BTS | Barcelona Technology School

Real Time Data Analysis

Master in Big Data Solutions 2020-2021

Learning Objectives

- Understand the Spark SQL framework
- Execute queries using SQL and Expressions
- Load data from a variety of formats
- Perform basic statistical analysis on data

Instructional Contents

- Introduction
- The limitations of relational databases
- PySpark SQL
- Understanding the data sources
- Structuring semi-structured data with Spark SQL
- Doing basic statistical analysis

MOTIVATION

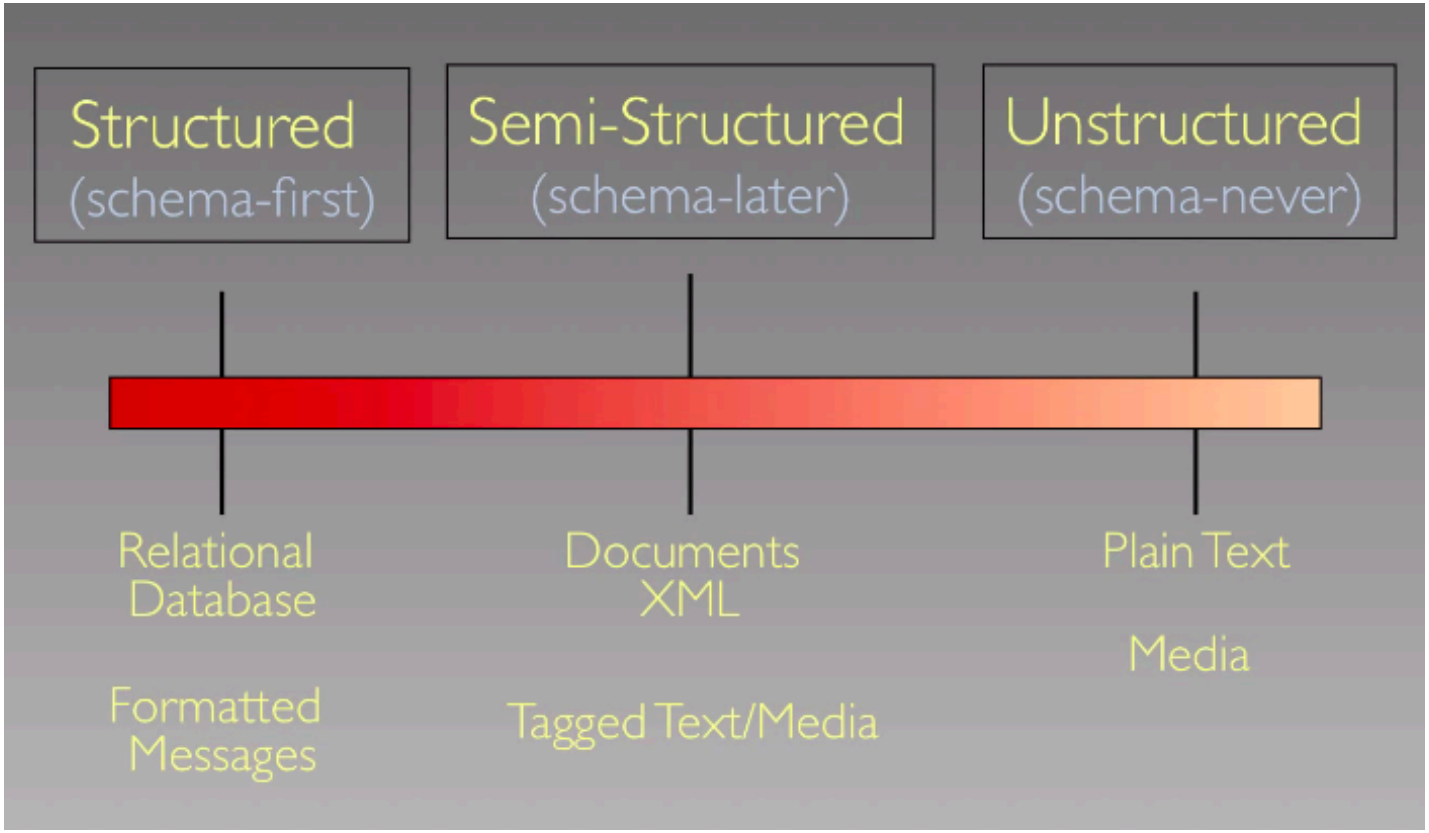
I am extremely grateful to the following for their contribution in shaping my early academic life by teaching me Apache Spark and making me familiar with the cutting edge terminologies in data processing and modelling.

- **Alexandre Perera Lluna**
  - Head, Centre de Recerca en Enginyeria Biomèdica (CREB- UPC)
  - Teaching Courses- Introduction to Bioinformatics and Data Mining for Biomedical Databases

- **Joaquim Gabarro**
  - Associate Professor, UPC (ALBCOM Research Group)
  - Research Interests: Design and Analysis of Parallel Data Structures, Parallel and Concurrent Program Verification.

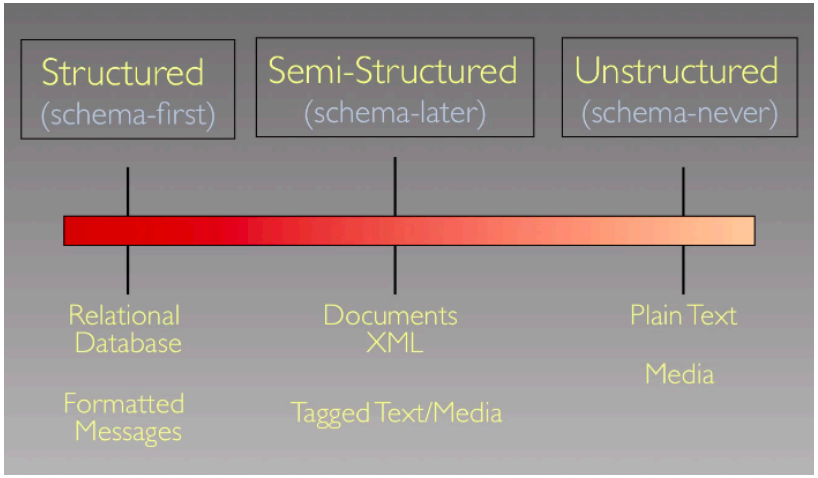
THE STRUCTURE SPECTRUM

Understanding the data structure spectrum in terms of schema



THE STRUCTURE SPECTRUM

Understanding the data structure spectrum in terms of schema



Structured data is the most preferred format of data. It allows least effort storage, querying, analytics, reporting etc.

Relational Database: consists of a set of **Relations**
Relation consists of two parts mainly-

1. **Schema**: It is a description of a particular collection of data using a Data Model. In the case of Structured Data, the schema clearly specifies the name of the relation as well as each column's name and type
   For example,
   Students(sid: string, name: string,
   login: string, age: integer, gpa:real)

2. **Instance**: the actual data at any given time
   - #Rows= Cardinality
   - #Columns= Degree

THE STRUCTURE SPECTRUM

From Structured Data to Structured Query Language (SQL)

SQL: Preferred language for the relational databases; For example. Consider the STUDENT relation discussed previously. SQL can be used to CREATE, MODIFY, DELETE and many such operations as well as queries on the relation

Example: Instance of Students Relation

Students(*sid*:string, *name*:string, *login*:string, *age*:integer, *gpa*:real)

sid	name	login	age	gpa
53666	Jones	jones@eecs	18	3.4
53688	Smith	smith@statistics	18	3.2
53650	Smith	smith@math	19	3.8

- Cardinality = 3 (rows)
- Degree = 5 (columns)
- All rows (tuples) are distinct

SQL offers a significant amount of ease and flexibility in querying the databases. For example, consider the following query which has generated he aforesaid result

- Single-table queries are straightforward
- To find all 18 year old students, we can write:

```
SELECT *
  FROM Students S
 WHERE S.age=18
```

- To find just names and logins:

```
SELECT S.name, S.login
  FROM Students S
 WHERE S.age=18
```

BTS | Barcelona Technology School

The limitations of relational databases

THE LIMITATIONS OF RELATIONAL DATABASES

Why relational databases are not enough?

Most enterprises now in the 21st century are loaded with rich data stores and repositories, and want to take maximum advantage of their ‘Big Data’ for actionable insights.

Relational data stores are easy to build and query.
However, as data starts increasing in volume and variety, the relational approach does not scale well enough for building big data applications and analytical system.

Although, Relational databases might be popular, but they don’t scale very well unless we invest in a proper Big Data management strategy.

This involves thinking about potential data sources, data volume, constraints, schemas, ETL (extract-transform-load), access and querying patterns and much more!

THE LIMITATIONS OF RELATIONAL DATABASES

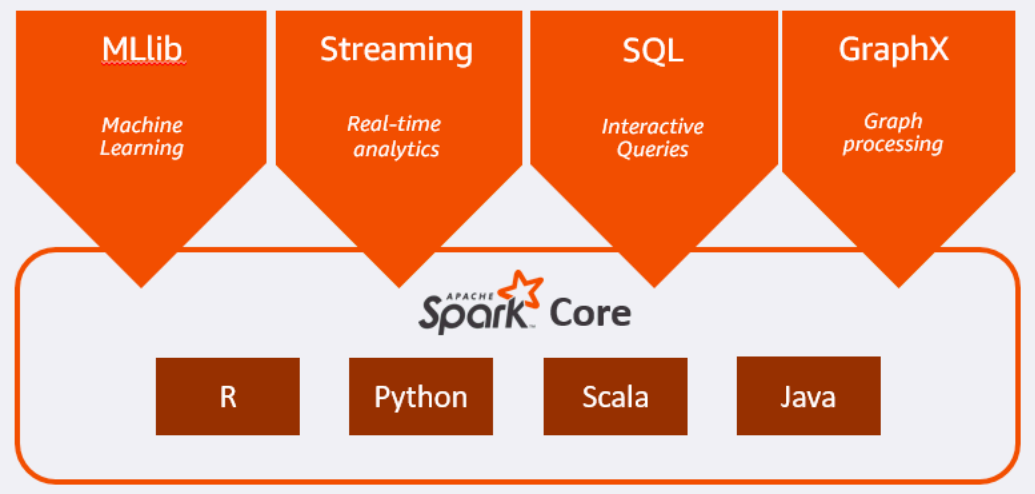
Introducing the Apache Spark Framework

Most enterprises now in the 21st century are loaded with rich data stores and repositories, and want to take maximum advantage of their 'Big Data' for actionable insights.

Relational data stores are easy to build and query.
However, as data starts increasing in volume and variety, the relational approach does not scale well enough for building big data applications and analytical system.

Although, Relational databases might be popular, but they don't scale very well unless we invest in a proper Big Data management strategy.

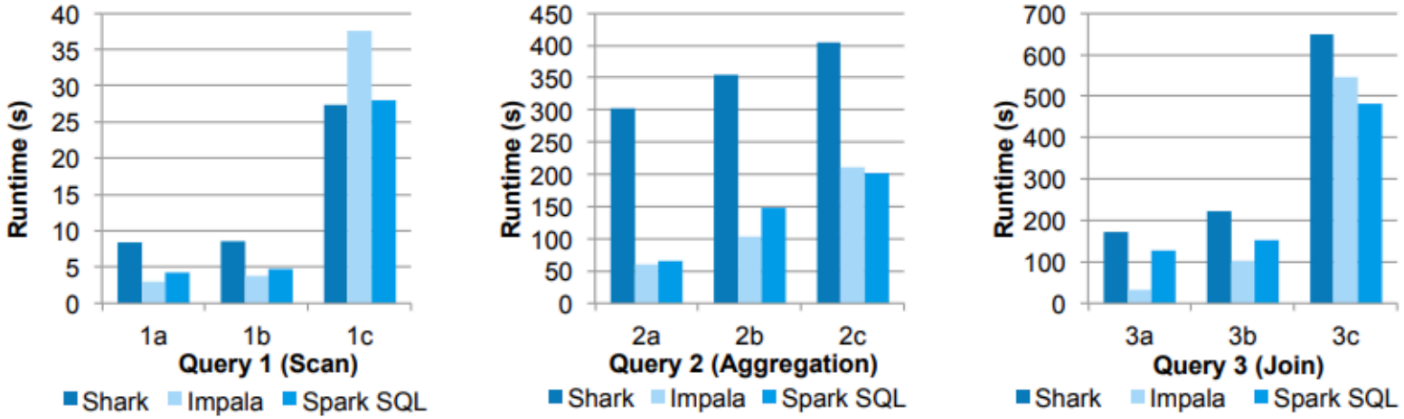
This involves thinking about potential data sources, data volume, constraints, schemas, ETL (extract-transform-load), access and querying patterns and much more!

