Spark SQL and DataFrames

Spark SQL

- Spark SQL is the Spark component for structured data processing
- It provides a programming abstraction called *Dataframe* and can act as a distributed SQL query engine
  - The input data can be queried by using
    1. Ad-hoc methods
    2. 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

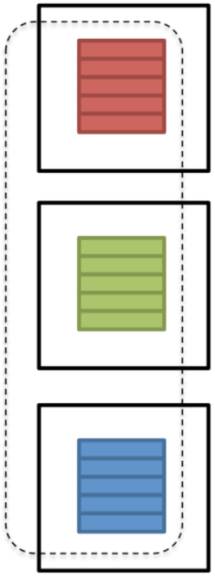
  => Programs based on **Dataframe** are usually **faster than standard RDD**-based programs

Spark SQL vs Spark RDD APIs

RDD
Unstructured

vs

DataFrame
Structured



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

Distributed list of objects

~Distributed SQL table

DataFrames

- DataFrame
  - Distributed collection of structured data
    - 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, ..)
  - Benefits from Spark SQL's optimized execution engine exploiting the information about the data structure

Spark Session

- All the Spark SQL functionalities are based on an instance of the

  `pyspark.sql.`**`SparkSession`** class

  - Import it in your standalone applications

    ```
    from pyspark.sql import SparkSession
    ```

- To instance a SparkSession object:

  ```
  spark = SparkSession.builder.getOrCreate()
  ```

Spark Session

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

```
spark.stop()
```

DataFrames

DataFrames

- DataFrame
  - It is a distributed collection of data organized into named columns
  - It is equivalent to a relational table
- **DataFrames** are lists of Row objects
- Classes used to define DataFrames
  - **pyspark.sql.DataFrame**
  - **pyspark.sql.Row**

DataFrames

- DataFrames can be created 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 DataFrames 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 in unknown

Creating DataFrames from csv files

- The creation of a DataFrame from a csv file is based the
  - **load(path)** method of the **pyspark.sql.DataFrameReader** class
    - Path is the path of the input file
  - You get a **DataFrameReader** with the **read()** method of the **SparkSesssion** class.

```
df = spark.read.load(path, options…)
```

Creating DataFrames from csv files: Example

- Create a DataFrame from a csv file (persons.csv) 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 (i.e., it can be missing)
  - The first line contains the header (i.e., the names of the attributes/columns)
  - Example of csv file

```
name,age
Andy,30
Michael,
Justin,19
```

Creating DataFrames from csv files: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format="csv",
                     header=True,
                     inferSchema=True)
```

Creating DataFrames from csv files: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format="csv",
                     header=True,
                     inferSchema=True)
```

This is used to specify the format of the input file

Creating DataFrames from csv files: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format="csv",
                     header=True,
                     inferSchema=True)
```

This 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
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format="csv",
                     header=True,
                     inferSchema=True)
```

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 complaint 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 in 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
  - **load(path)** method of the **pyspark.sql.DataFrameReader** class
    - Path is the path of the input file
  - You get a **DataFrameReader** with the **read()** method of the **SparkSesssion** class

```
df = spark.read.load(path, format="json", …)
```

    - or

```
df = spark.read.json(path, …)
```

Creating DataFrames from JSON files

- The same API allows also reading "standard" multiline JSON files
  - Set the multiline option to true by setting the argument **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
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(  "persons.json",
                       format="json")
```

Creating DataFrames from JSON files: Example 1

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(  "persons.json",
                       format="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
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(   "folder_JSONFiles/",
                        format="json",
                        multiLine=True)
```

Creating DataFrames from JSON files: Example 2

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(   "folder_JSONFiles/",
                        format="json",
                        multiLine=True)
```

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.

Creating DataFrames from RDDs or Python lists

- The content of an RDD of tuples or the content of a Python list of tuples can be stored in a DataFrame by using the **spark.createDataFrame(data, schema)** method
  - **data**: RDD of tuples or Rows, Python list of tuples or Rows, or pandas DataFrame
  - **schema**: list of string with the names of the columns/attributes
    - **schema** is optional. If it is not specified the column names are set to _1, _2, …, _n for input RDDs/lists of tuples

Creating DataFrames from RDDs or Python lists: Example

- Create a DataFrame from a Python list containing the following data
  - (19, "Justin")
  - (30, "Andy")
  - (None, "Michael")
- The column names must be set to “age” and “name”

Creating DataFrames from RDDs or Python lists: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a Python list of tuples
profilesList = [(19, "Justin"), (30, "Andy"),
                (None, "Michael")]

# Create a DataFrame from the profilesList
df = spark.createDataFrame(profilesList,["age","name"])
```

