Spark DataSet / Data Transform :→

- Combining Data Frame
- Aggregation and Summerizing
- Applying function and Built in transform
- Built-in and Column-level Transformation
- Creating using → UDFs

Spark (DF) → Dataset [ROW]

→ Each Row in Dataframe

Single Record → Object type is [ROW]

Mostly we dont directly use it

But we have 3 cases we have to

Work with individual Spark Transformation

Collecting DF rows to the Driver

Manually Creating Row and Dataframe.

Mostly used in Developing or Unit testing.

```
Just now (<1s)

from pyspark.sql import SparkSession, DataFrame
from pyspark.sql.functions import *
from pyspark.sql.types import *

def to_date_df(df:DataFrame,formate: str, field:str) -> DataFrame:
    return df.withColumn(field, to_date(col(field), formate))
```

```
my_schema = StructType([
    StructField("ID", StringType()),
    StructField("EventDate", StringType())
])

my_rows = [Row("123", "04/05/2020"), Row("124", "4/5/2020"), Row("125", "04/5/2020"), Row("126", "4/05/2020")]
my_rdd = spark.sparkContext.parallelize(my_rows, 2)
my_df = spark.createDataFrame(my_rdd, my_schema)

my_df.printSchema()
my_df.show()
new_df = to_date_df(my_df, "M/d/y", "EventDate")
new_df.printSchema()
new_df.show()
```

```
root
 |-- ID: string (nullable = true)
 |-- EventDate: string (nullable = true)

+---+----------+
| ID| EventDate|
+---+----------+
|123|04/05/2020|
|124|  4/5/2020|
|125| 04/5/2020|
|126| 4/05/2020|
+---+----------+

root
 |-- ID: string (nullable = true)
 |-- EventDate: date (nullable = true)

+---+----------+
| ID| EventDate|
+---+----------+
|123|2020-04-05|
```

Spark → Comes with Bunch of Transformation

agg (*exprs)

cov (col1, col2)

Crosstab (col1, col2)

Cube (*cols)

filter (condition)

groupBy (*cols)

Join (other, on=None, how=None)

OrderBy (*cols, **kwargs)

replace (to_replace, value, subset)

rollup (*cols)

Select (*cols)

Sort (*cols, **kwargs

where (condition)

withColumn (colName, col)

avg (*cols)

max (*cols)

mean (*cols)

Sum (*cols)

→ when DF is not Columnar Struct → first we need to Create → Columnar Format → Then Perform Transformation function

For this we need ROW

For Example we have Unstructure Data

→ log File

```
83.149.9.216 - - [17/May/2015:10:05:03 +0000] "GET /presentations/logstash-monitorama-2013/images/kibana-search.png
83.149.9.216 - - [17/May/2015:10:05:43 +0000] "GET /presentations/logstash-monitorama-2013/images/kibana-dashboard3
83.149.9.216 - - [17/May/2015:10:05:47 +0000] "GET /presentations/logstash-monitorama-2013/plugin/highlight/highlig
83.149.9.216 - - [17/May/2015:10:05:12 +0000] "GET /presentations/logstash-monitorama-2013/plugin/zoom-js/zoom.js H
83.149.9.216 - - [17/May/2015:10:05:07 +0000] "GET /presentations/logstash-monitorama-2013/plugin/notes/notes.js HT
83.149.9.216 - - [17/May/2015:10:05:34 +0000] "GET /presentations/logstash-monitorama-2013/images/sad-medic.png HTT
83.149.9.216 - - [17/May/2015:10:05:57 +0000] "GET /presentations/logstash-monitorama-2013/css/fonts/Roboto-Bold.tt
83.149.9.216 - - [17/May/2015:10:05:50 +0000] "GET /presentations/logstash-monitorama-2013/css/fonts/Roboto-Regular
```

log Dump

If you Read this file in Dataframe → get Row of String.

we Cannot have Column as its unstructured DF

```
file_df = spark.read.text('D:\\SPARK\\pythonProject\\SparkSQL\\data\\apache_logs.txt')
file_df.printSchema()
```

