

# Spark SQL

# Spark SQL

- Spark SQL is the Spark component for structured data processing
- It provides a programming abstraction called Dataset and can act as a distributed SQL query engine
  - The input data can be queried by using
    - Ad-hoc methods
    - Or an SQL-like language

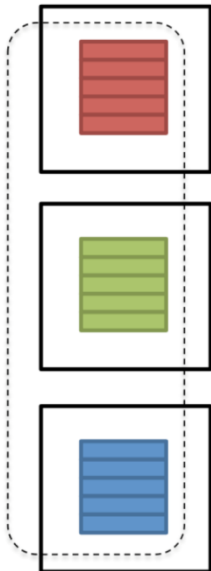
# Spark SQL vs Spark RDD APIs

- The interfaces provided by Spark SQL provide more information about the structure of both the data and the computation being performed
  - Spark SQL uses this extra information to perform extra optimizations based on an “SQL-like” optimizer called Catalyst
- => Programs based on **Datasets** are usually **faster than standard RDD**-based programs

# Spark SQL vs Spark RDD APIs

RDD

Unstructured



Distributed list of objects

vs

Dataset/DataFrame

Structured

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	43	B Jones
Chem	61	M Kennedy

~Distributed SQL table

# Datasets and DataFrames

- Dataset
  - Distributed collection of structured data
    - It provides the benefits of RDDs
      - Strong typing
      - Ability to use powerful lambda functions
    - And the benefits of Spark SQL's optimized execution engine exploiting the information about the data structure
      - Compute the best execution plan before executing the code

# Datasets and DataFrames

- DataFrame

- A “particular” Dataset organized into named columns
  - It is conceptually equivalent to a table in a relational database
  - It can be created reading data from different types of external sources (CSV files, JSON files, RDBMs, ..)
  - It is not characterized by the strong typing feature
- A DataFrame is simply a Dataset of Row objects
  - i.e., DataFrame is an alias for Dataset<Row>

# Spark Session

- All the Spark SQL functionalities are based on an instance of the `org.apache.spark.sql.Session` class
- To instance a Session object use the `Session.builder()` method

```
Session ss =  
Session.builder().appName("App.Name").getOrCreate();
```

# Spark Session

- To “close” a Spark Session use the **SparkSession.stop()** method

```
ss.stop();
```



# DataFrames

# DataFrames

- DataFrame
  - It is a distributed collection of data organized into named columns
  - It is equivalent to a relational table
- **DataFrames** are Datasets of Row objects, i.e., **Dataset<Row>**
- Classes used to define DataFrames
  - **org.apache.spark.sql.Dataset;**
  - **org.apache.spark.sql.Row;**

# DataFrames

- DataFrames can be constructed from different sources
  - Structured (textual) data files
    - E.g., csv files, json files
  - Existing RDDs
  - Hive tables
  - External relational databases

# Creating DataFrames from csv files

- Spark SQL provides an API that allows creating a DataFrame directly from CSV files
- Example of csv file
  - Name, Age
  - Andy, 30
  - Michael,
  - Justin, 19
- The file contains name and age of three persons
  - The age of the second person is unknown

# Creating DataFrames from csv files

- The creation of a DataFrame from a csv file is based the
  - `Dataset<Row> load(String path)` method of the `org.apache.spark.sql.DataFrameReader` class
    - Path is the path of the input file
  - And the `DataFrameReader read()` method of the `SparkSession` class

# Creating DataFrames from csv files: Example

- Create a DataFrame from a csv file containing the profiles of a set of persons
  - Each line of the file contains name and age of a person
    - Age can assume the null value
  - The first line contains the header, i.e., the name of the attributes/columns

# Creating DataFrames from csv files: Example

```
// Create a Spark Session object and set the name of the application  
SparkSession ss = SparkSession.builder().appName("Test  
SparkSQL").getOrCreate();
```

```
// Create a DataFrame from persons.csv  
DataFrameReader dfr=ss.read().format("csv").option("header",  
true).option("inferSchema", true);
```

```
Dataset<Row> df = dfr.load("persons.csv");
```

# Creating DataFrames from csv files: Example

```
// Create a Spark Session object and set the name of the application  
SparkSession ss = SparkSession.builder().appName("Test  
SparkSQL").getOrCreate();
```

```
// Create a DataFrame from persons.csv  
DataFrameReader dfr=ss.read().format("csv").option("header",  
true).option("inferSchema", true);
```

```
Dataset<Row> df = dfr.load("persons.csv");
```

This method is used to specify the format of the input file



# Creating DataFrames from csv files: Example

```
// Create a Spark Session object and set the name of the application  
SparkSession ss = SparkSession.builder().appName("Test  
SparkSQL").getOrCreate();
```

```
// Create a DataFrame from persons.csv  
DataFrameReader dfr=ss.read().format("csv").option("header",  
true).option("inferSchema", true);
```

```
Dataset<Row> df = dfr.load("persons.csv");
```

This method is used to specify that the first line of the file contains the name of the attributes/columns

# Creating DataFrames from csv files: Example

```
// Create a Spark Session object and set the name of the application  
SparkSession ss = SparkSession.builder().appName("Test  
SparkSQL").getOrCreate();
```

```
// Create a DataFrame from persons.csv  
DataFrameReader dfr=ss.read().format("csv").option("header",  
true).option("inferSchema", true);
```

```
Dataset<Row> df = dfr.load("persons.csv");
```

This method is used to specify that the system must infer the data types of each column. Without this option all columns are considered strings

# Creating DataFrames from JSON files

- Spark SQL provides an API that allows creating a DataFrame directly from a textual file where each line contains a JSON object
  - Hence, the **input file is not a “standard” JSON file**
    - It must be properly formatted in order to have **one JSON object (tuple) for each line**
  - The format of the input file is compliant with the **“JSON Lines text format”**, also called newline-delimited JSON

# Creating DataFrames from JSON files

- Example of JSON Lines text formatted file compatible with the Spark expected format

```
{"name":"Michael"}
```

```
{"name":"Andy", "age":30}
```

```
{"name":"Justin", "age":19}
```

- The example file contains name and age of three persons
  - The age of the first person is unknown

# Creating DataFrames from JSON files

- The creation of a DataFrame from JSON files is based on the same method used for reading csv files
  - **Dataset<Row> load(String path)** method of the **org.apache.spark.sql.DataFrameReader** class
    - Path is the path of the input file
  - And the **DataFrameReader read()** method of the **SparkSession** class
- The only difference is given by the parameter of the **format(string ..)** method

# Creating DataFrames from JSON files

- The same API allows also reading “standard” multiline JSON files
  - Set the multiline option to true by invoking `.option("multiline", true)` on the defined DataFrameReader for reading “standard” JSON files
  - This feature is available since Spark 2.2.0
- Pay attention that reading a set of **small JSON files** from HDFS is **very slow**

# Creating DataFrames from JSON files: Example 1

- Create a DataFrame from a JSON Lines text formatted file containing the profiles of a set of persons
  - Each line of the file contains a JSON object containing name and age of a person
    - Age can assume the null value

# Creating DataFrames from JSON files: Example 1

```
// Create a Spark Session object and set the name of the application  
SparkSession ss = SparkSession.builder().appName("Test  
SparkSQL").getOrCreate();
```

