ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

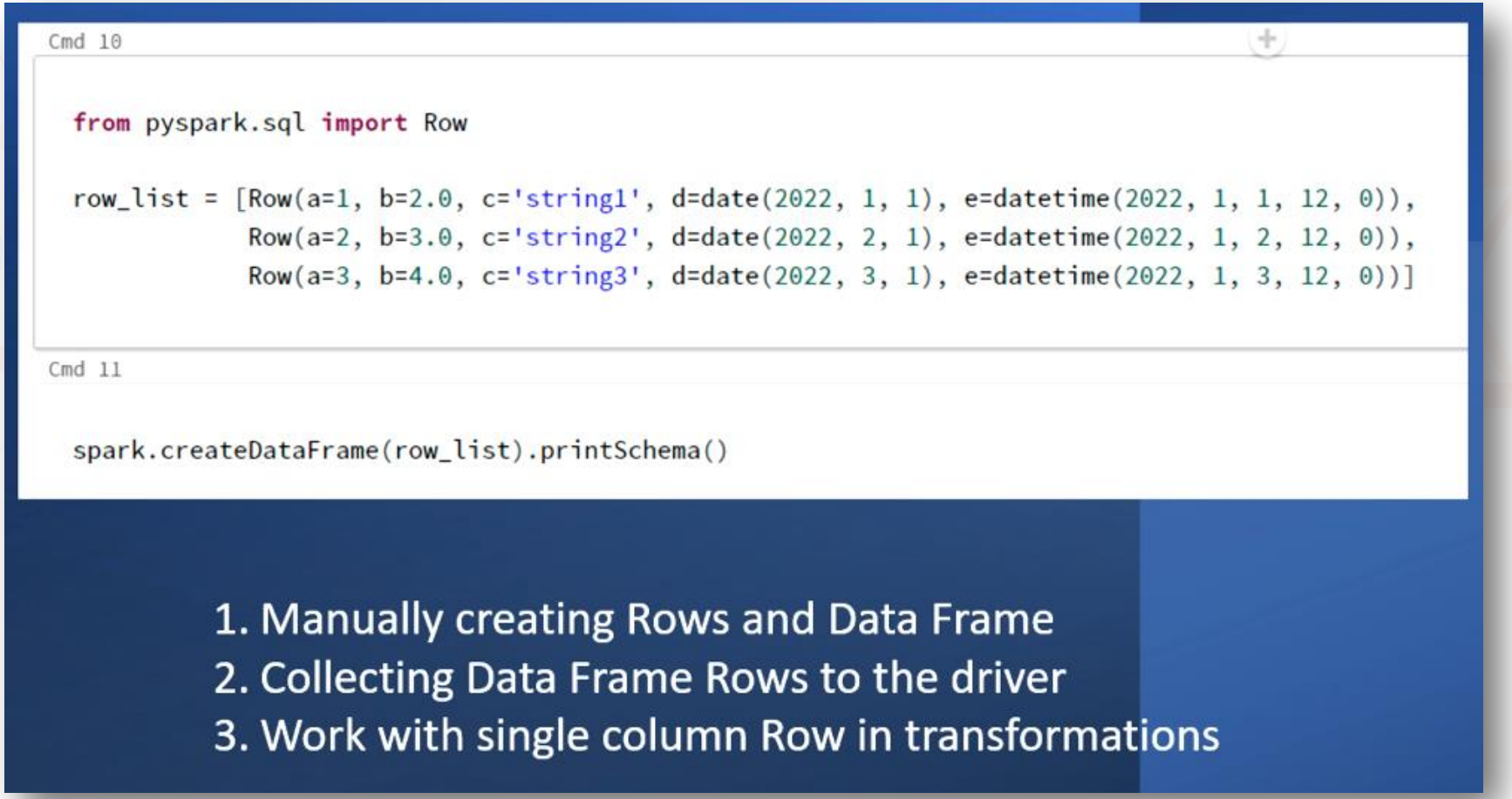
Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Spark Data Frame Rows

Spark DataFrame is similar to a table of rows and columns. A row in a DataFrame is a single record represented by an object of type Row. Most of the time, you do not directly work with the entire row. However, we have three specific scenarios when you might have to work with the Row object:

1. Manually creating Rows and DataFrame.
2. Collecting DataFrame rows to the driver.
3. Work with single column Rows in transformations

The first two scenarios are primarily used in unit testing or during development. And we already learned the first scenario in an earlier lecture. Here is a screenshot of the earlier example shown below.
We manually created a list of Row objects and used it to create a DataFrame.

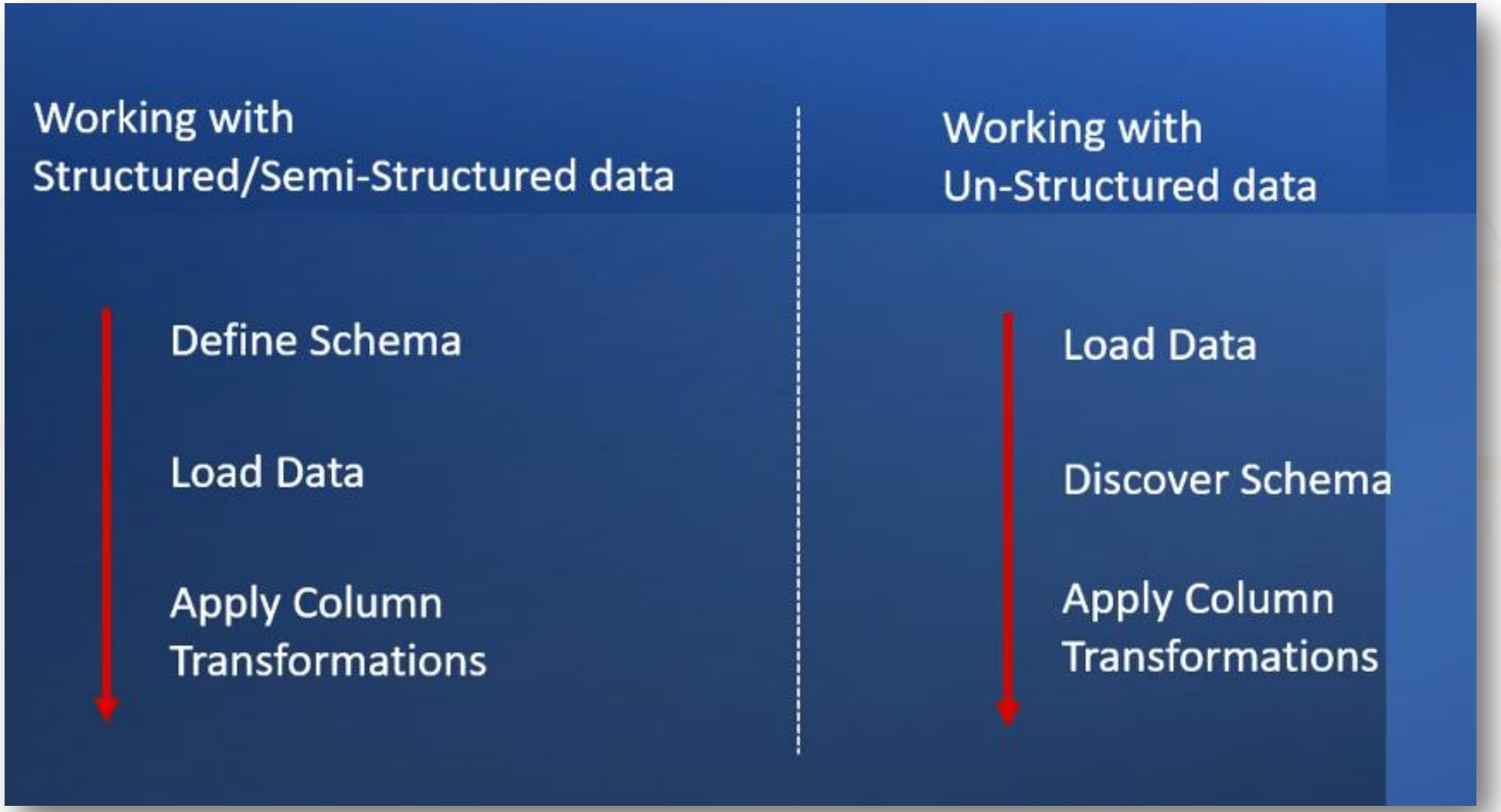


The third scenario of “working with single column rows in transformations” is mainly used with unstructured data.

I have this log file shown below. This file contains the application log from the Apache web server. This file contains a lot of information. But we are interested in the HTTP:// type of URLs. These are referrers of our website. They referred users to your website, and you want to take them out from this log and count who sent how many users.

```
83.149.9.216 - - [17/May/2015:10:05:03 +0000] "GET /presentations/logstash-monitorama-2013/images/kibana-search.png HTTP/1.1" 200 203023
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:43 +0000] "GET /presentations/logstash-monitorama-2013/images/kibana-dashboard3.png HTTP/1.1" 200 171717
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:47 +0000] "GET /presentations/logstash-monitorama-2013/plugin/highlight/highlight.js HTTP/1.1" 200 26185
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:12 +0000] "GET /presentations/logstash-monitorama-2013/plugin/zoom-js/zoom.js HTTP/1.1" 200 7697
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:07 +0000] "GET /presentations/logstash-monitorama-2013/plugin/notes/notes.js HTTP/1.1" 200 2892
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:34 +0000] "GET /presentations/logstash-monitorama-2013/images/sad-medic.png HTTP/1.1" 200 430406
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:57 +0000] "GET /presentations/logstash-monitorama-2013/css/fonts/Roboto-Bold.ttf HTTP/1.1" 200 38720
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:50 +0000] "GET /presentations/logstash-monitorama-2013/css/fonts/Roboto-Regular.ttf HTTP/1.1" 200 41820
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:24 +0000] "GET /presentations/logstash-monitorama-2013/images/frontend-response-codes.png HTTP/1.1" 200 5
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:50 +0000] "GET /presentations/logstash-monitorama-2013/images/kibana-dashboard.png HTTP/1.1" 200 321631
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:46 +0000] "GET /presentations/logstash-monitorama-2013/images/Dreamhost_logo.svg HTTP/1.1" 200 2126
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:11 +0000] "GET /presentations/logstash-monitorama-2013/images/kibana-dashboard2.png HTTP/1.1" 200 394967
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
83.149.9.216 - - [17/May/2015:10:05:19 +0000] "GET /presentations/logstash-monitorama-2013/images/apache-icon.gif HTTP/1.1" 200 8095
"http://semicomplete.com/presentations/logstash-monitorama-2013/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML,
Chrome/32.0.1700.77 Safari/537.36"
```

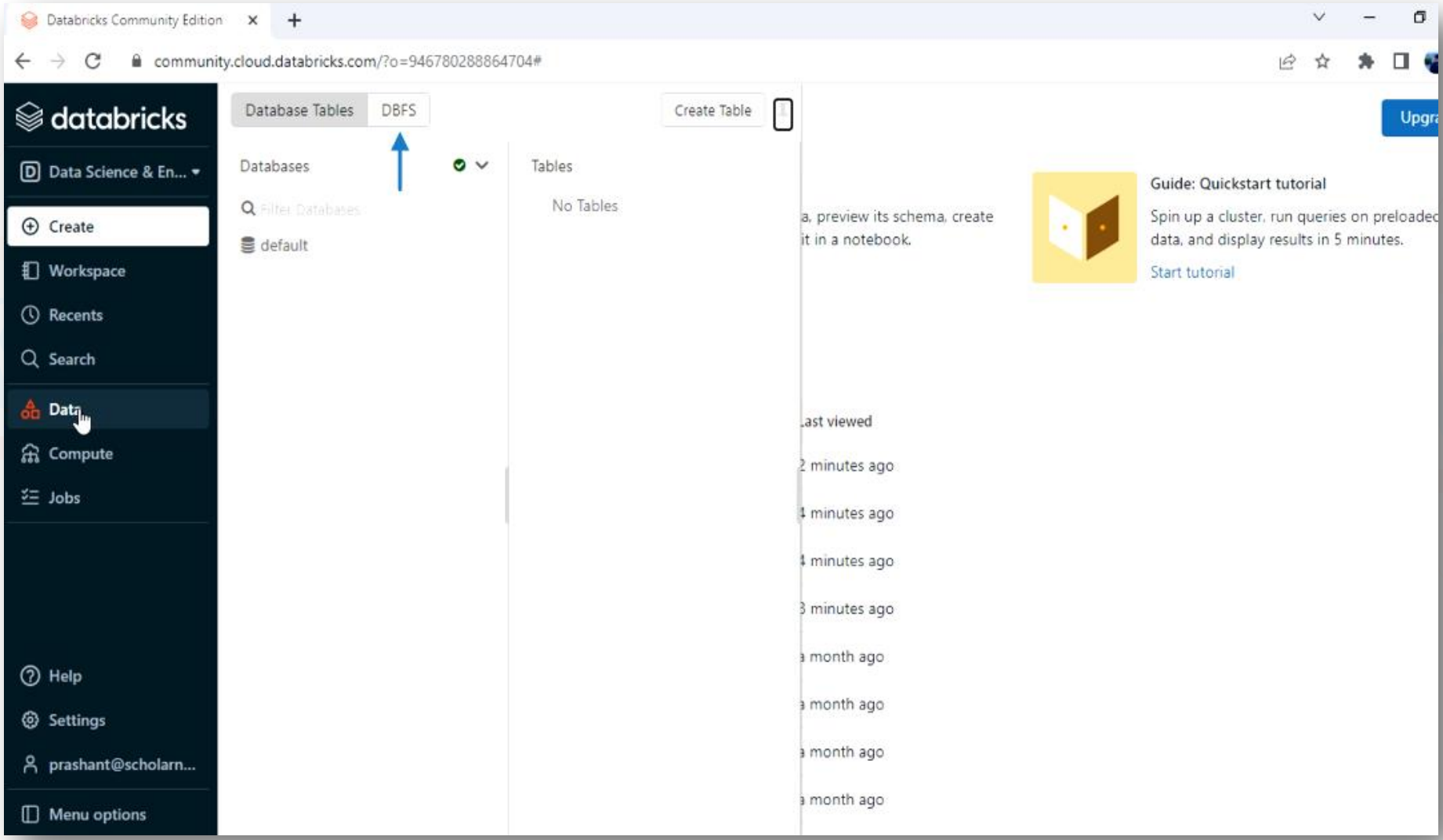
Spark allows you to work with structured/semi-structured and unstructured data. However, the approach is different as shown below.



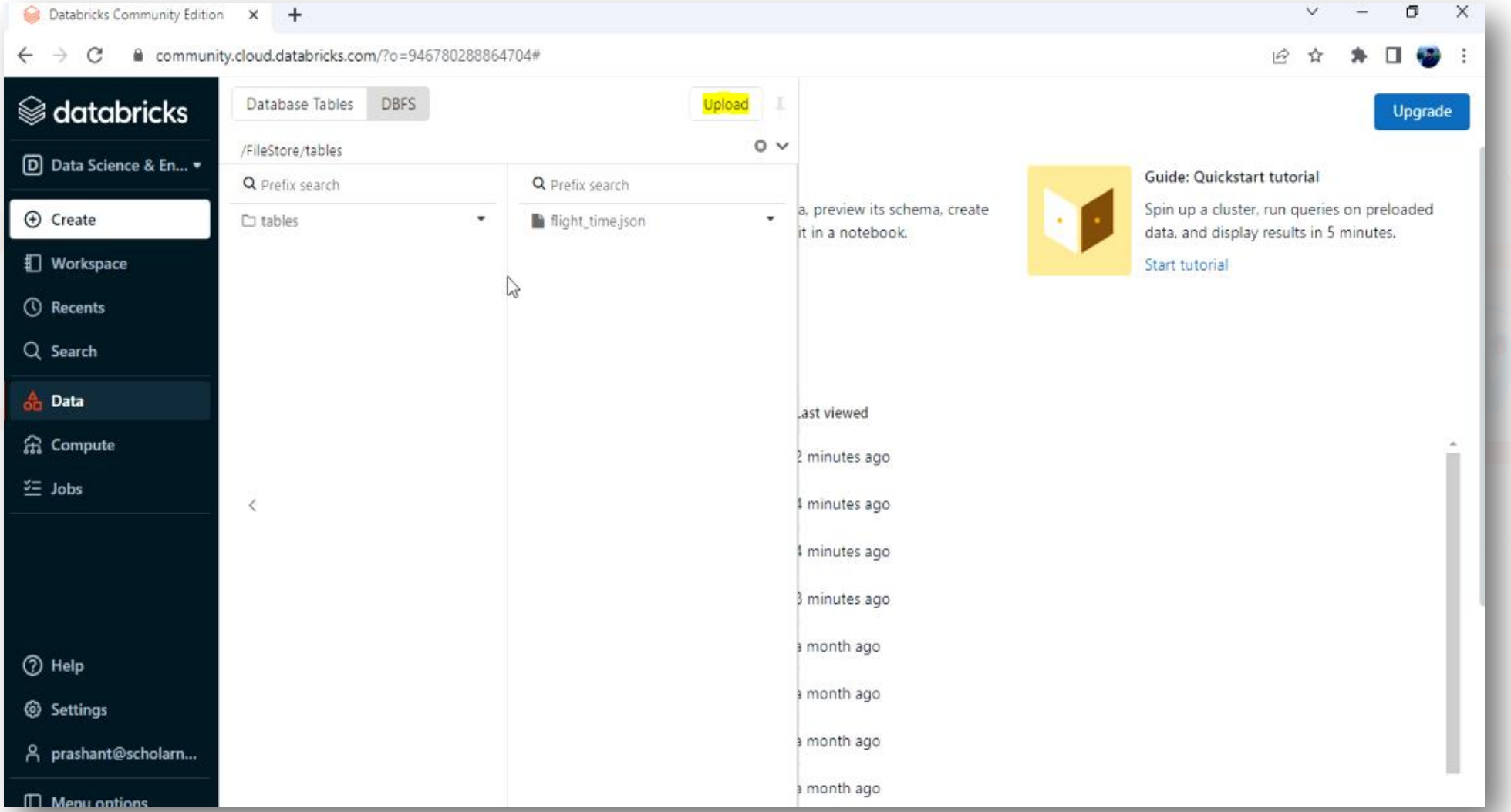
When you have structured or semi-structured data, you will define the schema of your data source.
Load it to a Spark Dataframe.
And start applying transformations.

But what if you have unstructured data.
The approach is different.
Load the unstructured data into a Dataframe. You won't have a meaningful schema. It is most likely a line of text in each row.
But then you can discover a schema and give your Dataframe a row/column structure.
Basically, we convert the unstructured data into structured and start transforming and analysing it.

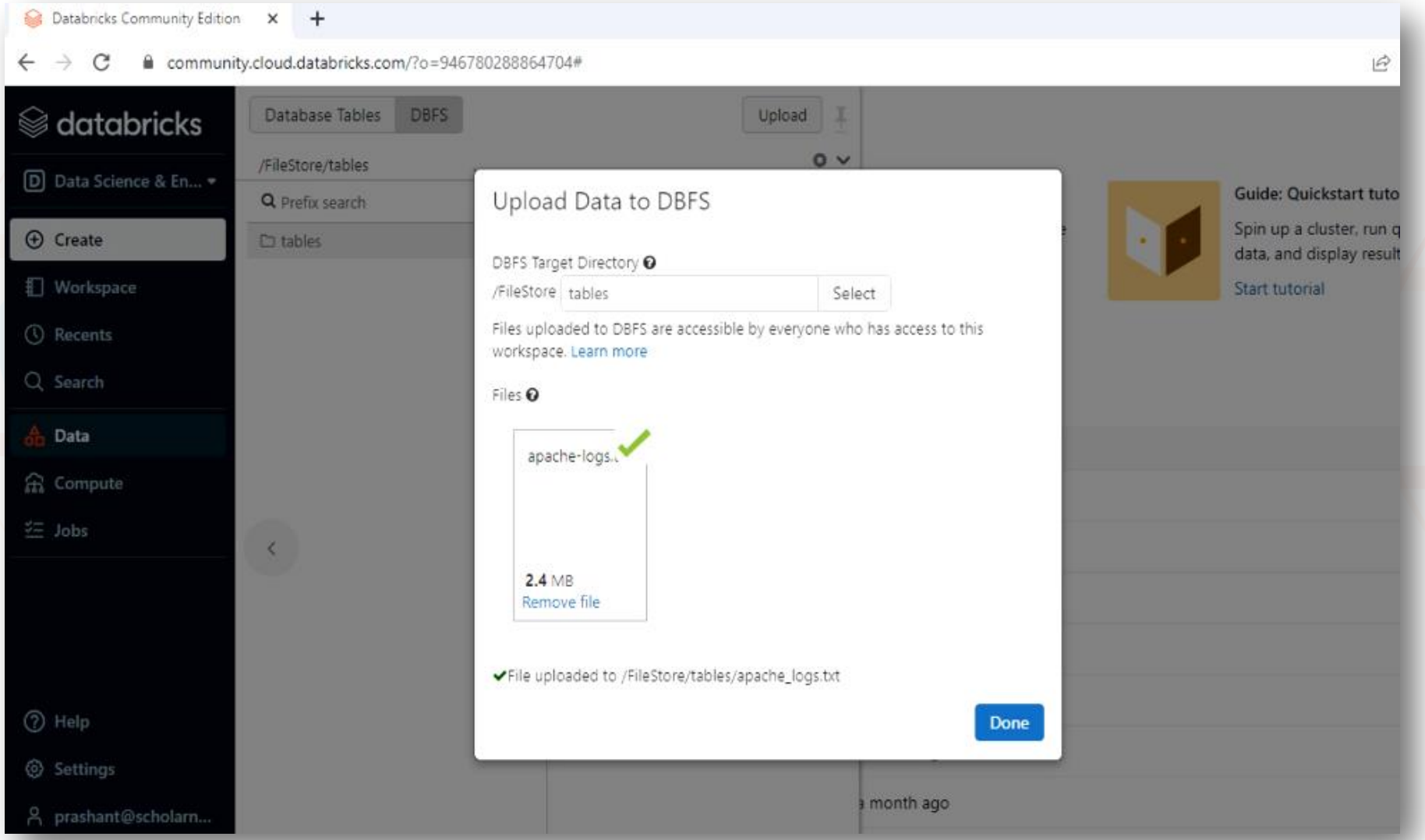
Go to your data bricks workspace and upload the log file into DBFS.
Go to the Data menu. And you will see the DBFS tab. Click that, and you can navigate to the filesystem.



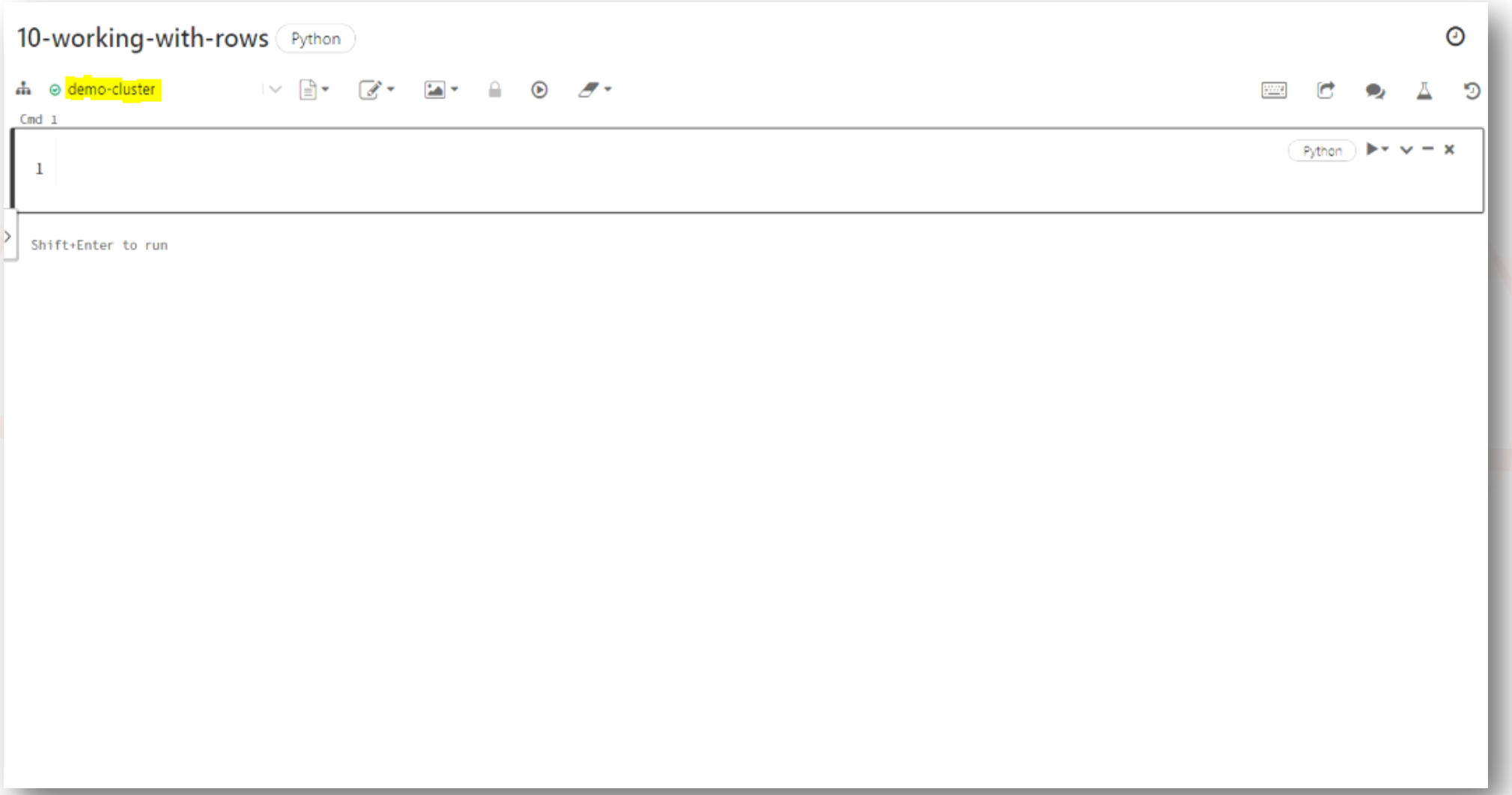
Select Filestore/tables and then click the upload button. And it will show the DBFS target directory here. You can click the upload option and choose your data file to upload.



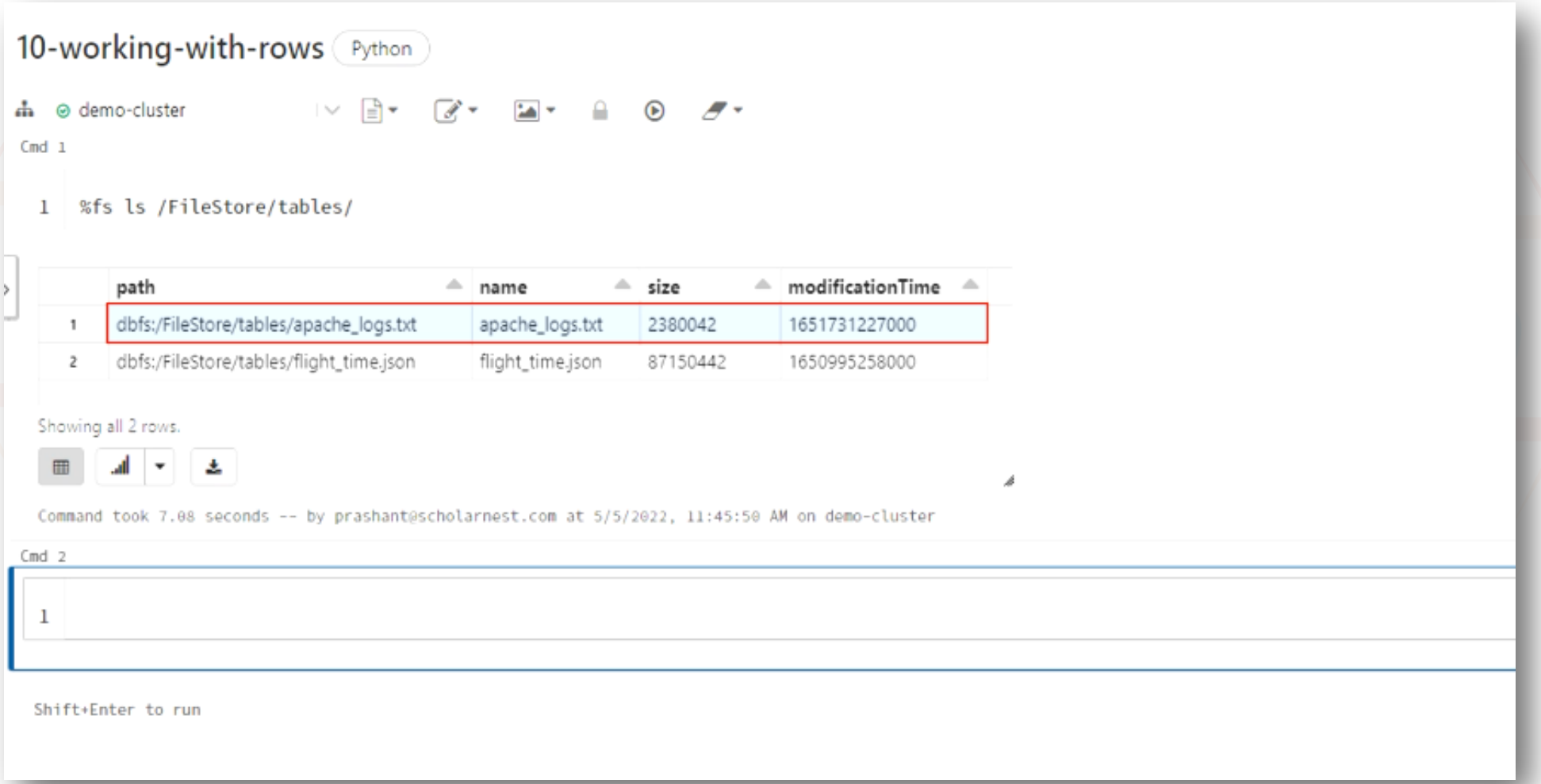
My data file is in the cloud, and we are now ready to start processing it.



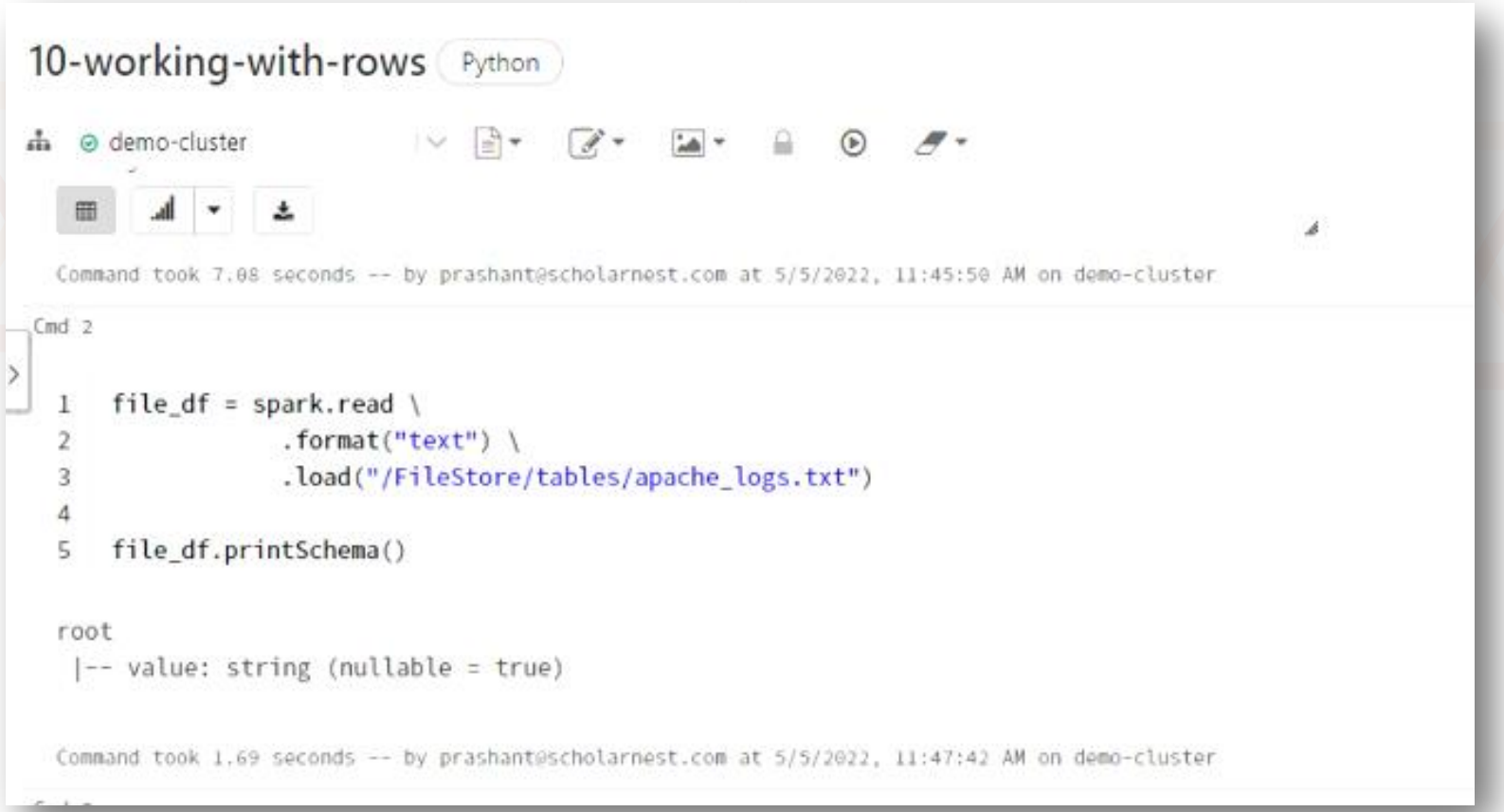
Create a new notebook.**(Reference : 10-working-with-rows)**
Make sure you attach a running cluster to your notebook.