→ this will String.

```
log_regex = r'^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+) (\S+)" (\d{3}) (\S+) "(\S+)" "([^"]*)'

log_df = file_df.select(regexp_extract(str: 'value', log_regex, idx: 1).alias('ip'),
                        regexp_extract(str: 'value', log_regex, idx: 4).alias('date'),
                        regexp_extract(str: 'value', log_regex, idx: 6).alias('request'),
                        regexp_extract(str: 'value', log_regex, idx: 10).alias('referrer'))
```

function

```
log_df.printSchema()
log_df \
    .withColumn(colName: "referrer", substring_index(str: "referrer", delim: "/", count: 3)) \
    .groupBy("referrer") \
    .count() \
    .show(n: 100, truncate=False)
```

```
root
 |-- ip: string (nullable = true)
 |-- date: string (nullable = true)
 |-- request: string (nullable = true)
 |-- referrer: string (nullable = true)
```

```
+-----------------------------------+-----+
|referrer                           |count|
+-----------------------------------+-----+
|http://ijavascript.cn              |1    |
|http://www.google.co.tz            |1    |
|http://www.google.ca               |6    |
```

Working with DF Columns :→

```
%fs head /databricks-datasets/airlines/part-00000
```

```
[Truncated to first 65536 bytes]
Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTime,UniqueCarrier,FlightNum,TailNum,ActualElapsedT
n,Dest,Distance,TaxiIn,TaxiOut,Cancelled,CancellationCode,Diverted,CarrierDelay,WeatherDelay,NASDelay,SecurityDelay,
1987,10,14,3,741,730,912,849,PS,1451,NA,91,79,NA,23,11,SAN,SFO,447,NA,NA,0,NA,0,NA,NA,NA,NA,NA,YES,YES
1987,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,-1,SAN,SFO,447,NA,NA,0,NA,0,NA,NA,NA,NA,NA,YES,NO
```

↓

After this Creating Dataframe.

```
airlinedf = spark.read \
    .format("csv") \
    .option("header","true") \
    .option('inferSchema','true') \
    .option('samplingRation','0.0001') \
    .load('/databricks-datasets/airlines/part-00000')
```

Infering Schema with small Sampling ratio

↓

helps to run quickly.

↓

We how have DF to show learn Column Expression

↓

Spark Dataframe Columns

↳ object of types Column

— We Cannot Manipulate Independently

— It get by Spark Transformation only.

↓

There are 2 ways to Do Transform

↓

Column String

↓

Simplest way to access Column.

↓

Column Object.

↓

It is Complex

↓

```
Just now (1s)
airlinedf.select('Origin','Dest','Distance').show(10)
(1) Spark Jobs
+------+----+--------+
|Origin|Dest|Distance|
+------+----+--------+
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
+------+----+--------+
```

As there is many ways to create Column object.

Most common Method to use

Column()

Col()

```
from pyspark.sql.functions import *

airlinedf.select(column("Origin"),col("Dest"),airlinesDF.Distance).show(10)
```

3 types 3 way.

How to Creat Column Expression :→

SparkDF → (2) way to create Column Expression

String or SQL

Column object Expression

For Example

year, Month, Date

3 Column ?

I want to combine this make as Date format

```
Last execution failed
1  airlinedf.select("Origin","Dest","Distance",to_date(concat(Year,Month,DayofMonth),'yyyyMMdd') as FlightDate).show(10)

› AnalysisException: [UNRESOLVED_COLUMN.WITH_SUGGESTION] A column or function parameter with name `to_date(concat(Year,Month,DayofMonth),'yyyyMMdd') as FlightDate`
cannot be resolved. Did you mean one of the following? [`DayofMonth`, `LateAircraftDelay`, `CancellationCode`, `WeatherDelay`, `ActualElapsedTime`].;
'Project [Origin#104, Dest#105, Distance#106, `to_date(concat(Year,Month,DayofMonth),'yyyyMMdd') as FlightDate]...
```

This will throughs Error

To_date (Concate(year, Month, DayofMonth)

As Select ( ) → only use

Column String

Column Object.

For this we need `expr( )`

`expr ( To_date ( concat (year, month, Day ) , 'yyyyMMdd') )`

This will Return Column Object

This is all String or SQL Expression

```
airlinedf.select("Origin","Dest","Distance",
                 expr("to_date(concat(Year,Month,DayofMonth),'yyyyMMdd') as FlightDate") ).show(10)
```

(1) Spark Jobs

Origin	Dest	Distance	FlightDate
SAN	SFO	447	1987-10-14
SAN	SFO	447	1987-10-15
SAN	SFO	447	1987-10-17
SAN	SFO	447	1987-10-18
SAN	SFO	447	1987-10-19
SAN	SFO	447	1987-10-21
SAN	SFO	447	1987-10-22
SAN	SFO	447	1987-10-23


```
airlinedf.select("Origin","Dest","Distance",
                 expr("to_date(concat(Year,Month,DayofMonth),'yyyyMMdd') as FlightDate") ).show(10)
```