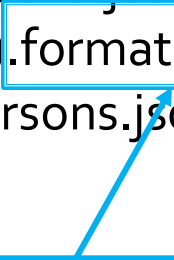
```
// Create a DataFrame from persons.json  
DataFrameReader dfr=ss.read().format("json");  
Dataset<Row> df = dfr.load("persons.json");
```



# Creating DataFrames from JSON files: Example 1

```
// Create a Spark Session object and set the name of the application  
SparkSession ss = SparkSession.builder().appName("Test  
SparkSQL").getOrCreate();
```

```
// Create a DataFrame from persons.json  
DataFrameReader dfr=ss.read().format("json");  
Dataset<Row> df = dfr.load("persons.json");
```



This method is used to specify the format of the input file

# Creating DataFrames from JSON files: Example 2

- Create a DataFrame from a folder containing a set of “standard” multiline JSON files
- Each input JSON file contains the profile of one person
  - Name and Age
    - Age can assume the null value

# Creating DataFrames from JSON files: Example 2

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();

// Create a DataFrame from persons.json
DataFrameReader dfr=ss.read().format("json")
                                .option("multiline", true);
Dataset<Row> df = dfr.load("folder_JSONFiles/");
```

# Creating DataFrames from JSON files: Example 2

```
// Create a Spark Session object and set the name of the application  
SparkSession ss = SparkSession.builder().appName("Test  
SparkSQL").getOrCreate();
```

```
// Create a DataFrame from persons.json  
DataFrameReader dfr=ss.read().format("json")  
    .option("multiline", true);  
Dataset<Row> df = dfr.load("folder_JSONFiles/");
```

This multiline option is set to true to specify that the input files are "standard" multiline JSON files

# Creating DataFrames from other data sources

- The DataFrameReader class (the same we used for reading a json file and store it in a DataFrame) provides other methods to read many standard (textual) formats and read data from external databases
  - Apache **parquet** files
  - External **relational database**, through a **JDBC** connection
  - **Hive** tables
  - Etc.

# From DataFrame to RDD

- The `JavaRDD<Row> javaRDD()` method of the `Dataset<Row>` class returns a JavaRDD of Row objects (JavaRDD<Row>) containing the content of the Dataset on which it is invoked
- Each `Row` object is like a vector containing the values of a record
  - It contains also the information about the schema of the data, i.e., the “name” of each cell of the vector

# From DataFrame to RDD

- Important methods of the **Row** class
  - int **fieldIndex(String columnName)**
    - It returns the index of a given field/column name
  - get methods
    - java.lang.Object **getAs(String columnName)**
      - Retrieve the content of a field/column given its name
    - String **getString(int position)**
    - double **getDouble(int position)**
    - ..
      - They retrieve the value of the field/column at position **position**

# From DataFrame to RDD: Example

- Create a DataFrame from a csv file containing the profiles of a set of persons
  - Each line of the file contains name and age of a person
  - The first line contains the header, i.e., the name of the attributes/columns
- Transform the input DataFrame into a JavaRDD, select only the name field/column and store it in the output folder



# From DataFrame to RDD: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Create a DataFrame from persons.csv
DataFrameReader dfr=ss.read().format("csv").option("header",
true).option("inferSchema", true);
```

```
Dataset<Row> df = dfr.load("persons.csv");
```

# From DataFrame to RDD: Example

```
// Define an JavaRDD based on the content of the DataFrame
JavaRDD<Row> rddPersons = df.javaRDD();

// Use the map transformation to extract the name field/column
JavaRDD<String> rddNames = rddPersons
    .map(inRow -> (String)inRow.getAs("name"));

// Store the result
rddNames.saveAsTextFile(outputPath);
```

# From DataFrame to RDD: Example

## – version based on getString()

```
// Define an JavaRDD based on the content of the DataFrame
JavaRDD<Row> rddPersons = df.javaRDD();

// Use the map transformation to extract the name field/column
JavaRDD<String> rddNames = rddPersons
    .map(inRow -> inRow.getString(inRow.fieldIndex("name")));

// Store the result
rddNames.saveAsTextFile(outputPath);
```

# From DataFrame to RDD: Example

## – version based on getString()

```
// Define an JavaRDD based on the content of the DataFrame  
JavaRDD<Row> rddPersons = df.javaRDD();
```

```
// Use the map transformation to extract the name field/column  
JavaRDD<String> rddNames = rddPersons  
    .map(inRow -> inRow.getString(inRow.fieldIndex("name")));
```

```
// Store the result  
rddNames.saveAsTextFile("hdfs://...");
```

Retrieve the position (index) of field/column name

A blue rectangular callout box with a blue border contains the text "Retrieve the position (index) of field/column name". A blue arrow points from the bottom-left corner of this box to the "inRow.fieldIndex(\"name\")" expression in the code line above.

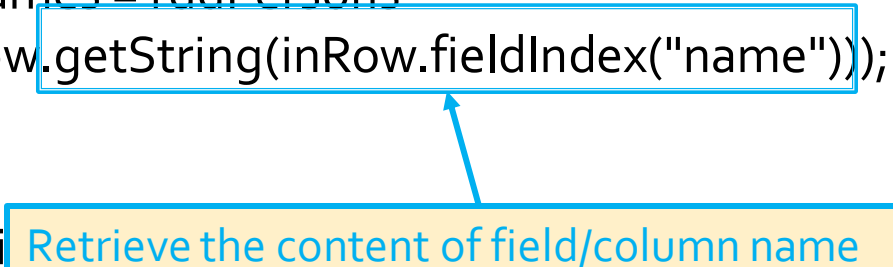
# From DataFrame to RDD: Example

## – version based on getString()

```
// Define an JavaRDD based on the content of the DataFrame
JavaRDD<Row> rddPersons = df.javaRDD();

// Use the map transformation to extract the name field/column
JavaRDD<String> rddNames = rddPersons
    .map(inRow -> inRow.getString(inRow.fieldIndex("name")));

// Store the result
rddNames.saveAsTextFile
```



Retrieve the content of field/column name

# Datasets

# Datasets

- **Datasets** are more general than DataFrames
  - Datasets are collections of “objects”
    - All the objects of the same dataset are associated with the same class
    - The structure/schema of the represented data is consistent with the attributes of the class of the contained objects
- DataFrame is an “alias” of Dataset<Row>

# Datasets

- Datasets are similar to RDDs
- However, **Datasets are more efficient than RDDs**
  - They use specialized Encoders to serialize the objects for processing or transmitting them over the network
    - Those encoders allows Spark to perform operations like filtering, sorting and hashing without deserializing the objects
  - The code based on Datasets is optimized by means of the Catalytic optimizer which exploits the schema of the data



# Datasets

- The objects stored in the Spark Datasets must be **JavaBean-compliant**
- Specifically,
  - The Java class associated with a JavaBean must implement Serializable
  - All its attributes/variables must have public setter and getter methods
  - All its attributes/variables should be private

# Creating Datasets from local collections

- Spark SQL provides a method that allows creating a Dataset from a local collection  
**`Dataset<T> createDataset(java.util.List<T> data, Encoder<T> encoder)`**
  - T is the data type (class) of the input elements
  - data is the local input list
  - An instance of the encoder associated with the stored T objects must be provided

# Creating Datasets from local collections: Example 1

- Create a Dataset<Person> from a local list of objects of type Person
- Person is a Java Bean class containing name and age of a single person
- Suppose the following persons are stored in the input list
  - Paolo, 40
  - Giovanni, 30
  - Luca, 32

# Creating Datasets from local collections: Example 1

```
public class Person implements Serializable {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

# Creating Datasets from local collections: Example 1

```
public class Person implements Serializable {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return name;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

Each instance of this class is characterized by name and age.