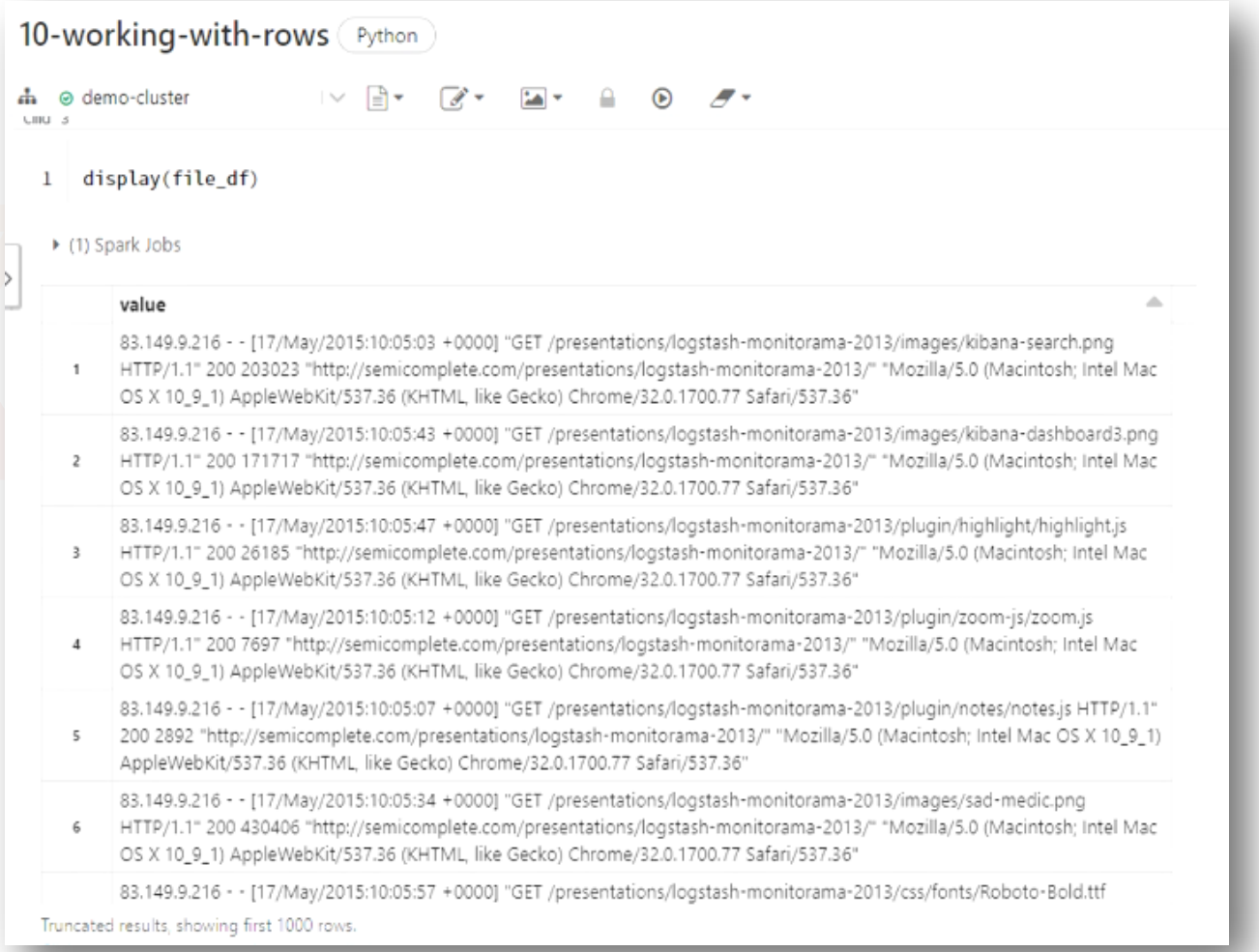
We have already uploaded the log file to the cloud storage. You can use the command shown below to see the file.
The next step is to load this file into a Spark DataFrame.



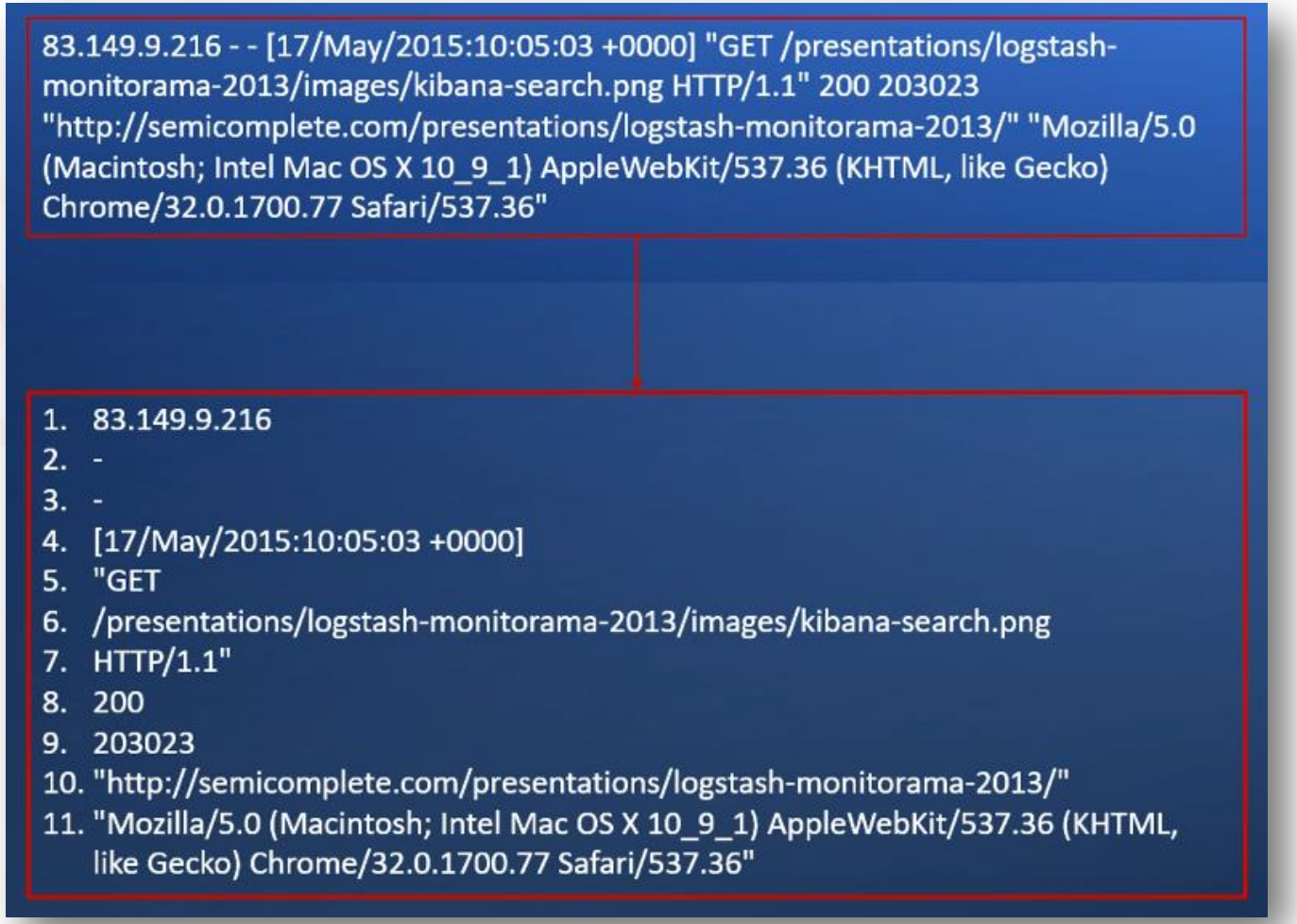
Here is the code to load the file into Spark Dataframe.
We start with the spark session variable, use the read attribute to get the DataFrameReader, set the format, and load it. The format is text. Then, I am also printing the schema of the data frame.
And you can see the output for this code below the cell. Dataframe must have a schema. You cannot create a schema-less Dataframe. We are loading a text file which is an unstructured data file. However, the Dataframe must have a schema even if you load an unstructured data file.



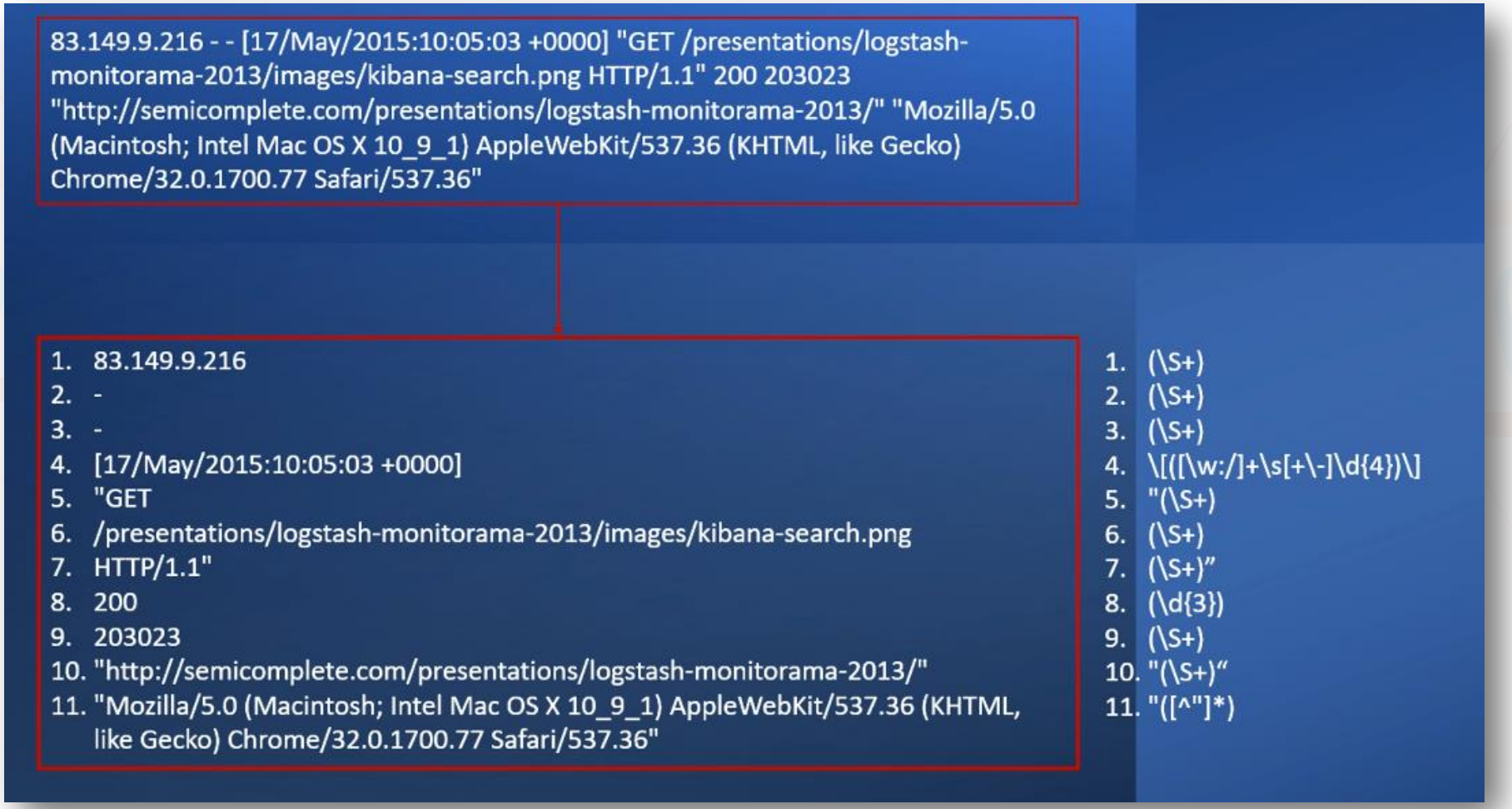
Here is the Dataframe. We have a single-column data frame. The column name is value.
Each record comes with one line of text, and it is an unstructured Dataframe. Now we want to do some analysis on this data and apply Dataframe transformations. However, most of the Dataframe transformations are applied to columns. And we do not have any column except the value column, because we have unstructured data.



I gave some structure to the data, by taking one record, and breaking it with spaces.
So I identified 11 fields in this data. The first one is the IP address. The second and third ones are hyphens. The fourth one is a date. Then we have Get or Post method. The next one is an absolute file name. Then we have protocol followed by two numbers. After all this, we have the referrer URL I am looking for. Once I get a URL, I do not care what else is there, so consider it one field.

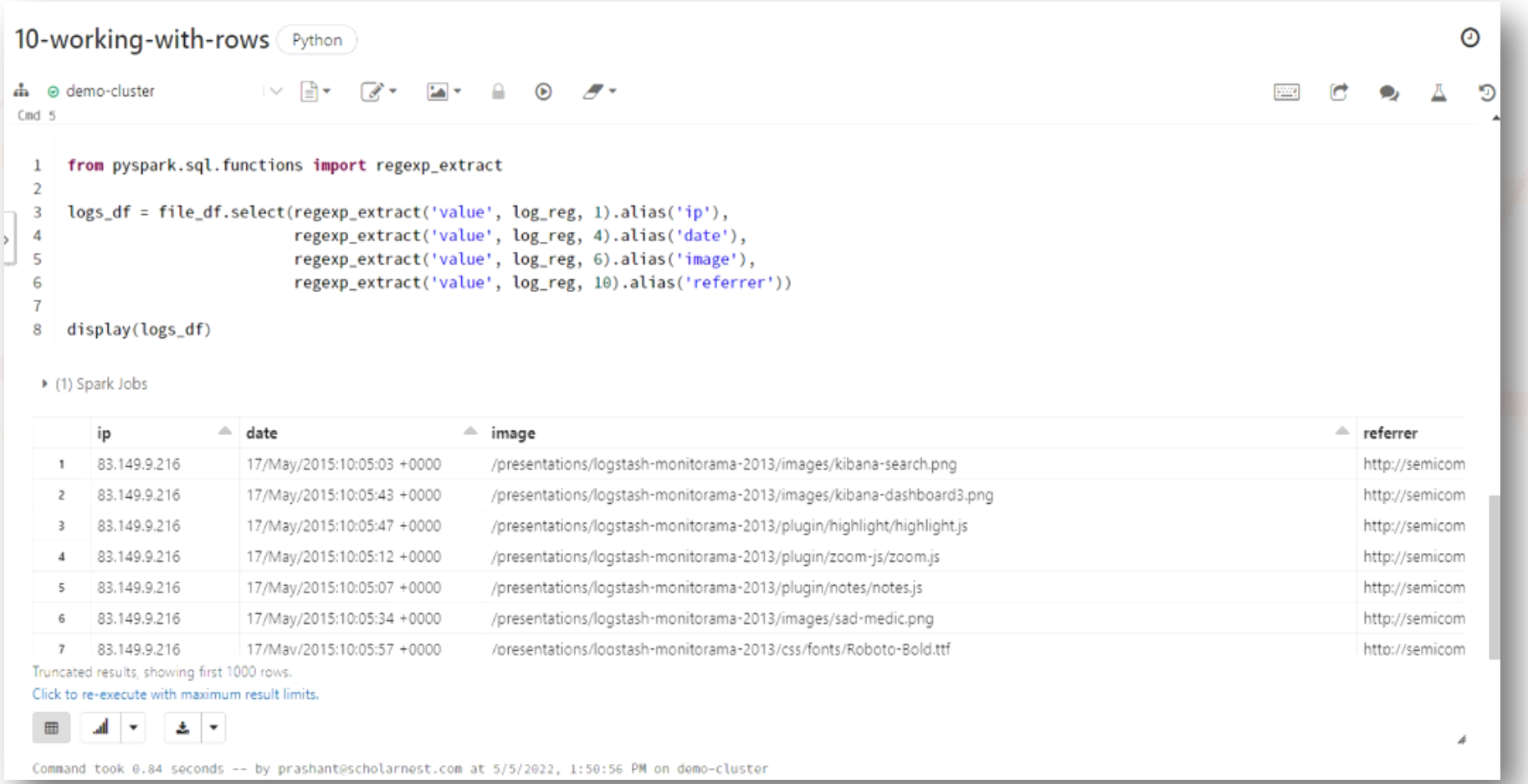


I saw the data and realized that the referrer's URL is in the file. However, we have another nine logical fields before the referrer. So If I can break this text line into eleven fields as listed here, I can put a structure to the given unstructured Dataframe. It is not easy to split this text using space characters. I thought of applying regular expressions. And here are eleven expressions assigned for each field.



Here is the regular expression that represents all the eleven fields.

Spark gives us regexp_extract() function to extract one field from a regular expression, and I have the code for that shown below. So I am using the select() method to take out four fields.
The regexp_extract() method takes three arguments. The first one is the column on which you want to apply the regular expression. The second argument is the regular expression itself.
The last argument is the position after splitting the value using the regular expression.



I know the first position is for IP address, the fourth is a date, the sixth is the image, and the tenth field is the referrer. I am only interested in a referrer. But I tried to extract four fields so you can learn how to extract multiple files.

10-working-with-rows Python

demo-cluster

Cmd 5

```
1  from pyspark.sql.functions import regexp_extract
2
3  logs_df = file_df.select(regexp_extract('value', log_reg, 1).alias('ip'),
4                           regexp_extract('value', log_reg, 4).alias('date'),
5                           regexp_extract('value', log_reg, 6).alias('image'),
6                           regexp_extract('value', log_reg, 10).alias('referrer'))
7
8  display(logs_df)
```

(1) Spark Jobs

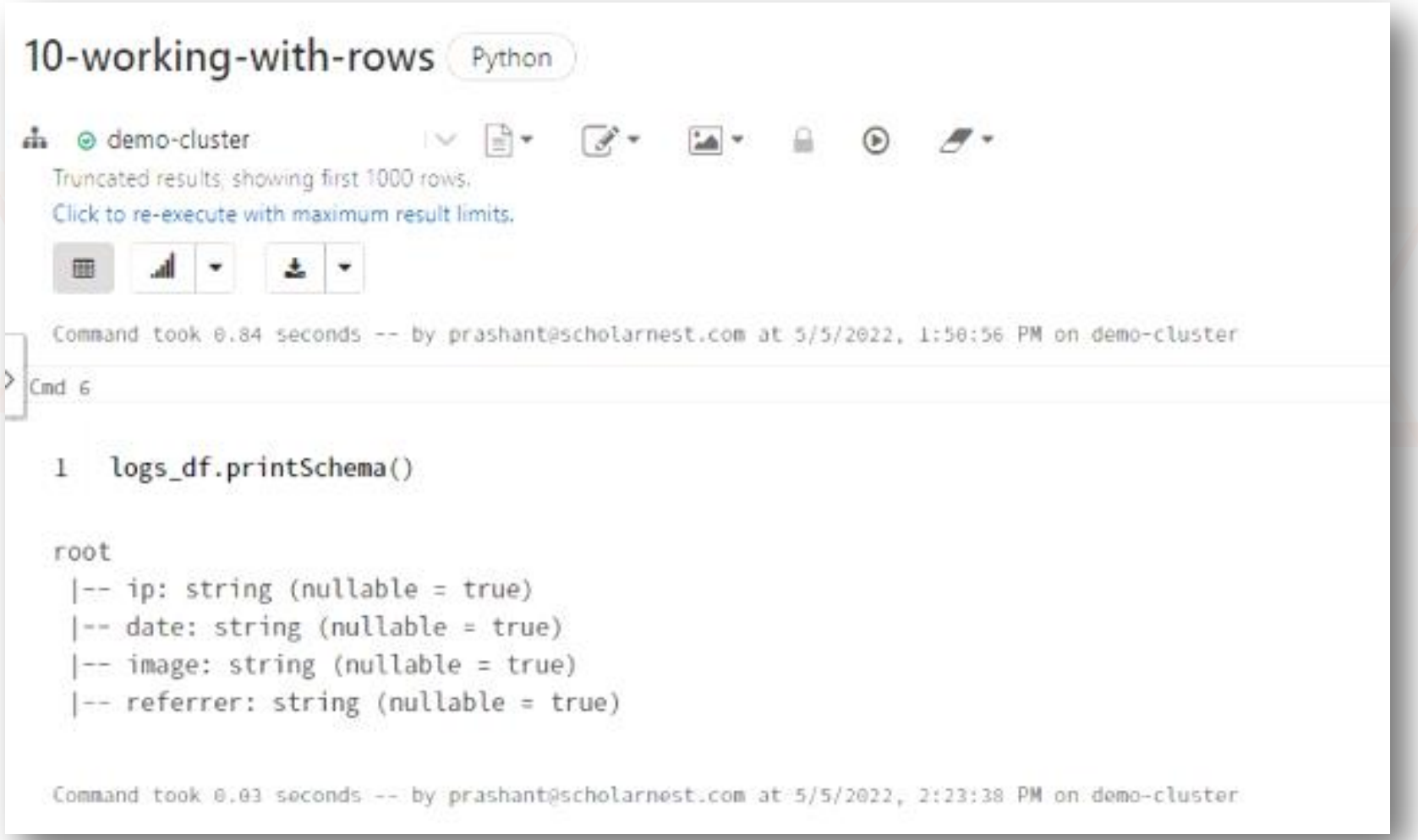
	ip	date	image	referrer
1	83.149.9.216	17/May/2015:10:05:03 +0000	/presentations/logstash-monitorama-2013/images/kibana-search.png	http://semicom
2	83.149.9.216	17/May/2015:10:05:43 +0000	/presentations/logstash-monitorama-2013/images/kibana-dashboard3.png	http://semicom
3	83.149.9.216	17/May/2015:10:05:47 +0000	/presentations/logstash-monitorama-2013/plugin/highlight/highlight.js	http://semicom
4	83.149.9.216	17/May/2015:10:05:12 +0000	/presentations/logstash-monitorama-2013/plugin/zoom-js/zoom.js	http://semicom
5	83.149.9.216	17/May/2015:10:05:07 +0000	/presentations/logstash-monitorama-2013/plugin/notes/notes.js	http://semicom
6	83.149.9.216	17/May/2015:10:05:34 +0000	/presentations/logstash-monitorama-2013/images/sad-medic.png	http://semicom
7	83.149.9.216	17/May/2015:10:05:57 +0000	/presentations/logstash-monitorama-2013/css/fonts/Roboto-Bold.ttf	http://semicom

Truncated results, showing first 1000 rows.

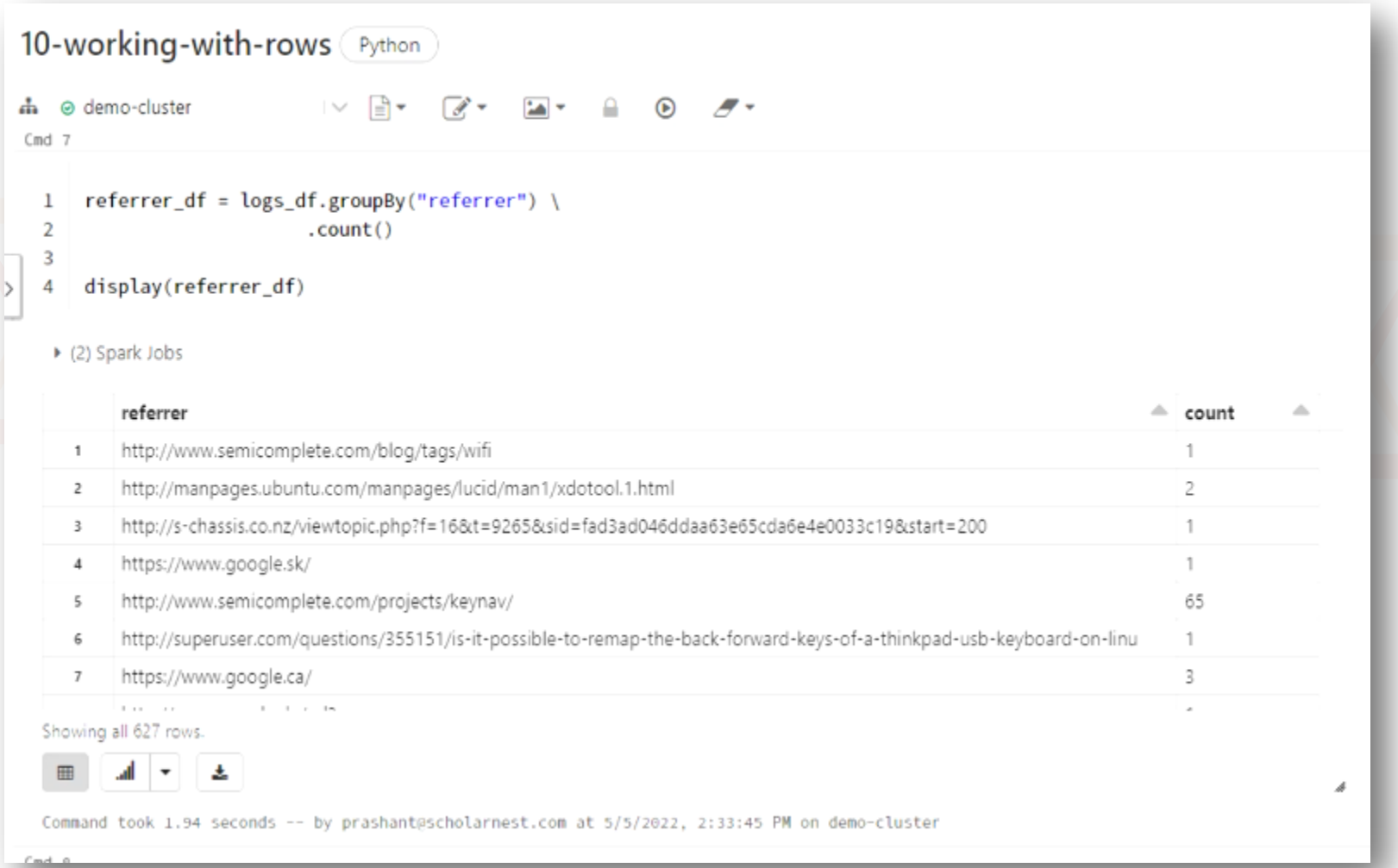
Click to re-execute with maximum result limits.

Command took 0.84 seconds -- by prashant@scholarnest.com at 5/5/2022, 1:50:56 PM on demo-cluster

Here is the schema definition for the Dataframe. Now I can apply some column transformations.

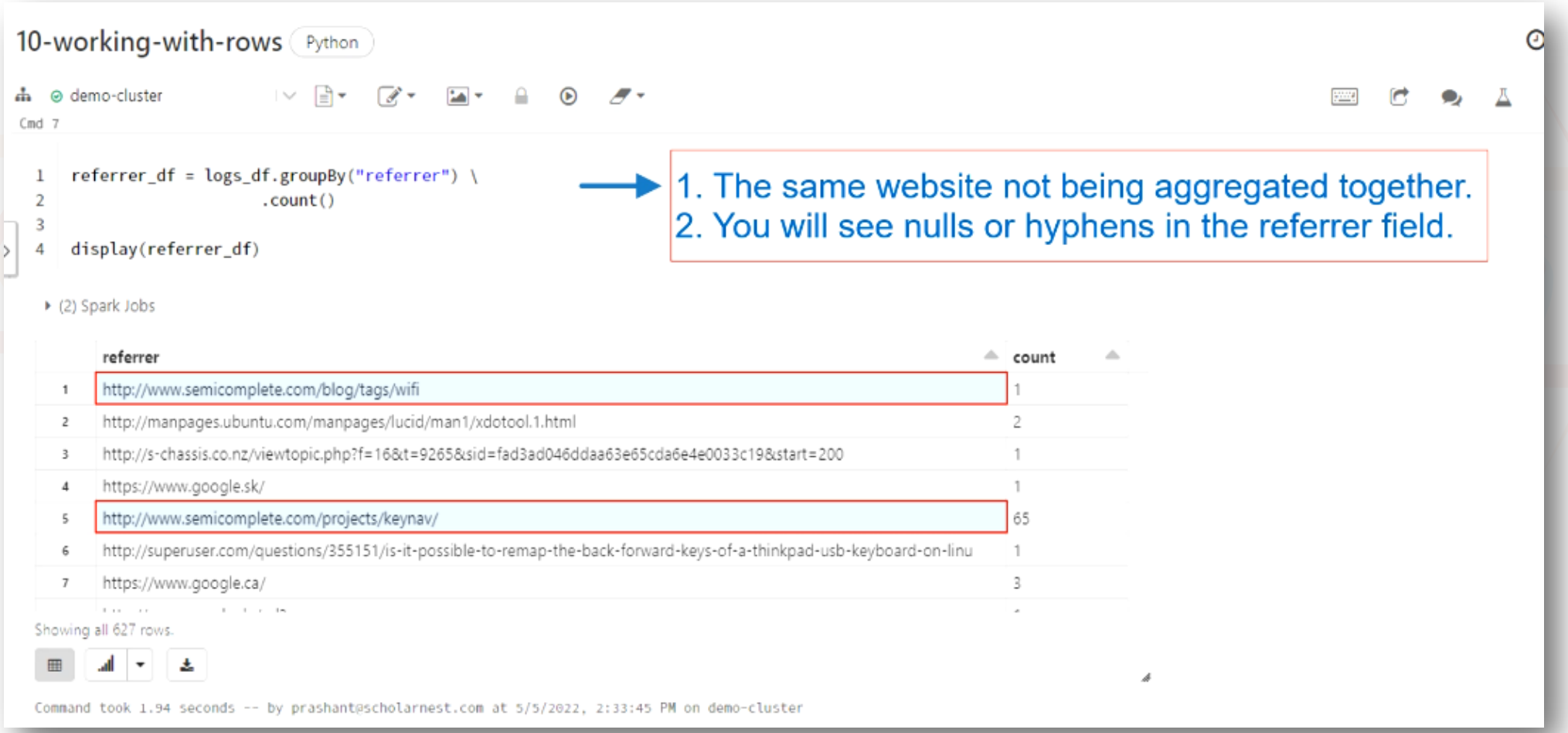


Now I have a structured Dataframe. I can easily group by the referrer, take a count, and display it.
You can see the results. And we now know who referred how many clicks.