(1) Spark Jobs

Origin	Dest	Distance	FlightDate
SAN	SFO	447	1987-10-14
SAN	SFO	447	1987-10-15
SAN	SFO	447	1987-10-17
SAN	SFO	447	1987-10-18
SAN	SFO	447	1987-10-19
SAN	SFO	447	1987-10-21
SAN	SFO	447	1987-10-22
SAN	SFO	447	1987-10-23

Column Object

```
airlinedf.select("Origin","Dest","Distance",
                 to_date(concat("Year","Month", "DayofMonth"),'yyyyMMdd').alias("FlightDate")
                 ).show(10)
```

(1) Spark Jobs

Origin	Dest	Distance	FlightDate
SAN	SFO	447	1987-10-14
SAN	SFO	447	1987-10-15
SAN	SFO	447	1987-10-17
SAN	SFO	447	1987-10-18
SAN	SFO	447	1987-10-19
SAN	SFO	447	1987-10-21
SAN	SFO	447	1987-10-22
SAN	SFO	447	1987-10-23
SAN	SFO	447	1987-10-24
SAN	SFO	447	1987-10-25

we need to to refer three → Dataframe

Column

Built in function

Either use these

or use SQL like Express

Create User Define function :→

```
if __name__ == "__main__":
    spark = SparkSession.builder \
            .appName('UDFFunction') \
            .master('local[2]') \
            .getOrCreate()

    logger = Log4J(spark)

    survey_df = spark.read \
                .option(key: "header", value: "true") \
                .option(key: "inferSchema", value: "true") \
                .csv('D:\\SPARK\\pythonProject\\SparkSQL\\data\\survey.csv')

    survey_df.show(10)
```

```
+------+
|Gender|
+------+
|Female|
|     M|
|  Male|
|  Male|
|  Male|
|  Male|
|Female|
```

How to make Standard

1 - Male
2 - Female
3 - Unknown

For this Create Custom function.

```
def parse_gender(gender):
    female_pattern = r"^f$|f.m|w.m"
    male_pattern = r"^m$|ma|m.l"

    if re.search(female_pattern, gender.lower()):
        return "Female"
    elif re.search(male_pattern, gender.lower()):
        return "Male"
    else:
        return "Unknown"
```

↳ How to use them :→ we have to Create Expression and use it

we have 2 approach → to develop the Expression.

Column object Expression

String Expression.

Column Object Transformation

- withColumn() :→ Allows to transform with single column.
  - without impacting other DF

```
survey_df2 = survey_df.withColumn(colName: "Gender", parse_gender("Gender"))
```

- Need to register my custom function
  - to use it
    - in UDF
    - UDF()

UDF (Parse gender , StringType())

- default is StringType and it optional

Column Object Expression

```
parse_gender_udf = udf(parse_gender, StringType())
survey_df2 = survey_df.withColumn(colName: "Gender", parse_gender_udf("Gender"))
```

UDF

This register as Dataframe UDF()

- It will not register in Catlog
- It will create → UDF → Serialized in Executor.

```
spark.udf.register(name: "parse_gender_udf",parse_gender, StringType())
```

SQL Expression

This will register in Catlog.

Act as SQL function.

```python
data_list = [
    ("Ravi", "28","1" ,"2020"),
    ("avi", "26","1" ,"81"),
    ("Sony", "27","12" ,"6"),
    ("lg", "8","12" ,"220"),
]

raw_df = spark.createDataFrame(data_list).toDF("name","day","month","year")
raw_df.printSchema()
```

This issue we can fix it by 4 digit ~~problem~~

To add Unique Identifier.

It will Unique across all Identifier

```python
df1= raw_df.withColumn("id", monotonically_increasing_id())
df1.show()
```

✓ 2.9s

```
+----+---+-----+----+-----------+
|name|day|month|year|         id|
+----+---+-----+----+-----------+
|Ravi| 28|    1|2020|          0|
|Sony| 27|   12|   6|          1|
| avi| 26|    1|  81| 8589934592|
|  lg|  8|   12| 220|17179869184|
+----+---+-----+----+-----------+
```

If the Column dont Exist it will added the new column.

```python
df2 = df1.withColumn("year" , expr(
    '''
    case when year < 21 then year + 2000
    when year < 100 then year + 1900
    else year
    end
    '''

))
```

✓ 0.1s

```python
df2.show()
```

✓ 1.3s

```
+----+---+-----+------+-----------+
|name|day|month|  year|         id|
+----+---+-----+------+-----------+
|Ravi| 28|    1|  2020|          0|
|Sony| 27|   12|2006.0|          1|
| avi| 26|    1|1981.0| 8589934592|
|  lg|  8|   12|2020.0|17179869184|
+----+---+-----+------+-----------+
```

Putting 0.0 decimal → How to fix this

By Casting ?

