Apache Spark
Part 2

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

- JSON – working with JSON
- UDF – user Defined Functions
- Aggregations
- Joins
- Streams

JSON – working with JSON

```python
# in Python
jsonDF = spark.range(1).selectExpr(""" '{"myJSONKey" :
{"myJSONValue" : [1, 2, 3]}}' as jsonString""")
```

- You can use the `get_json_object` to inline query a JSON object, be it a dictionary or array.
- You can use `json_tuple` if this object has only one level of nesting

```python
# in Python
from pyspark.sql.functions import get_json_object, json_tuple

jsonDF.select( get_json_object(col("jsonString"),
"$.myJSONKey.myJSONValue[1]") as "column",
json_tuple(col("jsonString"), "myJSONKey")).show(2)
```

- turn a StructType into a JSON string

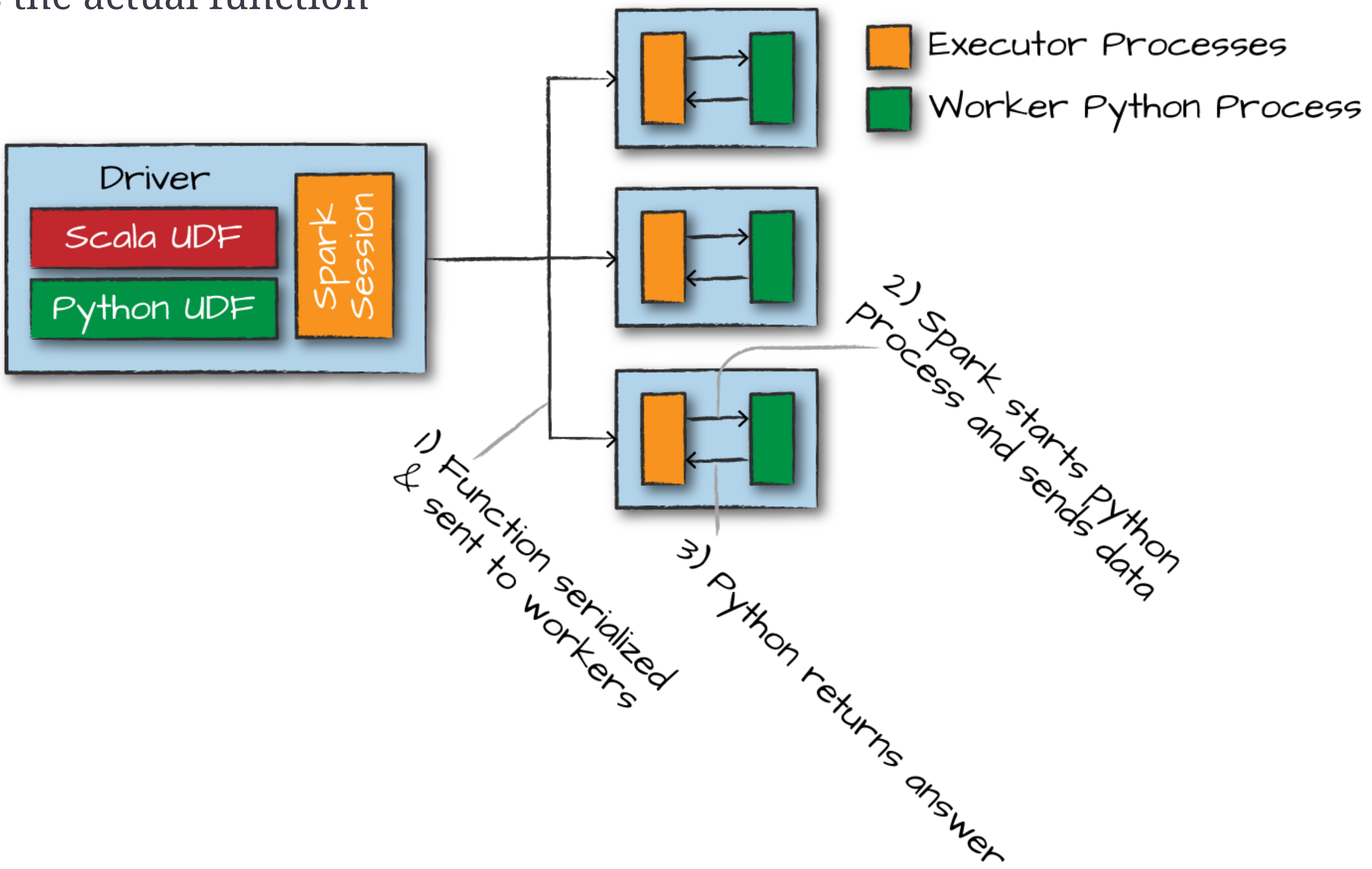
```
# in Python
from pyspark.sql.functions import to_json
df.selectExpr("(InvoiceNo, Description) as myStruct")\
.select(to_json(col("myStruct")))
```

```
# in Python
from pyspark.sql.functions import from_json
from pyspark.sql.types import *

parseSchema = StructType((
StructField("InvoiceNo",StringType(),True),
StructField("Description",StringType(),True)))
df.selectExpr("(InvoiceNo, Description) as myStruct")\
.select(to_json(col("myStruct")).alias("newJSON"))\
.select(from_json(col("newJSON"), parseSchema),
col("newJSON")).show(2)
```

