SP04_l1_23-DataFrames-API-SQL

November 3, 2023

1 Big Data - Course - DataFrames

1.1 DataFrames - wczytywanie danych

DataFrame można tworzyć - z istniejących RDD - wczytując z plików źródłowych (CSV, JSON, Avro, Parquet, ...) - łącząc się do zewnętrznych baz danych (Hive, Cassandra, JDBC, ...)

[1]:
```
from pyspark.sql import SparkSession

# Spark session & context
spark = SparkSession.builder.master("local").enableHiveSupport().getOrCreate()
```

[2]:
```
import os
def remove_file(file):
    if os.path.exists(file):
        os.remove(file)

remove_file("mondial.cities.json")
remove_file("mondial.countries.json")
```

[3]:
```
import requests
r = requests.get("https://jankiewicz.pl/bigdata/bigdata-sp/mondial.cities.
 ↪json", allow_redirects=True)
open('mondial.cities.json', 'wb').write(r.content)
r = requests.get("https://jankiewicz.pl/bigdata/bigdata-sp/mondial.countries.
 ↪json", allow_redirects=True)
open('mondial.countries.json', 'wb').write(r.content)
```

[3]:
```
130712
```

[4]:
```
df_cities = spark.read.json("mondial.cities.json")
df_countries = spark.read.json("mondial.countries.json")
```

1.2 DataFrame – schemat

Mając obiekt `pyspark.sql.DataFrame` możemy dowiedzieć się jaka jest budowa jego rekordów za pomocą metody `printSchema()` Atrybuty odnoszące się także do schematu to: - `schema` - atrybut, który daje w wyniku obiekt `pyspark.sql.types.StructType` reprezentujący strukturę `DataFrame`,

- `dtypes` - atrybut, który daje w wyniku listę krotek (nazwa pola, typ pola) - `columns` - atrybut, który daje w wyniku listę nazw kolumn

[5]:
```
df_cities.printSchema()
df_countries.printSchema()
```

```
root
 |-- _id: string (nullable = true)
 |-- country: string (nullable = true)
 |-- elevation: double (nullable = true)
 |-- latitude: double (nullable = true)
 |-- location: struct (nullable = true)
 |    |-- coordinates: array (nullable = true)
 |    |    |-- element: double (containsNull = true)
 |    |-- type: string (nullable = true)
 |-- longitude: double (nullable = true)
 |-- name: string (nullable = true)
 |-- other_names: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- population: long (nullable = true)
 |-- province: string (nullable = true)

root
 |-- _id: struct (nullable = true)
 |    |-- $oid: string (nullable = true)
 |-- area: double (nullable = true)
 |-- capital: string (nullable = true)
 |-- code: string (nullable = true)
 |-- gdp: double (nullable = true)
 |-- government: string (nullable = true)
 |-- independence: struct (nullable = true)
 |    |-- $date: string (nullable = true)
 |-- inflation: double (nullable = true)
 |-- name: string (nullable = true)
 |-- population: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- value: long (nullable = true)
 |    |    |-- year: long (nullable = true)
 |-- unemployment: double (nullable = true)
```

[6]:
```
type(df_cities.dtypes)
```

[6]:
```
list
```

[7]:
```
type(df_cities.dtypes[0])
```

[7]:
```
tuple
```

[8]:
```
df_cities.dtypes[0]
```

[8]:
```
('_id', 'string')
```

1.3 DataFrame – przetwarzanie danych

Obiekty DataFrame możemy przetwarzać na wiele sposobów:

- korzystając z metod DataFrame
- korzystając z funkcji (z reguły w ramach metod do przekształcania kolumn)
- korzystając z SQL

DataFrame API

projekcja: - `select(*cols: ColumnOrName) → DataFrame` - `cols`: str, Column, lub list - `selectExpr(*expr: Union[str, List[str]]) → DataFrame` - `expr`: do specyfikowania kolumn używamy ciągów znaków zawierających wyrażenia SQL

polaczenia: - `join(right: DataFrame, on: Union[str, List[str], Column, List[Column], None] = None, how: Optional[str] = None) → DataFrame`
- `right` - prawa strona połączenia - `on` - warunek połączeniowy oparty na kolumnach (warunek równościowy) lub wyrażeniu (warunku) połączeniowym - `how` - typ połączenia (domyślnie `inner`) - `crossJoin(right: DataFrame) → DataFrame`

typy połączeń: - `inner`, `cross`, `outer` - `full`, `fullouter`, `full_outer`, - `left`, `leftouter`, `left_outer`, - `right`, `rightouter`, `right_outer`, - `semi`, `leftsemi`, `left_semi` - `anti`, `leftanti`, `left_anti`

projekcja negatywna: - `drop(*cols: ColumnOrName) → DataFrame` - `cols`: str, Column, lub list

