all/online-retail-dataset.csv

Numbers

```
# in Python
from pyspark.sql.functions import expr, pow
fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
df.select(expr("CustomerId"),
fabricatedQuantity.alias("realQuantity")).show(2)
```

Strings

```
# in Python
from pyspark.sql.functions import initcap
df.select(initcap(col("Description"))).show()
```

```
# in Python
from pyspark.sql.functions import lower, upper
df.select(col("Description"), lower(col("Description")),
upper(lower(col("Description")))).show(2)
```

Regular Expressions

```
# in Python

from pyspark.sql.functions import regexp_replace

regex_string = "BLACK|WHITE|RED|GREEN|BLUE"
df.select( regexp_replace(col("Description"), regex_string,
"COLOR").alias("color_clean"), col("Description")).show(2)
```

Dates and Timestamps

```
# in Python
from pyspark.sql.functions import current_date, current_timestamp
dateDF = spark.range(10)\
      .withColumn("today", current_date())\
      .withColumn("now", current_timestamp())
dateDF.createOrReplaceTempView("dateTable")
```

```
dateDF.printSchema()
```

```
# in Python from pyspark.sql.functions import date_add, date_sub
dateDF.select(date_sub(col("today"), 5), date_add(col("today"),
5)).show(1)
```

```
-- in SQL
SELECT date_sub(today, 5), date_add(today, 5) FROM dateTable
```

```
# in Python
from pyspark.sql.functions import datediff, months_between, to_date

dateDF.withColumn("week_ago", date_sub(col("today"), 7))\
      .select(datediff(col("week_ago"), col("today")))\
      .show(1)

dateDF.select( to_date(lit("2016-01-01")).alias("start"),
to_date(lit("2017-05-22"))\
      .alias("end"))\
      .select(months_between(col("start"), col("end")))\
      .show(1)
```

Nulls in Data

Two things you can do with null values:
- you can explicitly drop nulls
- you can fill them with a value (globally or on a per-column basis)

Coalesce

Select the first non-null value from a set of columns

```
# in Python
from pyspark.sql.functions import coalesce
df.select(coalesce(col("Description"), col("CustomerId")))).show()
```

drop

```
df.na.drop()
df.na.drop("any")
```

```
# in Python
df.na.drop("all", subset=["StockCode", "InvoiceNo"])
```

fill

```
df.na.fill("All Null values become this string")
```

```
# in Python
df.na.fill("all", subset=["StockCode", "InvoiceNo"])
```

replace

```
# in Python df.na.replace([""], ["UNKNOWN"], "Description")
```

Arrays

split

```
# in Python
from pyspark.sql.functions import split
df.select(split(col("Description"), " ")).show(2)
```

```
-- in SQL
SELECT split(Description, ' ') FROM dfTable
```

Array Length

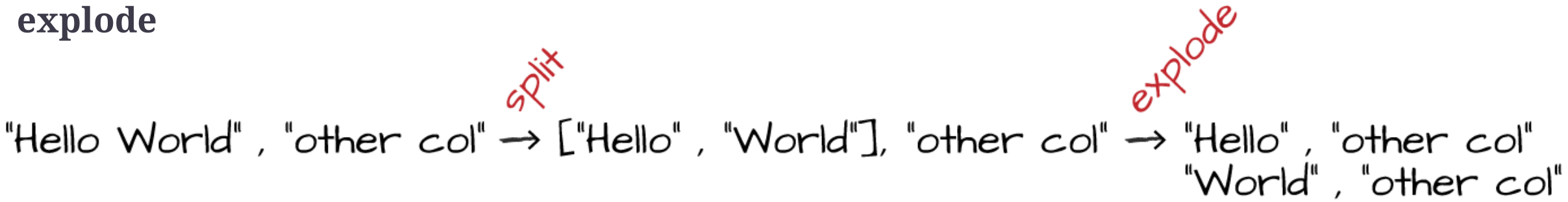
```
# in Python
from pyspark.sql.functions import size
df.select(size(split(col("Description"), " "))).show(2) # shows
5 and 3
```