This can be done by

- Inline Cast
- Change the schema

```
df3 = df1.withColumn("year" , expr(
    '''
    case when year < 21 then cast(year as int) + 2000
    when year < 100 then cast(year as int) + 1900
    else cast(year as int)
    end
    '''

))
df3.show()
```

✓ 1.3s

```
+----+---+-----+----+-----------+
|name|day|month|year|         id|
+----+---+-----+----+-----------+
|Ravi| 28|    1|2020|          0|
|Sony| 27|   12|2006|          1|
| avi| 26|    1|1981| 8589934592|
|  lg|  8|   12|2020|17179869184|
+----+---+-----+----+-----------+
```

```
df4 =df4.withColumn("year", col("year").cast(IntegerType()))
df4.show()
df4.printSchema()
```

✓ 1.2s

```
+----+---+-----+----+-----------+
|name|day|month|year|         id|
+----+---+-----+----+-----------+
|Ravi| 28|    1|2020|          0|
|Sony| 27|   12|2006|          1|
| avi| 26|    1|1981| 8589934592|
|  lg|  8|   12|2020|17179869184|
+----+---+-----+----+-----------+

root
 |-- name: string (nullable = true)
 |-- day: string (nullable = true)
 |-- month: string (nullable = true)
 |-- year: integer (nullable = true)
 |-- id: long (nullable = false)
```

Adding and Removing Columns and duplicate

```
df8 = df7.withColumn('dob', expr("to_date(concat(day,'/',month,'/',year),'d/M/y')"))
df8.show()
```

✓ 1.2s

```
+------+---+-----+----+-----------+----------+
|  name|day|month|year|         id|       dob|
+------+---+-----+----+-----------+----------+
|  Ravi| 28|    5|2020|          0|2020-05-28|
|milton|  2|    4|2020|          1|2020-04-02|
|   avi| 26|    7|1981| 8589934592|1981-07-26|
|    lg|  8|   12|2020| 8589934593|2020-12-08|
|  DELL| 17|   12|2020| 8589934594|2020-12-17|
|  Sony| 27|    1|2006|17179869184|2006-01-27|
|   Sam| 18|    1|2020|17179869185|2020-01-18|
+------+---+-----+----+-----------+----------+
```

```
df9= df7.withColumn('dob',to_date(expr("concat(day,'/',month,'/', year)"),'d/M/y'))
df9.show()
```

✓ 1.3s

```
+------+---+-----+----+-----------+----------+
|  name|day|month|year|         id|       dob|
+------+---+-----+----+-----------+----------+
|  Ravi| 28|    5|2020|          0|2020-05-28|
|milton|  2|    4|2020|          1|2020-04-02|
|   avi| 26|    7|1981| 8589934592|1981-07-26|
|    lg|  8|   12|2020| 8589934593|2020-12-08|
|  DELL| 17|   12|2020| 8589934594|2020-12-17|
|  Sony| 27|    1|2006|17179869184|2006-01-27|
|   Sam| 18|    1|2020|17179869185|2020-01-18|
+------+---+-----+----+-----------+----------+
```

Drop Column

```
df8 = df8.drop('day','month','year')
df8.show()
```

✓ 1.2s

```
+------+-----------+----------+
|  name|         id|       dob|
+------+-----------+----------+
|  Ravi|          0|2020-05-28|
|  DELL|          1|2020-12-17|
|  Sony|          2|2006-01-27|
|   avi| 8589934592|1981-07-26|
|    lg| 8589934593|2020-12-08|
|  Sony| 8589934594|2006-01-27|
|  Sony|17179869184|2006-01-27|
|   Sam|17179869185|2020-01-18|
|milton|17179869186|2020-04-02|
+------+-----------+----------+
```

Drop Duplicate

```
df8 = df8.dropDuplicates(['name','dob'])
df8.show()
```

✓ 1.5s

```
+------+-----------+----------+
|  name|         id|       dob|
+------+-----------+----------+
|  DELL|          1|2020-12-17|
|  Sony|          2|2006-01-27|
|  Ravi|          0|2020-05-28|
|   avi| 8589934592|1981-07-26|
|    lg| 8589934593|2020-12-08|
|   Sam|17179869185|2020-01-18|
|milton|17179869186|2020-04-02|
+------+-----------+----------+
```

