WIZELINE
ACADEMY

Grow your career:
Free courses in Artificial Intelligence, Software Development, User Experience and More

WIFI: WizelineAcademy
Password: academyGDL
Slack Channel:

@WizelineAcademy

academy.wizeline.com

/WizelineAcademy

Get notified about courses:
tinyurl.com/WL-academy

Week 2 - Session 2/2

# Big Data Engineering with Spark

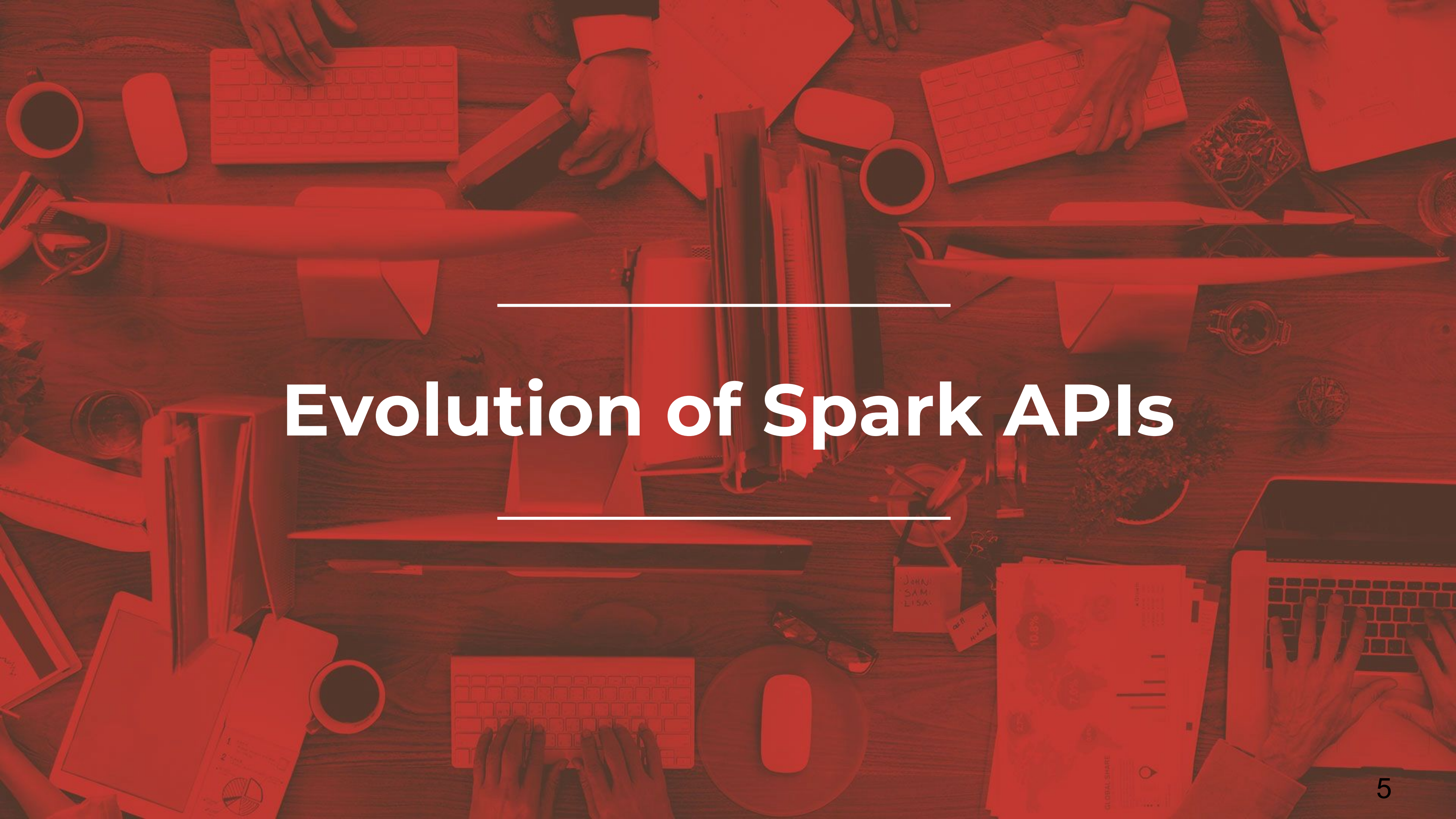
## RDDs, Dataframes and Datasets

# In this episode...

- Unstructured versus Structured Data
- Dataframes
- Datasets

# By the end of this session, you'll be able to explain:

- **Differences between unstructured and structured data**
- **How to use Dataframes**
- **How to use Datasets**
- **When to use Resilient Distributed Datasets (RDDs), Dataframes or Datasets**



# Evolution of Spark APIs

# Architecture

## Evolution of Spark APIs

**RDD**
*Resilient Distributed Dataset*
2011

→

**DataFrame**
2013

→

**Dataset**
2015

### RDD

Distributed collection of JVM objects

Functional operators (map, filter, reduce, etc.)