array_contains

```
# in Python from pyspark.sql.functions import array_contains
df.select(array_contains(split(col("Description"), " "),
"WHITE")).show(2)
```

```
-- in SQL
SELECT array_contains(split(Description, ' '), 'WHITE') FROM
dfTable
```

explode




```
# in Python
from pyspark.sql.functions import split, explode
df.withColumn("splitted", split(col("Description"), " "))\
      .withColumn("exploded", explode(col("splitted")))\ .
      select("Description", "InvoiceNo", "exploded")
      .show(2)

-- in SQL
SELECT Description, InvoiceNo, exploded FROM (SELECT *,
split(Description, " ") as splitted FROM dfTable) LATERAL VIEW
explode(splitted) as exploded
```

Complex Types

Structs Think of structs as DataFrames within DataFrames

```
df.selectExpr("(Description, InvoiceNo) as complex", "*")
```

```
df.selectExpr("struct(Description, InvoiceNo) as complex", "*")
```

```
# in Python
from pyspark.sql.functions import struct
complexDF = df.select(struct("Description", "InvoiceNo")\
      .alias("complex"))
complexDF.createOrReplaceTempView("complexDF")
```

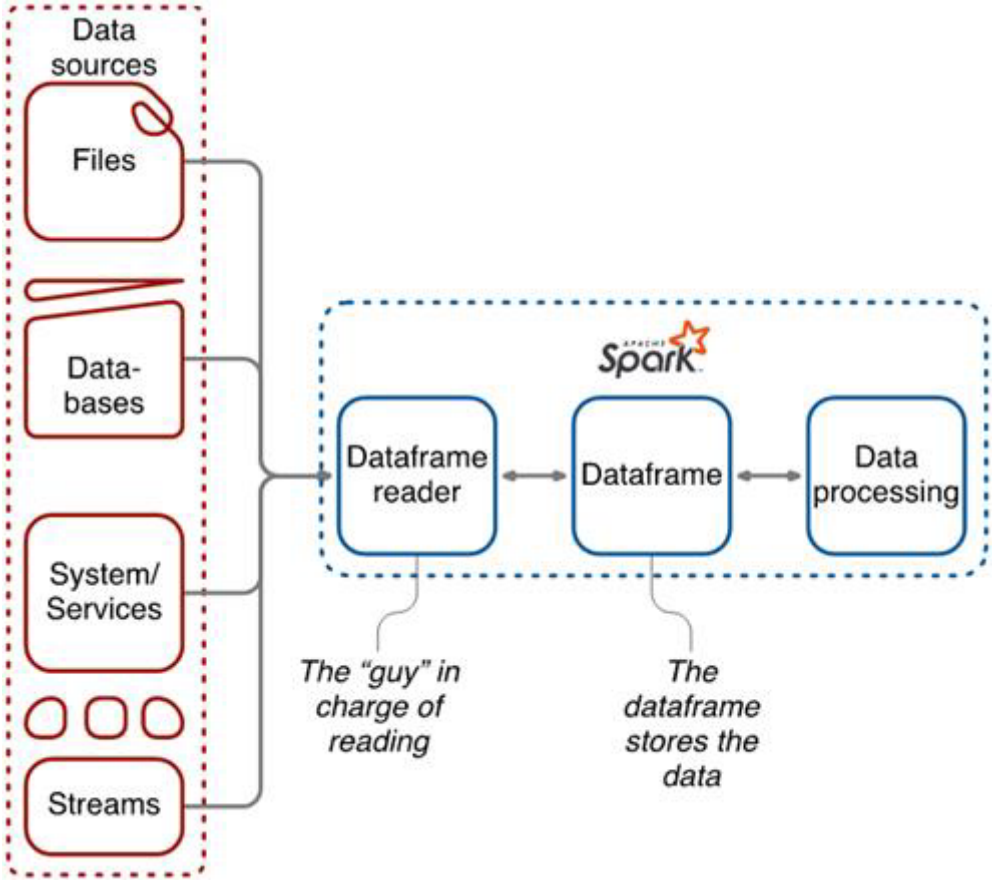
```
complexDF.select("complex.Description")
complexDF.select(col("complex").getField("Description"))
complexDF.select(“complex.*”)
```