Short Desc

```
df8 = df8.sort(expr('dob desc'))
df8.show()
```

✓ 1.4s

```
+------+-----------+----------+
|  name|         id|       dob|
+------+-----------+----------+
|   avi| 8589934592|1981-07-26|
|  Sony|          2|2006-01-27|
|   Sam|17179869185|2020-01-18|
|milton|17179869186|2020-04-02|
|  Ravi|          0|2020-05-28|
|    lg| 8589934593|2020-12-08|
|  DELL|          1|2020-12-17|
+------+-----------+----------+
```

Aggregation :→

All aggregation → in Spark are implemented by built in function

It Classified into (3)

- Simple Aggregation
  - avg()
  - count()
  - max()
  - min()
  - sum()
- Grouping Aggregation
- Windowing Aggregation
  - lead()
  - lag()
  - rank()
  - dense_rank()
  - cume_dist()

We can use them as Column Object Expression.

```
invoice_df.select(count("*").alias("total_invoice"),
                  sum("Quantity").alias("TotalQuantity"),
                  avg("UnitPrice").alias("Avg_UnitPrice"),
                  count_distinct("InvoiceNo").alias("CountDistinct")
                  ).show()
```

Result

```
+-------------+-------------+----------------+-------------+
|total_invoice|TotalQuantity|    Avg_UnitPrice|CountDistinct|
+-------------+-------------+----------------+-------------+
|       541909|      5176450|4.611113626088481|        25900|
+-------------+-------------+----------------+-------------+
```

We can also use a SQL Expression

instead of Select → SelectExpr()

```
invoice_df.selectExpr(
    *expr: "count(1) as count 1",
    "count(StockCode) as `count field`",
    "sum(Quantity) as TotalQuantity",
    "avg(UnitPrice) as AvgPrice"
).show()
```

count(*) = count(1)

this will not count null value

```
>>>> SQL Expression <<<<
+-------+-----------+-------------+----------------+
|count 1|count field|TotalQuantity|        AvgPrice|
+-------+-----------+-------------+----------------+
| 541909|     541908|      5176450|4.611113626086849|
+-------+-----------+-------------+----------------+
```

Grouping Aggregation :->

By using SQL Query ↴

```
invoice_df.createOrReplaceTempView("sales")
summary_Sql = spark.sql(
    '''
    SELECT Country, InvoiceNo,
        SUM(Quantity),
        ROUND(SUM(Quantity*UnitPrice),2) as InvoiceValue
    FROM sales
    GROUP BY 1,2
    '''
)
```

SQL Spark Text

Country	InvoiceNo	sum(Quantity)	InvoiceValue
United Kingdom	536446	329	440.89
United Kingdom	536508	216	155.52
United Kingdom	537018	-3	0.0
United Kingdom	537401	-24	0.0
United Kingdom	537811	74	268.86
United Kingdom	C537824	-2	-14.9
United Kingdom	538895	370	247.38

We Can do the Same for grouping → Mechanism. in df ↴

DataFrame Grouping

```
summary_df = invoice_df.groupBy("Country","InvoiceNo") \
            .agg(func.sum("Quantity").alias("Total_quantity"),
                 func.round(func.sum(func.expr("Quantity*UnitPrice")),2).alias("InvoiceValue")
            )

summary_df.show()
```

✓ 0.8s

Country	InvoiceNo	Total_quantity	InvoiceValue
United Kingdom	536446	329	440.89
United Kingdom	536508	216	155.52
United Kingdom	537018	-3	0.0
United Kingdom	537401	-24	0.0
United Kingdom	537811	74	268.86

```
invoi_up = invoice_df.groupBy('Country',
        func.weekofyear(func.to_timestamp(col='InvoiceDate',format='dd-MM-yyyy H.mm')).alias("WeekNumbe
        .agg(func.count_distinct('InvoiceNo').alias('NumInvoices'),
        func.sum('Quantity').alias('TotalQuantity'),
        func.round(func.sum(func.expr("Quantity*UnitPrice")),scale=2).alias('Invoice_Value'))
invoi_up.show()
```

✓ 1.5s Python

Country	WeekNumber	NumInvoices	TotalQuantity	Invoice_Value
EIRE	17	1	163	362.9
Norway	47	1	594	2016.78
Channel Islands	46	1	78	211.63
France	10	4	471	998.49

Example 2 →

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
EIRE	48	7	2822	3147.23
EIRE	49	5	1280	3284.1
EIRE	50	5	1184	2321.78
EIRE	51	5	95	276.84

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
France	48	4	1299	2808.16
France	49	9	2303	4527.01
France	50	6	529	537.32
France	51	5	847	1702.87

1st → Partitioning by Country first

2nd → Order by Weeknumber.