The schema of the Dataset based on objects of this class is: name, age

# Creating Datasets from local collections: Example 1

```
public class Person implements Serializable {  
    private String name;  
    private int age;
```

Setter and getter methods must be defined

```
        public String getName() {  
            return name;  
        }  
        public void setName(String name) {  
            this.name = name;  
        }  
  
        public int getAge() {  
            return age;  
        }  
        public void setAge(int age) {  
            this.age = age;  
        }  
    }  
}
```

# Creating Datasets from local collections: Example 1

```
// Create a Spark Session object and set the name of the application  
SparkSession ss = SparkSession.builder().appName("Test  
SparkSQL").getOrCreate();
```

# Creating Datasets from local collections: Example 1

```
// Create a local array of Persons  
ArrayList<Person> persons =new ArrayList<Person>();
```

```
Person person;
```

```
person = new Person();  
person.setName("Paolo");  
person.setAge(40);  
persons.add(person);
```

```
person = new Person();  
person.setName("Giovanni");  
person.setAge(30);  
persons.add(person);
```

```
person = new Person();  
person.setName("Luca");  
person.setAge(32);  
persons.add(person);
```



# Creating Datasets from local collections: Example 1

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);

// Define the Dataset based on the content of the local list of persons
Dataset<Person> personDS = ss.createDataset(persons, personEncoder);
```

# Default encoders

- The static methods of the `org.apache.spark.sql.Encoders` class can be used to define Encoders for the basic data types/classes
  - `Encoder<Integer> Encoders.INT()`
  - `Encoder<String> Encoders.STRING()`
  - ...

# Creating Datasets from local collections: Example 2

- Create a Dataset<Integer> from a local list of objects of type Integer
- Suppose the following integer values are stored in the input list
  - 40
  - 30
  - 32

# Creating Datasets from local collections: Example 2

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();

// Create a local list of integers
List<Integer> values = Arrays.asList(40, 30, 32);

// Encoders for most common types are provided in class Encoders
Dataset<Integer> primitiveDS =
    ss.createDataset(values, Encoders.INT());
```

# Creating Datasets from DataFrames

- DataFrames (i.e., Datasets of Row objects) can be converted into a “typed” Dataset
- The **Dataset<T> as(Encoder encoder)** method must be used

# Creating Datasets from DataFrames

- Create a Dataset<Person> from an input DataFrame
  - The content of the input DataFrame is loaded for the input file persons.csv
- Content of persons.csv
  - Name, Age
  - Andy, 30
  - Michael,
  - Justin, 19

# Creating Datasets from DataFrames

```
public class Person implements Serializable {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

# Creating Datasets from DataFrames

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Read the content of the input file and store it into a DataFrame
DataFrameReader dfr=ss.read().format("csv").option("header",
                                                    true).option("inferSchema", true);
Dataset<Row> df = dfr.load("persons.csv");
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Define a Dataset of Person objects from the df DataFrame
Dataset<Person> ds = df.as(personEncoder);
```



# Creating Datasets from DataFrames

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Read the content of the input file and store it into a DataFrame
DataFrameReader dfr=ss.read().format("csv").option("header",
                                                    true).option("inferSchema", true);
```

Instantiate an Encoder that is used to serialize Person objects

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Define a Dataset of Person objects from the df DataFrame
Dataset<Person> ds = df.as(personEncoder);
```

# Creating Datasets from DataFrames

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Read the content of the input file and store it into a DataFrame
DataFrameReader dfr=ss.read().format("csv").option("header",
                                                    true).option("inferSchema", true);
Dataset<Row> df = dfr.load("persons.csv");
```

```
// Define a Dataset of Person objects from the df DataFrame
Dataset<Person> ds = df.as(personEncoder);
```

This method is used to specify the type of the elements stored in the returned Dataset

```
// Define a Dataset of Person objects from the df DataFrame
Dataset<Person> ds = df.as(personEncoder);
```

# Creating Datasets from CSV or JSON files

- Define Datasets from CSV or JSON files
  - Define a DataFrame based on the input CSV/JSON files
  - “Convert” it into a Dataset by using the `as()` method and an Encoder object
    - Like we did in the previous example

# Creating Datasets for RDDs

- Spark SQL provides also a version of the createDataset method that allows creating a Dataset from an RDD

**Dataset<T> createDataset(RDD<T> inRDD, Encoder<T> encoder)**

- inRDD is the input RDD to be converted in a Dataset
- **Pay attention** that the first parameter is a **scala RDD** and not a JavaRDD
  - Use JavaRDD.toRDD(inJavaRDD) to convert a JavaRDD into a scala RDD

# Operations on Datasets and DataFrames

# Datasets operations

- A set of methods, implementing a set of specific operations, are available for the Datasets class
  - E.g., `show()`, `printSchema()`, `count()`, `distinct()`, `select()`, `filter()`
- Also a subset of the standard transformations of the RDDs are available
  - `filter()`, `map()`, `flatMap()`, ..

# Show

- The **void show(int numRows)** method of the **Dataset** class prints on the standard output the first numRows of the input Dataset
- The **void show()** method of the **Dataset** class prints on the standard output all the rows of the input Dataset

# Show: Example

- Create a Dataset from a csv file containing the profiles of a set of persons
  - The content of persons.csv is  
Name, Age  
Andy, 30  
Michael,  
Justin, 19
- Print the content of the first 2 persons (i.e., the first 2 rows of the Dataset)



# Show: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
dataset
DataFrameReader dfr=ss.read().format("csv").option("header",
true).option("inferSchema", true);
Dataset<Person> ds = dfr.load("persons.csv").as(personEncoder);
```

```
// Print, on the standard output, 2 rows of the DataFrame
ds.show(2);
```

# PrintSchema

- The **void printSchema()** method of the **Dataset** class prints on the standard output the schema of the Dataset
  - i.e., the name of the attributes of the data stored in the Dataset

# PrintSchema: Example

- Create a Dataset from a csv file containing the profiles of a set of persons
  - The content of persons.csv is  
Name,Age  
Andy,30  
Michael,  
Justin,19
- Print the schema of the created Dataset

# PrintSchema: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
dataset
DataFrameReader dfr=ss.read().format("csv").option("header",
true).option("inferSchema", true);
Dataset<Person> ds = dfr.load("persons.csv").as(personEncoder);
```

```
// Print, on the standard output, the schema of the DataFrame
ds.printSchema();
```

# Count

- The **long count()** method of the **Dataset** class returns the number of rows in the input Dataset

# Count: Example

- Create a Dataset from a csv file containing the profiles of a set of persons
  - The content of persons.csv is  
Name,Age  
Andy,30  
Michael,  
Justin,19
- Print the number of persons (i.e., rows) in the created Dataset

# Count: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
dataset
DataFrameReader dfr=ss.read().format("csv").option("header",
true).option("inferSchema", true);
Dataset<Person> ds = dfr.load("persons.csv").as(personEncoder);
```

```
// Print, on the standard output, the number of persons
System.out.println("The input file contains "+ds.count()+" persons");
```

# Distinct

- The **Dataset distinct()** method of the **Dataset** class returns a new Dataset that contains only the unique rows of the input Dataset
  - Pay attention that the distinct operation is always an heavy operation in terms of data sent on the network
  - A shuffle phase is needed