User-Defined Functions

- There are performance considerations
- The first step is the actual function



User-Defined Functions

```
# in Python
udfExampleDF = spark.range(5).toDF("num")

def power3(double_value): return double_value ** 3

power3(2.0)
```

```
# in Python
from pyspark.sql.functions import udf
power3udf = udf(power3)
```

```
# in Python from pyspark.sql.functions import col
udfExampleDF.select(power3udf(col("num"))).show(2)
```

User-Defined Functions

- we can register this UDF as a Spark SQL function

```
# in Python
from pyspark.sql.types import IntegerType, DoubleType
spark.udf.register("power3udf", power3, DoubleType())
```

```
# in Python
udfExampleDF.selectExpr("power3udf(num)").show(2)
# registered via Python
```

- careful with return types and conversions; Spark will not throw error, just Nulls

Aggregations

The act of collecting something together

Let’s use the retail data:

```
# in Python

df = spark.read.format("csv")\
      .option("header", "true")\
      .option("inferSchema", "true")\
      .load(“shared/spark-guide/data/retail-data/all/*.csv")\
      .coalesce(5)

df.cache()
df.createOrReplaceTempView("dfTable")
```

NYU

Aggregation Functions -DataFrame

count

countDistinct

approx_count_distinct

first and last

min and max

sum

sumDistinct

avg

Variance and Standard Deviation

skewness and kurtosis

Covariance and Correlation

Aggregating to Complex Types

Aggregation Functions

count

```
# in Python
from pyspark.sql.functions import count
df.select(count("StockCode")).show() # 541909

-- in SQL
SELECT COUNT(*) FROM dfTable
```

Aggregation Functions

countDistinct

```
// in Scala
import org.apache.spark.sql.functions.countDistinct
df.select(countDistinct("StockCode")).show() // 4070

# in Python from pyspark.sql.functions import countDistinct
df.select(countDistinct("StockCode")).show() # 4070

-- in SQL
SELECT COUNT(DISTINCT *) FROM DFTABLE
```

Aggregation Functions

approx_count_distinct

```
// in Scala
import org.apache.spark.sql.functions.approx_count_distinct
df.select(approx_count_distinct("StockCode", 0.1)).show() // 3364

# in Python
from pyspark.sql.functions import approx_count_distinct
df.select(approx_count_distinct("StockCode", 0.1)).show() # 3364

-- in SQL
SELECT approx_count_distinct(StockCode, 0.1) FROM DFTABLE
```

Aggregation Functions

first and last

```
// in Scala
import org.apache.spark.sql.functions.{first, last}
df.select(first("StockCode"), last("StockCode")).show()

# in Python
from pyspark.sql.functions import first, last
df.select(first("StockCode"), last("StockCode")).show()

-- in SQL
SELECT first(StockCode), last(StockCode) FROM dfTable
```

Aggregation Functions

min and max

```
// in Scala
import org.apache.spark.sql.functions.{min, max}
df.select(min("Quantity"), max("Quantity")).show()

# in Python
from pyspark.sql.functions import min, max
df.select(min("Quantity"), max("Quantity")).show()

-- in SQL
SELECT min(Quantity), max(Quantity) FROM dfTable
```