Dataframe Joins :→

Its all about → bringing together 2DF

Combine 2DF together by 2 things

Join Condition / Expression

Join Type:

Join type = "inner"

Join_Expr = Order_df.Prod_id == Product_df.prod_id

Order_df.Join(Product_renamed_df, Join_expr, "inner")

```
order_df.join(product_renamed_df,join_expr,"left") \
    .drop(product_renamed_df.prod_id) \
    .select("order_id","prod_id","prod_name","unit_price","list_price","qty") \
    .sort("order_id") \
    .show()
```

✓ 1.7s

order_id	prod_id	prod_name	unit_price	list_price	qty
01	07	32 GB Flash Storage	320	320	2
01	02	Optical Mouse	350	350	1
01	04	Wireless Keyboard	580	580	1
02	03	Wireless Mouse	450	450	1
02	06	16 GB Flash Storage	220	240	1
03	01	Scroll Mouse	195	250	1
04	09	NULL	270	NULL	3
04	08	64 GB Flash Storage	410	430	2
05	02	Optical Mouse	350	350	1

```
order_df.join(product_renamed_df,join_expr,"left") \
    .drop(product_renamed_df.prod_id) \
    .select("order_id","prod_id","prod_name","unit_price","list_price","qty") \
    .withColumn("prod_name", func.expr("coalesce(prod_name,prod_id)")) \
    .withColumn("list_price",func.expr("coalesce(list_price,unit_price)")) \
    .show()
```

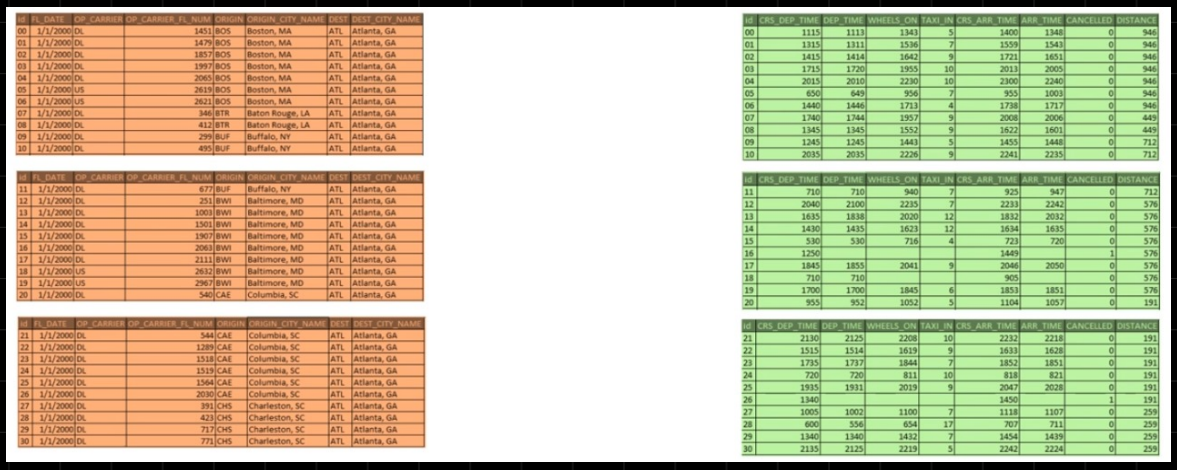
✓ 1.7s

order_id	prod_id	prod_name	unit_price	list_price	qty
01	07	32 GB Flash Storage	320	320	2
02	03	Wireless Mouse	450	450	1
01	02	Optical Mouse	350	350	1
01	04	Wireless Keyboard	580	580	1
03	01	Scroll Mouse	195	250	1
04	09	09	270	270	3
04	08	64 GB Flash Storage	410	430	2
05	02	Optical Mouse	350	350	1
02	06	16 GB Flash Storage	220	240	1

Spark offers two approaches to Join
- Shuffle Join
- Broadcast Join.