1.4 DataFrame – przetwarzanie danych (odwoływanie się do kolumn)

Może przymowac nastepujaca forme:

[9]:
```
# a) ciągu znaków (np. col: String)
df_cities.select("name").show()
df_cities.select("name", "population").show()
```

```
+------------+
|        name|
+------------+
|      Kavala|
|    Komotini|
|      Athina|
|    Peiraias|
|   Peristeri|
|    Acharnes|
|       Patra|
|      Kozani|
|    Ioannina|
|     Kerkyra|
```

```
|Thessaloniki|
|      Chania|
|      Durrës|
|   Ermoupoli|
|      Rhodes|
|     Tripoli|
|     Iraklio|
|       Lamia|
|    Chalkida|
|     Larissa|
+------------+
only showing top 20 rows

+------------+----------+
|        name|population|
+------------+----------+
|      Kavala|     58790|
|    Komotini|      null|
|      Athina|    664046|
|    Peiraias|    163688|
|   Peristeri|    139981|
|    Acharnes|    106943|
|       Patra|    213984|
|      Kozani|      null|
|    Ioannina|    112486|
|     Kerkyra|      null|
|Thessaloniki|    325182|
|      Chania|    108642|
|      Durrës|    113249|
|   Ermoupoli|      null|
|      Rhodes|    115490|
|     Tripoli|      null|
|     Iraklio|    173993|
|       Lamia|     75315|
|    Chalkida|    102223|
|     Larissa|    162591|
+------------+----------+
only showing top 20 rows
```

[10]:
```
# b) odwoływania się do kolumn (np. cols: Column*) i wyrażeń z użyciem (dzięki
 ↪metodom typu Column)
# - za pomocą notacji objDF["colname"] - konkretna kolumna z określonego
 ↪obiektu DataFrame
df_cities.select(df_cities["name"], df_cities["elevation"] / 1000).show()
```

```
+------------+------------------+
|        name|(elevation / 1000)|
```

```
+------------+------------------+
|      Kavala|               0.0|
|    Komotini|             0.045|
|      Athina|              0.07|
|    Peiraias|               0.0|
|   Peristeri|              0.05|
|    Acharnes|             0.186|
|       Patra|               0.0|
|      Kozani|              0.71|
|    Ioannina|              0.48|
|     Kerkyra|               0.0|
|Thessaloniki|               0.0|
|      Chania|               0.0|
|      Durrës|               0.0|
|   Ermoupoli|               0.0|
|      Rhodes|             0.026|
|     Tripoli|             0.655|
|     Iraklio|              0.15|
|       Lamia|              0.05|
|    Chalkida|               0.0|
|     Larissa|             0.067|
+------------+------------------+
only showing top 20 rows
```

[11]:
```
# - za pomocą wyrażenia objDF.colname - konkretna kolumna z określonego obiektu␣
 ↪DataFrame
df_cities.join(df_countries, df_cities.country == df_countries.code). \
    select(df_cities.name,
         df_cities.population / 1000000,
         df_countries.name).show()
```

```
+------------+----------------------+-------+
|        name|(population / 1000000)|   name|
+------------+----------------------+-------+
|      Kavala|               0.05879| Greece|
|    Komotini|                  null| Greece|
|      Athina|              0.664046| Greece|
|    Peiraias|              0.163688| Greece|
|   Peristeri|              0.139981| Greece|
|    Acharnes|              0.106943| Greece|
|       Patra|              0.213984| Greece|
|      Kozani|                  null| Greece|
|    Ioannina|              0.112486| Greece|
|     Kerkyra|                  null| Greece|
|Thessaloniki|              0.325182| Greece|
|      Chania|              0.108642| Greece|
|      Durrës|              0.113249|Albania|
```

```
|   Ermoupoli|                  null| Greece|
|      Rhodes|               0.11549| Greece|
|     Tripoli|                  null| Greece|
|     Iraklio|              0.173993| Greece|
|       Lamia|              0.075315| Greece|
|    Chalkida|              0.102223| Greece|
|     Larissa|              0.162591| Greece|
+------------+----------------------+-------+
only showing top 20 rows
```