NYU

Aggregation Functions

sum

```
// in Scala
import org.apache.spark.sql.functions.sum
df.select(sum("Quantity")).show() // 5176450

# in Python
from pyspark.sql.functions import sum
df.select(sum("Quantity")).show() # 5176450

-- in SQL
SELECT sum(Quantity) FROM dfTable
```

Aggregation Functions

sumDistinct

```
// in Scala
import org.apache.spark.sql.functions.sumDistinct
df.select(sumDistinct("Quantity")).show() // 29310

# in Python
from pyspark.sql.functions import sumDistinct
df.select(sumDistinct("Quantity")).show() # 29310

-- in SQL
SELECT SUM(Quantity) FROM dfTable -- 29310
```

Aggregations

avg

```
# in Python

from pyspark.sql.functions import sum, count, avg, expr
df.select( count("Quantity").alias("total_transactions"),
sum("Quantity").alias("total_purchases"),
avg("Quantity").alias("avg_purchases"),
expr("mean(Quantity)").alias("mean_purchases"))\ .selectExpr(
"total_purchases/total_transactions", "avg_purchases",
"mean_purchases")
.show()
```

Variance and Standard Deviation

```
# in Python
from pyspark.sql.functions import var_pop, stddev_pop
from pyspark.sql.functions import var_samp, stddev_samp

df.select(var_pop("Quantity"), var_samp("Quantity"),
stddev_pop("Quantity"), stddev_samp("Quantity")).show()

-- in SQL
SELECT var_pop(Quantity), var_samp(Quantity), stddev_pop(Quantity),
stddev_samp(Quantity) FROM dfTable
```

Aggregations

skewness and kurtosis

```
// in Scala
import org.apache.spark.sql.functions.{skewness, kurtosis}
df.select(skewness("Quantity"), kurtosis("Quantity")).show()

# in Python
from pyspark.sql.functions import skewness, kurtosis
df.select(skewness("Quantity"), kurtosis("Quantity")).show()

-- in SQL
SELECT skewness(Quantity), kurtosis(Quantity) FROM dfTable
```

Aggregations

Covariance and Correlation

```
// in Scala
import org.apache.spark.sql.functions.{corr, covar_pop, covar_samp}
df.select(corr("InvoiceNo", "Quantity"), covar_samp("InvoiceNo",
"Quantity"), covar_pop("InvoiceNo", "Quantity")).show()

# in Python
from pyspark.sql.functions import corr, covar_pop, covar_samp
df.select(corr("InvoiceNo", "Quantity"), covar_samp("InvoiceNo",
"Quantity"), covar_pop("InvoiceNo", "Quantity")).show()

-- in SQL SELECT corr(InvoiceNo, Quantity), covar_samp(InvoiceNo,
Quantity), covar_pop(InvoiceNo, Quantity) FROM dfTable
```

Aggregating to Complex Types

```
// in Scala
import org.apache.spark.sql.functions.{collect_set,
collect_list}
df.agg(collect_set("Country"), collect_list("Country")).show()

# in Python
from pyspark.sql.functions import collect_set, collect_list
df.agg(collect_set("Country"), collect_list("Country")).show()

-- in SQL
SELECT collect_set(Country), collect_set(Country) FROM dfTable
```

Aggregation Functions - Grouping

- Calculations based on groups in the data
- Two steps:
  1. GroupBy returns a **`RelationalGroupedDataset`**
  2. Aggregate returns a **`DataFrame`**

```
df.groupBy("InvoiceNo", "CustomerId").count().show()
```

```
-- in SQL
SELECT count(*) FROM dfTable GROUP BY InvoiceNo, CustomerId
```

Grouping with Expressions

```
// in Scala
import org.apache.spark.sql.functions.count
df.groupBy("InvoiceNo").agg( count("Quantity").alias("quan"),
expr("count(Quantity)")).show()

# in Python
from pyspark.sql.functions import count
df.groupBy("InvoiceNo").agg( count("Quantity").alias("quan"),
expr("count(Quantity)")).show()
```