### DataFrame

Distributed collection of Row objects

Expression-based operations and User Defined Functions (UDFs)

Logical plans and optimizer

Fast/efficient internal data representations

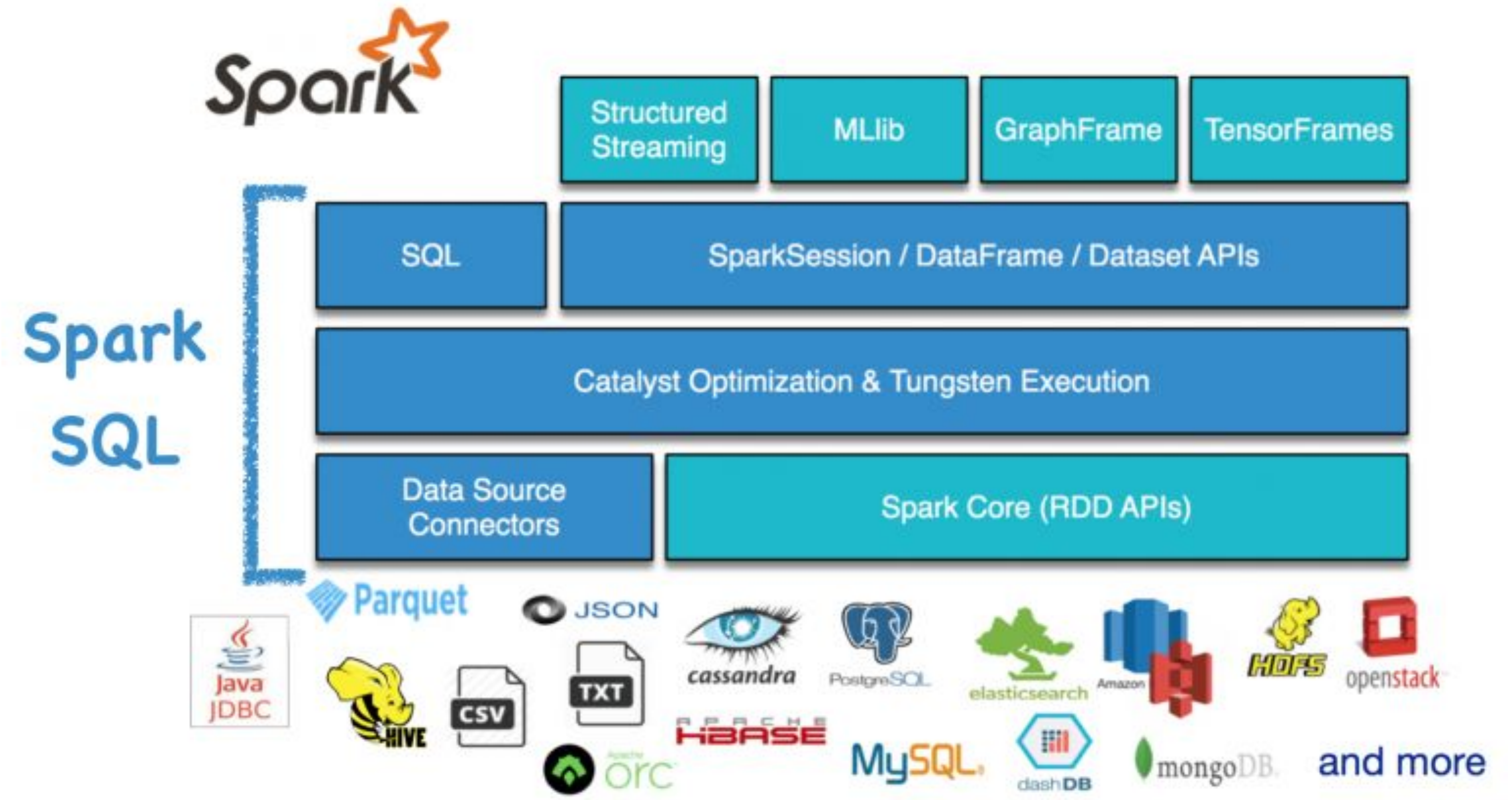
### Dataset

Internally rows, externally JVM objects

Almost "the best of both worlds": **type-safe + fast**

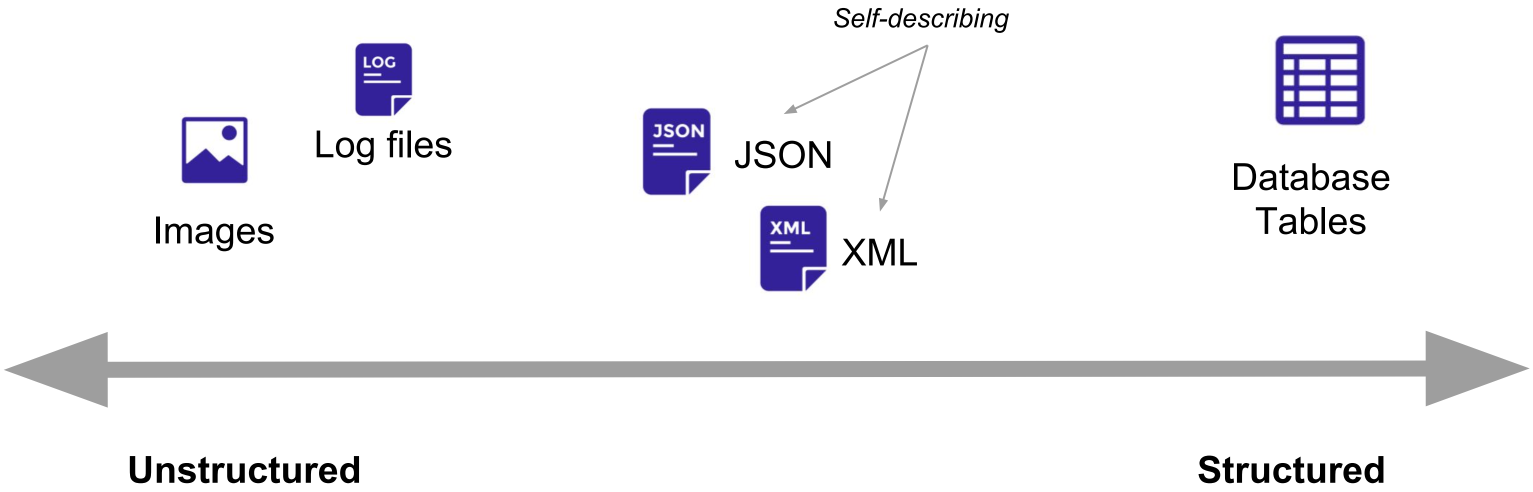
Slower than DataFrames, not as good for interactive data analysis, especially Python