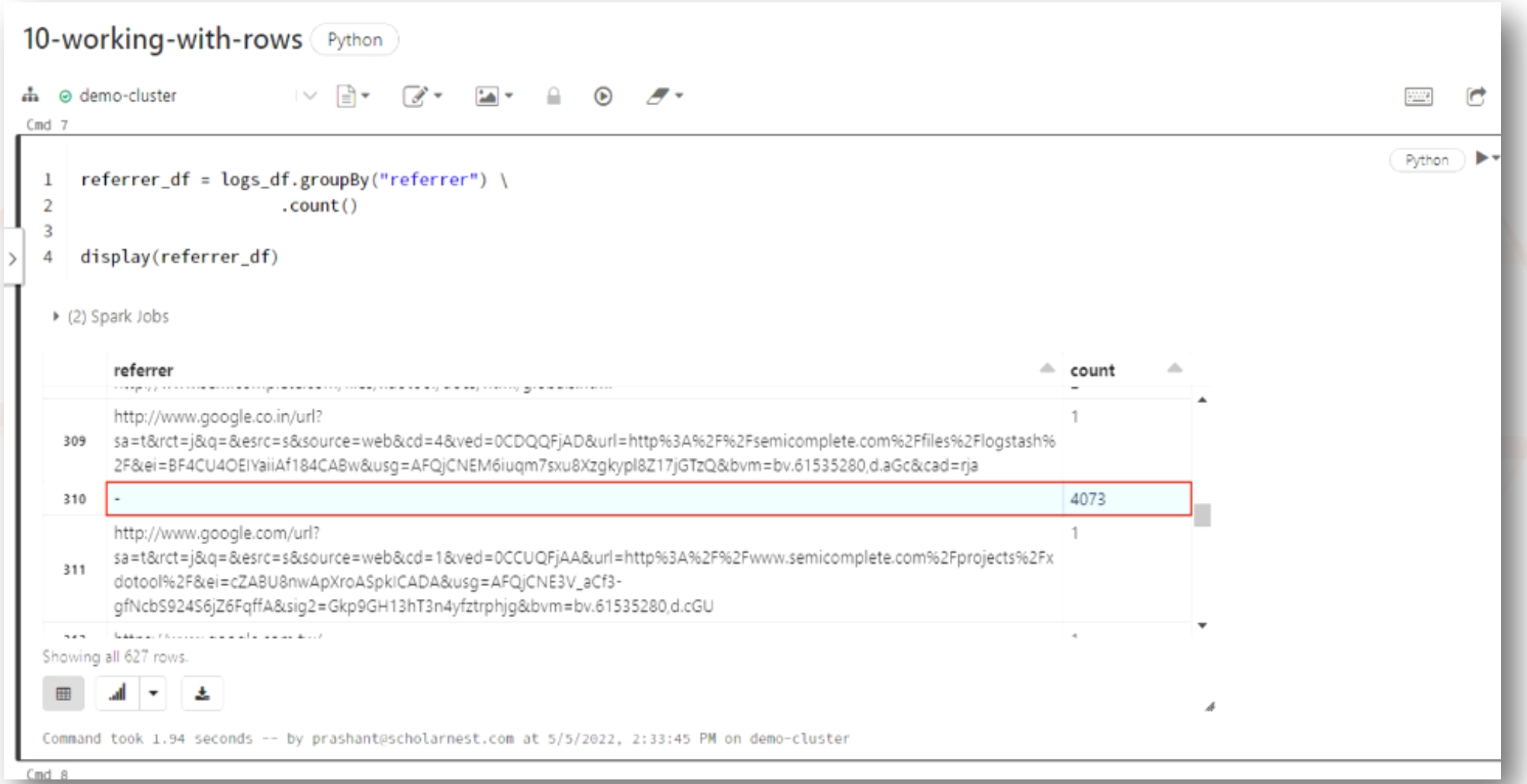


We have two problems in this Dataframe as listed below.

1. Look at the results, and you will see the same website not being aggregated together. I mean, these two URLs are the same websites. However, they are aggregating as two different websites.



2. I have another problem with this data. Scroll down, and you will see nulls or hyphens in the referrer field. Here are 4K plus hits that are not coming from any referrer.



Can you solve these problems listed below and give us the top five referrers? I leave it as an exercise for you.
The solution is included in the notebook source code. So you can refer to it.

1. The same website not being aggregated together.
2. You will see nulls or hyphens in the referrer field.
3. Display the top five referrers.

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

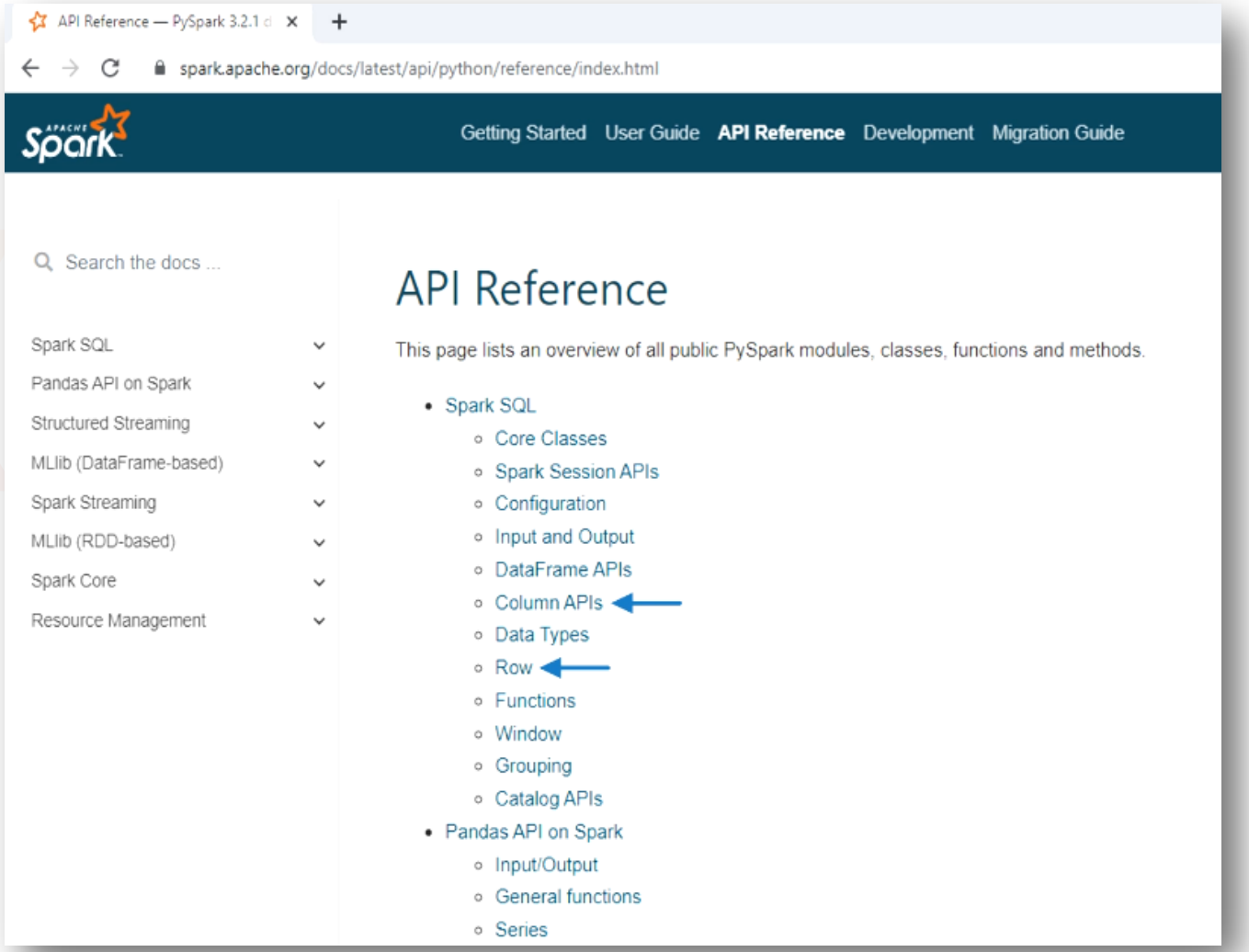
Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

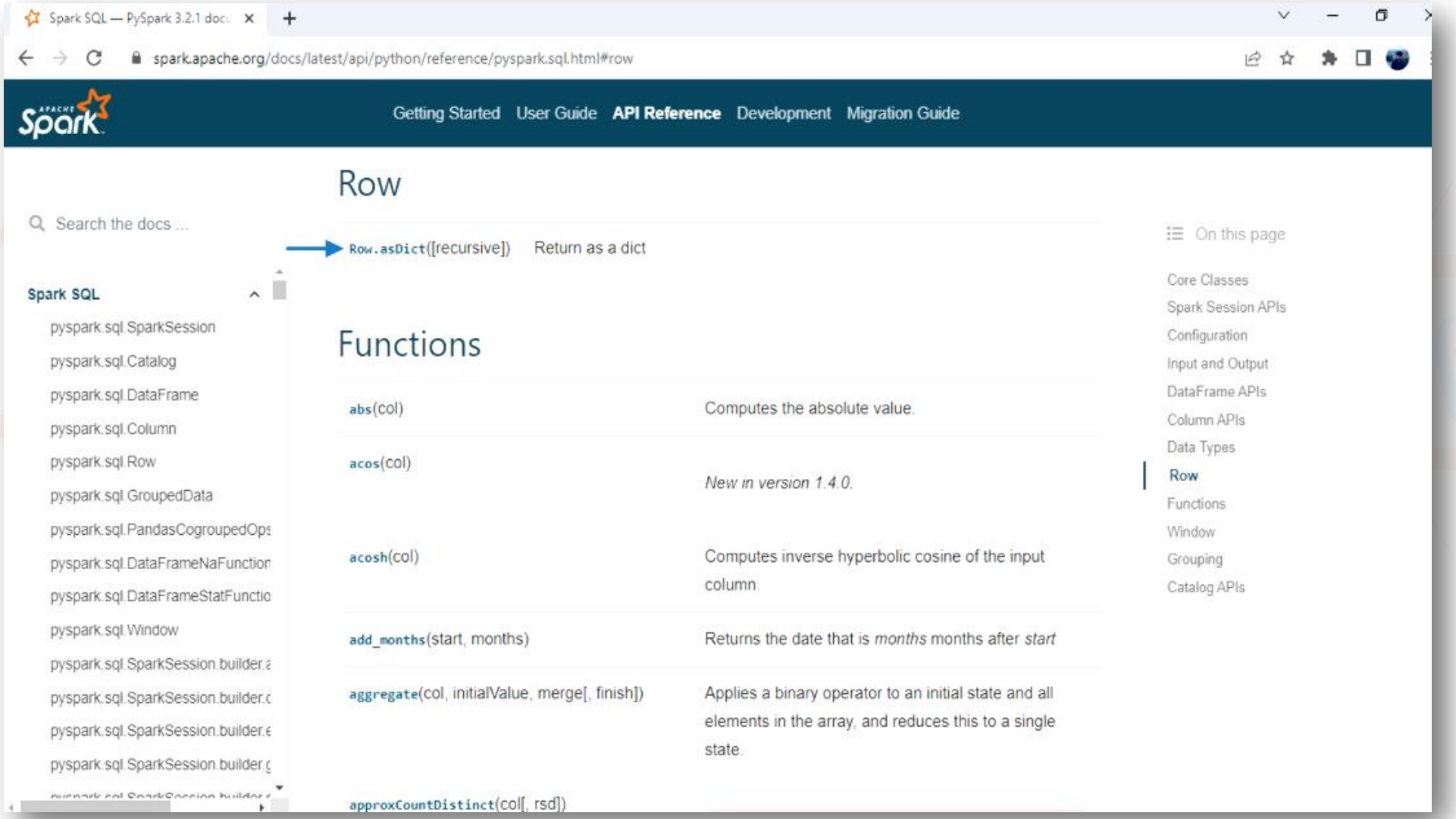
Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Spark Data Frame Columns

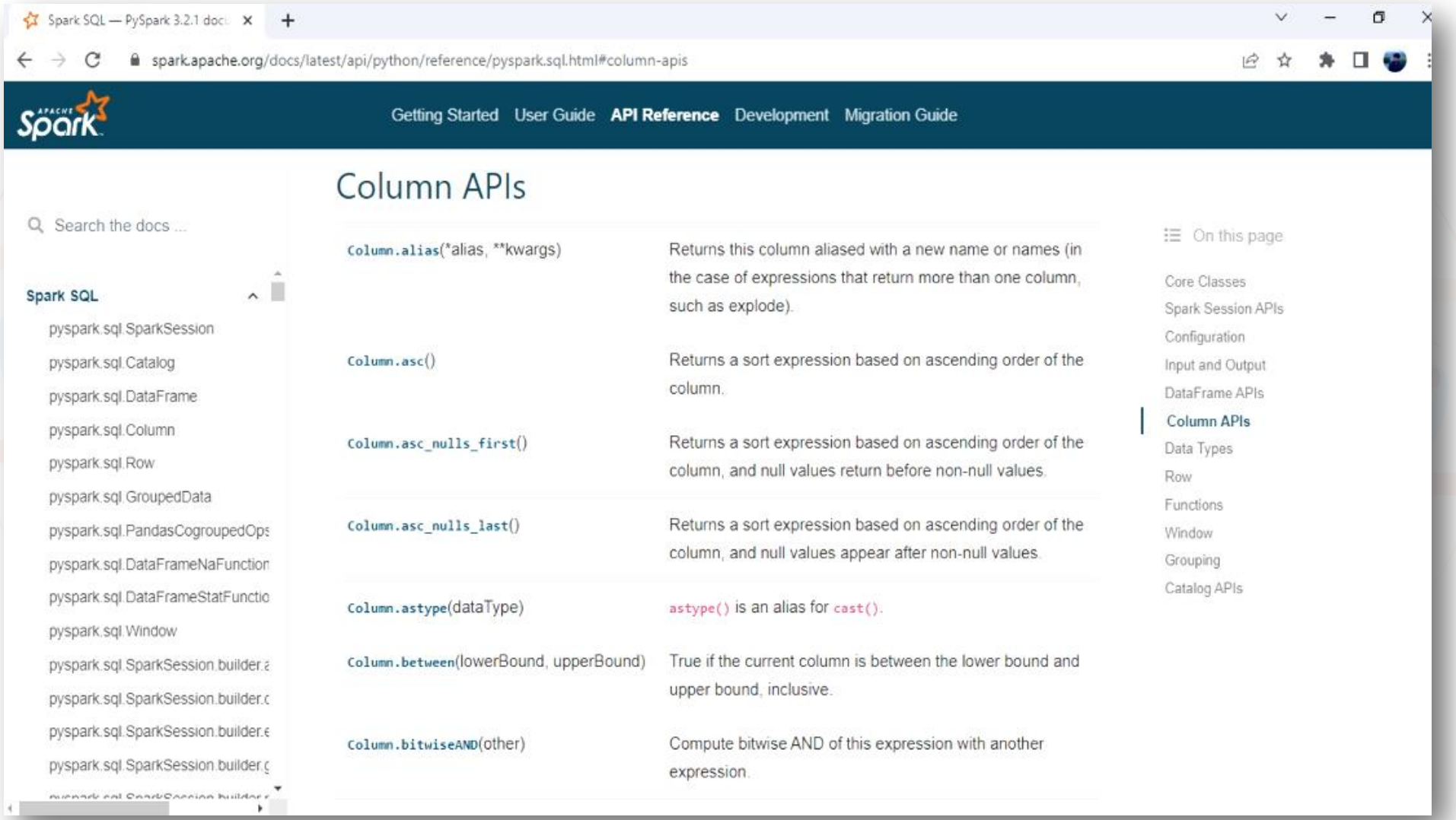
Spark also defined two objects to represent the Dataframe Row and column. Go to the latest Spark documentation - API reference page. **(Reference - https://spark.apache.org/docs/latest/api/python/reference/index.html)**
You will see two links: Column APIs and Row



If you go to the link for row. The row is a simple class, and it gives you only one method: asDict().
The asDict() method converts a Row into a Python dictionary. You are not likely to directly work with the Row objects. We use the Row objects in writing unit test cases for the Spark application.



Spark Dataframe is made up of Row and Column objects. The row object is rudimentary, and we only use it in unit testing. Now let us go to the link for column API, and you will see the that the column object has got many methods.



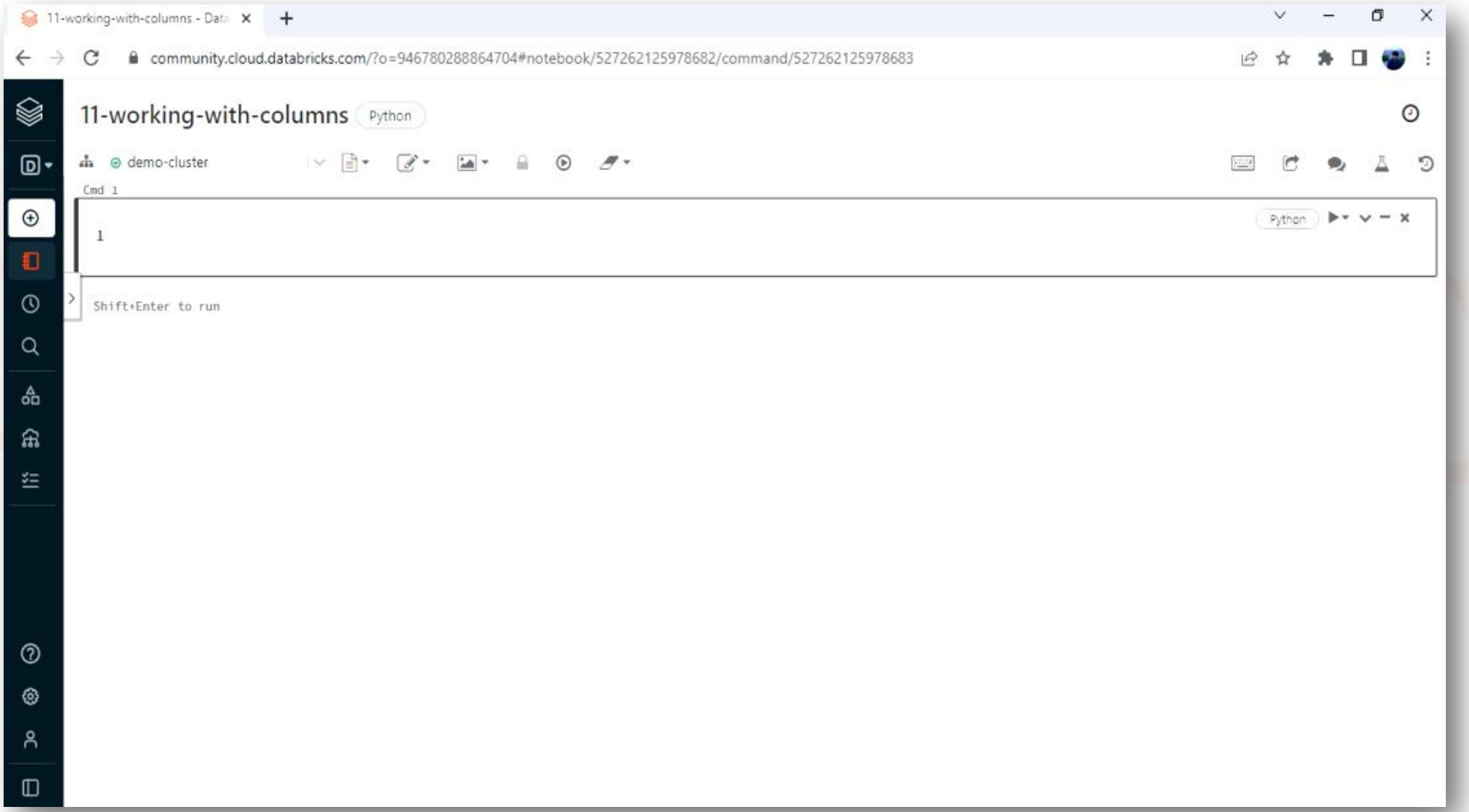
Spark Dataframe is made up of Rows and Column objects. We do not have any row transformation in spark.
And most of the Dataframe transformations are all about transforming the columns.

Column

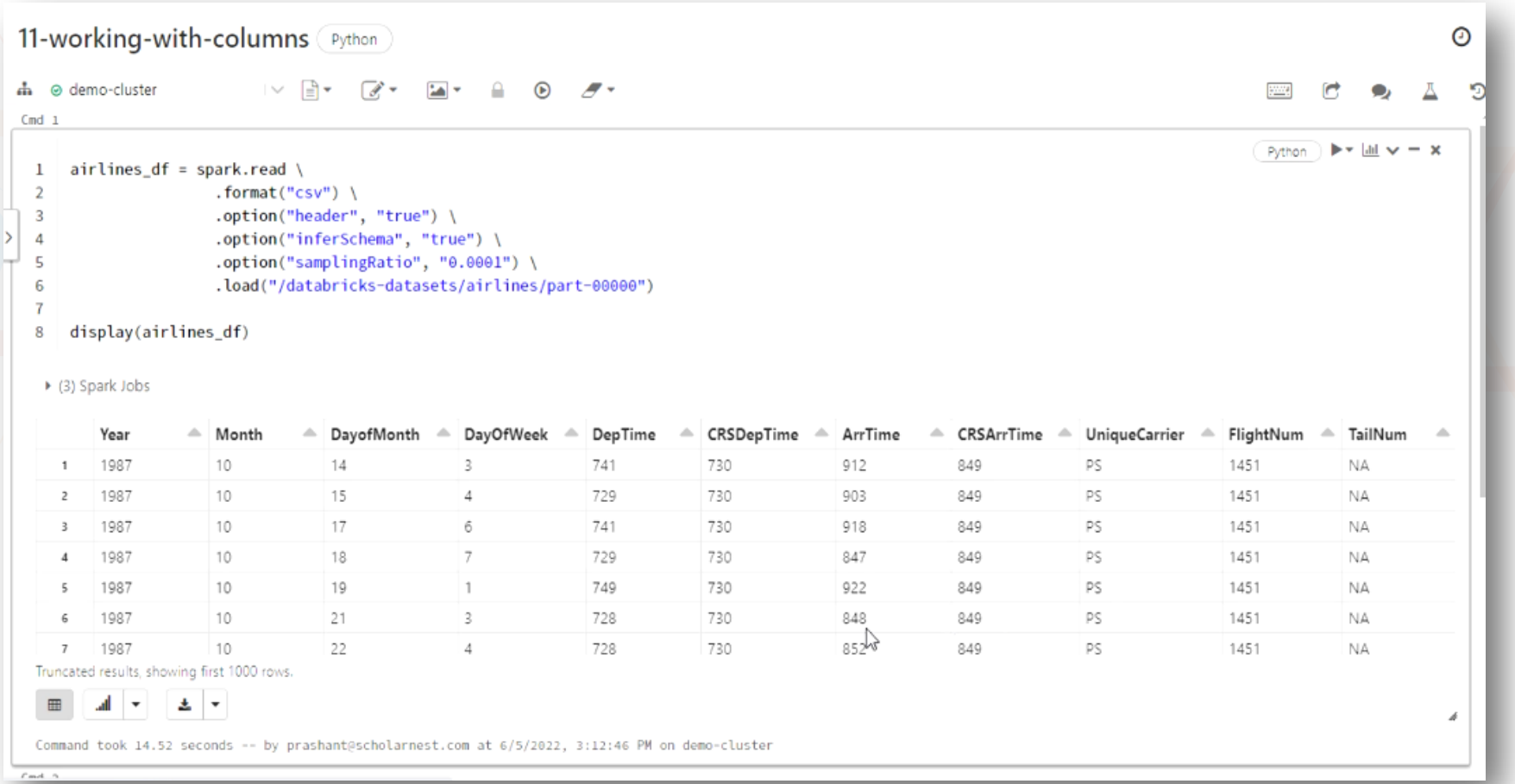
Row

Timestamp	Age	Gender	Country	state
2014-08-27 11:29:31	37	Female	United States	IL
2014-08-27 11:29:37	44	M	United States	IN
2014-08-27 11:29:44	32	Male	Canada	
2014-08-27 11:29:46	31	Male	United Kingdom	
2014-08-27 11:30:22	31	Male	United States	TX

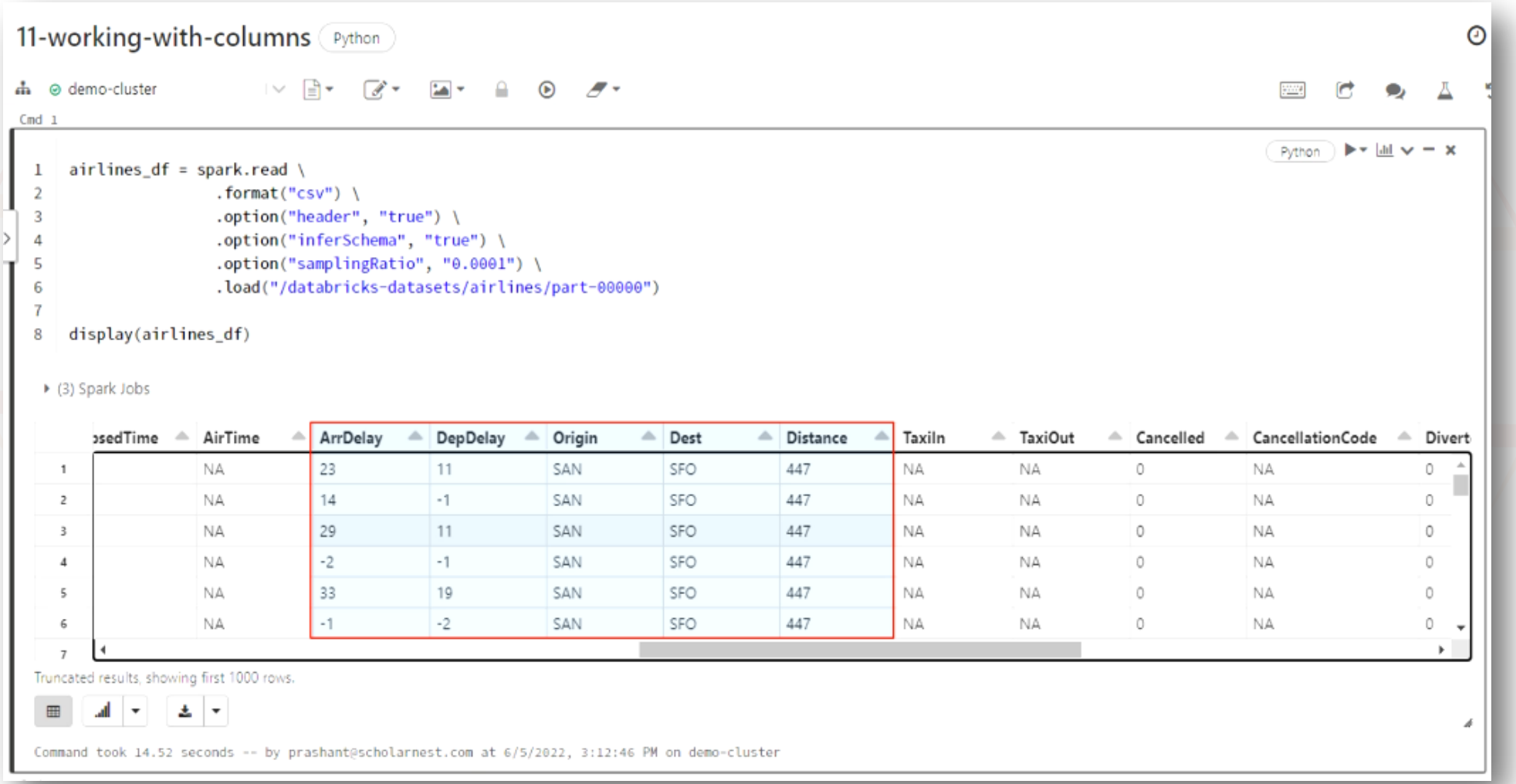
Log in to your Databricks community cloud and create a new notebook.**(Reference : 11-working-with-columns)**



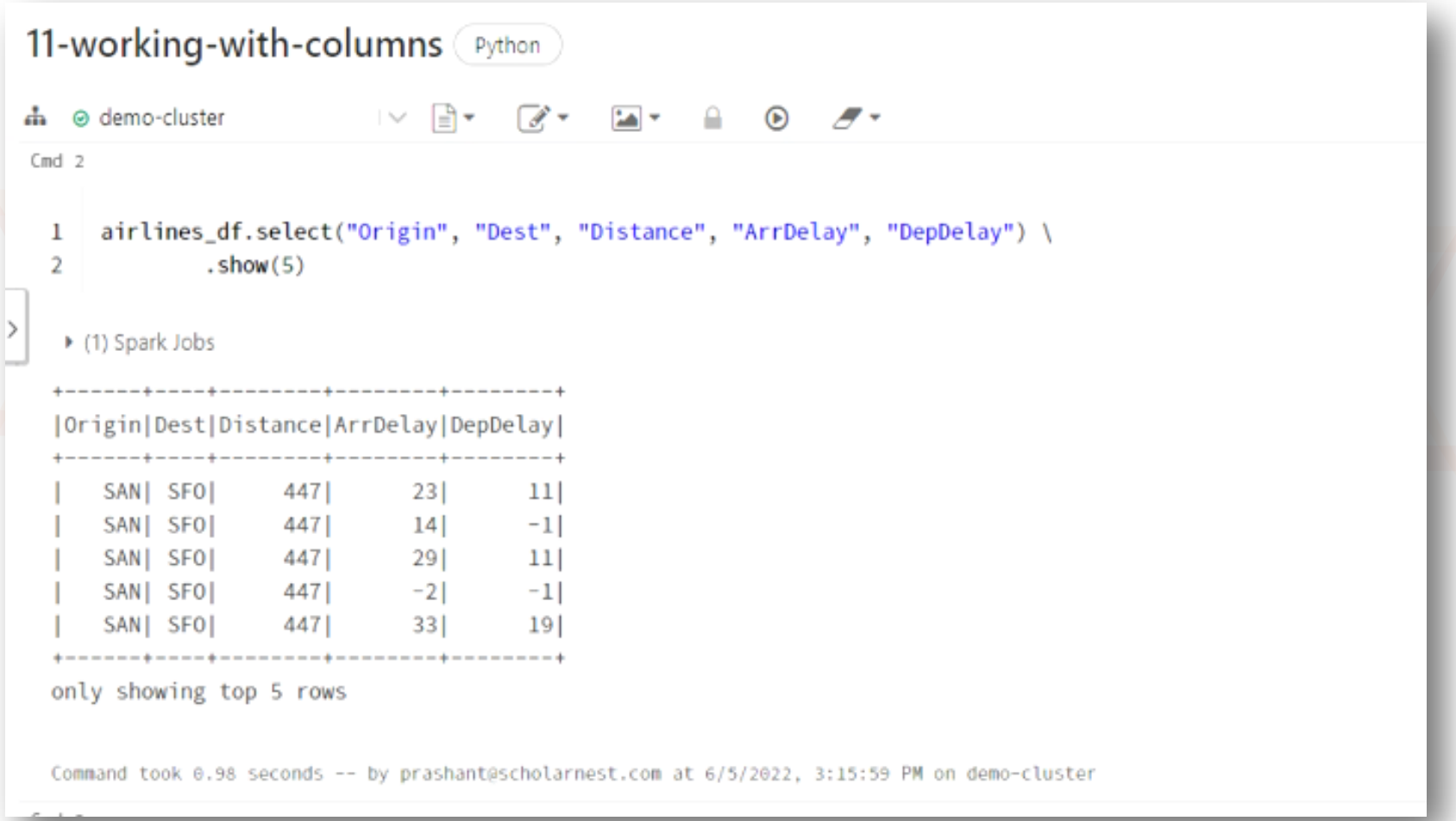
I have created a Dataframe.
I am reading data using the spark.read. The format is CSV, and I am setting three options: Header, Infer Schema, and Sampling Ratio. Finally, I am loading the data into a Dataframe.
Infer schema option will read a sample of the data file and try to infer the schema automatically. The default value of the sample size is one percent. You can increase or decrease the sampling ratio. I am setting it to a smaller value to ensure that this command runs quickly.



So we have an origin, destination, and distance columns.
We also have Arrival delays and departure delays before the origin column.
I want to select only these five columns.



Here is the code to select the desired 5 columns. And we can see the output for the same given below.



We have five approaches to referring to a column name.

1. "Origin"
2. expr("Origin")
3. airlines_df.Origin
4. column("Origin")
5. col("Origin")

The first approach is a column name string. The second approach uses the expr() function and the column name. The third one is using a dataframe.column name. The fourth approach is using a column() function. The last one is using the col() function.

The fourth and the fifth approaches are the same because the col() function is an alias for the column() function. So the fourth and the fifth approaches are the same.

But officially, we have five different approaches to refer to a column name. You can use any of these approaches in your select expression.

Here is the code to select the desired 5 columns. But I am referring to each column differently. I am referring to the Origin column using a plain string name. But I am referring to the destination using the expr() function. Similarly, I am referring to the distance using the dataframe.column name. The next one is using the column() function. And the last one is using the col() function.