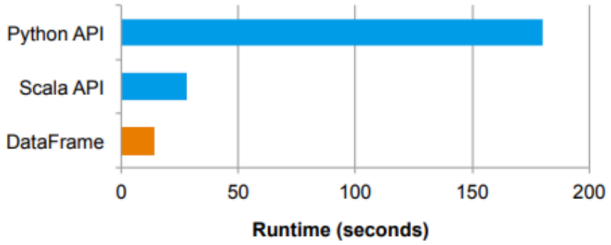


What is Apache Spark?
Blog by Amazon Web Services (AWS)

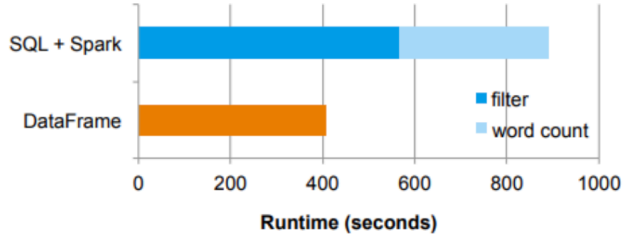
THE LIMITATIONS OF RELATIONAL DATABASES

Some performance benchmarking





Performance of an aggregation written using the native Spark Python and Scala APIs versus the DataFrame API.

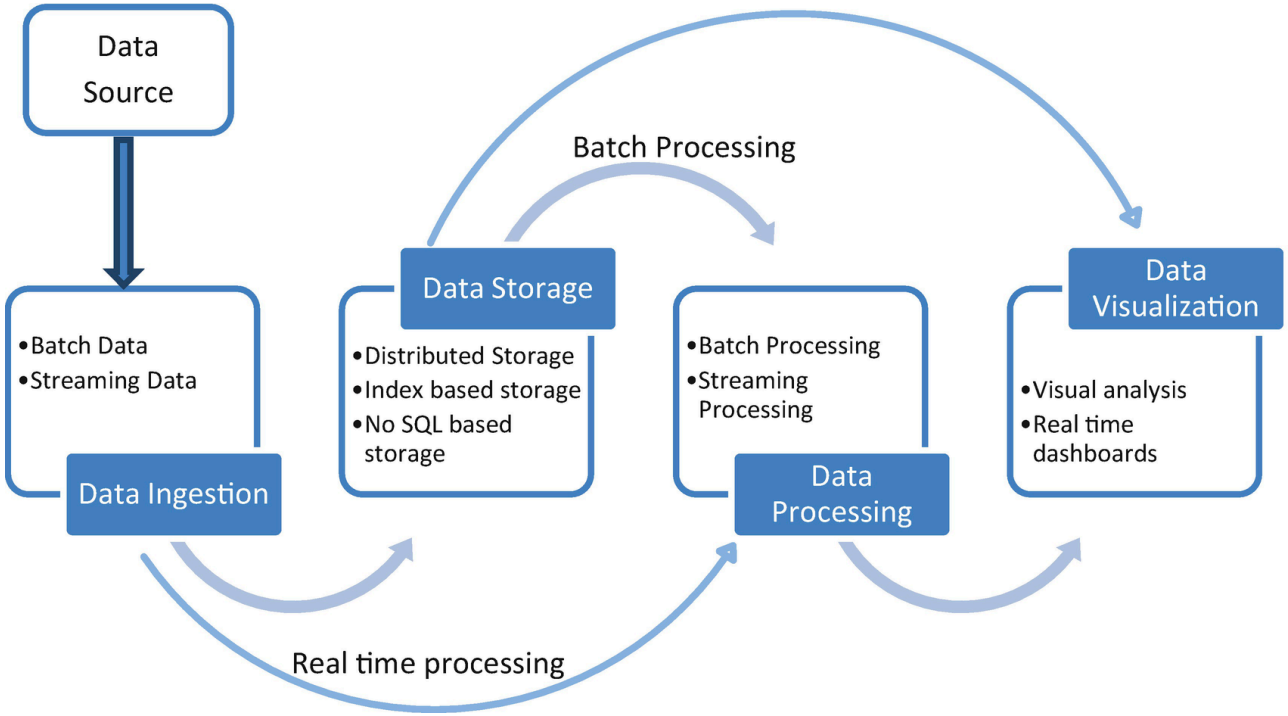


Performance of a two-stage pipeline written as a separate Spark SQL query and Spark job (above) and an integrated DataFrame job (below).

1. SQL at Scale with Apache Spark SQL and DataFrames here

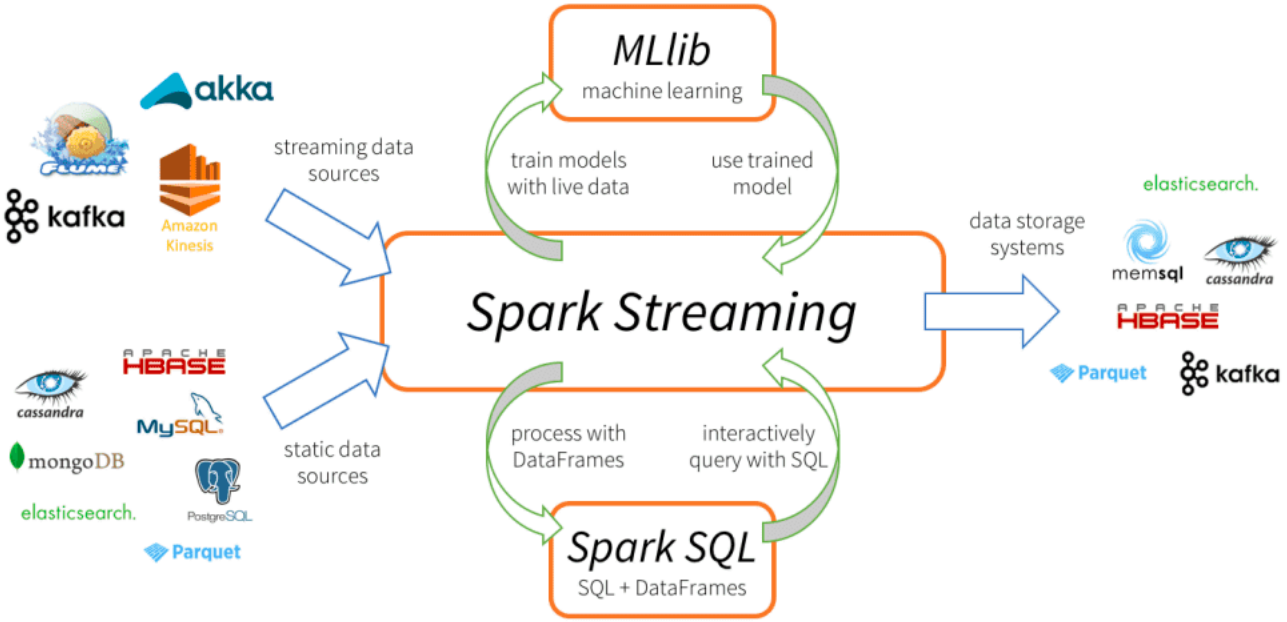
THE MODERN DATA PIPELINE

How does a typical data pipeline function?



THE MODERN DATA PIPELINE

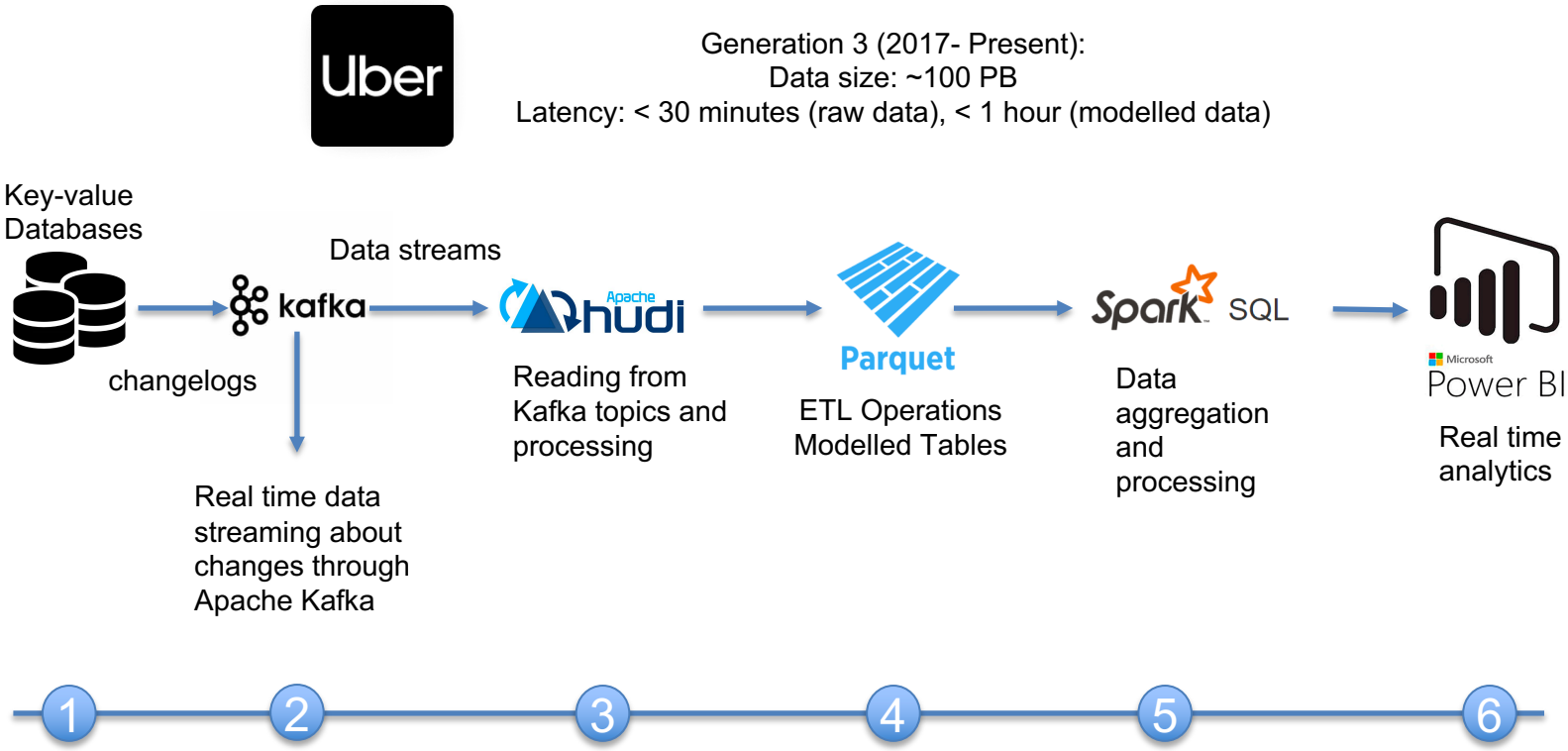
How does a typical data pipeline function?



Apache Spark solution offerings for a modern, near real time data processing pipeline

THE MODERN DATA PIPELINE

Example: Online Analytical Processing Case Study



BTS | Barcelona Technology School

Welcome to Spark SQL finally !!

INTRODUCING THE REVOLUTIONARY SPARK SQL

Few words to begin with..