Maps

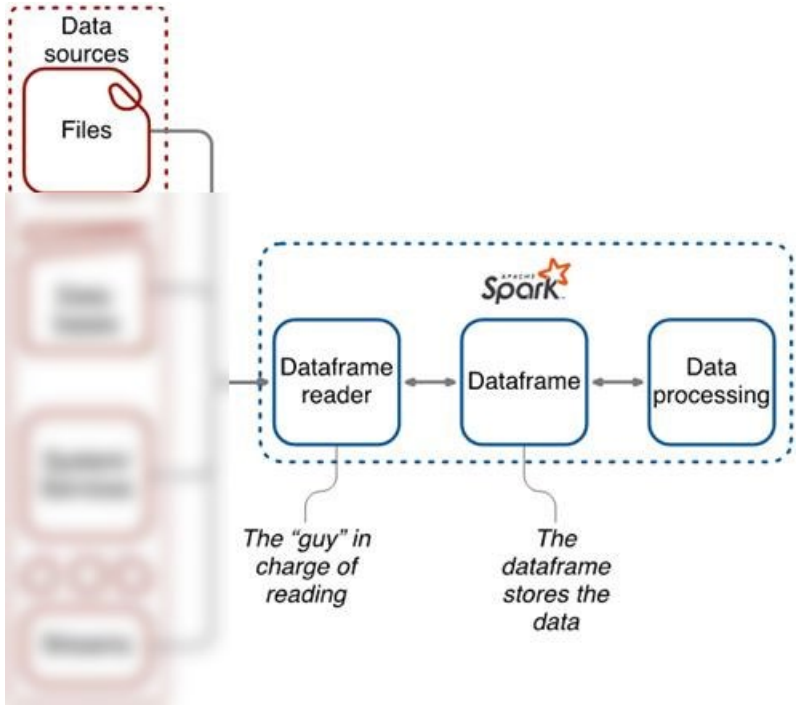
```
# in Python

from pyspark.sql.functions import create_map
df.select(create_map(col("Description"),
col("InvoiceNo")).alias("complex_map"))\ .show(2)
```

```
-- in SQL
SELECT map(Description, InvoiceNo) as complex_map FROM dfTable
WHERE Description IS NOT NULL
```

Ingestion


Data sources
Files
Data-bases
System/ Services
Streams
Spark
Dataframe reader
Dataframe
Data processing
The "guy" in charge of reading
The dataframe stores the data

Ingestion



org.apache.spark.sql
DataFrameReader

```
class DataFrameReader extends Logging
```

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader

SparkSession.read

- CSV
- SQL
- XML
- JSON
- TEXT
- TEXTFILE
- PARQUET

- format()
- option()
- schema()
- load()

SparkSession.read - Options

Format specific:

- **CSV**: https://docs.databricks.com/data/data-sources/read-csv.html#supported-options

  - header
  - delimiter
  - quote
  - inferSchema
  - …

Running Production Applications : spark-submit

```
Python

./bin/spark-submit \ --master local \
./examples/src/main/python/pi.py 10
```

