

# Data Engineering in the Cloud

## Manipulating Dataframes in Azure Databricks

Xuemao Zhang  
East Stroudsburg University

January 18, 2025

# Outline

- Caching a DataFrame
- Removing Duplicate Data
- Manipulating Date and Time
- Removing DataFrame Columns

# Caching a DataFrame

- Now, lunch the Azure Databricks workspace and create another notebook Notebook2
  - ▶ It will take some time to start the cluster
  - ▶ Repeat what we did in the last three labs if you do not have a cluster

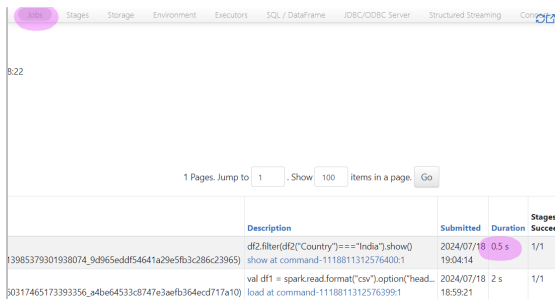
```
val df1 = spark.read.format("csv").  
option("header", "true").  
load("dbfs:/FileStore/shared_uploads/xzhang2@esu.edu/population-4.csv")  
val df2 = df1  
  .withColumnRenamed("Country (or dependency)", "Country")  
  .withColumnRenamed("Population (2020)", "Population_2020")  
  .withColumnRenamed("Yearly Change", "Yearly_Change")  
  .withColumnRenamed("Net Change", "Net_Change")  
  .withColumnRenamed("Density (P/Km²)", "Density_PKm2")  
  .withColumnRenamed("Land Area (Km²)", "Land_Area_Km2")  
  .withColumnRenamed("Migrants (net)", "Migrants_net")  
  .withColumnRenamed("Fert. Rate", "Fertility_Rate")  
  .withColumnRenamed("Med. Age", "Median_Age")  
  .withColumnRenamed("Urban Pop %", "Urban_Pop_Percent")  
  .withColumnRenamed("World Share", "World_Share")  
  
df2.printSchema()
```

# Caching a DataFrame

- Filter one of the rows and run the cell by executing the command given below.

```
df2.filter(df2("Country")==="India").show()
```

- Click on Spark Jobs option and then click on View. This process has taken 0.5 seconds.



	Description	Submitted	Duration	Stages
13985379301938074_9d965eddf54641a29e5fb3c286c23965	df2.filter(df2("Country")==="India").show() show at command-1118811312576400:1	2024/07/18 19:04:14	0.5 s	1/1
50317465173393356_a4be64533c8747e3aefb364ecd717a10	val df1 = spark.read.format("csv").option("head... load at command-1118811312576399:1	2024/07/18 18:59:21	2 s	1/1

# Caching a DataFrame

- **Caching data** is a good method for faster subsequent queries. This will help fetch subsequent data quickly and efficiently. To do so, perform caching of a DataFrame.
- Cache the DataFrame by executing the command given below.

```
df2.cache()
```

- And then run the above code again

```
df2.filter(df2("Country")==="India").show()
```

Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
df2.filter(df2("Country")==="India").show() show at command-1118811312576402:1	2024/07/18 19:11:39	73 ms	1/1	1/1
df2.filter(df2("Country")==="India").show() show at command-1118811312576402:1	2024/07/18 19:11:02	0.1 s	1/1	1/1
df2.filter(df2("Country")==="India").show() show at command-1118811312576402:1	2024/07/18 19:10:00	0.9 s	1/1	1/1
df2.filter(df2("Country")==="India").show() show at command-1118811312576400:1	2024/07/18 19:04:14	0.5 s	1/1	1/1

# Removing Duplicate Data

- Now we work on a new data set `population_duplicate_data.csv`
- Follow what we did before, create another notebook Notebook3 and upload the data

```
val df3 = spark.read.format("csv").  
  option("header", "true").  
  load("dbfs:/FileStore/shared_uploads/xzhang2@esu.edu/population_duplicate_data.csv")  
df3.printSchema()  
df3.cache()
```

# Removing Duplicate Data

Just now (1s) 2 Scala

```
df3.show()
```

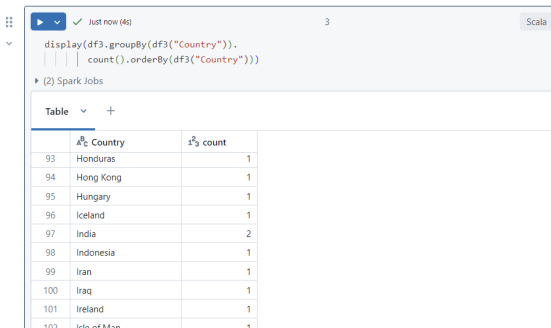
(1) Spark Jobs

Country	Population	YearlyChange	NetChange	Density	LandArea	Migrants	Fert_Rate	Med_Age	UrbanPop	WorldShare
China	1440297825	0.39%	5540090	153	9388211	-348399	1.7	38	61%	18.47%
India	1382345085	0.99%	13586631	464	2973190	-532687	2.2	28	35%	17.70%
India	1382345085	0.99%	13586631	464	2973190	-532687	2.2	28	35%	17.70%
United States	331341050	0.59%	1937734	36	9147420	954806	1.8	38	83%	4.25%
United States	331341050	0.59%	1937734	36	9147420	954806	1.8	38	83%	4.25%
Indonesia	274021604	1.07%	2898047	151	1811570	-98955	2.3	30	56%	3.51%
Pakistan	221612785	2.00%	4327022	287	770880	-233379	3.6	23	35%	2.83%
Brazil	212821986	0.72%	1509890	25	8358140	21200	1.7	33	88%	2.73%

# Removing Duplicate Data

- Group the values by country to understand how many of these have duplicate values by executing the command given below.
  - Aggregate the data using the `groupBy()` function

```
val groupedDF3 = df3.groupBy(df3("Country")).  
    count().orderBy(df3("Country"))  
display(groupedDF3)
```



The screenshot shows a Scala REPL interface. At the top, a status bar indicates "Just now (4s)" and "3" lines of code. The code entered is `display(df3.groupBy(df3("Country")).count().orderBy(df3("Country")))`. Below the code, it says "(2) Spark Jobs". The output is a table with two columns: "Country" and "count". The table lists 11 countries with their respective counts: Honduras (1), Hong Kong (1), Hungary (1), Iceland (1), India (2), Indonesia (1), Iran (1), Iraq (1), Ireland (1), and Israel (1).

	Country	count
93	Honduras	1
94	Hong Kong	1
95	Hungary	1
96	Iceland	1
97	India	2
98	Indonesia	1
99	Iran	1
100	Iraq	1
101	Ireland	1
102	Israel	1



# Removing Duplicate Data

```
val duplicatesDF3 = groupedDF3.filter("count > 1")  
display(duplicatesDF3)
```

Just now (1s) 4

```
val duplicatesDF3 = groupedDF3.filter("count > 1")  
display(duplicatesDF3)
```

(2) Spark Jobs

duplicatesDF3: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Country: string, count: long]

Table +

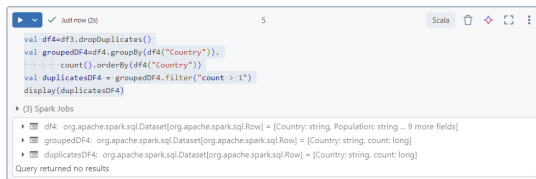
	Country	count
1	Australia	2
2	Bangladesh	2
3	Egypt	2
4	Finland	2
5	India	2
6	Italy	2
7	Japan	2
8	Maldives	2
9	Spain	2
10	Switzerland	2
11	United States	2

# Removing Duplicate Data

- Remove the duplicate values by executing the `dropDuplicates()` function .

```
val df4=df3.dropDuplicates()
val groupedDF4=df4.groupBy(df4("Country")).
    count().orderBy(df4("Country"))

val duplicatesDF4 = groupedDF4.filter("count > 1")
display(duplicatesDF4)
```



```
▶ Just now (2s) 5 Scala
```

```
val df4=df3.dropDuplicates()
val groupedDF4=df4.groupBy(df4("Country")).
    count().orderBy(df4("Country"))
val duplicatesDF4 = groupedDF4.filter("count > 1")
display(duplicatesDF4)
```

▶ (3) Spark Jobs

- ▶ df4: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Country: string, Population: string ... 9 more fields]
- ▶ groupedDF4: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Country: string, count: long]
- ▶ duplicatesDF4: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Country: string, count: long]

Query returned no results

# Manipulating Date and Time Values

- Now we work on a new data set `population_with_time.csv`
- Follow what we did before, create another notebook Notebook4 and upload the data

```
val df5 = spark.read.format("csv").option("header", "True").  
load("dbfs:/FileStore/shared_uploads/  
      xzhang2@esu.edu/population_with_time.csv")  
df5.printSchema()  
  
df5.cache()
```

# Manipulating Date and Time Values



Just now (<1s)

```
display(df5.select(df5("Time")))
```

► (1) Spark Jobs

Table

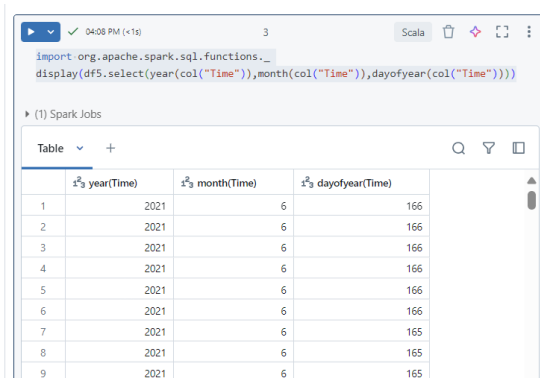


	A <sup>B</sup> <sub>C</sub> Time
1	2021-06-15T04:44:38.22...
2	2021-06-15T04:44:21.54...
3	2021-06-15T04:44:21.70...
4	2021-06-15T04:44:31.33...
5	2021-06-15T04:44:12.53...
6	2021-06-15T04:44:16.03...
7	2021-06-14T17:57:02.24...
8	2021-06-14T17:55:17.61...
9	2021-06-14T17:55:18.57...
10	2021-06-14T17:56:51.94...

# Manipulating Date and Time Values

- Display the year, month, and day separately in the “Time” column of the population\_with\_time file by executing the command given below.

```
import org.apache.spark.sql.functions._
display(df5.select(year(col("Time")),
                    month(col("Time")),dayofyear(col("Time"))))
```



The screenshot shows a Databricks notebook interface. At the top, there's a toolbar with a play button, a status bar showing '04:08 PM (<1s)' and '3' lines of code, and buttons for 'Scala', 'Copy', 'Run', 'Fullscreen', and 'More'. Below the toolbar, the code cell contains the following Spark SQL query:

```
import org.apache.spark.sql.functions._
display(df5.select(year(col("Time")),month(col("Time")),dayofyear(col("Time"))))
```

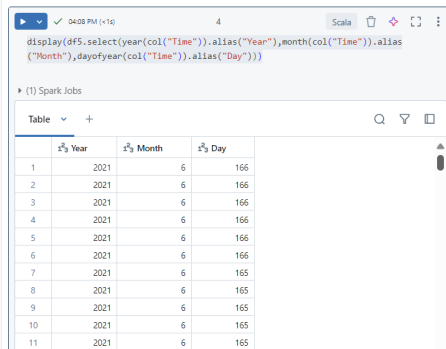
Below the code cell, there's a section titled '(1) Spark Jobs'. Underneath, there's a table view with the following columns: 'Table', 'year(Time)', 'month(Time)', and 'dayofyear(Time)'. The table contains 9 rows of data.

	year(Time)	month(Time)	dayofyear(Time)
1	2021	6	166
2	2021	6	166
3	2021	6	166
4	2021	6	166
5	2021	6	166
6	2021	6	166
7	2021	6	165
8	2021	6	165
9	2021	6	165

# Manipulating Date and Time Values

- We can also replace the name of any column with any other string. For example, we can distribute the Time column into year, month, and day columns using the alias() function by executing the command given below.

```
display(df5.select(year(col("Time")).alias("Year"),  
month(col("Time")).alias("Month"),  
dayofyear(col("Time")).alias("Day")))
```



The screenshot shows a Databricks notebook interface. At the top, there's a status bar with a play button, a checkmark, the time '04:08 PM (-11s)', the page number '4', and the language 'Scala'. Below this is a code editor containing the following Spark SQL command:

```
display(df5.select(year(col("Time")).alias("Year"),month(col("Time")).alias("Month"),dayofyear(col("Time")).alias("Day")))
```

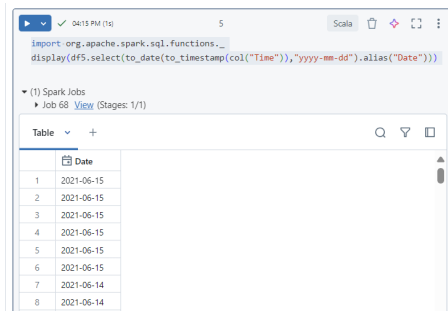
Below the code editor, it says '(1) Spark Jobs'. Underneath that is a table view. The table has a header row with columns: 'Year', 'Month', and 'Day'. The 'Year' column contains the value '2021' for all rows. The 'Month' column contains the value '6' for all rows. The 'Day' column contains values ranging from '165' to '166'. There are 11 rows in total, indexed from 1 to 11.

	Year	Month	Day
1	2021	6	166
2	2021	6	166
3	2021	6	166
4	2021	6	166
5	2021	6	166
6	2021	6	166
7	2021	6	165
8	2021	6	165
9	2021	6	165
10	2021	6	165
11	2021	6	165

# Manipulating Date and Time Values

- Convert the date into a particular format using the `to_date()` function.  
Execute the command given below.
  - ▶ code first converts the string to a timestamp using the `to_timestamp` function with the appropriate format that includes time and timezone, and then converts the timestamp to a date with `to_date`.

```
display(df5.select(to_date(to_timestamp(col("Time")),  
                           "yyyy-mm-dd").alias("Date")))
```



04:15 PM (1s) 5 Scala

```
import org.apache.spark.sql.functions._  
display(df5.select(to_date(to_timestamp(col("Time")), "yyyy-mm-dd").alias("Date")))
```

▼ (1) Spark Jobs  
▶ Job 68 [View](#) (Stages: 1/1)

Table +

	Date
1	2021-06-15
2	2021-06-15
3	2021-06-15
4	2021-06-15
5	2021-06-15
6	2021-06-15
7	2021-06-14
8	2021-06-14

# Removing DataFrame Columns

- Suppose we have a new data frame df6

```
val df6 = df5.withColumn("Date",  
    to_date(to_timestamp(col("Time")), "yyyy-MM-dd"))  
df6.printSchema()
```



# Removing DataFrame Columns

- To remove one of the columns (for example, WorldShare) use the `df.drop()` function and add the name of the column that has to be deleted
  - ▶ You can see that the WorldShare column is now removed from the Dataframe.

```
val df7 = df6.drop("WorldShare")  
df7.printSchema()
```



The screenshot shows a Scala IDE window titled "Just now (1s)" with a Scala file. The code defines `df7` as `df6.drop("WorldShare")` and prints its schema. The output shows that the `WorldShare` column has been removed from the original DataFrame.

```
val df7 = df6.drop("WorldShare")  
df7.printSchema()
```

```
df7: org.apache.spark.sql.DataFrame = [Country: string, Population: string ... 10 more fields]  
root  
|-- Country: string (nullable = true)  
|-- Population: string (nullable = true)  
|-- YearlyChange: string (nullable = true)  
|-- NetChange: string (nullable = true)  
|-- Density: string (nullable = true)  
|-- LandArea: string (nullable = true)  
|-- Migrants: string (nullable = true)  
|-- Fert_Rate: string (nullable = true)  
|-- Med_Age: string (nullable = true)  
|-- UrbanPop: string (nullable = true)  
|-- Time: string (nullable = true)  
|-- Date: date (nullable = true)  
  
df7: org.apache.spark.sql.DataFrame = [Country: string, Population: string ... 10 more fields]
```

# Summary and Preview

- Azure Databricks: An analytics platform based on Apache Spark that provides collaborative data science and machine learning environments
- Azure Synapse Analytics: An integrated analytics service that combines big data and data warehousing. It allows for data ingestion, preparation, management, and serving for business intelligence and machine learning purposes.
  - ▶ Azure Synapse can handle data warehousing, ETL processes, and SQL-based analytics,
- We can use Synapse's integrated data exploration, preparation, and management capabilities to transform and clean our data
  - ▶ In Synapse, output our processed data to an Azure Data Lake Storage (ADLS) Gen2 account.
  - ▶ Then Mount the ADLS storage in Databricks to access the prepared data.
  - ▶ Once mounted, read the data in Databricks for further processing.

# License



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).