# Spark SQL, DataFrames and Datasets



# Structured versus Unstructured Data

# Structured versus Unstructured Data



# Unstructured Data on Spark

## RDDs

RDDs are not aware of the *schema* of the data.

```
case class Flight(
  DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String, count: Integer
)
                                                              RDD[Flight]
```

| DEST_COUNTRY_NAME | ORIGIN_COUNTRY_NAME | count |
|---|---|---|
| United States | Romania | 1 |
| United States | Ireland | 264 |
| United States | India | 69 |
| Egypt | United States | 24 |
| Equatorial Guinea | United States | 1 |
| United States | Singapore | 25 |
| United States | Grenada | 54 |
| Costa Rica | United States | 477 |
| Senegal | United States | 29 |
| United States | Marshall Islands | 44 |

only showing top 10 rows

Spark knows that RDDs may be parameterized with arbitrary types (objects), but it does not know about the structure of the types.

***Pros: More flexibility with data types.***

***Cons: Optimization is hard without knowledge about the structure.***

# Unstructured Data Computation on Spark

## RDDs

Functional transformations are applied to RDDs.

User-defined **function literals** are used in higher-order functions like **map**, **flatMap** and **filter**.

```
rdd.map((x, y) => x+y)
```

***Pros: More flexibility with operations.***

***Cons: Optimization is hard or impossible to be done automatically.***

# Structured Data

Data is organized into one or more tables (relations).

Tables contain columns (attributes) and rows (records or tuples).

Tables typically represent a collection of objects of a certain type, such as customers or products.

Similar to relational database tables.

| DEST_COUNTRY_NAME | ORIGIN_COUNTRY_NAME | count |
|---|---|---|
| United States | Romania | 1 |
| United States | Ireland | 264 |
| United States | India | 69 |
| Egypt | United States | 24 |
| Equatorial Guinea | United States | 1 |
| United States | Singapore | 25 |
| Costa Rica | United States | 477 |

# Structured Data Computation

Computations can be more rigid,

think about:

```
SELECT

WHERE
ORDER BY
GROUP BY

COUNT
```

| DEST_COUNTRY_NAME | ORIGIN_COUNTRY_NAME | count |
|---|---|---|
| United States | Romania | 1 |
| United States | Ireland | 264 |
| United States | India | 69 |
| Egypt | United States | 24 |
| Equatorial Guinea | United States | 1 |
| United States | Singapore | 25 |
| Costa Rica | United States | 477 |

***Pros: Optimization is easier to be done automatically with structured data.***

***Cons: Less flexibility with computations and data types.***

# Optimizations (Dataframes and Datasets)

## Catalyst

Query optimizer.

Compiles Spark SQL programs down to RDD operations, which can be reordered to reduce the amount of data that must be read and prune unneeded partitioning.

## Tungsten

Off-heap serializer.

Provides highly-specialized data encoders, which are column based and off-heap (avoiding garbage collection overhead).