[12]:
```
# - za pomocą wyrażenia col("colname") - kolumna generyczna
from pyspark.sql.functions import col
df_countries.select(
  col("name"),
  col("population.value").getItem(0).alias("ludnosc") ).show()
```

```
+--------------+---------+
|          name|  ludnosc|
+--------------+---------+
|        France| 40502513|
|        Greece|  1096810|
|    Montenegro|   311341|
|         Spain| 18618086|
|       Germany| 68230796|
|       Austria|  4497873|
|Czech Republic|  8876260|
|       Albania|  1214489|
|       Hungary|  9204799|
|      Slovakia|  3436574|
| Liechtenstein|    13757|
|         Italy| 22182377|
|      Slovenia|  1101854|
|        Kosovo|  1584440|
|     Macedonia|   808724|
|        Serbia|  6732256|
|       Andorra|     6197|
|       Ukraine| 37297652|
|        Russia|102798657|
|       Belgium|  8879814|
+--------------+---------+
only showing top 20 rows
```

[13]:
```
# - za pomocą wyrażenia expr("exprWithCol")
from pyspark.sql.functions import expr
df_cities.select(expr("name"), expr("elevation/1000 as elev")).show()
```

```
+------------+-----+
|        name| elev|
+------------+-----+
|      Kavala|  0.0|
|    Komotini|0.045|
|      Athina| 0.07|
|    Peiraias|  0.0|
|   Peristeri| 0.05|
|    Acharnes|0.186|
|       Patra|  0.0|
|      Kozani| 0.71|
|    Ioannina| 0.48|
|     Kerkyra|  0.0|
|Thessaloniki|  0.0|
|      Chania|  0.0|
|      Durrës|  0.0|
|   Ermoupoli|  0.0|
|      Rhodes|0.026|
|     Tripoli|0.655|
|     Iraklio| 0.15|
|       Lamia| 0.05|
|    Chalkida|  0.0|
|     Larissa|0.067|
+------------+-----+
only showing top 20 rows
```

1.5 DataFrame – przetwarzanie danych

DataFrame API

- `alias(alias: str) → DataFrame`
- `where(condition) → DataFrame` // synonim: `filter`
  - `condition`: Column lub str - warunek selekcji
- `orderBy(*cols: Union[str, Column, List[Union[str, Column]]], **kwargs: Any) → DataFrame`
  - `cols`: str, list, lub Column - lista kolumn po których należy dane posortować
- `limit(num: int) → DataFrame`
- `intersect(other: DataFrame) → DataFrame`
- `intersectAll(other: DataFrame) → DataFrame`
- `union(other: DataFrame) → DataFrame`
- `unionAll(other: DataFrame) → DataFrame`
- `exceptAll(other: DataFrame) → DataFrame`

[14]:
```
# Jaki uzyskamy wynik?
df_cities.where(df_cities.elevation > 3000).orderBy("elevation").select("name").
  ↪show()
```

```
+--------------+
|          name|
+--------------+
|        Huaraz|
|      Huancayo|
|         Cusco|
|        La Paz|
|  Huancavelica|
|         Oruro|
|       Juliaca|
|          Puno|
|        Potosí|
|       El Alto|
|         Lhasa|
|Cerro de Pasco|
+--------------+
```

[15]:
```
# A tym razem?
big_cities = df_cities.where("population > 1000000")
high_cities = df_cities.where("elevation > 2000")
big_cities.intersect(high_cities).select("name","population").show()
```

```
+----------------+----------+
|            name|population|
+----------------+----------+
|     Addis Ababa|   3040740|
|          Puebla|   1434062|
|Ciudad de México|   8555272|
|           Quito|   1619146|
|          Sana'a|   1527861|
|  Nezahualcóyotl|   1104585|
|          Bogotá|   7776845|
|        Ecatepec|   1655015|
+----------------+----------+
```

1.6 DataFrame – przetwarzanie danych (grupowanie)

Do oddzielnej grupy można zaliczyć transformacje związane z grupowaniem

z reguły wymagają one pary transformacji:

- (1) transformacji grupujących, których wynik jest typu `pyspark.sql.group.GroupedData` patrz: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/grouping.html