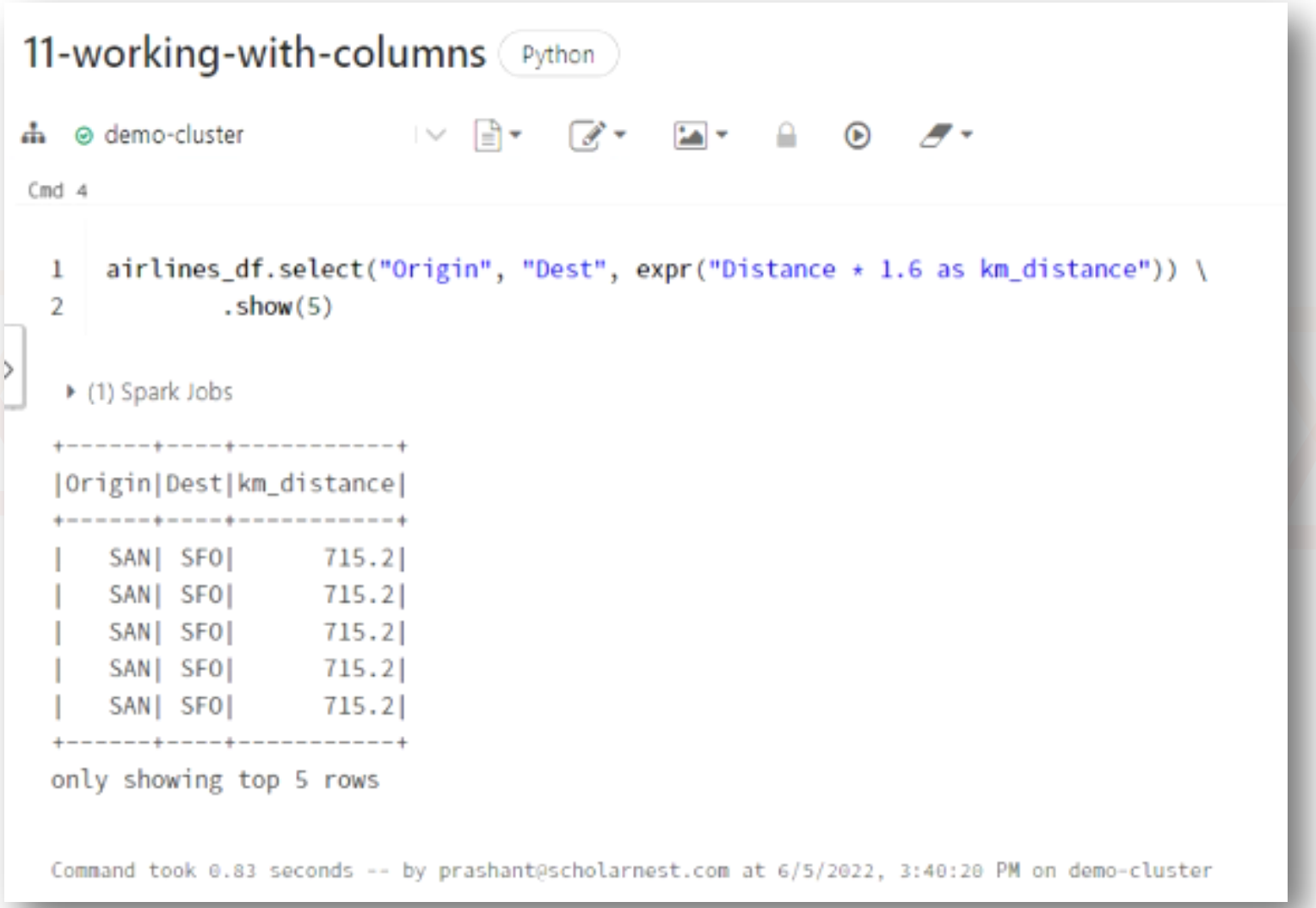
1. "Origin"
2. expr("Origin")
3. airlines_df.Origin
4. column("Origin")
5. col("Origin")

The first approach is for simplicity. I prefer using the first approach and simply writing the column name string. It is easy to use and looks neat.

But the first approach doesn't allow writing expressions.
And that's why we have the second approach of using the expr() function.

I am selecting three columns here in the code shown below. The first two columns are Origin and Destination. These are simple names, so they work. But the third column is a SQL expression. I am multiplying distance by 1.6 to convert it from miles to kilometers. So the third column is not a simple column name, it is an expression. You will see an error if you run it. And that's why we have the expr() function.

It works with expr() function.



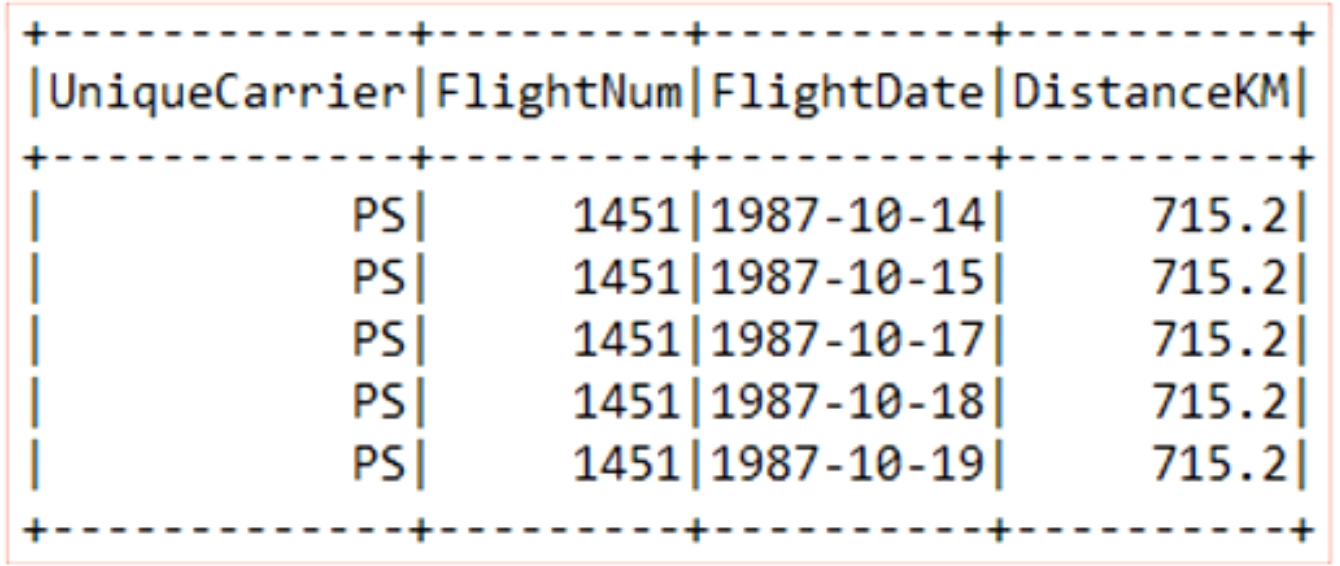
So we learned the need for the first two approaches.

1. "Origin" -> Used for plain column names and simplicity
2. expr("Distance * 1.6 as km_distance") -> Used to evaluate SQL like expression strings

The first one is a simple column name, and we use it for simplicity.
But when you have an expression, you must evaluate it using the expr() function.
So the second approach is for evaluating SQL-like expression strings.
The expr() function also works fine for simple column names. But we do not use it for column names.
We use the expr() function when we have an expression.

We have the following six fields in this given Dataframe as shown in the right hand side of the screenshot. I want to write a select expression to produce the following output shown in the left side of the screenshot.

1. UniqueCarrier
2. FlightNum
3. Year
4. Month
5. DayofMonth
6. Distance



UniqueCarrier	FlightNum	FlightDate	DistanceKM
PS	1451	1987-10-14	715.2
PS	1451	1987-10-15	715.2
PS	1451	1987-10-17	715.2
PS	1451	1987-10-18	715.2
PS	1451	1987-10-19	715.2

Here is one possible solution. The first and the second columns are simply the names. So I am not using the expr() function for those column names. But the third and the fourth columns are expressions. So I am using the expr() function.

11-working-columns Python

demo-cluster

Cmd 5

```
1  airlines_df.select("Origin",
2                     "FlightNum",
3                     expr("Year||'-'||Month||'-'||DayOfMonth as FlightDate"),
4                     expr("Distance * 1.6 as DistanceKM")) \
5          .show(5)
```

(1) Spark Jobs

```
+------+---------+----------+----------+
|Origin|FlightNum|FlightDate|DistanceKM|
+------+---------+----------+----------+
|   SAN|     1451|1987-10-14|     715.2|
|   SAN|     1451|1987-10-15|     715.2|
|   SAN|     1451|1987-10-17|     715.2|
|   SAN|     1451|1987-10-18|     715.2|
|   SAN|     1451|1987-10-19|     715.2|
+------+---------+----------+----------+
only showing top 5 rows
```

Command took 0.83 seconds -- by prashant@scholarnest.com at 6/5/2022, 3:43:55 PM on demo-cluster

The third approach is using the dataframe.column name.
We use the third approach to avoid column name ambiguity. We have another Dataframe airport_df shown below. The airlines_df is the main dataframe, and the airport_df is a lookup data frame. And I want to produce results like the one shown in the screenshot below.

11-working-with-columns Python

demo-cluster

Command took 0.83 seconds -- by prashant@scholarnest.com at 6/5/2022, 3:43:55 PM on demo-cluster

Cmd 6

```
1  data_list = [("SAN", "San Antonio"),
2               ("SFO", "San Francisco"),
3               ("BUR", "Hollywood Burbank"),
4               ("OAK", "Oakland")]
5
6  airport_df = spark.createDataFrame(data_list).toDF("Origin", "OriginAirportName")
7  display(airport_df)
```

(3) Spark Jobs

	Origin	OriginAirportName
1	SAN	San Antonio
2	SFO	San Francisco
3	BUR	Hollywood Burbank
4	OAK	Oakland

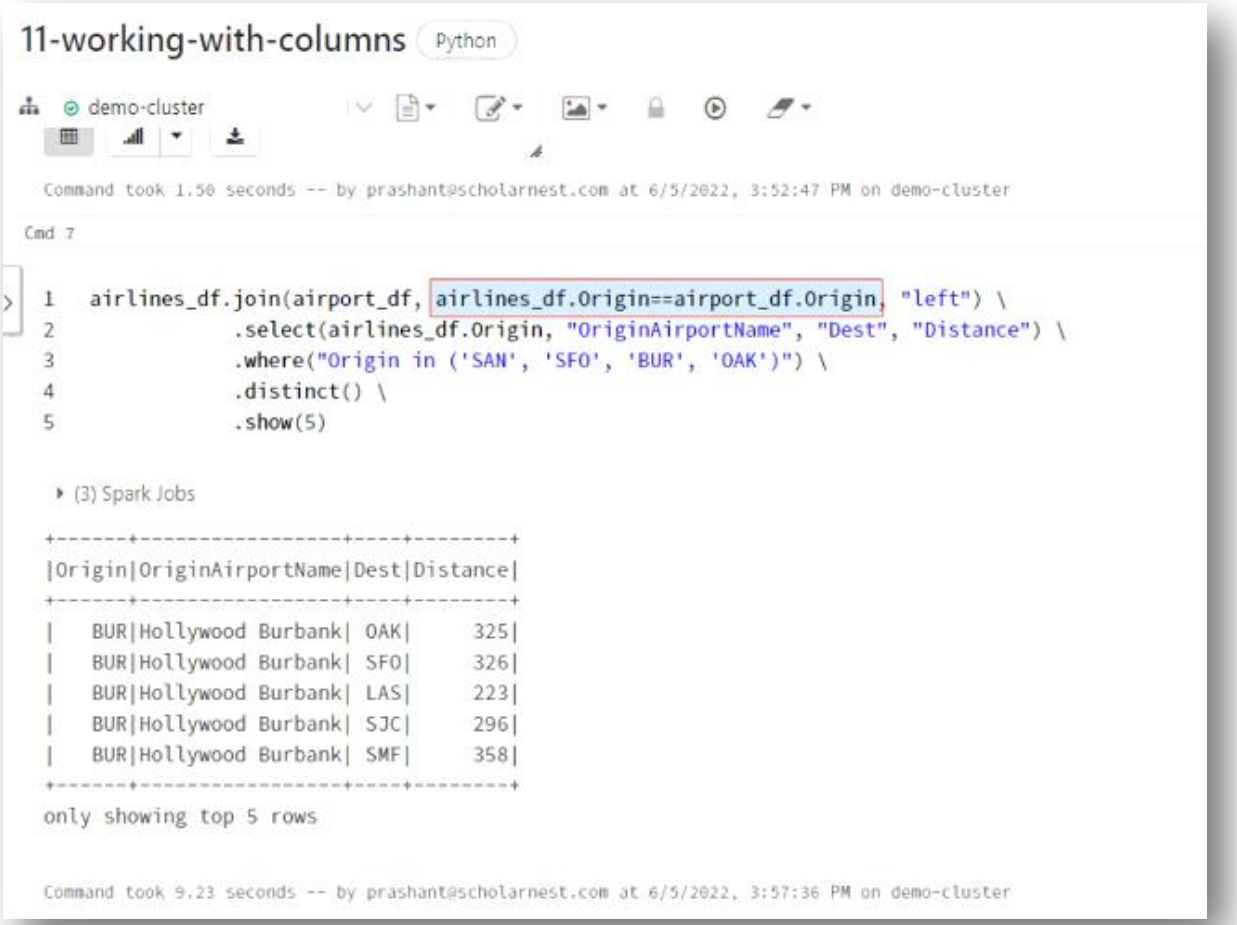
Showing all 4 rows.

```
+------+-----------------+----+--------+
|Origin|OriginAirportName|Dest|Distance|
+------+-----------------+----+--------+
|   BUR|Hollywood Burbank| OAK|     325|
|   BUR|Hollywood Burbank| SFO|     326|
|   BUR|Hollywood Burbank| LAS|     223|
|   BUR|Hollywood Burbank| SJC|     296|
|   BUR|Hollywood Burbank| SMF|     358|
+------+-----------------+----+--------+
```

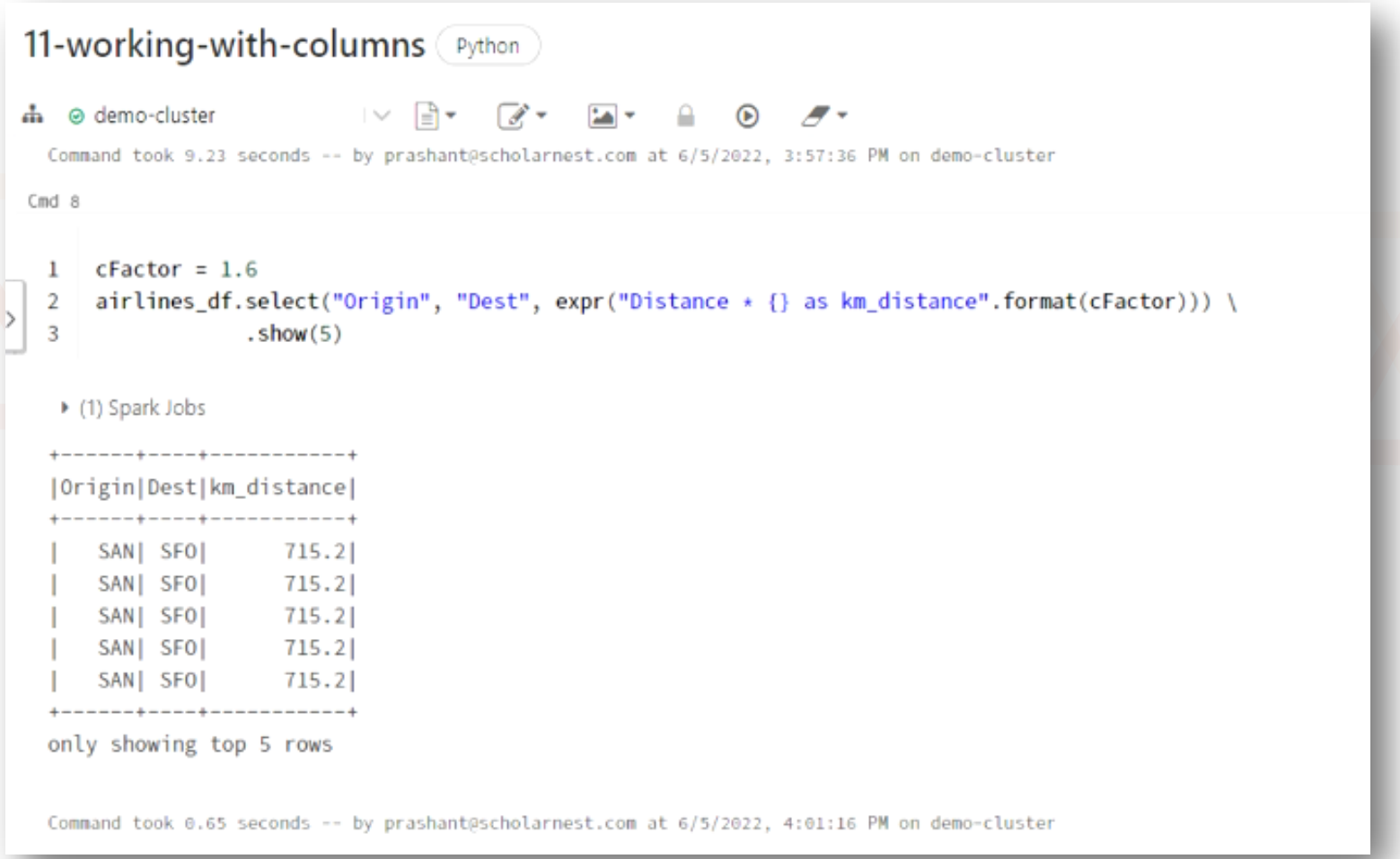
Command took 1.50 seconds -- by prashant@scholarnest.com at 6/5/2022, 3:52:47 PM on demo-cluster

Cmd 7

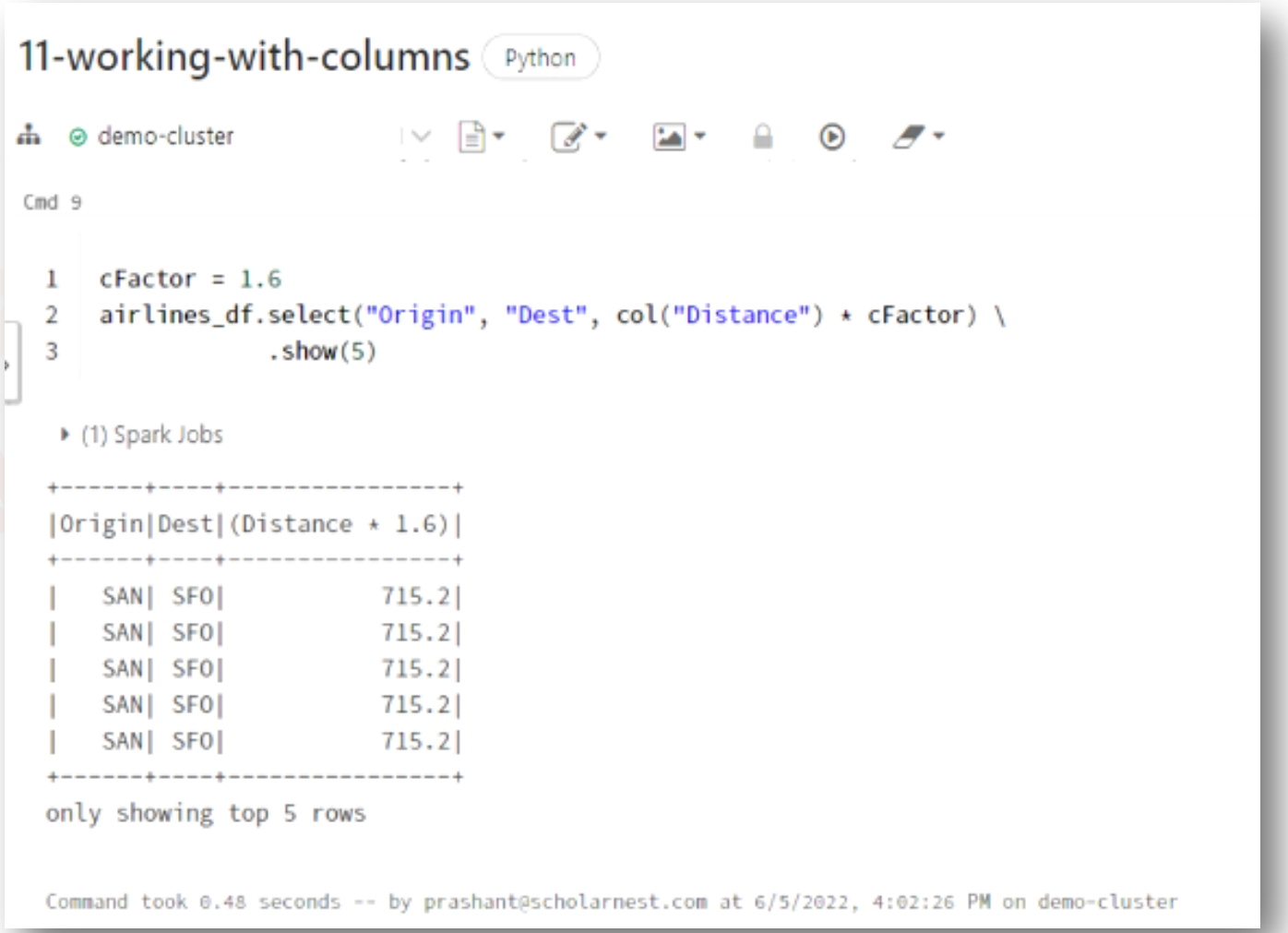
We can use the following code to achieve the desired result. We are doing a left join of airlines_df with the airport_df. Look at the join condition highlighted below.
We have a small problem with the airlines_df and the airport_df. They both have the same column name. The column name is the origin of both the data frames. And we cannot say Origin==Origin, Spark will complain about ambiguous column names. So we could use the dataframe.column name to avoid column name ambiguity.



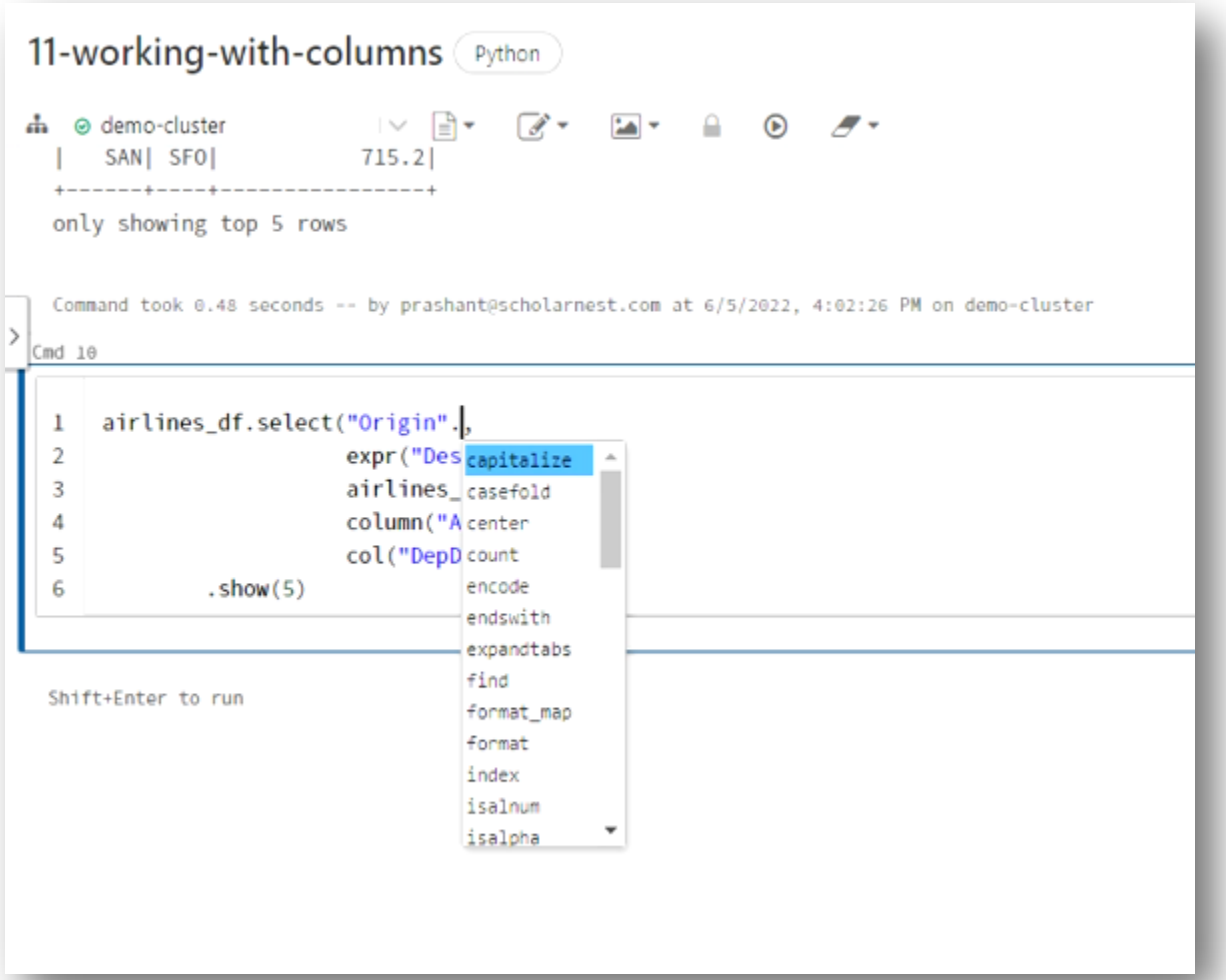
Here is an expression that we created earlier. I have a cFactor variable.
And I am using the variable inside the expression using Python string substitution technique.
But instead of using this string technique, some people prefer to use the col() function.



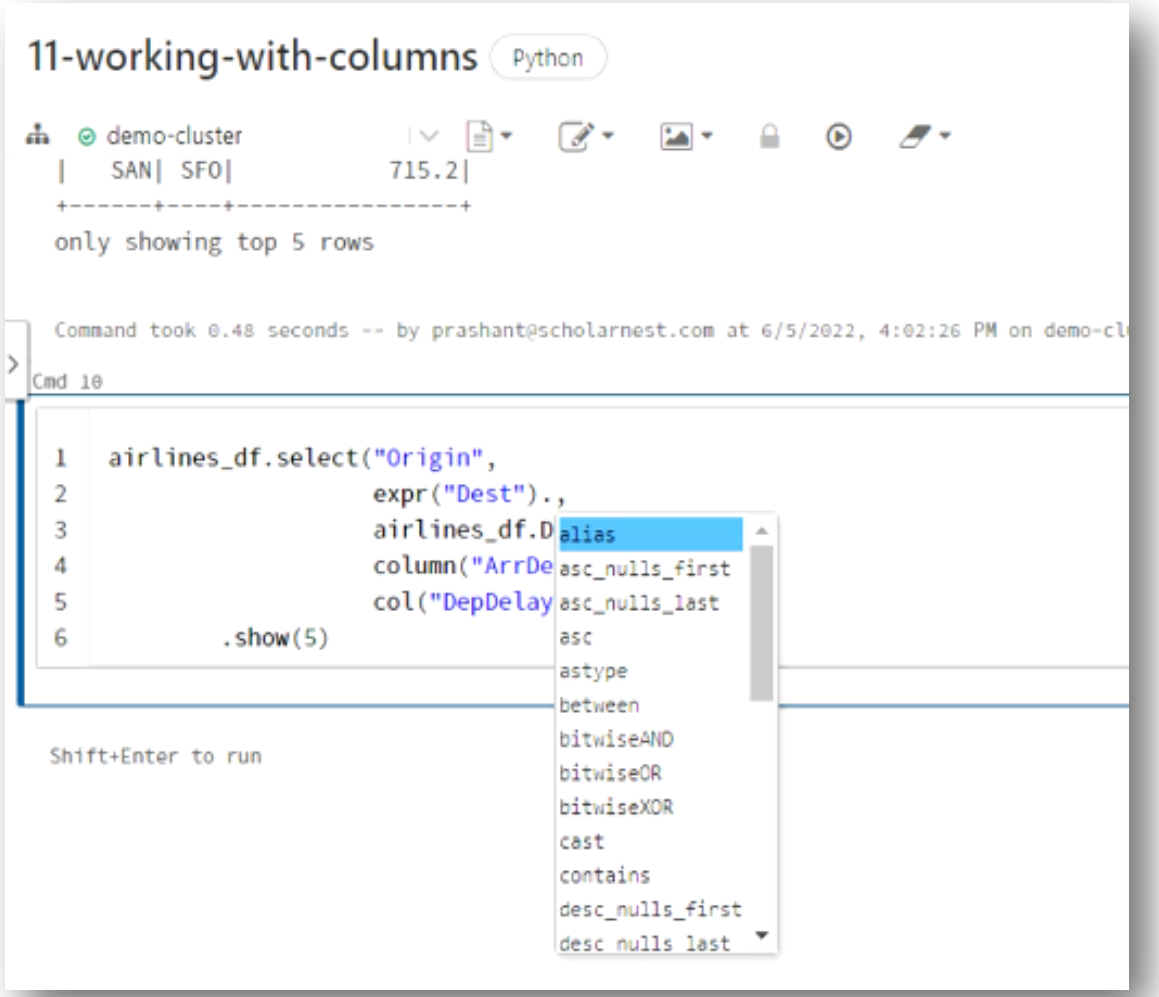
Here is the same code using the col() function.



Here is a Dataframe select statement. I am selecting five columns and using five different approaches to select the columns. The first one is a column name string. So place your cursor after the string name, type a dot symbol and press the tab. You will see a bunch of string functions such as capitalize, case fold, center, etc. These are all Python string manipulation functions.



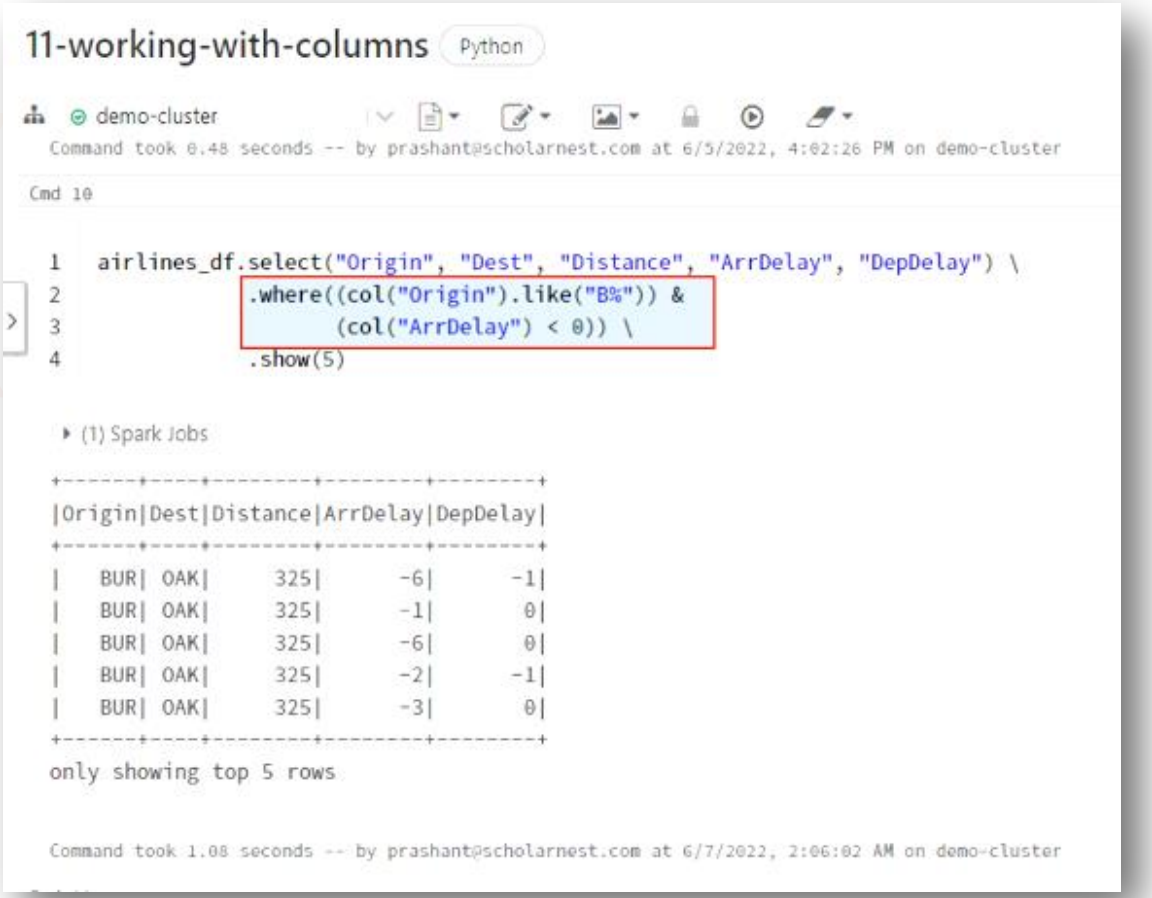
But if you try getting a similar lookup for other approaches, you will see column APIs.
Place your cursor at the end of the expr() function, type a dot, and press the tab.
You will see column APIs such as alias, asc, between, etc.
These are column APIs, and they help you create complex expressions.