NEXT…

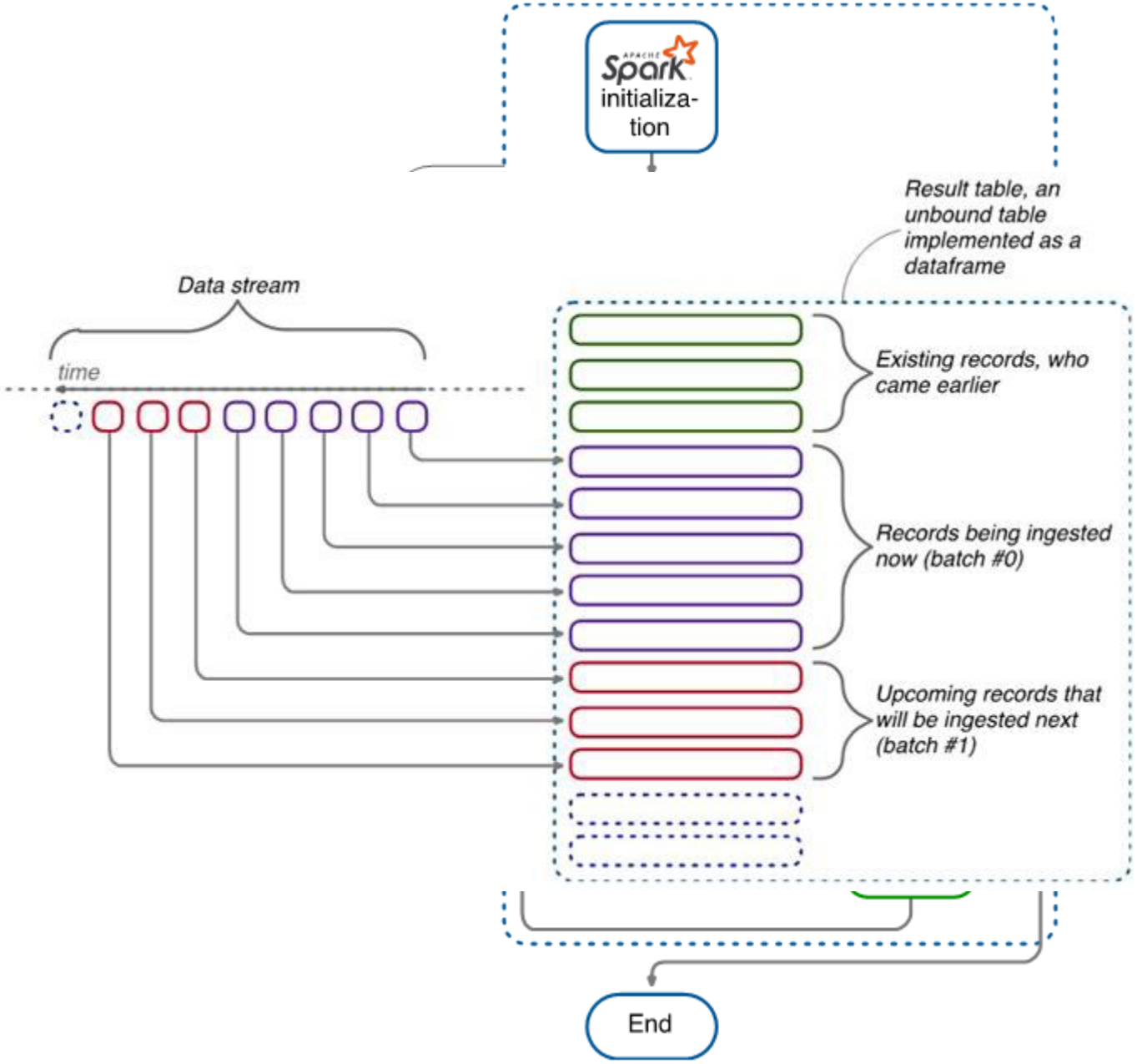
JSON – working with JSON

UDF – user Defined Functions

Aggregations

Joins

Streams


Spark initializa-tion
Result table, an unbound table implemented as a dataframe
Data stream
time
Existing records, who came earlier
Records being ingested now (batch #0)
Upcoming records that will be ingested next (batch #1)
End