# Distinct: Example

- Create a Dataset from a csv file containing the names of a set of persons
  - The content of names.csv is
    - Name
    - Andy
    - Michael
    - Justin
    - Michael
- Create a new Dataset without duplicates

# Distinct: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Read the content of the input file and store it in a Dataset<String>
// dataset
DataFrameReader dfr=ss.read().format("csv").option("header",
true).option("inferSchema", true);
Dataset<String> ds = dfr.load("names.csv").as(Encoders.STRING());
```

```
// Create a new Dataset without duplicates
Dataset<String> distinctNames=ds.distinct();
```

# Select

- The **Dataset<Row> select(String col<sub>1</sub>, .., String col<sub>n</sub>)** method of the **Dataset** class returns a new Dataset that contains only the specified columns of the input Dataset
- Pay attention that the **type** of the **returned** Dataset is **Dataset<Row>, i.e., a DataFrame**
  - If you need/want you can convert it to a Dataset of a different data type by using the **as()** method and an encoder

# Select

- Pay attention that the select method **can generate errors at runtime** if there are mistakes in the names of the columns

# Select: Example

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - One line per person
    - Each line contains name, age, and gender of a person
- Create a new Dataset containing only name and age of the persons

# Select: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

```
// Create a new Dataset containing only name and age of the persons
Dataset<Row> dfNamesAges = ds.select("name", "age");
```

# SelectExpr

- The **Dataset<Row> selectExpr(String expression1, .., String expressionN)** method of the **Dataset** class returns a new Dataset that contains a set of columns computed by combining the original columns
- Pay attention that
  - The **type** of the **returned** Dataset is **Dataset<Row>, i.e., a DataFrame**
  - This method **can generate errors at runtime** if there are typos in the expressions

# SelectExpr: Example

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a new DataFrame containing the same columns of the initial dataset and a new one associated with value of age incremented by one
  - The column associated with age+1 is renamed newAge in the returned DataFrame



# SelectExpr: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

```
// Create a new Dataset containing name, age, gender, and age+1 for
each person
Dataset<Row> df =
    ds.selectExpr("name", "age", "gender", "age+1 as newAge");
```

# SelectExpr: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
```

This part of the expression is used to specify the name of the column associated with the result of the first part of the expression in the returned dataset. Without this part of the expression, the name of the returned column will be "age+1"

each person

```
Dataset<Row> df =
    ds.selectExpr("name", "age", "gender", "age+1 as newAge");
```

# Map

- The `Dataset<U> map(scala.Function1<T,U> func, Encoder<U> encoder)` method of the `Dataset` class returns a new `Dataset`
  - Each element is obtained by applying the specified function on one element the input `Dataset`
  - **Returns a `Dataset`**
- The map transformation of the `Dataset` class is similar to the `map(..)` transformation of standard `RDDs`
  - The only difference is given by the `Encoder` parameter that is used to encode the returned objects

# Map

- This method can be used instead of `select(..)` and `selectExpr(..)` to select a subset of the input columns or a combinations of them
- The advantage is that we can identify “semantic” errors at compile time with this method
  - E.g., wrong fields/columns names are not allowed

# Map: Example

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a new DataFrame containing the same columns of the initial dataset and a new one associated with value of age incremented by one

# Map: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

# Map: Example

```
// Create a new Dataset containing name, surname and age+1
// for each person
Dataset<PersonNewAge> ds2 = ds
    .map(p -> {
        PersonNewAge newPersonNA =
            new PersonNewAge();
            newPersonNA.setName(p.getName());
            newPersonNA.setAge(p.getAge());
            newPersonNA.setGender(p.getGender());
            newPersonNA.setNewAge(p.getAge()+1);
            return newPersonNA; }
    , Encoders.bean(PersonNewAge.class));
```

# Map: Example

```
// Create a new Dataset containing name, surname and age+1  
// for each person
```

```
Dataset<PersonNewAge> ds2 = ds
```

```
.map(p -> {  
    PersonNewAge newPersonNA =  
        new PersonNewAge();  
    newPersonNA.setName(p.getName());  
    newPersonNA.setAge(p.getAge());  
    newPersonNA.setGender(p.getGender());  
    newPersonNA.setNewAge(p.getAge()+1);  
    return newPersonNA;  
}, Encoders.bean(PersonNewAge.class));
```

This lambda function returns for each input object a new object with the previous columns and a new one called `newAge` with value `age+1`



# Map: Example

```
// Create a new Dataset containing name, surname and age+1
```

```
// for each person
```

```
Dataset<PersonNewAge> ds2 = ds
```

```
.map(p -> {  
    PersonNewAge newPersonNA =  
        new PersonNewAge();  
        newPersonNA.setName(p.getName());  
        newPersonNA.setAge(p.getAge());  
        newPersonNA.setGender(p.getGender());  
        newPersonNA.setNewAge(p.getAge()+1);  
        return newPersonNA; }  
    , Encoders.bean(PersonNewAge.class));
```

**Note:** In the lambda function the getter and setter methods of the Person class are used => **We cannot access fields/columns that do not exist.**

# Map: Example

```
// Create a new Dataset containing name, surname and age+1
// for each person
Dataset<PersonNewAge> ds2 = ds
    .map(p -> {
        PersonNewAge newPersonNA =
            new PersonNewAge();
            newPersonNA.setName(p.getName());
            newPersonNA.setAge(p.getAge());
            newPersonNA.setGender(p.getGender());
            newPersonNA.setNewAge(p.getAge()+1);
            return newPersonNA; }
    , Encoders.bean(PersonNewAge.class));
```

The encoder for the returned object is the second parameter of this map transformation

# Map and Type-safety

- Pay attention that this `map(.., ..)` transformation checks also partially the “semantic” of the query **at compile time**
  - The code of the function cannot access columns that do not exist because we use the getter and setter methods of the two classes associated with the input and output objects to access the columns of the data
  - This is **not true for DataFrames/Dataset<Row>** because we use a different approach to access the columns of DataFrames/Dataframe<Row>
    - The `get(int index)` methods are used to retrieve the content of the columns from Row objects
    - We do not have a specific method associate with each column

# Filter

- The **Dataset filter(String conditionExpr)** method of the **Dataset** class returns a new Dataset that contains only the rows satisfying the specified condition
  - The condition is a Boolean expression where each atom of the expression is a comparison between an attribute and a value or between two attributes
  - Pay attention that this version of the filter method **can generate errors at runtime** if there are errors in the filter expression
    - The parameter is a string and the system cannot check the correctness of the expression at compile time

# Filter: Example

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a new Dataset containing only the persons with age between 20 and 31

# Filter: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

```
// Select the persons with age between 20 and 31
Dataset<Person> dsSelected = ds.filter("age >= 20 and age <= 31");
```

# Filter with lambda function

- The **Dataset filter(FilterFunction<T> func)** method of the **Dataset** class returns a new Dataset that contains only the elements for which the specified function returns true
  - It is similar to the filter transformation of RDD

# Filter with lambda function and Type-safety

- Pay attention that this version of the filter transformation checks also partially the “semantic” of the query at compile time
  - The code of the function cannot access columns that do not exist because we use the getter and setter methods of the two classes associated with the input and output objects to access the columns of the data
  - This is not true for DataFrames/Dataframe<Row> because we use a different approach to access the columns of DataFrames/Dataframe<Row>
    - The get(int index) methods are used to retrieve the content of the columns from Row objects
    - We do not have a specific method associate with each column