Grouping with Maps

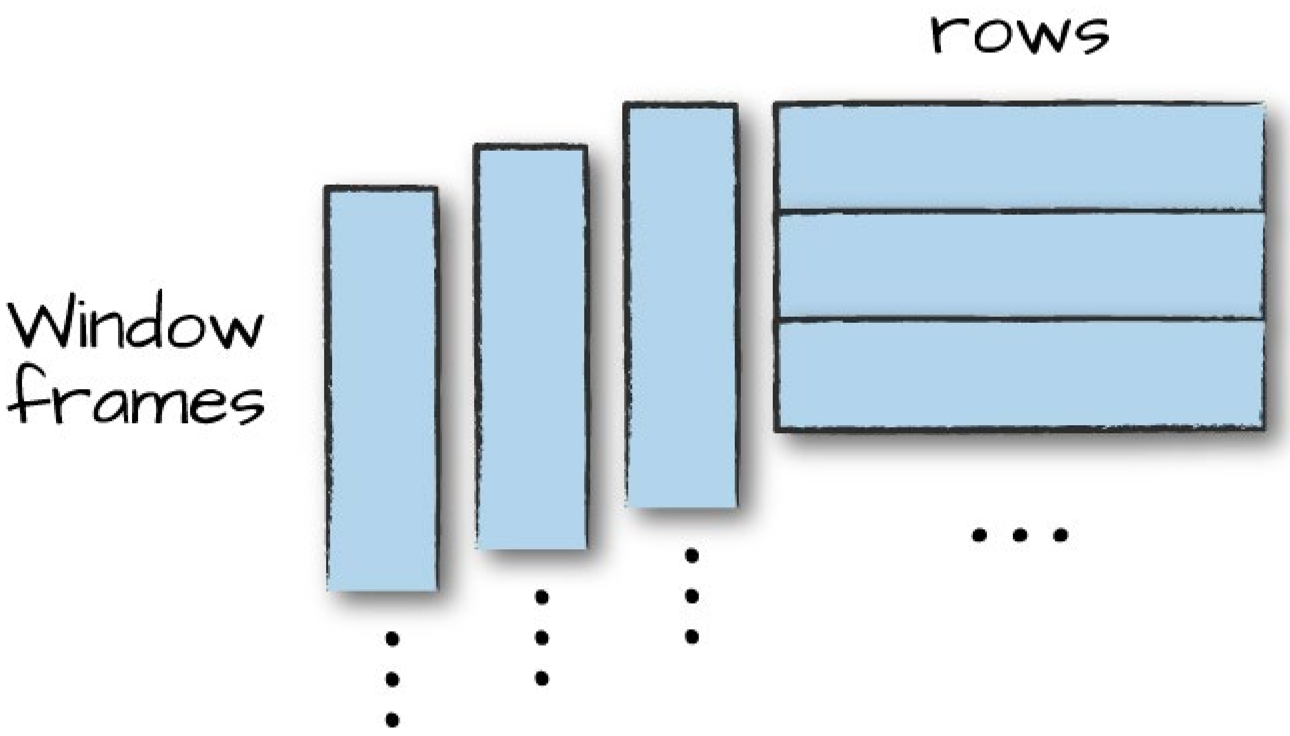
```
// in Scala
df.groupBy("InvoiceNo").agg("Quantity"->"avg", "Quantity"-
>"stddev_pop").show()

# in Python
df.groupBy("InvoiceNo").agg(expr("avg(Quantity)"),expr("stddev_pop(
Quantity)"))\
.show()

-- in SQL
SELECT avg(Quantity), stddev_pop(Quantity), InvoiceNo FROM dfTable
GROUP BY InvoiceNo
```

NYU

Window Functions



Let's add a time component

```
# in Python
from pyspark.sql.functions import col, to_date
dfWithDate = df.withColumn("date", to_date(col("InvoiceDate"),
"MM/d/yyyy H:mm"))
dfWithDate.createOrReplaceTempView("dfWithDate")
```

The first step to a window function is to create a window specification

```
# in Python
from pyspark.sql.window import Window
from pyspark.sql.functions import desc
windowSpec = Window\
      .partitionBy("CustomerId", "date")\
      .orderBy(desc("Quantity"))\
      .rowsBetween(Window.unboundedPreceding, Window.currentRow)
```

For example, if we want to use an aggregation function to learn more about each specific customer

```
# in Python
from pyspark.sql.functions import max
maxPurchaseQuantity = max(col("Quantity")).over(windowSpec)
```