  - `groupBy(*cols: ColumnOrName) → GroupedData`
    - `cols`: list, str lub Column
  - `rollup(*cols: ColumnOrName) → GroupedData`
    - `cols`: list, str lub Column
  - `cube(*cols: ColumnOrName) → GroupedData`
    - `cols`: list, str lub Column

- (2) transformacji wyznaczających agregaty
  - `agg(*exprs: Union[Column, Dict[str, str]]) → DataFrame`
    - `exprs` - agregacja za pomocą:
      - wbudowanych funkcji agregujących (`min`, `max`, `sum`, `count`, `avg`) `pyspark.sql.functions.*` lub
      - funkcji użytkownika utworzonych przy `pyspark.sql.functions.pandas_udf()`

[16]:
```
# I znowu - czym bedzie wynik?
from pyspark.sql.functions import *

df_cities.groupBy("country").agg(max("population").alias("maxpop")).show()
```

```
+-------+-------+
|country| maxpop|
+-------+-------+
|     LT| 538747|
|      K| 703963|
|     DZ|2364230|
|     RG|1667864|
|     CI|4395243|
|     AZ|2150800|
|     UA|2814258|
|    ARM|1066264|
|     RO|1883425|
|    JOR|1812941|
|   MNTS|      0|
|     NL| 812895|
|     BS| 248948|
|   FARX|  12245|
|   WEST| 163146|
|   REUN| 145347|
|     PL|1711324|
|   COOK|   null|
|     MK| 514967|
|    BRN| 143035|
+-------+-------+
only showing top 20 rows
```

[17]:
```
# A teraz?
df_cities.rollup(expr("round(population/1000000) as pop_mln")). \
         agg(count("_id"), max("elevation")). \
         orderBy(col("pop_mln").desc()).show()
```

```
+-------+----------+--------------+
|pop_mln|count(_id)|max(elevation)|
+-------+----------+--------------+
|   22.0|         1|           4.0|
|   14.0|         1|          40.0|
|   12.0|         3|         156.0|
|   11.0|         4|         760.0|
|   10.0|         4|          37.0|
|    9.0|         3|        2250.0|
|    8.0|         9|        2640.0|
|    7.0|         8|         505.0|
|    6.0|         7|         150.0|
|    5.0|        12|         938.0|
|    4.0|        21|        1892.0|
|    3.0|        51|        2355.0|
|    2.0|        88|        2850.0|
|    1.0|       471|        4150.0|
|    0.0|      2363|        4200.0|
|   null|       335|        4330.0|
|   null|      3381|        4330.0|
+-------+----------+--------------+
```

[18]:
```
# No i jeszcze może coś takiego?
df_cities.groupBy().agg(count("_id")).show()
```

```
+----------+
|count(_id)|
+----------+
|      3381|
+----------+
```

1.7 Akcje

Analogicznie jak w przypadku przetwarzania RDD, aby ostateczny wynik wrócił do programu sterownika, należy uwieńczyć serię transformacji akcją - `collect() → List[Row]` - `count() → int` - `show(n: int = 20, truncate: Union[bool, int] = True, vertical: bool = False) → None` - `n` - liczba wyświetlanych krotek - `truncate` - czy ucinać długie wartości w kolumnach - `vertical` - czy wyświetlać wiersze pionowo (kolumna pod kolumną) - `take(num: int) → List[Row]` - `printSchema(level: Optional[int] = None) → None[source]` - `level` - poziomy w zagnieżdżonych schematach - `explain(extended: Union[bool, str, None] = None, mode: Optional[str] = None) → None`

Zadanie

Znajdź kraje, w których suma ludności mieszkających w miastach leżących powyżej 1000 n.p.m. wynosi powyżej 10 mln. Dla każdego z nich podaj nazwę kraju oraz liczbę wspomnianych miast. Wyniki posortuj malejąco pod względem liczby miast.

Napiszmy wspólnie rozwiązanie:

[ ]:

Jaki byłby wynik gdyby show() zamienić na count()?

[ ]:

A gdyby usunąć 4 ostatnie linie i dodać count()?