# Filter with lambda function: Example

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a new Dataset containing only the persons with age between 20 and 31 and print them on the standard output

# Filter with lambda function: Example

```
// Create a Spark Session object and set the name of the application  
SparkSession ss = SparkSession.builder().appName("Test  
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects  
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>  
// dataset  
Dataset<Person> ds = ss.read().format("csv").option("header", true)  
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

# Filter with lambda function: Example

```
// Select the persons with age between 20 and 31
Dataset<Person> dsSelected =
    ds.filter(p -> {
        if (p.getAge() >= 20 && p.getAge() <= 31)
            return true;
        else
            return false;
    });
```

# Filter with lambda function: Example

```
// Select the persons with age between 20 and 31
```

```
Dataset<Person> dsSelected =
```

```
    ds.filter(p -> {
```

```
        if (p.getAge()>=20 && p.getAge()<=31)  
            return true;
```

```
        else
```

```
            return false;
```

```
    });
```

**Note:** In the lambda function the getter and setter methods of the Person class are used => **We cannot access fields/columns that do not exist.**

# Where

- The **Dataset where(String expression)** method of the **Dataset** class is an **alias** of the **filter(String conditionExpr)** method

# Join (inner join)

- The **Dataset<Row> join(Dataset<T> right, Column joinExprs)** method of the **Dataset** class is used to join two Datasets
  - It returns a Dataset<Row> (i.e., a DataFrame) that contains the join of the tuples of the two input Datasets based on the joinExprs join condition
- **Pay attention** that this method
  - **Can generate errors at runtime** if there are errors in the join expression
  - Returns a **DataFrame**

# Join (inner join): Example

- Create two Datasets
  - One based on the persons\_id.csv file that contains the profiles of a set of persons
    - Schema: uid,name,age
  - One based on the liked\_sports.csv file that contains the liked sports for each person
    - Schema: uid,sportname
- Join the content of the two Datasets and show it on the standard output

# Join (inner join): Example

```
public class PersonID implements Serializable {  
    private String name; private int age; private int uid;  
  
    public String getName() {  
        return name;    }  
    public void setName(String name) { this.name = name; }  
  
    public int getAge() { return age; }  
  
    public void setAge(int age) { this.age = age; }  
  
    public int getUid() { return uid; }  
  
    public void setUid(int uid) { this.uid = uid; }  
}
```



# Join (inner join): Example

```
public class UIDSport implements Serializable {  
    private int uid; private String sportname;  
  
    public String getSportname() { return sportname; }  
  
    public void setSportname(String sportname) {    this.sportname =  
        sportname; }  
  
    public int getUid() { return uid; }  
  
    public void setUid(int uid) { this.uid = uid; }  
}
```

# Join (inner join): Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();

// Define the encoder that is used to serialize PersonID objects
Encoder<PersonID> personIDEncoder = Encoders.bean(PersonID.class);

// Read persons_id.csv and store it in a Dataset<PersonID>
Dataset<PersonID> dsPersons = ss.read().format("csv")
    .option("header", true).option("inferSchema", true)
    .load("persons_id.csv").as(personIDEncoder);
```

# Join (inner join): Example

```
// Define the encoder that is used to serialize UIDSport objects
Encoder<UIDSport> uidSportEncoder = Encoders.bean(UIDSport.class);

// Read liked_sports.csv and store it in a Dataset<UIDSport>
Dataset<UIDSport> dsUidSports = ss.read().format("csv")
    .option("header", true).option("inferSchema", true)
    .load("liked_sports.csv").as(uidSportEncoder);

// Join the two input Datasets
Dataset<Row> dfPersonLikes = dsPersons.join(dsUidSports,
    dsPersons.col("uid").equalTo(dsUidSports.col("uid")));

// Print the result on the standard output
dfPersonLikes.show();
```

# Join (inner join): Example

```
// Define the encoder that is used to serialize UIDSport objects
Encoder<UIDSport> uidSportEncoder = Encoders.bean(UIDSport.class);

// Read liked_sports.csv and store it in a Dataset<UIDSport>
Dataset<UIDSport> dsUidSports = ss.read().format("csv")
    .option("header", true).option("inferSchema", true)
    .load("liked_sports.csv").as(uidSportEncoder);

// Join the two
Dataset<Row> dfPersonLikes = dsPersons.join(dsUidSports,
    dsPersons.col("uid").equalTo(dsUidSports.col("uid")));

// Print the result on the standard output
dfPersonLikes.show();
```

Specify the natural join condition on the uid columns

# Other Join Types

- Spark supports several join types
  - inner, outer, full, fullouter, leftouter, left, rightouter, right, leftsemi, leftanti, cross
  - Default join type: inner
  - leftanti is useful to implement the subtract operation

# Other Join Types

- The `Dataset<Row> join(Dataset<T> right, Column joinExprs, String joinType)` method of the `Dataset` class is used to join two Datasets based on the join type `joinType`
  - It returns a `Dataset<Row>` (i.e., a `DataFrame`) that contains the join of the tuples of the two input Datasets based on the `joinExprs` join condition and the `joinType` join type
- **Pay attention** that this method
  - **Can generate errors at runtime** if there are errors in the join expression
  - Returns a `DataFrame`

# Other Join Types: Example

- Create two Datasets
  - One based on the persons\_id.csv file that contains the profiles of a set of persons
    - Schema: uid,name,age
  - One based on the banned.csv file that contains the banned users
    - Schema: uid,bannedmotivation
- Select the profiles of the non-banned users and show them on the standard output

# Other Join Types: Example

```
public class PersonID implements Serializable {  
    private String name; private int age; private int uid;  
  
    public String getName() { return name; }  
  
    public void setName(String name) { this.name = name; }  
  
    public int getAge() { return age; }  
  
    public void setAge(int age) { this.age = age; }  
  
    public int getUid() { return uid; }  
  
    public void setUid(int uid) { this.uid = uid; }  
}
```



# Other Join Types: Example

```
public class Banned implements Serializable {  
    private int uid; private String bannedmotivation;  
  
    public int getUid() { return uid; }  
  
    public void setUid(int uid) { this.uid = uid; }  
  
    public String getBannedmotivation() { return bannedmotivation; }  
  
    public void setBannedmotivation(String bannedmotivation) {  
        this.bannedmotivation = bannedmotivation;  
    }  
}
```

# Other Join Types: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test SparkSQL Anti-
Join").getOrCreate();
```

```
// Read persons_id.csv and store it in a Dataset<PersonID> dataset
Dataset<PersonID> dsProfiles = ss.read().format("csv")
    .option("header", true).option("inferSchema", true)
    .load("persons_id.csv").as(Encoders.bean(PersonID.class));
```

```
// Read banned.csv and store it in a Dataset<Banned> dataset
Dataset<Banned> dsBanned = ss.read().format("csv")
    .option("header", true).option("inferSchema", true)
    .load("banned.csv").as(Encoders.bean(Banned.class));
```

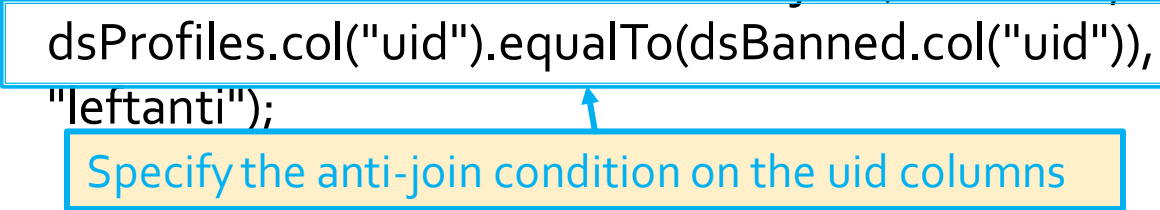
# Other Join Types: Example