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.1.1
      /_/

Using Scala version 2.12.10, OpenJDK 64-Bit Server VM, 14
Branch HEAD
Compiled by user ubuntu on 2021-02-22T01:33:19Z
Revision 1d550c4e90275ab418b9161925049239227f3dc9
Url https://github.com/apache/spark
Type --help for more information.
```

- SQL is a Spark library for structured data. It provides more information about the **structure of data** and **computation**
- PySpark DataFrame is an immutable **distributed collection of data** with named columns
- Designed for processing both structured (e.g relational database) and semi-structured data (e.g JSON)
- DataFrames in PySpark **support both SQL queries** (SELECT * from table) **or expression methods** (df.select())

INTRODUCING THE REVOLUTIONARY SPARK SQL

Few words to begin with..

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.1.1
      /_/

Using Scala version 2.12.10, OpenJDK 64-Bit Server VM, 14
Branch HEAD
Compiled by user ubuntu on 2021-02-22T01:33:19Z
Revision 1d550c4e90275ab418b9161925049239227f3dc9
Url https://github.com/apache/spark
Type --help for more information.
```

- SQL is a Spark library for structured data. It provides more information about the **structure of data** and **computation**
- PySpark DataFrame is an immutable **distributed collection of data** with named columns
- Designed for processing both structured (e.g relational database) and semi-structured data (e.g JSON)
- DataFrames in PySpark **support both SQL queries** (SELECT * from table) **or expression methods** (df.select())

SparkContext vs **SparkSession**:

Basically, SparkSession is the Entry point for DataFrame API

- SparkContext is the main entry point for creating RDDs
- SparkSession provides a single point of entry to interact with Spark DataFrames
- SparkSession is used to create DataFrame, register DataFrames, execute SQL queries
- SparkSession is available in PySpark shell as spark

INTRODUCING THE REVOLUTIONARY SPARK SQL

Creating DataFrames

Two different methods of creating DataFrames in PySpark

- From existing RDDs using SparkSession's createDataFrame() method: You may have already witnessed this in previous sessions.

- From various data sources (CSV, JSON, TXT) using SparkSession's read method (Schema controls the data and helps DataFrames to optimize queries Schema provides information about column name,type of data in the column, empty values etc.)

```
df= spark.read.option("multiline","true").json("./data/train_schedules.json")
df.show(4)
```

```
+-------+---+---------+------+------------+-------------------+--------------------+------------+
|arrival|day|departure|    id|station_code|       station_name|          train_name|train_number|
+-------+---+---------+------+------------+-------------------+--------------------+------------+
|   None|  1| 07:55:00|302214|          FM|KACHEGUDA FALAKNUMA|Falaknuma Lingamp...|       47154|
|   None|  1| 18:55:00|281458|         TCR|            THRISUR|Thrissur Guruvayu...|       56044|
|   None|  1| 15:05:00|309335|         PBR|          PORBANDAR|Porbandar Muzaffa...|       19269|
|   None|  1| 13:30:00|283774|           R|          RAIPUR JN|  RAIPUR ITWARI PASS|       58205|
+-------+---+---------+------+------------+-------------------+--------------------+------------+
only showing top 4 rows
```

INTRODUCING THE REVOLUTIONARY SPARK SQL

Playing with DataFrames: Actions

DataFrame actions are of the following types-

- Exploring the DATA SCHEMA
- Examining the FIRST FEW ROWS, ROWS COUNT, COLUMN NAMES etc.
- Performing SUMMARY STATISTICS

Some methods that helps us in achieving the aforesaid objectives are-
printSchema, head, show, count, columns and describe

```
df.printSchema()
```

```
root
 |-- arrival: string (nullable = true)
 |-- day: long (nullable = true)
 |-- departure: string (nullable = true)
 |-- id: long (nullable = true)
 |-- station_code: string (nullable = true)
 |-- station_name: string (nullable = true)
 |-- train_name: string (nullable = true)
 |-- train_number: string (nullable = true)
```

```
df.columns
```

```
['arrival',
 'day',
 'departure',
 'id',
 'station_code',
 'station_name',
 'train_name',
 'train_number']
```

INTRODUCING THE REVOLUTIONARY SPARK SQL

Playing with DataFrames: Transformations

DataFrame transformations are of the following types-

- Selecting a subset of attributes using SELECT etc.
- Aggregations using GROUP BY etc.
- Restructuring using DROP_DUPLICATE, RENAMING etc.

Some of the methods to achieve these tasks are select, Iter, groupby, orderby, dropDuplicates and withColumnRenamed

```
df.groupBy("station_name").count().show(5)
```

```
+--------------------+-----+
|        station_name|count|
+--------------------+-----+
|      AMBALA CANT JN|  201|
|  YAMUNA BRIDGE AGRA|   64|
|        CHIK BANAVAR|   56|
|Saharsa Kutchery ...|   12|
|        CHENNAI PARK|   84|
+--------------------+-----+
only showing top 5 rows
```

```
df.groupBy("station_name").count().orderBy("station_name").show(6)
```

```
+------------------+-----+
|      station_name|count|
+------------------+-----+
|                  |    2|
|A-CABIN BONDAMUNDA|   48|
|             ABADA|  182|
|          ABHAIPUR|   56|
|  ABHAYAPURI ASSAM|   32|
|          ABJUGANJ|   14|
+------------------+-----+
only showing top 6 rows
```

INTRODUCING THE REVOLUTIONARY SPARK SQL

Playing with DataFrames: Transformations

Dot-Notation and Structured-Query-Language (SQL)

SELECT columns: 3 approaches

1

```
df.select("train_number", "station_code", "departure",).show(4)
```

```
+------------+------------+---------+
|train_number|station_code|departure|
+------------+------------+---------+
|       47154|          FM| 07:55:00|
|       56044|         TCR| 18:55:00|
|       19269|         PBR| 15:05:00|
|       58205|           R| 13:30:00|
+------------+------------+---------+
only showing top 4 rows
```

INTRODUCING THE REVOLUTIONARY SPARK SQL

Playing with DataFrames: Transformations

Dot-Notation and Structured-Query-Language (SQL)

SELECT columns: 3 approaches

1
```
df.select("train_number", "station_code", "departure",).show(4)
```

```
+------------+------------+---------+
|train_number|station_code|departure|
+------------+------------+---------+
|       47154|          FM| 07:55:00|
|       56044|         TCR| 18:55:00|
|       19269|         PBR| 15:05:00|
|       58205|           R| 13:30:00|
+------------+------------+---------+
only showing top 4 rows
```

2
```
df.select(df.train_number, df.station_code, df.departure).show(4)
```

```
+------------+------------+---------+
|train_number|station_code|departure|
+------------+------------+---------+
|       47154|          FM| 07:55:00|
|       56044|         TCR| 18:55:00|
|       19269|         PBR| 15:05:00|
|       58205|           R| 13:30:00|
+------------+------------+---------+
only showing top 4 rows
```

INTRODUCING THE REVOLUTIONARY SPARK SQL

Playing with DataFrames: Transformations

Dot-Notation and Structured-Query-Language (SQL)

SELECT columns: 3 approaches

1
```
df.select("train_number", "station_code", "departure",).show(4)
```

```
+------------+------------+---------+
|train_number|station_code|departure|
+------------+------------+---------+
|       47154|          FM| 07:55:00|
|       56044|         TCR| 18:55:00|
|       19269|         PBR| 15:05:00|
|       58205|           R| 13:30:00|
+------------+------------+---------+
only showing top 4 rows
```

2
```
df.select(df.train_number, df.station_code, df.departure).show(4)
```

```
+------------+------------+---------+
|train_number|station_code|departure|
+------------+------------+---------+
|       47154|          FM| 07:55:00|
|       56044|         TCR| 18:55:00|
|       19269|         PBR| 15:05:00|
|       58205|           R| 13:30:00|
+------------+------------+---------+
only showing top 4 rows
```

3
```
df.select(col("train_number"), col("station_code"), col("departure")) \
.show(4)
```

```
+------------+------------+---------+
|train_number|station_code|departure|
+------------+------------+---------+
|       47154|          FM| 07:55:00|
|       56044|         TCR| 18:55:00|
|       19269|         PBR| 15:05:00|
|       58205|           R| 13:30:00|
+------------+------------+---------+
only showing top 4 rows
```

Note:

1. Always refrain to use these 3 different approaches in the same query

INTRODUCING THE REVOLUTIONARY SPARK SQL

Playing with DataFrames: The Structured-Query-Language (SQL) Approach

Aggregation is simpler with Structured-Query-Language (SQL) approach. How?

Consider the following two queries which give the same result as in the previous slide:

- Expression based query:

```
df.select("train_number", "station_code", "departure",).show(4)
```

- SQL based query:

```
df.createOrReplaceTempView("schedules")
query= """
SELECT train_number, station_code, station_code
FROM schedules
LIMIT 4
"""
spark.sql(query).show()
```

Which is more friendly to user in terms of understanding?

INTRODUCING THE REVOLUTIONARY SPARK SQL

Playing with DataFrames: The Structured-Query-Language (SQL) Approach

Working with **Window Function**:

A window function is like an aggregate function, except that it gives an output for every row in the dataset instead of a single row per group.

A running sum using a window function is simpler than what is required using joins. The query duration can also be much faster.

- Express operations more simply than dot notation or queries
- Each row uses the values of other rows to calculate its value

Whether to use dot notation or SQL is a personal preference. However, there are cases where SQL is simpler to perform operations. Also as demonstrated in the video lesson, there are also cases where the dot notation gives a counterintuitive result, such as when a second aggregation on a column clobbers a prior aggregation on that column. For example, the basic syntax of **agg** method in PySpark is only able to do a single aggregation on each column at a time.

INTRODUCING THE REVOLUTIONARY SPARK SQL

Playing with DataFrames: The Structured-Query-Language (SQL) Approach

Working with **Window Function**:

- **OVER** clause and **ORDER BY** Clause:
  OVER clause must contain an ORDER BY clause to determine how to sequence the rows

```
# OVER Clause: Adding row numbers
df.createOrReplaceTempView("schedules")
query= """
SELECT train_number, station_code , departure, ROW_NUMBER() OVER (ORDER BY train_number) AS row_number
       FROM schedules
       WHERE train_number= 12301

"""
spark.sql(query).show(5)
```

```
+------------+------------+---------+----------+
|train_number|station_code|departure|row_number|
+------------+------------+---------+----------+
|       12301|         HWH| 16:55:00|         1|
|       12301|         LLH| 16:58:00|         2|
|       12301|         BEQ| 17:00:00|         3|
|       12301|         BLY| 17:01:00|         4|
|       12301|         BZL| 17:03:00|         5|
+------------+------------+---------+----------+
only showing top 5 rows
```

INTRODUCING THE REVOLUTIONARY SPARK SQL

Playing with DataFrames: The Structured-Query-Language (SQL) Approach

Working with **Window Function**:

- **OVER** clause and **ORDER BY** Clause:
  OVER clause must contain an ORDER BY clause to determine how to sequence the rows

```
# OVER Clause: Adding row numbers
df.createOrReplaceTempView("schedules")
query= """
SELECT train_number, station_code , departure, ROW_NUMBER() OVER (ORDER BY train_number) AS row_number
       FROM schedules
       WHERE train_number= 12301

"""
spark.sql(query).show(5)
```

```
+------------+------------+---------+----------+
|train_number|station_code|departure|row_number|
+------------+------------+---------+----------+
|       12301|         HWH| 16:55:00|         1|
|       12301|         LLH| 16:58:00|         2|
|       12301|         BEQ| 17:00:00|         3|
|       12301|         BLY| 17:01:00|         4|
|       12301|         BZL| 17:03:00|         5|
+------------+------------+---------+----------+
only showing top 5 rows
```

- **LEAD** function allows us to query more than one row at a time without having the need to join the table to itself

```
# LEAD Clause: Upcoming arrival time
query= """
SELECT train_number, station_code , departure, ROW_NUMBER() OVER (ORDER BY train_number) AS row_number,
       LEAD(departure, 1) OVER (ORDER BY train_number) AS upcoming_arrival
       FROM schedules
       WHERE train_number= 12301

"""
spark.sql(query).show(5)
```

```
+------------+------------+---------+----------+----------------+
|train_number|station_code|departure|row_number|upcoming_arrival|
+------------+------------+---------+----------+----------------+
|       12301|         HWH| 16:55:00|         1|        16:58:00|
|       12301|         LLH| 16:58:00|         2|        17:00:00|
|       12301|         BEQ| 17:00:00|         3|        17:01:00|
|       12301|         BLY| 17:01:00|         4|        17:03:00|
|       12301|         BZL| 17:03:00|         5|        17:05:00|
+------------+------------+---------+----------+----------------+
only showing top 5 rows
```

BTS | Barcelona Technology School

Data Sources

USING DATA SOURCES

Data Sources? Why do we need Data Sources?