[ ]:

2 pyspark.sql.group.GroupedData

Typ `pyspark.sql.group.GroupedData` posiada kilka metod będących transformacjami

- odpowiadającymi za wykonywanie funkcji agregujących:
  - `agg(*exprs: Union[Column, Dict[str, str]]) → DataFrame`
    - `exprs` - agregacja za pomocą:
      - wbudowanych funkcji agregujących (`min`, `max`, `sum`, `count`, `avg`) `pyspark.sql.functions.*` lub
      - funkcji użytkownika utworzonych przy `pyspark.sql.functions.pandas_udf()`
  - `avg(*cols: str) → DataFrame`
  - `count() → DataFrame`
  - `max(*cols: str) → DataFrame`
  - `mean(*cols: str) → DataFrame`
  - `min(*cols: str) → DataFrame`
  - `sum(*cols: str) → DataFrame`
- wykonujących operację pivot
  - `pivot(pivot_col: str, values: Optional[List[LiteralType]] = None) → GroupedData`

[19]:
```
df_cities.alias("ci").join(df_countries, df_countries.code == df_cities.
↪country). \
 where(df_cities.elevation > 1000). \
 groupBy(df_countries.name). \
 sum("ci.population").show()
```

```
+-------------+---------------+
|         name|sum(population)|
+-------------+---------------+
|        Yemen|        1986794|
|  Philippines|         301926|
```

```
|      Eritrea|         380568|
|       Turkey|        4469031|
|        Zaire|        1988378|
|       Malawi|        1335704|
|  Afghanistan|        2747200|
|       Rwanda|         603049|
|      Algeria|         634266|
|    Argentina|         693667|
|       Angola|           null|
|      Ecuador|        1951034|
|      Lesotho|         227880|
|   Madagascar|         945558|
|      Myanmar|         380665|
|         Peru|        2321711|
|        China|       16293745|
|        India|        1180570|
|United States|        3700249|
|   Tajikistan|           null|
+-------------+---------------+
only showing top 20 rows
```

[20]:
```
# Przyklad z pivot
# Na jakie pytanie odpowiada poniższe zapytanie?
df_countries. \
  groupBy(expr("round(area/1000000) as area")). \
  pivot("government", ["republic","parliamentary democracy"]). \
  count().where("`republic` is not null").show()
```

```
+----+--------+----------------------+
|area|republic|parliamentary democracy|
+----+--------+----------------------+
| 0.0|      46|                    22|
| 1.0|      17|                  null|
| 3.0|       1|                  null|
| 2.0|       3|                  null|
+----+--------+----------------------+
```

3 Funkcje

Oprócz znajomości różnorodnych metod do przetwarzania danych typów: `DataFrames` (https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/dataframe.html), `GroupedData` (https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/grouping.html), `Column` (https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/column.html), `Row` (z tylko jedną metodą `asDict`) (https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/row.h bardzo przydatna jest znajomość bardzo licznych funkcji zdefiniowanych w ramach pakietu

`pyspark.sql.functions` (https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/functions.html

W znaczącej większości są to funkcje działające na obiektach typu `Column` i dające w wyniku obiekty tego typu. Oznacza, że są one używane przez większość omówionych wcześniej metod.

Funkcje te są klasyfikowane następująco:

- Agregujące (`avg`, `count`, `max`, `min`, `sum`, `mean`, `variance`, `collect_list`, `collect_set`, `covar_pop`, `first`, `last`, `skewness`, `countDistinct`, `sumDistinct`, `kurtosis`, …)
- Działające na kolekcjach (`explode`, `from_json`, `json_tuple`, `sort_array`, …)
- Daty i czasu (`add_months`, `current_date`, `current_time`, `datediff`, `last_day`, `months_between`, `next_day`, `year`, `month`, `to_date`, …)
- Matematyczne (`cos`, `acos`, `bin`, `ceil`, `exp`, `floor`, `log`, `log10`, `log2`, `hex`, `pow`, `radians`, `shiftLeft`, `sqrt`, `round`, `factorial`, `sin`, `asin`, `tan`, `atan`, `degrees`, `radians`, …)
- Wyznaczające różnorodne wartości kontrolne (`crc32`, `hash`, `md5`, `sha1`, `sha2`)
- Nieagregujące (`abs`, `coalesce`, `broadcast`, `least`, `isnull`, `greatest`, `least`, `rand`, `when`, `spark_partition_id`, …)
- Sortujące (`asc`, `asc_nulls_first`, `asc_nulls_last`, `desc`, `desc_nulls_first`, `desc_nulls_last`)
- Działające na ciągach znaków (`ascii`, `concat`, `decode`, `encode`, `initcap`, `instr`, `length`, `levenshtein`, `lower`, `lpad`, `ltrim`, `regexp_extract`, `regexp_replace`, `reverse`, `split`, …)
- Analityczne (funkcje okna) (`cume_dist`, `dense_rank`, `lag`, `lead`, `ntile`, `percent_rank`, `row_number`)