```
// Apply a left-anti join to select the records of the users
// occurring only in dsProfiles and not in dsBanned
Dataset<Row> nonBannedProfiles = dsProfiles.join(dsBanned,
        dsProfiles.col("uid").equalTo(dsBanned.col("uid")),
        "leftanti");

// Print the result on the standard output
nonBannedProfiles.show();
```

# Other Join Types: Example

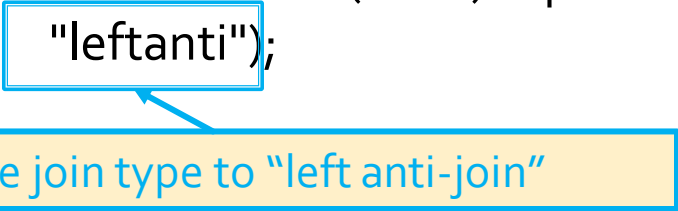
```
// Apply a left-anti join to select the records of the users
// occurring only in dsProfiles and not in dsBanned
Dataset<Row> nonBannedProfiles = dsProfiles.join(dsBanned,
    dsProfiles.col("uid").equalTo(dsBanned.col("uid")),
    "leftanti");
// Print the result on the standard output
nonBannedProfiles.show();
```

A blue-bordered callout box with a yellow background contains the text "Specify the anti-join condition on the uid columns". A blue arrow points from the bottom of this box to the join condition in the code above it.

Specify the anti-join condition on the uid columns

# Other Join Types: Example

```
// Apply a left-anti join to select the records of the users  
// occurring only in dsProfiles and not in dsBanned  
Dataset<Row> nonBannedProfiles = dsProfiles.join(dsBanned,  
    dsProfiles.col("uid").equalTo(dsBanned.col("uid")),  
    "leftanti");  
nonBannedProfiles.show();
```



Set the join type to "left anti-join"

# Aggregates functions

- Aggregate functions are provided to compute aggregates over the set of values of columns
- Some of the provided aggregate functions/methods are:
  - `avg(column)`, `count(column)`, `sum(column)`, `abs(column)`, etc.
  - See the static methods of the [org.apache.spark.sql.functions](#) class for a complete list
  - Pay attention that these are static methods of the [functions](#) class

# Aggregates functions

- The **agg(*aggregate functions*)** method of the Dataset class is used to specify which aggregate functions we want to apply
  - The result is a Dataset<Row>, i.e., a DataFrame
  - We can apply multiple aggregate functions at the same time by specifying a list of functions
- Pay attention that this methods **can generate errors at runtime** if there are semantic errors
  - E.g., wrong attribute names, wrong data types

# Aggregates functions: Example 1

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a Dataset containing the average value of age



# Aggregates functions: Example 1

- Input file

name,age

Andy,30

Michael,15

Justin,19

Andy,40

- Expected output

avg(age)

26.0

# Aggregates functions: Example 1

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

```
// Compute the average of age
Dataset<Row> averageAge = ds.agg(avg("age"));
```

# Aggregates functions: Example 2

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a Dataset containing the average value of age and the number of records (i.e., lines)

# Aggregates functions: Example 2

- Input file

name,age

Andy,30

Michael,15

Justin,19

Andy,40

- Expected output

avg(age),count(\*)

26.0,4

# Aggregates functions: Example 2

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

```
// Compute average and number of records
// We return two columns (one for each aggregate function)
Dataset<Row> avgAgeCount = ds.agg(avg("age"), count("*"));
```

# groupBy and aggregates functions

- The method **RelationalGroupedDataset** **groupBy(String col<sub>1</sub>, .., String col<sub>n</sub>)** method of the **Dataset** class combined with a set of aggregate methods of the **RelationalGroupedDataset** class can be used to split the input data in groups and compute aggregate function over each group
- Pay attention that this methods **can generate errors at runtime** if there are semantic errors
  - E.g., wrong attribute names, wrong data types

# groupBy and aggregates functions

- Specify which attributes are used to split the input data in groups by using the **RelationalGroupedDataset groupBy(String col1, .., String coln)** method
- Then, apply the aggregate functions you want to compute by final result
  - The result is a Dataset<Row>, i.e., a DataFrame

# groupBy and aggregates functions

- Some of the provided aggregate functions/methods are
  - avg(column), count(column), sum(column), abs(column), etc.
  - The agg(..) method can be used to apply multiple aggregate functions at the same time over each group
- See the static methods of the [org.apache.spark.sql.functions](#) class for a complete list



# groupBy and aggregates functions:

## Example 1

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a Dataset containing for each name the average value of age

# groupBy and aggregates functions: Example 1

- Input file

name,age

Andy,30

Michael,15

Justin,19

Andy,40

- Expected output

name,avg(age)

Andy,35

Michael,15

Justin,19

# groupBy and aggregates functions:

## Example 1

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

# groupBy and aggregates functions:

## Example 1

```
// Group data by name
RelationalGroupedDataset rgd=ds.groupBy("name");

// Compute the average of age for each group
Dataset<Row> nameAverageAge = rgd.avg("age");
```

# groupBy and aggregates functions:

## Example 2

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a Dataset containing for each name the average value of age and the number of person with that name

# groupBy and aggregates functions: Example 2

- Input file

name,age

Andy,30

Michael,15

Justin,19

Andy,40

- Expected output

name,avg(age),count(name)

Andy,35,2

Michael,15,1

Justin,19,1

# groupBy and aggregates functions:

## Example 2

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

# groupBy and aggregates functions:

## Example 2

```
// Group data by name
```

```
RelationalGroupedDataset rgd=ds.groupBy("name");
```

```
// Compute average and number of rows for each group
```

```
// We use the aggr method to return two columns (one for each  
aggregate function)
```

```
Dataset<Row> nameAvgAgeCount = rgd.agg(avg("age"),count("name"));
```



# Sort

- The `Dataset<T> sort(String col1, .., String coln)` method of the `Dataset<T>` class returns a new Dataset that
  - contains the same data of the input one
  - but the content is sorted by `col1, .., coln` in **ascending order**
- Pay attention that the sort method **can generate errors at runtime** if there are mistakes in the names of the used columns

# Sort: Descending order

- The content of the Dataset can be sorted also in descending order by using
  - `Dataset<T> sort(Column col1, .., Column coln)`
  - And the `desc()` method of the `org.apache.spark.sql.Column` class
- If data are sorted by considering a set of columns, we can specify for each Column is the ascending or descending order must be used

# Sort: Example

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - One line per person
    - Each line contains name, age, and gender of a person
- Create a new Dataset containing the content of the input Dataset sorted by descending age
  - If the age value is the same, sort data by ascending name

# Sort: Example

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();

// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);

// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);

// Create a new Dataset with data sorted by desc. age, asc. name
Dataset<Person> sortedAgeName =
    ds.sort(new Column("age").desc(), new Column("name"));
```