Advanced Groupings

Grouping Sets

Rollups

Cube

Grouping Metadata

Pivot

NYU

Pivot

convert a row into a column

```
// in Scala
val pivoted =
dfWithDate.groupBy("date").pivot("Country").sum()

# in Python
pivoted = dfWithDate.groupBy("date").pivot("Country").sum()
```

Joins

Join Types

- Inner joins (keep rows with keys that exist in the left and right datasets)
- Outer joins (keep rows with keys in either the left or right datasets)
- Left outer joins (keep rows with keys in the left dataset)
- Right outer joins (keep rows with keys in the right dataset)
- Left semi joins (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)
- Left anti joins (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)
- Natural joins (perform a join by implicitly matching the columns between the two datasets with the same names)
- Cross (or Cartesian) joins (match every row in the left dataset with every row in the right dataset)

Create sample data

```
# in Python
person = spark.createDataFrame([ (0, "Bill Chambers", 0, [100]),
(1, "Matei Zaharia", 1, [500, 250, 100]), (2, "Michael Armbrust",
1, [250, 100])])\ .toDF("id", "name", "graduate_program",
"spark_status")

graduateProgram = spark.createDataFrame([ (0, "Masters", "School of
Information", "UC Berkeley"), (2, "Masters", "EECS", "UC
Berkeley"), (1, "Ph.D.", "EECS", "UC Berkeley")])\ .toDF("id",
"degree", "department", "school")

sparkStatus = spark.createDataFrame([ (500, "Vice President"),
(250, "PMC Member"), (100, "Contributor")])\ .toDF("id", "status")
```

Register them to use in SQL

```
person.createOrReplaceTempView("person")
graduateProgram.createOrReplaceTempView("graduateProgram")
sparkStatus.createOrReplaceTempView("sparkStatus")
```

Join Condition for the examples

```
# in Python
joinExpression = person["graduate_program"] ==
graduateProgram['id']
```

Inner Joins (default join)

```
person.join(graduateProgram, joinExpression).show()

-- in SQL SELECT * FROM person JOIN graduateProgram ON
person.graduate_program = graduateProgram.id
```

Outer Joins

```
joinType = "outer“
person.join(graduateProgram, joinExpression, joinType).show()

-- in SQL SELECT * FROM person FULL OUTER JOIN graduateProgram ON
graduate_program = graduateProgram.id
```

Left Outer Joins

```
joinType = "left_outer“
graduateProgram.join(person, joinExpression, joinType).show()

-- in SQL
SELECT * FROM graduateProgram LEFT OUTER JOIN person ON
person.graduate_program = graduateProgram.id
```

Right Outer Joins

```
joinType = "right_outer“
person.join(graduateProgram, joinExpression, joinType).show()

-- in SQL SELECT * FROM person RIGHT OUTER JOIN graduateProgram
ON person.graduate_program = graduateProgram.id
```

Left Semi Joins

Semi joins do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. If the value does exist, those rows will be kept in the result

```
joinType = "left_semi"graduateProgram.join(person,
joinExpression, joinType).show()
```

Left Anti Joins

Keep only the values that do not have a corresponding key in the second DataFrame

```
joinType = "left_anti" graduateProgram.join(person,
joinExpression, joinType).show()

-- in SQL SELECT * FROM graduateProgram LEFT ANTI JOIN person ON
graduateProgram.id = person.graduate_program
```

NYU

Natural Joins

Natural joins make implicit guesses at the columns on which you would like to join.

Implicit is always dangerous!

DataFrames/tables share a column name (`id`), but it may means different things…

for example:

```
-- in SQL
SELECT * FROM graduateProgram NATURAL JOIN person
```

Cross (Cartesian) Joins

Cross joins will join every single row in the left DataFrame to ever single row in the right DataFrame.

```
joinType = "cross"
graduateProgram.join(person, joinExpression, joinType).show()

-- in SQL
SELECT * FROM graduateProgram CROSS JOIN person ON
graduateProgram.id = person.graduate_program
```

Joins on Complex Types

```
# in Python
from pyspark.sql.functions import expr
person.withColumnRenamed("id", "personId")\
.join(sparkStatus, expr("array_contains(spark_status, id)"))\
.show()

-- in SQL
SELECT * FROM (select id as personId, name, graduate_program,
spark_status FROM person) INNER JOIN sparkStatus ON
array_contains(spark_status, id)
```

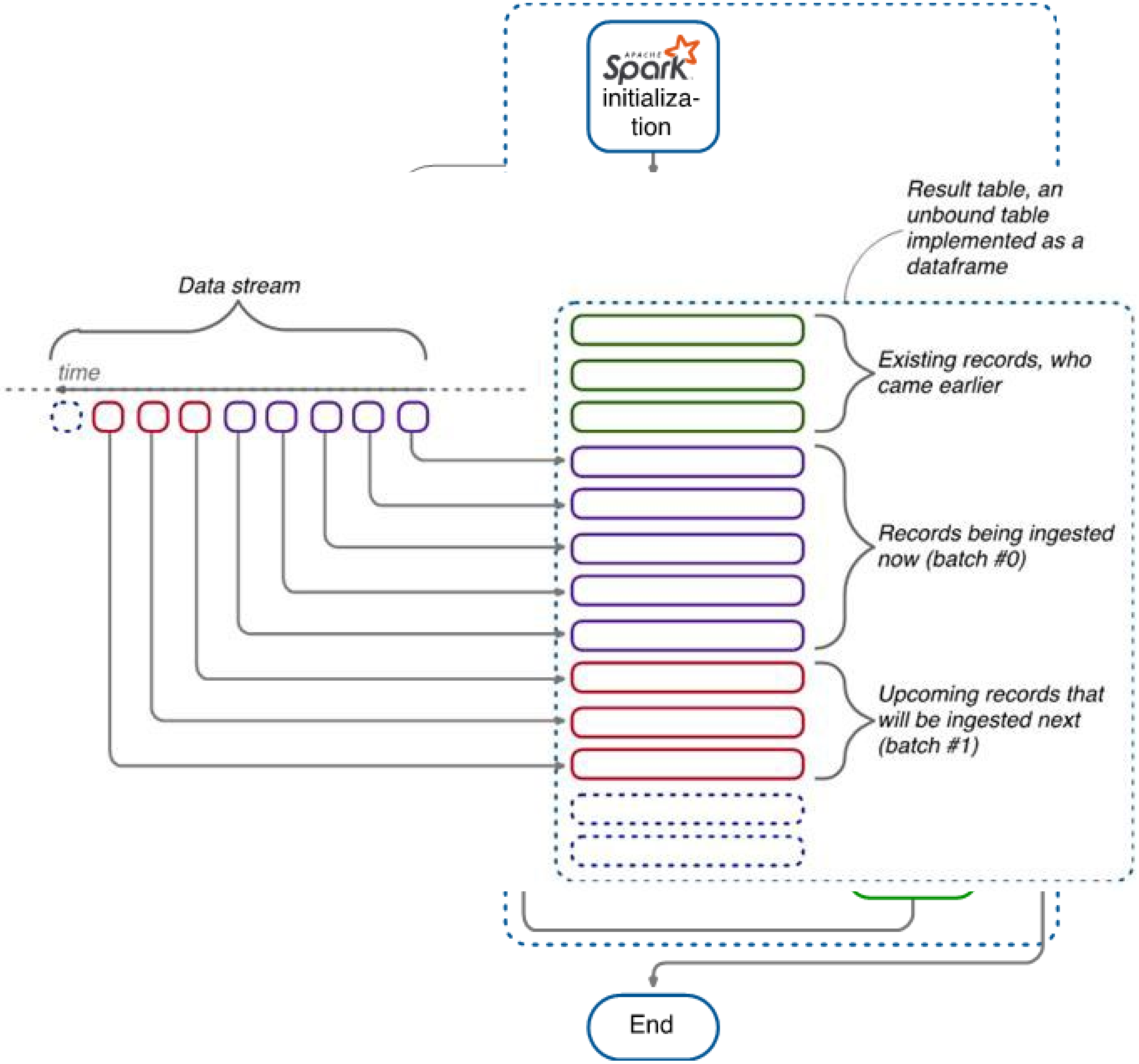
map() vs mapPartitions()

map()
applies a function to each row in a DataFrame/Dataset and returns the new transformed `Dataset`.

`mapPartitions()`
provides a facility to do heavy initializations (for example Database connection) once for each partition instead of doing it on every DataFrame row

NYU

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