4 SQL

4.1 Wprowadzenie

Aby móc korzystać ze składni SQL wystarczy zarejestrować DataFrame jako tymczasową perspektywę

- dostępną w ramach bieżącej sesji za pomocą metody `createOrReplaceTempView`
- dostępną globalnie za pomocą metody `createGlobalTempView` (odwołania realizowane jest wówczas za pomocą prefiksu predefiniowanej bazy danych `global_temp.` np.: `global_temp.g_cities`)

[21]:
```
df_cities.createOrReplaceTempView("cities")
df_countries.createOrReplaceTempView("countries")
```

[22]:
```
spark.sql("SELECT name, elevation FROM cities WHERE elevation > 3000").show()
```

```
+-------------+---------+
|         name|elevation|
+-------------+---------+
|        Lhasa|   4200.0|
|       La Paz|   3640.0|
|      El Alto|   4150.0|
|        Oruro|   3735.0|
|       Potosí|   4067.0|
|       Huaraz|   3052.0|
```

```
|         Cusco|   3399.0|
|  Huancavelica|   3676.0|
|      Huancayo|   3259.0|
|Cerro de Pasco|   4330.0|
|          Puno|   3830.0|
|       Juliaca|   3825.0|
+--------------+---------+
```

[23]:
```
spark.sql("""SELECT location.coordinates, co.name
          FROM cities ci JOIN countries co
          ON (ci.country = co.code)
          WHERE ci.name = 'New York'""").show()
```

```
+---------------+-------------+
|    coordinates|         name|
+---------------+-------------+
|[-74.01, 40.71]|United States|
+---------------+-------------+
```

[24]:
```
top3_highest = spark.sql("SELECT co.name, round(avg(elevation)) avg_elev " +
          "FROM cities ci JOIN countries co ON (ci.country = co.code) " +
          "GROUP BY co.name " +
          "HAVING count(*) > 80 " +
          "ORDER BY avg_elev DESC " +
          "LIMIT 3 ")
top3_highest.show()
```

```
+------+--------+
|  name|avg_elev|
+------+--------+
|Mexico|  1078.0|
|Turkey|   648.0|
|Brazil|   408.0|
+------+--------+
```

- Czym jest metoda `sql` w powyższym przykładzie?
- Czym jest metoda `show` w powyższym przykładzie?

4.2 Dostepna skladnia SQL

Od Spark 2.0 pelna obsluga SQL2003

4.3 Funkcjonalność Hive

Dzięki połączeniu funkcjonalności Spark SQL i Hive w SparkSession, możemy trwale zapisywać wyniki swoich działań

[25]:
```
spark.sql("drop table if exists nazwy_krajow")
spark.sql("create table nazwy_krajow (nazwa varchar(1000))")
```

[25]:
```
DataFrame[]
```

[26]:
```
spark.sql("drop table if exists kraje")
spark.sql("create table kraje as select * from cities")
```

[26]:
```
DataFrame[]
```

[27]:
```
spark.sql("SELECT name FROM kraje").show()
```

```
+------------+
|        name|
+------------+
|      Kavala|
|    Komotini|
|      Athina|
|    Peiraias|
|   Peristeri|
|    Acharnes|
|       Patra|
|      Kozani|
|    Ioannina|
|     Kerkyra|
|Thessaloniki|
|      Chania|
|      Durrës|
|   Ermoupoli|
|      Rhodes|
|     Tripoli|
|     Iraklio|
|       Lamia|
|    Chalkida|
|     Larissa|
+------------+
only showing top 20 rows
```

5 Typy danych

DataFrame zawiera obiekty `Row`, które nie są parametryzowane przez typ

Jednak ze względów optymalizacyjnych elementy `Row` korzystają z określonego zbioru typów, które odpowiadają określonym typom występującym w Scali lub w Pythonie

5.1 Typy proste:

Typ SQL	Typ Scali	Typ Pythona
`ByteType`	`Byte`	`int` lub `long`
`ShortType`	`Short`	`int` lub `long`
`IntegerType`	`Int`	`int` lub `long`
`LongType`	`Long`	`long`
`FloatType`	`Float`	`float`
`DoubleType`	`Double`	`float`
`BinaryType`	`Array[Byte]`	`bytearray`
`BooleanType`	`Boolean`	`bool`
`StringType`	`String`	`string`
`TimestampType`	`java.sql.Timestamp`	`datetime.datetime`
`DateType`	`java.sql.Date`	`datetime.date`