# Dataset, DataFrames and the SQL language

# Datasets, DataFrames and the SQL language

- Sparks allows querying the content of a Dataset also by using the SQL language
  - In order to do this a “table name” must be assigned to each Dataset
- The **void createOrReplaceTempView (String tableName)** method of the **Dataset<Row>** class can be used to assign a “table name” to the Dataset on which it is invoked

# DataFrames and the SQL language

- Once the Datasets have been mapped to “table names”, SQL-like queries can be executed
  - The executed queries return DataFrame objects
- The **Dataset<Row> sql(String sqlQueryText)** method of the **SparkSession** class can be used to execute an SQL-like query
  - sqlQueryText is an SQL-like query
- Currently some SQL features are not supported

# DataFrames and the SQL language: Example 1

- Create a Dataset from a CSV file containing the profiles of a set of persons
  - Each line of the file contains containing name, age, and gender of a person
- Create a new DataFrame containing only the persons with age between 20 and 31 and print them on the standard output
  - Use the SQL language to perform this operation



# DataFrames and the SQL language:

## Example 1

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

# DataFrames and the SQL language:

## Example 1

```
// Assign the "table name" people to the ds Dataset
ds.createOrReplaceTempView("people");

// Select the persons with age between 20 and 31
// by querying the people table
Dataset<Row> selectedPersons =
    ss.sql("SELECT * FROM people WHERE age>=20 and age<=31");

// Print the result on the standard output
selectedPersons.show();
```

# DataFrames and the SQL language: Example 2

- Create two Datasets
  - One based on the persons\_id.csv file that contains the profiles of a set of persons
    - Schema: uid,name,age
  - One based on the liked\_sports.csv file that contains the liked sports for each person
    - Schema: uid,sportname
- Join the content of the two Datasets and show it on the standard output

# DataFrames and the SQL language:

## Example 2

```
public class PersonID implements Serializable {  
    private String name; private int age; private int uid;  
  
    public String getName() {  
        return name;    }  
    public void setName(String name) { this.name = name; }  
  
    public int getAge() { return age; }  
  
    public void setAge(int age) { this.age = age; }  
  
    public int getUid() { return uid; }  
  
    public void setUid(int uid) { this.uid = uid; }  
}
```

# DataFrames and the SQL language:

## Example 2

```
public class UIDSport implements Serializable {  
    private int uid; private String sportname;  
  
    public String getSportname() { return sportname; }  
  
    public void setSportname(String sportname) {    this.sportname =  
        sportname; }  
  
    public int getUid() { return uid; }  
  
    public void setUid(int uid) { this.uid = uid; }  
}
```

# DataFrames and the SQL language:

## Example 2

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();

// Define the encoder that is used to serialize PersonID objects
Encoder<PersonID> personIDEncoder = Encoders.bean(PersonID.class);

// Read persons_id.csv and store it in a Dataset<PersonID>
Dataset<PersonID> dsPersons = ss.read().format("csv")
    .option("header", true).option("inferSchema", true)
    .load("persons_id.csv").as(personIDEncoder);

// Assign the "table name" people to the dfPersons Dataset
dsPersons.createOrReplaceTempView("people");
```

# DataFrames and the SQL language:

## Example 2

```
// Define the encoder that is used to serialize UIDSport objects
Encoder<UIDSport> uidSportEncoder = Encoders.bean(UIDSport.class);

// Read liked_sports.csv and store it in a Dataset<UIDSport>
Dataset<UIDSport> dsUidSports = ss.read().format("csv")
    .option("header", true).option("inferSchema", true)
    .load("liked_sports.csv").as(uidSportEncoder);

// Assign the "table name" liked to the dfUidSports Dataset
dsUidSports.createOrReplaceTempView("liked");
```

# DataFrames and the SQL language: Example 2

```
// Join the two input tables by using the SQL-like syntax
Dataset<Row> dfPersonLikes =
    ss.sql("SELECT * from people, liked where people.uid=liked.uid");

// Print the result on the standard output
dfPersonLikes.show();
```



# DataFrames and the SQL language:

## Example 3

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a DataFrame containing for each name the average value of age and the number of person with that name
  - Print its content on the standard output

# DataFrames and the SQL language: Example 3

- Input file

name,age

Andy,30

Michael,15

Justin,19

Andy,40

- Expected output

name,avg(age),count(name)

Andy,35,2

Michael,15,1

Justin,19,1

# DataFrames and the SQL language:

## Example 3

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

```
// Assign the "table name" people to the df Dataset
ds.createOrReplaceTempView("people");
```

# DataFrames and the SQL language:

## Example 3

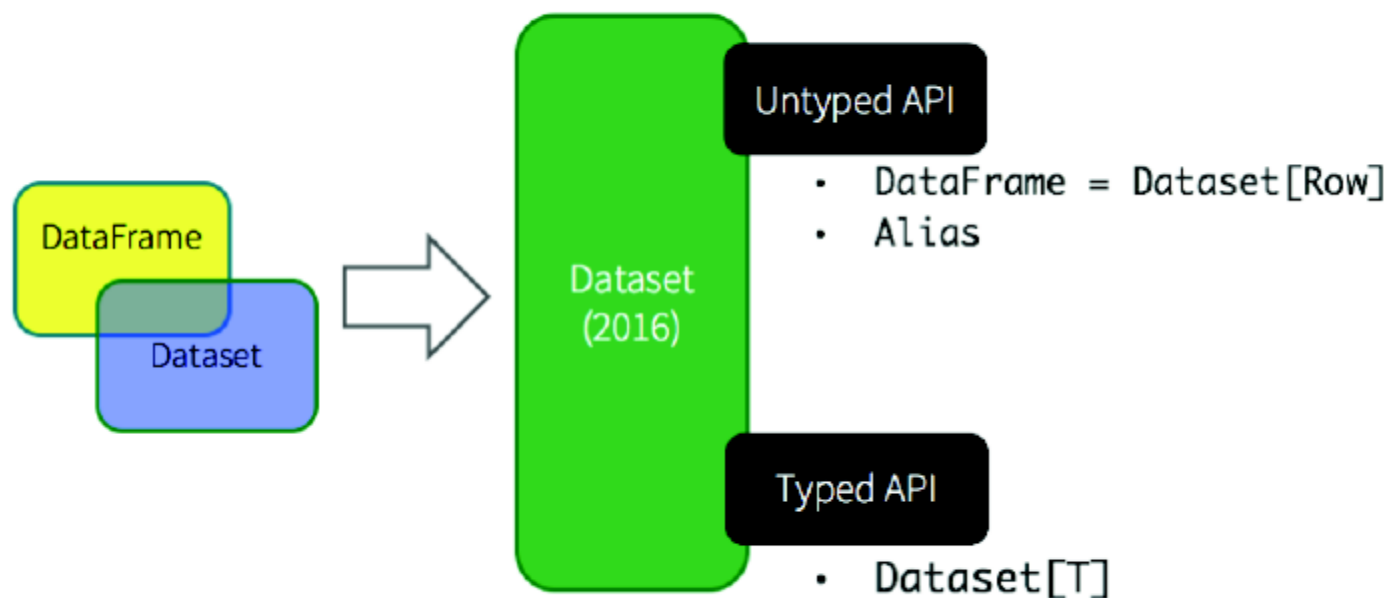
```
// Define groups based on the value of name and compute average and
// number of rows for each group
Dataset<Row> nameAvgAgeCount =
    ss.sql("SELECT name, avg(age), count(name) FROM people
           GROUP BY name");