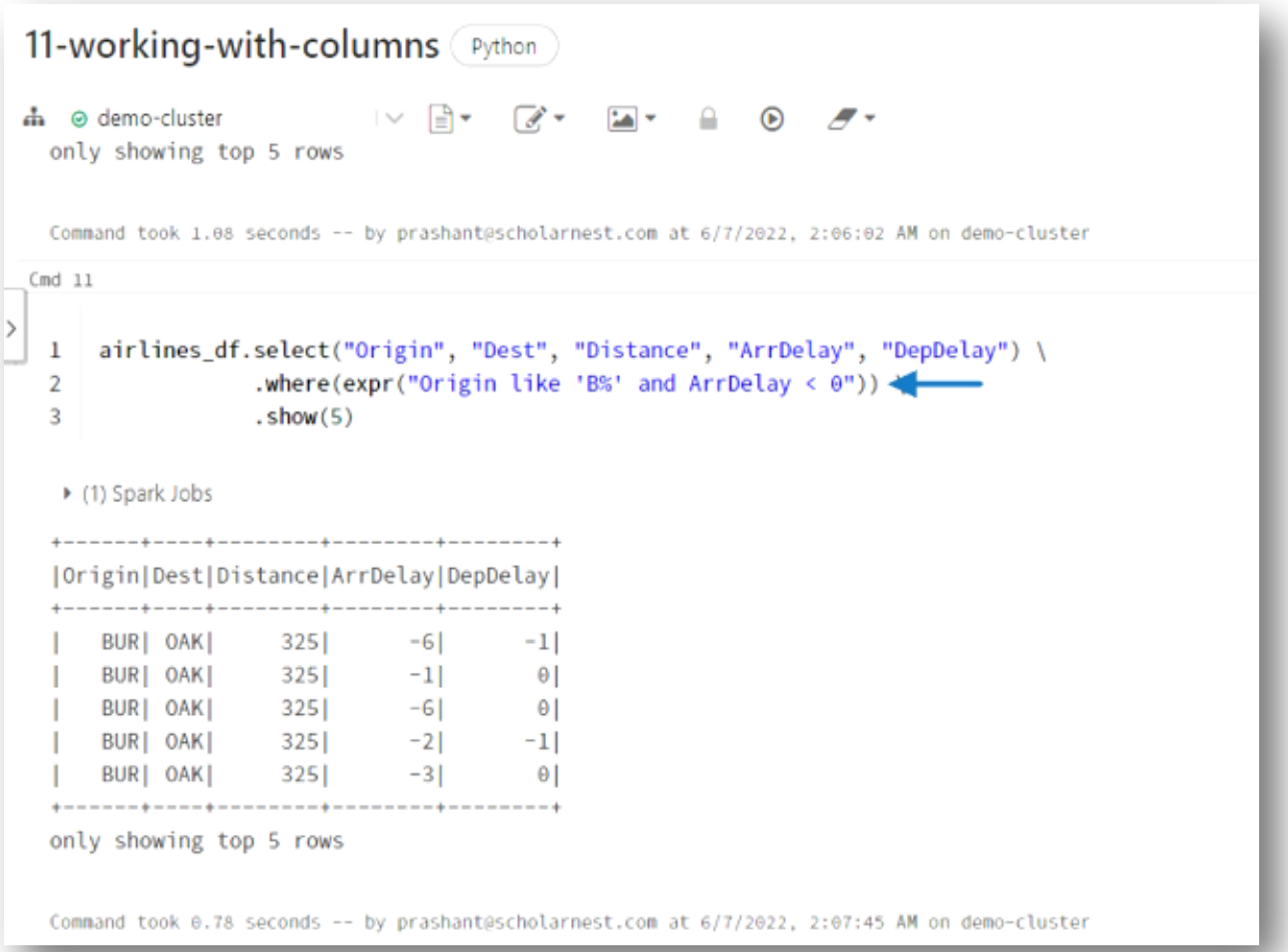
So I am selecting five columns here in this example. Then I am applying a where() condition. I wanted to apply two conditions.

1. where the origin is like B%
2. ArrDelay < 0

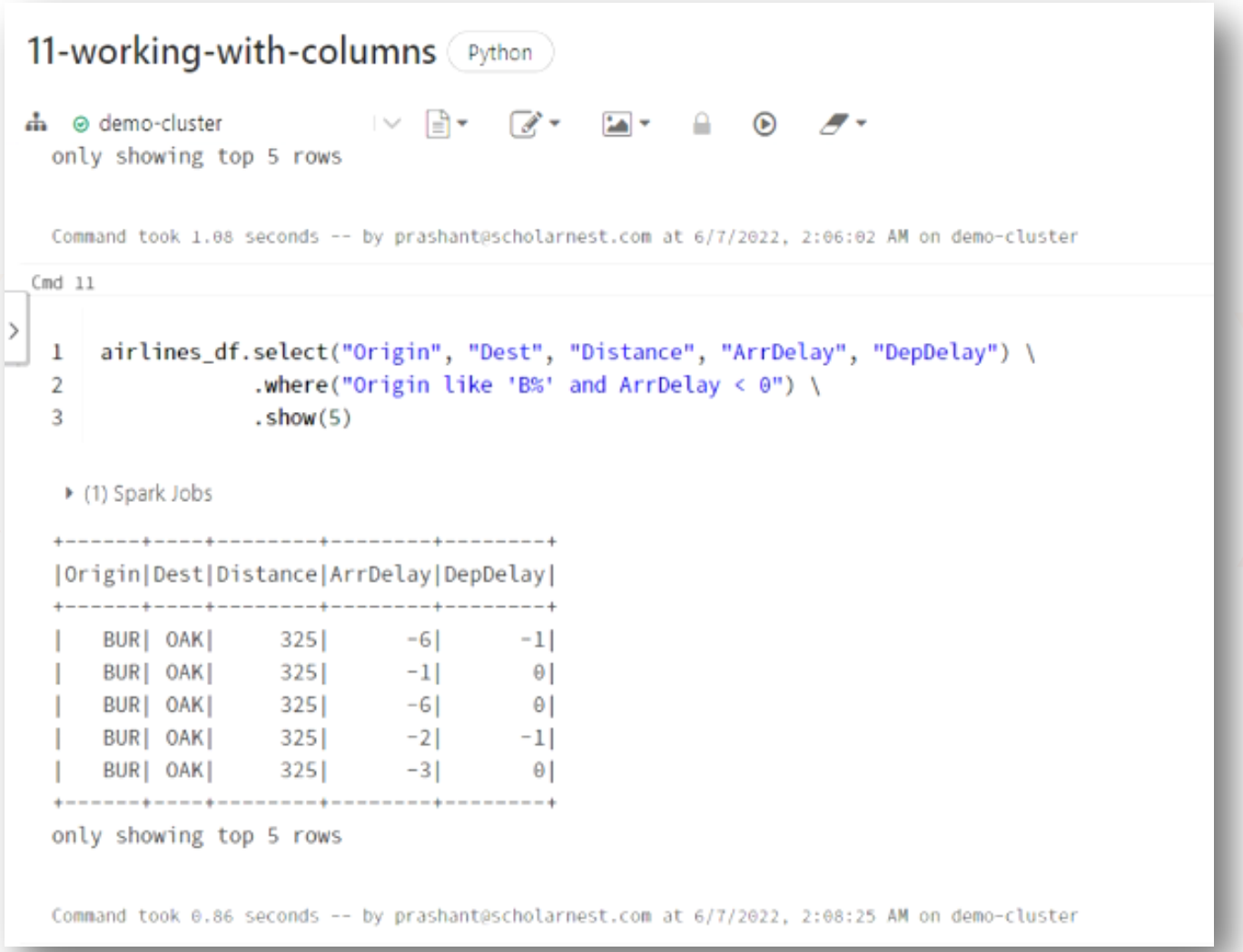
I defined both the conditions here. But I implement it using the Dataframe Column APIs. The col("Origin") will give me a Dataframe column object. The like() is a Column API. Similarly, I defined the second condition. This approach to defining expressions might look complicated for many developers. So we often do it using string expressions.



Here is the same example but using the string expression. Look at the string expression here. This approach looks easy for SQL developers. The earlier approach looks easy for Python or Java Experts. But you have both options. Feel free to use any of the approaches, and Spark performs the same for both approaches. I am using the expr() function inside the where() method. But the expr() is options inside the where() method.



The where() method expects an expression, and it will automatically evaluate the string. So you can remove the expr() function and simply use the string.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

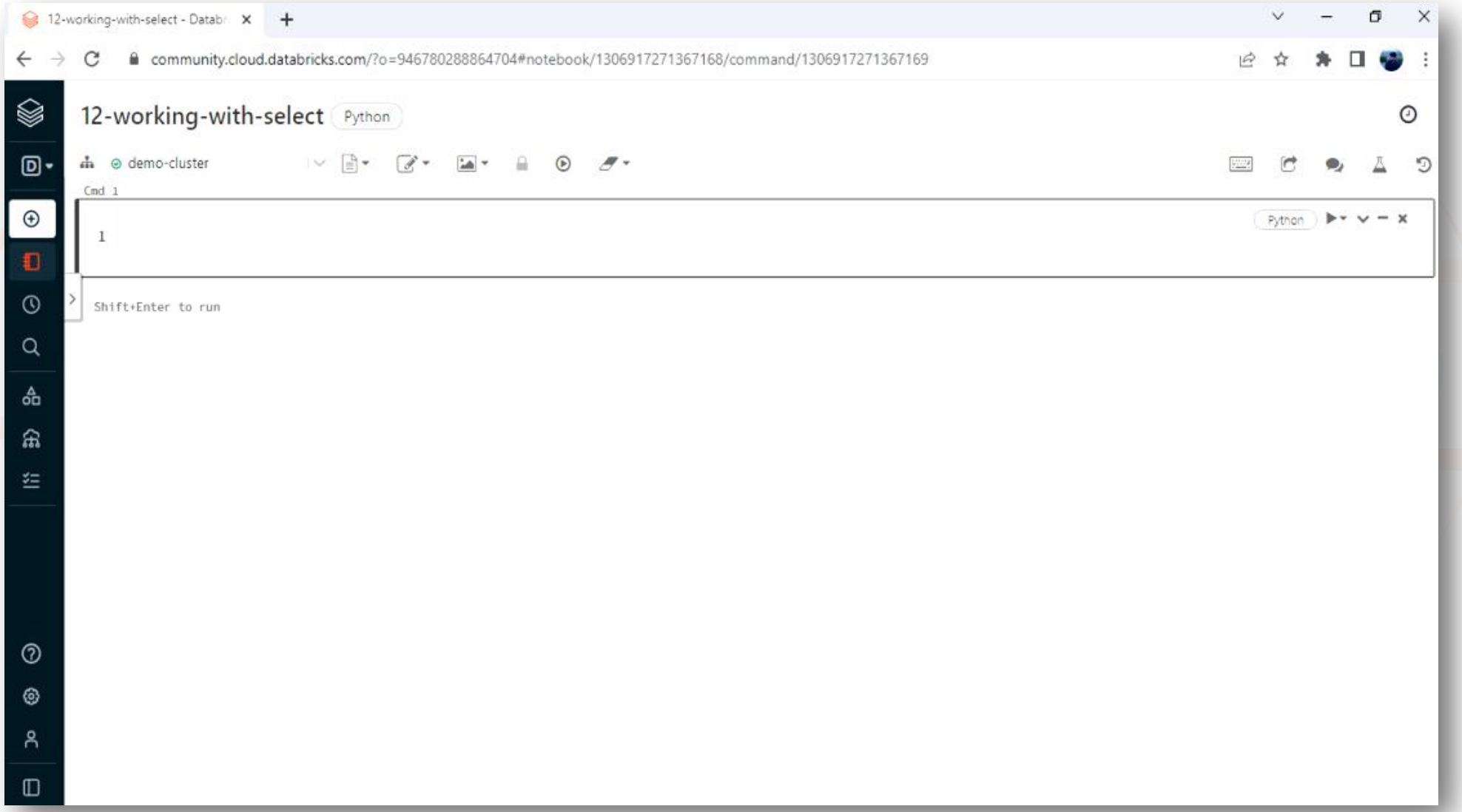
Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Working with Select and SelectExpr

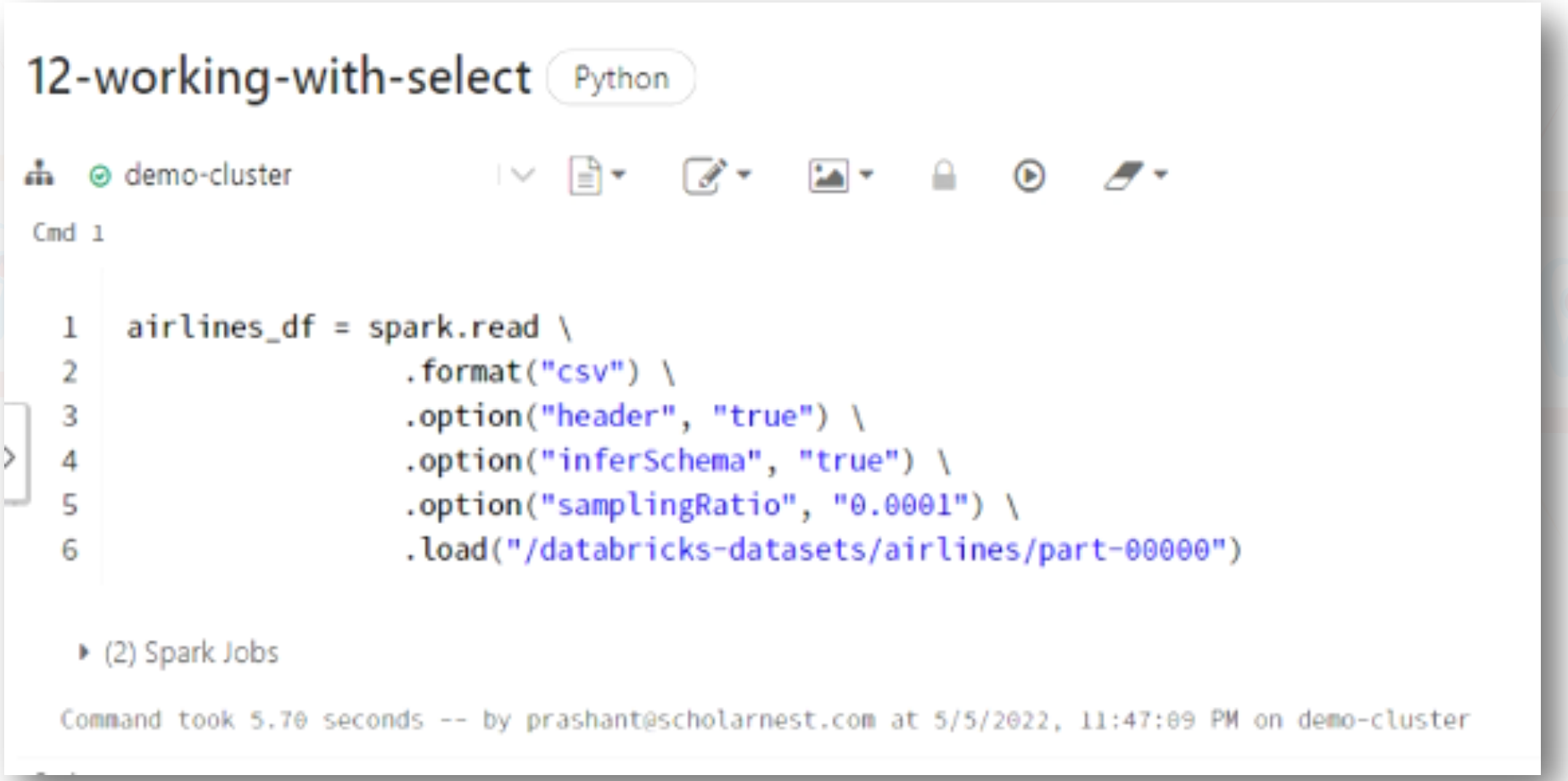
Spark gives you two methods to select one or more columns from your dataframe.

1. select()
2. selectExpr()

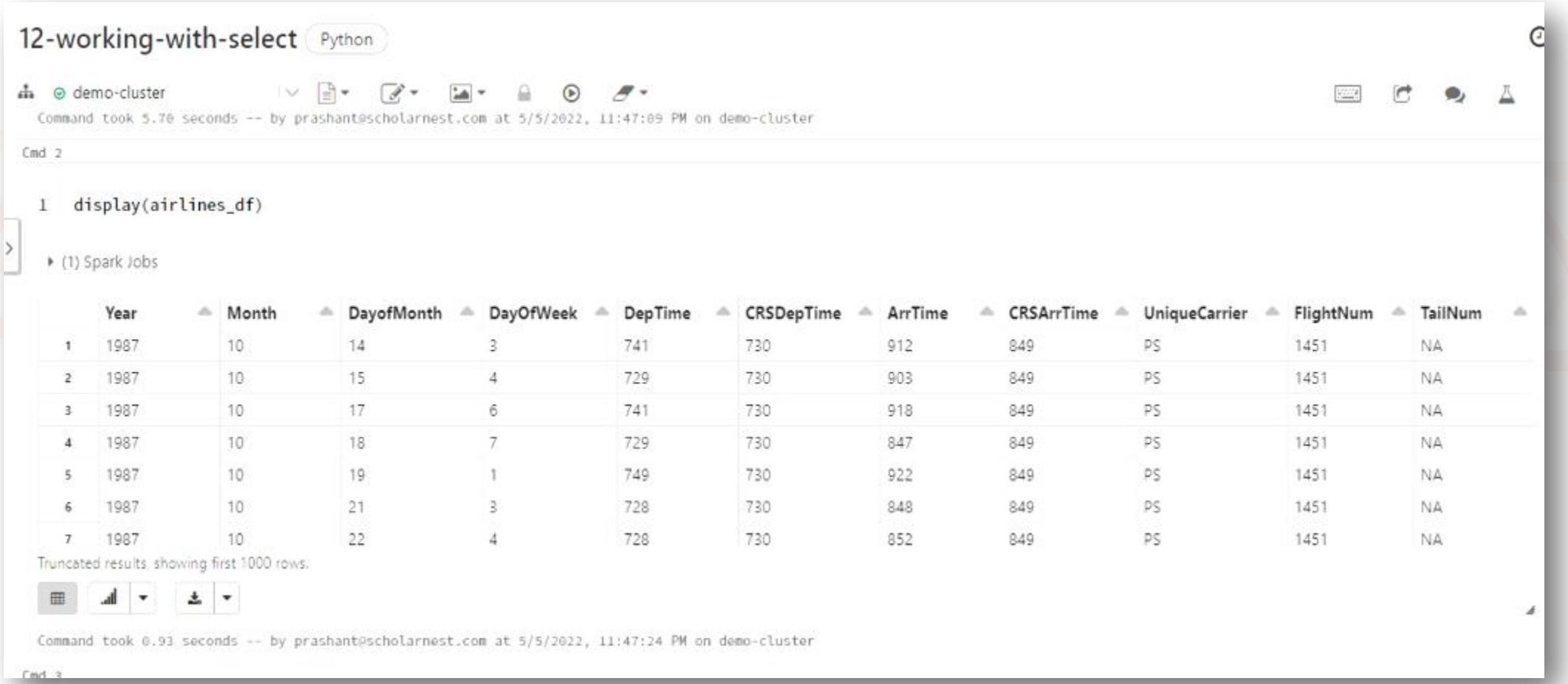
Go to your Databricks workspace and create a new notebook **(Reference - 12-working-with-select)**



We have created a Dataframe for using the select() method. I am reading one airline sample data from Databricks samples dataset and creating a Dataframe.



We have airlines_df, and we can start applying transformations. Let's assume this dataframe was a table. So you could run a select * query on this table like this - **SELECT * FROM table_name;**



	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	UniqueCarrier	FlightNum	TailNum
1	1987	10	14	3	741	730	912	849	PS	1451	NA
2	1987	10	15	4	729	730	903	849	PS	1451	NA
3	1987	10	17	6	741	730	918	849	PS	1451	NA
4	1987	10	18	7	729	730	847	849	PS	1451	NA
5	1987	10	19	1	749	730	922	849	PS	1451	NA
6	1987	10	21	3	728	730	848	849	PS	1451	NA
7	1987	10	22	4	728	730	852	849	PS	1451	NA

So I am selecting four columns here in this code. The select method takes a comma-separated list of column names and returns only those columns. You can give column names or column objects. The select method is designed to accept both, and you can also mix the argument types. You can give a column name or supply a column object in the same select expression.

Cmd 4

```
1  from pyspark.sql.functions import col, column
2
3  airlines_df.select("CRSDepTime", col("DepTime"), airlines_df.CRSArrTime, column("ArrTime")).show(5)
```

▸ (1) Spark Jobs

```
+----------+-------+----------+-------+
|CRSDepTime|DepTime|CRSArrTime|ArrTime|
+----------+-------+----------+-------+
|       730|    741|       849|    912|
|       730|    729|       849|    903|
|       730|    741|       849|    918|
|       730|    729|       849|    847|
|       730|    749|       849|    922|
+----------+-------+----------+-------+
only showing top 5 rows
```

Command took 0.85 seconds -- by prashant@scholarnest.com at 5/5/2022, 11:51:17 PM on demo-cluster

You can also give expressions that result in a column object as shown in the code below. The first argument is a column name. The second argument is also a column name, but I am wrapping it inside the expr() function. The expr() function evaluates an expression and returns a column object. We do not have any expression here. DepTime is just a column name. So I should not have wrapped it inside the expr() function. However, the expr() function doesn't complain. It will simply evaluate the column name to a column object. The third argument is an expression. I am deducting the scheduled departure time from the departure time to find the delay and renaming the result as a delay.

Cmd 5

```
1  from pyspark.sql.functions import expr
2
3  airlines_df.select("CRSDepTime", expr("DepTime"), expr("DepTime - CRSDepTime as delay")).show(5)
```

▸ (1) Spark Jobs

```
+----------+-------+-----+
|CRSDepTime|DepTime|delay|
+----------+-------+-----+
|       730|    741|   11|
|       730|    729|   -1|
|       730|    741|   11|
|       730|    729|   -1|
|       730|    749|   19|
+----------+-------+-----+
only showing top 5 rows
```

Command took 0.71 seconds -- by prashant@scholarnest.com at 5/5/2022, 11:53:05 PM on demo-cluster

The select() method takes only two types of arguments.

1. Column Name
2. Column Object

You can see some negative delays. I don't want them. I mean, the flight went early. But we are calculating delay here. So I want to see only positive delays and set all the negative values to 0.
How can we do it?

Cmd 5

```
1  from pyspark.sql.functions import expr
2
3  airlines_df.select("CRSDepTime", expr("DepTime"), expr("DepTime - CRSDepTime as delay")).show(5)
```

▸ (1) Spark Jobs

```
+----------+-------+-----+
|CRSDepTime|DepTime|delay|
+----------+-------+-----+
|       730|    741|   11|
|       730|    729|   -1|
|       730|    741|   11|
|       730|    729|   -1|
|       730|    749|   19|
+----------+-------+-----+
only showing top 5 rows
```

Command took 0.71 seconds -- by prashant@scholarnest.com at 5/5/2022, 11:53:05 PM on demo-cluster

Here is the code for that. I have the same SQL expression, and I evaluate it using the expr() function.
The expr() function will evaluate the expression to a column object, and it goes to the select() method.

Cmd 6

```
1  airlines_df.select("CRSDepTime", "DepTime", expr("case when (DepTime - CRSDepTime) > 0 then (DepTime - CRSDepTime) else 0 end as delay")).show(5)
```

▸ (1) Spark Jobs

```
+----------+-------+-----+
|CRSDepTime|DepTime|delay|
+----------+-------+-----+
|       730|    741|   11|
|       730|    729|    0|
|       730|    741|   11|
|       730|    729|    0|
|       730|    749|   19|
+----------+-------+-----+
only showing top 5 rows
```

Command took 0.97 seconds -- by prashant@scholarnest.com at 5/6/2022, 12:01:08 AM on demo-cluster

Sometime, using the expr() function inside the select() looks inconvenient.
I mean, using expressions in the select() method is a common thing.
We often use it. But we have a shortcut to avoid the expr() function explicitly, and that is using the selectExpr() method. The selectExpr() is the same as the select, but it takes an expression.
The select method takes a column name or column object. However, the selectExpr() only takes SQL-like expressions.

Cmd 7

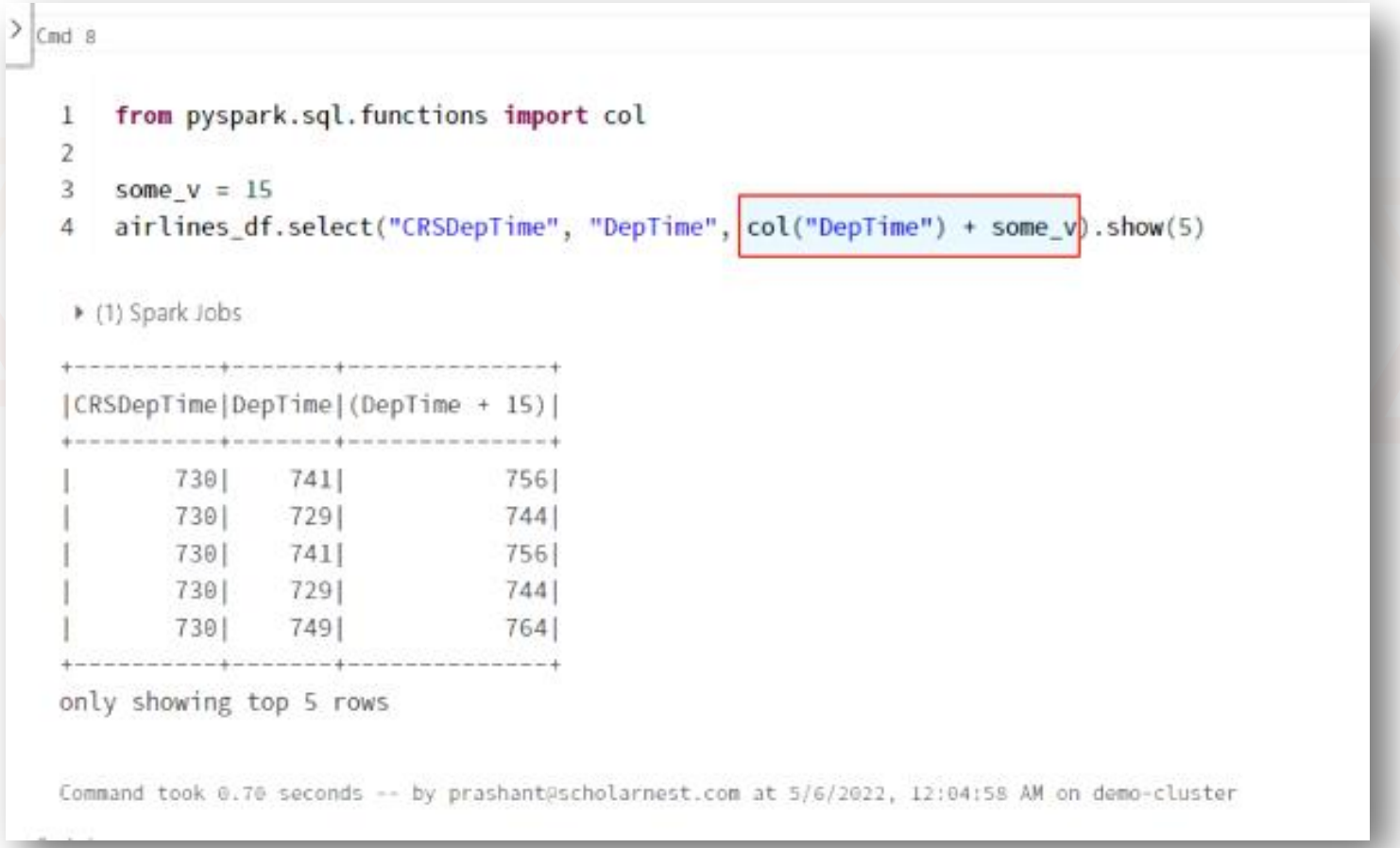
```
1  airlines_df.selectExpr("CRSDepTime", "DepTime", "case when (DepTime - CRSDepTime) > 0 then (DepTime - CRSDepTime) else 0 end as delay").show(5)
```

▸ (1) Spark Jobs

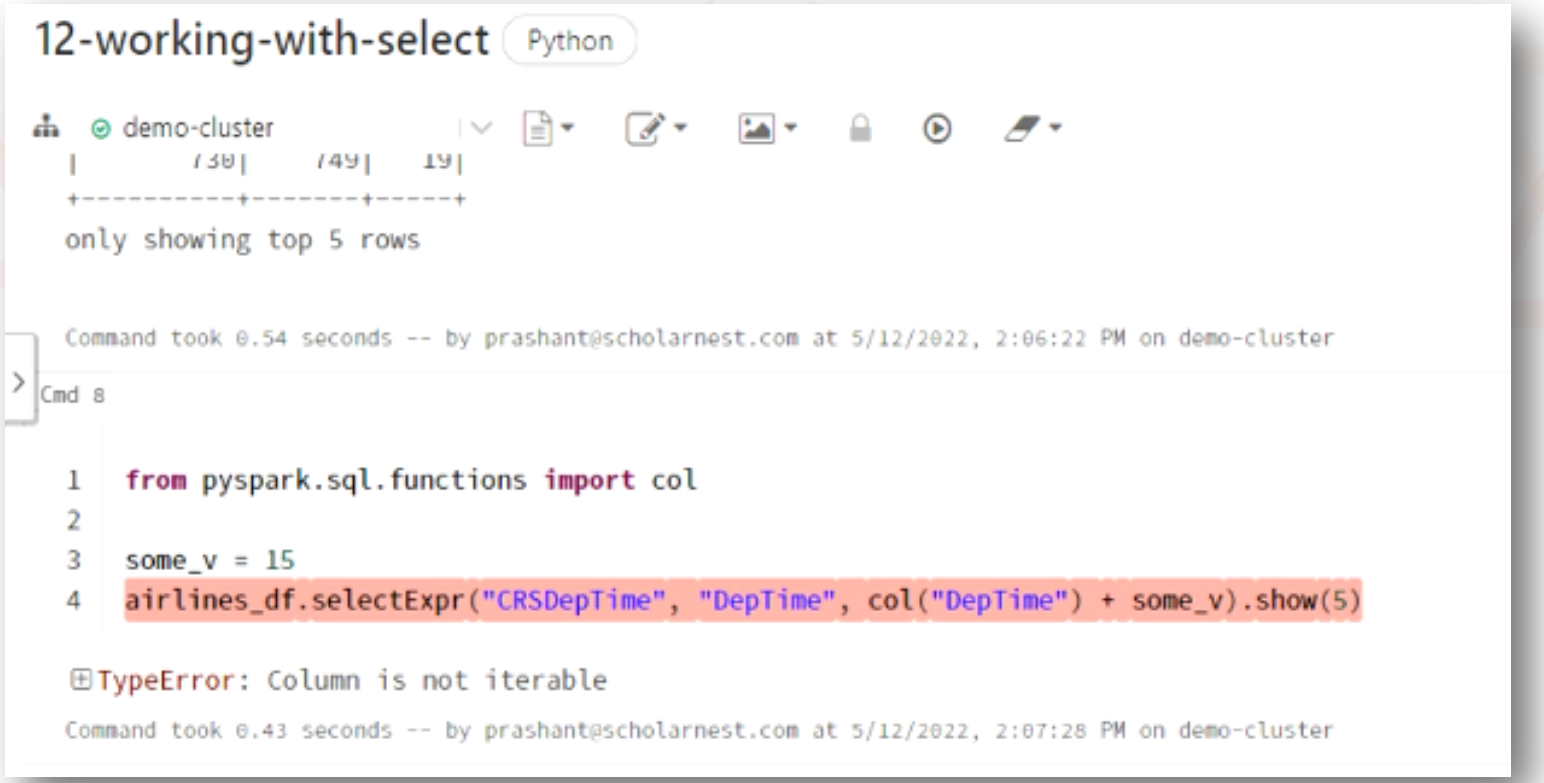
```
+----------+-------+-----+
|CRSDepTime|DepTime|delay|
+----------+-------+-----+
|       730|    741|   11|
|       730|    729|    0|
|       730|    741|   11|
|       730|    729|    0|
|       730|    749|   19|
+----------+-------+-----+
only showing top 5 rows
```

Command took 0.49 seconds -- by prashant@scholarnest.com at 5/6/2022, 12:02:16 AM on demo-cluster

Here is another example.
Here third argument to the select() method is using a dataframe expression.
And the select() method takes it, because the expression ultimately evaluates to a column object. But if you try using selectExpr(), you will see an error.



Here we are trying to use the selectExpr(), and you can see an error.
So understand the differece between select() and selectExpr(). The select method takes a column name and a column object. You can use expressions as long as it evaluates a column object.
The selectExpr() only takes SQL-like expressions. You can also use column names because they evaluate as an expression.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

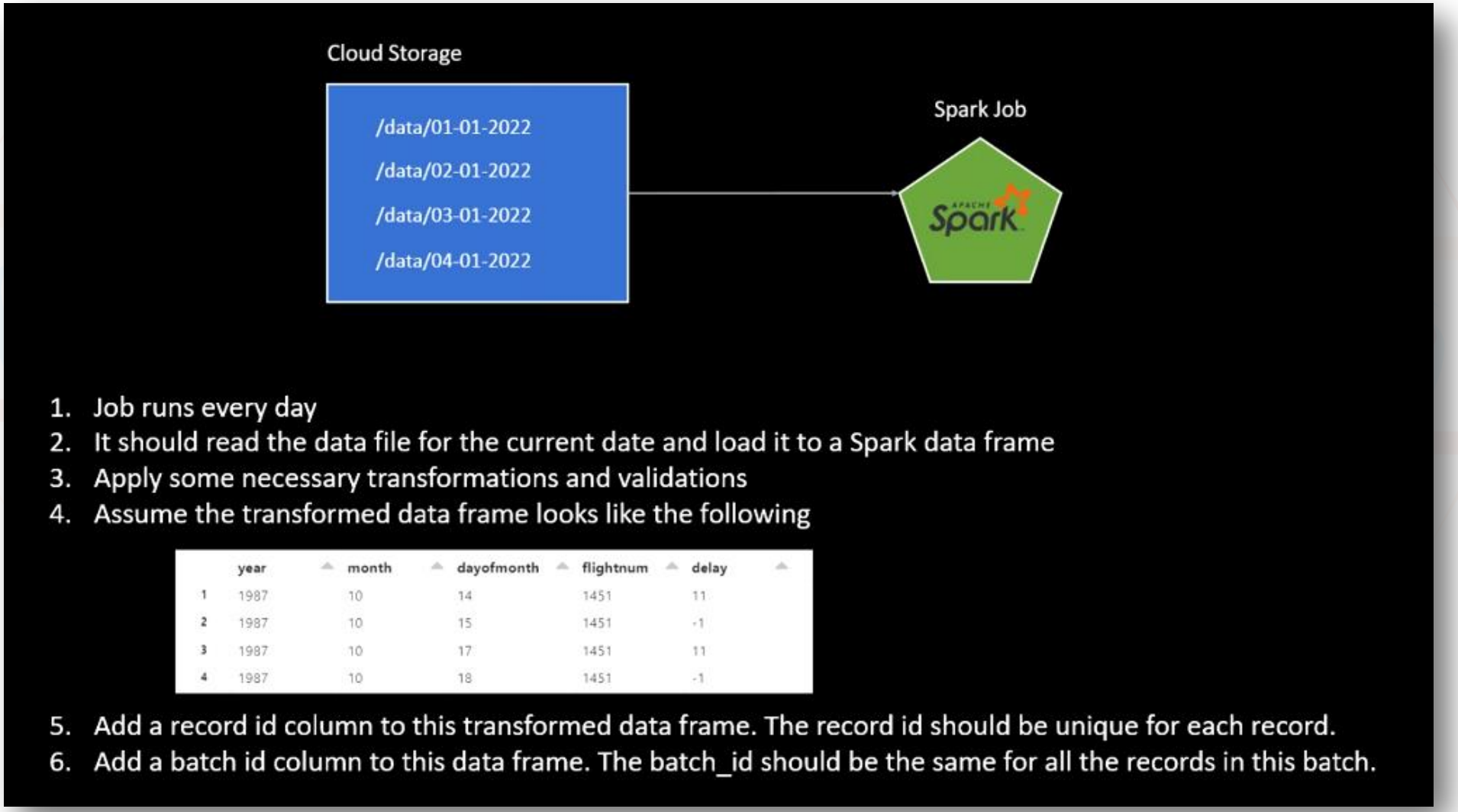
Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Working with Literal Values

We have a simple requirement. You receive data in a cloud storage directory.
The data comes every day in a date-wise directory. You are asked to create a Spark batch job to do the following.



1. Job runs every day
2. It should read the data file for the current date and load it to a Spark data frame
3. Apply some necessary transformations and validations
4. Assume the transformed data frame looks like the following

	year	month	dayofmonth	flightnum	delay
1	1987	10	14	1451	11
2	1987	10	15	1451	-1
3	1987	10	17	1451	11
4	1987	10	18	1451	-1

5. Add a record id column to this transformed data frame. The record id should be unique for each record.
6. Add a batch id column to this data frame. The batch_id should be the same for all the records in this batch.

The first requirement is to run the job every day.
And running a Job every day requires scheduling your Spark notebook or your Application file to run daily at a scheduled time.
We are not going into the details of scheduling for now.
I will cover Job scheduling in the latter part of the course.
So leave that requirement for now.

The second requirement is to read the data file for the current date.
There are many ways to achieve this requirement.
One easy method is to have date-wise directories.
Parameterize the load() method to read data from the current-date directory.
I will also leave this requirement and show you an example in the coming lecture.

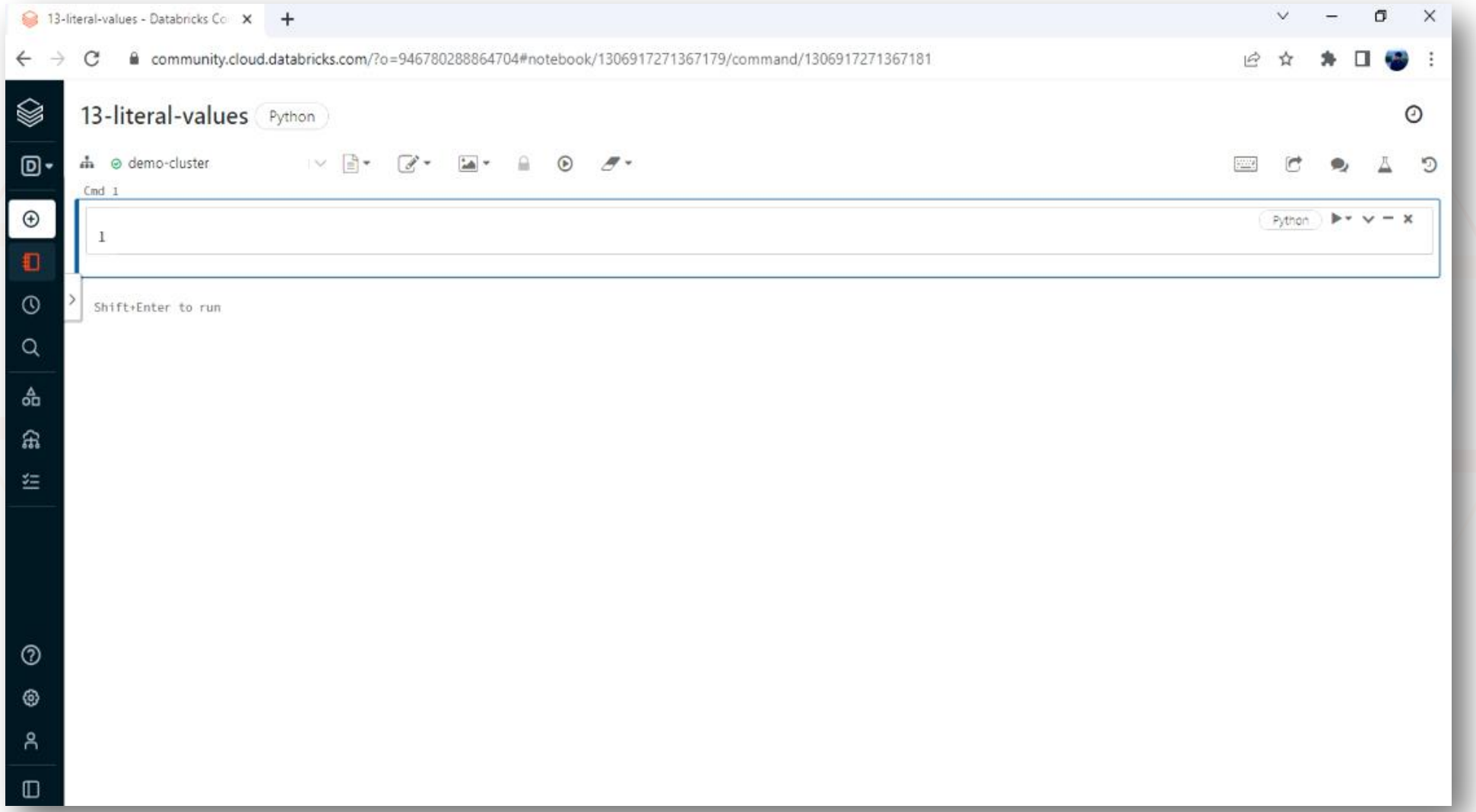
The third requirement is to apply some transformations.
Let's leave this one also for now. We are learning transformations throughout the course.
And the focus of this lecture is on the fifth and the sixth requirement.
So let's leave this requirement also.

The fourth one shows you a sample dataframe.
So we will start from this requirement.
I will create this sample dataframe for you.
And you can learn how to implement the following requirements. Make sense?

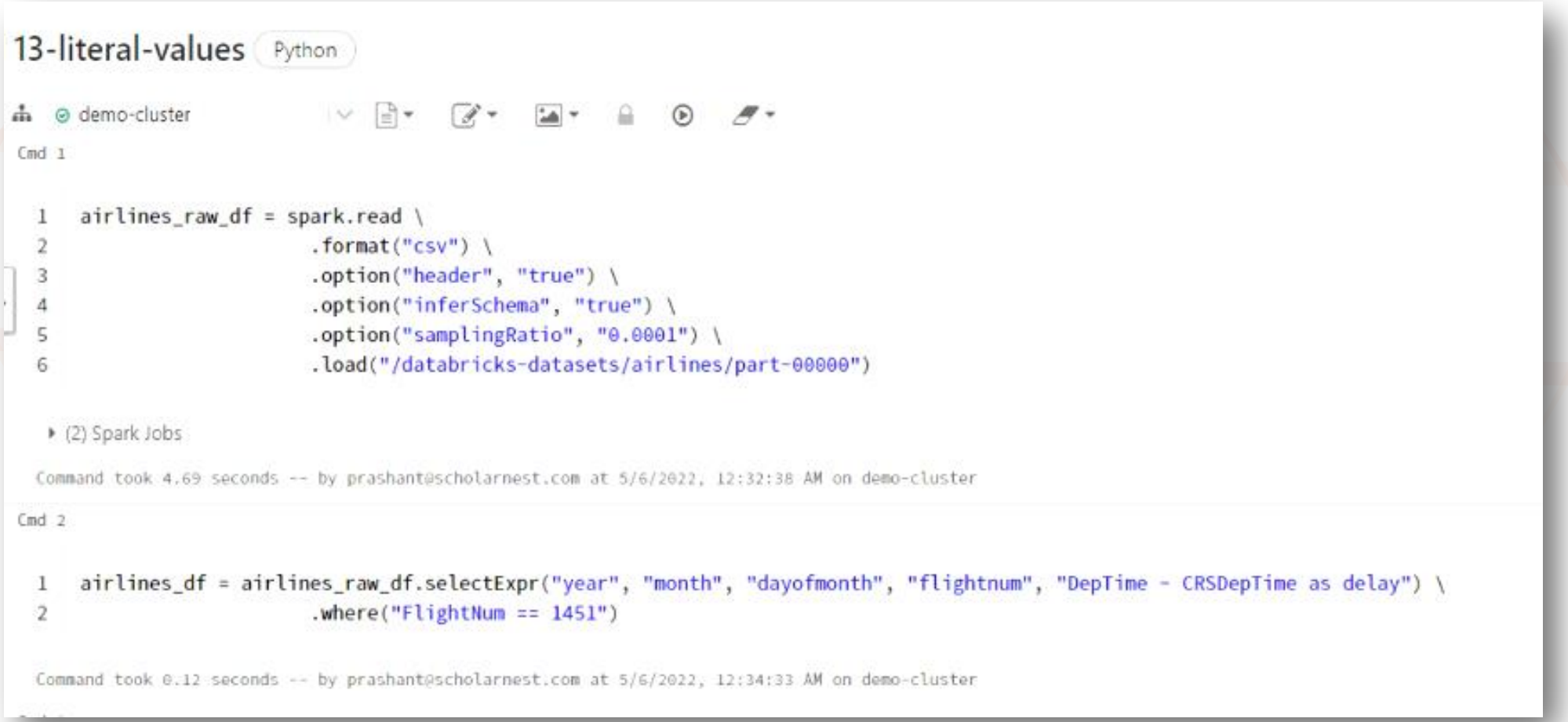
Great! So our primary focus is on the fifth and sixth items.
The fifth item wants us to add a unique record id for each record in the given dataframe.

Let's achieve this requirement, and then we will look into the last one.

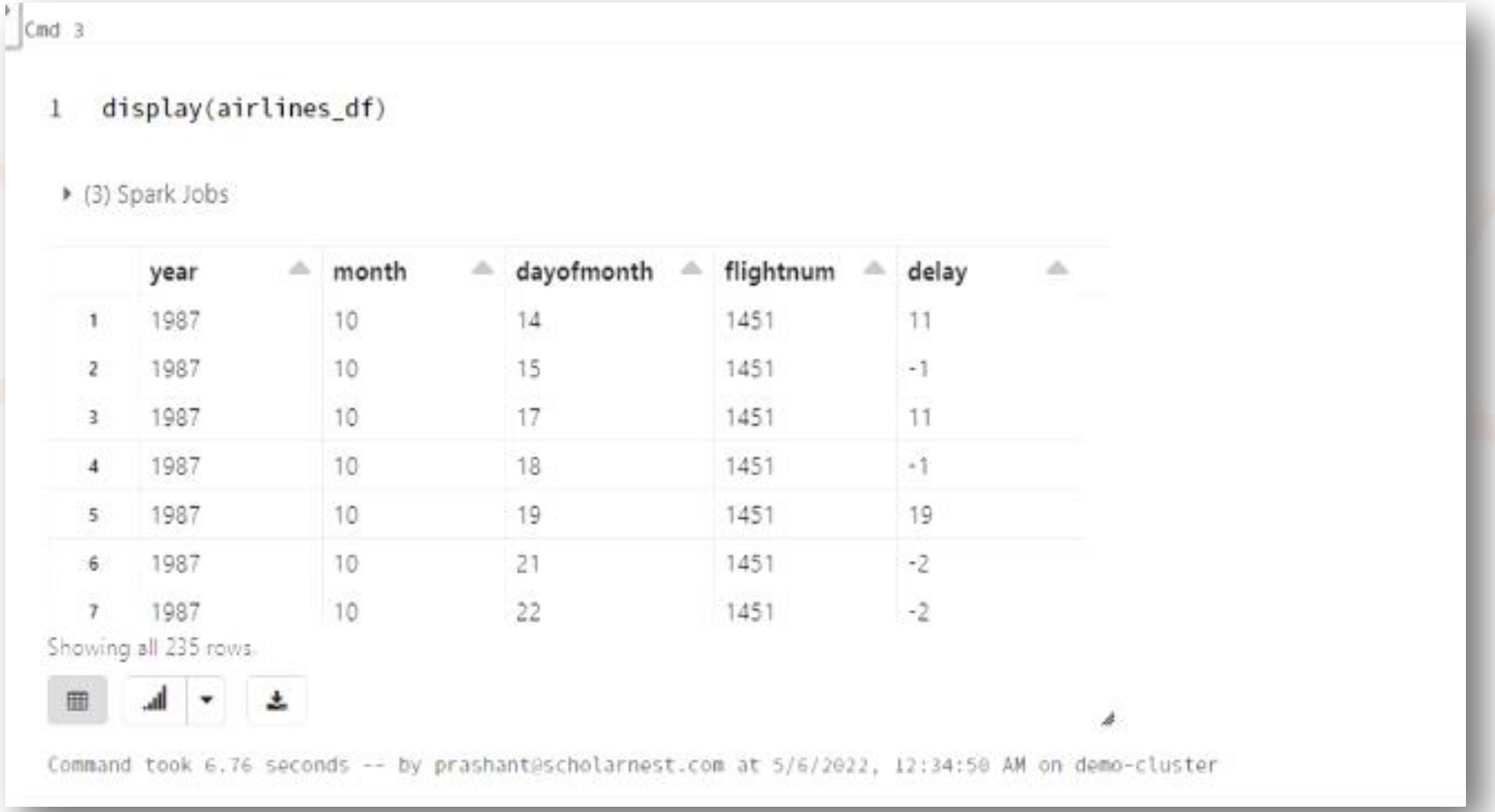
Go to your Databricks workspace and create a new notebook. **(Reference : 13-literal-values)**



The first one is the same Dataframe that we created in our earlier lecture. We need to apply some transformations and prepare a Dataframe as per the example. So we transformed it and created a new Dataframe.



We got the desired Dataframe. Now we have the following requirement:
Add a record id column to this transformed dataframe. The record id should be unique for each record.


Cmd 3

```
1  display(airlines_df)
```

(3) Spark Jobs

	year	month	dayofmonth	flightnum	delay
1	1987	10	14	1451	11
2	1987	10	15	1451	-1
3	1987	10	17	1451	11
4	1987	10	18	1451	-1
5	1987	10	19	1451	19
6	1987	10	21	1451	-2
7	1987	10	22	1451	-2

Showing all 235 rows.

Command took 6.76 seconds -- by prashant@scholarnest.com at 5/6/2022, 12:34:50 AM on demo-cluster

How to do it? Can we use some function that generates a unique id for us?
We have two places to look for the available functions.

1. Built-in Spark SQL functions
2. Dataframe functions.

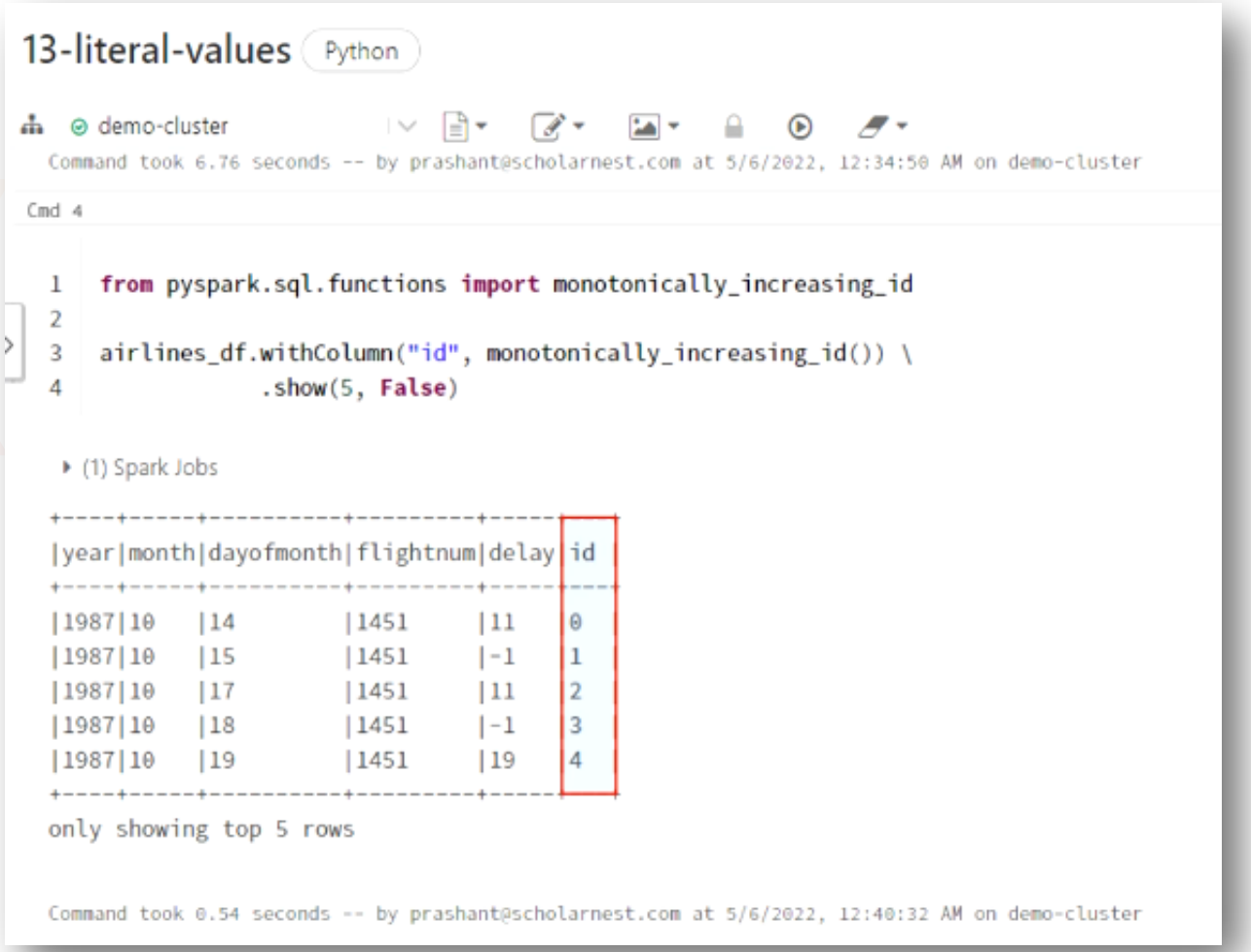
I checked both the places and identified two potential functions that we can use.

1. uuid()
2. monotonically_increasing_id()

The uuid() function is a built-in spark SQL function.
It generates a 32-character unique id.
We can use it to generate a UUID for each record.

The monotonically_increasing_id() is a datframe function.
It generates a long number starting from 0.
We can use this function also to add a unique id for our records.

The monotonically_increasing_id() is a dataframe function. So you must import it. Once imported, you can use the withColumn() transformation to add a new column to your dataframe. The withColumn() takes two arguments. The first argument is the column name, and the second argument is the logic. The logic is as simple as using a function and generating a monotonically_increasing_id. You can see in the output there is the id column at the end, it starts with zero and goes on increasing by one.



We have another alternative of using the UUID() built-in function. So I added another column and used the UUID() built-in function. Since the UUID() is a built-in function, I must use the expr() function to evaluate it. And we can see a unique id for each record. You can use any one of these depending upon your requirement.

13-literal-values Python

demo-cluster

Command took 6.76 seconds -- by prashant@scholarnest.com at 5/6/2022, 12:34:50 AM on demo-cluster

Cmd 4

```
1  from pyspark.sql.functions import monotonically_increasing_id, expr
2
3  airlines_df.withColumn("id", monotonically_increasing_id()) \
4             .withColumn("RecordID", expr("uuid()")) \
5             .show(5, False)
```

(1) Spark Jobs

```
+----+-----+----------+---------+-----+---+------------------------------------+
|year|month|dayofmonth|flightnum|delay|id |RecordID                            |
+----+-----+----------+---------+-----+---+------------------------------------+
|1987|10   |14        |1451     |11   |0  |bbfb4a69-2c6e-4f6e-a695-27edf83ccc74|
|1987|10   |15        |1451     |-1   |1  |506c57dc-c092-4c5e-b555-5acb7905919f|
|1987|10   |17        |1451     |11   |2  |b84dcefd-777c-40e2-94fa-c29b3e7bd769|
|1987|10   |18        |1451     |-1   |3  |12fed1e8-7b7e-42a7-a699-5cde16d11e8e|
|1987|10   |19        |1451     |19   |4  |5cf3a766-ac8e-40f7-9479-8dbba2e7f03f|
+----+-----+----------+---------+-----+---+------------------------------------+
only showing top 5 rows
```

Command took 0.52 seconds -- by prashant@scholarnest.com at 5/6/2022, 12:48:35 AM on demo-cluster

We want to add a batch id column to this dataframe. The batch_id should be the same for all the records in this batch. So it is like a hardcoded id for all the records.
How will you do it?
Can you even think of a SQL way of doing it?
Assume you have a table. You want to select all columns and add one extra column with a hardcoded value.
Here is a SQL expression to do it:

SELECT *, "batch-123456789" as batchID from table_name;

I wanted to add a new column to my Dataframe, so I used them with column transformation. The column name is BatchID, and the logic to create a column is a literal value. But if we try running it, we will get an error. Because the Dataframe doesn't understand the literal value. The second argument to the withColumn() must be an expression that evaluates a column object. This string is not an expression. I can wrap it with an expr() function, but even the expr() function will not evaluate it. Because the string here is a literal value. Expressions are created using column names, functions, and operators. And this string doesn't represent any column name. It is neither a function. It is a literal value, and Dataframe API cannot understand literal values until we explicitly tell it.

I can import a Python UUID package and create a unique id to use the same example. Then I can use the python uid variable inside the lit() function.
So instead of hardcoding a manually created id, I can use a variable to automatically generate a unique id and use it as my batch id.

13-literal-values Python

demo-cluster

Command took 6.76 seconds -- by prashant@scholarnest.com at 5/6/2022, 12:34:50 AM on demo-cluster

Cmd 4

```
1  from pyspark.sql.functions import monotonically_increasing_id, expr, lit
2
3  import uuid
4  uid = str(uuid.uuid4())
5
6  airlines_df.withColumn("id", monotonically_increasing_id()) \
7             .withColumn("RecordID", expr("uuid()")) \
8             .withColumn("BatchID", lit(uid)) \
9             .show(5, False)
```

(1) Spark Jobs

```
+----+-----+----------+---------+-----+---+------------------------------------+------------------------------------+
|year|month|dayofmonth|flightnum|delay|id |RecordID                            |BatchID                             |
+----+-----+----------+---------+-----+---+------------------------------------+------------------------------------+
|1987|10   |14        |1451     |11   |0  |5836314d-3702-4aa7-bd69-06ed0265ba97|932aed37-4837-4eaf-883c-540cc06c1132|
|1987|10   |15        |1451     |-1   |1  |aec77f55-2cfc-4c7b-bc65-bcf2a45f4bfd|932aed37-4837-4eaf-883c-540cc06c1132|
|1987|10   |17        |1451     |11   |2  |6dee1ebf-a0dd-4f43-b3d9-b2ed6b0260c9|932aed37-4837-4eaf-883c-540cc06c1132|
|1987|10   |18        |1451     |-1   |3  |0fc12fc1-d6e5-4280-9923-b88ee368e008|932aed37-4837-4eaf-883c-540cc06c1132|
|1987|10   |19        |1451     |19   |4  |8591b94f-ad50-4edb-ba25-91709408b2cc|932aed37-4837-4eaf-883c-540cc06c1132|
+----+-----+----------+---------+-----+---+------------------------------------+------------------------------------+
only showing top 5 rows
```

Command took 0.67 seconds -- by prashant@scholarnest.com at 5/6/2022, 12:51:01 AM on demo-cluster

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

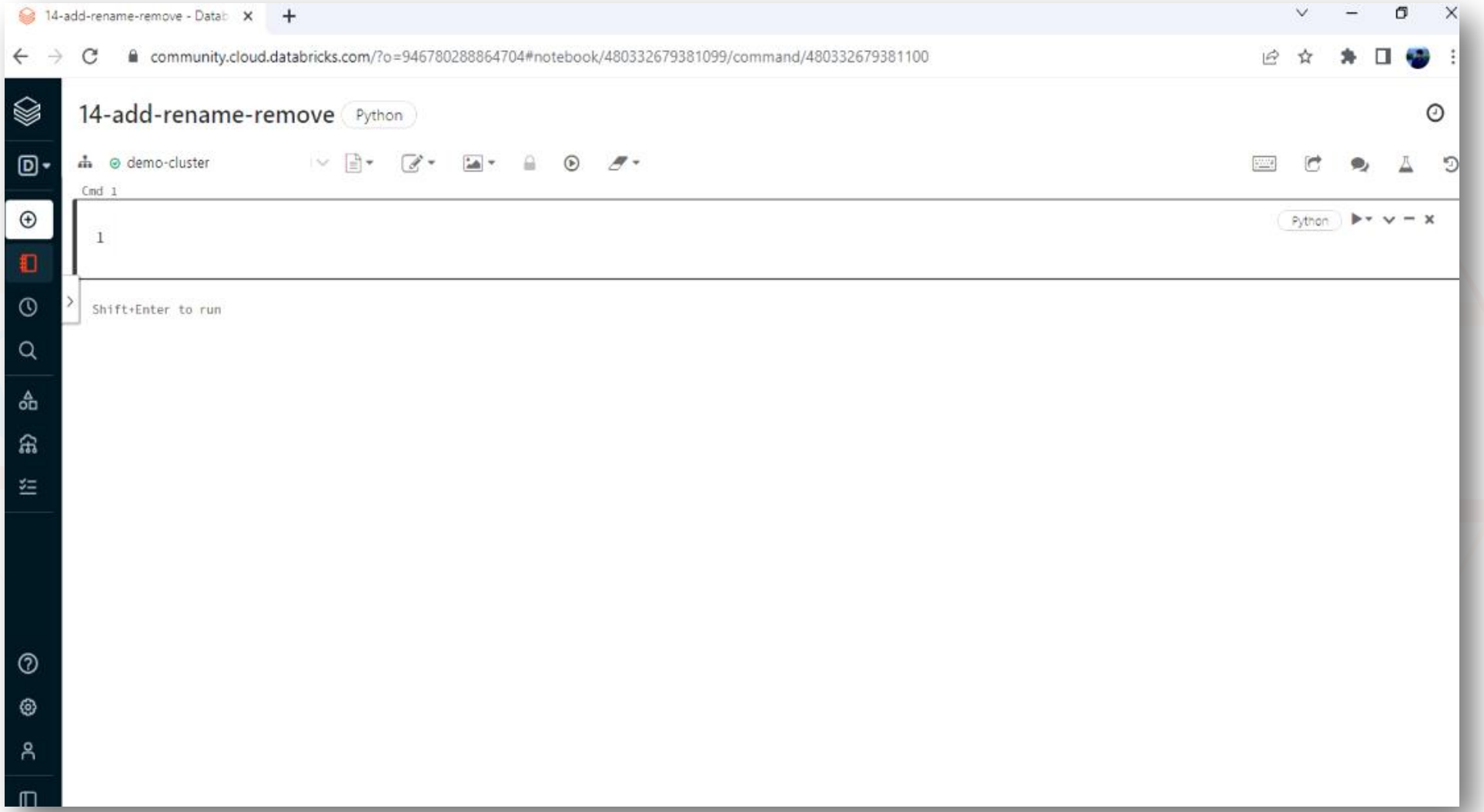
Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Adding, Remaining and Removing Columns

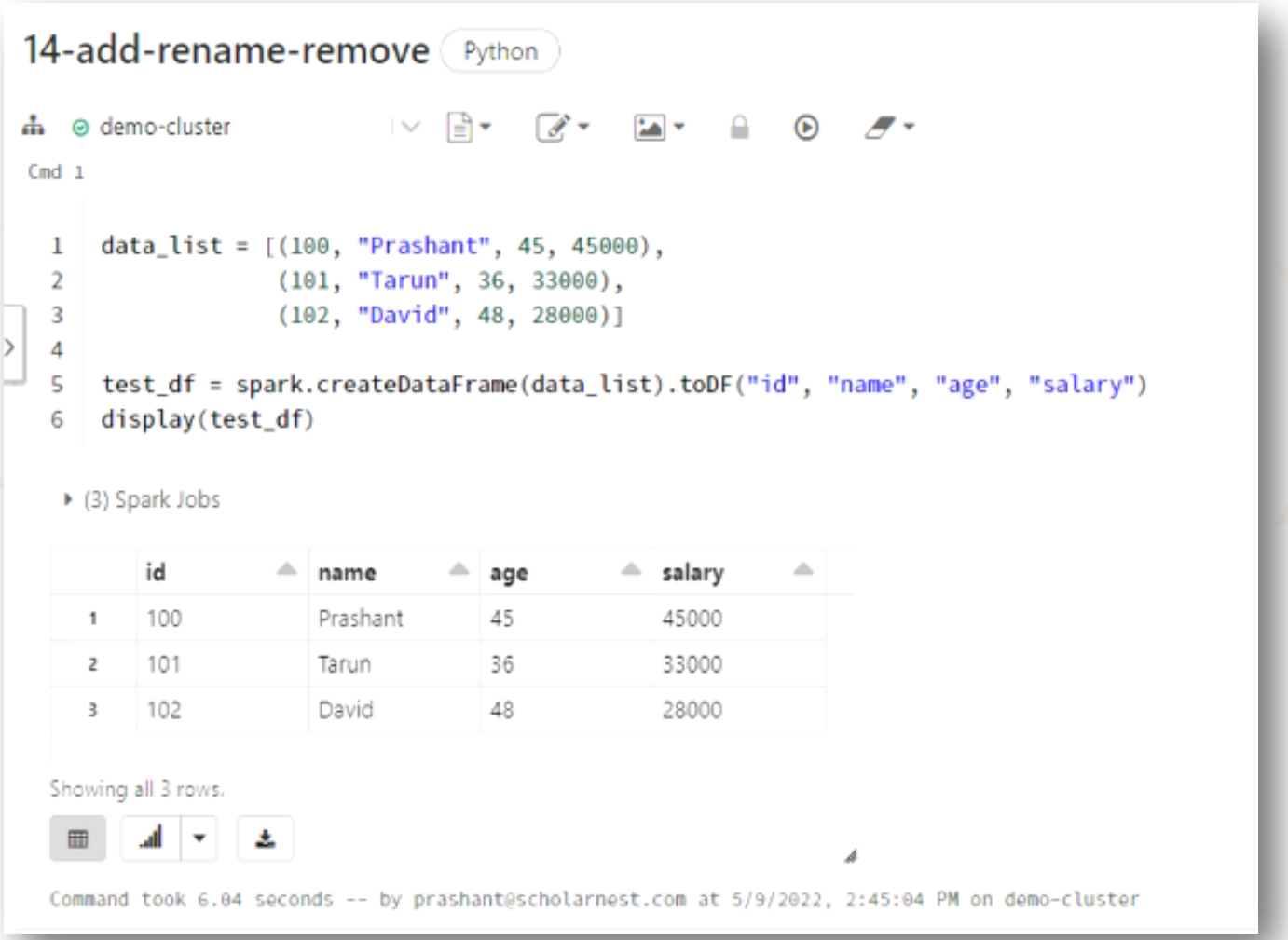
I want to talk about the following quickly in this chapter. The first thing that we want to cover is adding new columns to your dataframe. Spark dataframe gives you withColumn() transformation to do that. You can use withColumn() transformation to add a new column or transform an existing column. We have already used it in many places, and I hope you have already learned it. But let me give you a quick example once again.

1. Adding new columns to your data frame
    - withColumn()
2. Renaming Existing Columns in your data frame
    - withColumnRenamed()
3. Dropping columns from your data frame
    - drop()

Go to your Databricks workspace and create a new notebook **(Reference : 14-add-rename-remove).**



I am creating a data list and using it to define a data frame. You already learned this in an earlier lecture. Now I want to use a new column for this data frame. So I can use the withColumn() transformation.



So you can use the withColumn() method to add a new field to your dataframe. The withColumn() method takes two arguments. The first argument is the column name and the second column is an expression to determine the value of the new columns. The withColumn() can add new columns or transform existing columns. The increment column doesn't exist in this dataframe, so the withColumn() will add it. However, if the column already exists, the withColumn() will simply transform it.

Cmd 2

```
1  from pyspark.sql.functions import expr
2
3  test_df1 = test_df.withColumn("increment", expr("salary * 10/100"))
4  display(test_df1)
```

▸ (3) Spark Jobs

	id	name	age	salary	increment
1	100	Prashant	45	45000	4500
2	101	Tarun	36	33000	3300
3	102	David	48	28000	2800

Showing all 3 rows.

Command took 1.57 seconds -- by prashant@scholarnest.com at 5/9/2022, 2:47:13 PM on demo-cluster

The salary column already existed, so the withColumn() updates the same column. The withColumn() is super convenient. However, it has got some problems. Each withColumn() method introduces a new projection in the SQL execution plan. Therefore, calling it multiple times can generate big SQL execution plans, which can cause performance issues and even StackOverflowException. To avoid this, we prefer to use the select() expression and avoid using the long list of withColumn() method.

Cmd 3

```
1  test_df2 = test_df1.withColumn("salary", expr("salary + increment"))
2  display(test_df2)
```

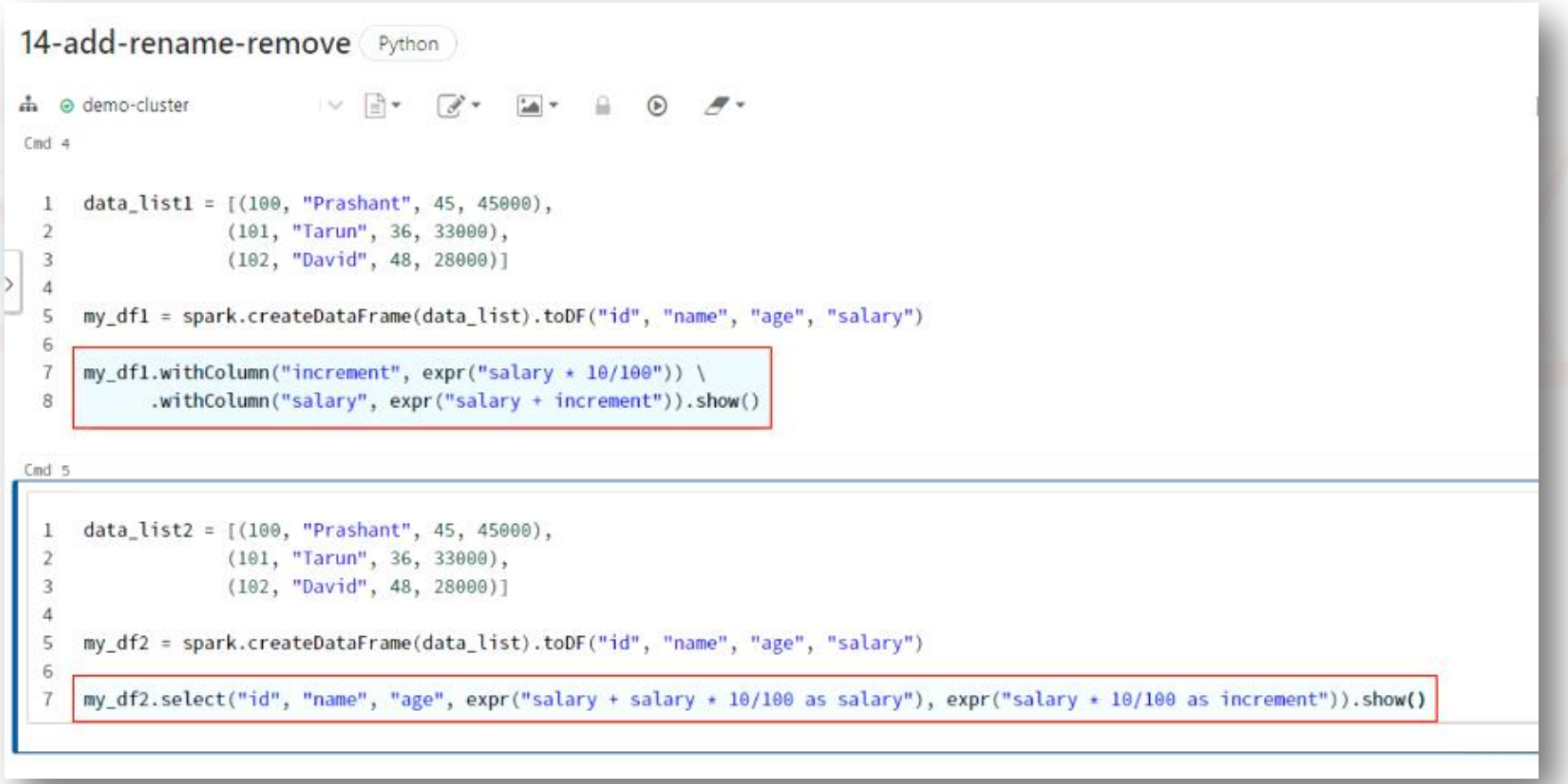
▸ (3) Spark Jobs

	id	name	age	salary	increment
1	100	Prashant	45	49500	4500
2	101	Tarun	36	36300	3300
3	102	David	48	30800	2800

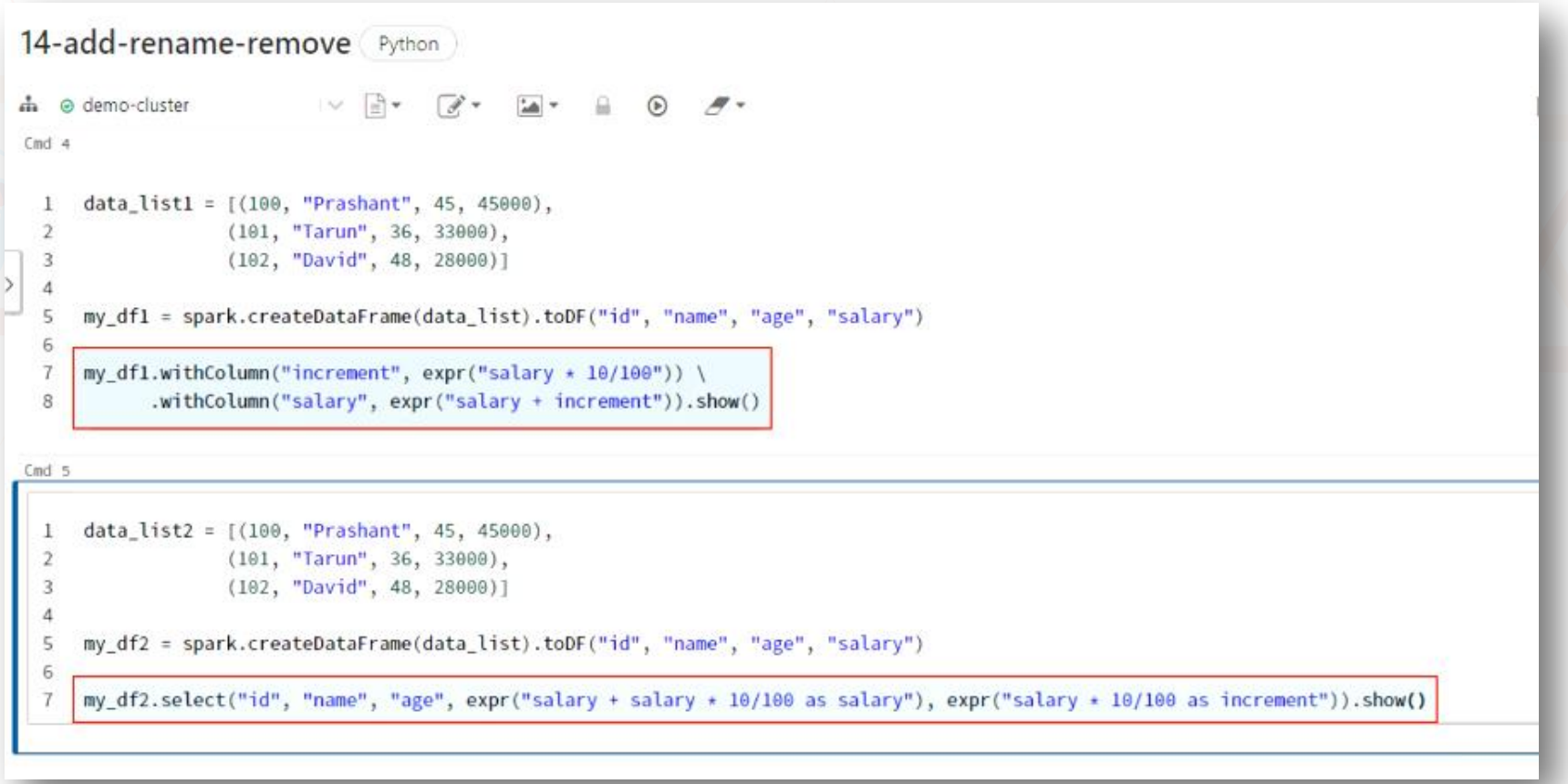
Showing all 3 rows.

Command took 1.30 seconds -- by prashant@scholarnest.com at 5/9/2022, 2:49:07 PM on demo-cluster

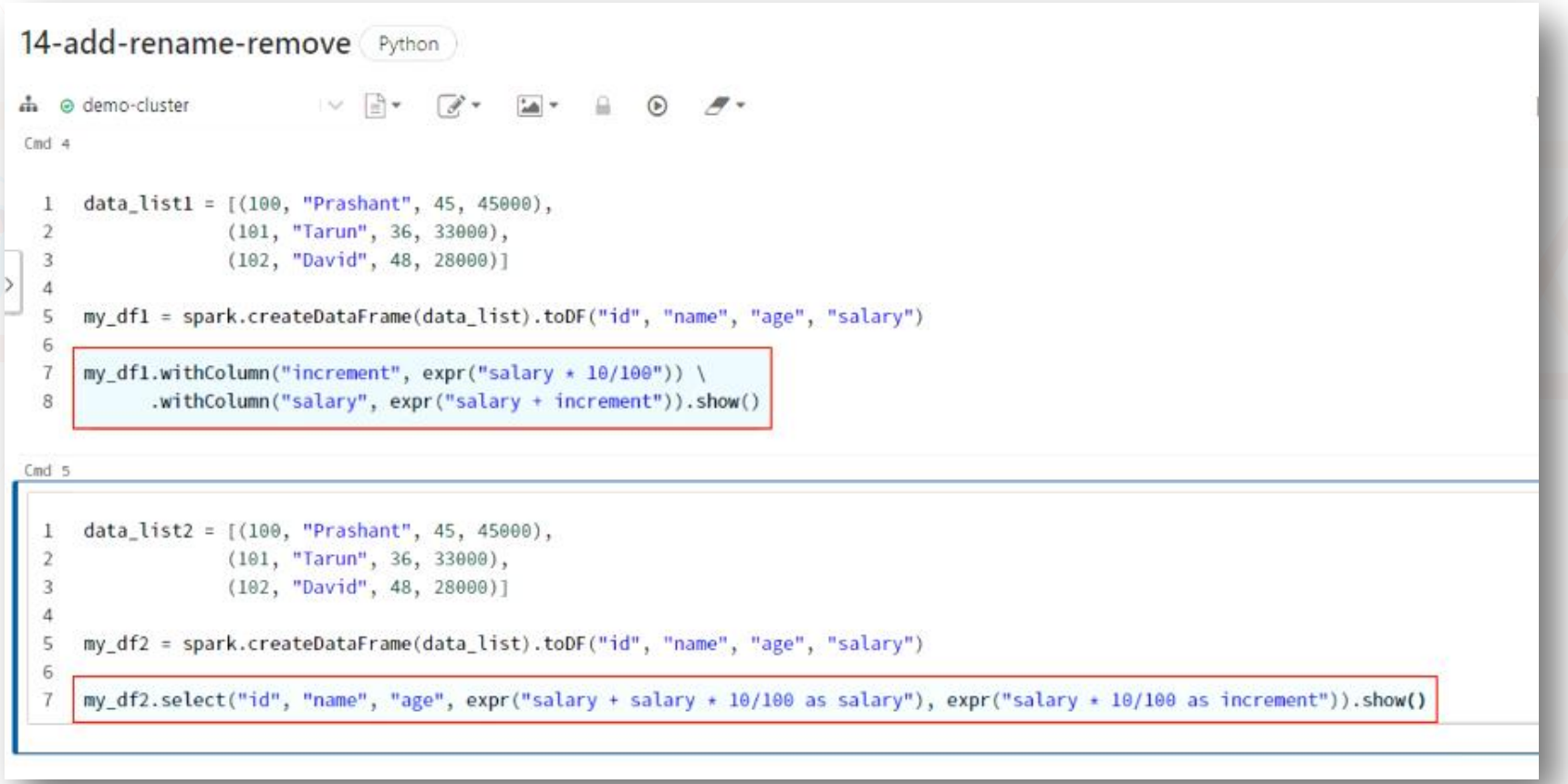
Now look at both codes and compare them.
I am creating data_list1 and then creating a dataframe using the same.
Then I am using the withColumn() method to add an increment column.
Then I am again using a withColumn() to change the salary column value.
Finally, I will show my_df1.



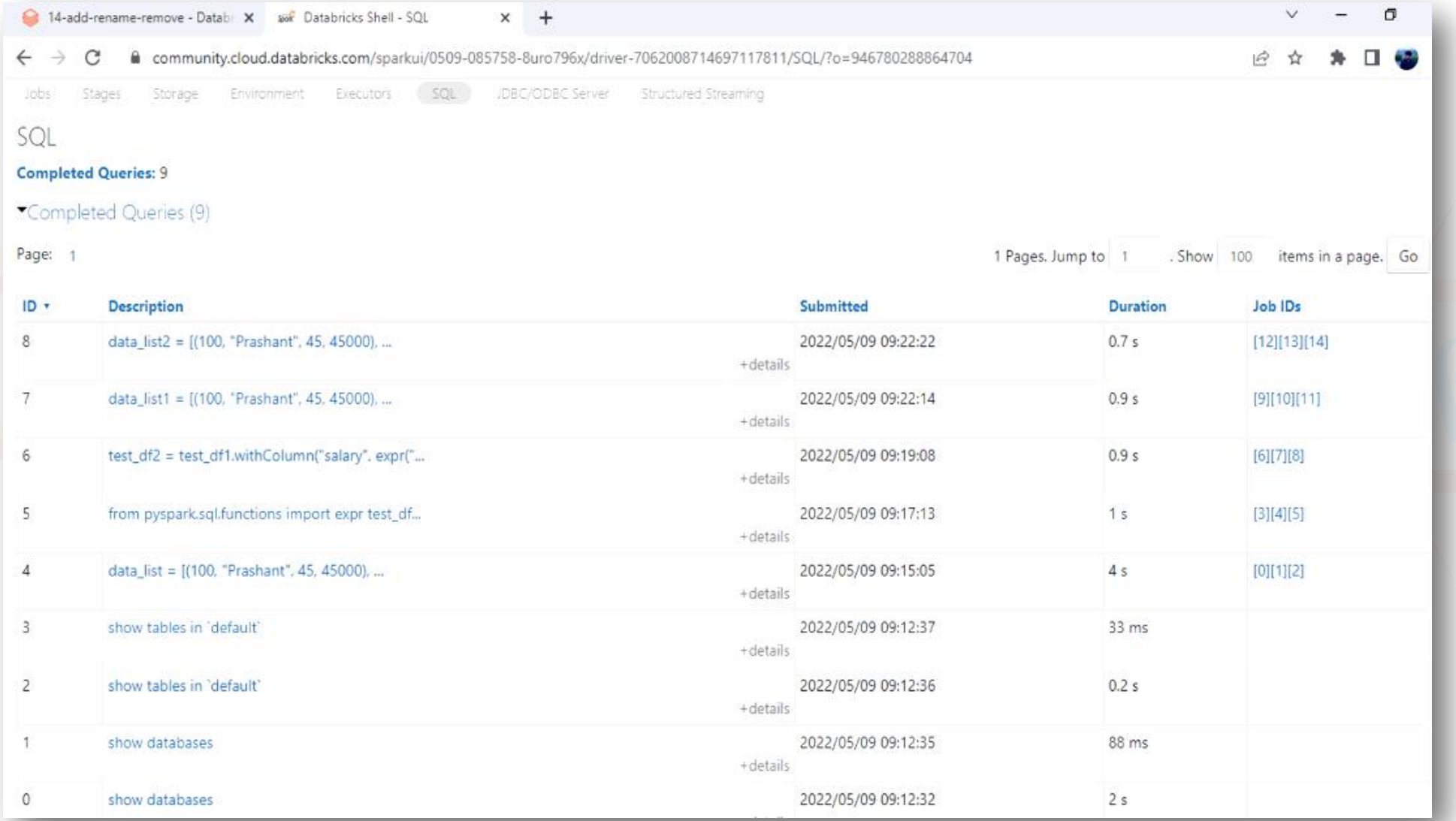
Now, look at the next cell.
I am doing the same thing here but taking a different approach. This time, I am creating data_list2, which is precisely the same as data_list1. Then I am using data_list2 to create my_df2. The my_df2 is also precisely the same as my_df1. Finally, I am using a select() method. The select method selects id, name, and age without any change. But I am using an expression to change the salary column value. Then I use another expression to add an increment column to my dataframe.



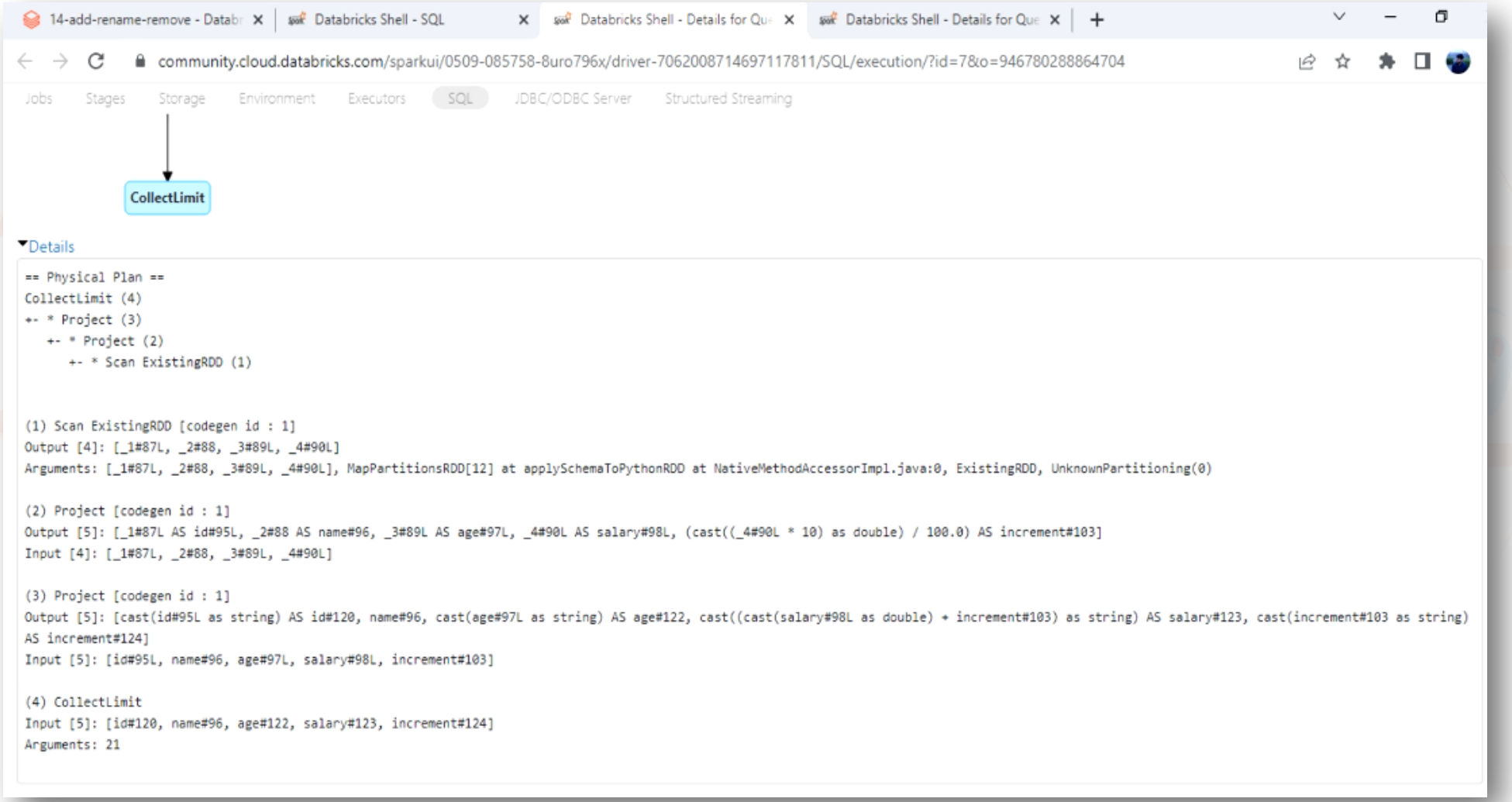
Now compare the two withColumn() expressions in the first cell with the select() method in the second cell. Both the codes are doing the same thing. However, they take a different approach. The first approach is using a chain of two withColumn() methods. The second approach is using a single select() method. Coding the first approach is neat and convenient. But coding the second approach is a bit clumsy.



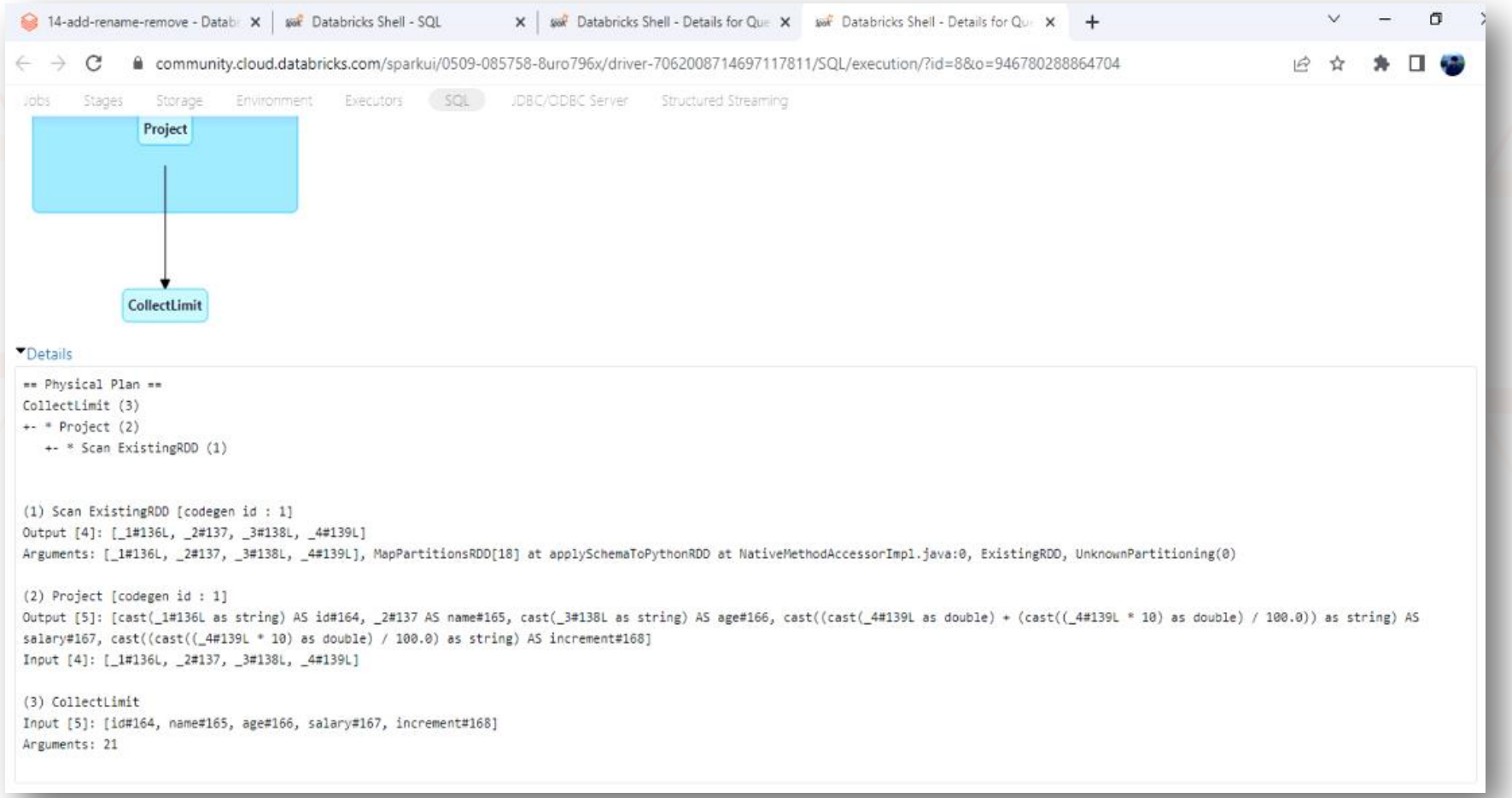
To check which one performs better. Run both the cells shown in the last slide, and check for the Spark UI, then go to the SQL tab. Open ID 8 and 7 in new tab.



Come to the first one and check for the physical plan. The plan shows four steps, and we have two project operations in the SQL plan. So this one is the plan for the two consecutive withColumn() methods.

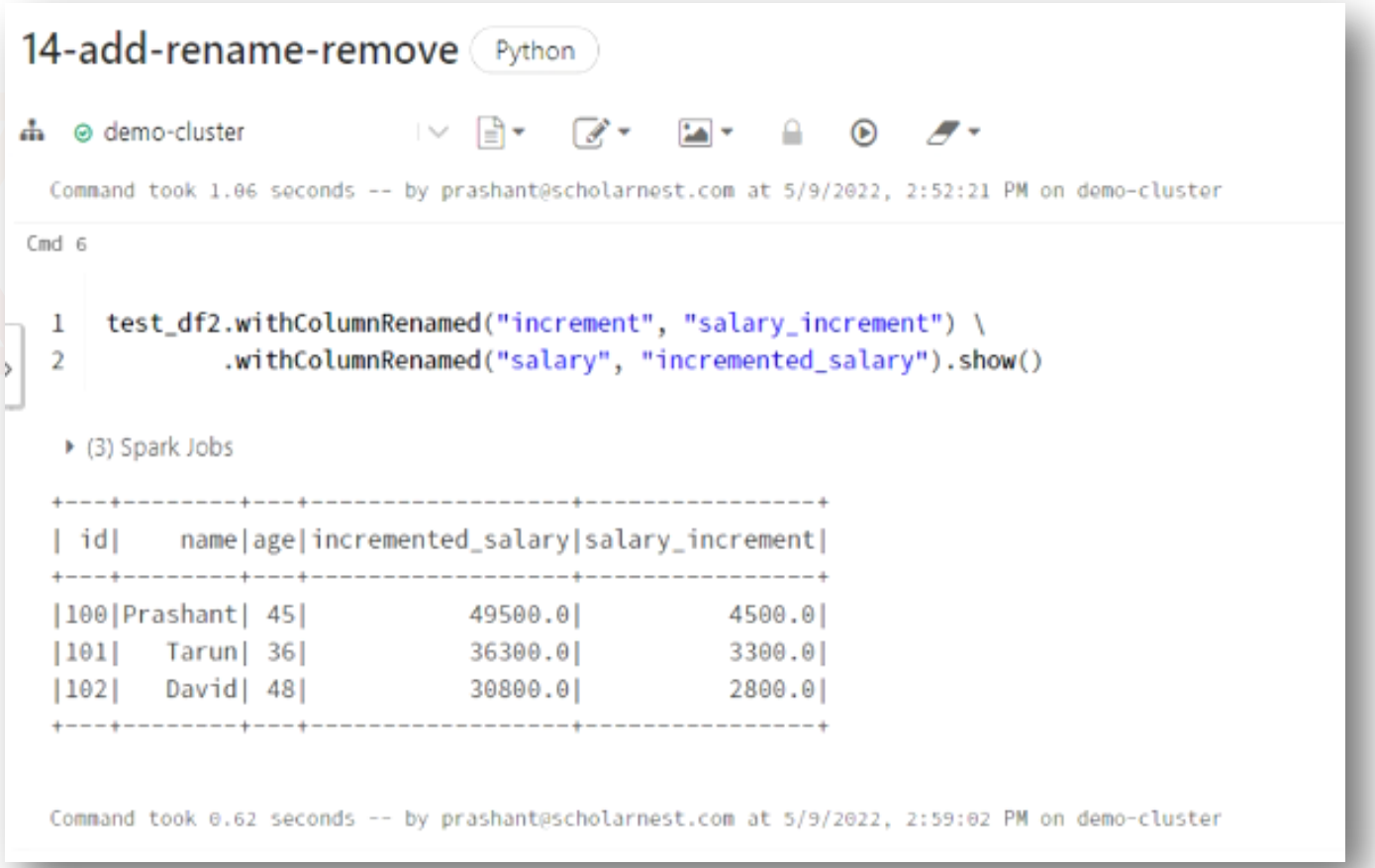


Come to the second one and check for the physical plan. This plan shows only three steps and only one project operation. So the select() method worked faster. Every withColumn() adds one project operation in the SQL plan and slows down the SQL. So we recommend combining multiple withColumn() methods with a single select() method. This approach will cut down extra projections and speed up your SQL.



Spark dataframe gives you the withColumnRenamed() method to rename an existing column. Here is an example shown below. I am using a chain of two withColumnRenamed() methods to rename two columns. You should note two things here.

1. The withColumnRenamed() is a no-op if the schema doesn't contain the given column name.
2. The withColumnRenamed() will also add one projection to your SQL plan. So chaining multiple withColumnRenamed() is not recommended.