5.2 Typy złożone:

Typ SQL	Typ Scali	Typ Pythona
`ArrayType(elemType, nullable)`	`Array[T]`	`list`, `tuple` lub `array`
`MapType(keyType, valueType, valNullable)`	`Map[K, V]`	`dict`
`StructType(List[StructField])`	`case class`	`list` lub `tuple`

[28]:
```
spark.conf.set("spark.sql.repl.eagerEval.enabled",True)
```

6 Wizualizacja

6.1 Konwersja Spark DataFrame do Pandas DataFrame

Aby udostępnić wyniki zadań Sparka do wizualizacji za pomocą bibliotek Pythona warto dokonać konwersji Spark DataFrame do Pandas DataFrame

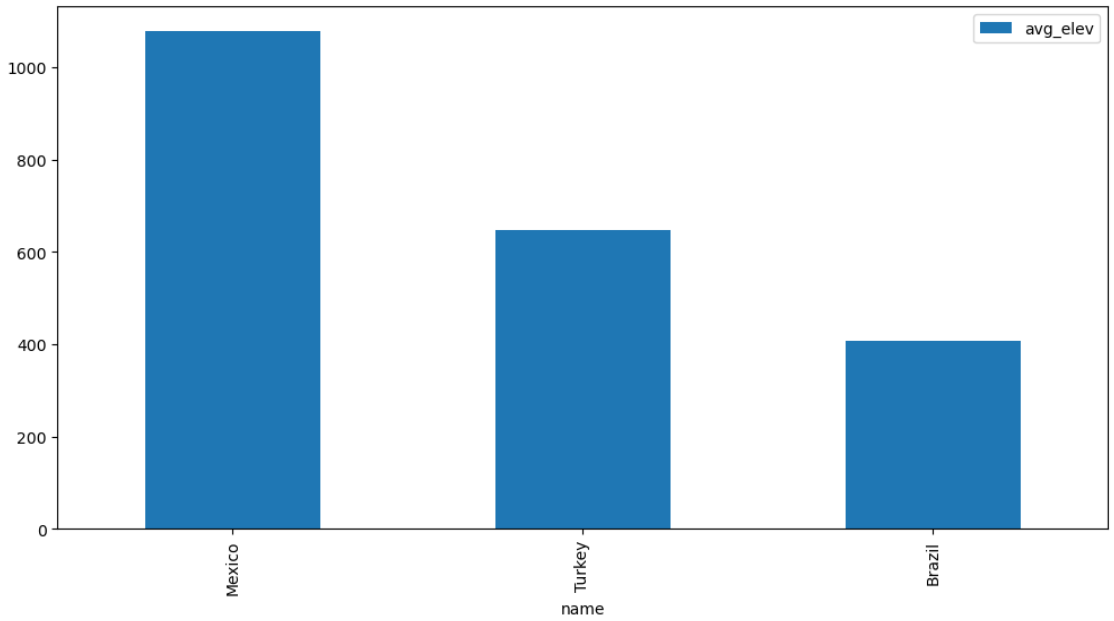
[29]:
```
spark.conf.set("spark.sql.execution.arrow.enabled", "false")
pandas_top3_highest = top3_highest.toPandas()

pandas_top3_highest.set_index('name', inplace=True)
pandas_top3_highest.head()
```

[29]:
```
        avg_elev
name
Mexico    1078.0
Turkey     648.0
Brazil     408.0
```

[30]:
```
import matplotlib.pyplot as plt
```

[31]:
```
pandas_top3_highest.plot(kind='bar',figsize=(12,6));
```



7 A co ponad DataFrame API?

DataFrame API dla języka Python jest wzorowany na DataFrame API dla języka Scala i Java. Podobieństwo jest na tyle duże, że przesiadka pomiędzy językami jest stosunkowo prosta.

Problem jest w tym, że API to daleko odbiega od tego, które wykorzystywane jest w popularnych bibliotekach języka Python, w szczególności bibliotece Pandas.

Aby wyeliminować ten problem Spark stworzył *Pandas API on Spark*.

Jednak ten temat, podobnie jak biblioteki usprawniające przetwarzanie przy wykorzystaniu języka Python (np. Apache Arrow), to osobna opowieść.

[ ]: