ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

Absolute Beginner to Specialization in Apache Spark and Azure Databricks

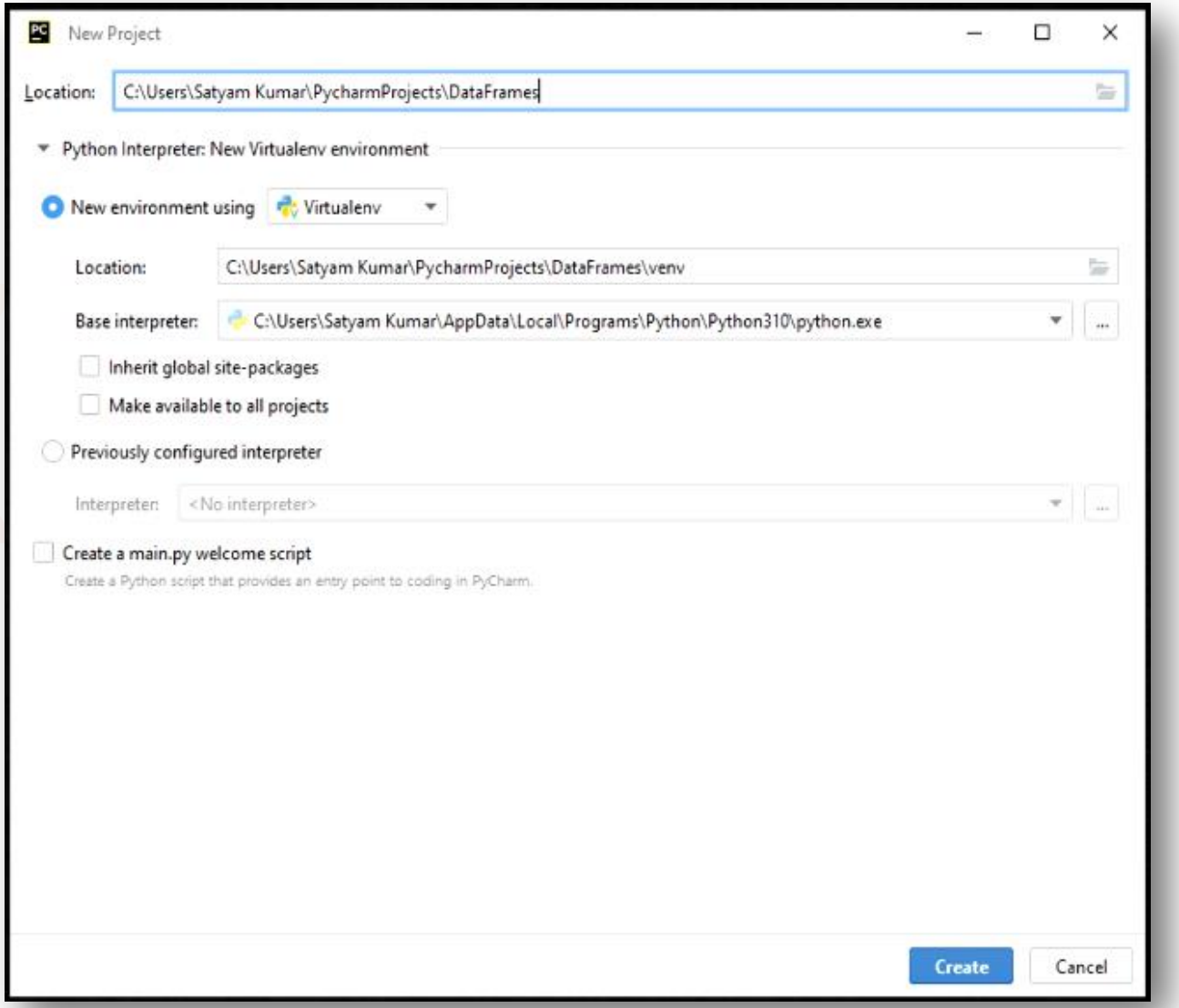
Creating Spark Session

Assume you are given a task to create a small Spark job to do the following:

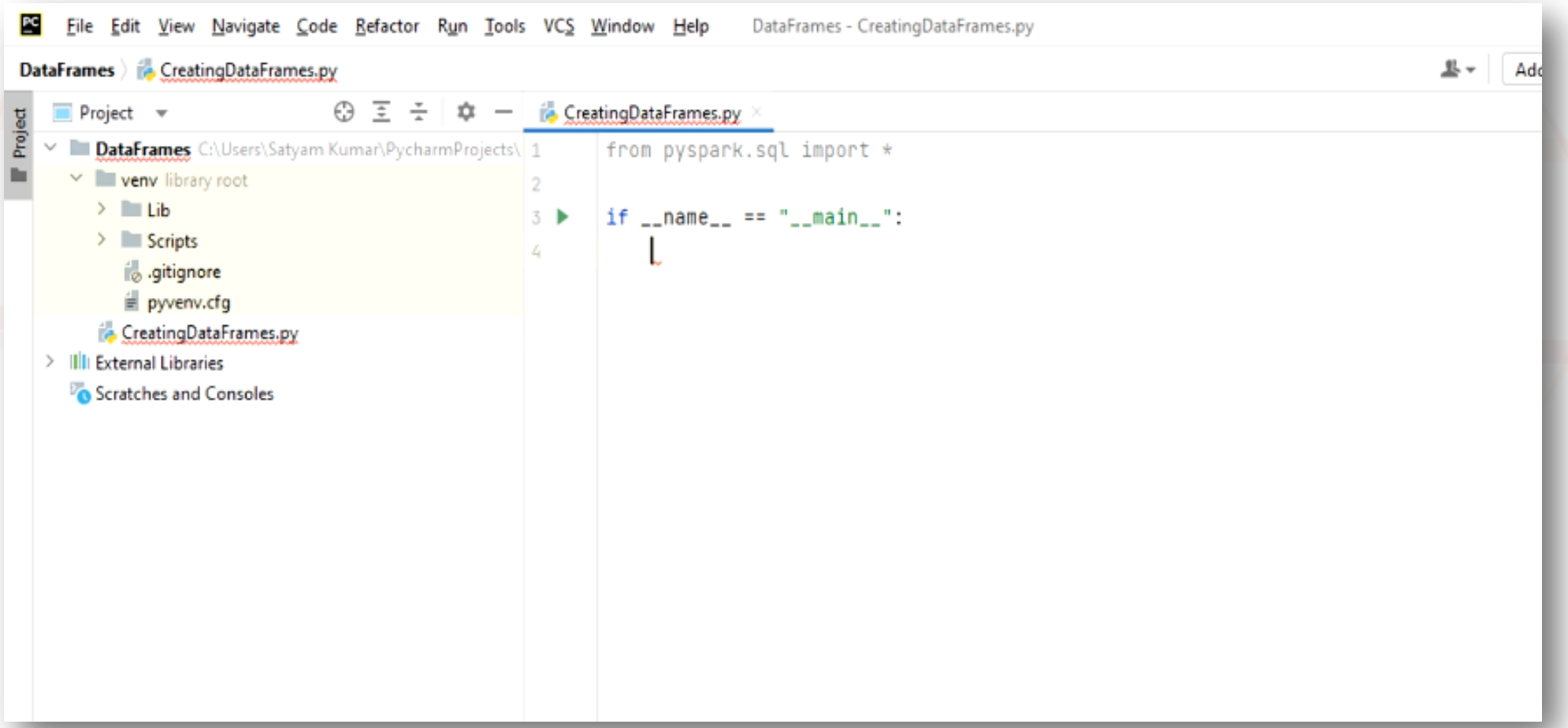
Requirement

1. Read a CSV data file
2. Perform some standard data validation and data type correction to prepare it for analysis
3. Save the valid data using the parquet data file format

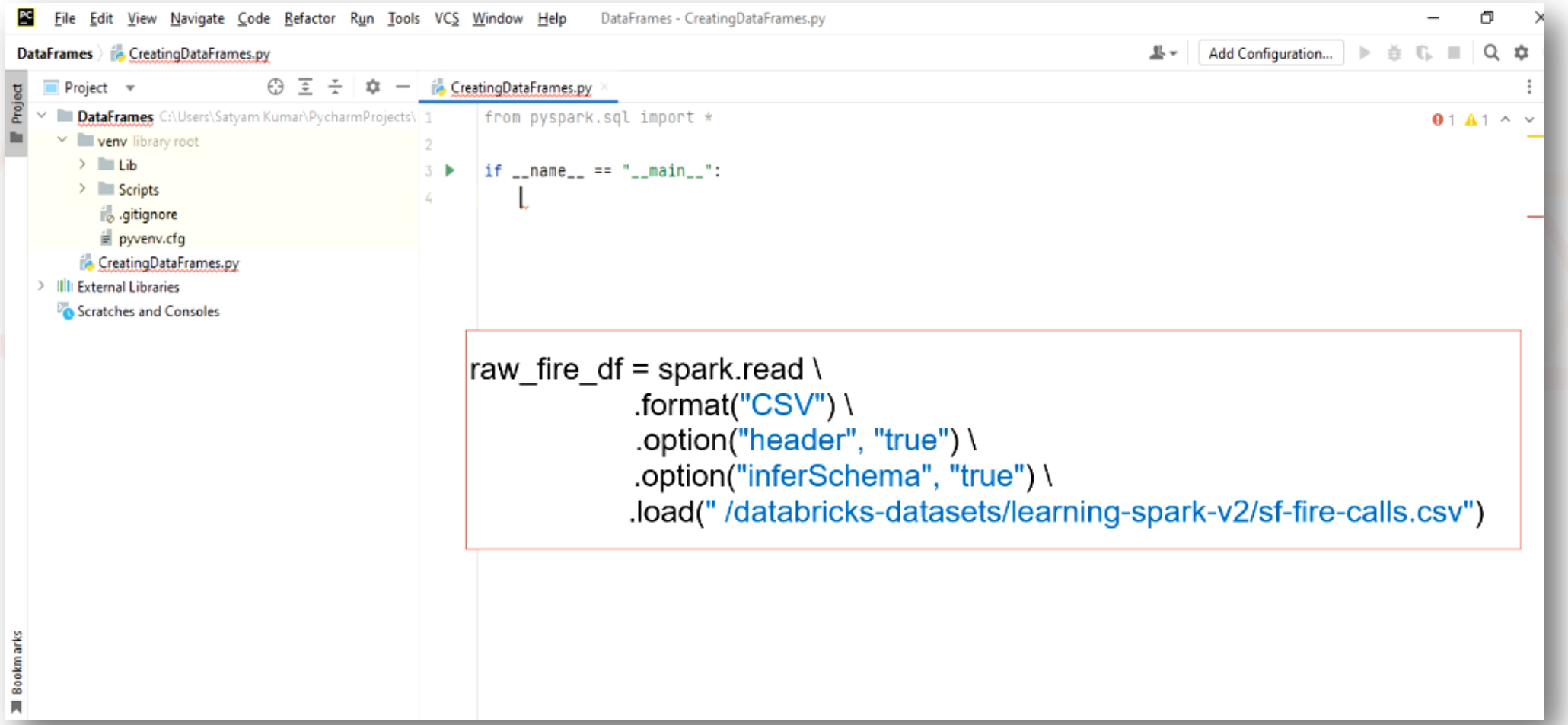
Start your PyCharm IDE and create a new project. Give a name to your project and create it. **(Reference : DataFrames)**



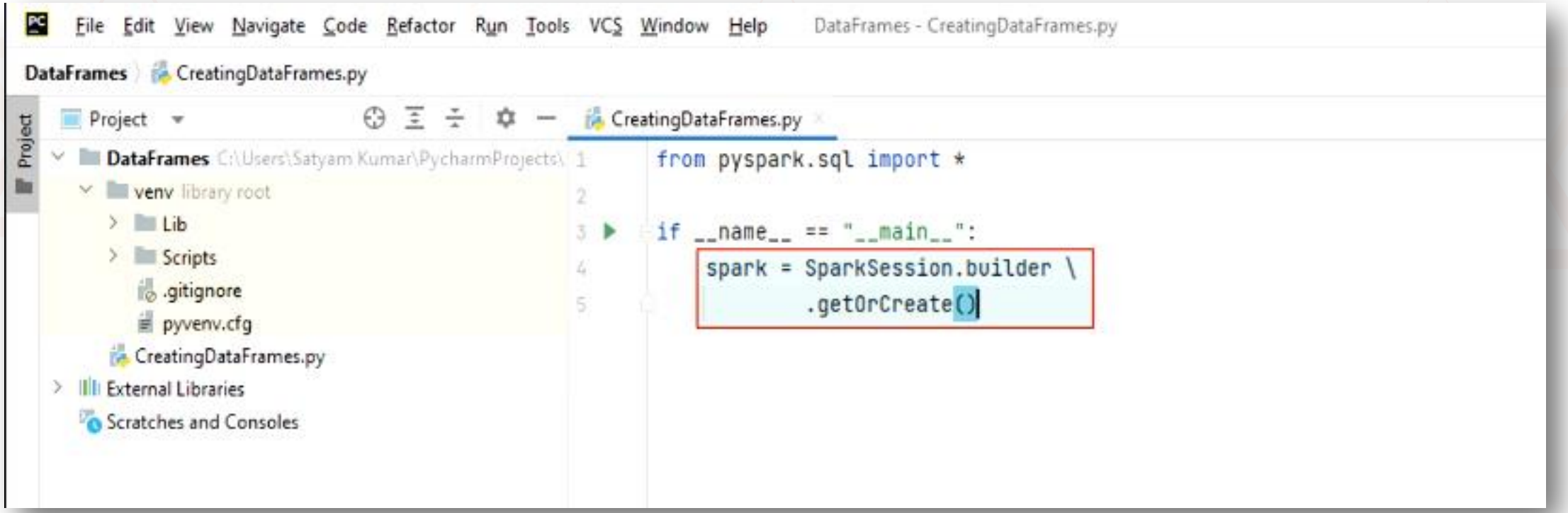
Create a new Python file in your project. We will be using the PySpark SQL package so let me import the package. I am starting a new Spark project, so I need an entry point for my application. So I have defined the main method as shown below.



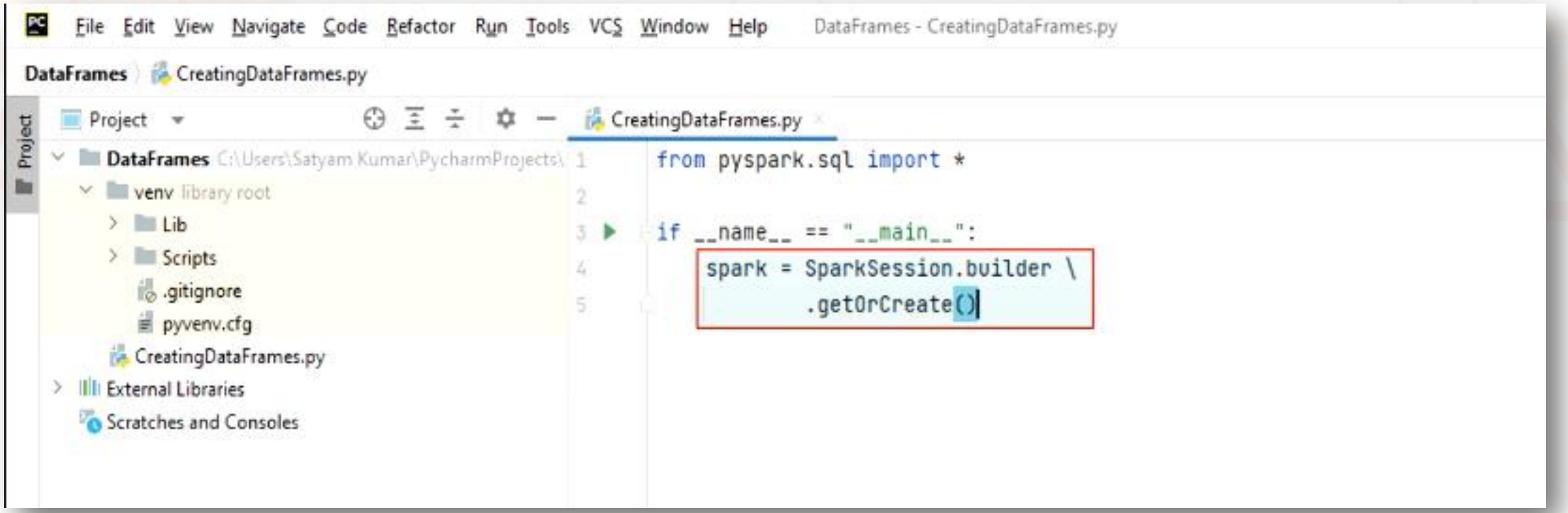
The first requirement is to read a CSV data file. How will you do it? Here it is. We used this code to read a CSV file and create a Spark DataFrame. So we can use the same code template here.



But we have a small problem. We do not have the Spark Session variable.
I created the earlier example in a Databricks Notebook. The notebook environment automatically creates a Spark session object and makes it available for us. But when you are creating a Spark program in a Python file, you must create the SparkSession yourself as shown below.



But we have a small problem. We do not have the Spark Session variable.
I created the earlier example in a Databricks Notebook. The notebook environment automatically creates a Spark session object and makes it available for us. But when you are creating a Spark program in a Python file, you must create the SparkSession yourself as shown below. The code shown here is the bare minimum to start writing the Spark program in a python file.



Spark Session is your entry point for Spark programming, and it offers you the following attributes and methods.

SparkSession

Attributes

`builder`	A class attribute having a `Builder` to construct `SparkSession` instances.
`read`	Returns a `DataFrameReader` that can be used to read data in as a `DataFrame`.
`readStream`	Returns a `DataStreamReader` that can be used to read data streams as a streaming `DataFrame`.
`catalog`	Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc.
`conf`	Runtime configuration interface for Spark.
`udf`	Returns a `UDFRegistration` for UDF registration.
`sparkContext`	Returns the underlying `SparkContext`.
`version`	The version of Spark on which this application is running.

Methods

`createDataFrame`(data[, schema, ...])	Creates a `DataFrame` from an `RDD`, a list or a `pandas.DataFrame`.
`getActiveSession`()	Returns the active `SparkSession` for the current thread, returned by the builder.
`newSession`()	Returns a new `SparkSession` as new session, that has separate SQLConf, registered temporary views and UDFs, but shared `SparkContext` and table cache.
`range`(start[, end, step, numPartitions])	Create a `DataFrame` with single `pyspark.sql.types.LongType` column named `id`, containing elements in a range from `start` to `end` (exclusive) with step value `step`.
`sql`(sqlQuery)	Returns a `DataFrame` representing the result of the given query.
`stop`()	Stop the underlying `SparkContext`.
`table`(tableName)	Returns the specified table as a `DataFrame`.

Let's focus on the attributes for now. The first one is the builder. Once you have the session builder, you can use builder methods to create and configure your spark session. And here is the list of session builder methods.
So we have five session builder methods. The appName allows you to set a unique name for your Spark application.

SparkSession

Attributes

`builder`	A class attribute having a `Builder` to construct `SparkSession` instances.
`read`	Returns a `DataFrameReader` that can be used to read data in as a `DataFrame`.
`readStream`	Returns a `DataStreamReader` that can be used to read data streams as a streaming `DataFrame`.
`catalog`	Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc.
`conf`	Runtime configuration interface for Spark.
`udf`	Returns a `UDFRegistration` for UDF registration.
`sparkContext`	Returns the underlying `SparkContext`.
`version`	The version of Spark on which this application is running.

Builder Methods

`SparkSession.builder.appName`(name)	Sets a name for the application, which will be shown in the Spark web UI.
`SparkSession.builder.master`(master)	Sets the Spark master URL to connect to, such as "local" to run locally, "local[4]" to run locally with 4 cores, or "spark://master:7077" to run on a Spark standalone cluster.
`SparkSession.builder.config`([key, value, conf])	Sets a config option.
`SparkSession.builder.enableHiveSupport`()	Enables Hive support, including connectivity to a persistent Hive metastore, support for Hive SerDes, and Hive user-defined functions.
`SparkSession.builder.getOrCreate`()	Gets an existing `SparkSession` or, if there is no existing one, creates a new one based on the options set in this builder.

The next one is to set the location of the master node of your Spark cluster. I do not recommend using this method at all. You should never set the master from your code.
Why? Because you will run your code on your local machine or in the development cluster. Then you want to run the same application in a testing cluster. Finally, the code goes to the production cluster.
The master node location is different everywhere. So we do not hardcode it in the application. Instead, we set the master node location from outside the application code.

The config method allows you to configure your spark session.
The next one is to enable Hive support in your Spark application. Enabling Hive support allows you to access Hive database tables from Spark. You should enable this option when your target is the Hadoop cluster. But this option is not meaningful in the Databricks Cloud environment. Because you do not have Hadoop and Hive in Databricks Cloud.
The last and the most important one is to getOrCreate a Spark session. Spark application must have only one active Spark session at any given point. So the getOrCreate API protects you from incorrectly creating two sessions. If you already created a SparkSession earlier, the getOrCreate will return the same session; else, it will create a new session. So even if you call the getOrCreate method more than once, you will get the same Spark session. Make sense?

We have quite a few other attributes of Spark Session.
The read and the readStream are the most important. They give you a DataFrameReader and open your gateway to the DataFrame API.

The catalog is your Spark metadata interface.
The conf is your spark runtime configuration interface.
The UDF here gives you a UDFRegistration interface.

The last one is the SparkContext.
A SparkContext represents the connection to a Spark cluster and can use to create RDD and broadcast variables on the cluster.
Memorize this definition.

The SparkContext offers you two functionalities.

1. It holds and manages a connection to your Spark cluster
2. It is your gateway to Spark Core APIs

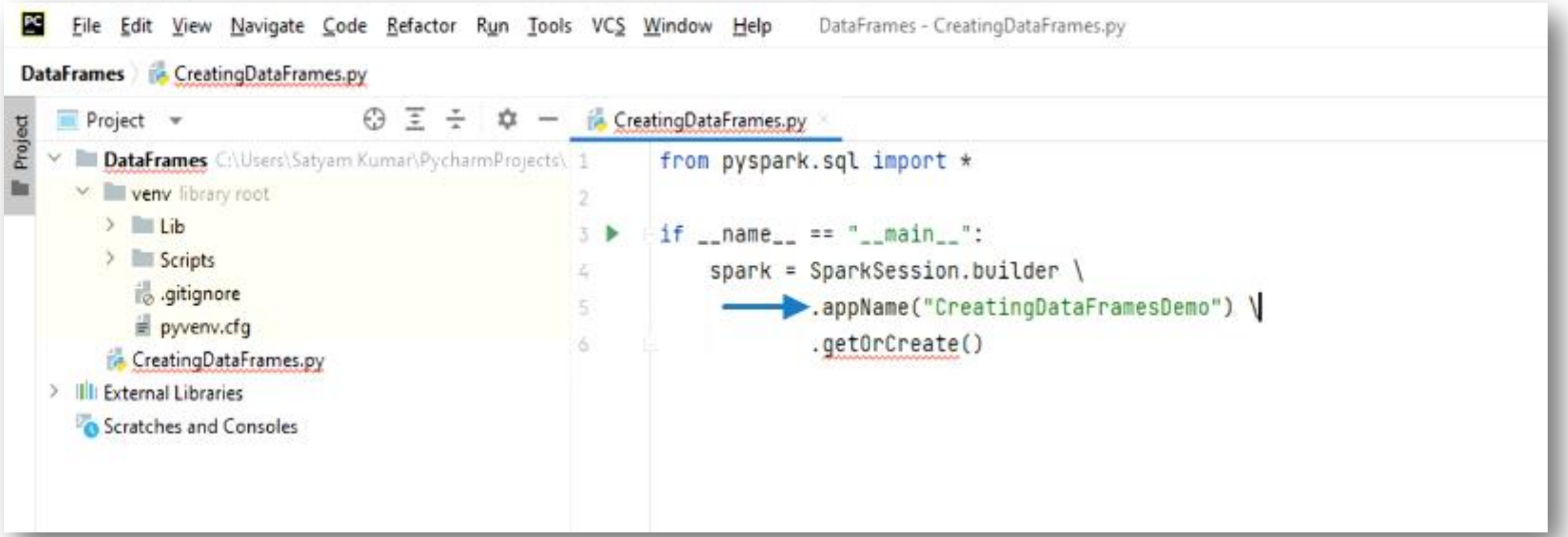
So Spark session also offers you two functionalities.

1. It is your gateway to Spark DataFrame APIs
2. It holds the SparkContext.

Now you can combine the SparkContext into SparkSession.

1. It is your gateway to Spark DataFrame APIs
2. It holds the SparkContext.
    1. It holds and manages a connection to your Spark cluster
    2. It is your gateway to Spark Core APIs

You learned to create a SparkSession.
It is the bare minimum code. But I recommend setting a unique application name.
Simply add a line just before the getOrCreate and use Session Builder's appName() method to set a Unique application name.



Summary

You always need a SparkSession.

Notebooks and command-line tools automatically create a SparkSession for you

But you must create it yourself in all other environments.

We learned the bare minimum code for creating a Spark Session

I also talked about the session builder and the SparkContext

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

Creating Data Frames Programmatically

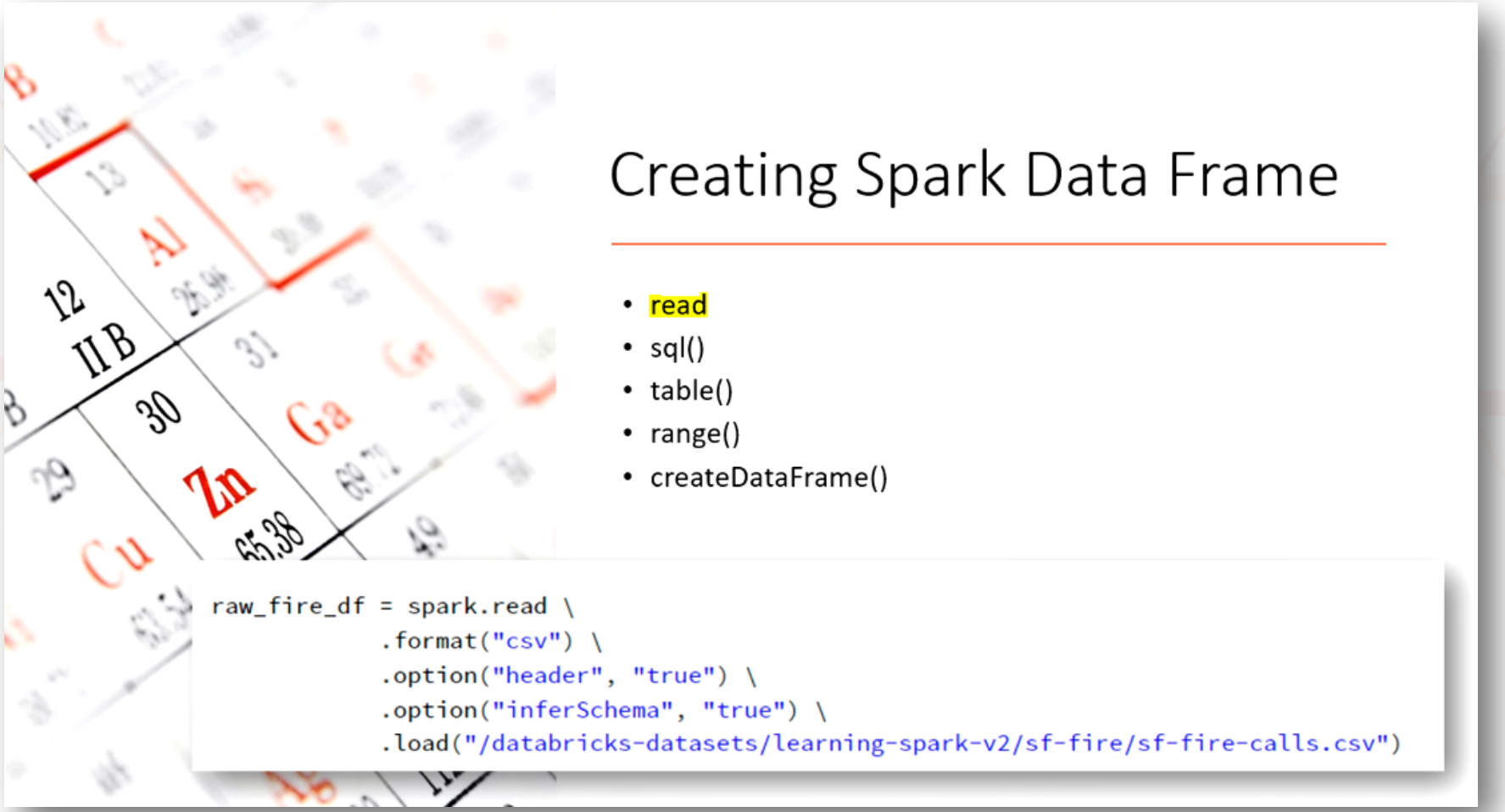
We learned that Spark session is the starting point. However, Spark programming is all about reading, processing, and writing data. And Spark Dataframe is your tool for doing all of this.
But how do you create a Spark Dataframe?

We have many ways to create a Spark Dataframe. Here is a list of some approaches:

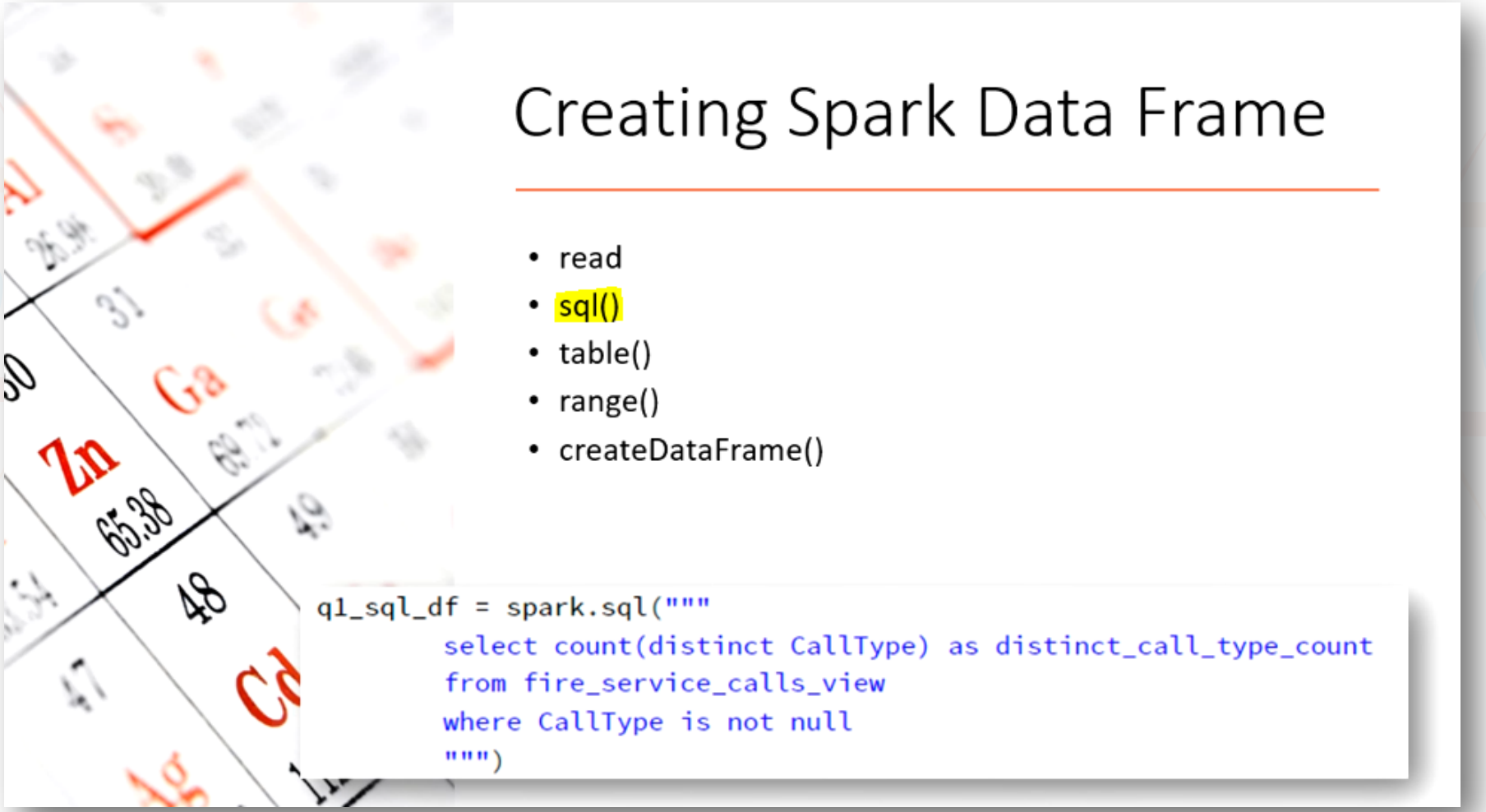
1. read
2. sql()
3. table()
4. range()
5. createDataFrame()

All of these are offered by SparkSession.

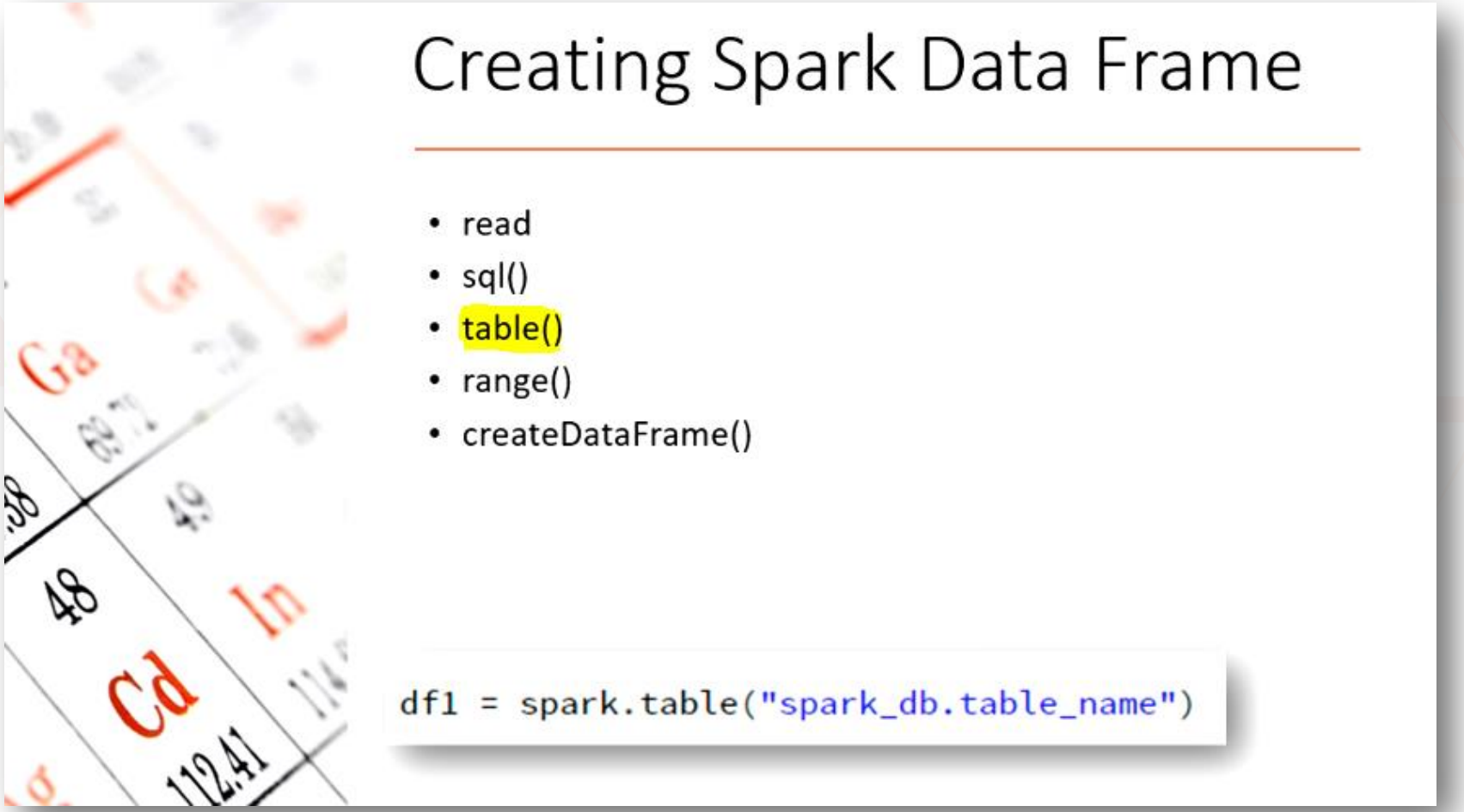
We already learned about the Spark Session read attribute. Here is the code shown below that we used in an earlier example. The *SparkSession.read* gives you a *DataframeReader*, and you can use it to read data from a variety of data sources. We learned to read data from a CSV file, but *DataframeReader* supports many other sources.



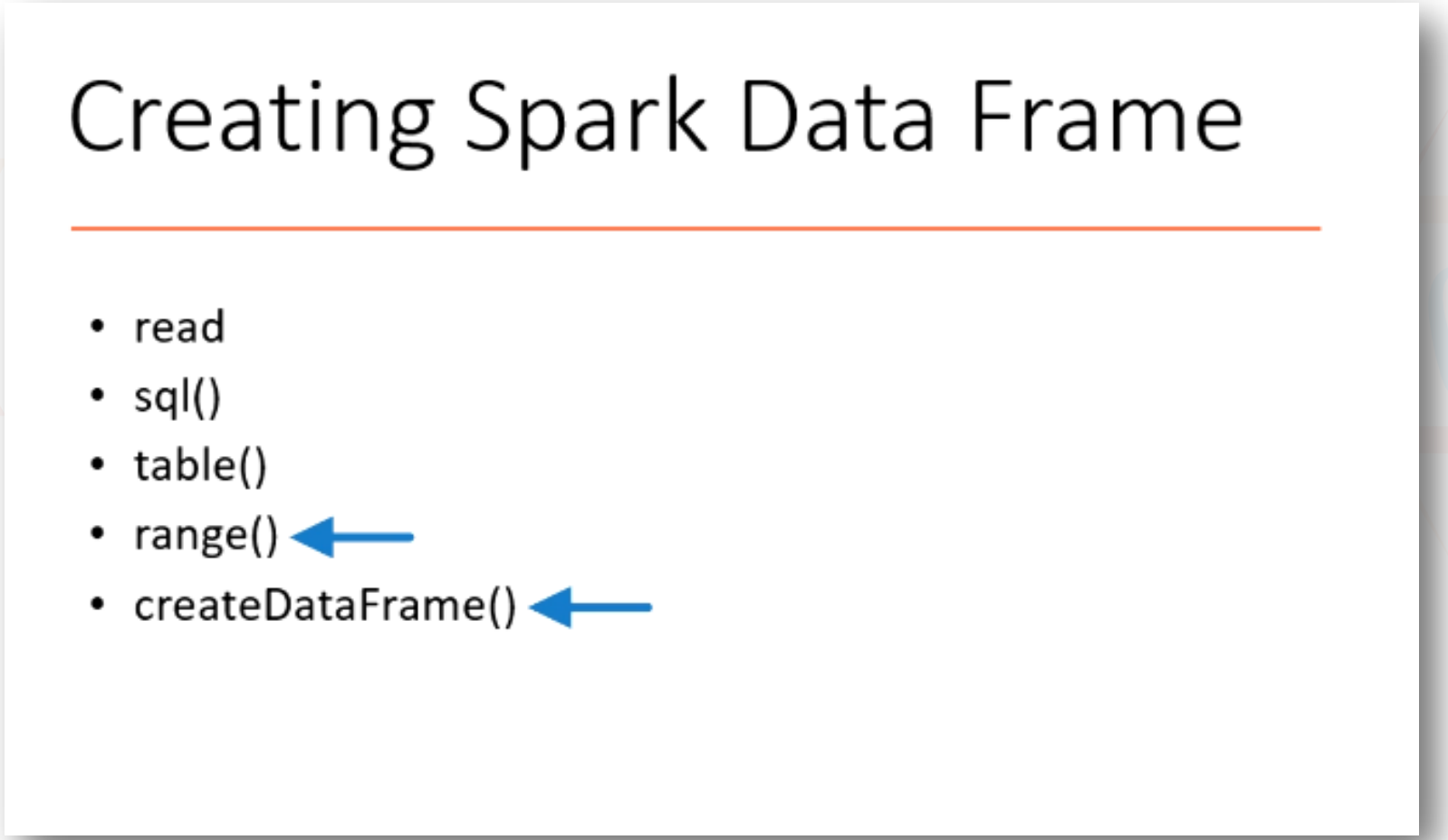
We also learned to use the *SparkSession.sql()* method. Here is the earlier example.
So you can use a Spark SQL in the *SparkSession.sql()* method, and the result of the SQL is a Spark Dataframe.



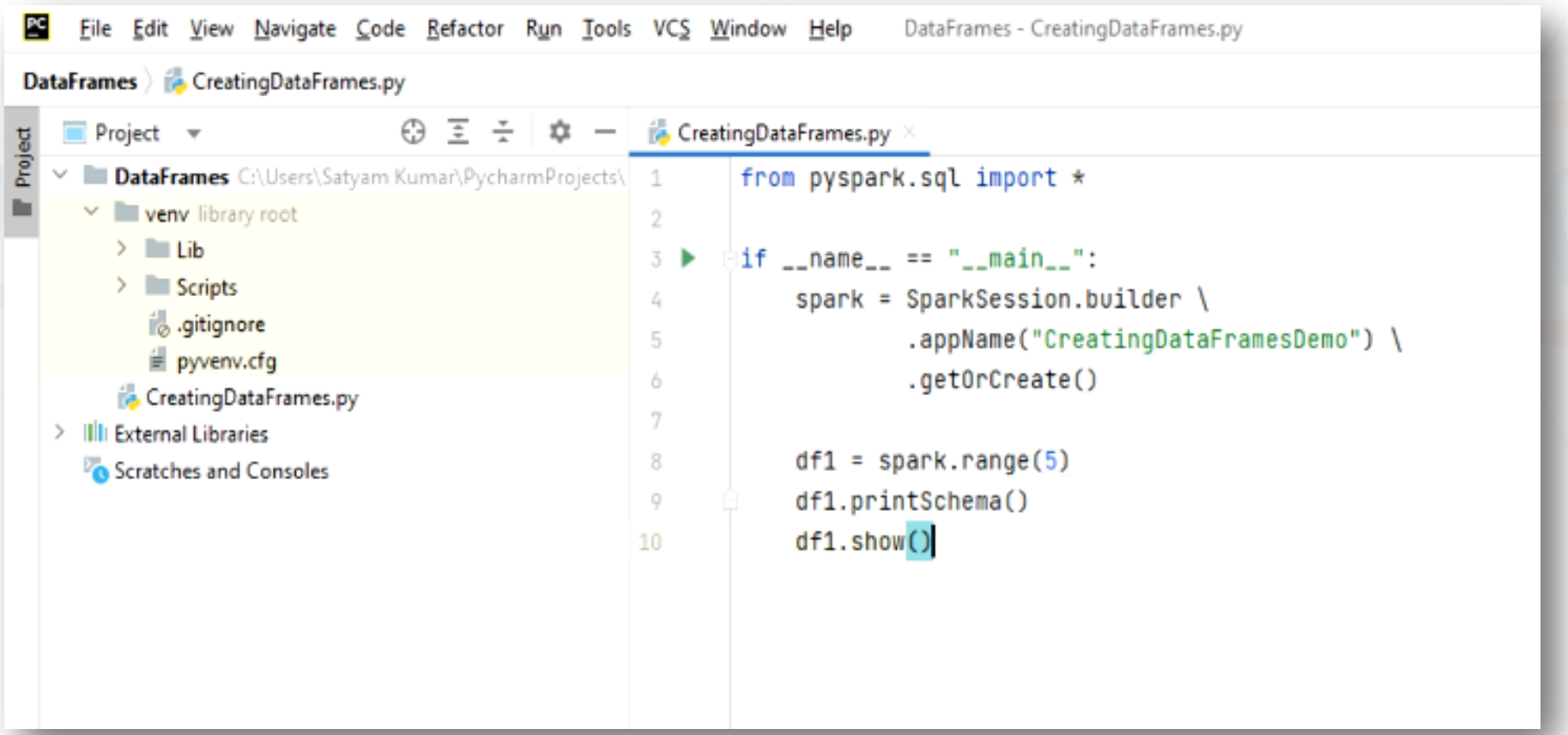
The third method is even more straightforward than the SQL() method. You can use *spark.table()* method and pass in a table name to create a Dataframe. And we have an example code shown below. So you can give a database and table name to read a Spark table and create a Dataframe out of it. So if someone is asking you to convert a Spark table to a Spark Dataframe, you can use *spark.table()* method.



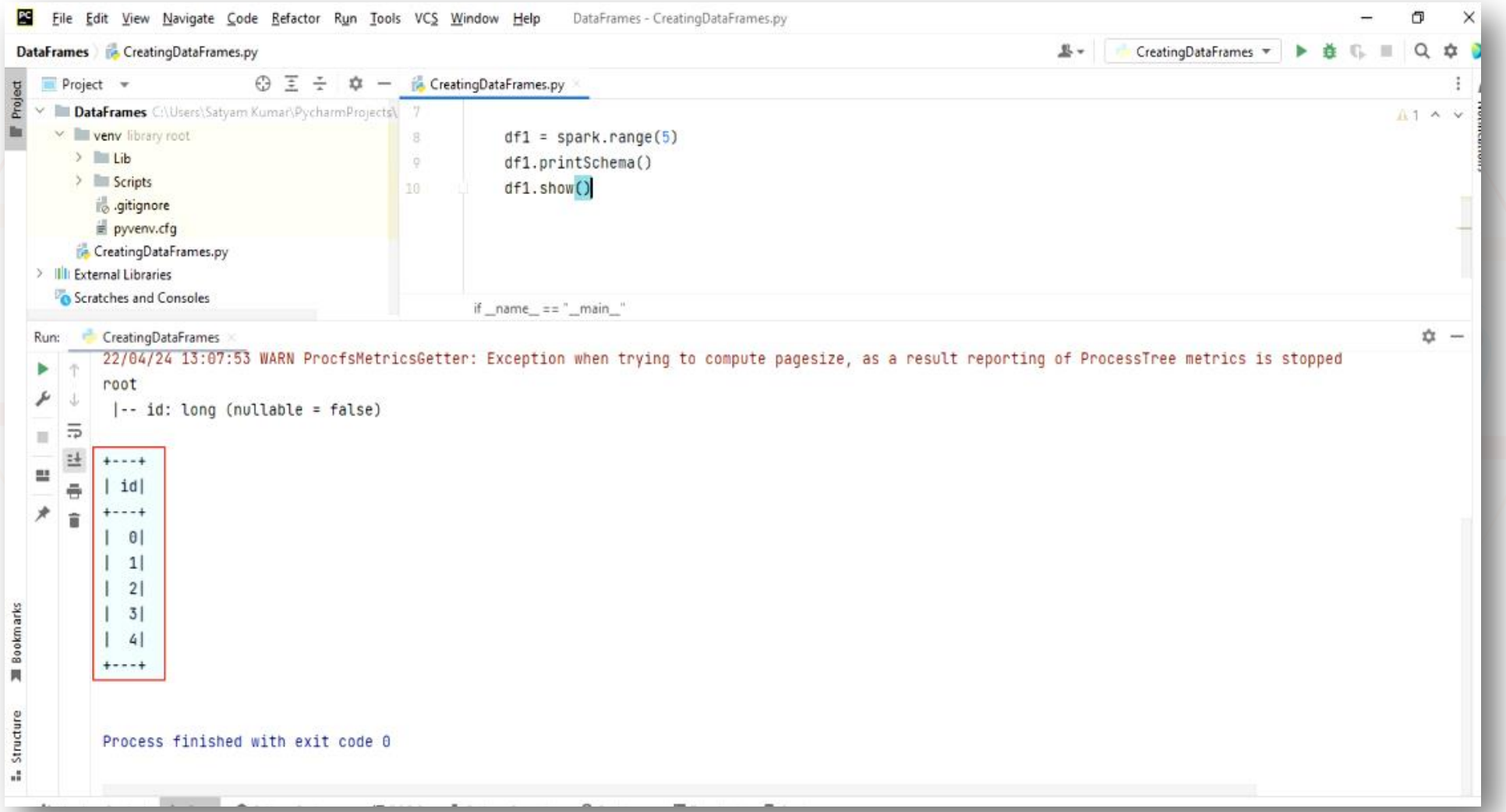
The main focus of this document was to help you understand the range() and the *createDataFrame()* methods.
These two methods allow you to create a Dataframe without reading data from a source.



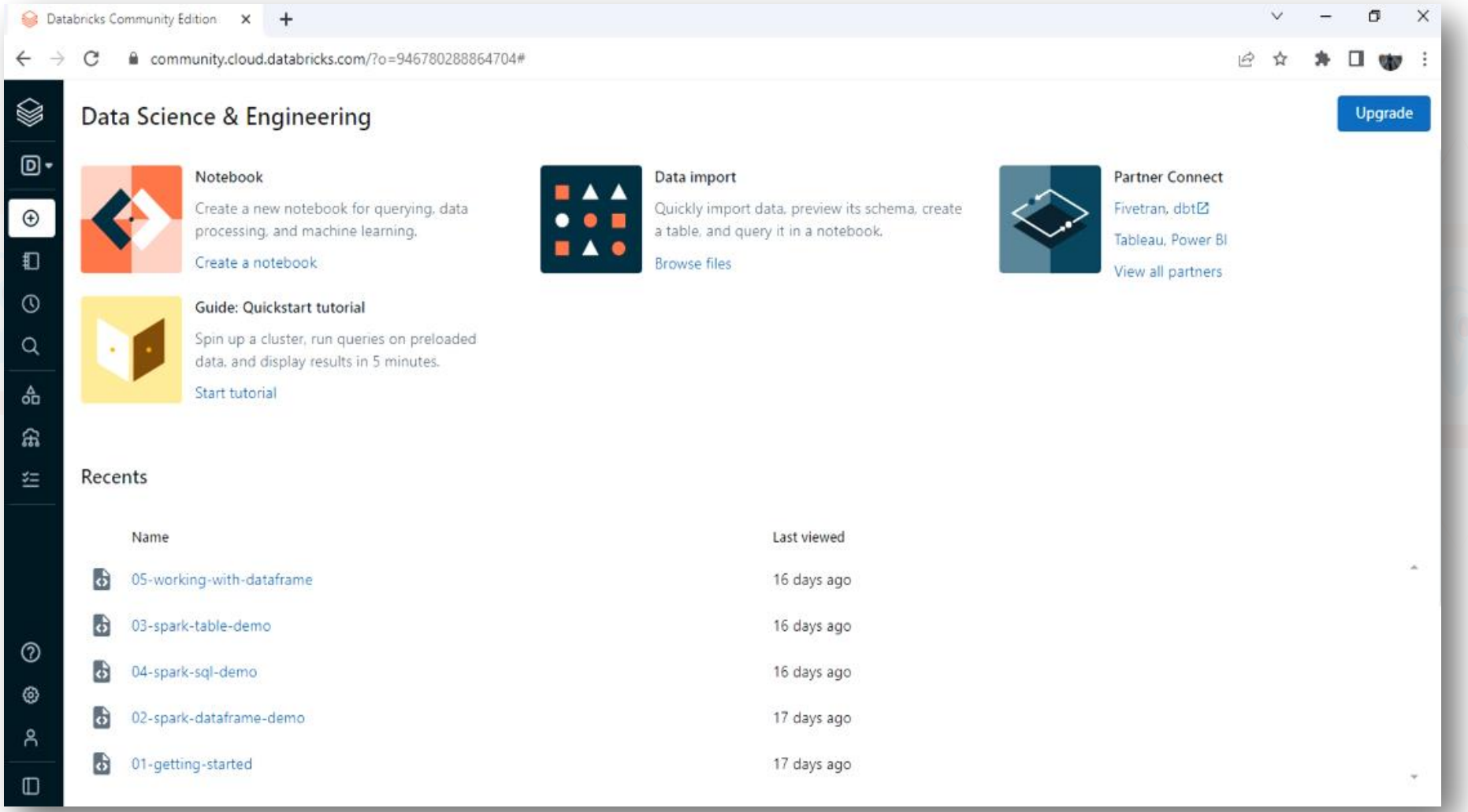
Here I have opened a previous project which I created earlier. **(Reference – DataFrames)**
In this example, we learned to create a Spark Session. Now, the next step is to create a Data Frame.
I have written the code for that. We have a single-column Dataframe. The *spark.range()* allows you to create a single-column Dataframe. You might need a single-column data frame for testing something. And *spark.range()* is an easy method to create it. The range() method takes many parameters like start, end, step, and the number of partitions. You can show this Dataframe and check the schema.



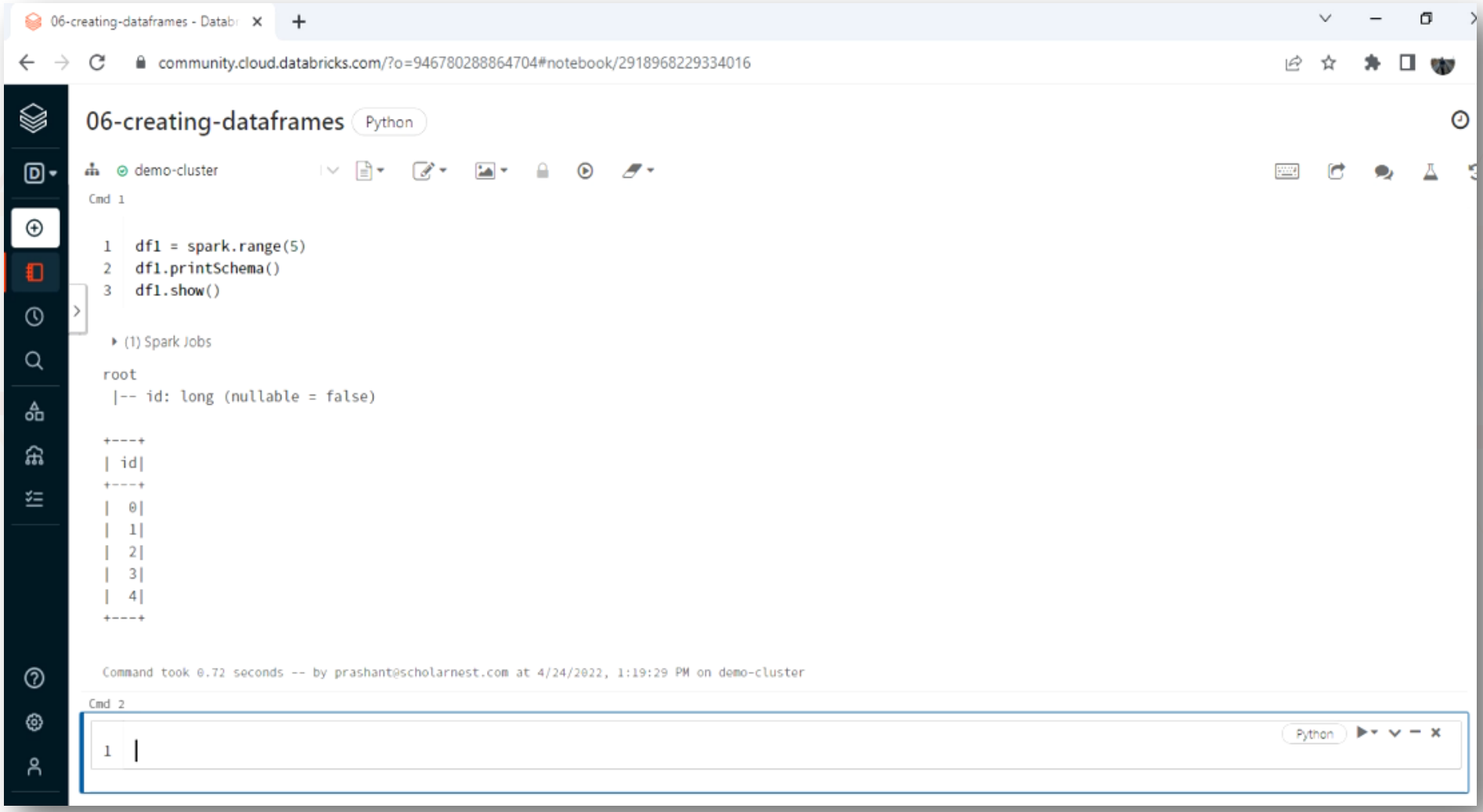
If we run the code, we can see the results as shown below.
By default, the column name is id, and its data type is long. The Dataframe values start from zero and stop at 5.



If you find it difficult to work with IDE, you can switch to Notebooks.
Here I am in my Databricks Workspace.
Create a new notebook. **(Reference - 06-creating-dataframes.ipynb)**



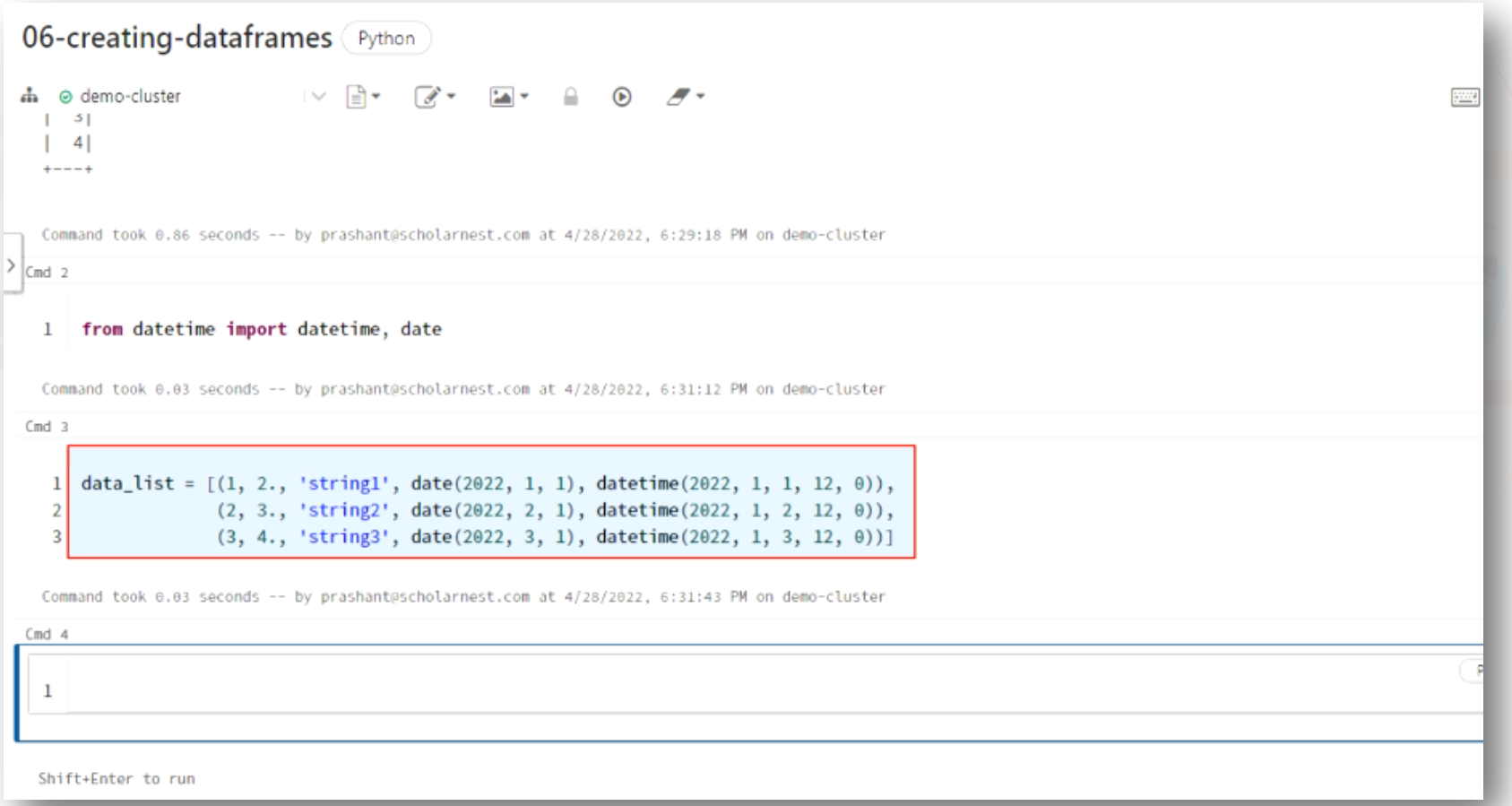
Now you can go back to the IDE and copy the last three lines of the code. Databricks notebook automatically creates a Spark session variable for me. So I do not need to create a Spark session variable. If you run the cell, you will get the same output.



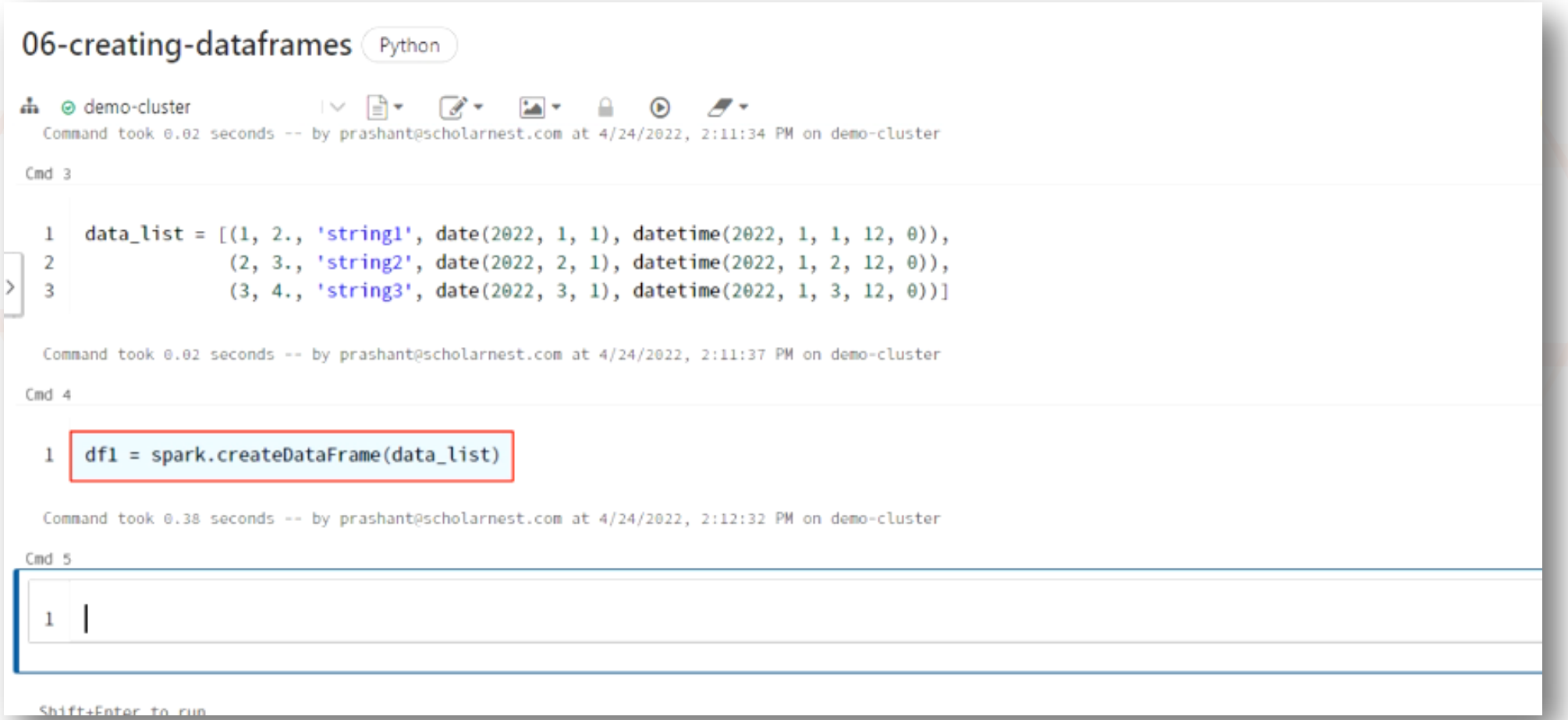
The range method is basic. It only supports creating a single-column Dataframe.
But we also have *spark.createDataFrame()* method. It converts a list into a Spark Dataframe.
Look at the requirement given in the image below to create a Dataframe programmatically.
You have column names shown here. Then you have some rows shown in the diagram. You also expect column data types, as shown at the top.

long	double	string	date	timestamp
a	b	c	d	e
1	2.0	string1	2000-01-01	2000-01-01 12:00:00
2	3.0	string2	2000-02-01	2000-01-02 12:00:00
3	4.0	string3	2000-03-01	2000-01-03 12:00:00

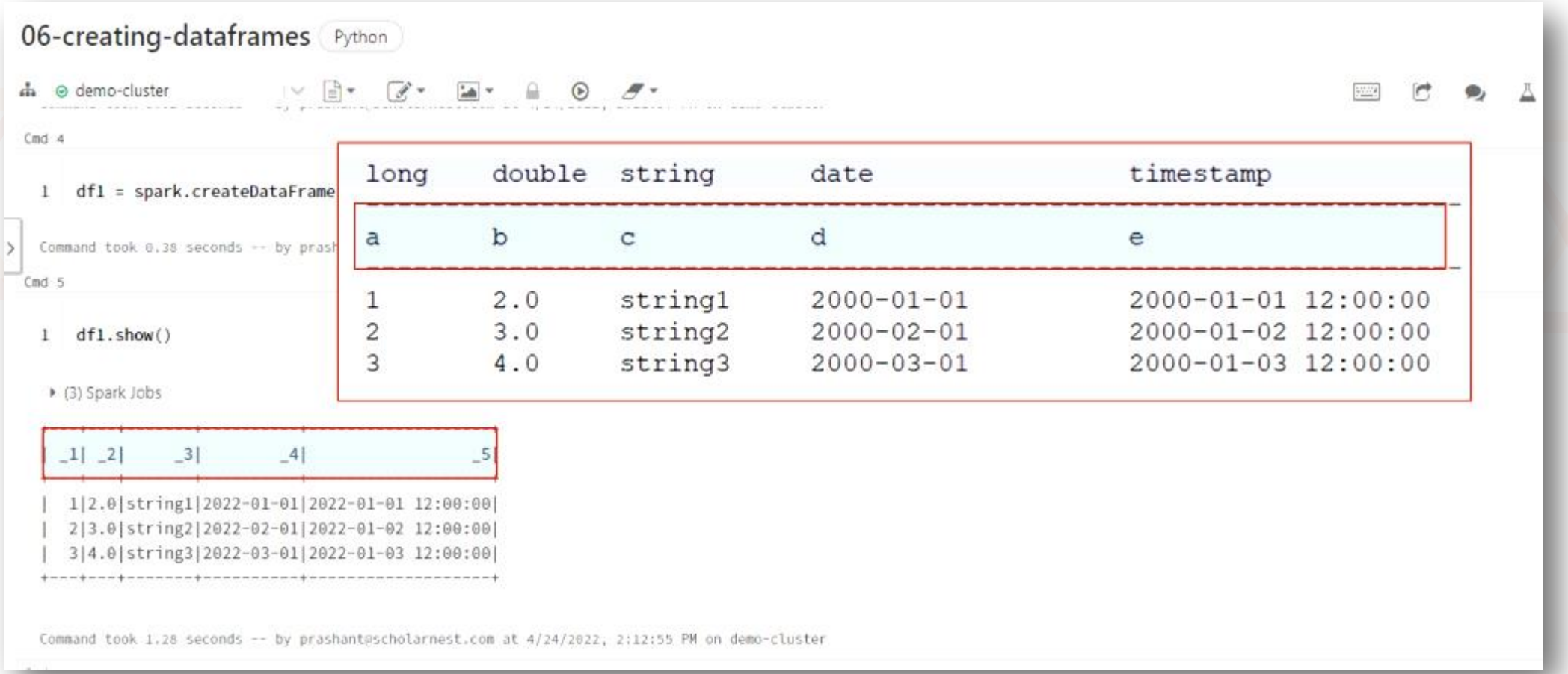
The *createDataFrame()* takes a Python list and converts it into a Dataframe. We define a python list inside a squire brackets. Then we define each record as a tuple inside a pair of parenthesis. I wanted three records, so I defined three types inside the squire bracket.
The column is a date, and the last one is a datetime, so I am using Python date and datetime functions from the Python datetime module. You must import the date and datetime functions from the datetime module.



Once you have the list, you can create a Dataframe as shown below. I am using *spark.createDataFrame()* method and passing the *data_list*.
The *createDataFrame()* method will convert the list into a Spark Dataframe and return it.
So the df1 is my Spark Dataframe.

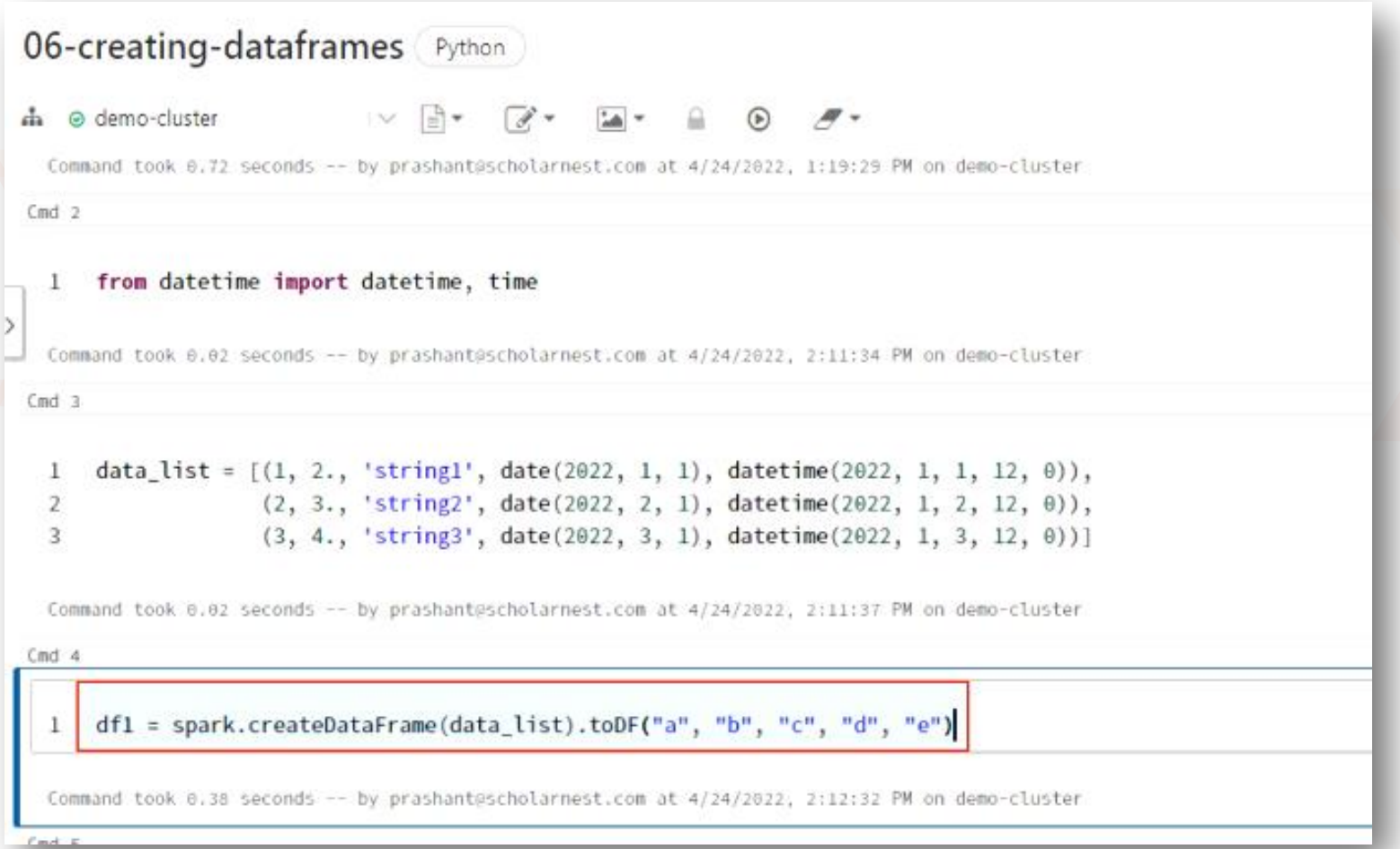


You can show the Dataframe and check if you got the desired output. Adjacent to the output is an image of the desired Dataframe. I am almost there except for a tiny problem.
The column names are not as per my requirement. It took some default column names here. But I wanted to give some different column names. I will fix this in a while, but before that let us check the schema.

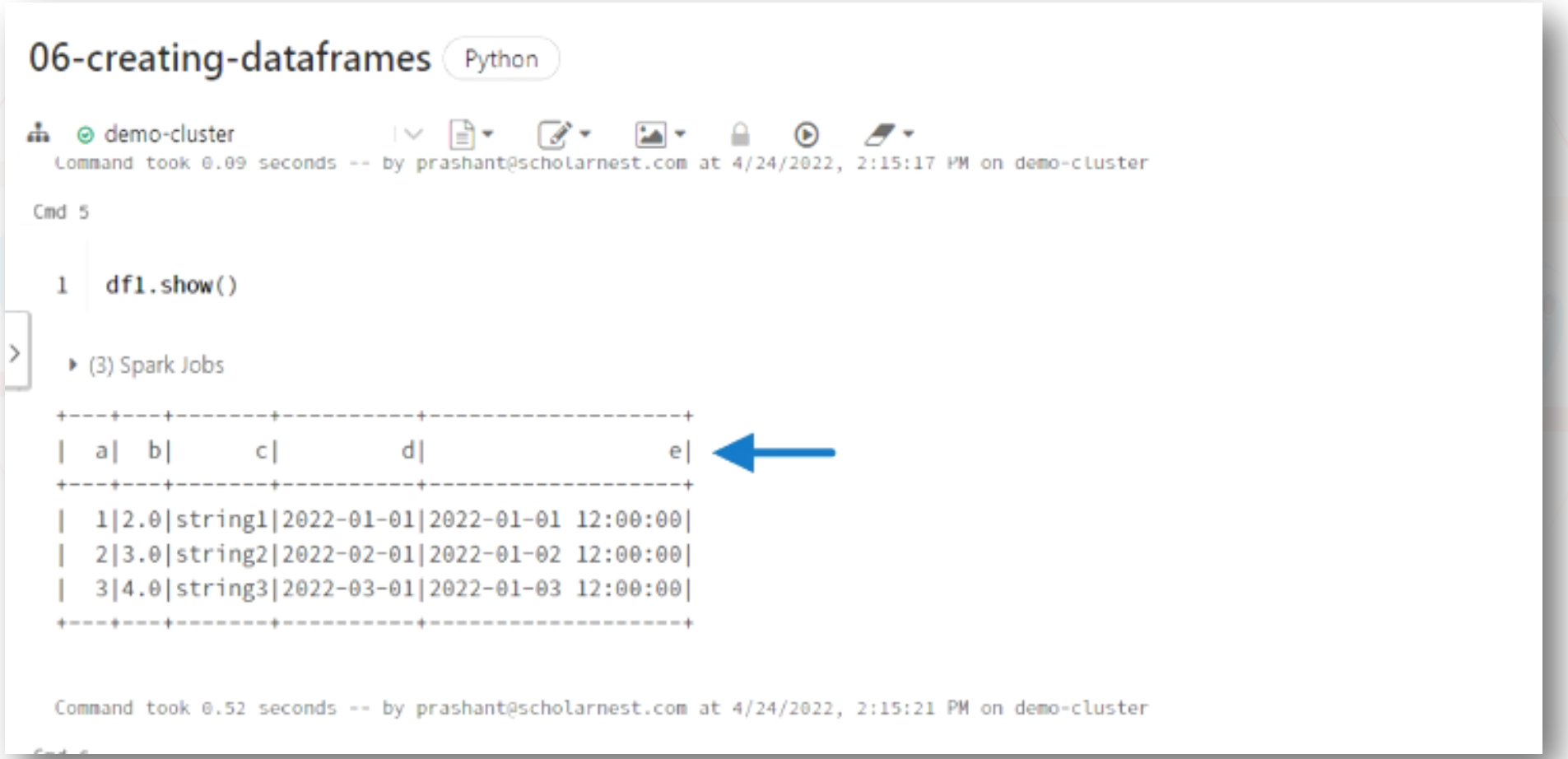


Here is the schema shown below. It looks like data types are fine. The *createDataFrame* automatically infers the schema of the Dataframe based on the given data. So do remember to define your data list with correct data types so the *createDataFrame* can easily infer it.

Now let's come back to the column name problem. How do I fix these column names?
One easy way is to use the *toDF()* method. The *toDF()* is a Dataframe transformation.
So once you have the Dataframe, you can use the *toDF()* to change all the column names at once. You can chain it after the *createDataFrame()* as shown below.



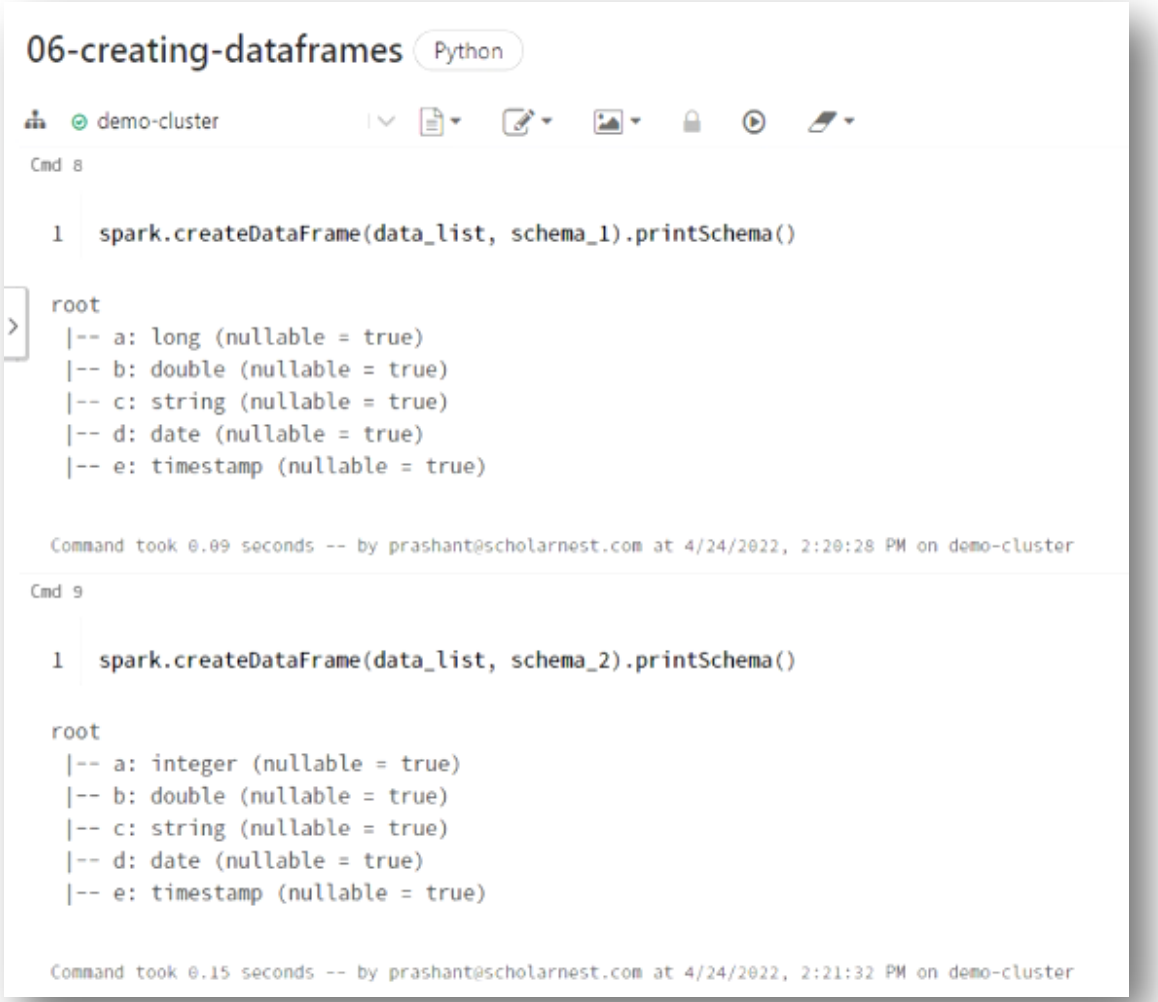
You can again check your Dataframe, and the column names are rectified as desired.



We created a *data_list* and used it to create a Dataframe. The *createDataFrame()* automatically infers the schema and generates some default column names. But you can supply a schema definition and enforce it. And here are two ways to define a schema as shown below.
The first one is simply a list of column names.
The second approach defines the column names and data types both.

I am using the first schema to show you the result.
I supplied a list of column names, so the *createDataFrame* should use the given columns names instead of the default names. Data types are still correct because the *createDataFrame()* could infer it correctly from the *data_list*.

Then in the second schema, you can see that I purposefully change the data type for the first column. I know the *createDataFrame* will infer the first column to be a long value. But I want it to be an integer. I tried enforcing a schema and stopped *createDataFrame* from inferring it, and it worked. You can see the output in the bottom section of the image below.



The *createDataFrame* is even more flexible.
We used Python list and schema definition for creating a Spark Dataframe.
However, the create Dataframe can also take the following.

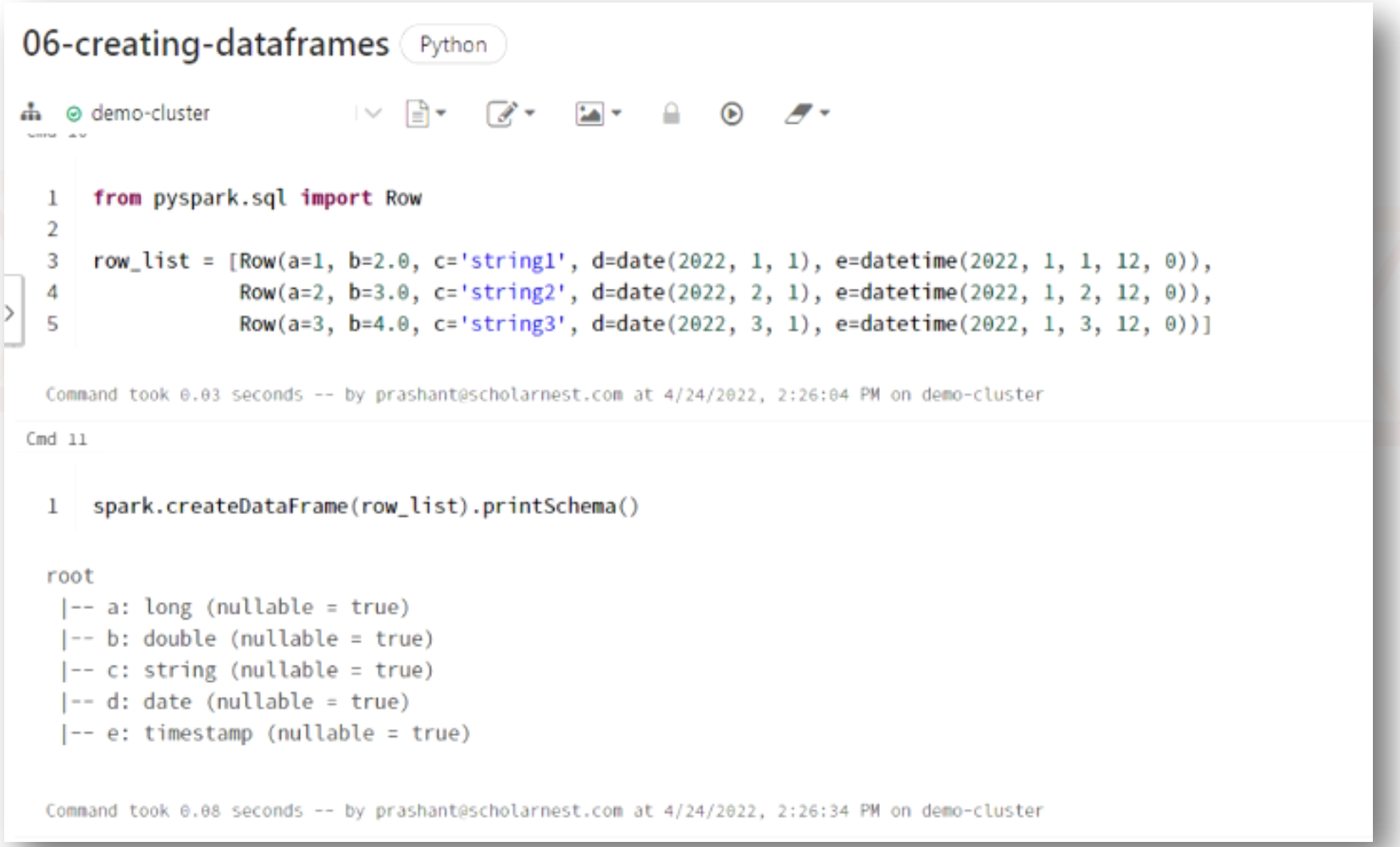
1. List of DataFrame Row
2. Pandas DataFrame
3. Spark RDD

List of DataFrame Row:

I am importing the Row class from the pyspark.sql package. You can create a list of rows and use them to create a Dataframe programmatically. Each Row is a comma-separated list of column names and column value pairs. Once you have the row list, you can use it in the *createDataFrame()* method as we used the python list.

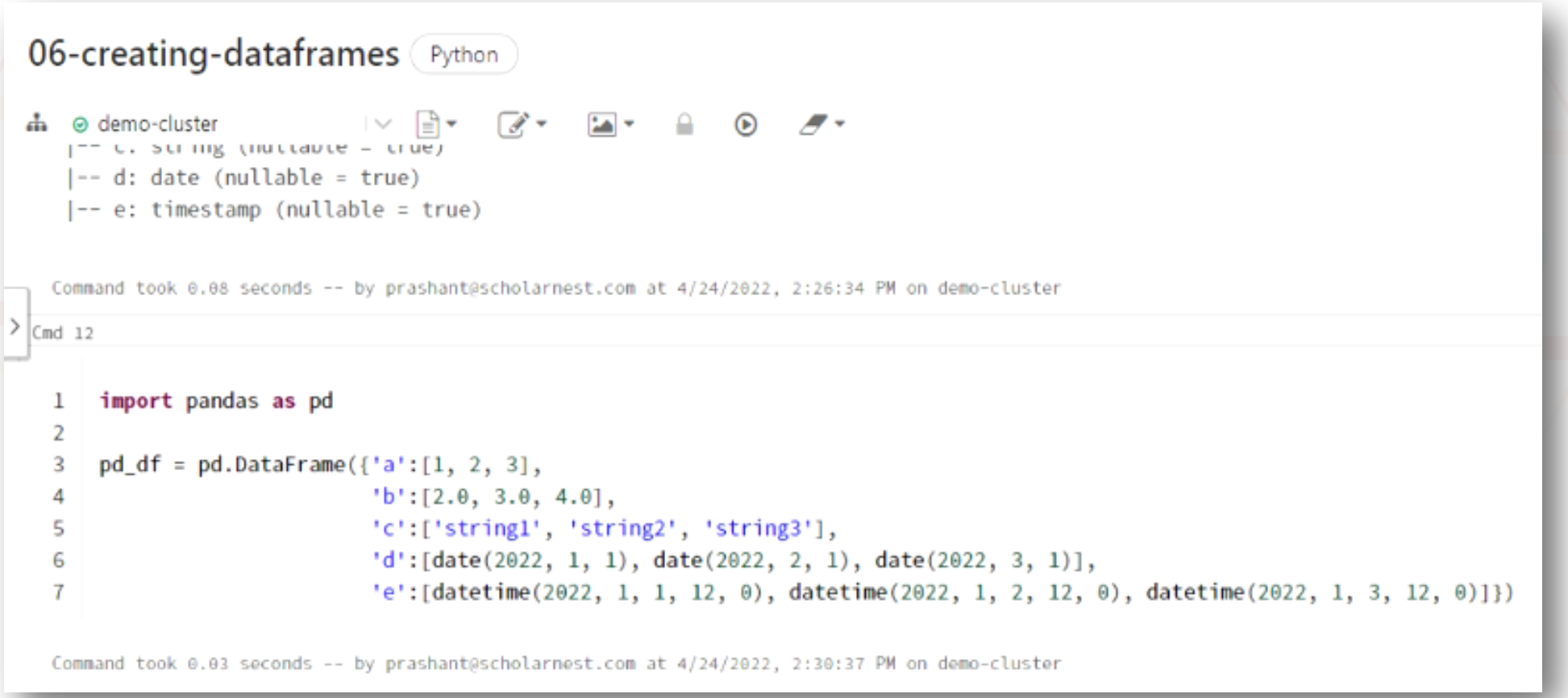
List of DataFrame Row:

You can print the schema to see that row list comes with the column names, so the *createDataFrame()* method takes the correct names. However, it will infer the schema. But you can also pass the schema to the *createDataFrame()* and enforce the data types.



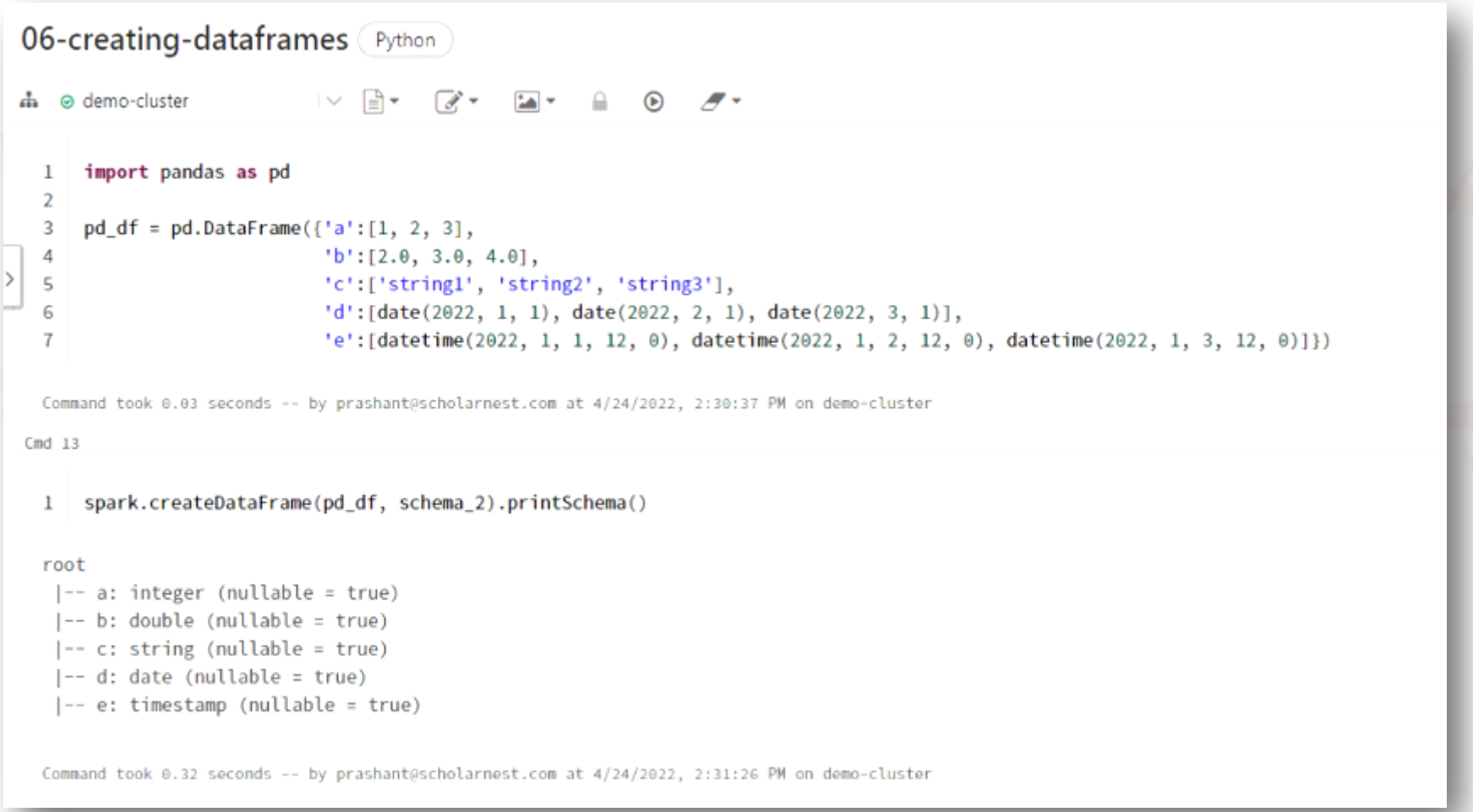
Pandas DataFrame:

So I am importing the panda's package and defining a panda Dataframe. You might not have worked with pandas package, and that's fine. We are just showing an example, so you know how to convert a pandas Dataframe into a Spark Dataframe.



Pandas DataFrame:

You can use the *createDataFrame()* to print the schema. I used *schema_2* so I could enforce the schema on this Dataframe. However, the schema is an optional argument. If you do not supply the schema, the spark will infer it from the given values.



Spark RDD:

Spark core package defines an RDD data structure. The full form of the RDD is Resilient Distributed Dataset. However, it is as simple as an in-memory distributed data structure. In the earlier versions of Spark, we used RDDs for data processing. However, the later versions of Spark came up with the Dataframe. Dataframe is also a distributed data structure. It is built on top of the RDD. However, Dataframe is optimized and more efficient than the RDD.

The RDD cannot have a schema. It is schema-less. But Dataframe must have a schema.

You cannot create a schema-less Dataframe. It must have a schema. If you do not provide a schema, Spark will apply some default schema. We saw the same earlier, *createDataFrame()* method infers the data types and adds some default column names to create a default schema.

Hence, Spark Dataframe is a more advanced, optimized, and schema-full data structure.

We recommend using Dataframes as much as possible. Spark RDD is part of the Spark core package, and it is schema-less. We should avoid using RDD and use Dataframes.

Spark RDD:

Here is the code for manually creating an RDD. I am using the spark session to get the SparkContext. Why SparkContext? Because SparkContext is my gateway to the Spark core package. Then we use the parallelize() method to create an RDD.
Once you have an RDD, you can impose a schema and convert it to a Dataframe.

Spark RDD:

You can see the desired output if you print the schema.

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 Dataframes Schema

Requirement:

You are given a flight time data set. This data set comes in a JSON file, and you are asked to load this Dataframe and create a Spark table. For your reference, here is one sample record, and here are details about the columns shown below.

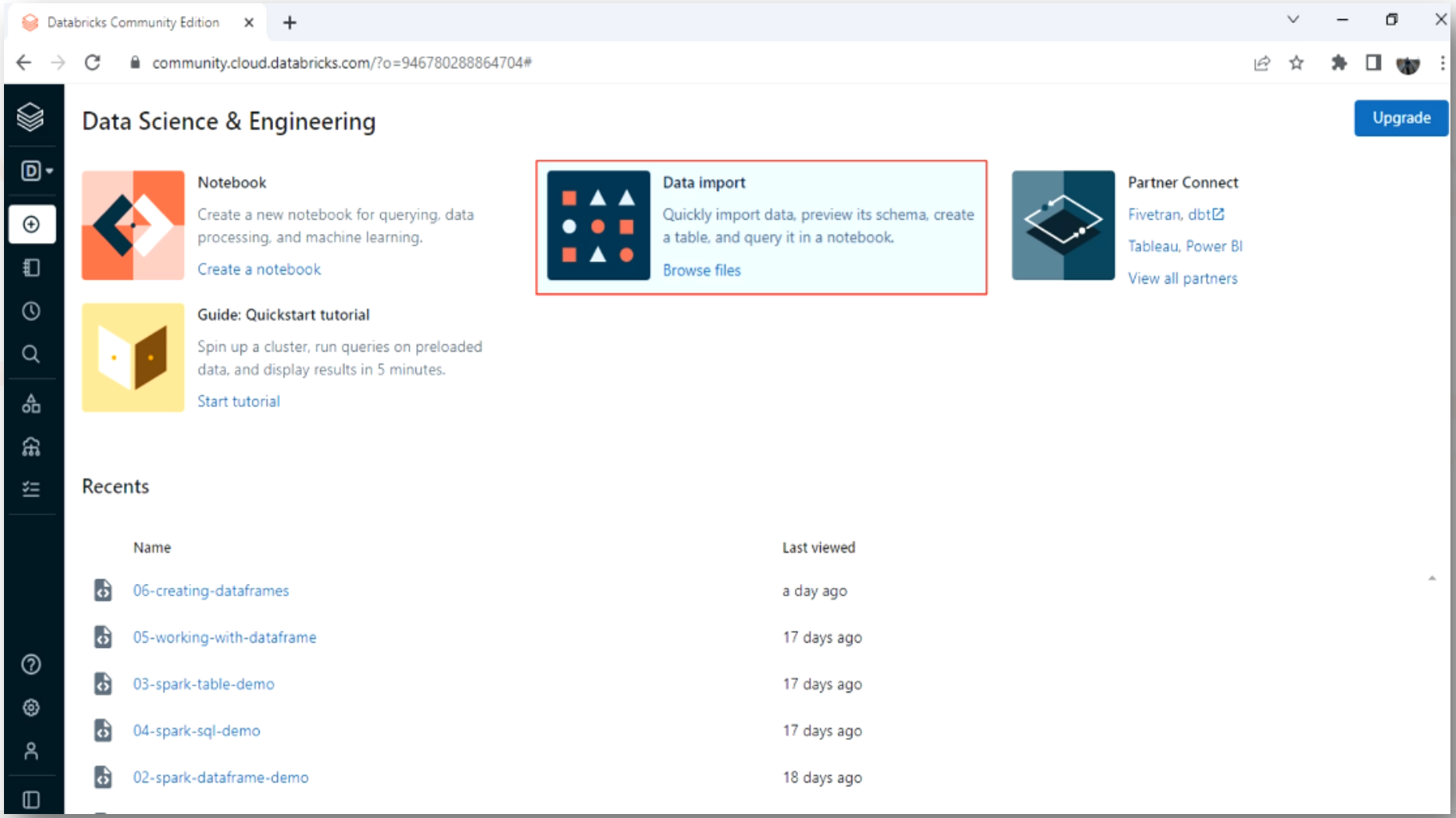
Flight Delay Data


```
{
  "FL_DATE": "1/1/2000",
  "OP_CARRIER": "DL",
  "OP_CARRIER_FL_NUM": 1451,
  "ORIGIN": "BOS",
  "ORIGIN_CITY_NAME": "Boston, MA",
  "DEST": "ATL",
  "DEST_CITY_NAME": "Atlanta, GA",
  "CRS_DEP_TIME": 1115,
  "DEP_TIME": 1113,
  "WHEELS_ON": 1343,
  "TAXI_IN": 5,
  "CRS_ARR_TIME": 1400,
  "ARR_TIME": 1348,
  "CANCELLED": 0,
  "DISTANCE": 946
}
```


1. Flight date in MM/DD/YYYY format
2. Unique carrier code
3. Flight number
4. Origin IATA Airport Code
5. Origin City Name
6. Destination IATA Airport Code
7. Destination City Name
8. Scheduled Departure time in HHMM
9. Actual Departure time in HHMM
10. Flight Duration in HHMM
11. TaxiIn taxi in time, in minutes
12. Scheduled Arrival time in HHMM
13. Actual Departure time in HHMM
14. Was it canceled? 1 for a true, and 0 is false
15. Distance in miles

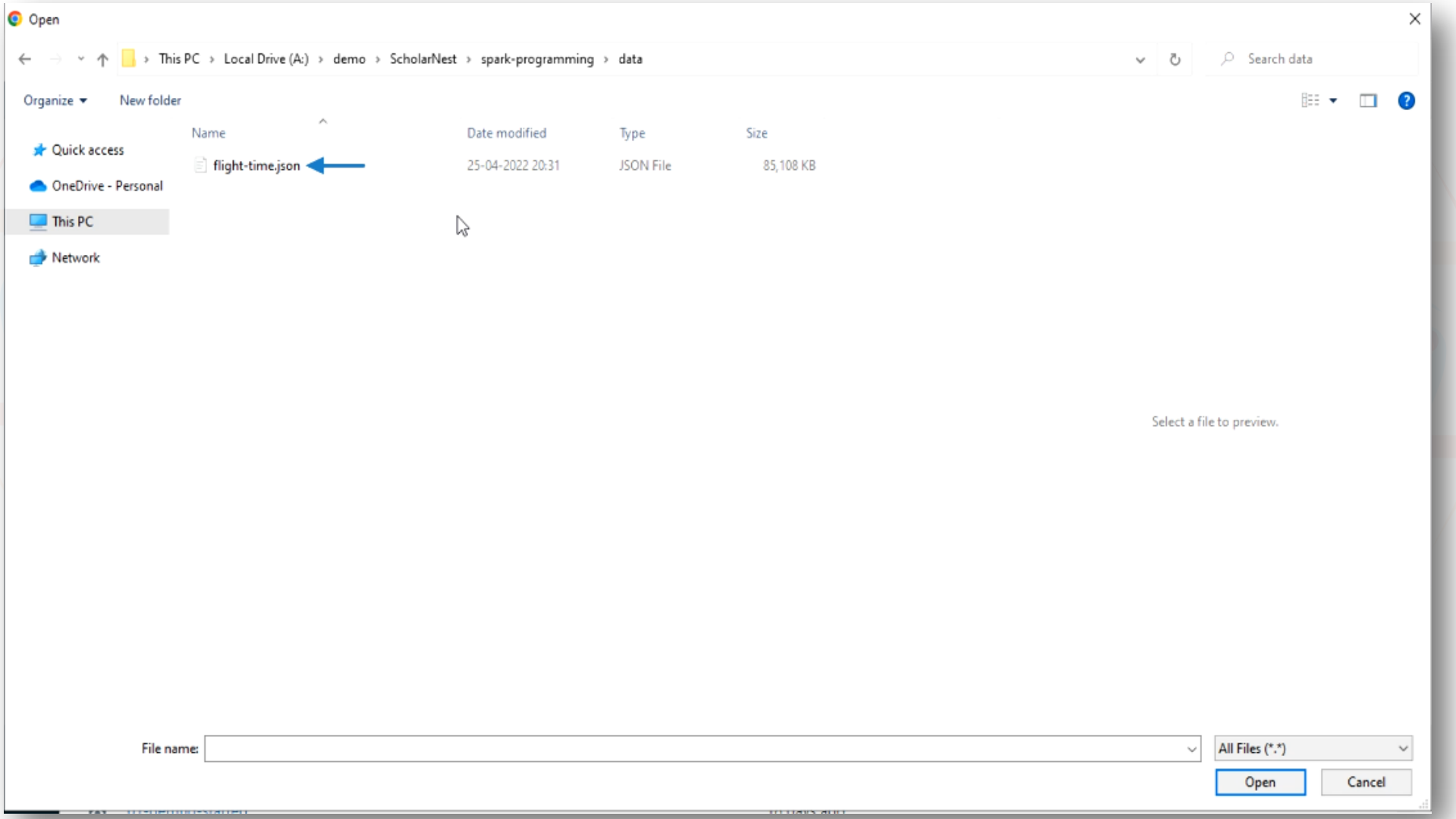
Load the table into a Dataframe:

Go to your Databricks workspace. My data file is available on my local machine. I will upload the data file to the cloud so I can work with it. Click this Data import button highlighted below.



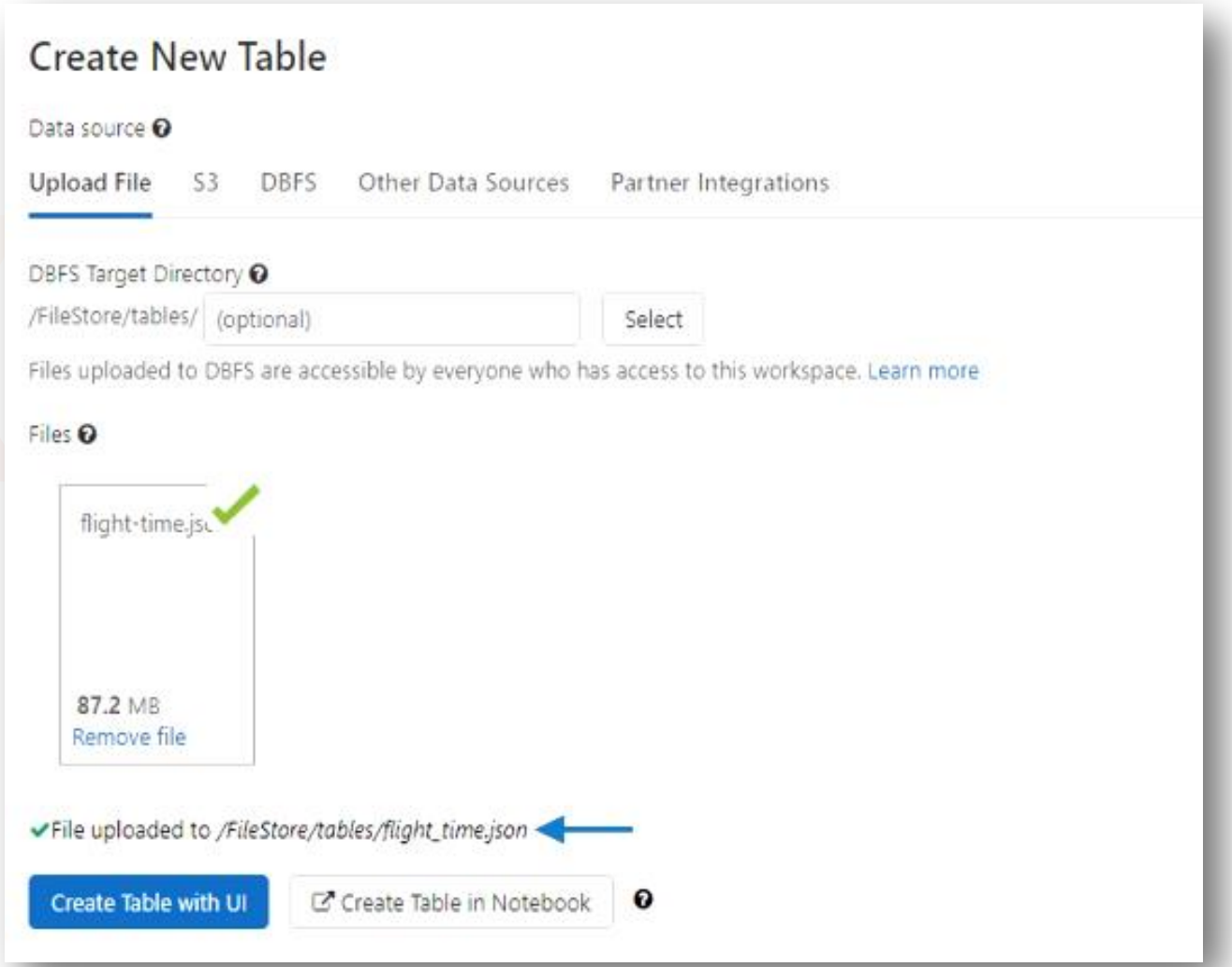
Load the table into a Dataframe:

Navigate to the folder that contains the file and upload it.

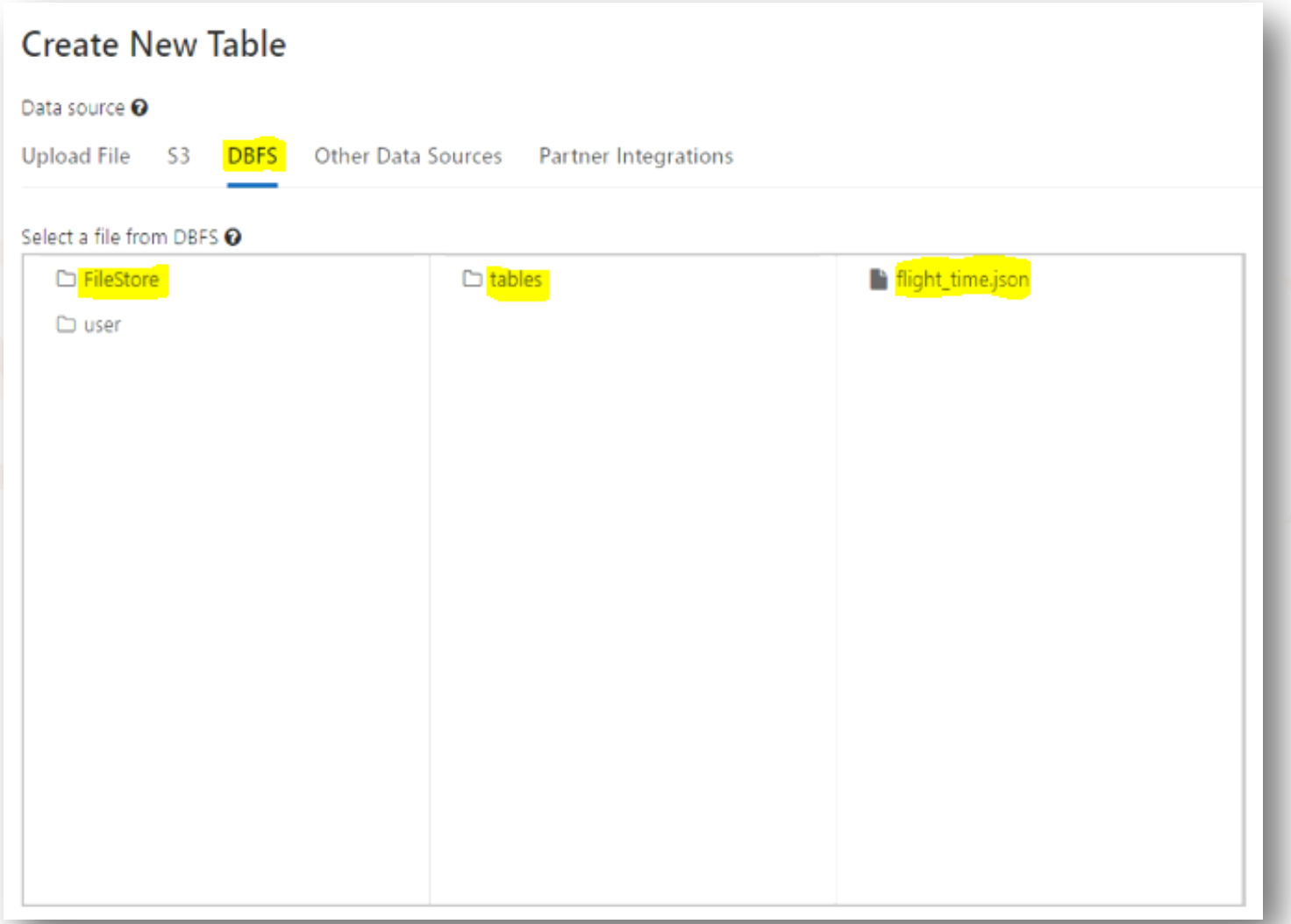


Load the table into a Dataframe:

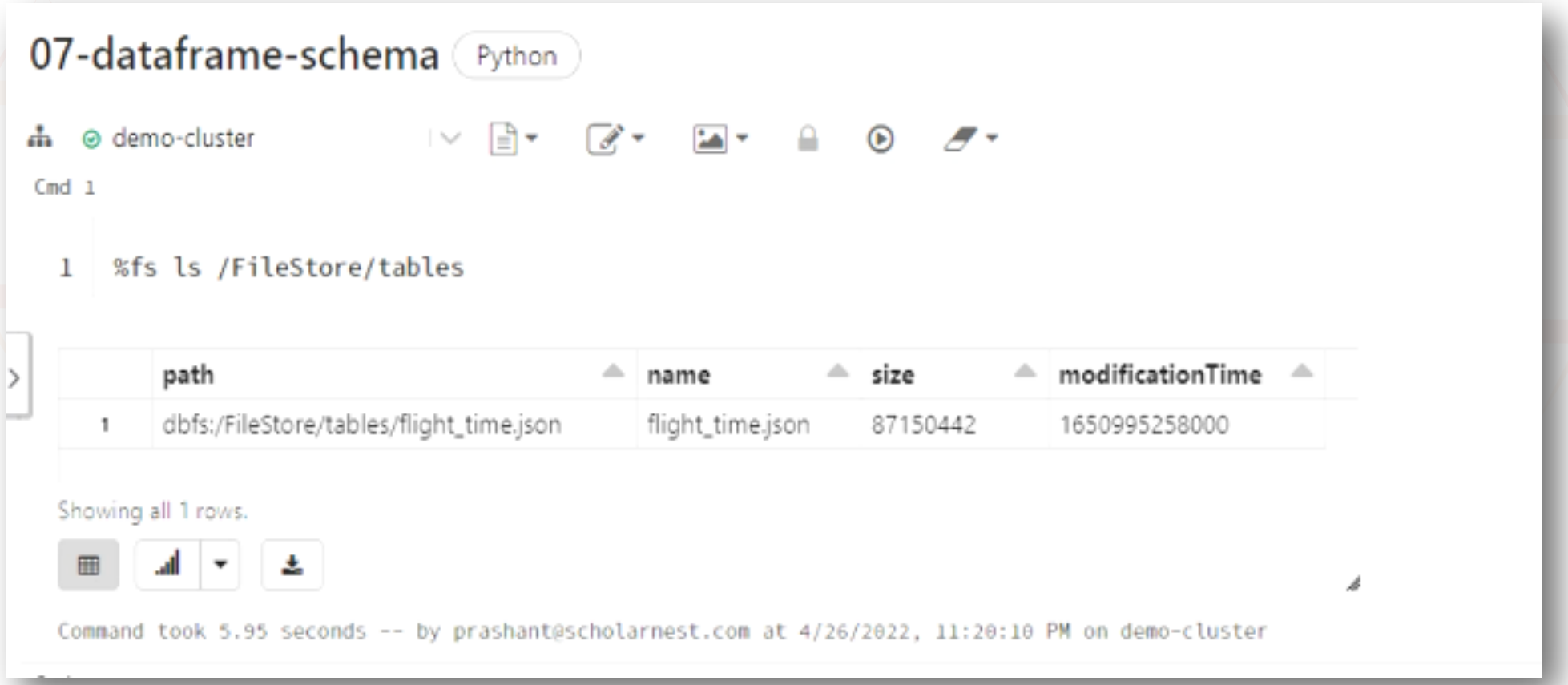
Your file is uploaded. It goes into the Databricks File System and sits in the */FileStore/tables/* directory. The Databricks File System is popularly known as DBFS, and the community edition DBFS is built on top of the Amazon S3.



To check your file, go to the DBFS tab and browse to the *FileStore/tables/*, and you will see the file. So my data file is now available in the cloud. Now you can create a new notebook to write code to meet the given requirement.



Create a new notebook. **(Reference - 07-dataframe-schema.ipynb)**
You can use the %fs ls command to check your data file, and we see the same in the output below.
The %fs ls is a DBFS command, and it is similar to the Linux ls command.



You have seen a similar code to create a Dataframe earlier.
I changed the format from CSV to JSON. Because this time, I am loading JSON data.
I also removed the header and inferSchema options. Because a JSON file doesn't come with a header row. The header option is only applicable for the CSV files.
The inferSchema option is not applicable for JSON. Because a JSON document comes with the column names in every record. Spark will always read the column names. In fact, it will read the column name with every record.

Cmd 2

```
1  flight_time_raw_df = spark.read \
2                       .format("json") \
3                       .load("/FileStore/tables/flight_time.json")
```

▸ (1) Spark Jobs

Command took 20.42 seconds -- by prashant@scholarnest.com at 4/26/2022, 11:22:50 PM on demo-cluster

Look at the schema of the data frame we created. You can see that Spark automatically inferred the schema for this Dataframe. Some columns are long, and others are string. And that is not correct.

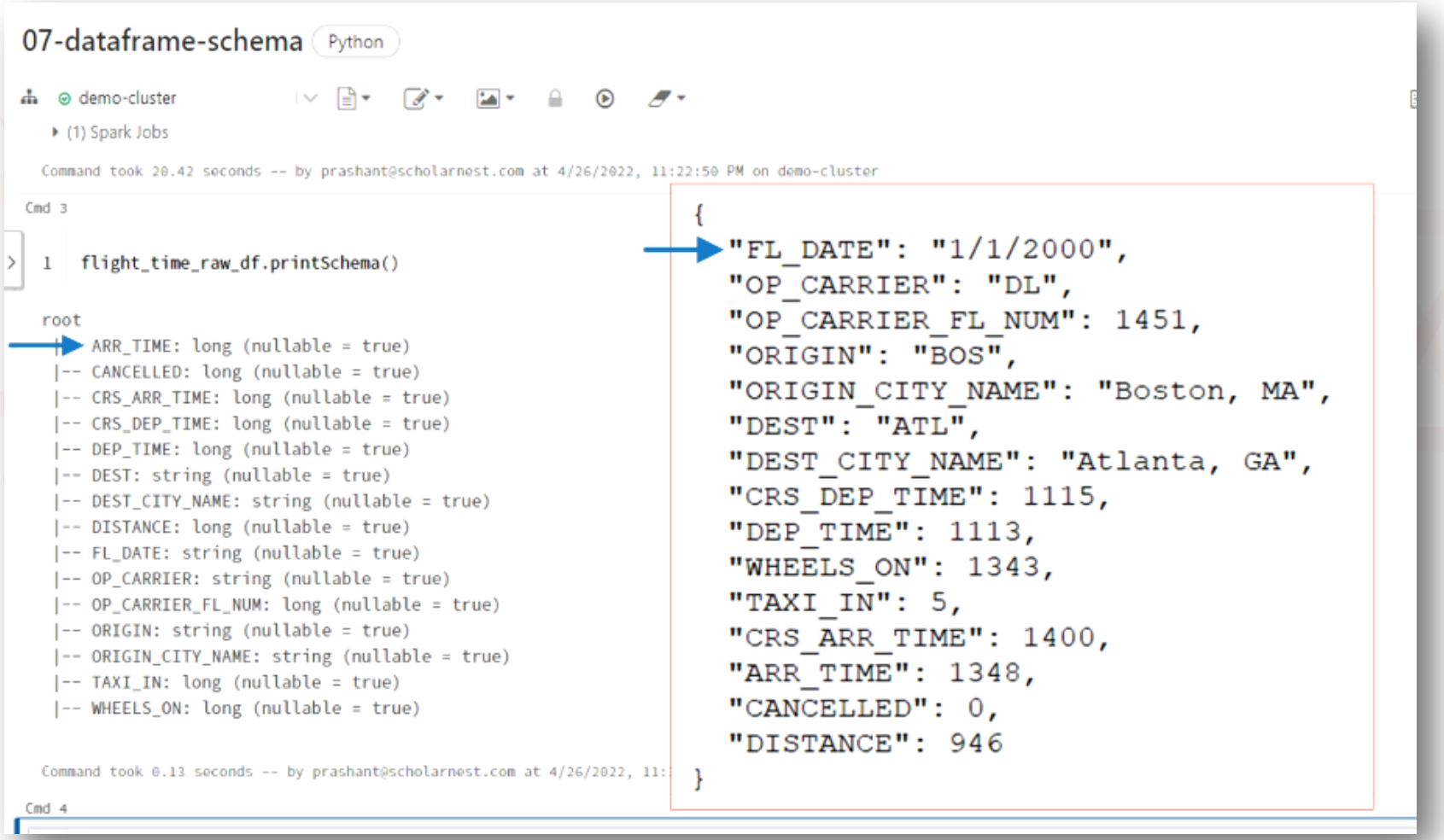
Schema inference is not reliable.

We have the following problems in schema inference:

1. Automatic inferring of schema is often incorrect
2. Inferring schema is additional work for Spark, and it takes some extra time
3. Schema inference is conflicting with the schema validation
4. It might also change the column order

Look at the schema of the data frame we created. You can see that Spark automatically inferred the schema for this Dataframe. Some columns are long, and others are string. And that is not correct.

Schema inference may also change the order of the columns. You can see it here if you compare the schema with your data file. The first column in Dataframe is ARR_TIME, whereas the first column in the file is FL_DATE. Column reordering is not a big problem. We should avoid creating a logic that depends on the column ordering but changing order often causes confusion. So we must avoid it.



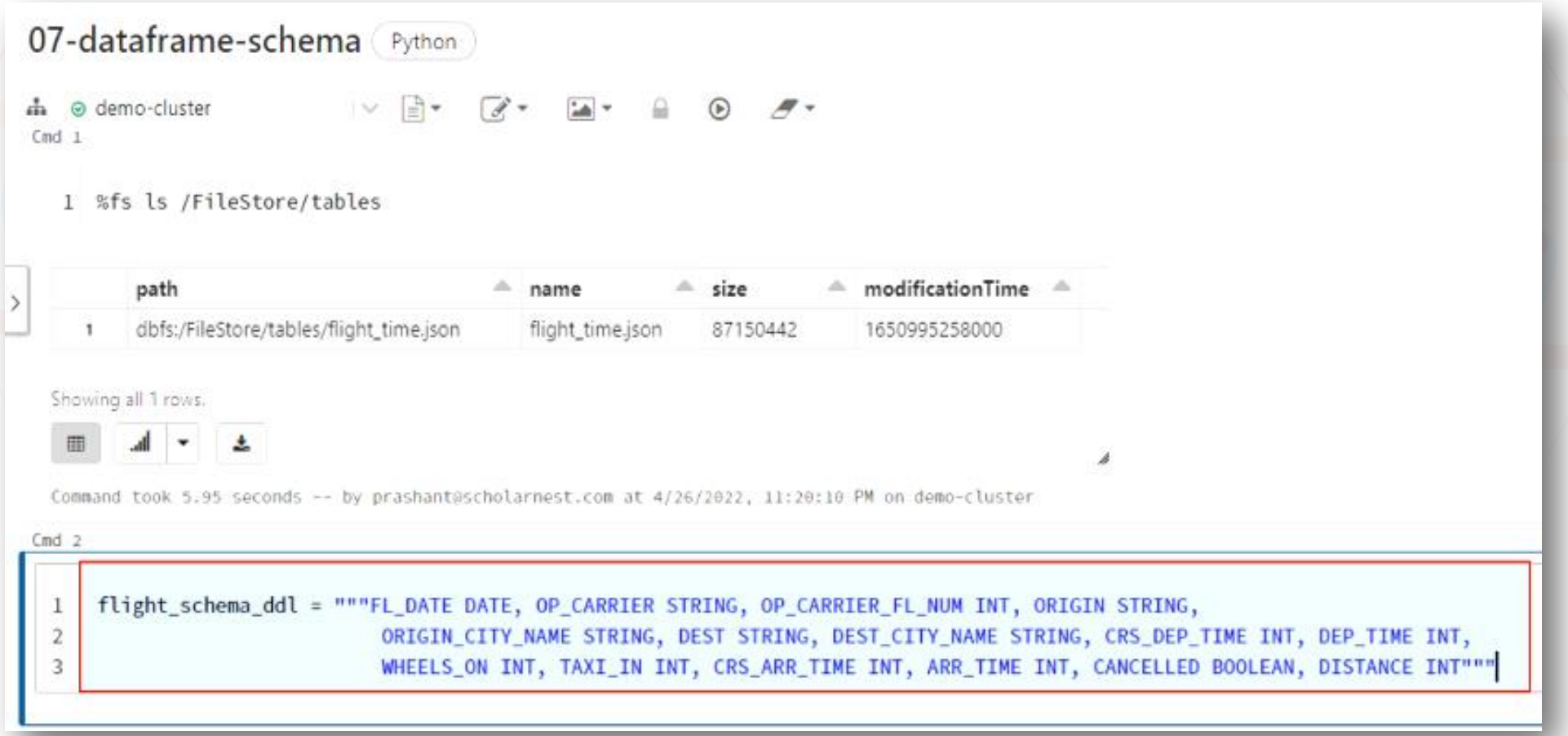
We have two steps to enforce a schema:

1. Create a schema definition for your data file.
2. Explicitly enforce the schema

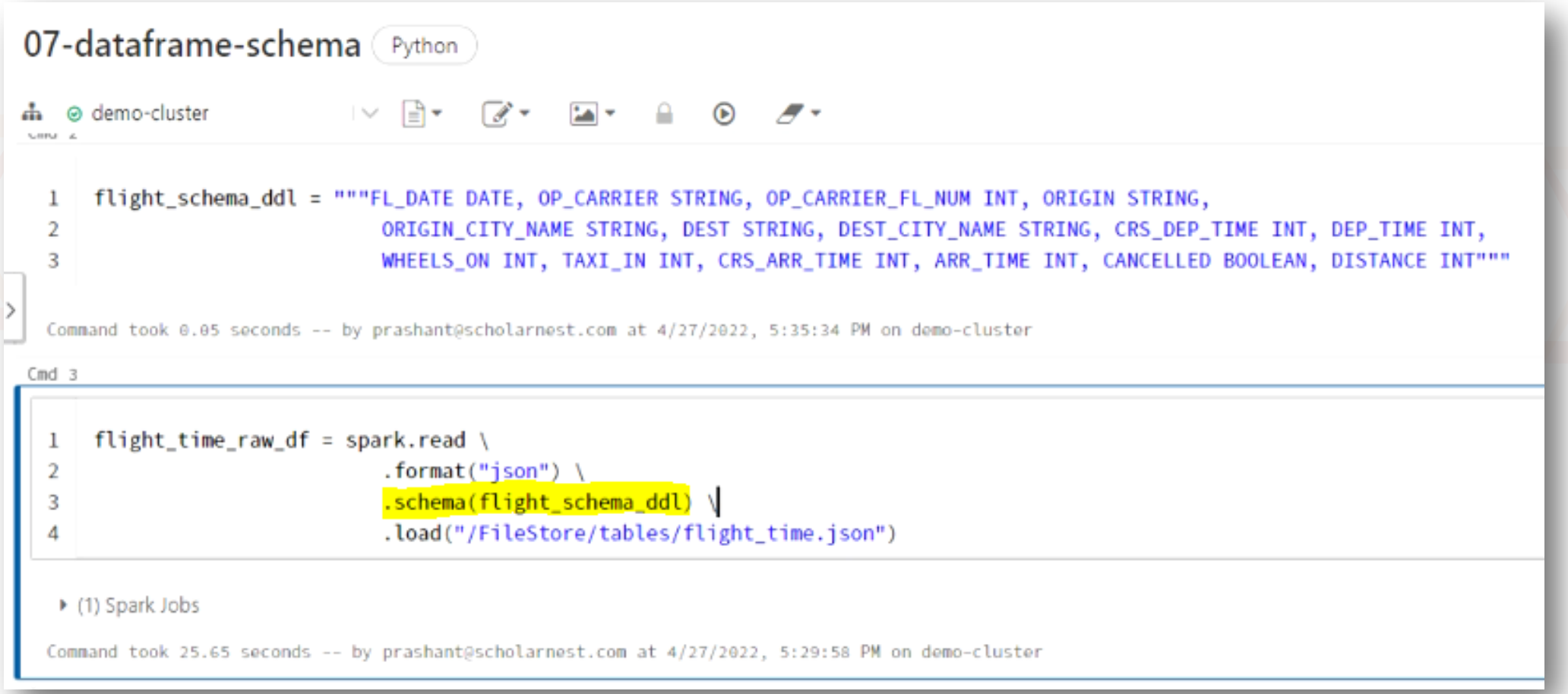
The first step is to create a schema definition.
We have two approaches to do it.

1. Schema DDL String
2. StructType Object

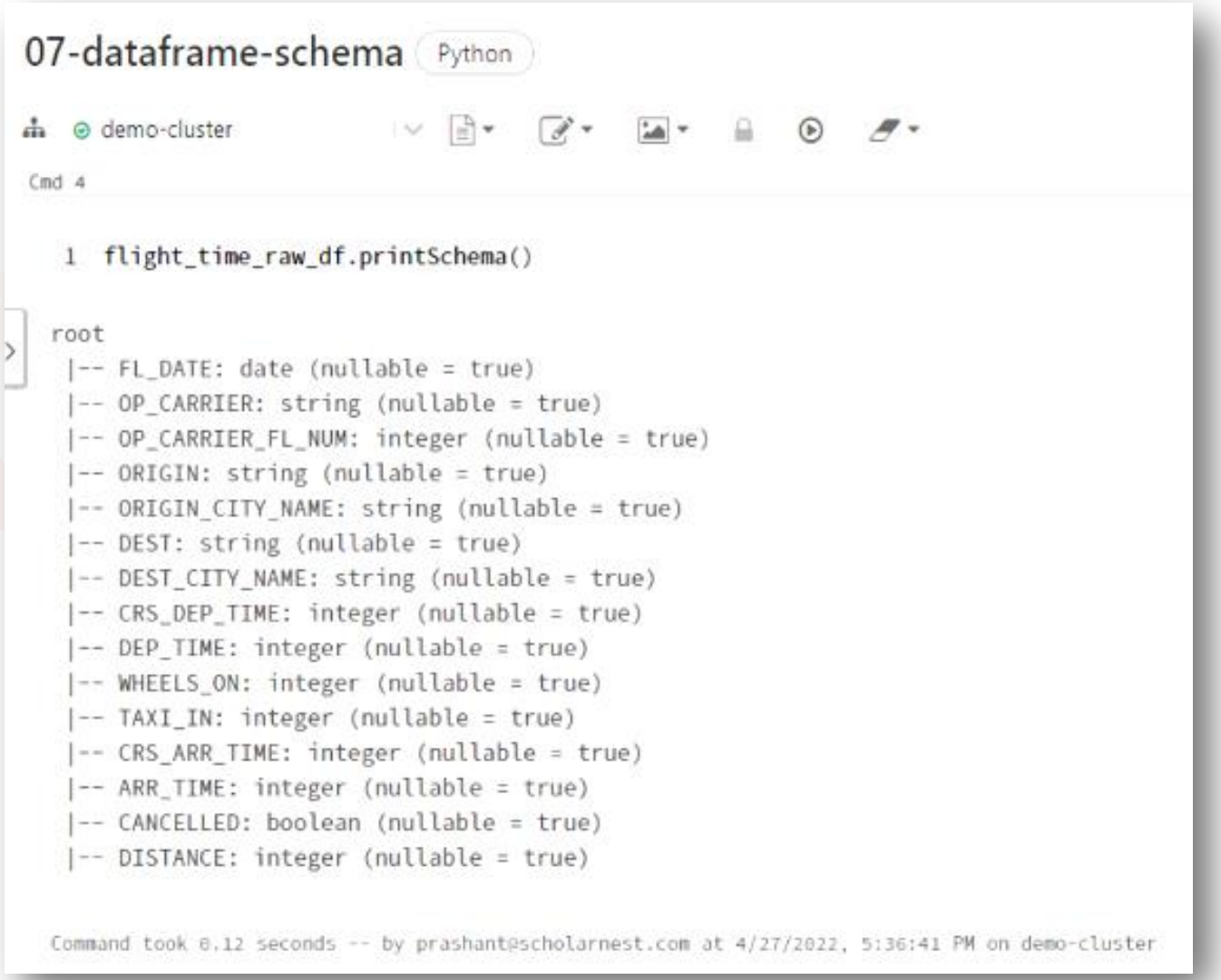
Schema DDL string is the easiest method.
Here is my schema definition for the flight-time data. A schema DDL is a comma-separated string of column names and data types. So the first column name is FL_DATE, and it is a date. Similarly, the second column name is OP_CARRIER, and it is a string.
You can create a list of all the columns and their data types.



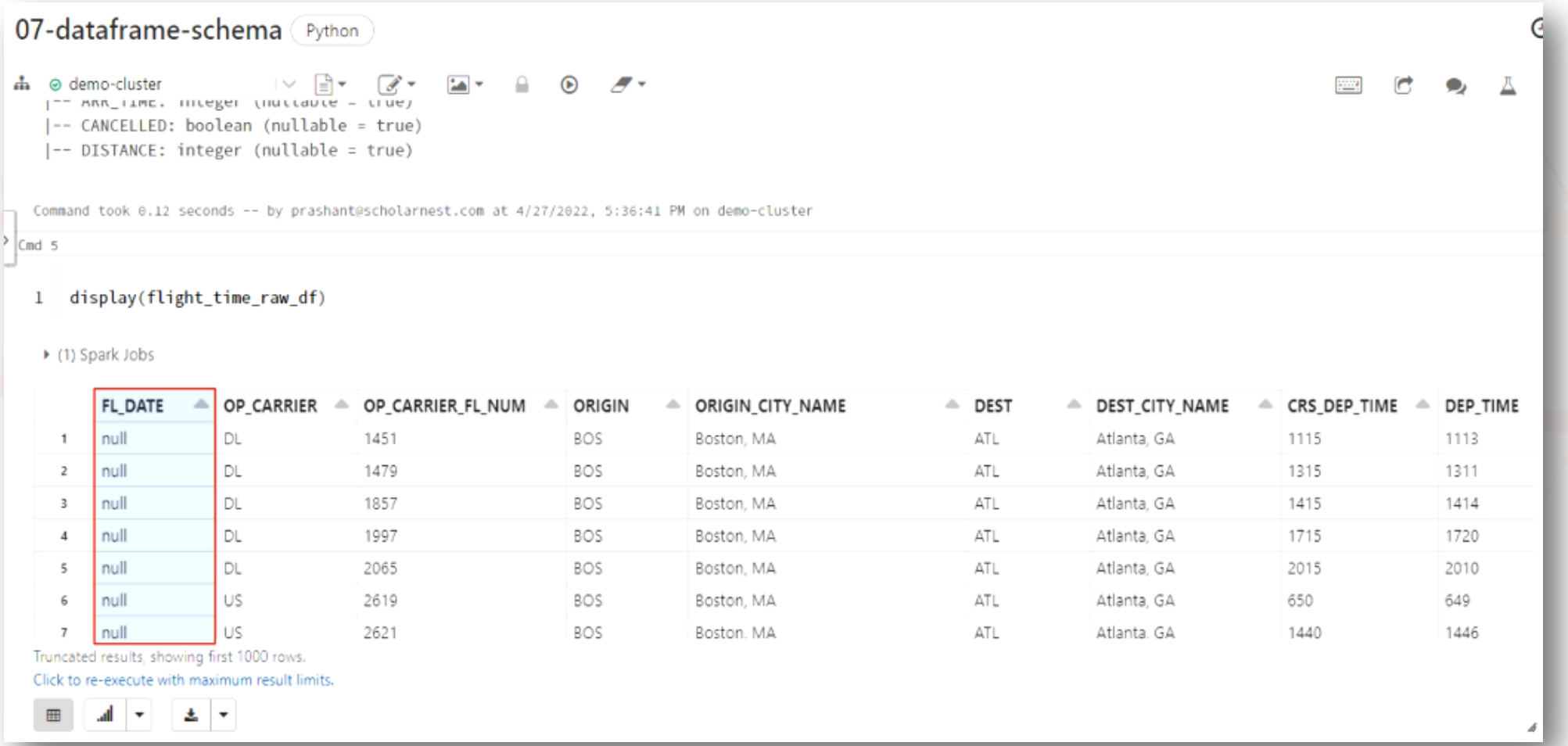
Once a schema is defined, you can enforce it to the DataframeReader by adding a schema method to your DataframeReader.
So here I am telling the DataframeReader to read a JSON format file using the flight_schema_ddl.



You can re-check the schema.
The print schema is now showing it correctly.

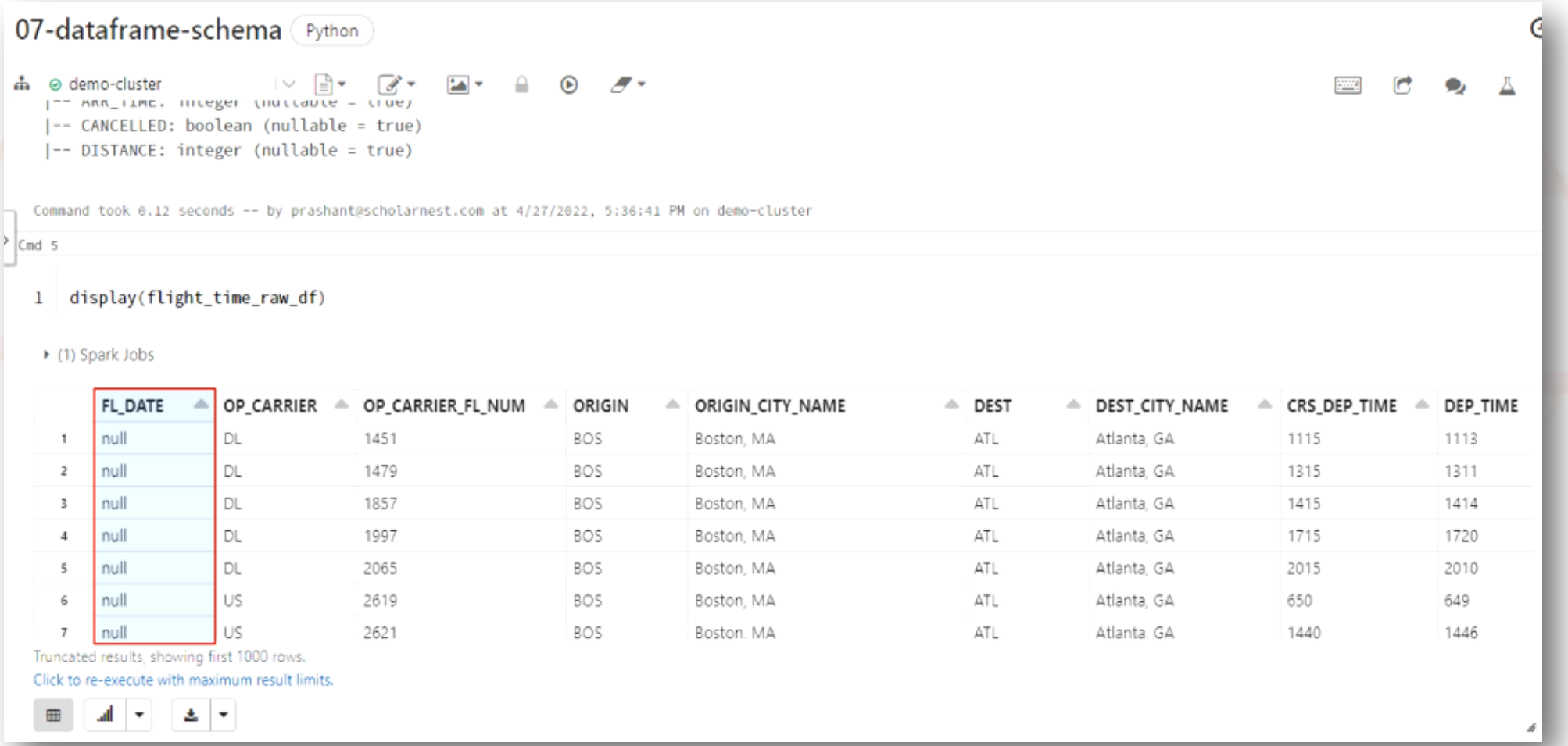


We can check the data once and see if we loaded it correctly.
So here is my Dataframe shown below. It looks like FL_DATE is not read correctly, and hence it is NULL. I do see some nulls in between, but other rows have values.



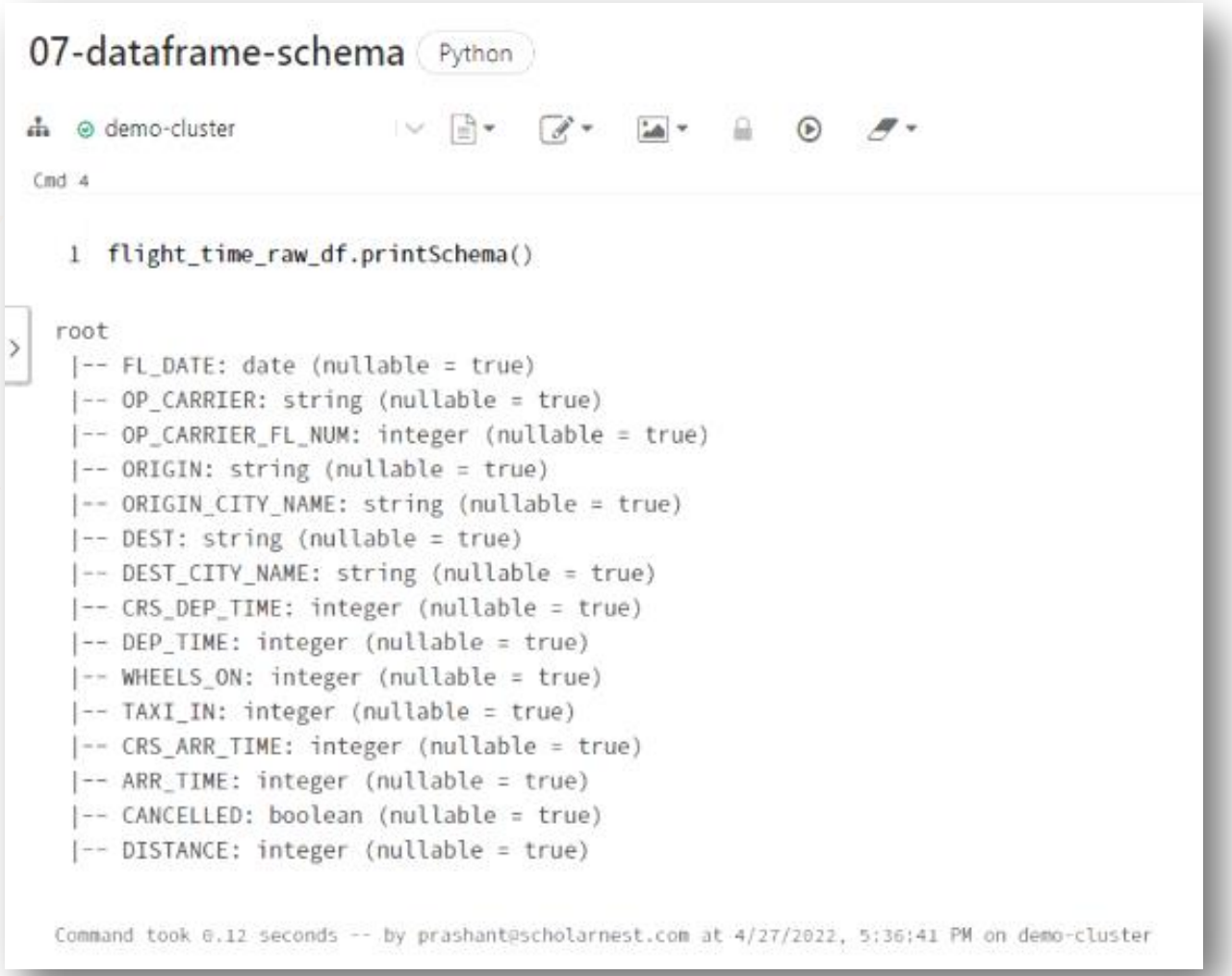
	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST	DEST_CITY_NAME	CRS_DEP_TIME	DEP_TIME
1	null	DL	1451	BOS	Boston, MA	ATL	Atlanta, GA	1115	1113
2	null	DL	1479	BOS	Boston, MA	ATL	Atlanta, GA	1315	1311
3	null	DL	1857	BOS	Boston, MA	ATL	Atlanta, GA	1415	1414
4	null	DL	1997	BOS	Boston, MA	ATL	Atlanta, GA	1715	1720
5	null	DL	2065	BOS	Boston, MA	ATL	Atlanta, GA	2015	2010
6	null	US	2619	BOS	Boston, MA	ATL	Atlanta, GA	650	649
7	null	US	2621	BOS	Boston, MA	ATL	Atlanta, GA	1440	1446

We can check the data once and see if we loaded it correctly.
So here is my Dataframe shown below. It looks like FL_DATE is not read correctly, and hence it is NULL. I do see some nulls in between, but other rows have values. The CANCELLED column is also null everywhere.



	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST	DEST_CITY_NAME	CRS_DEP_TIME	DEP_TIME
1	null	DL	1451	BOS	Boston, MA	ATL	Atlanta, GA	1115	1113
2	null	DL	1479	BOS	Boston, MA	ATL	Atlanta, GA	1315	1311
3	null	DL	1857	BOS	Boston, MA	ATL	Atlanta, GA	1415	1414
4	null	DL	1997	BOS	Boston, MA	ATL	Atlanta, GA	1715	1720
5	null	DL	2065	BOS	Boston, MA	ATL	Atlanta, GA	2015	2010
6	null	US	2619	BOS	Boston, MA	ATL	Atlanta, GA	650	649
7	null	US	2621	BOS	Boston, MA	ATL	Atlanta, GA	1440	1446

Here is my schema definition shown below.
The FL_DATE is a date, and the CANCELLED column is a BOOLEAN.
The date, timestamp, and the boolean values ate often loaded incorrectly, or they became null. You will see it almost everywhere.



If you check the requirements, you can see that FL_DATE is the flight date in MM/dd/yyyy format, and the CANCELLED is one or zero.

The default data format for DataFramereader is yyyy-MM-dd.
So the DataFramereader expected the FL_DATE to be in yyyy-MM-dd.

Similarly, the default format for a BOOLEAN is TRUE or FALSE.
So the DataFrameReader expected the FL_DATE to be TRUE or FALSE.

When the data is not found in the expected format, it fails the schema validation, and the Spark reads a null.
It doesn't throw an error. However, you can change this behaviour by setting the mode option.

So am setting the DataFrameReader mode to FAILFAST.
So now, if the DataFrameReader detects an incorrect value, it will stop and throw an exception.

07-dataframe-schema Python

demo-cluster

```
1  flight_schema_ddl = """FL_DATE DATE, OP_CARRIER STRING, OP_CARRIER_FL_NUM INT, ORIGIN STRING,
2                         ORIGIN_CITY_NAME STRING, DEST STRING, DEST_CITY_NAME STRING, CRS_DEP_TIME INT, DEP_TIME INT,
3                         WHEELS_ON INT, TAXI_IN INT, CRS_ARR_TIME INT, ARR_TIME INT, CANCELLED BOOLEAN, DISTANCE INT"""
```

Command took 0.05 seconds -- by prashant@scholarnest.com at 4/27/2022, 5:35:34 PM on demo-cluster

Cmd 3

```
1  flight_time_raw_df = spark.read \
2                         .format("json") \
3                         .schema(flight_schema_ddl) \
4                         .option("mode", "FAILFAST") \
5                         .load("/FileStore/tables/flight_time.json")
```

Command took 0.59 seconds -- by prashant@scholarnest.com at 4/27/2022, 5:36:18 PM on demo-cluster

Cmd 4

The DataFrameReader mode option can take three values.

1. FAIL FAST

   The fail-fast will throw an error when detecting a malformed record.

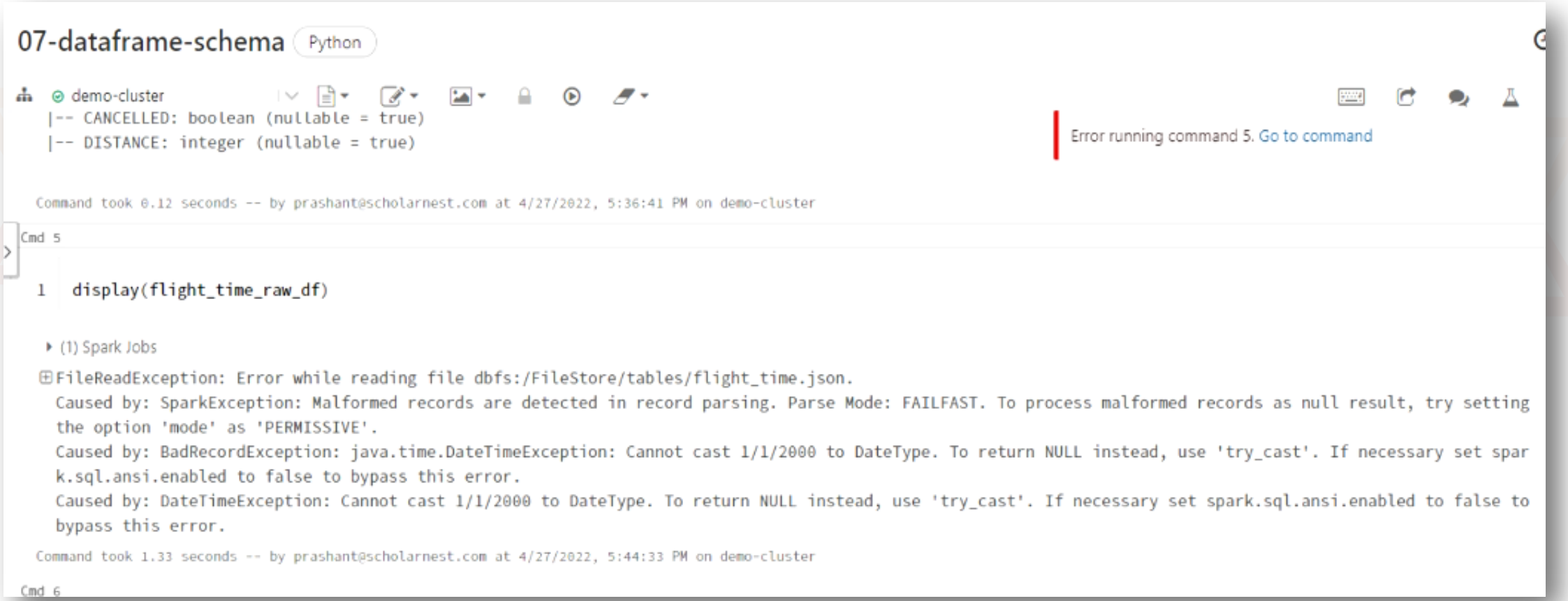
2. DROPMALFORMED

   The DROPMALFORMED will leave the malformed record and load others.

3. PERMISSIVE

   The PERMISSIVE is the default value, and it will set the malformed column to null.

If you try running the display. You see an exception as shown below.
But how do we fix it?
Well, the error is coming because the data is not in the expected format.
We cannot change the file data. But we can change the expected format.



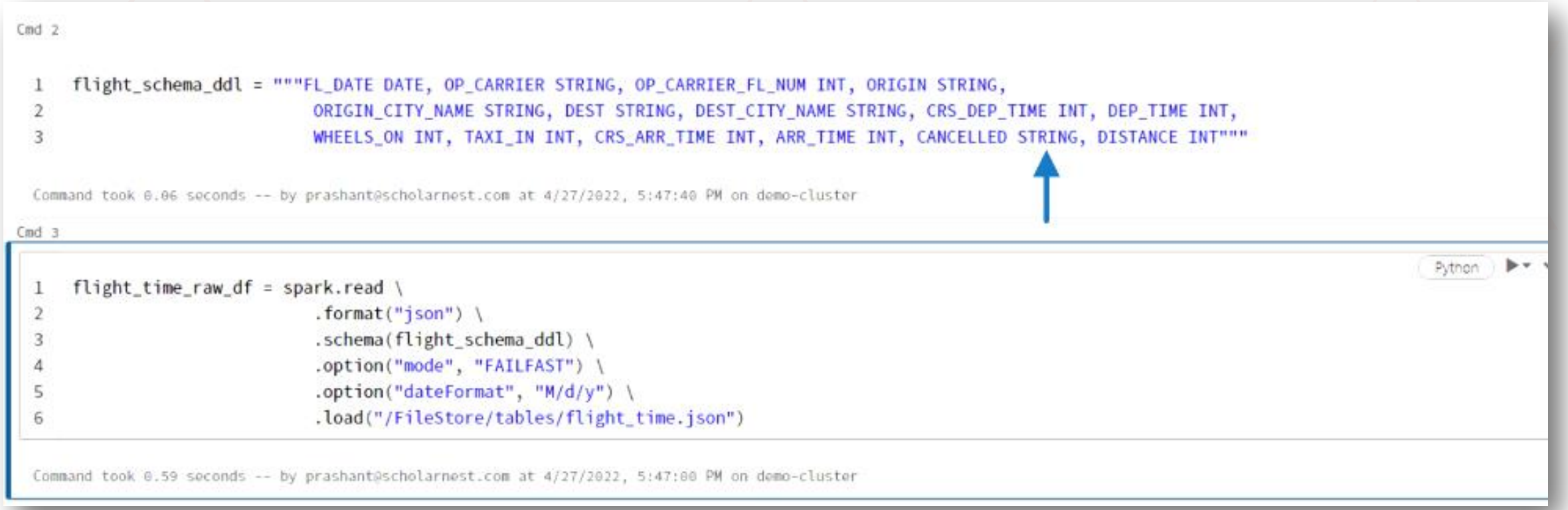
I am adding another DataFrameReader option.
I am setting the expected date format to M/d/y because my file data is in the M/d/y format.
With this change, the FL_DATE column will load correctly.

Cmd 3

```
1  flight_time_raw_df = spark.read \
2                       .format("json") \
3                       .schema(flight_schema_ddl) \
4                       .option("mode", "FAILFAST") \
5                       .option("dateFormat", "M/d/y") <--
6                       .load("/FileStore/tables/flight_time.json")
```

Command took 0.59 seconds -- by prashant@scholarnest.com at 4/27/2022, 5:47:00 PM on demo-cluster

Unfortunately, the DataFrameReader does not allow us to change the expected Boolean format. So I do not have an option to load the zero/one as Boolean.
But that's not a problem, we have a straightforward approach. Set the columns to string and load it.
We can convert it to appropriate data types after loading.
So I have changed the CANCELLED column to a string so we can load it.



Now if you try to check the Dataframe, you can see that it is loaded correctly as shown below.

Cmd 5

```
1  display(flight_time_raw_df)
```

▸ (1) Spark Jobs

	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST	DEST_CITY_NAME	CRS_DEP_TIME	DEP_TIME
1	2000-01-01	DL	1451	BOS	Boston, MA	ATL	Atlanta, GA	1115	1113
2	2000-01-01	DL	1479	BOS	Boston, MA	ATL	Atlanta, GA	1315	1311
3	2000-01-01	DL	1857	BOS	Boston, MA	ATL	Atlanta, GA	1415	1414
4	2000-01-01	DL	1997	BOS	Boston, MA	ATL	Atlanta, GA	1715	1720
5	2000-01-01	DL	2065	BOS	Boston, MA	ATL	Atlanta, GA	2015	2010
6	2000-01-01	US	2619	BOS	Boston, MA	ATL	Atlanta, GA	650	649
7	2000-01-01	US	2621	BOS	Boston, MA	ATL	Atlanta, GA	1440	1446

Truncated results, showing first 1000 rows.

Click to re-execute with maximum result limits.

We have two approaches to defining a Schema.
1. Schema DDL String
2. StructType Object

We saw the first approach.
Now let's define the schema using the StructType object definition.

We have to import Spark types from the pyspark.sql.types package.
Then I can define a variable and set it to StructType(). The StructType() method takes a list of StructField(). You can put a comma and add all the available fields as shown below.

07-dataframe-schema Python

demo-cluster

Cmd 6

```
1  from pyspark.sql.types import *
2
3  flight_schema = StructType([
4      StructField("FL_DATE", DateType()),
5      StructField("OP_CARRIER", StringType()),
6      StructField("OP_CARRIER_FL_NUM", IntegerType()),
7      StructField("ORIGIN", StringType()),
8      StructField("ORIGIN_CITY_NAME", StringType()),
9      StructField("DEST", StringType()),
10     StructField("DEST_CITY_NAME", StringType()),
11     StructField("CRS_DEP_TIME", IntegerType()),
12     StructField("DEP_TIME", IntegerType()),
13     StructField("WHEELS_ON", IntegerType()),
14     StructField("TAXI_IN", IntegerType()),
15     StructField("CRS_ARR_TIME", IntegerType()),
16     StructField("ARR_TIME", IntegerType()),
17     StructField("CANCELLED", IntegerType()),
18     StructField("DISTANCE", IntegerType())
19 ])
```

Command took 0.07 seconds -- by prashant@scholarnest.com at 4/27/2022, 7:21:55 PM on demo-cluster

We use the StructType object in the same way as we used the DDL schema.
Here is the code for the same.
So, you can define the schema using a DDL string or using StructType.

Cmd 7

```
1  flight_time_raw_df_1 = spark.read \
2                         .format("json") \
3                         .schema(flight_schema) \
4                         .option("mode", "FAILFAST") \
5                         .option("dateFormat", "M/d/y") \
6                         .load("/FileStore/tables/flight_time.json")
```

Command took 0.51 seconds -- by prashant@scholarnest.com at 4/27/2022, 7:23:59 PM 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

Writing Spark Dataframes

Requirement:

In the previous lecture, we saw the following requirement.

We have a JSON data set, and you are asked to load this dataframe and create a Spark table.

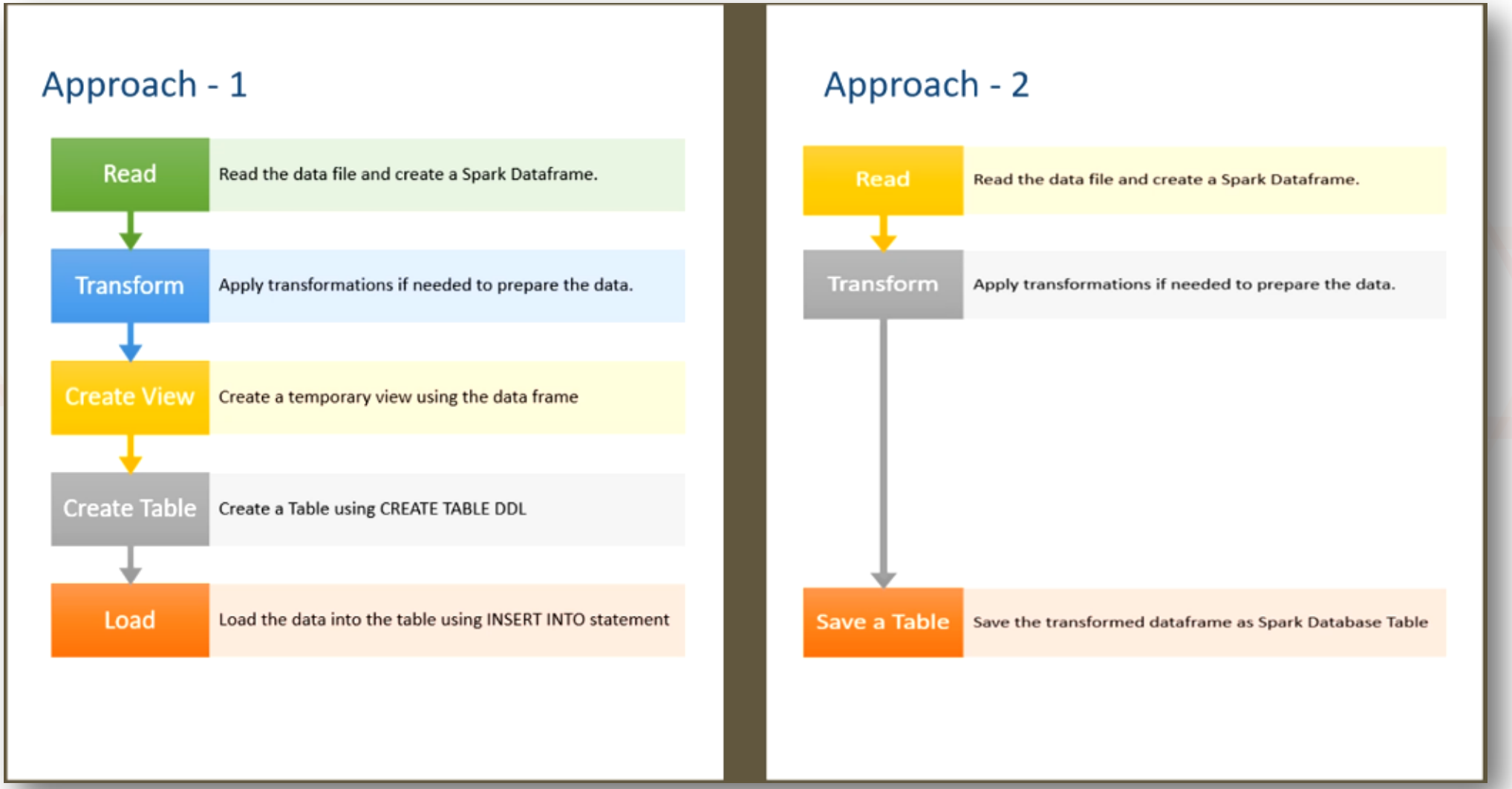
For your reference, here is one sample record, and here are details about the columns shown below.

Flight Delay Data

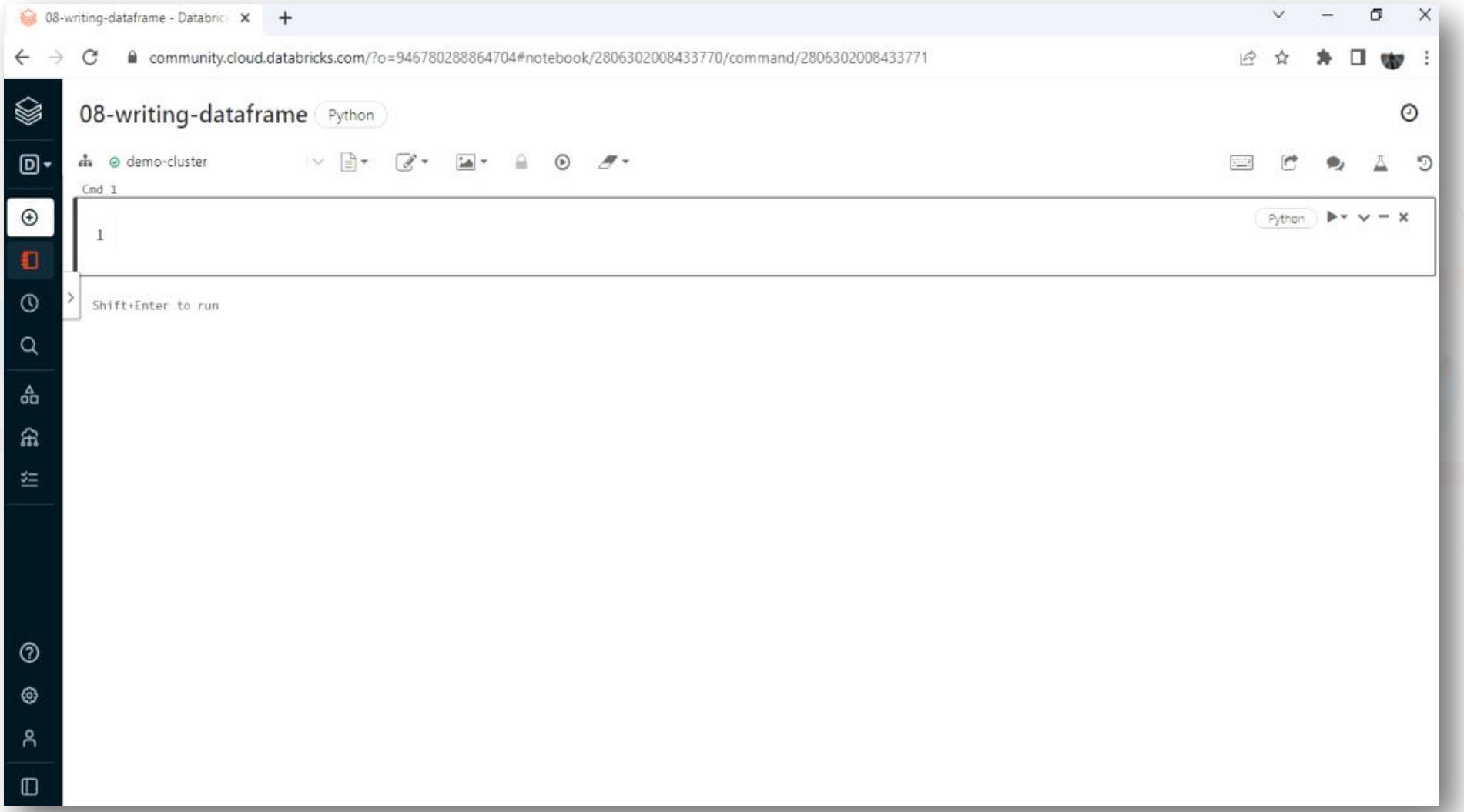

```
{
  "FL_DATE": "1/1/2000",
  "OP_CARRIER": "DL",
  "OP_CARRIER_FL_NUM": 1451,
  "ORIGIN": "BOS",
  "ORIGIN_CITY_NAME": "Boston, MA",
  "DEST": "ATL",
  "DEST_CITY_NAME": "Atlanta, GA",
  "CRS_DEP_TIME": 1115,
  "DEP_TIME": 1113,
  "WHEELS_ON": 1343,
  "TAXI_IN": 5,
  "CRS_ARR_TIME": 1400,
  "ARR_TIME": 1348,
  "CANCELLED": 0,
  "DISTANCE": 946
}
```


1. Flight date in MM/DD/YYYY format
2. Unique carrier code
3. Flight number
4. Origin IATA Airport Code
5. Origin City Name
6. Destination IATA Airport Code
7. Destination City Name
8. Scheduled Departure time in HHMM
9. Actual Departure time in HHMM
10. Flight Duration in HHMM
11. TaxiIn taxi in time, in minutes
12. Scheduled Arrival time in HHMM
13. Actual Departure time in HHMM
14. Was it canceled? 1 for a true, and 0 is false
15. Distance in miles

In order to serve our requirement, we have the following two approaches.
We saw the first approach in the previous documentation, now let us see the second approach in this documentation.



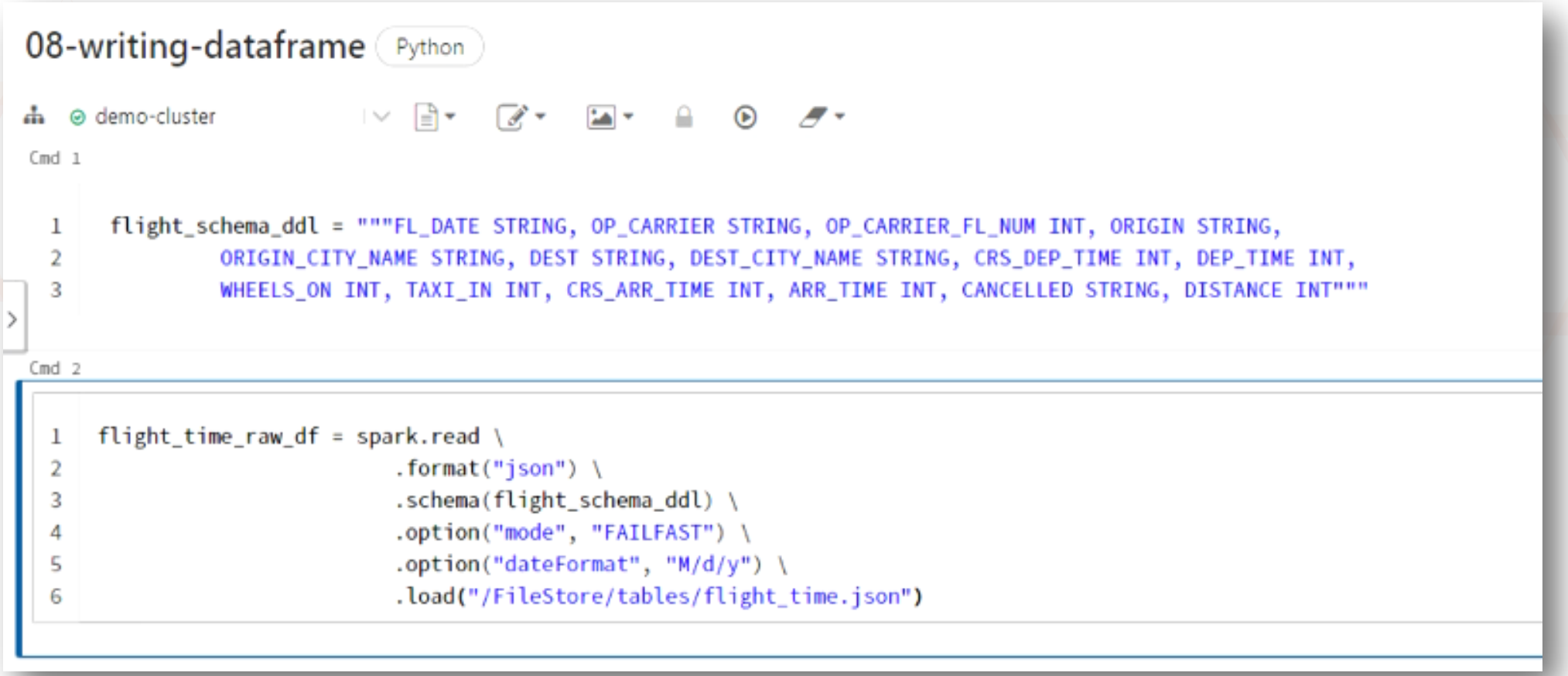
Go to your Databricks Workspace and create a new notebook. **(Reference - 08-writing-dataframe.ipynb)**



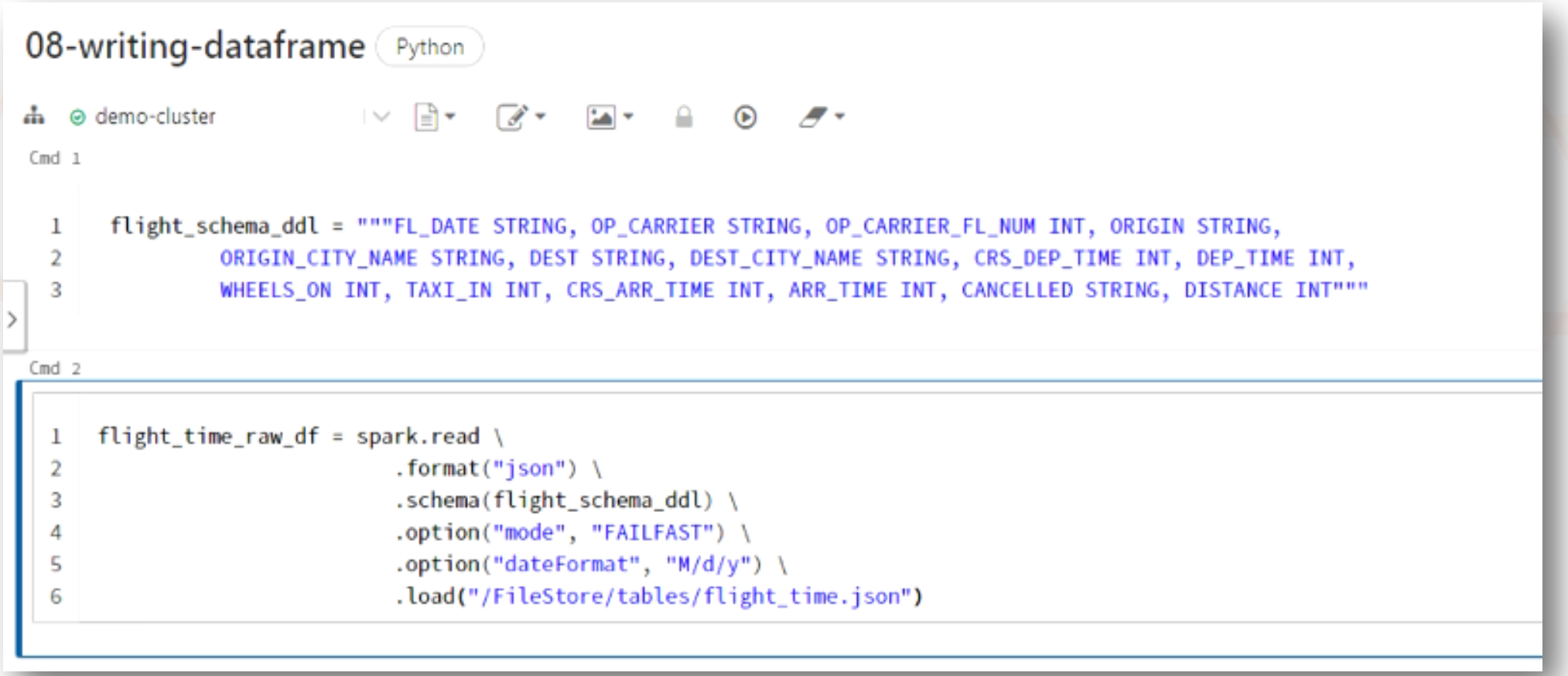
We already wrote the code for the following:

1. Define a flight delay dataset schema.
2. Load data from the file and create a Dataframe.

So I have copy-pasted the code here.



So we have this flight_time_raw_df. And I want to save this dataframe into a Spark table. That is my requirement.
But before saving this data as a Spark table, I want to correct the FL_DATE and CANCELLED data types.



We have already learned about the following:

1. withColumn() transformations
2. to_date() function

In fact, we converted a string field to a date field in an earlier lecture. So I guess you can fix the FL_DATE quickly.

But how do you fix the CANCELLED?

The idea is to look for something that helps you write the following logic:

if CANCELLED == 1 than true else false

Functions in Spark are defined in two places.

1. pyspark.sql.functions package
2. Built-In Spark SQL functions

You can check for the Built-In SQL functions in the documentation, and you will see an if function, and it takes three arguments.

The first argument is an expression that must evaluate as true or false.

The other two arguments are true or false.

I can use this function in my Dataframe code to transform the CANCELLED column from a number to a boolean.

Here is the code to transform the FL_DATE and CANCELLED columns.
I am importing to_date() and expr() functions because I need them. Then I am starting with flight_time_raw_df and transforming two columns using the withColumn() transformation.
The withColumn() takes two arguments. The first argument is the column's name that you want to transform. The second argument is the transformation logic. The logic for FL_DATE is as simple as applying the to_date() function, so we change it from a string to a date.
The logic for the CANCELLED column is also straightforward. We want to apply the if() built-in function as shown here. If CANCELLED is one, make it True else, make it False. Also, my logic for transforming the CANCELLED column is a SQL expression. It is not a dataframe expression. So I must explicitly evaluate the SQL expression. And that's why we use the expr() function.

Cmd 3

```
1  from pyspark.sql.functions import to_date, expr
2
3  flight_time_df = flight_time_raw_df \
4                          .withColumn("FL_DATE", to_date("FL_DATE", "M/d/y")) \
5                          .withColumn("CANCELLED", expr("if(CANCELLED==1, true, false)"))
```

Command took 0.09 seconds -- by prashant@scholarnest.com at 4/29/2022, 1:05:17 AM on demo-cluster

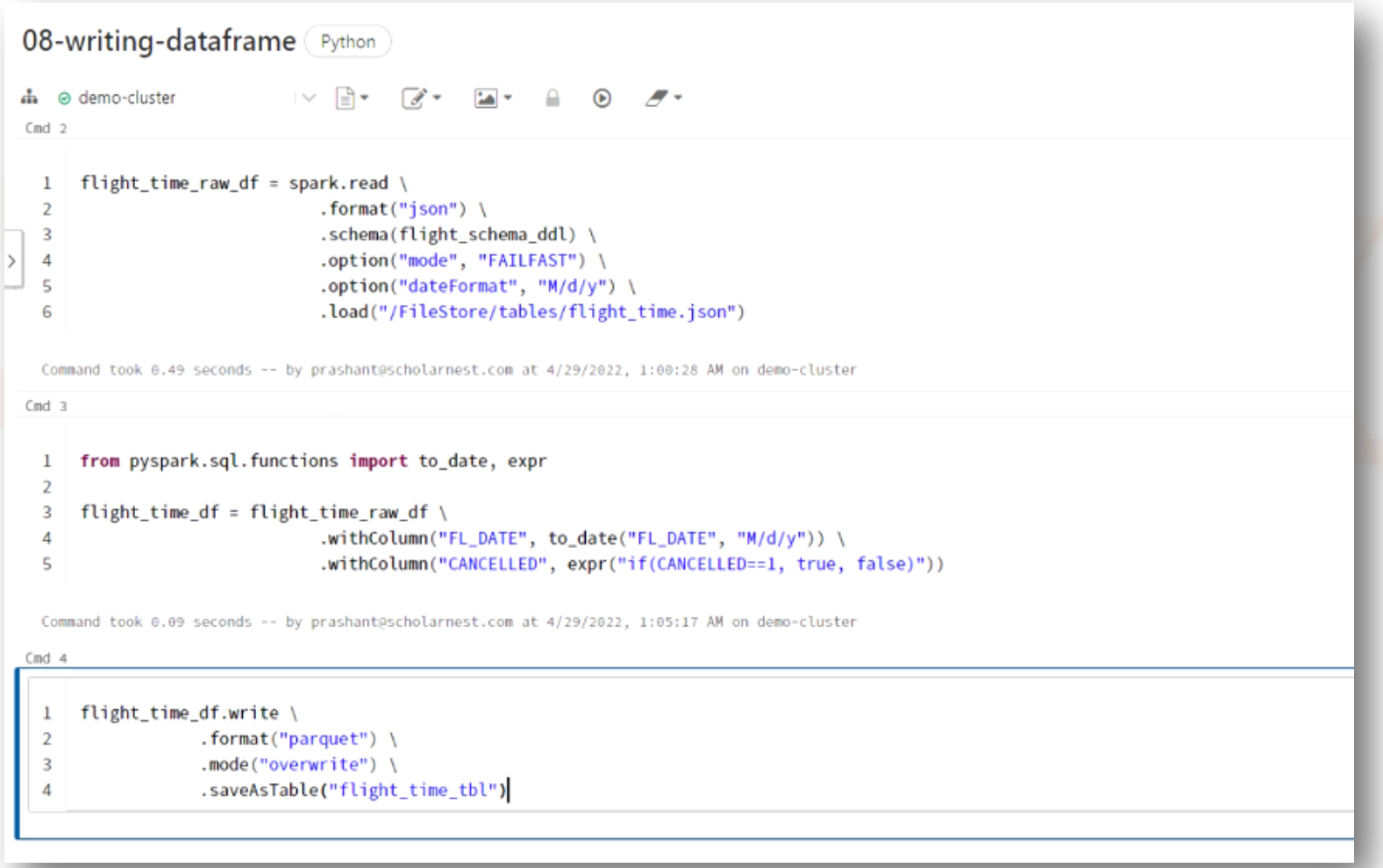
Cmd 4

You can save your Dataframe into a database table using the code shown below.

Cmd 4

```
1  flight_time_df.write \
2              .format("parquet") \
3              .mode("overwrite") \
4              .saveAsTable("flight_time_tbl")
```

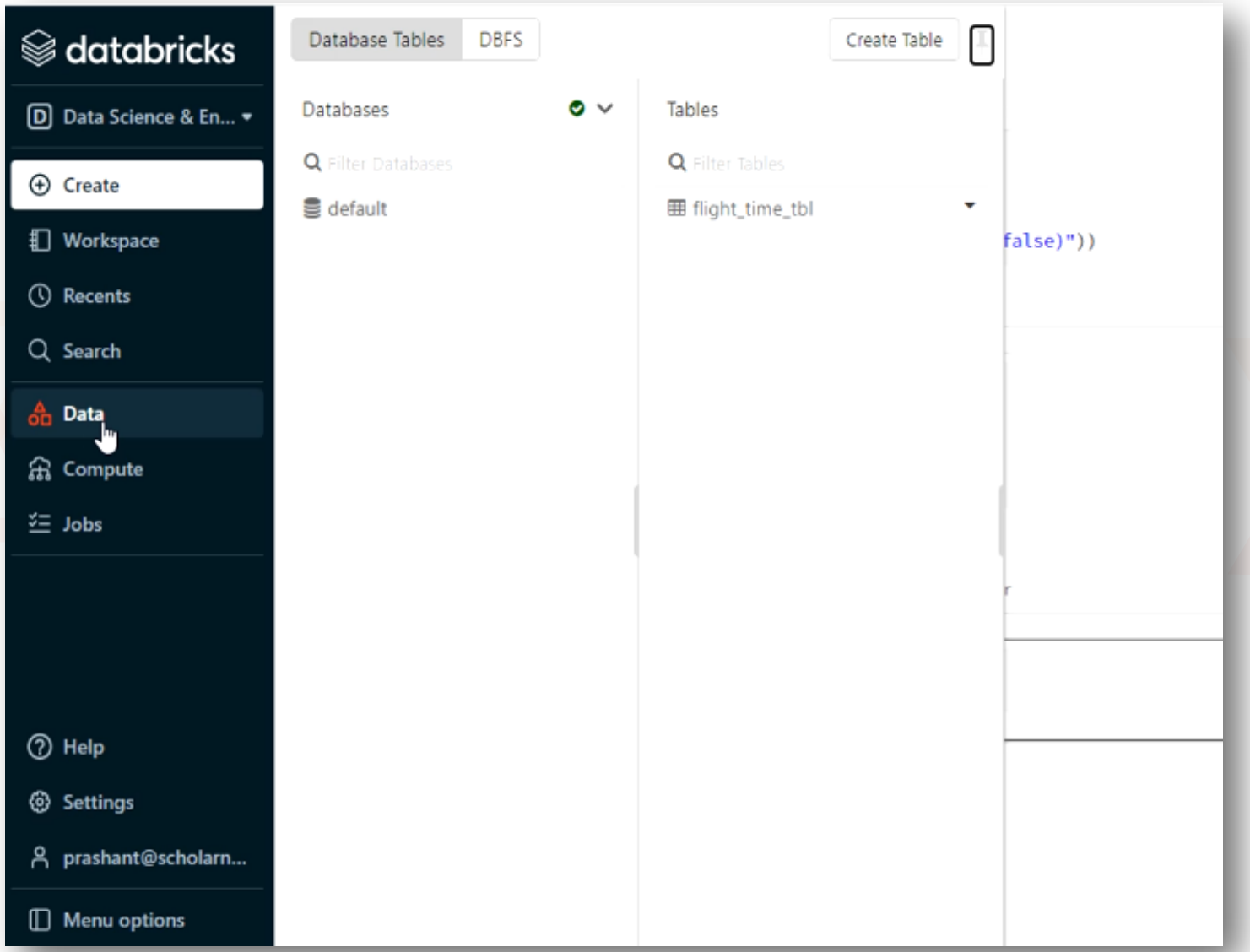
If you look at the code, it looks similar to the DataFrameReader. So you can see the spark.read here. And you can see the dataframe.write here. The first code block is reading data from a source and creating a Dataframe. The second code block is from writing Dataframe to a target. The first one starts with a spark session and access the DataFrameReader using the read attribute. The second one starts with a Dataframe and access the DataFrameWriter using the write attribute. Once you have a DataFrameWriter, the rest of the code structure is the same as the DataFrameReader code. We can set the format, set some options, and save it.



DataFrameWriter supports the following modes:

1. append: Append contents of this DataFrame to existing data.
2. overwrite: Overwrite existing data.
3. error or errorifexists: Throw an exception if data already exists.
4. ignore: Silently ignore this operation if data already exists.

If you look at the data section, you will see a table is created for your Dataframe.



You can also describe the table from your notebook as shown below.
And you will see that both FL_DATE and CANCELLED columns are rectified as per your requirement.

```
1  %sql
2  describe extended flight_time_tbl
```

	col_name	data_type	comment
1	FL_DATE	date	null
2	OP_CARRIER	string	null
3	OP_CARRIER_FL_NUM	int	null
4	ORIGIN	string	null
5	ORIGIN_CITY_NAME	string	null
6	DEST	string	null
7	DEST_CITY_NAME	string	null

Showing all 30 rows.

Command took 0.48 seconds -- by prashant@scholarnest.com at 4/29/2022, 1:09:48 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

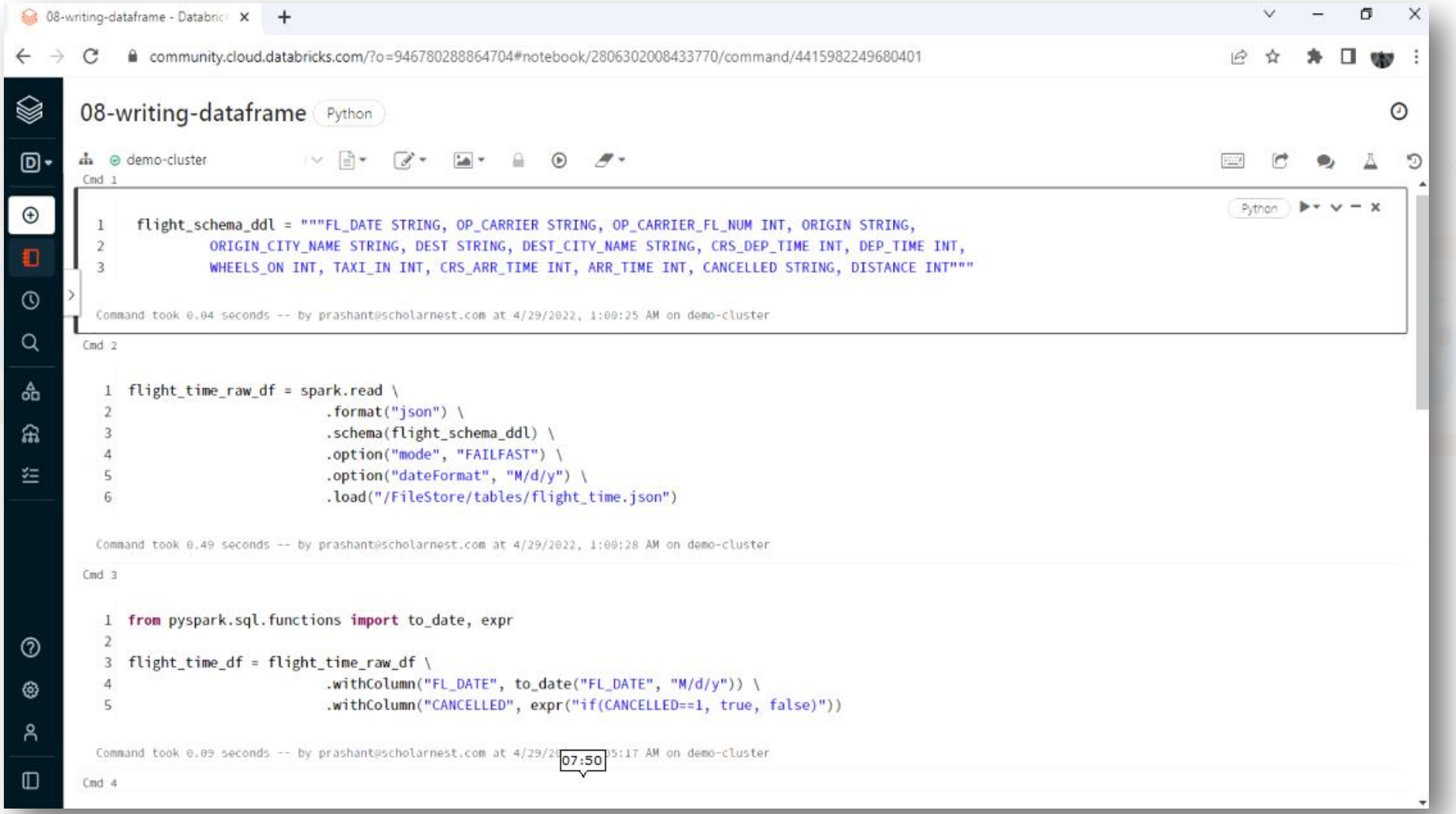
Managed Vs External Tables

Spark allows you to create two types of tables.

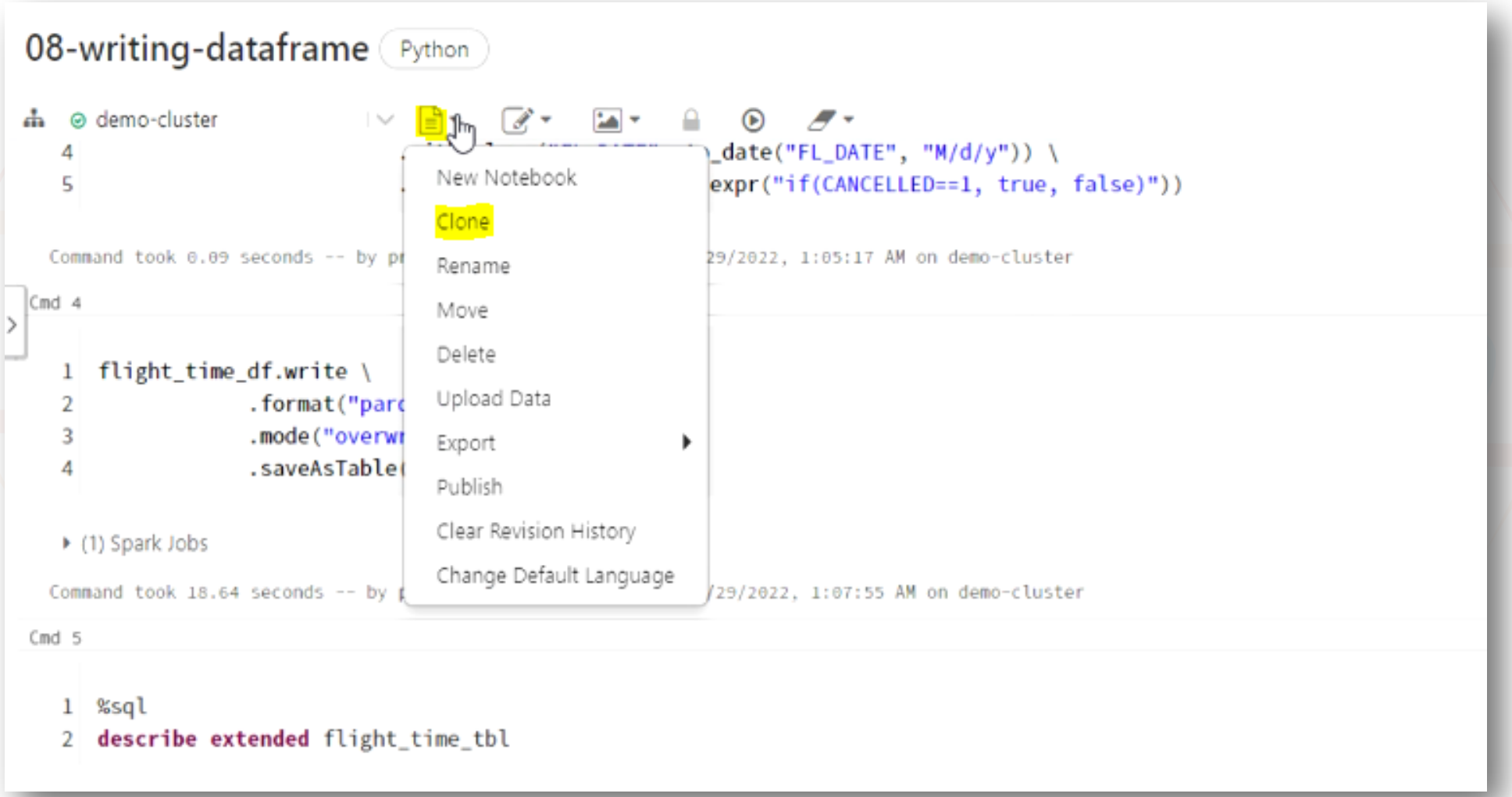
1. Managed Tables
2. External Tables

Go to your databricks workspace and open an existing workbook **(Reference - 08-writing-dataframe)**.

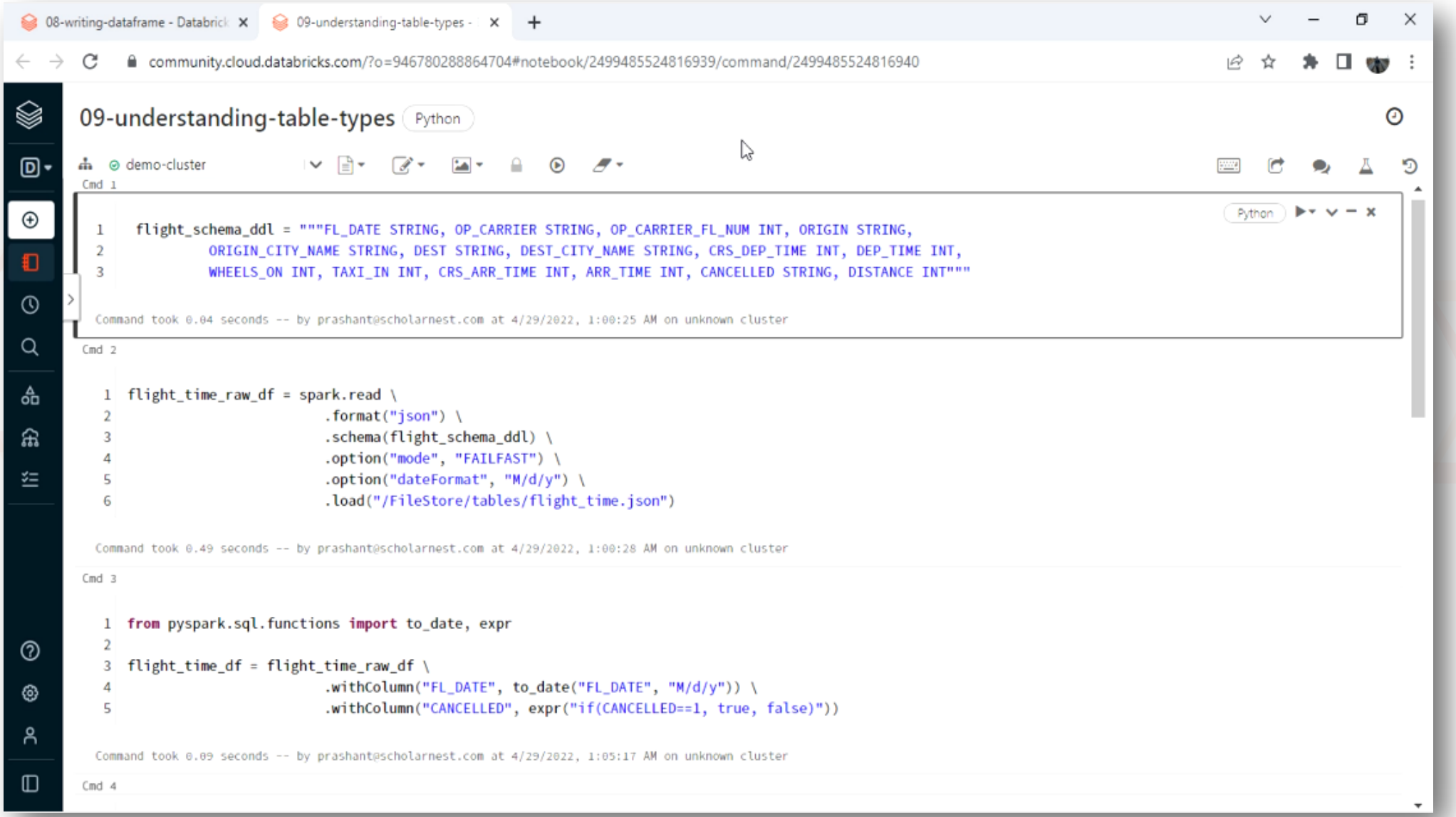
In this example, we learned to create a database table.



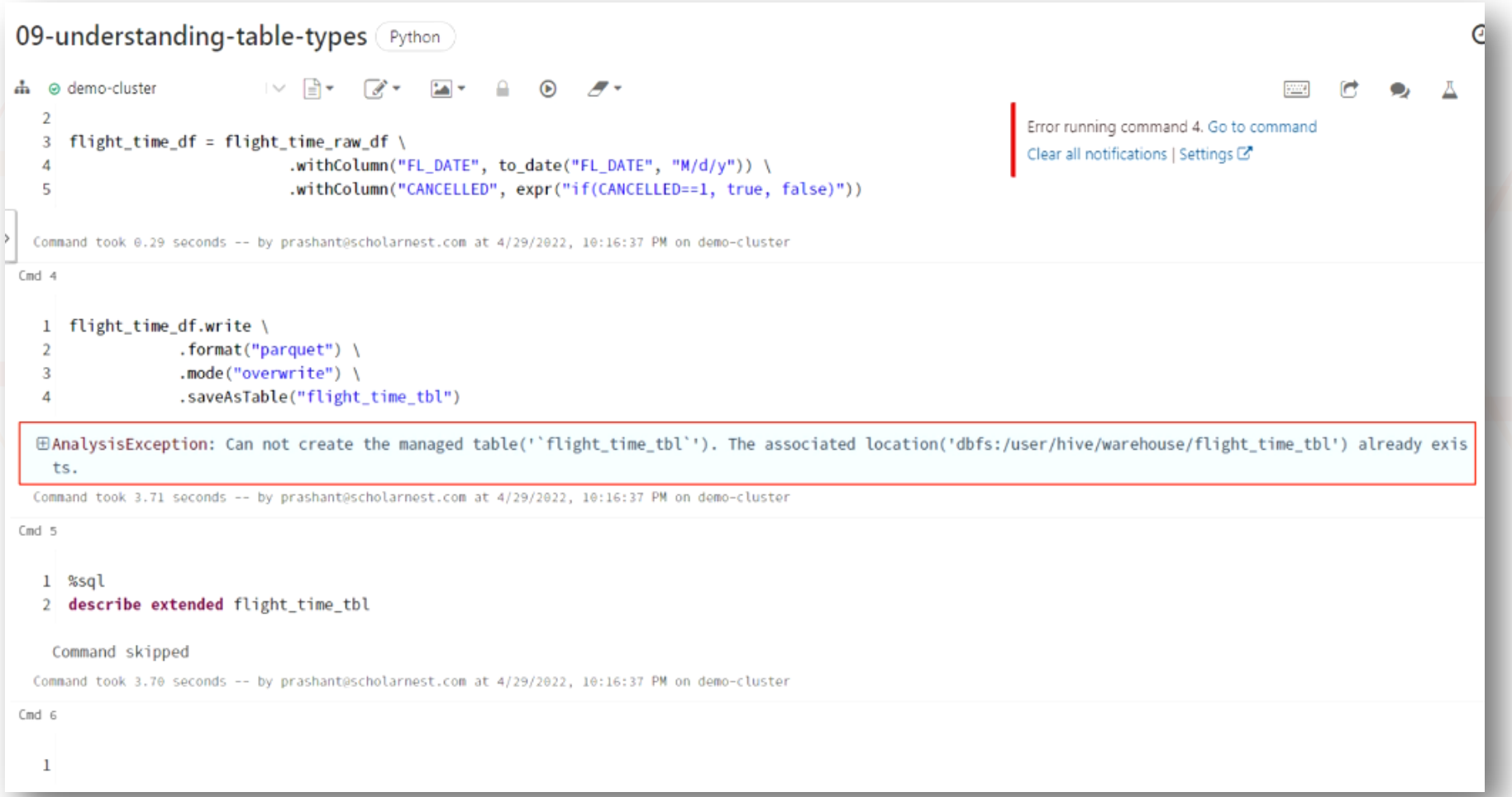
Click the file menu and clone your notebook as shown below.



