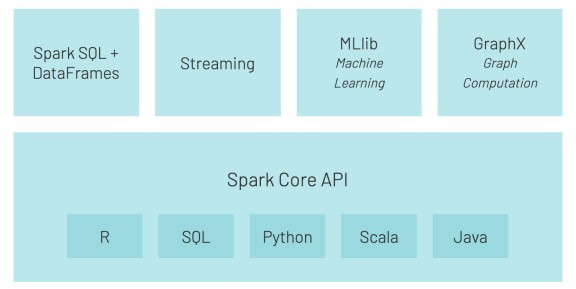
Today we will look into the Spark SQL and DataFrames that is using Spark Core API.



“Spark SQL is a spark module for structured data processing and data querying. It provides programming abstraction called DataFrames and can also serve as distributed SQL query engine. It enables unmodified Hadoop Hive queries to run up to 100x faster on existing deployments and data. It also provides powerful integration with the rest of the Spark ecosystem (e.g.: integrating SQL query processing with machine learning).” (Apache Spark Tutorial).

Start your Azure Databricks workspace and create new Notebook. I named mine as: *Day22\_SparkSQL*

and set the language: *SQL*. Now let’s explore the functionalities of Spark SQL.

# Loading Data

We will load data from */databricks-datasets* using Spark SQL, R and Python languages. The CSV dataset will be **data\_geo.csv** in the following folder:

%scala

display(dbutils.fs.ls("/databricks-datasets/samples/population-vs-price"))

# Loading using Python

%python

data = spark.read.csv("/databricks-datasets/samples/population-vs- price/data\_geo.csv", header="true", inferSchema="true")

And materialize the data using to create a view with name *data\_geo\_py*:

%python data.createOrReplaceTempView("data\_geo\_py")

And run the following SQL Statement:

SELECT \* FROM data\_geo\_py LIMIT 10

# Loading using SQL

DROP TABLE IF EXISTS data\_geo;

CREATE TABLE data\_geo

USING com.databricks.spark.csv

OPTIONS (path "/databricks-datasets/samples/population-vs-price/data\_geo.csv",

header "true", inferSchema "true")

And run the following SQL Statement:

SELECT \* FROM data\_geo LIMIT 10

# Loading using R

%r library(SparkR)

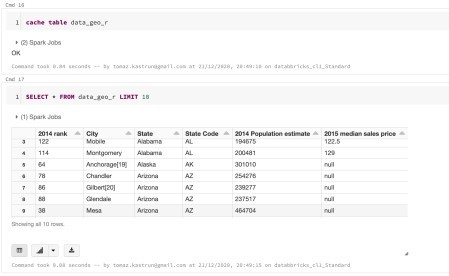
data\_geo\_r <- read.df("/databricks-datasets/samples/population-vs-price/ data\_geo.csv", source = "csv", header="true", inferSchema = "true") registerTempTable(data\_geo\_r, "data\_geo\_r")

Cache the results:

CACHE TABLE data\_geo\_r

And run the following SQL Statement:

SELECT \* FROM data\_geo\_r LIMIT 10



All three DataFrames are the same (unless additional modification are done; like: dropping rows with null values, etc).

# Viewing DataFrame

Viewing DataFrame is done by simple SELECT statement, the ANSI SQL Standard. E.g.:

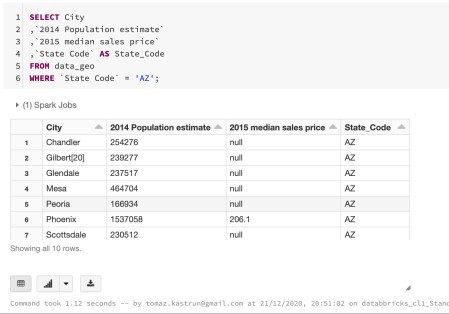
SELECT City

,`2014 Population estimate`

,`2015 median sales price`

,`State Code` AS State\_Code FROM data\_geo

WHERE `State Code` = 'AZ';



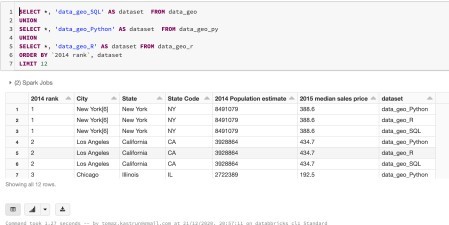
You can also combine all three DataFrames that were imported using three different languages (SQL, R, Python).

SELECT \*, 'data\_geo\_SQL' AS dataset FROM data\_geo UNION

SELECT \*, 'data\_geo\_Python' AS dataset FROM data\_geo\_py UNION

SELECT \*, 'data\_geo\_R' AS dataset FROM data\_geo\_r ORDER BY `2014 rank`, dataset

LIMIT 12



# Running SQL

* 1. **Date and Time functions**

SELECT

CURRENT\_TIMESTAMP() AS now

,date\_format(CURRENT\_TIMESTAMP(), "L") AS Month\_

,date\_format(CURRENT\_TIMESTAMP(), "LL") AS Month\_LeadingZero

,date\_format(CURRENT\_TIMESTAMP(), "y") AS Year\_

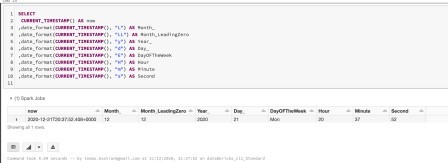
,date\_format(CURRENT\_TIMESTAMP(), "d") AS Day\_

,date\_format(CURRENT\_TIMESTAMP(), "E") AS DayOFTheWeek

,date\_format(CURRENT\_TIMESTAMP(), "H") AS Hour

,date\_format(CURRENT\_TIMESTAMP(), "m") AS Minute

,date\_format(CURRENT\_TIMESTAMP(), "s") AS Second



# Built-in functions

SELECT

COUNT(\*) AS Nof\_rows

,SUM(`2014 rank`) AS Sum\_Rank

,AVG(`2014 rank`) AS Avg\_Rank

,SUM(CASE WHEN `2014 rank` > 150 THEN 1 ELSE -1 END) AS Sum\_case

,STD(`2014 rank`) as stdev

,MAX(`2014 rank`) AS Max\_Val

,MIN(`2014 rank`) AS Min\_Val

,KURTOSIS (`2014 rank`) as Kurt

,SKEWNESS(`2014 rank`) AS Skew

,CAST(SKEWNESS(`2014 rank`) AS INT) AS Skew\_cast FROM data\_geo



# SELECT INTO

You can also store results using SELECT INTO statement, with table being predifined:

DROP TABLE IF EXISTS tmp\_data\_geo;

CREATE TABLE tmp\_data\_geo (`2014 rank` INT, State VARCHAR(64), `State Code` VARCHAR(2))

INSERT INTO tmp\_data\_geo FROM data\_geo SELECT

`2014 rank`

,State

,`State Code`

WHERE `2014 rank` >= 50 AND `2014 rank` < 60 AND `State Code` = "C";

SELECT \* FROM tmp\_data\_geo;



# JOIN

SELECT

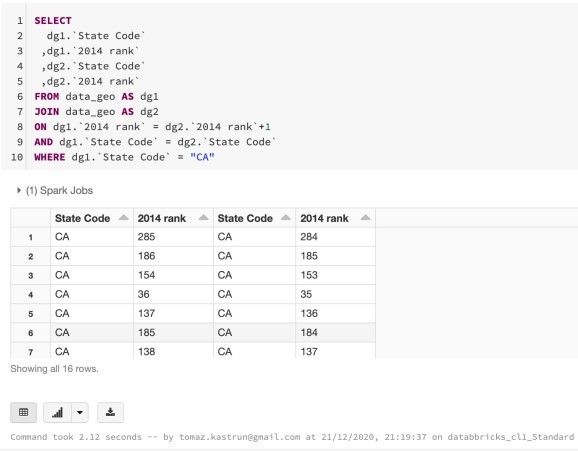
dg1.`State Code`

,dg1.`2014 rank`

,dg2.`State Code`

,dg2.`2014 rank` FROM data\_geo AS dg1 JOIN data\_geo AS dg2

ON dg1.`2014 rank` = dg2.`2014 rank`+1 AND dg1.`State Code` = dg2.`State Code` WHERE dg1.`State Code` = "CA"



# Common Table Expressions

WITH cte AS (

SELECT \* FROM data\_geo

WHERE `2014 rank` >= 50 AND `2014 rank` < 60

)

SELECT \* FROM cte;

# Inline tables

SELECT \* FROM VALUES

("WA", "Seattle"),

("WA", "Tacoma"),

("WA", "Spokane") AS data(StateName, CityName)