Shuffle Join

- most commonly used in
- Implimentation Done by Hadoop.
  - MapReduce kind of



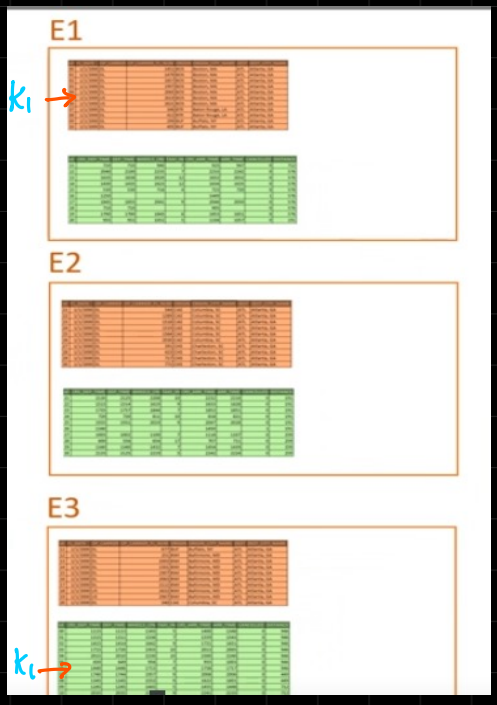
3 Partition → distribute across different Executor

If $K_1 = E1$ $K_1 = E3$

→ How to Perform Join its not Possible

Becoz keys are in different Executor

→ It need to be in same Executor



So Join can be Performed in 2 Stages

1st → Each Executor Map those records using Join Key → Send it to the Exchange → (Map Phase)

Map Exchange

This Map Exchange be available to be Picked up by **Spark Framework**

Spark Framework will Pick this record

Send it to the **Reduce Exchange**

going to Collect all the record for the Same Key.

For Example

→ All the record with Same Key → 0-1

Will goto Same Exchange.

This Exchange Known as **Shuffle Partion**

we need to decide how many Shuffle Partition needed.

Shuffle Partition

=

Number of Nodes

So Each Partition handle by one Executor

So all Partition Process Parallely.

Now Data got transferred from → Map Exchange → To Reduce Exchange

do Shuffle operation

↳ It Can block network

↳ Slow down Join operation

After this → this Record Combines

And Create new Dataframe Partition

For this we need to use Sort Merge Join -algo.

```
if __name__ == "__main__":

    spark = SparkSession.builder \
        .appName("ShuffleJoin") \
        .master("local[3]") \
        .getOrCreate()

    flight_time_df1 = spark.read.json('data/d1')
    flight_time_df2 = spark.read.json('data/d2')

    spark.conf.set("spark.sql.shuffle.partitions",3)
```

local[3] → 3 nodes

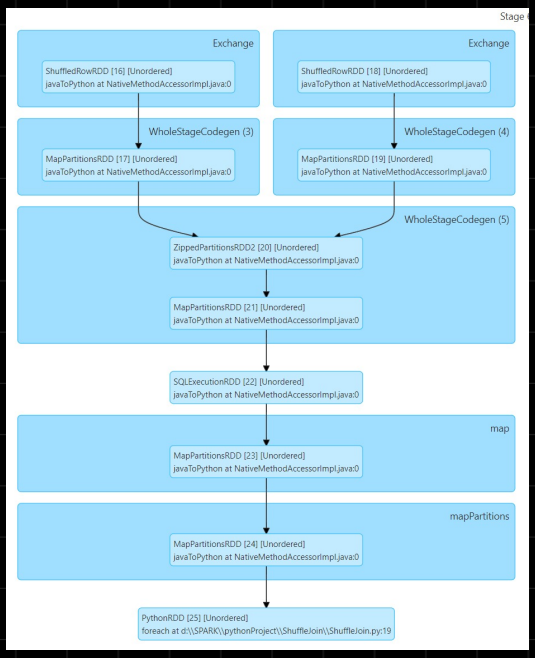
3 Partition

```
join_expr = flight_time_df1.id == flight_time_df2.id
join_df = flight_time_df1.join(flight_time_df2, join_expr,"inner")

join_df.foreach(lambda f :  None)
```

this is transformation

↳ It will only activate when we take an → action



Joining Dataframe

are of 2 type

→ Large to Large

→ Large to Small

↓

Tiny → or fixed with Single Executor

Large - Large

↳ filter Join → Perform Almost all aggregation

↳ filtering out necessary data.

global_online_sales

order_id	city	product_code	store_code
S001	New York	ABX-512	abc.com
S002	London	GCY-213	abc.com

us_warehouse_events

order_id	city	warehouse_code	event_type	event_date
S001	New York	NY516	order placed	15-Jan-2020
S001	New York	NY516	dispatched	16-Jan-2020

This not needed

↓

So filter As we are only looking for (us warehousing only

look for all Possible opportunities to Reduce dataframe Size

↓

Then perform Joins

(European market don't will not help.

→ look for → { Number Shuffle Partition
Number of Executors }

↓

helps with Parallelism of my Joins.

What maximum Possible Parallelism

↓

Executor = 500
Shuffle Partition = 400
Unique Key = 200

↓

If I have 200 Unique Key then I will have Only 200 Shuffle Partition other 200 will remain Blank.