# Structured vs Unstructured Data Recap

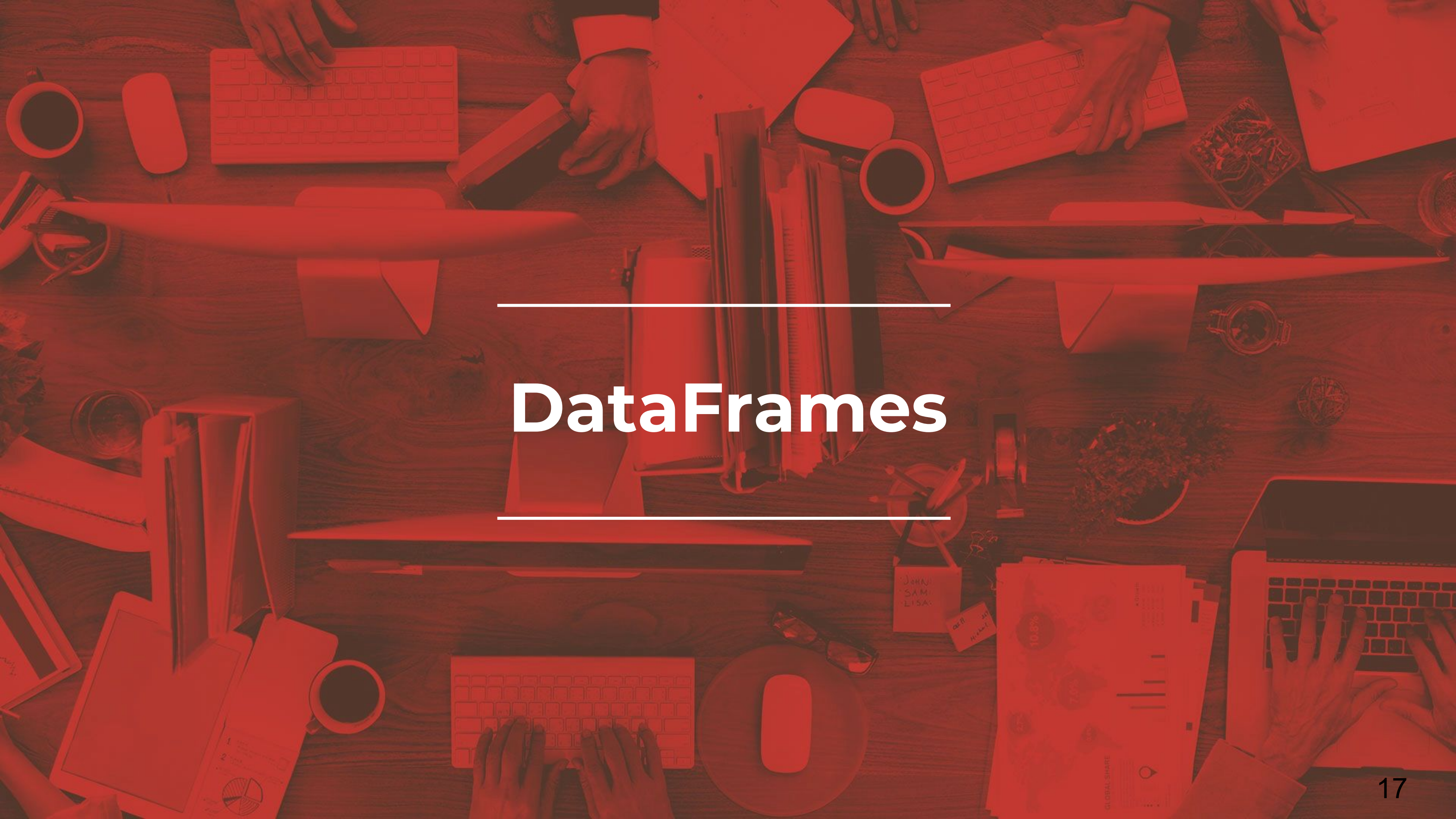
What We've Learned So Far

- RDDs are suitable to process unstructured data. This gives flexibility of computation, but there is no opportunity for automatic optimization.
- Dataframes and Datasets are better to process structured data. Automatic optimizations are performed by Spark, but data and computation are more rigid.

# Q&A



WIZELINE



# DataFrames

# DataFrames

- A DataFrame is a Dataset organized into named columns.
- Conceptually similar to a table in a relational database.
- Optimizations occur under the hood.
- A DataFrame is represented by a Dataset of Rows.
- Have or Require a known schema.
- Albeit, they are **untyped.**
- Transformations on DataFrames are known as **untyped transformations.**

# DataFrames

## Creating DataFrames

**DataFrames** can be created from an existing **RDD with** .toDF

```
val flight_data_RDD = spark.sparkContext.parallelize(Seq(
        ("United States", "Romania", "1"),
        ("United States", "Ireland", "264"),
        ("United States", "Singapore", "25")))

flight_data_RDD: org.apache.spark.rdd.RDD[(String, String, String)] = ParallelCollectionRDD[29] at parallelize
```

```
val flight_data_DF = flight_data_RDD.toDF("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME", "count")

flight_data_DF: org.apache.spark.sql.DataFrame = [DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string ... 1 more field]
```

# DataFrames

## Creating DataFrames

**DataFrames** can be created using `spark.createDataFrame()`

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.{StructField, StringType, IntegerType}
val someData = Seq(
  Row("F3", 3),
  Row("F4", 5),
  Row("F5", 8)
)

val someSchema = List(
  StructField("F", StringType, true),
  StructField("n", IntegerType, true)
)

val someDF = spark.createDataFrame(
  spark.sparkContext.parallelize(someData),
  StructType(someSchema)
)
```

# DataFrames

## Creating DataFrames

**DataFrames** can be created by reading from a **data source**: JSON, CSV, Parquet, and such.

```
val flight_data = spark
  .read
  .option("inferSchema", "true")
  .option("header", "true")
  .csv("gs://de-training-input/flight-data/*.csv")
```

```
flight_data: org.apache.spark.sql.DataFrame = [DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string ...
```

# DataFrame Transformations

Look Similar to SQL

Transformations on `DataFrames` are operations which return a `DataFrame` and their evaluation is lazy.

```
// This import is needed to use the $-notation/
import spark.implicits._
```

```
//Select routes with more than 1000000 flights
flight_data.filter($"count" > 100000).show()
```

```
+----------------+-------------------+------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME| count|
+----------------+-------------------+------+
|    United States|      United States|348113|
|    United States|      United States|370002|
|    United States|      United States|352742|
|    United States|      United States|343132|
|    United States|      United States|347452|
|    United States|      United States|358354|
+----------------+-------------------+------+
```

```
// Print the schema in a tree format
flight_data.printSchema
```

```
root
 |-- DEST_COUNTRY_NAME: string (nullable = true)
 |-- ORIGIN_COUNTRY_NAME: string (nullable = true)
 |-- count: integer (nullable = true)
```

```
// Select only a column
flight_data.select("DEST_COUNTRY_NAME").show(3)
```

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|    United States|
|    United States|
|    United States|
+-----------------+
only showing top 3 rows
```

Other common transformations: `join`, `limit`, `orderBy`, `where`, `as`, `sort`, `union`, `drop`, etc.

# Grouping and Aggregating on Dataframes

One of the Most Common Tasks

## groupBy

Function that returns a **`RelationalGroupedDataset`**

`RelationalGroupedDataset` has several standard aggregation functions, such as `count`, `sum`, `max`, `min` and `avg`.

```
titanic_df.groupBy("Sex").avg("Age").show()

+------+-----------------+
|   Sex|         avg(Age)|
+------+-----------------+
|female|27.91570881226053 7|
|  male| 30.72664459161148|
+------+-----------------+
```

# Dataframe Operations

Example Dissected: `groupBy`, `count`

```
val dataFrameWay = flight_data
  .groupBy("DEST_COUNTRY_NAME")
  .count()
```

$$\begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \ldots \end{pmatrix} \textbf{.groupBy(} \; \textit{column} \; \textbf{)} \begin{pmatrix} (\, \mathbf{x_1}, \text{grouped-data}_1 \,) \\ (\, \mathbf{x_2}, \text{grouped-data}_2 ) \\ \ldots \end{pmatrix} \textbf{.count( )} \begin{pmatrix} (\, \mathbf{x_1}, \textbf{count(}\text{grouped-data}_1\textbf{)} \,) \\ (\, \mathbf{x_2}, \textbf{count(}\text{grouped-data}_1\textbf{)} \,) \\ \ldots \end{pmatrix}$$

$$\mathbf{G_i} \leftarrow \textit{agg-function} \; (\, \text{grouped-data}_i \,)$$
$$\mathbf{G_i} \leftarrow \textbf{count} \; (\, \text{grouped-data}_i \,)$$
$$\ldots$$

## groupBy with agg (Untyped Transformation)

You can also call agg and a `SparkSQL` function, such as: .

```
titanic_df.groupBy("Sex").agg(avg("Age"), avg("Fare")).show()

+------+-----------------+-----------------+
|   Sex|         avg(Age)|        avg(Fare)|
+------+-----------------+-----------------+
|female|27.915708812260537| 44.47981783439487|
|  male| 30.72664459161148|25.523893414211418|
+------+-----------------+-----------------+
```

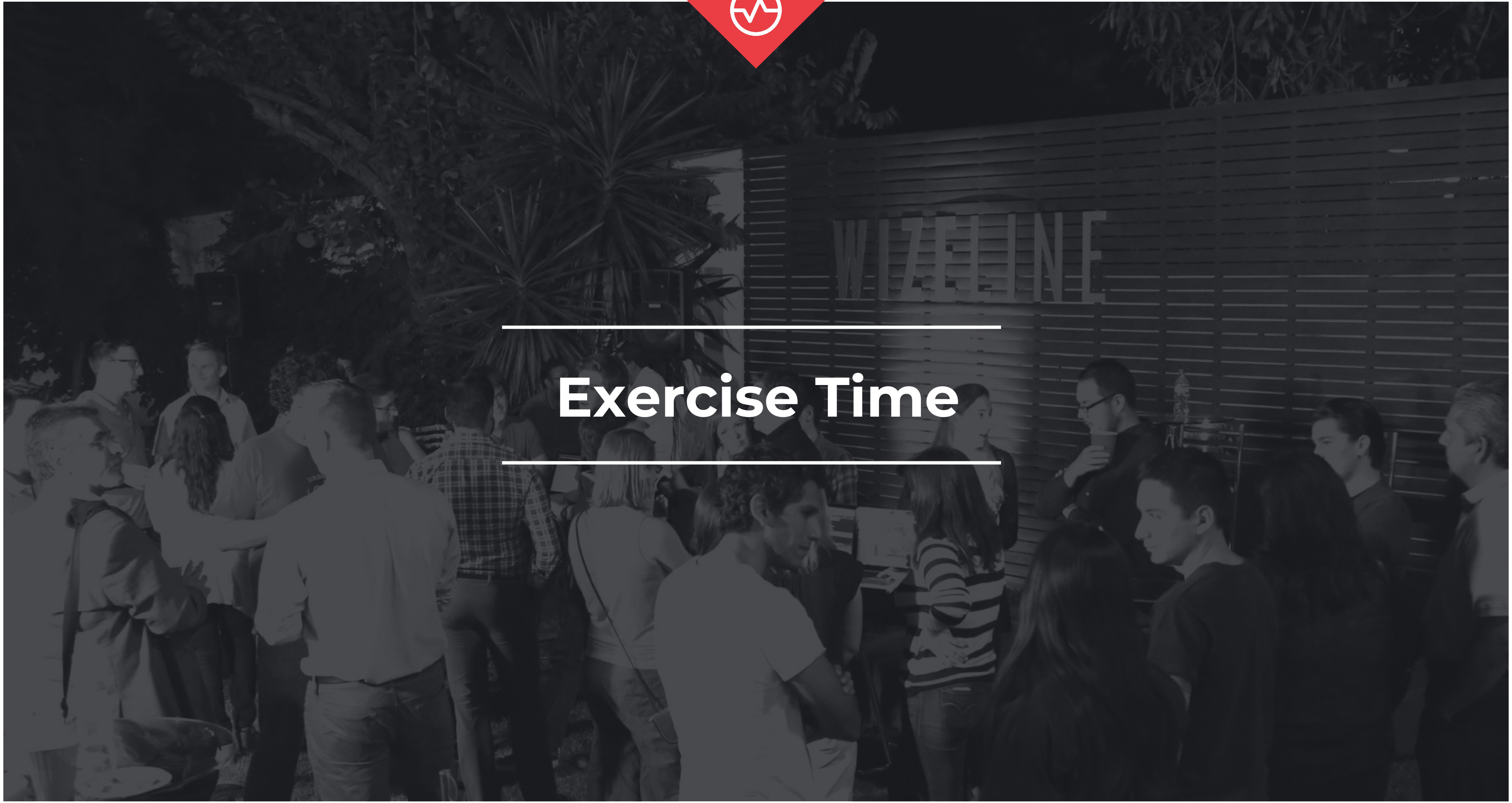
# Dataframes Recap

What We've Learned So Far

- Dataframes have a schema but are untyped.
- You can perform untyped transformations to Dataframes, including aggregations after a `groupBy`.

# Q&A

# 5 minutes




# Exercise Time

*Exercise*

# Grouping and Aggregation on DataFrames

Using the Titanic dataset at gs://de-training-input/titanic/train.csv , **can you calculate the ground truth (number of survivors and deaths) registered on the dataset?**

The expected answer is an object that contains the **survivor** and **death** count columns:

Example:

```
ground_truth: org.apache.spark.sql.DataFrame = [survivors: bigint, deaths: bigint]
+-----------+---------+
| survivors |  deaths  |
+-----------+---------+
|    350     |  620     |
+-----------+---------+
```

***Hint:*** *The reported ground truth on the dataset description is not the answer because we are not using the complete dataset.*

# Titanic Dataset

Dataset Object Creation

Load the dataset

```
val titanic_df = sqlContext.read.format("csv")
  .option("header", "true").option("inferSchema", "true")
  .load("gs://de-training-input/titanic/train.csv")
```

Scala

```
titanic_df = sqlContext.read.format("csv") \
    .option("header", "true").option("inferSchema", "true") \
    .load("gs://de-training-input/titanic/train.csv")
```

Python

Display the Dataframe schema

```
titanic_df.printSchema()
```

# Datasets

# Datasets

## Creating Datasets

**Datasets** can be created in different ways:

- From an existing **RDD** or **DataFrame**.

```scala
import spark.implicits._ // .toDS requires this
val myDS = myRDD.toDS
val myDS = myDF.toDS
```

- From a **data source**: JSON, CSV, Parquet, and such.

```scala
val myDS =
spark.read.json("people.json").as[Person]
```

A Class defines the structure/type

- From common Scala types.

```scala
val myDS = List(
     “Wizeline”, “Academy”, “Spark”).toDS
val myDS = spark.createDataset(
     List(“Wizeline”, “Academy”, “Spark”))
```

# Typed Columns

On **Datasets** *typed* operations usually act on `TypedColumn`.

To create a TypedColumn, call `.as[. . .]` on an *untyped* `Column`.

```
$"price".as[Double]
```

# Dataset Transformations

Typed and Untyped Transformations

## Untyped transformations

Transformations for `DataFrames` are available for `Datasets` too.

## Typed transformations

Typed variants of many `DataFrame` transformations and additional *higher order RDD-like* transformations like `map`, `flatMap`, etc. are also available.

# Common Typed Dataset Transformations

**Filtering**

```
filter(pred: T => Boolean): Dataset[T]
```

**Mapping**

```
map[U](f: T => U): Dataset[U]
```

```
flatMap[U](f: T => TraversableOnce[U]):
Dataset[U]
```

**Distinct**

```
distinct(): Dataset[T]
```

**Grouping**

```
groupByKey[K](f: T => K):
KeyValueGroupedDataset[K, T]
```

# Grouped Operations on Datasets

## KeyValueGroupedDataset

Calling `groupByKey` on a `Dataset` return a **`KeyValueGroupedDataset`**.

**`KeyValueGroupedDataset`** has several aggregation operations that return **`Dataset`** `or` **`KeyValueGroupedDataset`**

# Common KeyValueGroupedDataset Aggregations

## mapValues

```
mapValues[W](func: (V) => W): KeyValueGroupedDataset[K, W]
```

Returns a new KeyValueGroupedDataset where the given function func has been applied to the data.

# Common KeyValueGroupedDataset Aggregations

## reduceGroups

```
reduceGroups(f: (V,V) => V): Dataset[(K, V)]
```

Reduce each group using a given binary function.

The function must be commutative and associative to ensure a deterministic result.

# Common KeyValueGroupedDataset Aggregations

## mapGroups

```
mapGroups[U](f: (K, Iterator[V]) => U): Dataset[U]
```

Applies the given function to each group of data.

The function must be commutative and associative to ensure a deterministic result.

```
val flights_grouped = flight_data_ds
  .groupByKey(_.ORIGIN_COUNTRY_NAME)
  .mapGroups{case(k, iter) => (k, iter.map(x => x.count).toArray)}
  .orderBy("_1")
```

```
flights_grouped: org.apache.spark.sql.Dataset[(String, Array[Int])]
```

# Common KeyValueGroupedDataset Aggregations

## flatMapGroups

```
flatMapGroups[U](f: (K, Iterator[V]) => TraversableOnce[U]):
Dataset[U]
```

Applies the given function to each group of data, flattening the results.

# Using the General agg Operation

## agg

```
agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
```

Computes the given aggregation, returning:

- A Dataset of tuples for each unique key.
- The result of computing this aggregation over all elements in the group.

TypedColumns

```
val flights_ag = flight_data_ds.groupByKey(
        flight => flight.ORIGIN_COUNTRY_NAME
    )
    .agg(
        sum("count").alias("total_flights_from").as[Long],
        max("count").as[Double].alias("max_flights_from").as[Double]
    )
    .orderBy(desc("max_flights_from")).show(10)
```

# Using Aggregator

## Aggregator

A base class for user-defined aggregations, which can be used in Dataset operations to take all of the elements of a group and reduce them to a single value.

```
import org.apache.spark.sql.expressions.Aggregator
import org.apache.spark.sql.{Encoder, Encoders}

case class Data(i: Int)

val customSummer =  new Aggregator[Data, Int, Int] {
  def zero: Int = 0
  def reduce(b: Int, a: Data): Int = b + a.i
  def merge(b1: Int, b2: Int): Int = b1 + b2
  def finish(r: Int): Int = r
  def bufferEncoder: Encoder[Int] = Encoders.scalaInt
  def outputEncoder: Encoder[Int] = Encoders.scalaInt
}.toColumn

val ds = List(Data(3), Data(5), Data(8)).toDS
val aggregated = ds.select(customSummer)
```

Extracts an `int` from a specific class and adds them up.

# Datasets Recap

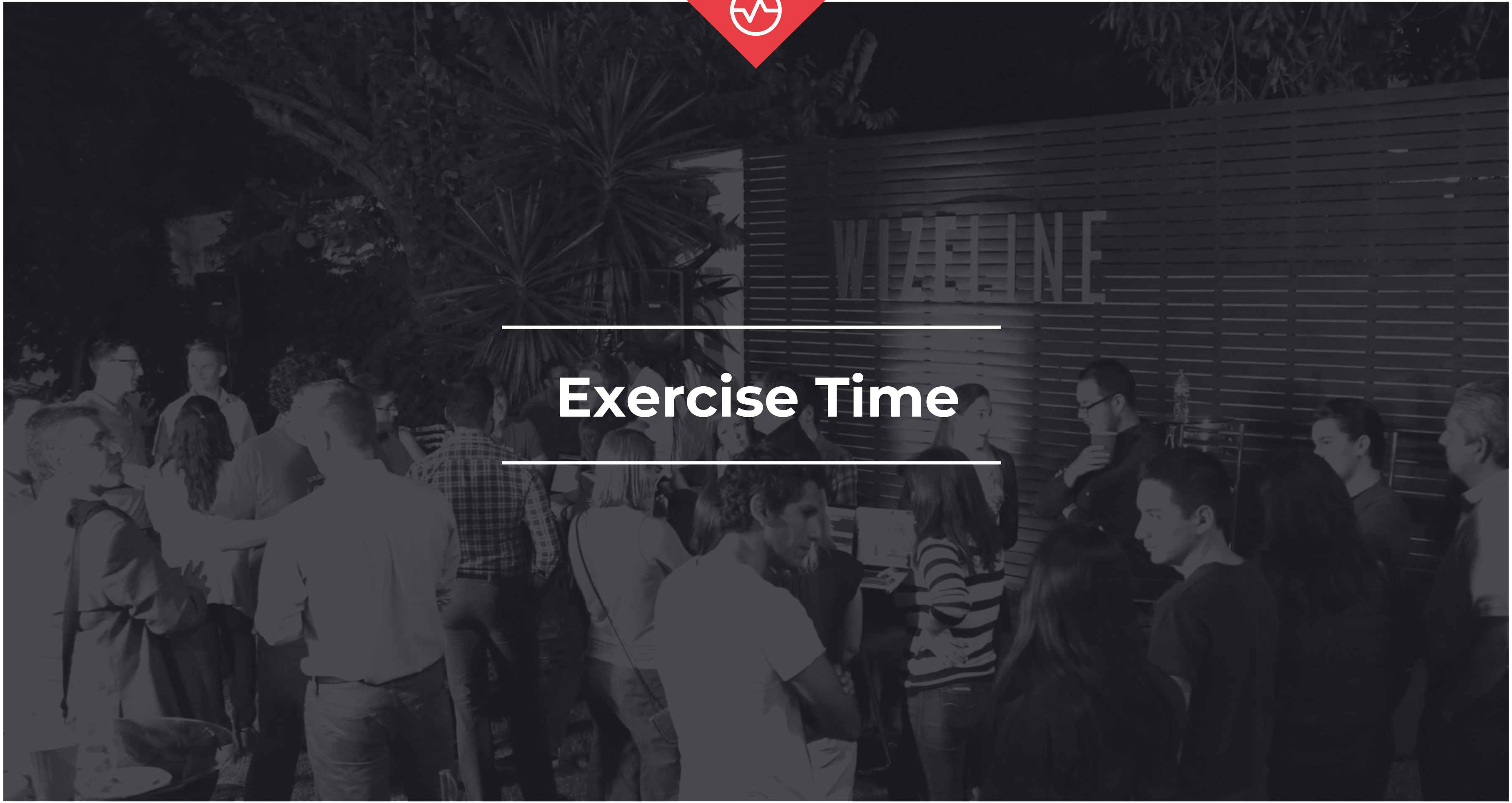
What We've Learned So Far

- Datasets have a schema and are also typed.
- You can perform all the untyped transformation (as in Dataframes) and also the typed transformations.
- `groupByKey` helps you leverage the typing system and perform grouped aggregations.

# Q&A

WIZELINE

# Exercise Time



*Exercise*

# Grouping and Aggregation on Datasets

Using the Flights dataset, **can you calculate the total registered flights per ORIGIN and the number of total destinations for those places?**

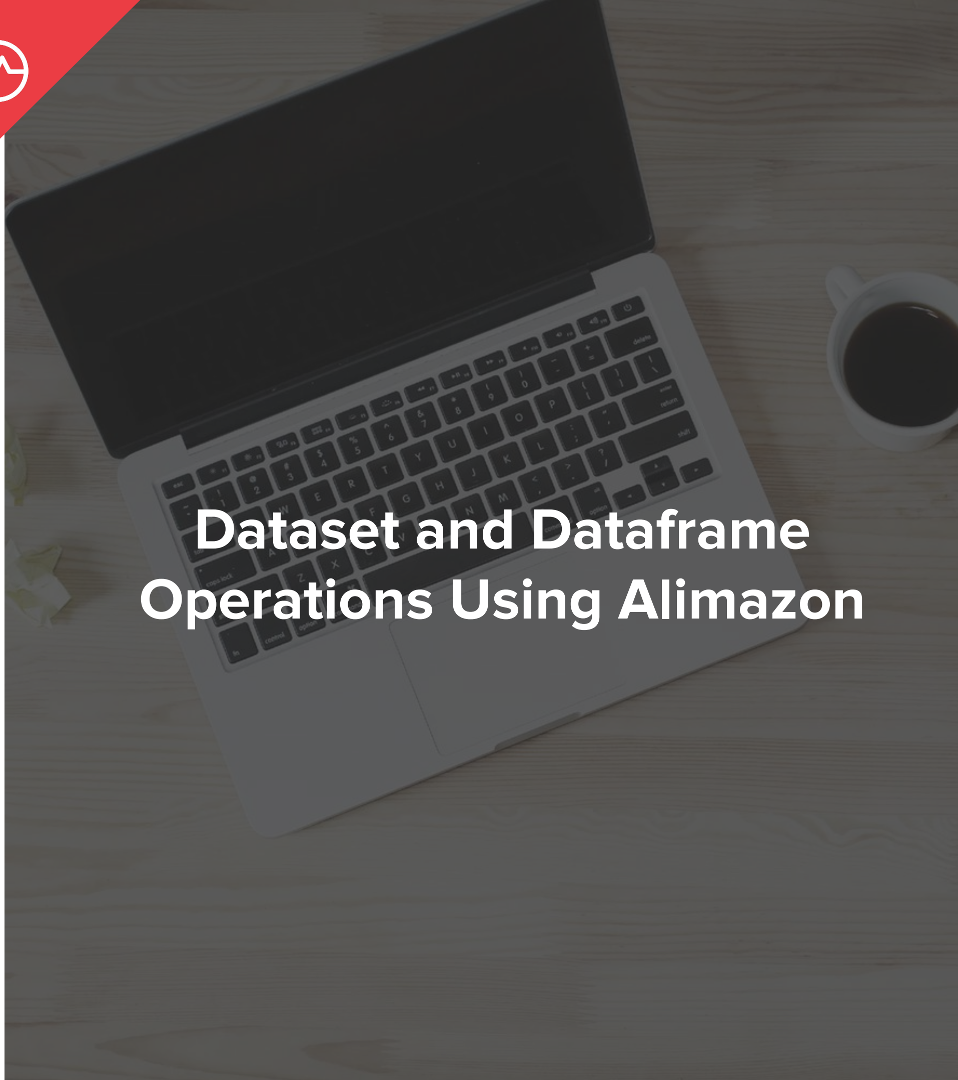
The expected answer is a Dataset that contains the **ORIGIN_COUNTRY_NAME**, **total_flights_from,** and **total_flights_to** columns:

```
flights_ag: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]
```

*Hint: If using Datasets, try using data types different from Int for the typed transformations.*

# Assignment

To Work on Your Own at Home



# Assignment

Dataset and Dataframe Operations

## Using the Alimazon Client Purchase Orders Dataset:

## 1. Best Selling Hours (ID: best_selling_hours)

### Problem Description

Alimazon's Marketing Director has issued a request to identify the "spending behaviour by hour" for all kinds of products and report them sorted in nondecreasing order of gross sales.

In particular, they want to know: the average gross sales as well as the minimum and maximum registered sales of the orders placed by hour. Marketing will be using that information to run campaigns during the identified "peak hours"

### Input Dataset

You can find the input dataset at:

```
gs://de-training-input/alimazon/200000/client-orders/
```

# Assignment

Dataset and Dataframe Operations

## Expected Output Format

- Your final dataset should contain the information described in the problem statement sorted by **average_spending** (in nondecreasing order).
- The hour column goes from 0 to 23.
- Make sure to round the numbers in **average_spending** to two decimal places (you can use the **bround** function to accomplish that; see here for details)

| hour | average_spending | min_spending | max_spending |
|---|---|---|---|

## Output Dataset

You can write your output dataset to your assigned output bucket following this format:

```
gs://de-training-output-<student-name>/assignment-4-best_selling_hours-<attempt>
```

Where `<attempt>` is an increasing integer (beginning from 0) that distinguishes the multiple attempts you tried (you need this because you don't have permission to erase contents from the output bucket).

# Assignment

Dataset and Dataframe Operations

## 2. Monthly Discount (ID: monthly_disscount)

### Problem Description

Every month, Alimazon's marketing team assigns a **10%** discount to the **top ten** products that registered the highest number of sales over the **previous six months,** from the time the discount is calculated.

The team wants to operationalize the report below.

Using the Alimazon Client Purchase Orders Dataset, calculate the new product price on a report which should contain: **product_id, total_products_sold, total_registered_sales, unit_product_price, new_unit_product_price**.

### Input Dataset

You can find the input dataset at:

```
gs://de-training-input/alimazon/200000/client-orders/
```

# Assignment

Dataset and Dataframe Operations

## Expected Output Format

- Your final dataset should contain the information described in the problem statement sorted by **total_products_sold** (in nondecreasing order).
- Make sure to round the numbers in **unit_product_price** and **new_unit_product_price** to two decimal places (you can use the [bround](bround) function)

**| product_id | total_products_sold | total_registered_sales | unit_product_price | new_unit_product_price |**

## Output Dataset

You can write your output dataset to your assigned output bucket following this format:

```
gs://de-training-output-<student-name>/assignment-4-monthly_disscount-<attempt>
```

Where `<attempt>` is an increasing integer (beginning from 0) that distinguishes the multiple attempts that you tried.

# Assignment

Dataset and Dataframe Operations

## 3. Client Orders Distribution (ID: client_orders_dist)

### Problem Description

Also, the sales department at Alimazon is interested in understanding the distribution of **purchases-by-client**, so we were asked to generate that information.

**Note:** The distribution of purchases-by-user is a table that contains the number of clients (**clients_count**) that have bought the same number of products (**products_count**).

### Input Dataset

You can find the input dataset at:

```
gs://de-training-input/alimazon/200000/client-orders/
```

# Assignment

Dataset and Dataframe Operations

## Expected Output Format

- Your final dataset should contain the information described in the problem statement sorted by **clients_count** (in nondecreasing order).

| products_count | clients_count |
|---|---|

## Output Dataset

You can write your output dataset to your assigned output bucket following this format:

```
gs://de-training-output-<student-name>/assignment-4-client_orders_dist-<attempt>
```

Where `<attempt>` is an increasing integer (beginning from 0) that distinguishes the multiple attempts that you tried.

Where Can I Get this Presentation?

# Slack channel PDF

Your feedback is very valuable!

**https://goo.gl/forms/LhMpYc1Pr67dFDbw1**