Broadly speaking,
"**Every Apache Spark application starts with loading the data and ends with saving the data**"

But, loading and saving the data is not easy. Why?

- **Data Formats**: Different enterprises prefer to store their data in different formats.
- **Data Parsing**: reading records in text formats and parsing it, deserializing data stored in binary formats etc.
- **Data Transformation**: converting, say your java objects to JSON records etc.

Such procedures severely affects the ability of an application to be flexible resulting into inflexible input/ output logic and a lot errors….

USING DATA SOURCES

Wait…Did somebody talk about Binary files?

Binary Files?

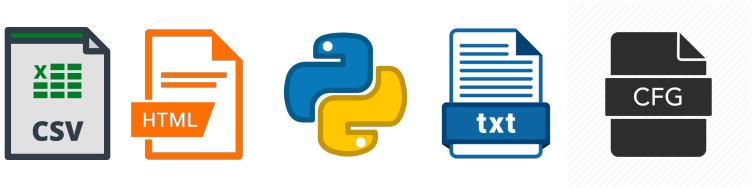
Interesting to hear that we are going to read binary files. But what are binary files and in fact, why do we need binary files. In order to understand these questions, first understand that files in a computer are broadly classified into-

- Text files: must represent reasonable text (i.e. must be a reasonable sequence of bytes), and can be edited in any text editor program.

- Binary files: have no inherent constraints (can be any sequence of bytes), and must be opened in an appropriate program that knows the specific file format (such as Media Player, Photoshop, Office and others).

Surprisingly, it is a fact that **every text file is in fact a binary file but, the reverse is not true**.

Since Spark 3.0, Spark supports binary file data source, which reads binary files and converts each file into a single record that contains the raw content and metadata of the file. It produces a DataFrame with the following columns and possibly partition columns:



Examples of text files

Examples of binary files

- `path`: StringType
- `modificationTime`: TimestampType
- `length`: LongType
- `content`: BinaryType

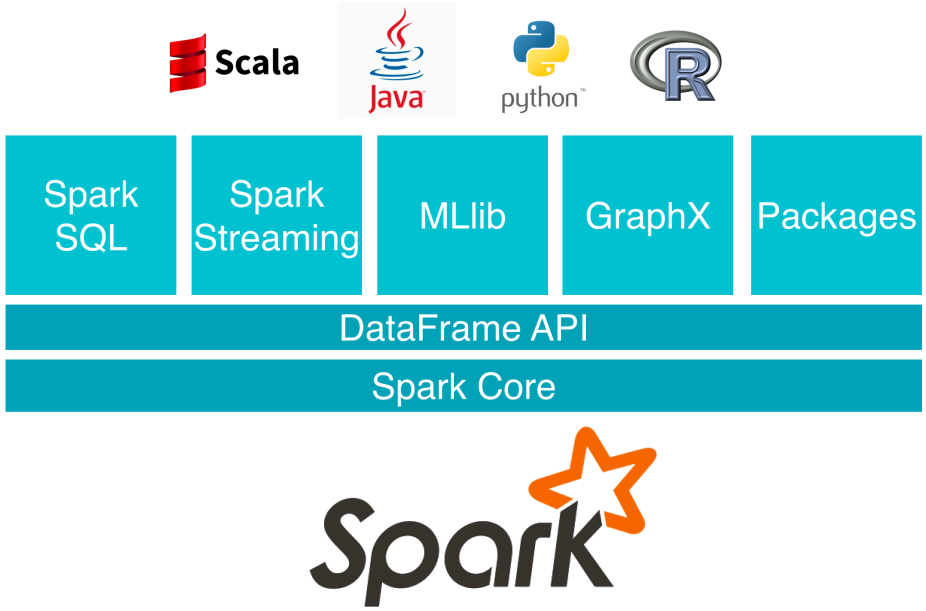
USING DATA SOURCES

Back to Data Sources. What is the solution?

Broadly speaking,
"**Every Apache Spark application starts with loading the data and ends with saving the data**"

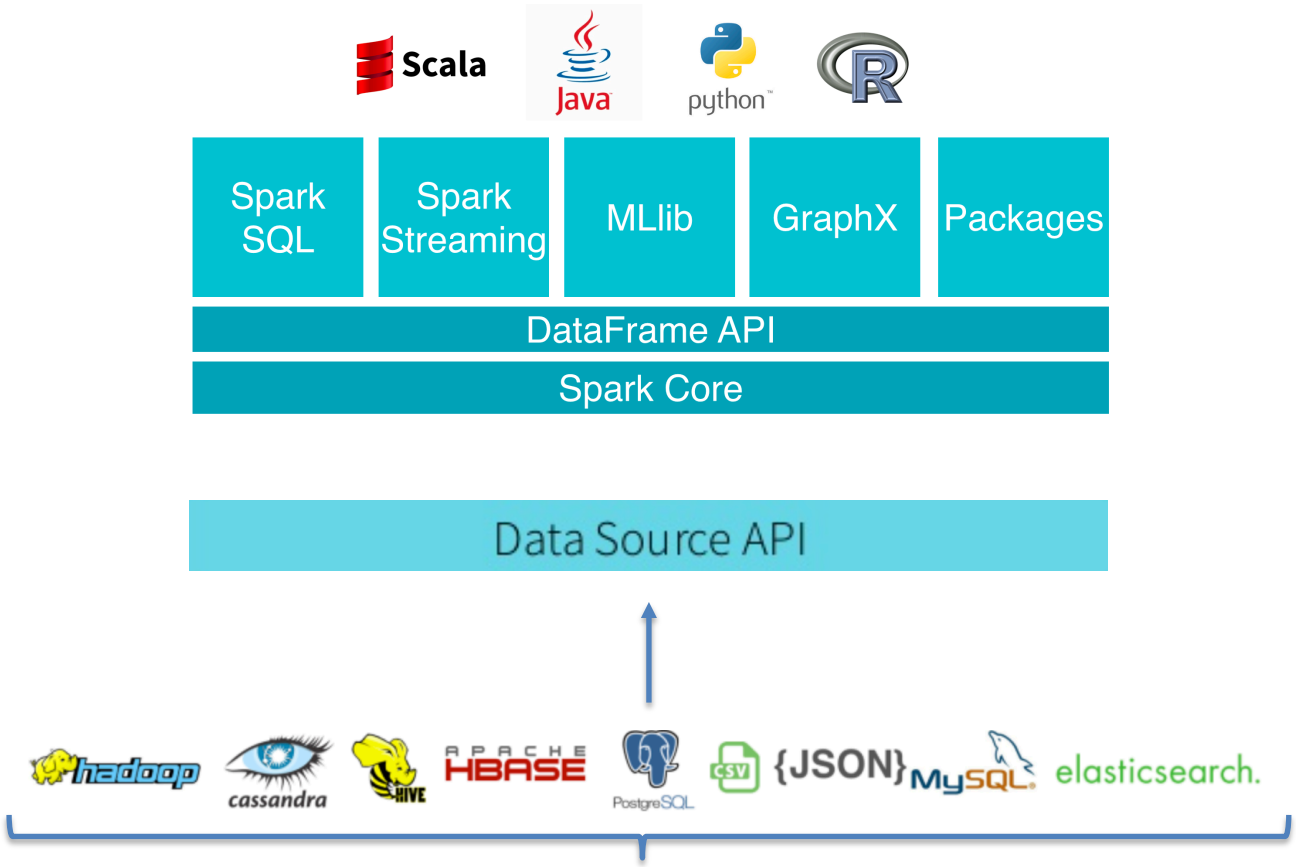
But, loading and saving the data is not easy.

So, what is the solution?
The solution is the majestic Apache Spark! Once again !!!



USING DATA SOURCES

Introducing the Spark's Data Sources- Application Programming Interface (API)



Multiple sources/ formats/ types of data, all enabled to be read using a simple API

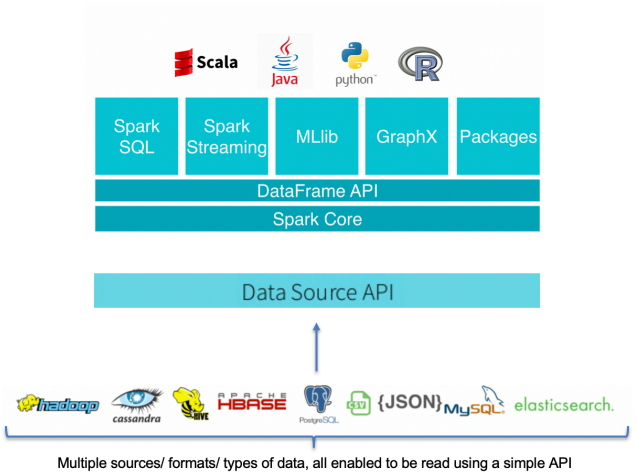
USING DATA SOURCES

Objectives of the Data Sources API

- The Data Source API in Spark is a convenient feature that enables developers to write libraries to connect to data stored in various sources with Spark. Equipped with the Data Source API, users can load/save data from/to different data formats and systems with minimal setup and configuration

- Spark SQL supports operating on a variety of data sources through the DataFrame interface. A DataFrame can be operated on using relational transformations and can also be used to create a temporary view.

- Registering a DataFrame as a temporary view allows you to run SQL queries over its data.



USING DATA SOURCES

Simplifying the loading/ saving operations

There are some generic load and save functions provided by the Data Sources API of Apache Spark. These are described along with their characteristics as follows-

1. **Load**
    1. Data source name: What source we are loading the data from
    2. Options: parameters for a specific data source, for e.g. data path
    3. Schema: If the data source accepts a user-specified schema

1. **Save**
    1. Data source name: What source we are going to save the data to
    2. Save mode: What should we do when data already exists
    3. Options: parameters for a specific data source, for e.g. data path

USING DATA SOURCES

Simplifying the loading/ saving operations

There are some generic load and save functions provided by the Data Sources API of Apache Spark. These are described along with their characteristics as follows-

1. **Load**
    1. Data source name: What source we are loading the data from
    2. Options: parameters for a specific data source, for e.g. data path
    3. Schema: If the data source accepts a user-specified schema

1. **Save**
    1. Data source name: What source we are going to save the data to
    2. Save mode: What should we do when data already exists
    3. Options: parameters for a specific data source, for e.g. data path

```
df = spark.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

USING DATA SOURCES

Simplifying the loading/ saving operations

There are some generic load and save functions provided by the Data Sources API of Apache Spark. These are described along with their characteristics as follows-

1. **Load**
    1. Data source name: What source we are loading the data from
    2. Options: parameters for a specific data source, for e.g. data path
    3. Schema: If the data source accepts a user-specified schema

1. **Save**
    1. Data source name: What source we are going to save the data to
    2. Save mode: What should we do when data already exists
    3. Options: parameters for a specific data source, for e.g. data path

```
df = spark.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

```
df = spark.read.load("examples/src/main/resources/people.json", format="json")
df.select("name", "age").write.save("namesAndAges.parquet", format="parquet")
```

USING DATA SOURCES

Simplifying the loading/ saving operations

There are some generic load and save functions provided by the Data Sources API of Apache Spark. These are described along with their characteristics as follows-

1. **Load**
    1. Data source name: What source we are loading the data from
    2. Options: parameters for a specific data source, for e.g. data path
    3. Schema: If the data source accepts a user-specified schema

1. **Save**
    1. Data source name: What source we are going to save the data to
    2. Save mode: What should we do when data already exists
    3. Options: parameters for a specific data source, for e.g. data path

```
df = spark.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

```
df = spark.read.load("examples/src/main/resources/people.json", format="json")
df.select("name", "age").write.save("namesAndAges.parquet", format="parquet")
```

```
df = spark.read.load("examples/src/main/resources/people.csv",
                     format="csv", sep=";", inferSchema="true", header="true")
```

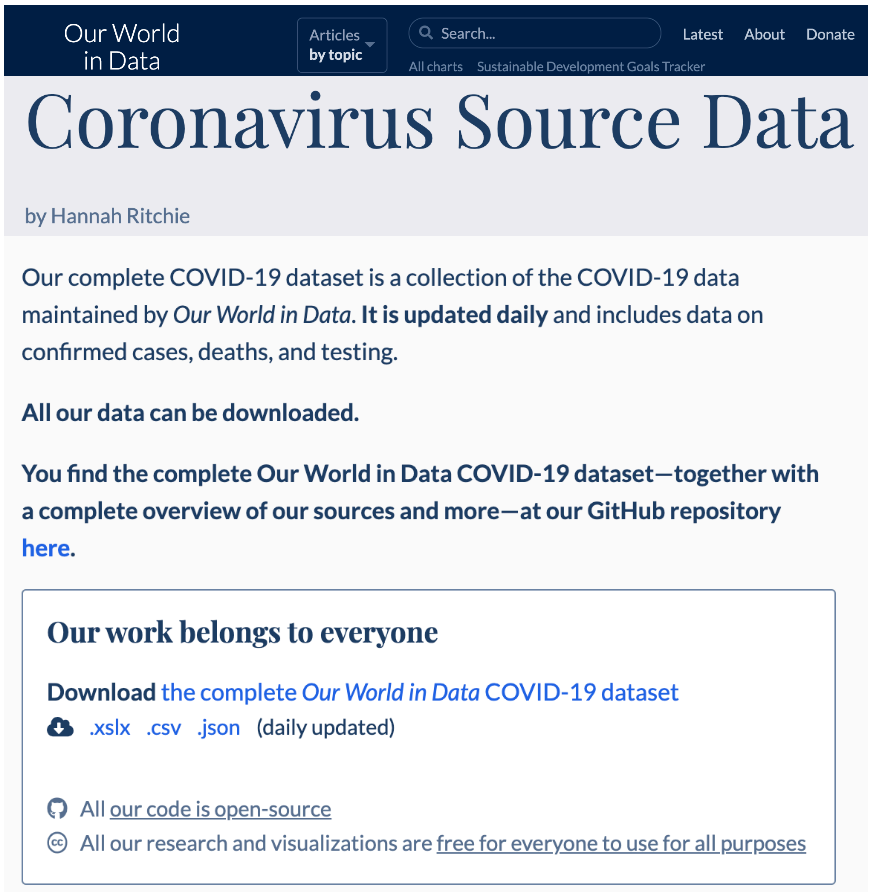
Note:

In the simplest form, the default data source (`parquet` unless otherwise configured by `spark.sql.sources.default`) will be used for all operations.

[1] Read more about the Generic File Source Options here-
https://spark.apache.org/docs/latest/sql-data-sources-generic-options.html

USING DATA SOURCES

Reading Comma Separated Values (CSV)



Let us first examine how does a comma separated values (CSV) file looks like. We can observe below that the first highlighted chunk is the header with values separated by a comma.

```
iso_code,continent,location,date,total_cases,new_cases,new_cases_smoothed,total_deaths,new_deaths,new_deaths_smoothed,total_cas
es_per_million,new_cases_per_million,new_cases_smoothed_per_million,total_deaths_per_million,new_deaths_per_million,new_deaths_
smoothed_per_million,reproduction_rate,icu_patients,icu_patients_per_million,hosp_patients,hosp_patients_per_million,weekly_icu
_admissions,weekly_icu_admissions_per_million,weekly_hosp_admissions,weekly_hosp_admissions_per_million,new_tests,total_tests,t
otal_tests_per_thousand,new_tests_per_thousand,new_tests_smoothed,new_tests_smoothed_per_thousand,positive_rate,tests_per_case,
tests_units,total_vaccinations,people_vaccinated,people_fully_vaccinated,new_vaccinations,new_vaccinations_smoothed,total_vacci
nations_per_hundred,people_vaccinated_per_hundred,people_fully_vaccinated_per_hundred,new_vaccinations_smoothed_per_million,str
ingency_index,population,population_density,median_age,aged_65_older,aged_70_older,gdp_per_capita,extreme_poverty,cardiovasc_de
ath_rate,diabetes_prevalence,female_smokers,male_smokers,handwashing_facilities,hospital_beds_per_thousand,life_expectancy,huma
n_development_index
AFG,Asia,Afghanistan,2020-02-
24,1.0,1.0,,,,,0.026,0.026,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,8.33,38928341.0,54.422,18.6,2.581,1.337,1803.987,,597.029,9.59,,,37.7
46,0.5,64.83,0.511
AFG,Asia,Afghanistan,2020-02-
25,1.0,0.0,,,,,0.026,0.0,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,8.33,38928341.0,54.422,18.6,2.581,1.337,1803.987,,597.029,9.59,,,37.746
,0.5,64.83,0.511
```

In order to start making any sense out of the data, we need to download it. The small code chunk downloads it. (Please note that after this operation was done, we moved the file to the data folder where every data related to our course resides)

```
! wget 'https://covid.ourworldindata.org/data/owid-covid-data.csv'
```

```
--2021-04-21 00:57:03--  https://covid.ourworldindata.org/data/owid-covid-data.csv
Resolving covid.ourworldindata.org (covid.ourworldindata.org)... 104.21.233.138, 104.21.233.137
Connecting to covid.ourworldindata.org (covid.ourworldindata.org)|104.21.233.138|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/csv]
Saving to: ‘owid-covid-data.csv’

owid-covid-data.csv     [                 <=>]  20,43M  6,32MB/s    in 3,4s

2021-04-21 00:57:07 (5,95 MB/s) - ‘owid-covid-data.csv’ saved [21419439]
```

USING DATA SOURCES

Reading Comma Separated Values (CSV)

```
df = spark.read.csv('data/owid-covid-data.csv', sep= ',', header= True)
df.count()
```

```
82837
```

```
df.select("continent", "location", "date",
          "total_cases", "total_deaths",
          "total_cases_per_million",
          "population"
         ).show(5)
```

```
+---------+-----------+----------+-----------+------------+----------------------+----------+
|continent|   location|      date|total_cases|total_deaths|total_cases_per_million|population|
+---------+-----------+----------+-----------+------------+----------------------+----------+
|     Asia|Afghanistan|2020-02-24|        1.0|        null|                 0.026|38928341.0|
|     Asia|Afghanistan|2020-02-25|        1.0|        null|                 0.026|38928341.0|
|     Asia|Afghanistan|2020-02-26|        1.0|        null|                 0.026|38928341.0|
|     Asia|Afghanistan|2020-02-27|        1.0|        null|                 0.026|38928341.0|
|     Asia|Afghanistan|2020-02-28|        1.0|        null|                 0.026|38928341.0|
+---------+-----------+----------+-----------+------------+----------------------+----------+
only showing top 5 rows
```

USING DATA SOURCES

Exploring Comma Separated Values (CSV)

```
df = spark.read.csv('data/owid-covid-data.csv', sep= ',', header= True)
df.count()
```

```
82837
```

```
df.select("continent", "location", "date",
          "total_cases", "total_deaths",
          "total_cases_per_million",
          "population"
         ).show(5)
```

```
+---------+-----------+----------+-----------+------------+---------------------+----------+
|continent|   location|      date|total_cases|total_deaths|total_cases_per_million|population|
+---------+-----------+----------+-----------+------------+---------------------+----------+
|     Asia|Afghanistan|2020-02-24|        1.0|        null|                0.026|38928341.0|
|     Asia|Afghanistan|2020-02-25|        1.0|        null|                0.026|38928341.0|
|     Asia|Afghanistan|2020-02-26|        1.0|        null|                0.026|38928341.0|
|     Asia|Afghanistan|2020-02-27|        1.0|        null|                0.026|38928341.0|
|     Asia|Afghanistan|2020-02-28|        1.0|        null|                0.026|38928341.0|
+---------+-----------+----------+-----------+------------+---------------------+----------+
only showing top 5 rows
```

```
df.createOrReplaceTempView("COVID")
covid_data= spark.sql("SELECT location AS country, date, total_cases,\
            total_cases_per_million, population \
            FROM COVID WHERE location ='India'")
covid_data.show(5)
```

```
+-------+----------+-----------+---------------------+------------+
|country|      date|total_cases|total_cases_per_million|  population|
+-------+----------+-----------+---------------------+------------+
|  India|2020-01-30|        1.0|                0.001|1380004385.0|
|  India|2020-01-31|        1.0|                0.001|1380004385.0|
|  India|2020-02-01|        1.0|                0.001|1380004385.0|
|  India|2020-02-02|        2.0|                0.001|1380004385.0|
|  India|2020-02-03|        3.0|                0.002|1380004385.0|
+-------+----------+-----------+---------------------+------------+
only showing top 5 rows
```

USING DATA SOURCES

Reading Tab Separated Values (TSV)


```
df = spark.read.csv("data/2021-04-19-dataset.tsv", header= True, sep= '\t')
df.count()
```

```
Out[62]: 1645911
```

```
df.printSchema()
```

```
root
 |-- tweet_id: string (nullable = true)
 |-- date: string (nullable = true)
 |-- time: string (nullable = true)
 |-- lang: string (nullable = true)
 |-- country_code: string (nullable = true)
```

```
df.show()
```

```
+------------------+----------+--------+----+------------+
|          tweet_id|      date|    time|lang|country_code|
+------------------+----------+--------+----+------------+
|1383994244709314567|2021-04-19|04:01:55|  en|        NULL|
|1383994244763844612|2021-04-19|04:01:55|  en|        NULL|
|1383994244986130433|2021-04-19|04:01:56|  es|        NULL|
|1383994245065756673|2021-04-19|04:01:56|  in|        NULL|
|1383994245200056321|2021-04-19|04:01:56|  en|        NULL|
```

Next is the dataset of tweets acquired from the twitter related to COVID-19 chatter. Now, for the purposes of our analysis, we have downloaded the raw data file from the GitHub repository of the lab in the form of .tsv file extension.

So, interestingly, we have a lot of tweets gathered for our analysis but the most important question is what do such tweets contain? Surprisingly, we can see the content of such tweets merely by using their tweet_id. For example, to see the content so tweet_id 1383994244709314567,
all we need to do is visit
https://twitter.com/anyuser/status/1383994244709314567
and we would be redirected to the actual tweet

USING DATA SOURCES

Reading Other file formats

```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

# DataFrames can be saved as Parquet files, maintaining the schema information.
peopleDF.write.parquet("people.parquet")

# Read in the Parquet file created above.
# Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a DataFrame.
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL statements.
parquetFile.createOrReplaceTempView("parquetFile")
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

Parquet files

[1] Hive tables as Data sources
https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html

USING DATA SOURCES

Reading Other file formats

```
from os.path import abspath

from pyspark.sql import SparkSession
from pyspark.sql import Row

# warehouse_location points to the default location for managed databases and tables
warehouse_location = abspath('spark-warehouse')

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL Hive integration example") \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()

# spark is an existing SparkSession
spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

# Queries are expressed in HiveQL
spark.sql("SELECT * FROM src").show()
# +---+-------+
# |key|  value|
# +---+-------+
# |238|val_238|
# | 86| val_86|
# |311|val_311|
```

Hive Tables

[1] Hive tables as Data sources
https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html

USING DATA SOURCES

Reading Other file formats

Since Spark 3.0, Spark supports binary file data source, which reads binary files and converts each file into a single record that contains the raw content and metadata of the file. It produces a DataFrame with the following columns and possibly partition columns:

path: StringType
modificationTime: TimestampType
length: LongType
content: BinaryType

To read whole binary files, you need to specify the data source format as binaryFile. To load files with paths matching a given glob pattern while keeping the behavior of partition discovery, you can use the general data source option pathGlobFilter. For example, the following code reads all PNG files from the input directory:

```
spark.read.format("binaryFile").option("pathGlobFilter", "*.png").load("/path/to/data")
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

GET https://think.cs.vt.edu/corgis/datasets/json/airlines/airlines.json Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings


```
{
    "Airport": {
        "Code": "ATL",
        "Name": "Atlanta, GA: Hartsfield-Jackson Atlanta International"
    },
    "Time": {
        "Label": "2003/06",
        "Month": 6,
        "Month Name": "June",
        "Year": 2003
    },
    "Statistics": {
        "# of Delays": {
            "Carrier": 1009,
            "Late Aircraft": 1275,
            "National Aviation System": 3217,
            "Security": 17,
            "Weather": 328
        },
        "Carriers": {
            "Names": "American Airlines Inc.,JetBlue Airways,Continental Air Lines Inc.,Delta Air Lines Inc.,Atlantic Southeast
                Airlines,AirTran Airways Corporation,America West Airlines Inc.,Northwest Airlines Inc.,ExpressJet Airlines Inc.,
                United Air Lines Inc.,US Airways Inc.",
            "Total": 11
        },
        "Flights": {
            "Cancelled": 216,
            "Delayed": 5843,
            "Diverted": 27,
            "On Time": 23974,
            "Total": 30060
        },
        "Minutes Delayed": {
            "Carrier": 61606,
            "Late Aircraft": 68335,
            "National Aviation System": 118831,
            "Security": 518,
            "Total": 268764,
            "Weather": 19474
        }
    }
},
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Semi_Sructured_Data_Analysis") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Semi_Sructured_Data_Analysis") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

```
import requests, json

url = 'https://think.cs.vt.edu/corgis/datasets/json/airlines/airlines.json'
resp = requests.get(url=url)
json_string = json.dumps(resp.json())
json_data= json.loads(json_string)

with open('airlines.json', 'w') as json_file:
    json.dump(json_data, json_file)
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Semi_Sructured_Data_Analysis") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

```
import requests, json

url = 'https://think.cs.vt.edu/corgis/datasets/json/airlines/airlines.json'
resp = requests.get(url=url)
json_string = json.dumps(resp.json())
json_data= json.loads(json_string)

with open('airlines.json', 'w') as json_file:
    json.dump(json_data, json_file)
```

```
df = spark.read.json("airlines.json")
df.show(6)
```

```
+--------------------+--------------------+--------------------+
|             Airport|          Statistics|                Time|
+--------------------+--------------------+--------------------+
|{ATL, Atlanta, GA...|{{1009, 1275, 321...|{2003/06, 6, June...|
|{BOS, Boston, MA:...|{{374, 495, 685, ...|{2003/06, 6, June...|
|{BWI, Baltimore, ...|{{296, 477, 389, ...|{2003/06, 6, June...|
|{CLT, Charlotte, ...|{{300, 472, 735, ...|{2003/06, 6, June...|
|{DCA, Washington,...|{{283, 268, 487, ...|{2003/06, 6, June...|
|{DEN, Denver, CO:...|{{516, 323, 664, ...|{2003/06, 6, June...|
+--------------------+--------------------+--------------------+
only showing top 6 rows
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Understanding the schema of the JSON through the dataframe
df.printSchema()
```

```
root
 |-- Airport: struct (nullable = true)
 |    |-- Code: string (nullable = true)
 |    |-- Name: string (nullable = true)
 |-- Statistics: struct (nullable = true)
 |    |-- # of Delays: struct (nullable = true)
 |    |    |-- Carrier: long (nullable = true)
 |    |    |-- Late Aircraft: long (nullable = true)
 |    |    |-- National Aviation System: long (nullable = true)
 |    |    |-- Security: long (nullable = true)
 |    |    |-- Weather: long (nullable = true)
 |    |-- Carriers: struct (nullable = true)
 |    |    |-- Names: string (nullable = true)
 |    |    |-- Total: long (nullable = true)
 |    |-- Flights: struct (nullable = true)
 |    |    |-- Cancelled: long (nullable = true)
 |    |    |-- Delayed: long (nullable = true)
 |    |    |-- Diverted: long (nullable = true)
 |    |    |-- On Time: long (nullable = true)
 |    |    |-- Total: long (nullable = true)
 |    |-- Minutes Delayed: struct (nullable = true)
 |    |    |-- Carrier: long (nullable = true)
 |    |    |-- Late Aircraft: long (nullable = true)
 |    |    |-- National Aviation System: long (nullable = true)
 |    |    |-- Security: long (nullable = true)
 |    |    |-- Total: long (nullable = true)
 |    |    |-- Weather: long (nullable = true)
 |-- Time: struct (nullable = true)
 |    |-- Label: string (nullable = true)
 |    |-- Month: long (nullable = true)
 |    |-- Month Name: string (nullable = true)
 |    |-- Year: long (nullable = true)
```


```
{
    "Airport": {
        "Code": "ATL",
        "Name": "Atlanta, GA: Hartsfield-Jackson Atlanta International"
    },
    "Time": {
        "Label": "2003/06",
        "Month": 6,
        "Month Name": "June",
        "Year": 2003
    },
    "Statistics": {
        "# of Delays": {
            "Carrier": 1009,
            "Late Aircraft": 1275,
            "National Aviation System": 3217,
            "Security": 17,
            "Weather": 328
        },
        "Carriers": {
            "Names": "American Airlines Inc.,JetBlue Airways,Continental Air Lines Inc.,Delta Air Lines Inc.,Atlantic Southeast
                Airlines,AirTran Airways Corporation,America West Airlines Inc.,Northwest Airlines Inc.,ExpressJet Airlines Inc.,
                United Air Lines Inc.,US Airways Inc.",
            "Total": 11
        },
        "Flights": {
            "Cancelled": 216,
            "Delayed": 5843,
            "Diverted": 27,
            "On Time": 23974,
            "Total": 30060
        },
        "Minutes Delayed": {
            "Carrier": 61606,
            "Late Aircraft": 68335,
            "National Aviation System": 118831,
            "Security": 518,
            "Total": 268764,
            "Weather": 19474
        }
    }
},

.....
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Subsetting the dataframe to analyse the statistics of # (number) of delays
airport_delay_statistics = df.select("Airport.Code", "Airport.Name",
                                     "Statistics.# of Delays.Carrier",
                                     "Statistics.# of Delays.Late Aircraft",
                                     "Statistics.# of Delays.Security",
                                     "Statistics.# of Delays.Weather")
airport_delay_statistics.printSchema()
```

```
root
 |-- Code: string (nullable = true)
 |-- Name: string (nullable = true)
 |-- Carrier: long (nullable = true)
 |-- Late Aircraft: long (nullable = true)
 |-- Security: long (nullable = true)
 |-- Weather: long (nullable = true)
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Subsetting the dataframe to analyse the statistics of # (number) of delays
airport_delay_statistics = df.select("Airport.Code", "Airport.Name",
                                     "Statistics.# of Delays.Carrier",
                                     "Statistics.# of Delays.Late Aircraft",
                                     "Statistics.# of Delays.Security",
                                     "Statistics.# of Delays.Weather")
airport_delay_statistics.printSchema()
```

```
root
 |-- Code: string (nullable = true)
 |-- Name: string (nullable = true)
 |-- Carrier: long (nullable = true)
 |-- Late Aircraft: long (nullable = true)
 |-- Security: long (nullable = true)
 |-- Weather: long (nullable = true)
```

Number of Delays:

1. **Carrier**: The number of delays and cancellations due to circumstances within the airline's control (e.g. maintenance or crew problems, aircraft cleaning, baggage loading, fueling, etc.) in this month.
2. **Late Aircraft**: The number of delays and cancellations caused by a previous flight with the same aircraft arriving late, causing the present flight to depart late in this month.
3. **National Aviation System**: The number of delays and cancellations attributable to the national aviation system that refer to a broad set of conditions, such as non-extreme weather conditions, airport operations, heavy traffic volume, and air traffic control in this month.
4. **Security**: Number of delays or cancellations caused by evacuation of a terminal or concourse, re-boarding of aircraft because of security breach, inoperative screening equipment and/or long lines in excess of 29 minutes at screening areas in this month.
5. **Weather**: Number of delays or cancellations caused by significant meteorological conditions (actual or forecasted) that, in the judgment of the carrier, delays or prevents the operation of a flight such as tornado, blizzard or hurricane in this month.

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Subsetting the dataframe to analyse the statistics of # (number) of delays
airport_delay_statistics = df.select("Airport.Code", "Airport.Name",
                                     "Statistics.# of Delays.Carrier",
                                     "Statistics.# of Delays.Late Aircraft",
                                     "Statistics.# of Delays.Security",
                                     "Statistics.# of Delays.Weather")
airport_delay_statistics.printSchema()
```

```
root
 |-- Code: string (nullable = true)
 |-- Name: string (nullable = true)
 |-- Carrier: long (nullable = true)
 |-- Late Aircraft: long (nullable = true)
 |-- Security: long (nullable = true)
 |-- Weather: long (nullable = true)
```

```
# Creating a summary statistics of the dataframe
airport_delay_statistics.describe().show()
```

```
+-------+----+--------------------+-----------------+-----------------+-----------------+-----------------+
|summary|Code|                Name|          Carrier|    Late Aircraft|         Security|          Weather|
+-------+----+--------------------+-----------------+-----------------+-----------------+-----------------+
|  count|4408|                4408|             4408|             4408|             4408|             4408|
|   mean|null|                null| 574.6324863883848|789.078947368421|  5.5755444646098|78.21687840290382|
| stddev|null|                null|329.61647461501815| 561.79842030889|6.007046080059749|75.18172623192343|
|    min| ATL|Atlanta, GA: Hart...|              112|               86|               -1|                1|
|    max| TPA|Washington, DC: W...|             3087|             4483|               94|              812|
+-------+----+--------------------+-----------------+-----------------+-----------------+-----------------+
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Creating a summary statistics of the dataframe
airport_delay_statistics.describe().show()
```

```
+-------+----+--------------------+-----------------+-----------------+-----------------+-----------------+
|summary|Code|                Name|          Carrier|   Late Aircraft|         Security|          Weather|
+-------+----+--------------------+-----------------+-----------------+-----------------+-----------------+
|  count|4408|                4408|             4408|             4408|             4408|             4408|
|   mean|null|                null| 574.6324863883848|789.078947368421|  5.5755444646098|78.21687840290382|
| stddev|null|                null|329.61647461501815| 561.79842030889|6.007046080059749|75.18172623192343|
|    min| ATL|Atlanta, GA: Hart...|              112|               86|               -1|                1|
|    max| TPA|Washington, DC: W...|             3087|             4483|               94|              812|
+-------+----+--------------------+-----------------+-----------------+-----------------+-----------------+
```

```
# Creating a summary statistics by without columns having NAs
airport_delay_statistics.describe().dropna().show()
```

```
+-------+----+--------------------+-------+-------------+--------+-------+
|summary|Code|                Name|Carrier|Late Aircraft|Security|Weather|
+-------+----+--------------------+-------+-------------+--------+-------+
|  count|4408|                4408|   4408|         4408|    4408|   4408|
|    min| ATL|Atlanta, GA: Hart...|    112|           86|      -1|      1|
|    max| TPA|Washington, DC: W...|   3087|         4483|      94|    812|
+-------+----+--------------------+-------+-------------+--------+-------+
```

```
# Creating a summary statistics by without columns having NAs
airport_delay_statistics.describe().drop("Code", "Name").show()
```

```
+-------+-----------------+----------------+-----------------+-----------------+
|summary|          Carrier|   Late Aircraft|         Security|          Weather|
+-------+-----------------+----------------+-----------------+-----------------+
|  count|             4408|            4408|             4408|             4408|
|   mean| 574.6324863883848|789.078947368421|  5.5755444646098|78.21687840290382|
| stddev|329.61647461501815| 561.79842030889|6.007046080059749|75.18172623192343|
|    min|              112|              86|               -1|                1|
|    max|             3087|            4483|               94|              812|
+-------+-----------------+----------------+-----------------+-----------------+
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Creating a summary statistics of the dataframe
airport_delay_statistics.describe().show()
```

```
+-------+----+--------------------+-----------------+-----------------+-----------------+-----------------+
|summary|Code|                Name|          Carrier|   Late Aircraft|         Security|          Weather|
+-------+----+--------------------+-----------------+-----------------+-----------------+-----------------+
|  count|4408|                4408|             4408|             4408|             4408|             4408|
|   mean|null|                null| 574.6324863883848|789.078947368421|  5.5755444646098|78.21687840290382|
| stddev|null|                null|329.61647461501815| 561.79842030889|6.007046080059749|75.18172623192343|
|    min| ATL|Atlanta, GA: Hart...|              112|               86|               -1|                1|
|    max| TPA|Washington, DC: W...|             3087|             4483|               94|              812|
+-------+----+--------------------+-----------------+-----------------+-----------------+-----------------+
```

```
# Creating summary statistics of the dataframe attributes
airport_delay_statistics.select("Carrier", "Late Aircraft",
                                "Security","Weather").describe().show(4)
```

```
+-------+-----------------+-----------------+-----------------+-----------------+
|summary|          Carrier|   Late Aircraft|         Security|          Weather|
+-------+-----------------+-----------------+-----------------+-----------------+
|  count|             4408|             4408|             4408|             4408|
|   mean| 574.6324863883848|789.078947368421|  5.5755444646098|78.21687840290382|
| stddev|329.61647461501815| 561.79842030889|6.007046080059749|75.18172623192343|
|    min|              112|               86|               -1|                1|
|    max|             3087|             4483|               94|              812|
+-------+-----------------+-----------------+-----------------+-----------------+
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Sorting the airport delay statistics based on the Weather
airport_statistics.select("Code", "Name", "Weather").sort("Weather").show(10)
```

```
+----+--------------------+-------+
|Code|                Name|Weather|
+----+--------------------+-------+
| MDW|Chicago, IL: Chic...|      1|
| PDX|Portland, OR: Por...|      2|
| BWI|Baltimore, MD: Ba...|      2|
| FLL|Fort Lauderdale, ...|      2|
| PDX|Portland, OR: Por...|      3|
| IAD|Washington, DC: W...|      3|
| MCO|Orlando, FL: Orla...|      4|
| PDX|Portland, OR: Por...|      4|
| FLL|Fort Lauderdale, ...|      4|
| MDW|Chicago, IL: Chic...|      4|
+----+--------------------+-------+
only showing top 10 rows
```

Now, as we can see that the table provides the list of airports having the least number of delays due to adverse weather conditions.

Now, as we can see that the table provides the list of airports having the least number of delays due to adverse weather conditions.

However, it is interesting to note that we do not have complete names of the airports and rather we have to compromise by merely using the names of the location where such airports are located. How should we resolve this?

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Sorting the airport delay statistics based on the Weather
airport_statistics.select("Code", "Name", "Weather").sort("Weather").show(10)
```

```
+----+--------------------+-------+
|Code|                Name|Weather|
+----+--------------------+-------+
| MDW|Chicago, IL: Chic...|      1|
| PDX|Portland, OR: Por...|      2|
| BWI|Baltimore, MD: Ba...|      2|
| FLL|Fort Lauderdale, ...|      2|
| PDX|Portland, OR: Por...|      3|
| IAD|Washington, DC: W...|      3|
| MCO|Orlando, FL: Orla...|      4|
| PDX|Portland, OR: Por...|      4|
| FLL|Fort Lauderdale, ...|      4|
| MDW|Chicago, IL: Chic...|      4|
+----+--------------------+-------+
only showing top 10 rows
```

Now, as we can see that the table provides the list of airports having the least number of delays due to adverse weather conditions.

Now, as we can see that the table provides the list of airports having the least number of delays due to adverse weather conditions.

However, it is interesting to note that we do not have complete names of the airports and rather we have to compromise by merely using the names of the location where such airports are located. How should we resolve this?

```
# Sorting the airport delay statistics based on the Weather
airport_statistics.select("Code", "Name", "Weather").sort("Weather").show(10, truncate= False)
```

```
+----+-----------------------------------------------------------------+-------+
|Code|Name                                                             |Weather|
+----+-----------------------------------------------------------------+-------+
|MDW |Chicago, IL: Chicago Midway International                        |1      |
|PDX |Portland, OR: Portland International                             |2      |
|BWI |Baltimore, MD: Baltimore/Washington International Thurgood Marshall|2      |
|FLL |Fort Lauderdale, FL: Fort Lauderdale-Hollywood International      |2      |
|PDX |Portland, OR: Portland International                             |3      |
|IAD |Washington, DC: Washington Dulles International                  |3      |
|MCO |Orlando, FL: Orlando International                               |4      |
|PDX |Portland, OR: Portland International                             |4      |
|FLL |Fort Lauderdale, FL: Fort Lauderdale-Hollywood International      |4      |
|MDW |Chicago, IL: Chicago Midway International                        |4      |
+----+-----------------------------------------------------------------+-------+
only showing top 10 rows
```

Notice the presence of the parameter "**truncate= False**"

It allows us to avoid any truncation of the strings in any of the columns (or, attributes as we say in the context of data frames) so that we can read the particular cell and access its information completely

ANALYSING SEMI-STRUCTURED DATA USING SPARK SQL

```
# Sorting the airport delay statistics based on the Weather
airport_statistics.select("Code", "Name", "Weather").sort("Weather").show(10)
```

```
+----+--------------------+-------+
|Code|                Name|Weather|
+----+--------------------+-------+
| MDW|Chicago, IL: Chic...|      1|
| PDX|Portland, OR: Por...|      2|
| BWI|Baltimore, MD: Ba...|      2|
| FLL|Fort Lauderdale, ...|      2|
| PDX|Portland, OR: Por...|      3|
| IAD|Washington, DC: W...|      3|
| MCO|Orlando, FL: Orla...|      4|
| PDX|Portland, OR: Por...|      4|
| FLL|Fort Lauderdale, ...|      4|
| MDW|Chicago, IL: Chic...|      4|
+----+--------------------+-------+
only showing top 10 rows
```

Now, as we can see that the table provides the list of airports having the least number of delays due to adverse weather conditions.

Now, as we can see that the table provides the list of airports having the least number of delays due to adverse weather conditions.

However, it is interesting to note that we do not have complete names of the airports and rather we have to compromise by merely using the names of the location where such airports are located. How should we resolve this?

```
# Sorting the airport delay statistics based on the Weather
airport_statistics.select("Code", "Name", "Weather").sort("Weather").show(10, truncate= False)
```

```
+----+-------------------------------------------------------------------+-------+
|Code|Name                                                               |Weather|
+----+-------------------------------------------------------------------+-------+
|MDW |Chicago, IL: Chicago Midway International                          |1      |
|PDX |Portland, OR: Portland International                               |2      |
|BWI |Baltimore, MD: Baltimore/Washington International Thurgood Marshall|2      |
|FLL |Fort Lauderdale, FL: Fort Lauderdale-Hollywood International       |2      |
|PDX |Portland, OR: Portland International                               |3      |
|IAD |Washington, DC: Washington Dulles International                    |3      |
|MCO |Orlando, FL: Orlando International                                 |4      |
|PDX |Portland, OR: Portland International                               |4      |
|FLL |Fort Lauderdale, FL: Fort Lauderdale-Hollywood International       |4      |
|MDW |Chicago, IL: Chicago Midway International                          |4      |
+----+-------------------------------------------------------------------+-------+
only showing top 10 rows
```

But, wait !!
We wanted to analyse the data to obtain the list of airports that are least delayed due to adverse conditions. However, we can observe some of the airports are being repeated in the datasets which definitely is not going to provide us with the list of unique airports. How can be resolve this problem?

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Using groupby to find the list of least affected unique airports
from pyspark.sql.functions import col, sum, count
airport_statistics.select("Code", "Name", "Weather") \
    .groupBy("Code", "Name").agg(sum("Weather") \
    .alias("Weather")).sort(col("Weather")).show(10, truncate=False)
```

```
+----+-------------------------------------------------------------+-------+
|Code|Name                                                         |Weather|
+----+-------------------------------------------------------------+-------+
|PDX |Portland, OR: Portland International                         |2791   |
|SAN |San Diego, CA: San Diego International                       |5056   |
|FLL |Fort Lauderdale, FL: Fort Lauderdale-Hollywood International|5084   |
|TPA |Tampa, FL: Tampa International                               |5092   |
|IAD |Washington, DC: Washington Dulles International              |5849   |
|SEA |Seattle, WA: Seattle/Tacoma International                    |5953   |
|MIA |Miami, FL: Miami International                               |7487   |
|CLT |Charlotte, NC: Charlotte Douglas International               |7738   |
|MCO |Orlando, FL: Orlando International                           |8124   |
|DCA |Washington, DC: Ronald Reagan Washington National            |8504   |
+----+-------------------------------------------------------------+-------+
only showing top 10 rows
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Using groupby to find the list of most affected unique airports
from pyspark.sql.functions import col, sum, count
airport_statistics.select("Code", "Name", "Weather") \
    .groupBy("Code", "Name").agg(sum("Weather") \
    .alias("Weather")).sort(col("Weather").desc()).show(10, truncate=False)
```

```
+----+-------------------------------------------------------+-------+
|Code|Name                                                   |Weather|
+----+-------------------------------------------------------+-------+
|ATL |Atlanta, GA: Hartsfield-Jackson Atlanta International |40113  |
|DFW |Dallas/Fort Worth, TX: Dallas/Fort Worth International|30476  |
|ORD |Chicago, IL: Chicago O'Hare International              |24358  |
|LGA |New York, NY: LaGuardia                               |16350  |
|DEN |Denver, CO: Denver International                      |15556  |
|EWR |Newark, NJ: Newark Liberty International               |14668  |
|LAX |Los Angeles, CA: Los Angeles International            |14652  |
|IAH |Houston, TX: George Bush Intercontinental/Houston     |13062  |
|BOS |Boston, MA: Logan International                       |11955  |
|SFO |San Francisco, CA: San Francisco International        |11751  |
+----+-------------------------------------------------------+-------+
only showing top 10 rows
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Find the list of most affected airports due to delays caused by security
# Filtering rows using a WHERE clause
airport_statistics.select("Code", "Name", "Security") \
    .groupBy("Code", "Name").agg(sum("Security") \
    .alias("Security")).sort(col("Security").desc()) \
    .where((col("Code")== "ATL") | (col("Code")== "TPA")) \
    .show(10, truncate=False)
```

```
+----+-------------------------------------------------------+--------+
|Code|Name                                                   |Security|
+----+-------------------------------------------------------+--------+
|ATL |Atlanta, GA: Hartsfield-Jackson Atlanta International|1066    |
|TPA |Tampa, FL: Tampa International                         |500     |
+----+-------------------------------------------------------+--------+
```

```
# Find the list of most affected airports due to delays caused by security
# Filtering rows using a FILTER clause
airport_statistics.select("Code", "Name", "Security") \
    .groupBy("Code", "Name").agg(sum("Security") \
    .alias("Security")).sort(col("Security").desc()) \
    .filter((col("Code")== "ATL") | (col("Code")== "TPA")) \
    .show(10, truncate=False)
```

```
+----+-------------------------------------------------------+--------+
|Code|Name                                                   |Security|
+----+-------------------------------------------------------+--------+
|ATL |Atlanta, GA: Hartsfield-Jackson Atlanta International|1066    |
|TPA |Tampa, FL: Tampa International                         |500     |
+----+-------------------------------------------------------+--------+
```

USING DATA SOURCES

Introduction to JavaScript Notation Object (JSON)

```
# Find the list of most affected airports due to delays caused by security
# Filtering rows using a WHERE clause
airport_statistics.select("Code", "Name", "Security") \
    .groupBy("Code", "Name").agg(sum("Security") \
    .alias("Security")).sort(col("Security").desc()) \
    .where((col("Code")== "ATL") | (col("Code")== "TPA")) \
    .show(10, truncate=False)
```

```
+----+-------------------------------------------------------+--------+
|Code|Name                                                   |Security|
+----+-------------------------------------------------------+--------+
|ATL |Atlanta, GA: Hartsfield-Jackson Atlanta International|1066    |
|TPA |Tampa, FL: Tampa International                         |500     |
+----+-------------------------------------------------------+--------+
```

```
# Find the list of most affected airports due to delays caused by security
# Filtering rows using a WHERE clause

condition_1= (col("Code")== "ATL")
condition_2= (col("Code")== "TPA")

airport_statistics.select("Code", "Name", "Security") \
    .groupBy("Code", "Name").agg(sum("Security") \
    .alias("Security")).sort(col("Security").desc()) \
    .where(condition_1 | condition_2) \
    .show(10, truncate=False)
```

```
+----+-------------------------------------------------------+--------+
|Code|Name                                                   |Security|
+----+-------------------------------------------------------+--------+
|ATL |Atlanta, GA: Hartsfield-Jackson Atlanta International|1066    |
|TPA |Tampa, FL: Tampa International                         |500     |
+----+-------------------------------------------------------+--------+
```

BTS | Barcelona Technology School

Doing Basic Statistics

DOING BASIC STATISTICS

Making sense with Summary Statistics

Once again,
Let's start with the summary statistics. It'll allow us to examine the basic statistics related to how the data is organised.

```
from pyspark.ml.stat import *
from pyspark.ml.linalg import Vectors
from pyspark.sql import Row
credit= spark.read.csv('./data/german_credit.csv', sep= ',', header= True)
```

```
# Summary statistics
num_cols = ['Account Balance','No of dependents']
credit.select(num_cols).describe().show()
```

```
+-------+------------------+--------------------+
|summary|   Account Balance|   No of dependents|
+-------+------------------+--------------------+
|  count|              1000|                1000|
|   mean|             2.577|               1.155|
| stddev|1.2576377271108936|0.36208577175319395|
|    min|                 1|                   1|
|    max|                 4|                   2|
+-------+------------------+--------------------+
```

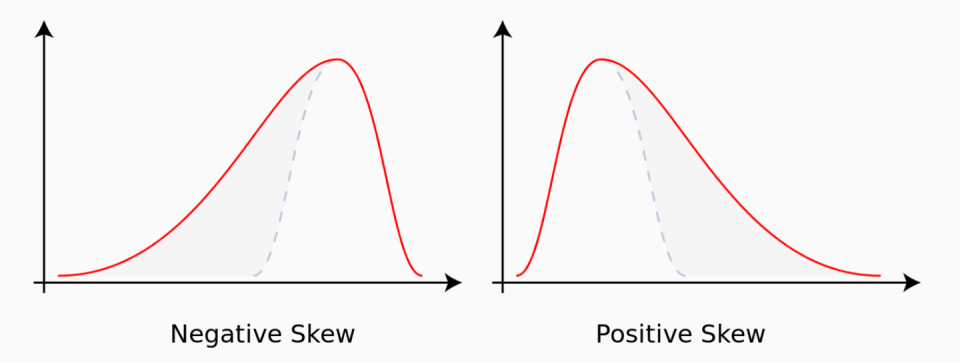
[1] Statistical and Mathematical Functions with DataFrames in Apache Spark
https://databricks.com/blog/2015/06/02/statistical-and-mathematical-functions-with-dataframes-in-spark.html

DOING BASIC STATISTICS

Detecting Skewness and Kurtosis

- Skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean. The skewness value can be positive or negative, or undefined. For a unimodal distribution, negative skew commonly indicates that the tail is on the left side of the distribution, and positive skew indicates that the tail is on the right.

- Kurtosis is a measure of the “tailedness” of the probability distribution of a real-valued random variable.



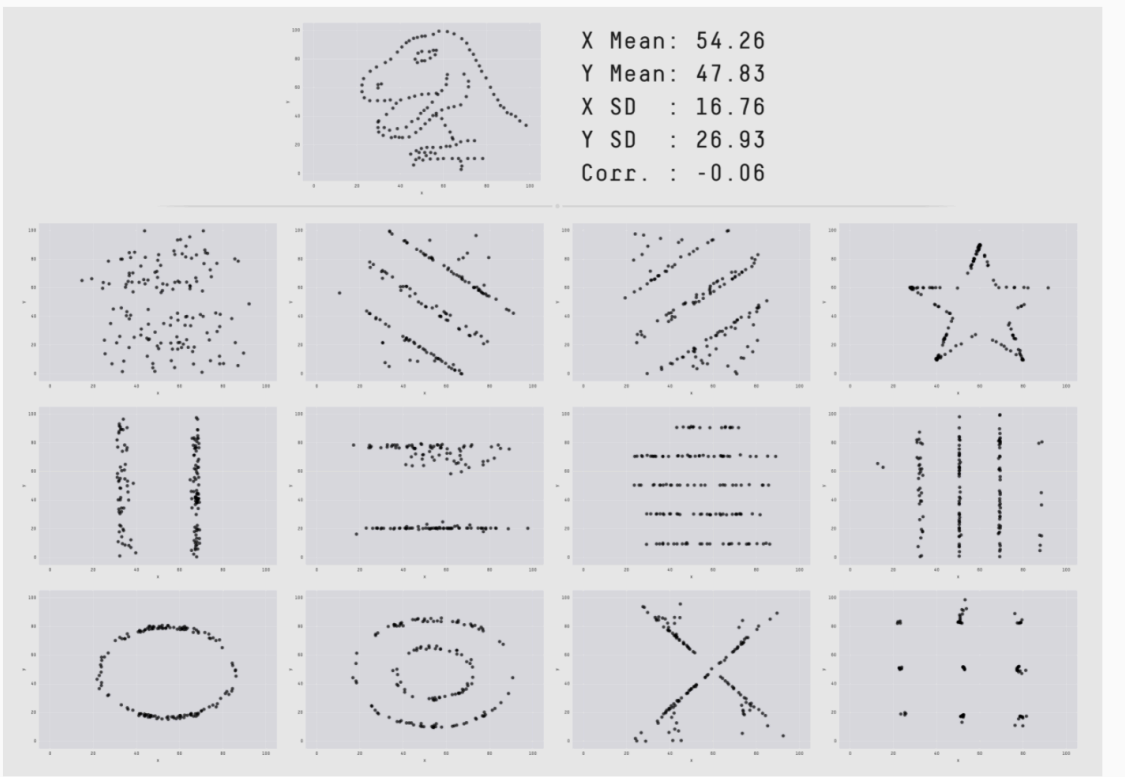

```
from pyspark.sql.functions import col, skewness, kurtosis
credit.select(skewness("Age (years)"),kurtosis("Age (years)")).show()
```

```
+--------------------+--------------------+
|skewness(Age (years))|kurtosis(Age (years))|
+--------------------+--------------------+
|   1.0231743160548064|   0.6114371688367672|
+--------------------+--------------------+
```

DOING BASIC STATISTICS

Statistics can be misleading too!!

F. J. Anscombe once said that make both calculations and graphs. Both sorts of output should be studied; each will contribute to understanding. These 13 datasets in Figure below (the Datasaurus, plus 12 others) each have the same summary statistics (x/y mean, x/y standard deviation, and Pearson's correlation) to two decimal places, while being drastically different in appearance.



Same Stats, Different Graphs

In such cases wherever, there is any suspicion, always compute statistics but do not forget to plot the data to see how the graph looks like.

[1] Same Stats Different Graphs

DOING BASIC STATISTICS

Computing Correlation

- Calculating the correlation between two series of data is a common operation in Statistics. In spark.ml we provide the flexibility to calculate pairwise correlations among many series. The supported correlation methods are currently Pearson's and Spearman's correlation.

- Correlation computes the correlation matrix for the input Dataset of Vectors using the specified method. The output will be a DataFrame that contains the correlation matrix of the column of vectors

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import Correlation

data = [(Vectors.sparse(4, [(0, 1.0), (3, -2.0)]),),
        (Vectors.dense([4.0, 5.0, 0.0, 3.0]),),
        (Vectors.dense([6.0, 7.0, 0.0, 8.0]),),
        (Vectors.sparse(4, [(0, 9.0), (3, 1.0)]),)]
df = spark.createDataFrame(data, ["features"])

r1 = Correlation.corr(df, "features").head()
print("Pearson correlation matrix:\n" + str(r1[0]))

r2 = Correlation.corr(df, "features", "spearman").head()
print("Spearman correlation matrix:\n" + str(r2[0]))
```

DOING BASIC STATISTICS

Chi-Square Test

- Skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean. The skewness value can be positive or negative, or undefined. For a unimodal distribution, negative skew commonly indicates that the tail is on the left side of the distribution, and positive skew indicates that the tail is on the right.
- Kurtosis is a measure of the “tailedness” of the probability distribution of a real-valued random variable.

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import ChiSquareTest

data = [(0.0, Vectors.dense(0.5, 10.0)),
        (0.0, Vectors.dense(1.5, 20.0)),
        (1.0, Vectors.dense(1.5, 30.0)),
        (0.0, Vectors.dense(3.5, 30.0)),
        (0.0, Vectors.dense(3.5, 40.0)),
        (1.0, Vectors.dense(3.5, 40.0))]
df = spark.createDataFrame(data, ["label", "features"])

r = ChiSquareTest.test(df, "features", "label").head()
print("pValues: " + str(r.pValues))
print("degreesOfFreedom: " + str(r.degreesOfFreedom))
print("statistics: " + str(r.statistics))
```

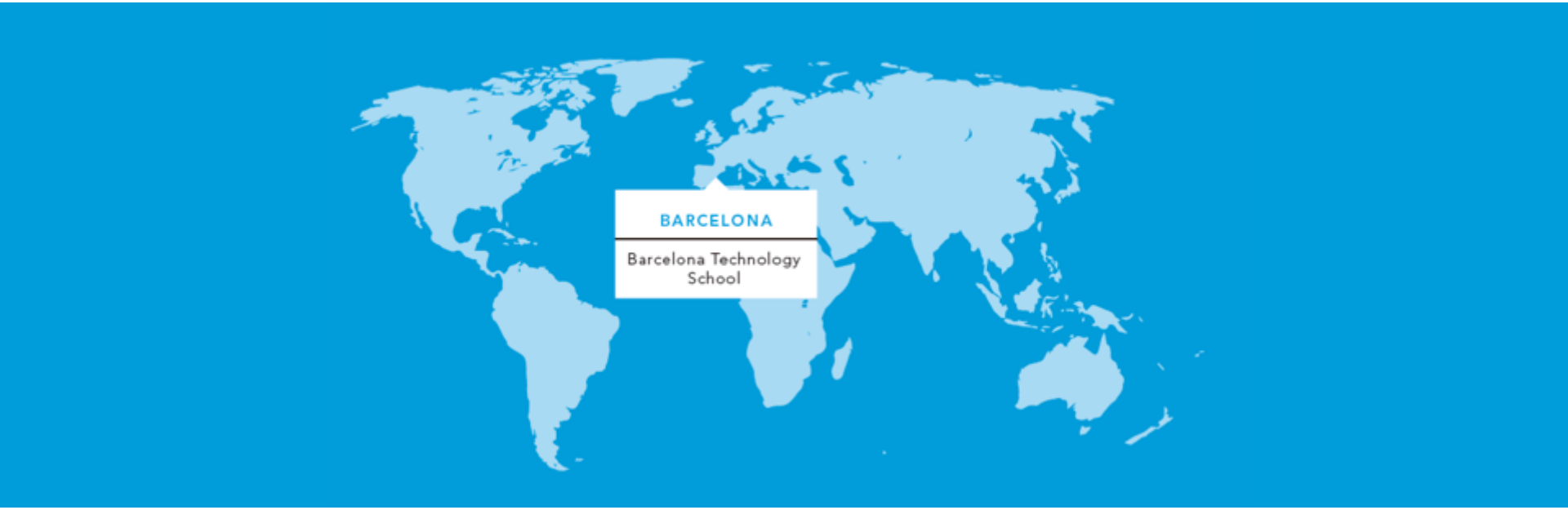
BTS | Barcelona Technology School

References

This course has been prepared by drawing some inspiration and references from many similar course. Some of the most similar have been highlighted below for the ready reference-

- CS105X BerkeleyX: Introduction to Apache Spark by UC Berkeley
- COMPSCI516: Apache Spark 101 by Duke University
- CS 4240: Large Scale Parallel Data Processing by Northeastern University
- EECS E6893: Big Data Analytics by Columbia University
- EECS E6895: Advanced Big Data Analytics by Columbia University
- STA 663**:** Computational Statistics and Statistical Computing (2018) by Duke University

Thanks a lot for your time!


BARCELONA
Barcelona Technology School