# EXISTS

WITH cte AS (

SELECT \* FROM data\_geo

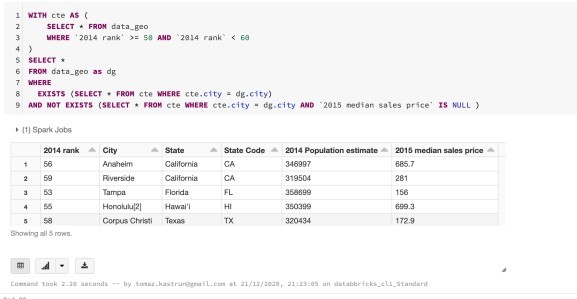
WHERE `2014 rank` >= 50 AND `2014 rank` < 60

) SELECT \*

FROM data\_geo as dg WHERE

EXISTS (SELECT \* FROM cte WHERE cte.city = dg.city)

AND NOT EXISTS (SELECT \* FROM cte WHERE cte.city = dg.city AND `2015 median sales price` IS NULL )



# Window functions

SELECT

City

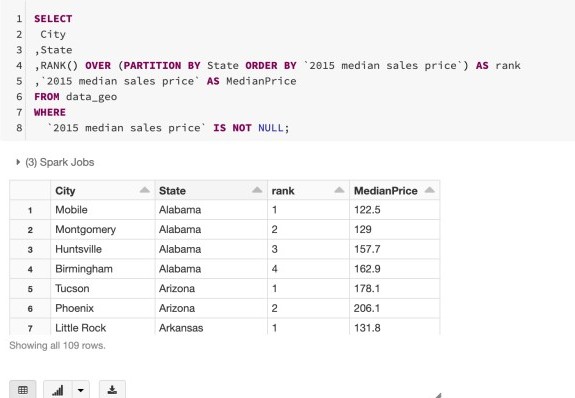
,State

,RANK() OVER (PARTITION BY State ORDER BY `2015 median sales price`) AS rank

,`2015 median sales price` AS MedianPrice FROM data\_geo

WHERE

`2015 median sales price` IS NOT NULL;

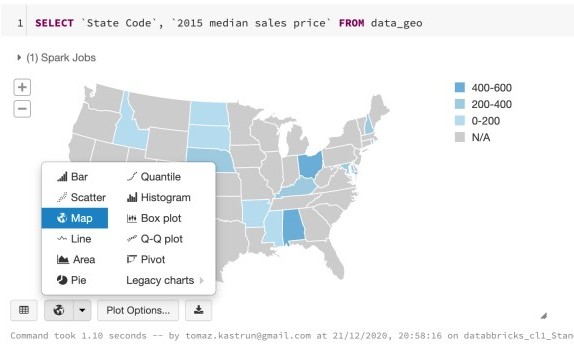


# Exploring the visuals

Results of a SQL SELECT statements that are returned as a table, can also be visualised. Given the following SQL Statement:

SELECT `State Code`, `2015 median sales price` FROM data\_geo

in the result cell you can select the plot icon and pick Map.



Furthermore, using “Plot Options…” you can change the settings of the variables on the graph,

aggregations and data series. With additional query:

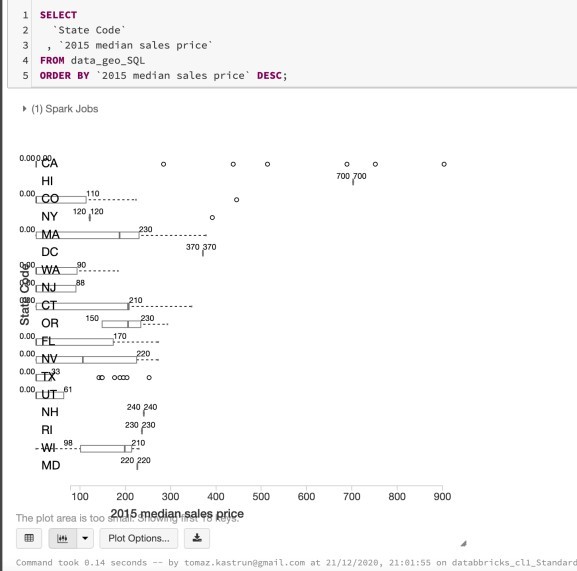
SELECT

`State Code`

, `2015 median sales price` FROM data\_geo\_SQL

ORDER BY `2015 median sales price` DESC;

you can also create a box-plot; again selecting the desired plot type.



There are also many other visuals available and much more SQL statements to explore and feel free to go a step further and beyond this blogpost.