From DataFrame to RDD

- The **rdd** member of the **DataFrame** class returns an RDD of Row objects containing the content of the DataFrame on which it is invoked
- Each **Row** object is like a dictionary containing the values of a record
  - It contains column names in the keys and column values in the values

From DataFrame to RDD

Usage of the **Row** class

- The fields in it can be accessed:
  - like attributes `(row.key)`
    - where `key` is a column name
  - like dictionary values `(row["key"])`
  - `for key in row` will search through row keys
- asDict() method:
  - Returns the Row content as a Python dictionary

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 an RDD, select only the name field/column and store the result in the output folder

From DataFrame to RDD: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(   "persons.csv",
                        format="csv",
                        header=True,
                        inferSchema=True)
```

From DataFrame to RDD: Example

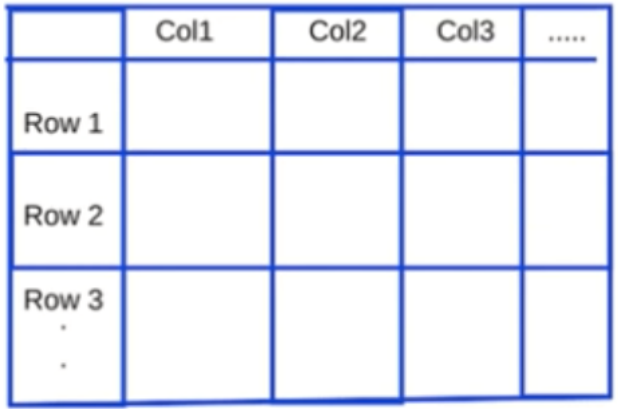
```
# Define an RDD based on the content of
# the DataFrame
rddRows = df.rdd

# Use the map transformation to extract
# the name field/column
rddNames = rddRows.map(lambda row: row.name)

# Store the result
rddNames.saveAsTextFile(outputPath)
```

Operations on DataFrames

DataFrame operations

- Now you know how to create DataFrames

- How to manipulate them?
- How to compute statistics?



DataFrame operations

- A set of specific methods are available for the DataFrame class
  - E.g., show(), printSchema(), count(), distinct(), select(), filter()
- Also the standard collect() and count() actions are available

Show

- The **show(n)** method of the **DataFrame** class prints on the standard output the first **n** of the input DataFrame
  - Default value of **n**: 20

Show: Example

- Create a DataFrame 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 DataFrame)

Show: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(   "persons.csv",
                        format="csv",
                        header=True,
                        inferSchema=True)

df.show(2)
```

PrintSchema

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

Count

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

Distinct

- The **distinct()** method of the **DataFrame** class returns a new DataFrame that contains only the unique rows of the input DataFrame
  - 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 DataFrame from a csv file containing the names of a set of persons
  - The content of names.csv is

    name

    Andy

    Michael

    Justin

    Michael
  - The first line is the header
- Create a new DataFrame without duplicates

Distinct: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from names.csv
df = spark.read.load(   "names.csv",
                        format="csv",
                        header=True,
                        inferSchema=True)

df_distinct = df.distinct()
```

Select

- The **select(col1, .., coln)** method of the **DataFrame** class returns a new DataFrame that contains only the specified columns of the input DataFrame
- Use **'*'** as special column to select all columns
- 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 DataFrame from the persons2.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
  - Example

    name,age,gender
    Paul,40,male
    John,40,male
    ..
- Create a new DataFrame containing only name and age of the persons

Select: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons2.csv
df = spark.read.load(  "persons2.csv",
                       format="csv",
                       header=True,
                       inferSchema=True)

dfNamesAges = df.select("name", "age")
```

SelectExpr

- The **selectExpr(expression1, .., expressionN)** method of the **DataFrame** class is a variant of the select method, where expr can be an SQL expression.
- Example:

```
df.selectExpr("name", "age")
df.selectExpr("name", "age + 1 AS new_age")
```

SelectExpr: Example

- Create a DataFrame from the persons2.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 plus an additional column called newAge containing the value of age incremented by one

SelectExpr: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load("persons2.csv",
                     format="csv",
                     header=True,
                     inferSchema=True)

# Create a new DataFrame with four columns:
# name, age, gender, newAge = age +1
dfNewAge = df.selectExpr("name", "age", "gender",
                         "age+1 as newAge")
```

SelectExpr: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load("persons2.csv",
                     format="csv",
                     header=True,
```

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 DataFrame. Without this part of the expression, the name of the returned column will be "age+1"

```
dfNew = df.selectExpr("name", "age", "gender",
                      "age+1 as newAge")
```

Filter

- The **filter(conditionExpr)** method of the **DataFrame** class returns a new DataFrame that contains only the rows satisfying the specified condition
  - The condition is a Boolean **SQL** expression
  - 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 DataFrame 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 and age of a person
- Create a new DataFrame containing only the persons with age between 20 and 31

Filter: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(   "persons.csv",
                        format="csv",
                        header=True,
                        inferSchema=True)

df_filtered = df.filter("age>=20 and age<=31")
```

Where

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

Join

- The **join(right, on, how)** method of the **DataFrame** class is used to join two DataFrames
  - It returns a DataFrame that contains the join of the tuples of the two input DataFrames based on the **on** join condition

Join

- **on**: the join condition
  - It can be:
    - A string: the join column
    - A list of strings: multiple join columns
    - A condition/an expression on the columns.
      - E.g.:
        - `joined_df = df.join(df2, df.name == df2.name)`
- **how**: the type of join
  - inner, cross, outer, full, full_outer, left, left_outer, right, right_outer, left_semi, and left_anti
  - Default: inner

Join

- **Pay attention** that this method
  - **Can generate errors at runtime** if there are errors in the join expression

Join: Example

- Create two DataFrames
  - 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 DataFrames (uid is the join column) and show it on the standard output

Join: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.csv",
                            format="csv",
                            header=True,
                            inferSchema=True)

# Read liked_sports.csv and store it in a DataFrame
dfUidSports = spark.read.load("liked_sports.csv",
                              format="csv",
                              header=True,
                              inferSchema=True)

# Join the two input DataFrames
dfPersonLikes = dfPersons.join(dfUidSports,
                               dfPersons.uid == dfUidSports.uid)

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

Join: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.csv",
                            format="csv",
                            header=True,
                            inferSchema=True)

# Read liked_sports.csv and store it in a DataFrame
dfUidSports = spark.read.load("liked_sports.csv",
                              format="csv",
                              header=True,

# Join the two input DataFrames
dfPersonLikes = dfPersons.join(dfUidSports,
                               dfPersons.uid == dfUidSports.uid)

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

Specify the join condition on the uid columns

Join: Example #2

- Create two DataFrames
  - 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

Join: Example #2

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.csv",
                            format="csv",
                            header=True,
                            inferSchema=True)

# Read banned.csv and store it in a DataFrame
dfBannedUsers = spark.read.load("banned.csv",
                              format="csv",
                              header=True,
                              inferSchema=True)

# Apply the Left Anti Join on the two input DataFrames
dfSelectedProfiles = dfPersons.join(dfBannedUsers,
                       dfPersons.uid == dfBannedUsers.uid,
                       "left_anti“)

# Print the result on the standard output
dfSelectedProfiles.show()
```

Join: Example #2

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.csv",
                            format="csv",
                            header=True,
                            inferSchema=True)

# Read banned.csv and store it in a DataFrame
dfBannedUsers = spark.read.load("banned.csv",
                                format="csv",
                                header=True,
                                inferSchema=True)

# Apply the Left
dfSelectedProfiles = dfPersons.join(dfBannedUsers,
                                    dfPersons.uid == dfBannedUsers.uid,
                                    "left_anti")

# Print the result on the standard output
dfSelectedProfiles.show()
```

Specify the (anti) join condition on the uid columns

Join: Example #2

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.csv",
                            format="csv",
                            header=True,
                            inferSchema=True)

# Read banned.csv and store it in a DataFrame
dfBannedUsers = spark.read.load("banned.csv",
                                format="csv",
                                header=True,
                                inferSchema=True)

# Apply the Left Anti j                    input DataFrames
dfSelectedProfiles = di                    fBannedUsers,
                       dfPersons.uid == dfBannedUsers.uid,
                       "left_anti")

# Print the result on the standard output
dfSelectedProfiles.show()
```

Use 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.
  - Each aggregate function returns one value computed by considering all the values of the input column

Aggregates functions

- The **agg(expr)** method of the DataFrame class is used to specify which aggregate function we want to apply on one input column
  - The result is a DataFrame containing one single row and one single column
  - The name of the return column is "function_name(column)"
- Pay attention that this methods **can generate errors at runtime**
  - E.g., wrong attribute name, wrong data type

Aggregates functions: Example

- Create a DataFrame 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 and age of a person
- Create a Dataset containing the average value of age

Aggregates functions: Example

- Input file

  name,age

  Andy,30

  Michael,15

  Justin,19

  Andy,40

- Expected output

  avg(age)

  26.0

Aggregates functions: Example

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format="csv",
                     header=True,
                     inferSchema=True)

# Compute the average of age
averageAge = df.agg({"age": "avg"})
```

groupBy and aggregates functions

- The method **groupBy(col1, .., coln)** method of the **DataFrame** class combined with a set of aggregate methods 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 **groupBy(col1, .., coln)** method
- Then, apply the aggregate functions you want to compute by final result
  - The result is 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 **pyspark.sql.GroupedData** class for a complete list

groupBy and aggregates functions: Example 1

- Create a DataFrame 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 and age of a person
- Create a DataFrame containing the 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
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(   "persons.csv",
                        format="csv",
                        header=True,
                        inferSchema=True)

grouped = df.groupBy("name").avg("age")
```

groupBy and aggregates functions: Example 2

- Create a DataFrame 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 and age of a person
- Create a DataFrame containing the 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
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(  "persons.csv",
                       format="csv",
                       header=True,
                       inferSchema=True)

grouped = df.groupBy("name")
           .agg({"age": "avg", "name": "count"})
```

Sort

- The **sort(col1, .., coln, ascending=True)** method of the **DataFrame** class returns a new DataFrame that
  - contains the same data of the input one
  - but the content is sorted by **col1, .., coln**
  - **Ascending** determines ascending vs. descending

DataFrames and the SQL language

DataFrames and the SQL language

- Sparks allows querying the content of a DataFrame also by using the SQL language
  - In order to do this a **"table name"** must be assigned to a DataFrame
- The **createOrReplaceTempView (tableName)** method of the **DataFrame** class can be used to assign a "table name" to the DataFrame on which it is invoked

DataFrames and the SQL language

- Once the DataFrame has been mapped to "table names", SQL-like queries can be executed
  - The executed queries return DataFrame objects
- The **sql(query)** method of the **SparkSession** class can be used to execute an SQL-like query
  - **query** is an SQL-like query
- Currently some SQL features are not supported
  - E.g., nested subqueries in the WHERE clause are not allowed

DataFrames and the SQL language: Example 1

- Create a DataFrame from a JSON file containing the profiles of a set of persons
  - Each line of the file contains a JSON object 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
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(    "persons.json",
                         format="json")

# Assign the “table name” people to the df DataFrame
df.createOrReplaceTempView("people");

# Select the persons with age between 20 and 31
# by querying the people table
selectedPersons =
spark.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 DataFrames
  - 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 DataFrames and show it on the standard output

DataFrames and the SQL language: Example 2

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.csv",
                            format="csv",
                            header=True,
                            inferSchema=True)

# Assign the “table name” people to the dfPerson
dfPersons.createOrReplaceTempView("people")

# Read liked_sports.csv and store it in a DataFrame
dfUidSports = spark.read.load("liked_sports.csv",
                              format="csv",
                              header=True,
                              inferSchema=True)

# Assign the “table name” liked to dfUidSports
dfUidSports.createOrReplaceTempView("liked")
```

DataFrames and the SQL language: Example 2

```
# Join the two input tables by using the
#SQL-like syntax
dfPersonLikes = spark.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 DataFrame 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 and age 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
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(    "persons.json",
                         format="json")

# Assign the “table name” people to the df DataFrame
df.createOrReplaceTempView("people")

# Define groups based on the value of name and
# compute average and number of records for each group
nameAvgAgeCount = spark.sql("SELECT name, avg(age),
            count(name) FROM people GROUP BY name")

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

Save DataFrames

Save DataFrames

- The content of DataFrames can be stored on disk by using two approches
  1. Convert DataFrames to traditional RDDs by using the **rdd** method of the **DataFrame**

     And then use saveAsTextFile(outputFolder)
  2. Use the **write()** method of DataFrames, that returns a **DatFrameWriter** class instance

Save DataFrames: Example 1

- Create a DataFrame 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 DataFrame in the output folder by using the saveAsTextFile(..) method

Save DataFrames: Example 1

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(   "persons.csv",
                        format="csv",
                        header=True,
                        inferSchema=True)

# Save it
df.rdd.saveAsTextFile(outputPath)
```

Save DataFrames: Example 2

- Create a DataFrame 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 DataFrame in the output folder by using the write() method
  - Store the result by using the CSV format

Save DataFrames: Example 2

```
# Create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame from persons.csv
df = spark.read.load(   "persons.csv",
                        format="csv",
                        header=True,
                        inferSchema=True)

# Save it
df.write.csv(outputPath, header=True)
```

UDFs: 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(name, function, datatype) on the SparkSession
  - name: name of the defined UDF
  - function: lambda function 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 object
spark = SparkSession.builder.getOrCreate()

# Define the UDF
# name: length
# output: integer value
spark.udf.register("length", lambda x: len(x))
```

UDFs: User Defined Functions – Example

- Use of the defined UDF in a selectExpr transformation

```
result = inputDF.selectExpr("length(name) as size")
```

- Use of the defined UDF in a SQL query

```
result = spark.sql("SELECT length(name) FROM
                    profiles")
```