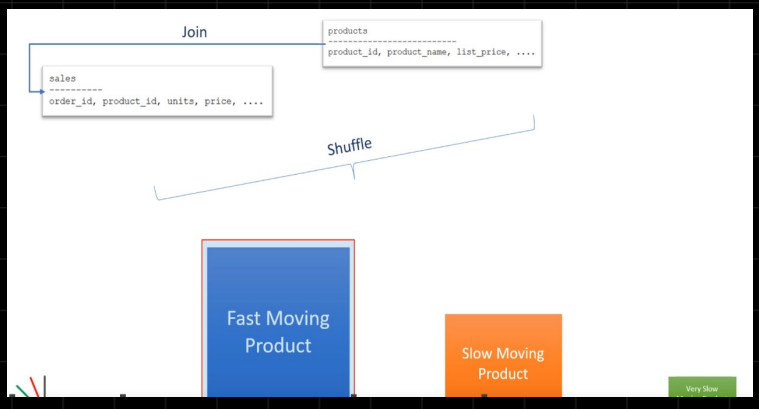
Even you have (500 Nodes 400 Shuffle)

↓

It will make Only 200.

→ So → In Unique Key → limiting Scalability

need to increase Join Keys

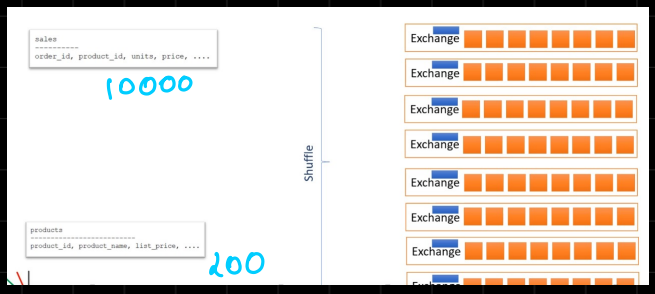


Some are slow moving
Some are fast moving

So better partition

Large – Large Join → Shuffle Joins

Large - Small Join → Broadcast Joins

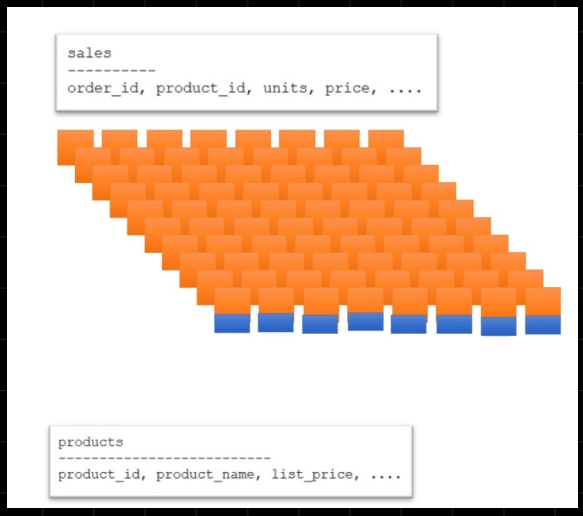


Shuffle Join

So for this it will create

200 K Reduce Exchange

Shuffle the 10000 data to this its huge take more bandwidth

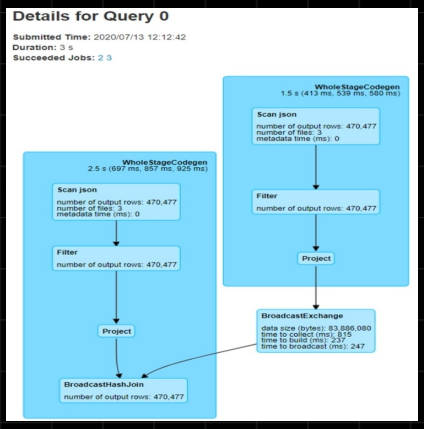


Here 200 Shuffle to 10000 as its small and it perform faster than other

Broadcast Join

It Eleminate the need of Shuffle

Perform a Join with out Shuffle




```
join_expr = flight_time_df1.id == flight_time_df2.id
join_df = flight_time_df1.join(func.broadcast(flight_time_df2), join_expr,"inner")
```

Implementing Bucketing :->

Even after doing Brodcast Join → Java Memory
Shuffle Join → not Enough out of memory.

If we know we are going to Join 2 Dataset
Bucket ← then advice to use Bucket

Bucketing Dataset also need Shuffle however that needed Only once

It Can be done → into 2 steps
- Bucket Dataset

What is a best Performance ?
- its not possible as getting Ideal Performance
- Predictable Performance