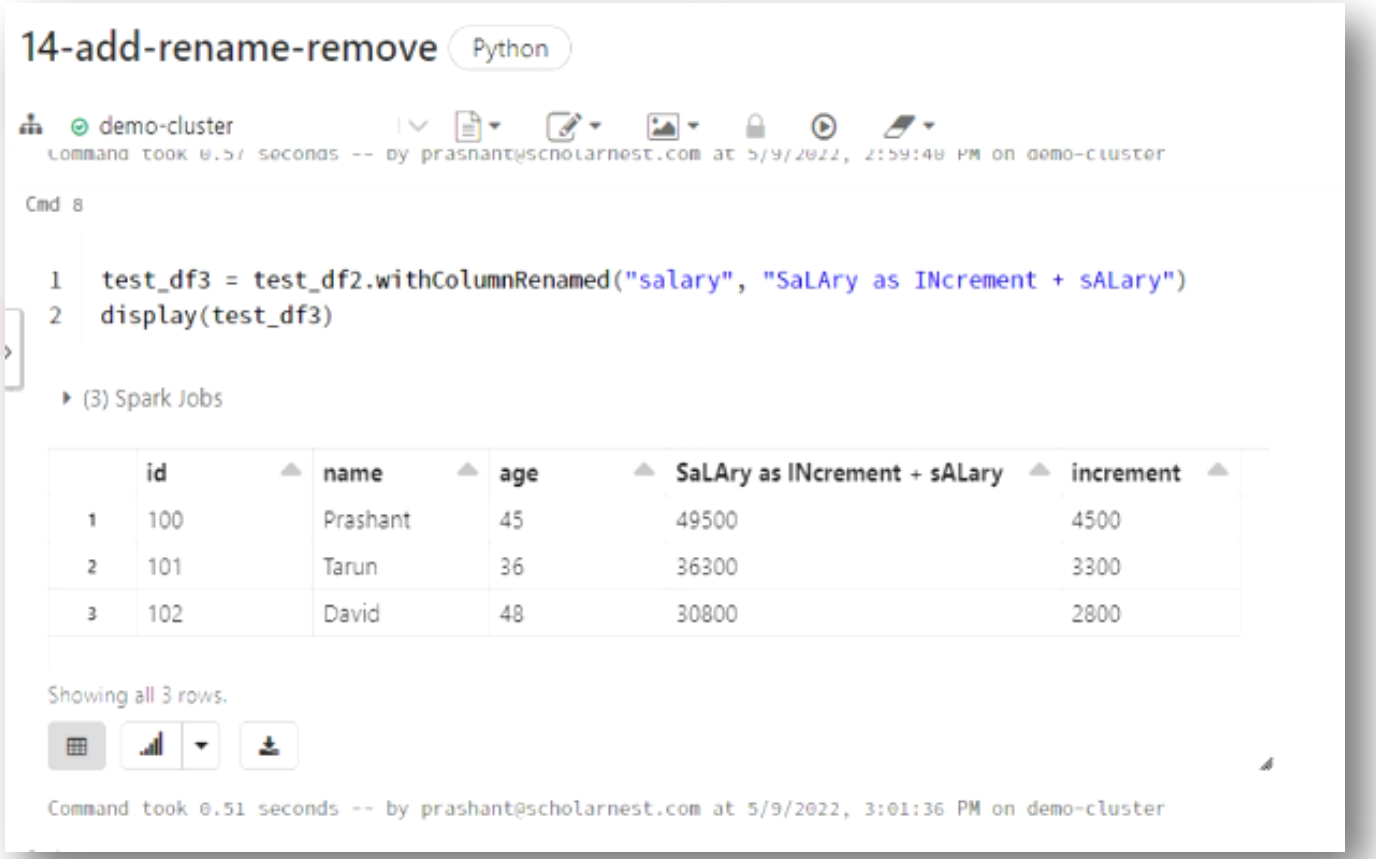


Here are some facts about column naming.

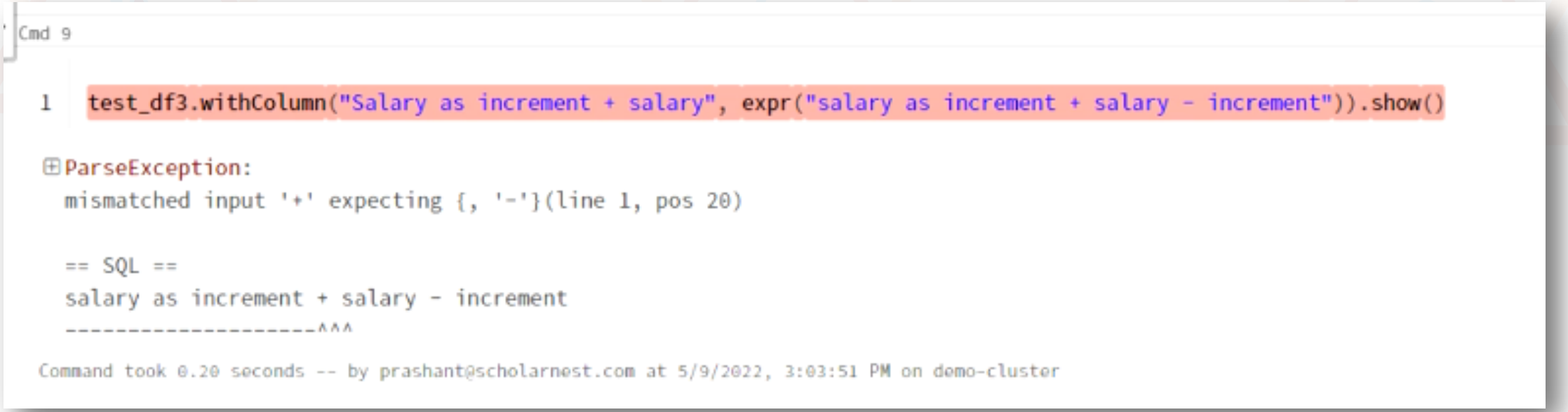
1. Dataframe Column names are case insensitive.
2. You might be able to give a wizard name with spaces and use keywords without any problem.
   But you will face problems when using those names in expressions.

In the example shown below, I renamed the salary column to a wizard name.

This new name has upper and lower case letters. We also have a keyword. The as is a keyword. I also have an arithmetic symbol in the column name. Such names are complete nonsense. But you can do it.



So you won't have any problem creating crappy column names.
But you face a problem when you want to use them in an expression. In the code shown below, I want to subtract increment from the "salary as increment + salary." I am using all small case letters in the column name. And that's fine because Spark Dataframe column names are not case-sensitive. But we get an error.

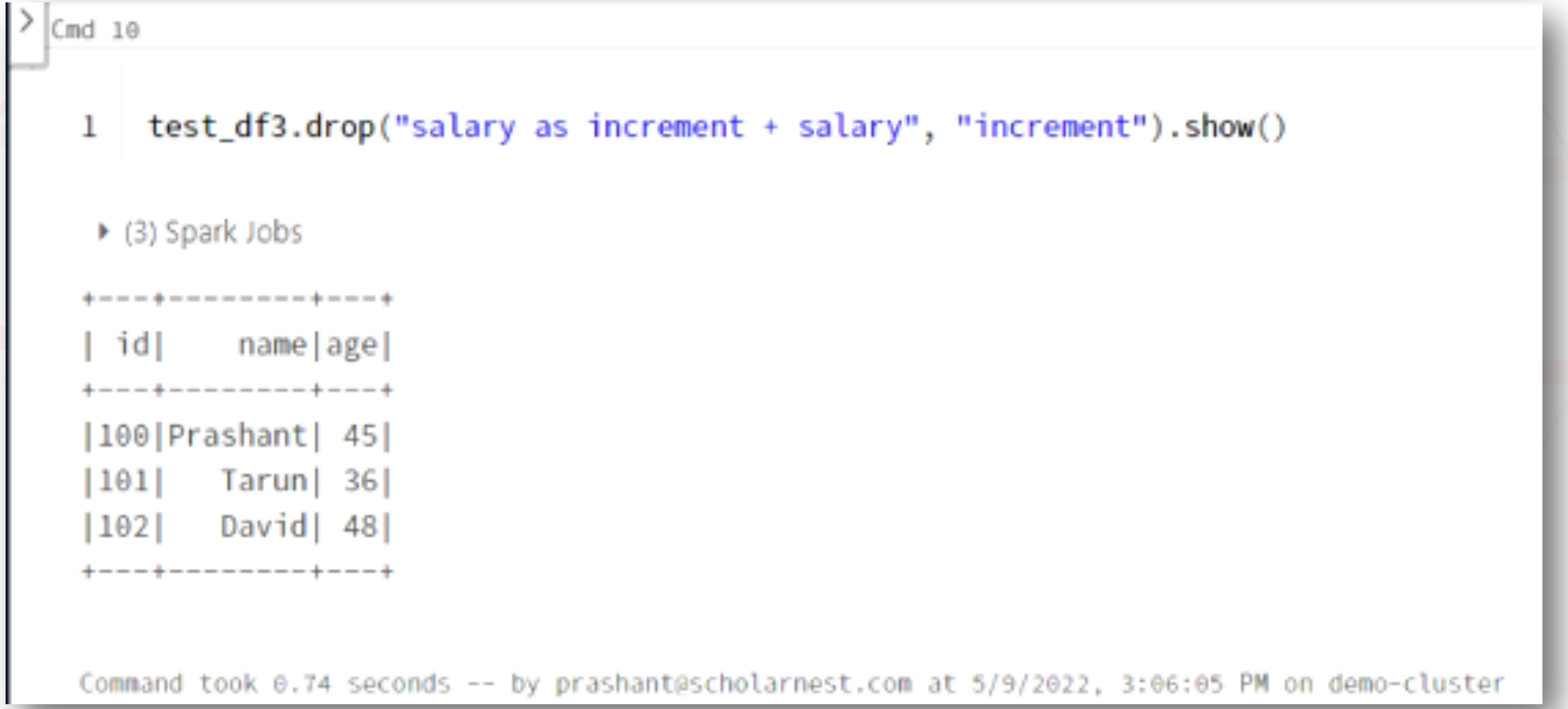


You should not name your columns in such away.
The recommendation is to follow a standard naming convention and avoid giving such weird names.
But if you do, you can still use them in the expression. Enclose your column names in a pair of backtick characters. You will find the backtick character below the escape key on your keyboard. And you can see that it works.

```
Cmd 9

1  test_df3.withColumn("Salary as increment + salary", expr("`salary as increment + salary` - increment")).show()

▸ (3) Spark Jobs

+---+--------+---+---------------------------+---------+
| id|    name|age|Salary as increment + salary|increment|
+---+--------+---+---------------------------+---------+
|100|Prashant| 45|                    45000.0|   4500.0|
|101|   Tarun| 36|                    33000.0|   3300.0|
|102|   David| 48|                    28000.0|   2800.0|
+---+--------+---+---------------------------+---------+

Command took 0.95 seconds -- by prashant@scholarnest.com at 5/9/2022, 3:04:52 PM on demo-cluster
```

So the point is straight.
Do not create column names with spaces, keywords, and symbols.
But you may still see such column names when you read data from external systems and sources.
Spark allows you to enclose such column names in a pair of backtick and use them in your expressions.
But you should rename such columns and give them a meaningful standard name.

To delete or remove a column from your Dataframe. Spark gives you a drop() method.
Here in this example, I want to drop two columns from the test_df3 Dataframe.
So I can use the drop() method and pass a comma-separated list of columns.
It will drop all the given columns and give you a new Dataframe.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

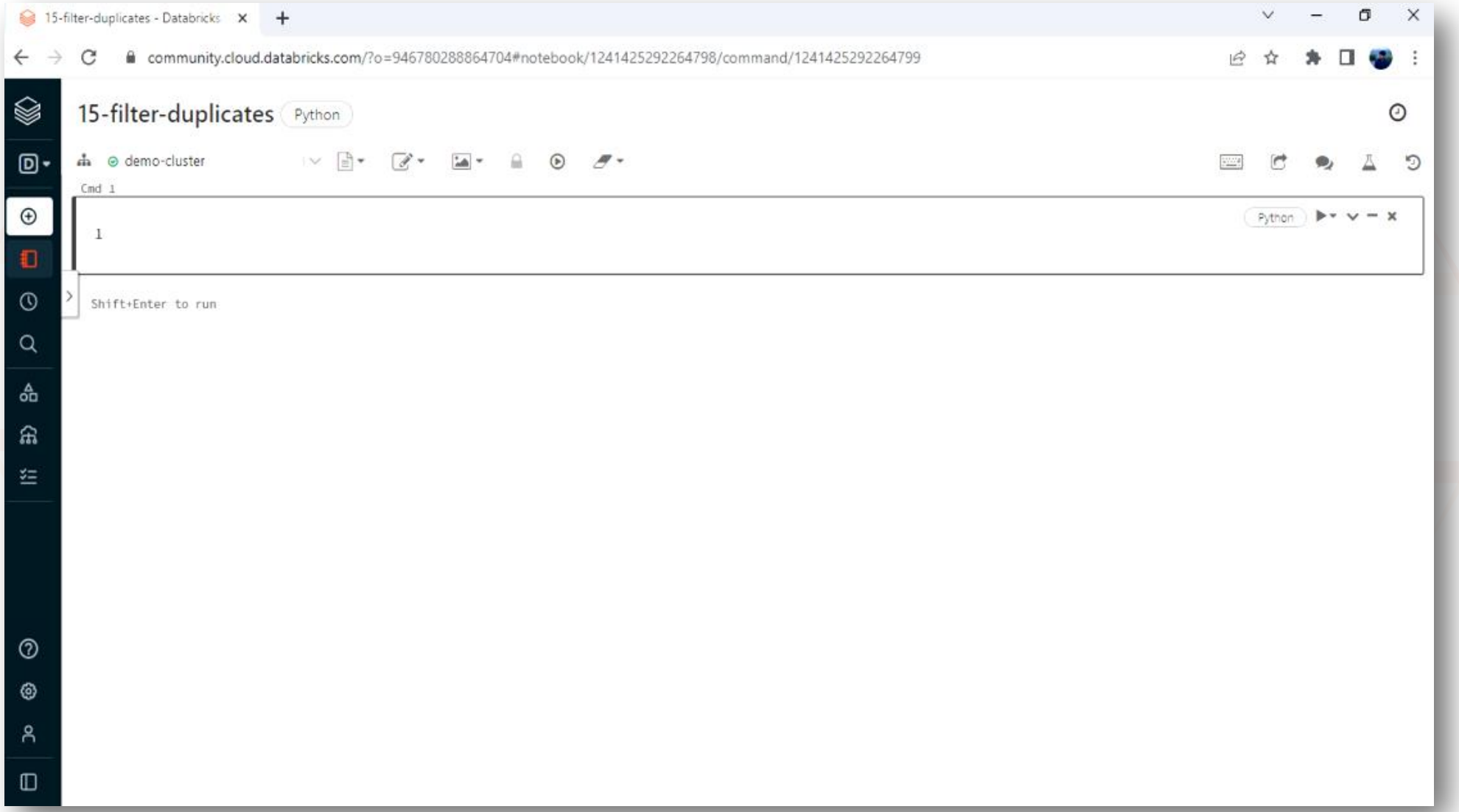
Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Filtering Records and Finding Unique Rows

In this chapter, we will talk about the following:

1. How to filter records from your Spark Dataframe
2. How to eliminate duplicates from your Dataframe

Go to your Databricks workspace and create a new notebook **(Reference : 15-filter-duplicates)**



Here we have a Dataframe. Now let's assume we want to filter records where the source column starts with Mum.

15-filter-duplicates Python

demo-cluster

```
8
9  df = spark.createDataFrame(data_list) \
10         .toDF("id", "source", "destination", "distance")
11
12 display(df)
```

(3) Spark Jobs

	id	source	destination	distance
1	101	Mumbai	Goa	587
2	102	Mumbai	Bangalore	985
3	102	Mumbai	Bangalore	985
4	103	Delhi	Chennai	2208
5	104	Delhi	Chennai	2208
6	105	Bangalore	Kolkata	1868
7	105	Bangalore	Kolkata	1865

Showing all 7 rows.

Command took 6.19 seconds -- by prashant@scholarnest.com at 5/11/2022, 8:33:04 PM on demo-cluster

To filter records where the source column starts with Mum, we can use the following SQL command.

```
SELECT * FROM table_name
WHERE source like 'Mum%'
```

How to do it in a dataframe?

Spark gives you two methods.

1. filter()
2. where()

The filter method allows you to filter records in the dataframe.
And the where() method is an alias of the filter() method.
So both are the same.

The filter condition is the same as the condition in a SQL expression. Then, I am chaining one more condition using the where method. There is no difference between filter() and where() because the where() method is just an alias of the filter() method.

Cmd 2

```
1  df.filter("source like 'Mum%'") \
2      .where("destination like 'Ban%'") \
3      .show()
```

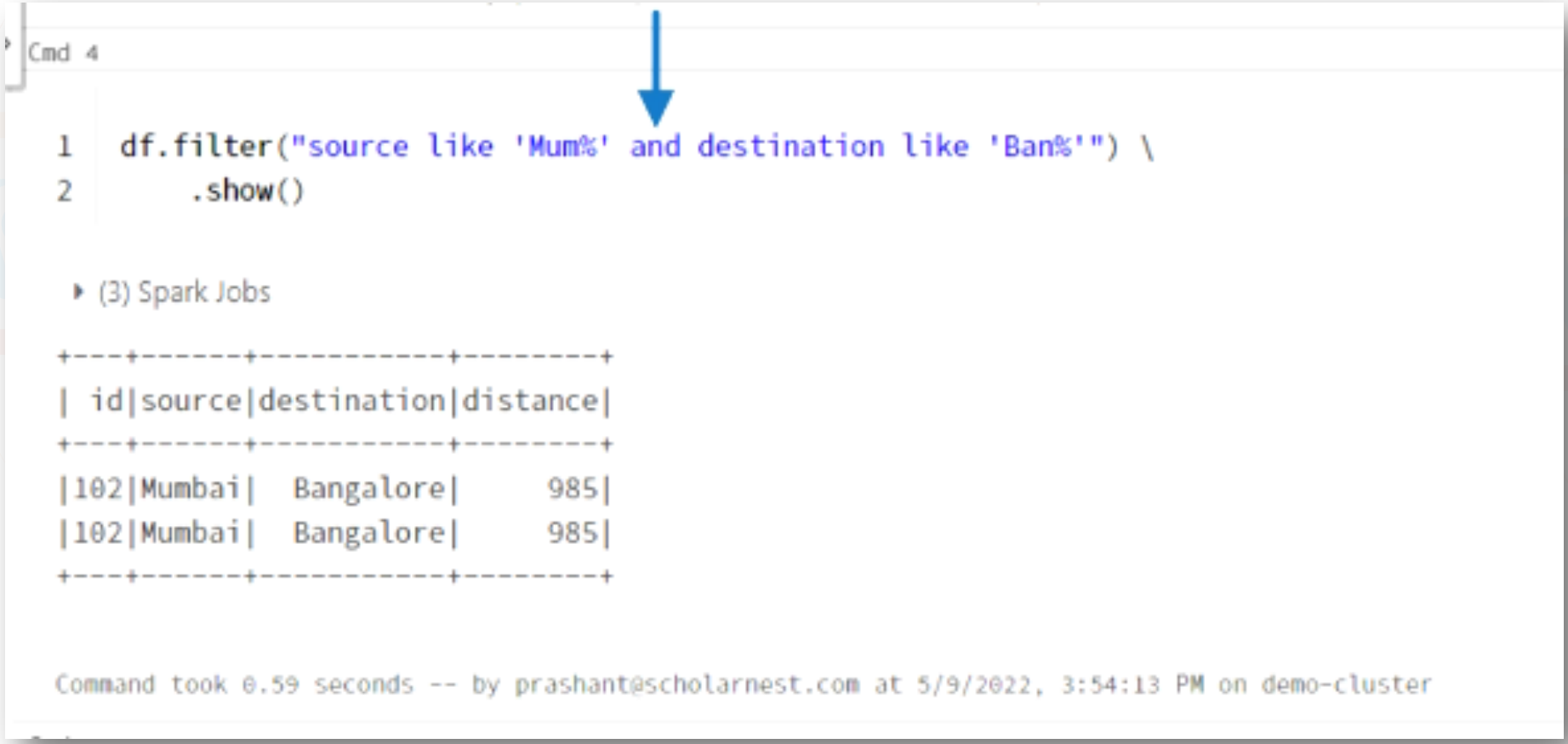
▸ (3) Spark Jobs

```
+---+------+-----------+--------+
| id|source|destination|distance|
+---+------+-----------+--------+
|102|Mumbai|  Bangalore|     985|
|102|Mumbai|  Bangalore|     985|
+---+------+-----------+--------+
```

Command took 0.79 seconds -- by prashant@scholarnest.com at 5/11/2022, 8:51:47 PM on demo-cluster

I used SQL conditions in the filter and where methods. And that's the most convenient method. But the filter method also allows you to specify conditions using functions and column API. This approach to writing expression is complex and confusing for SQL developers.
And there is no harm in taking the SQL approach and defining your expressions.
But learning the functional approach is good to have even though you rarely use it in practice. So this code takes the source column and applies column API to add the like() condition.

We can also use the previous example like this. I am explicitly using the AND operator between two conditions.
And this approach is more intuitive.
So we recommend using conditions explicitly.



Column APIs also allow using the and condition. The syntax is a little different as shown below. So we use a single ampersand between two conditions.

Cmd 5

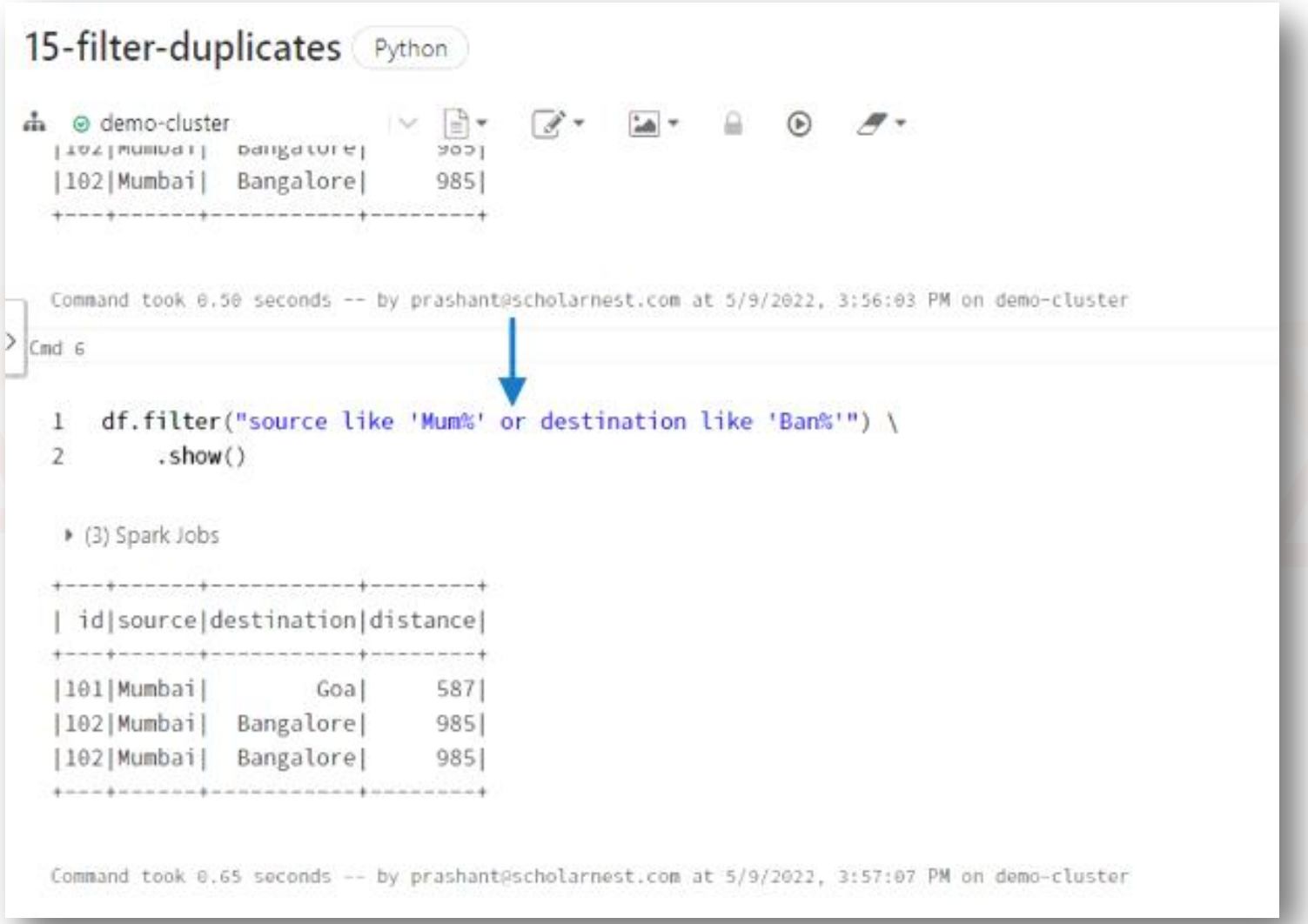
```
1  from pyspark.sql.functions import col
2
3  df.filter(col("source").like("Mum%") & col("destination").like("Ban%")) \
4      .show()
```

▸ (3) Spark Jobs

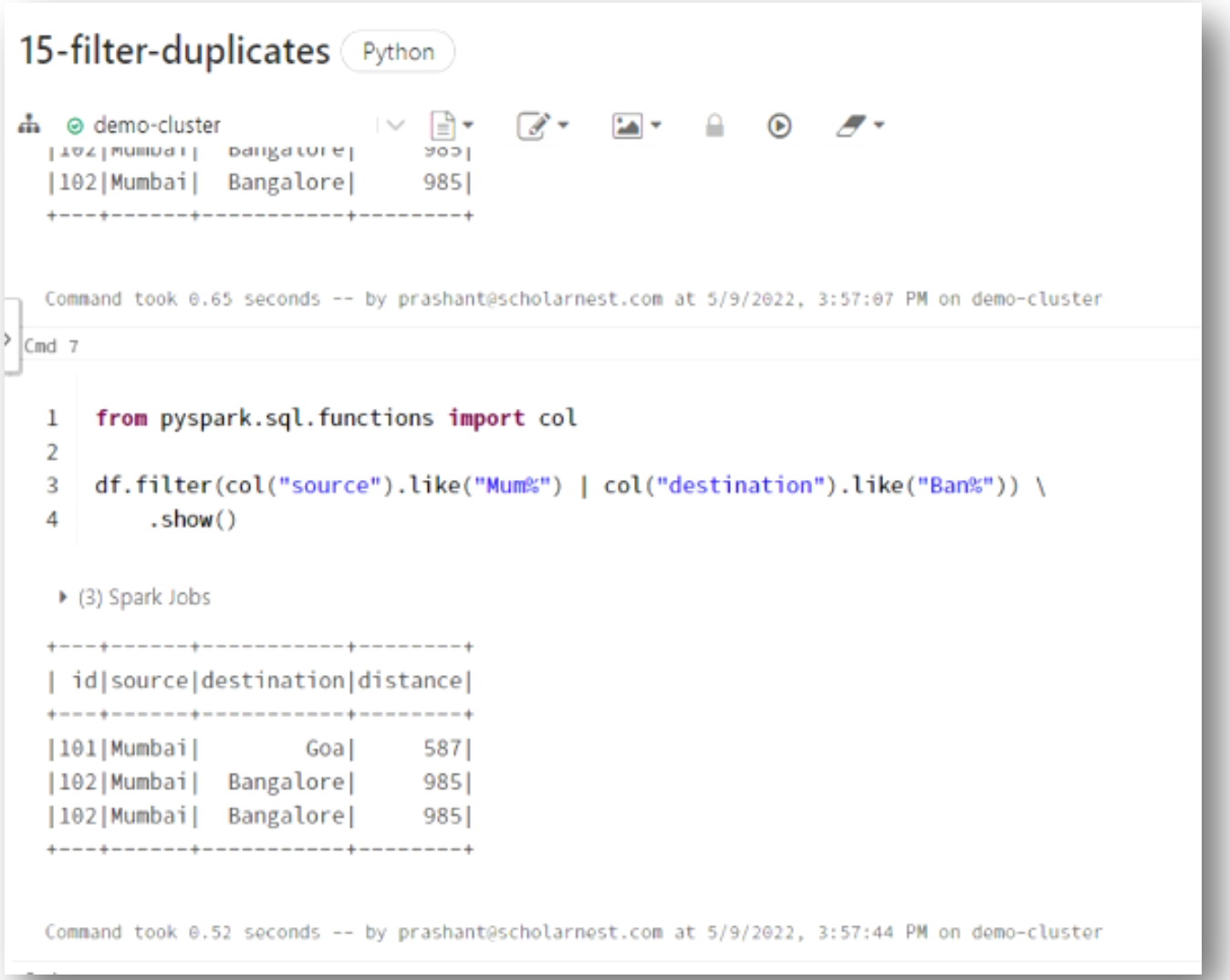
```
+---+------+-----------+--------+
| id|source|destination|distance|
+---+------+-----------+--------+
|102|Mumbai|  Bangalore|     985|
|102|Mumbai|  Bangalore|     985|
+---+------+-----------+--------+
```

Command took 0.50 seconds -- by prashant@scholarnest.com at 5/9/2022, 3:56:03 PM on demo-cluster

You can also use OR condition as shown below.



We can change the ampersand to a pipe for using the OR condition in column API.



Here is the Dataframe shown below. And we have few problems for duplicity in this Dataframe. Let's try it one by one.

Cmd 8

```
1  df.show()
```

▸ (3) Spark Jobs

```
+---+---------+-----------+--------+
| id|   source|destination|distance|
+---+---------+-----------+--------+
|101|   Mumbai|        Goa|     587|
|102|   Mumbai|  Bangalore|     985|
|102|   Mumbai|  Bangalore|     985|
|103|    Delhi|    Chennai|    2208|
|104|    Delhi|    Chennai|    2208|
|105|Bangalore|    Kolkata|    1868|
|105|Bangalore|    Kolkata|    1865|
+---+---------+-----------+--------+
```

Command took 0.65 seconds -- by prashant@scholarnest.com at 5/11/2022, 9:26:10 PM on demo-cluster

Cmd 9

102 - Entire row is duplicate
105 - Record ID is duplicate
103/104 - ID is different, but other fields are duplicate

Here is the solution to the first problem. So one record is gone.
You can see only one record for 102.
The distinct() method is similar to the DISTINCT in SQL queries.
If your record is duplicated across all the columns, you can use the distinct() method.

We also have one more alternative for the first problem, the dropDuplicates() method. So the result of the dropDuplicates() and the distinct() is the same.

Cmd 10

```
1  df.dropDuplicates().show()
```

▸ (2) Spark Jobs

```
+---+---------+-----------+--------+
| id|   source|destination|distance|
+---+---------+-----------+--------+
|101|   Mumbai|        Goa|     587|
|102|   Mumbai|  Bangalore|     985|
|103|    Delhi|    Chennai|    2208|
|104|    Delhi|    Chennai|    2208|
|105|Bangalore|    Kolkata|    1868|
|105|Bangalore|    Kolkata|    1865|
+---+---------+-----------+--------+
```

Command took 0.68 seconds -- by prashant@scholarnest.com at 5/11/2022, 9:27:28 PM on demo-cluster

Now, look at the second problem. The record id 105 is duplicate, but other column values are different. This one is not a whole record duplication. So the distinct() method does not help. However, the dropDuplicates() method is more powerful. It can take a list of columns and eliminate duplicates based on those columns as shown below. So the drop dropDuplicates() will look at the id column and remove all records where the id is duplicate. And this guy solves two problems in one shot. It eliminated a duplicate of 102 and also 105. The 102 was a complete duplicate. That one is gone because the id was also duplicated. The 105 is fine because the id was duplicated.

Now let's go to the third problem.
103 and 104 are duplicates, but the ID is different.
How do we eliminate it? We can use the dropDuplicates() once again. But this time, we want to remove duplicates on all columns except the id column. Let me add another dropDuplicates() as shown below.

Cmd 11

```
1  df.dropDuplicates(["id"]) \
2      .dropDuplicates(["source", "destination", "distance"]) \
3      .show()
```

▸ (3) Spark Jobs

```
+---+---------+-----------+--------+
| id|   source|destination|distance|
+---+---------+-----------+--------+
|102|   Mumbai|  Bangalore|     985|
|105|Bangalore|    Kolkata|    1868|
|101|   Mumbai|        Goa|     587|
|103|    Delhi|    Chennai|    2208|
+---+---------+-----------+--------+
```

Command took 2.21 seconds -- by prashant@scholarnest.com at 5/11/2022, 9:28:56 PM on demo-cluster

So we learned two methods to remove duplicates.

1. distinct()
2. dropDuplicates()

The distinct method is similar to DISTINCT in the SQL, and it does not support complex scenarios.

The dropDuplicates() is more powerful. It takes a list of columns, compares records only using the given columns, and eliminates duplicates.

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com