// Print the result on the standard output
nameAvgAgeCount.show();
```

# Dataset vs DataFrames vs SQL

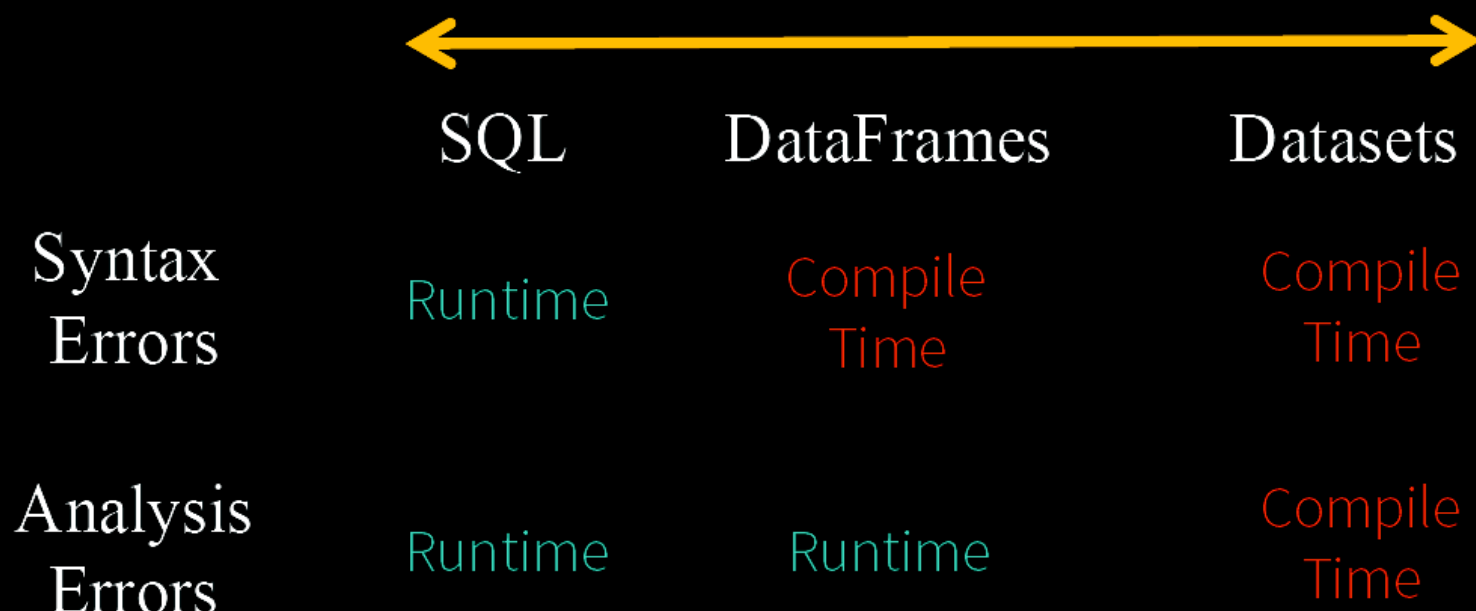
# Dataset vs DataFrames vs SQL

## Unified Apache Spark 2.0 API



# Dataset vs DataFrames vs SQL

## Structured APIs In Spark



Analysis errors are reported before a distributed job starts

# Dataset vs DataFrames vs SQL

- With the SQL-like approach the errors can be related also to a wrong syntax of the SQL query
  - With Datasets and DataFrames we do not have this problem
- With the SQL-like approach and DataFrames we can have a runtime error related to the wrong name of an attribute
  - With `Datasets<Type>` (with `T<>Row`) we do not have this problem if we use type-safe methods such as `map()` and `filter()`



# Save Datasets and DataFrames

# Save Datasets and DataFrames

- The content of Datasets (and DataFrames) can be stored on disk by using two approaches
  - 1 Convert Datasets (and DataFrames) to traditional RDDs by using the `JavaRDD<T> javaRDD()` method of the `Dataset<T>`
    - It returns a JavaRDD containing the content of the Dataset on which it is invokedAnd then use `saveAsTextFile(String outputFolder)`
  - 2 Use the `DataFrameWriter<Row> write()` method of Datasets combined with `format(String filetype)` and `void save(String outputFolder)` method

# Save datasets: Example 1

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Store the Dataset in the output folder by using the `saveAsTextFile(..)` method

# Save datasets: Example 1

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

```
// Save the file on the disk
ds.JavaRDD().saveAsTextFile(outputPath);
```

# Save datasets: Example 2

- Create a Dataset from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Store the Dataset in the output folder by using the write() method
  - Store the result by using the CSV format

# Save datasets: Example 2

```
// Create a Spark Session object and set the name of the application
SparkSession ss = SparkSession.builder().appName("Test
SparkSQL").getOrCreate();
```

```
// Define the encoder that is used to serialize Person objects
Encoder<Person> personEncoder = Encoders.bean(Person.class);
```

```
// Read the content of the input file and store it in a Dataset<Person>
// dataset
Dataset<Person> ds = ss.read().format("csv").option("header", true)
.option("inferSchema", true).load("persons.csv").as(personEncoder);
```

```
// Save the file on the disk by using the CSV format
ds.write().format("csv").option("header", true).save(outputPath);
```

# Spark SQL: User Defined Functions

# UDFs: User Defined Functions

- Spark SQL provides a set of system predefined functions
  - `hour(Timestamp)`, `abs(Integer)`, ..
  - Those functions can be used in some transformations (e.g., `selectExpr(..)`, `sort(..)`) but also in the SQL queries
- Users can defined their personalized functions
  - They are called User Defined Functions (UDFs)



# UDFs: User Defined Functions

- UDFs are defined/registered by invoking the `udf().register(String name, UDF function, DataType datatype)` on the `JavaSparkSession`
  - name: name of the defined UDF
  - function: lambda function/class used to specify how the parameters of the function are used to generate the returned value
    - One of more input parameters
    - One single returned value
  - datatype: SQL data type of the returned value

# UDFs: User Defined Functions – Example

- Define a UDFs that, given a string, returns the length of the string

```
// Create a Spark Session
```

```
SparkSession ss = SparkSession.builder().appName("Spark  
Example").getOrCreate();
```

```
// Define the UDF
```

```
// name: length
```

```
// input: String
```

```
// output: Integer
```

```
ss.udf().register("length", (String name) -> name.length(),  
                          DataTypes.IntegerType);
```

# UDFs: User Defined Functions – Example

- Use of the defined UDF in a selectExpr transformation

```
// Create a Spark Session  
Dataset<Row> result=  
    inputDF.selectExpr("length(name) as size");
```

- Use of the defined UDF in a SQL query

```
// Create a Spark Session  
Dataset<Row> result=  
    ss.sql("SELECT length(name) FROM profiles");
```

# UDAFs: User Defined Aggregate Functions

- Sparks allows defining personalized aggregate function
  - They are used to aggregate the values of a set of tuples
- They are based on the implementation of the `org.apache.spark.sql.expressions.UserDefinedAggregateFunction` abstract class

# UDAFs: User Defined Aggregate Functions

- The definition of the class associated with an aggregate function is associated with many variables and methods
  - Definition of input, intermediate, and returned schemas
  - Definition of the update and merge procedures
    - Update the internal status value by combining it with a new input record
    - Merge the local status results of two partitions
    - Convert the internal status into the final returned result