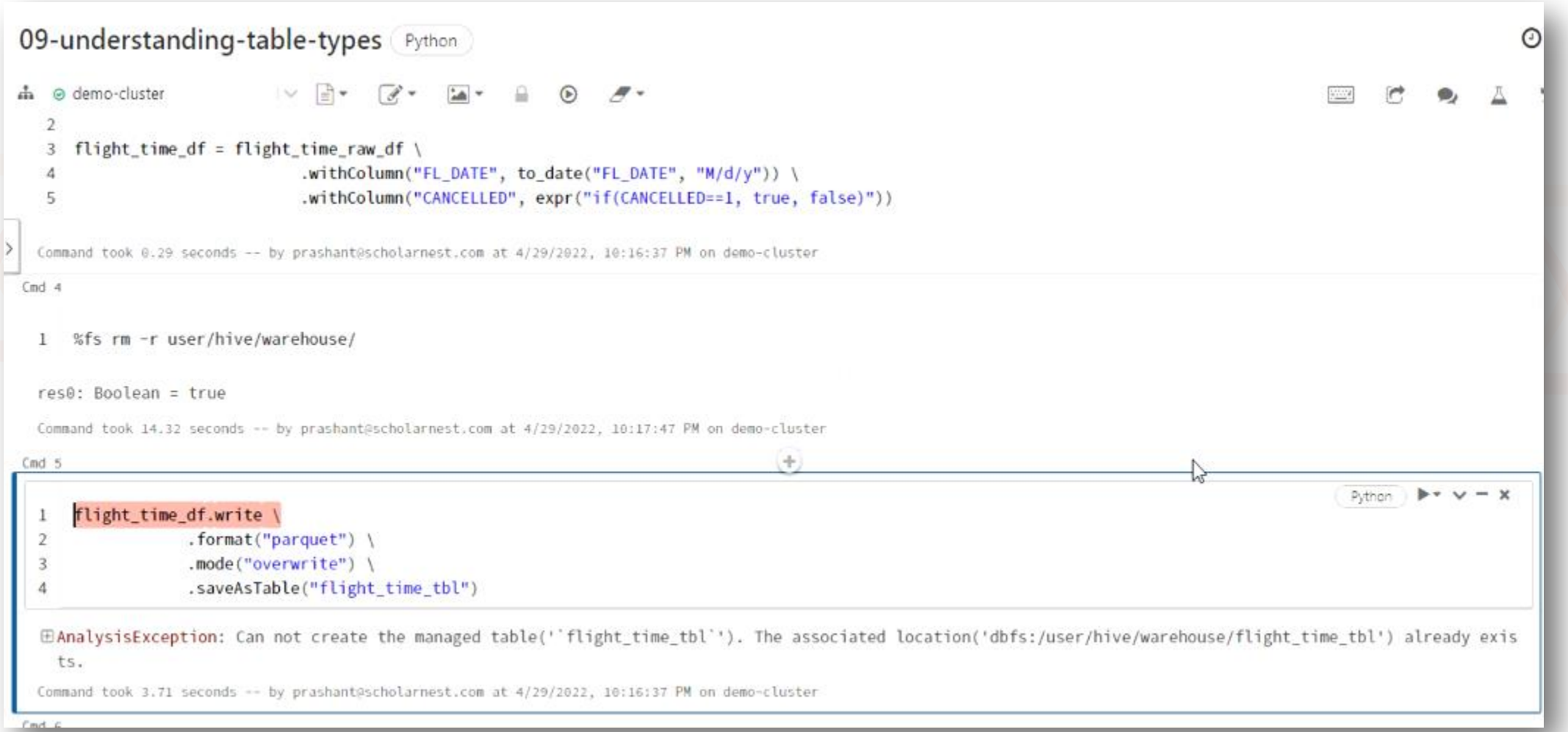
Once cloned, rename your notebook and we will start working on this notebook.
(Reference - 09-understanding-table-types.ipynb)



If you try running this notebook once. You might see an error as shown below.
Your cluster was terminated, and you created a new cluster. So, you will see this error.
Because the metadata and compute layers were removed by the Databricks. But the data files are still there.



You can fix the error using the %fs rm -r command, add a new cell above and write the %fs command as shown below.
Now, if you try to run the cells it will work perfectly fine.



Scroll down to see the output of the describe command, and you will see the table type.
This table is a managed table.
So by default, the saveAsTable() method created a managed table.
You can also create a table using the create table DDL statement.
That one also creates a managed table by default.

We have two properties of a managed table:

1. Spark creates managed table at a predefined warehouse location.
2. Spark manages table metadata and table data together.

You might ask the following questions about the first point:

1. Can we define this location?
2. Can we change it?

The answer is YES-NO.

You can configure the default warehouse directory location using the spark.SQL.warehouse.dir configuration. You can go to Spark UI environment tab and search for the spark.SQL.warehouse.dir configuration.

The value for this configuration will be redacted for security reasons.

But the spark.SQL.warehouse.dir defines the default location for the managed tables.

You can set this configuration before starting the Spark cluster.

There is two things to learn here:

1. Spark creates all the managed tables at a fixed warehouse directory location.
2. You can set the warehouse directory location using the spark.SQL.warehouse.dir configuration.

We have two properties of a managed table:

1. Spark creates managed table at a predefined warehouse location.
2. Spark manages table metadata and table data together.

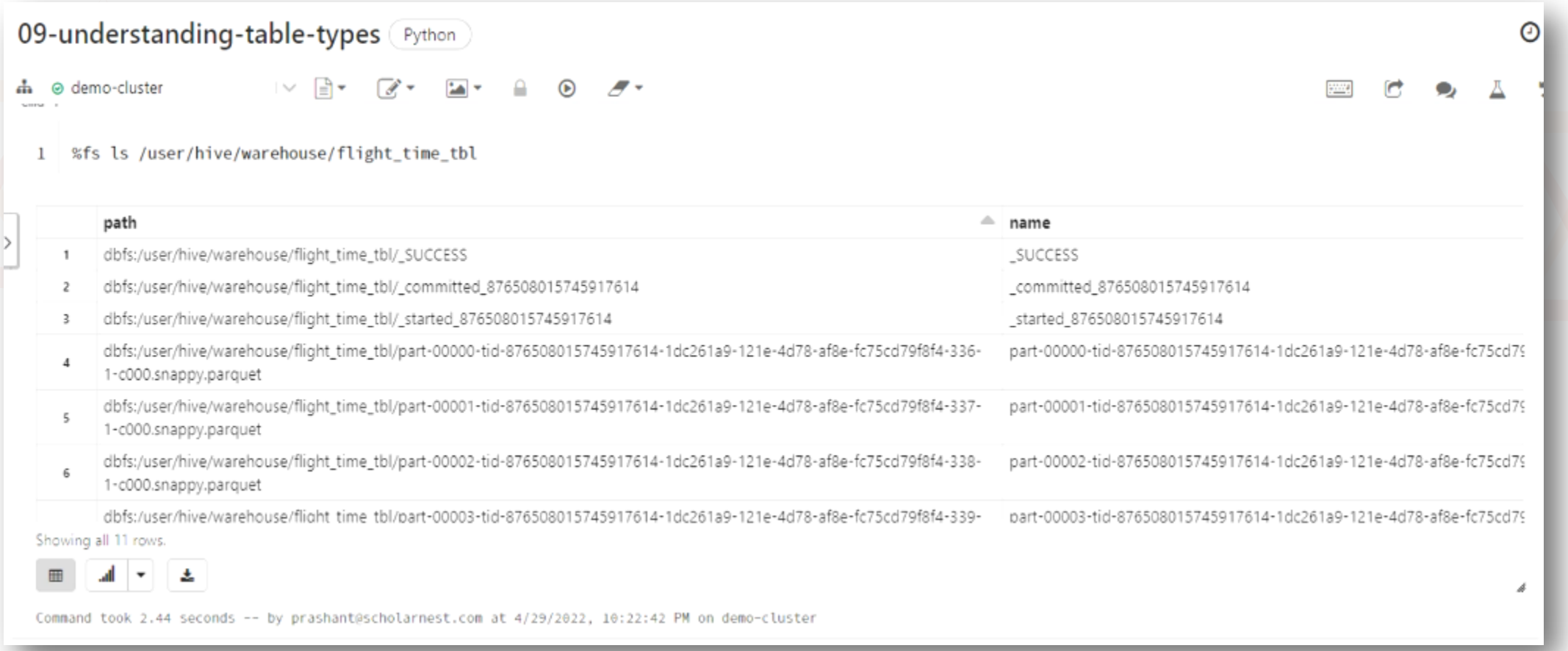
Now, moving on to the second topic.

Spark manages table metadata and table data together.

It means, Spark creates table data and metadata both when you create a managed table.

Similarly, Spark deletes the table data and metadata both when you drop the table.

We created the flight_time_tbl. So spark created data in the */user/hive/warehouse/flight_time_tbl* directory. And Spark also created metadata in the metadata store. You can check the data directory using the %fs ls command as shown below. You will see the results, there are some parquet files here. These are your data files. Spark stored my data in the parquet files.



We have two methods to query the metadata store:

1. SQL Commands
2. Spark Catalog API

Now let me drop the table.
It was a managed table, so Spark will delete the table metadata and also delete the data files.

Cmd 10

```
1  %sql
2  drop table flight_time_tbl
```

OK

Command took 4.03 seconds -- by prashant@scholarnest.com at 4/29/2022, 10:25:23 PM on demo-cluster

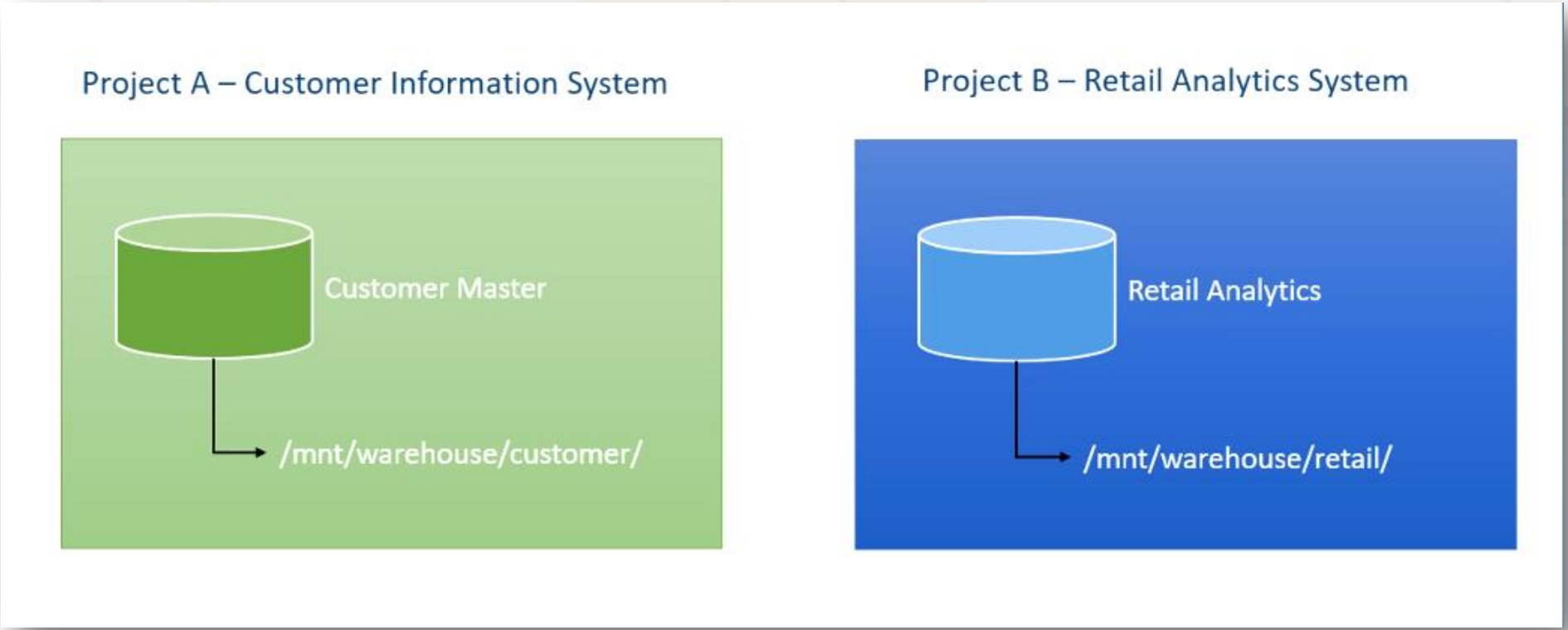
Now you can run your $fs ls command, and you will see an *FileNotFoundException*. So the table data is gone. Let me re-run the show tables command. And you can see the Query returned no results. So the metadata is also gone.

Now let's talk about the external tables.
The external tables are also referred to as unmanaged tables.
However, the external table is a more popular name.

So what are external tables?

Spark offers an external table so you can create a spark table over shared external data.
External tables are a mechanism for sharing data across projects or storage layers.

Assume you have two projects. The first project collects and manages customer information. So you have a customer master table in the project. The table data is stored in some directory in the project warehouse directory location. Now you are starting a new project for retail data analysis. This project has a different warehouse directory location where you store the data.
We have two separate projects and two different data stores.
Both the projects belong to the same organization, but we have two different data owners.
Project A owns the customer data, and project B owns the retail transactions.
But you have a requirement in Project B to access and use the customer master table.



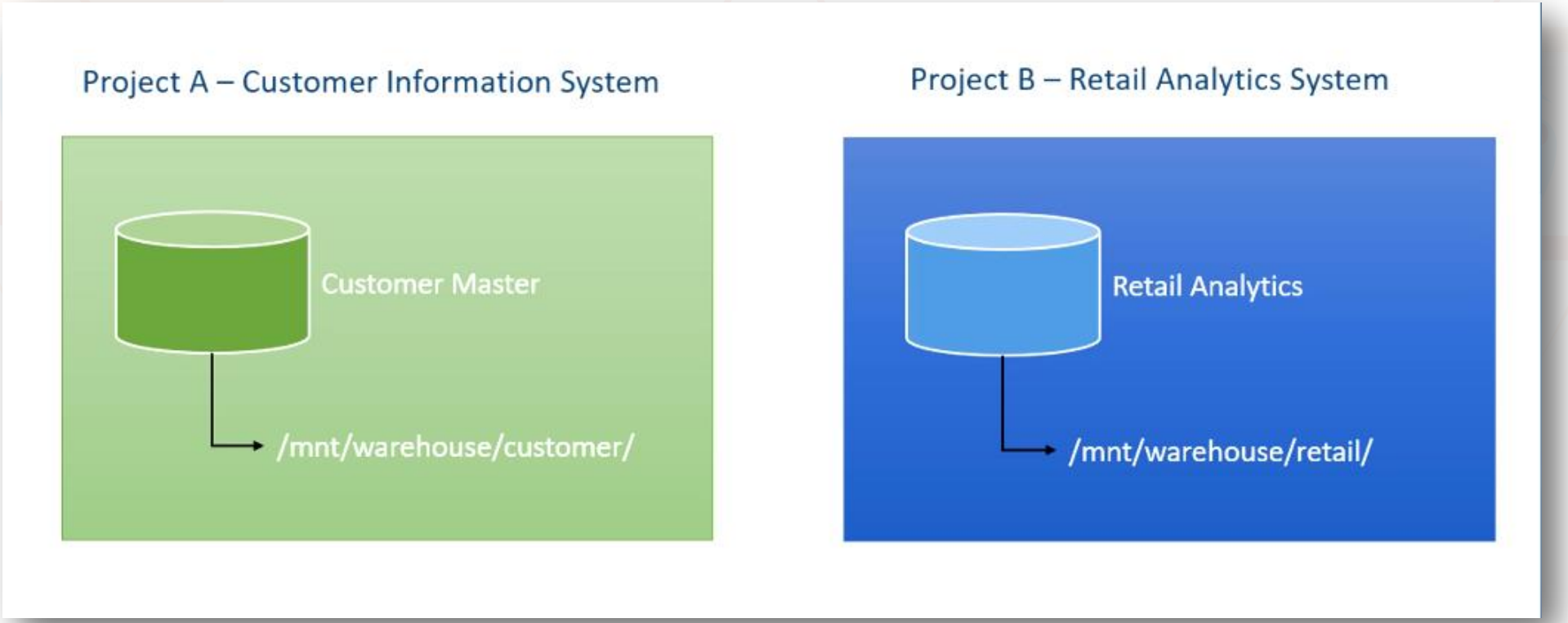
We have two approaches to solve our problem:

1. Copy the customer master data from project A to project B and use it.
2. Create an external table from project A to project B and use the same dataset.

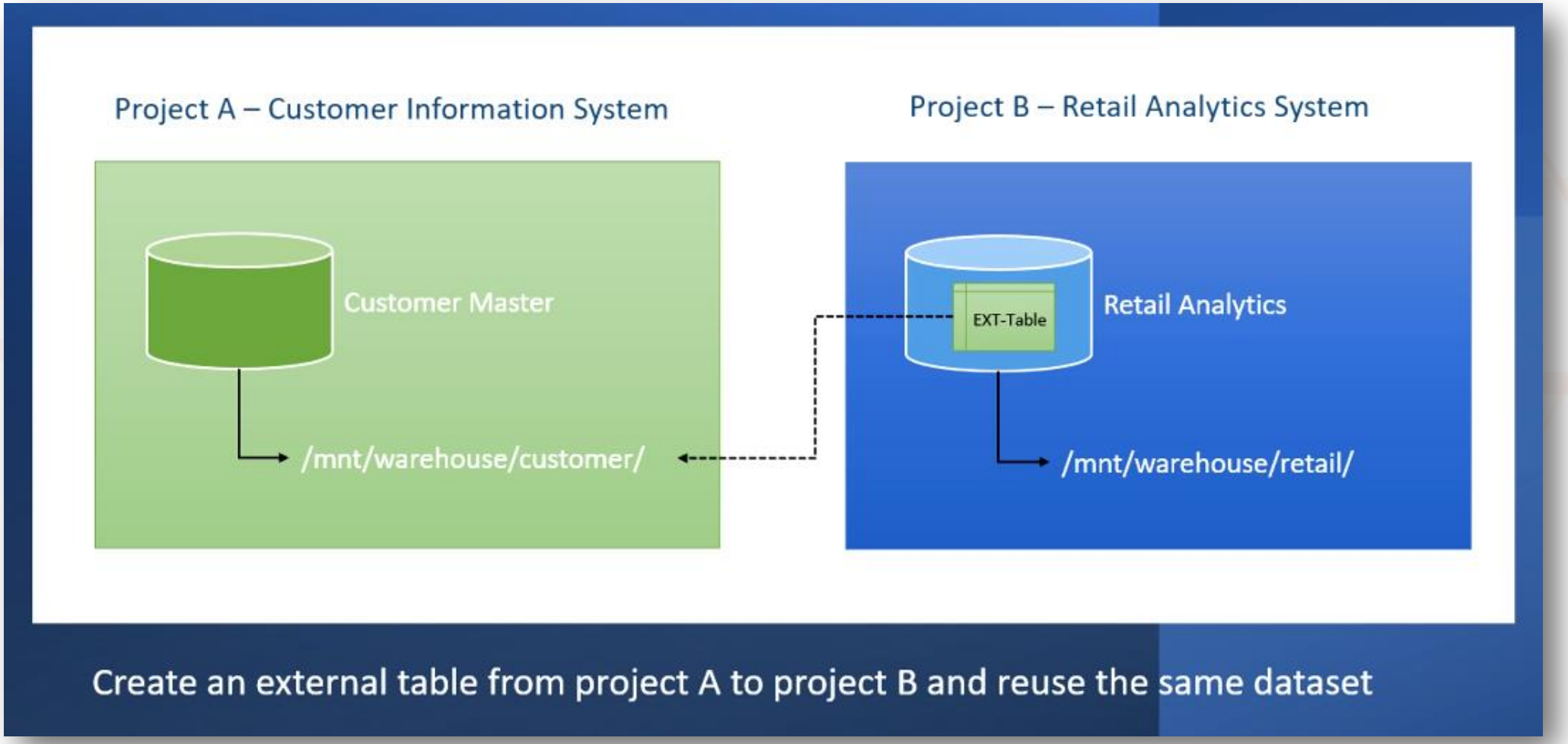
The first approach is problematic.

Copying data at two places requires extra space, and you also need to keep both copies in sync.

So it is much better to use the second approach.



We can create an external table from project A to Project B and use the same data.
So the idea behind the external table is to share data across ownerships and storage layers.
Project A could be using a different storage layer than project B.
But they can share the same data via external tables as long as Spark supports both storage layers.



I dropped the flight_time_tbl table, so let me recreate it.
Assume the flight_time_tbl is a Project A managed table. And Project B wants to reuse my table.
So they can create an external table on my flight_time_tbl and read data from it.
How to create an external table? They can use create table DDL as shown below.

Cmd 11

```
1  %sql
2  create table if not exists ext_flight_time_tbl like flight_time_tbl
3  location "/user/hive/warehouse/flight_time_tbl"
```

```
OK

Command took 0.79 seconds -- by prashant@scholarnest.com at 4/30/2022, 6:26:10 PM on demo-cluster
```

What is the LIKE in the statement given below? You can use the LIKE clause to copy the schema from an existing table. So it will look into the column definitions of flight_time_tbl and create the same columns and data types for ext_flight_time_tbl.
So the LIKE clause is a quick way to copy the structure of an existing table.
This statement will create a managed table. But we wanted to create an external table.
You can set the location, and spark will create an external table. We want to share the same data, so we should point this ext_flight_time_tbl to the same data files where the flight_time_tbl stores its data.

Now you can describe it and check the table type and location.
So it is an external table, and it points to the same location. So, it means we have two tables:
1. flight_time_tbl
2. ext_flight_time_tbl

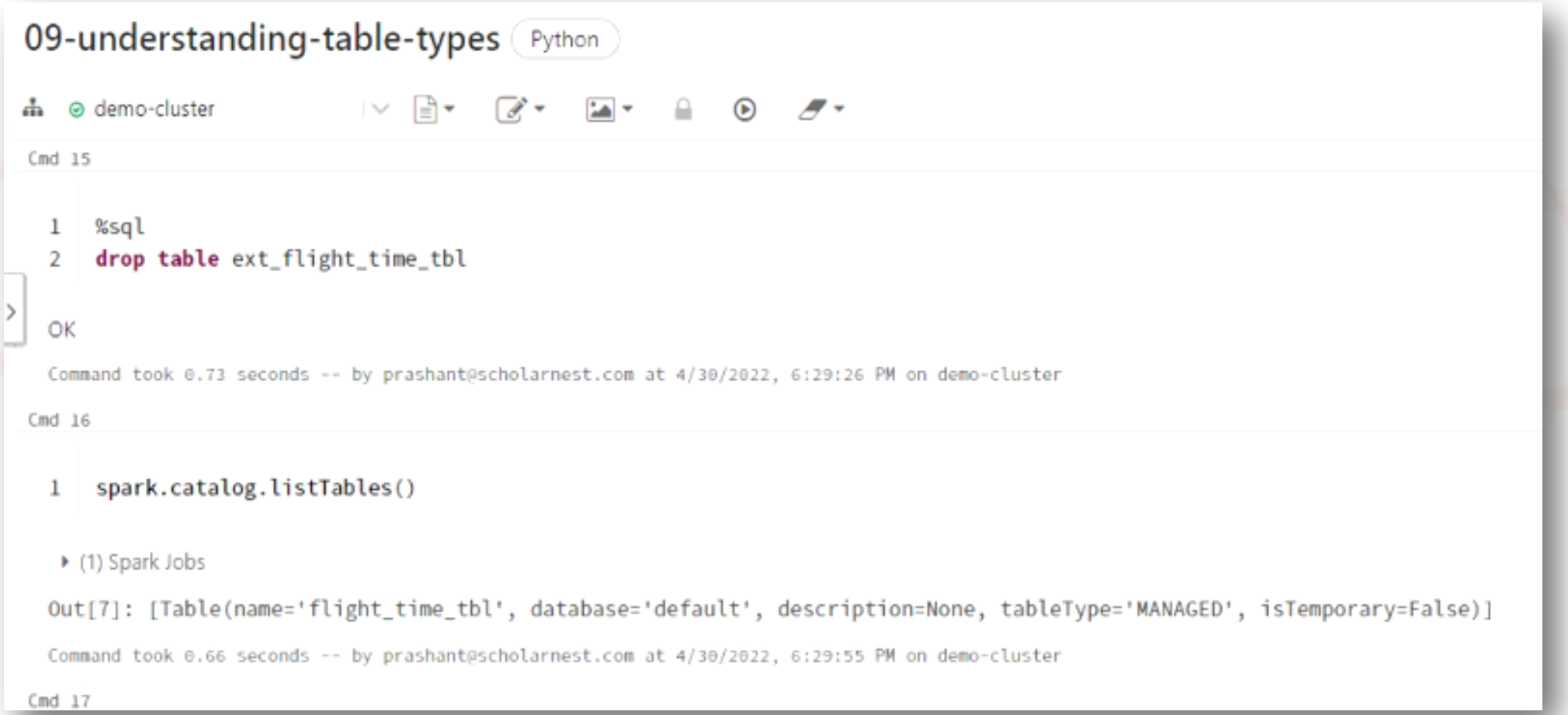
You can even check the catalog API, we have two tables:

1. flight_time_tbl
2. ext_flight_time_tbl

```
1  spark.catalog.listTables()

▸ (2) Spark Jobs

Out[6]: [Table(name='ext_flight_time_tbl', database='default', description=None, tableType='EXTERNAL', isTemporary=False),
 Table(name='flight_time_tbl', database='default', description=None, tableType='MANAGED', isTemporary=False)]

Command took 1.12 seconds -- by prashant@scholarnest.com at 4/30/2022, 6:27:33 PM on demo-cluster
```


You can count the records in the ext_flight_time_tbl. I created the table and haven't loaded the data in this external table. But you can see the record count. Because the data already exists at the given location, and the external table simply points to the same data.
You can use the external table in the same way as a managed table. I mean, you can read it, update it, overwrite it or do whatever you want. But you cannot drop it. I mean, you can drop an external table, but Spark will delete the metadata only.

Try the DROP statement on the external table. And now if you query the catalog, it will show you only one table.



So the external table is gone. But data is still there. And it makes sense also because the original table exists. If you query the original table as shown below, you can see the records here.

Summary

Spark Tables

1. Managed Tables
   - Created at a predefined warehouse directory location
   - Drop Table will delete data and metadata both
2. External Tables
   - Shared data across project/storage
   - Set the location/path to create an external table
   - Allows all operations such as append/overwrite
   - Best practice is to avoid append/overwrite external tables
   - Drop table will delete metadata only

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

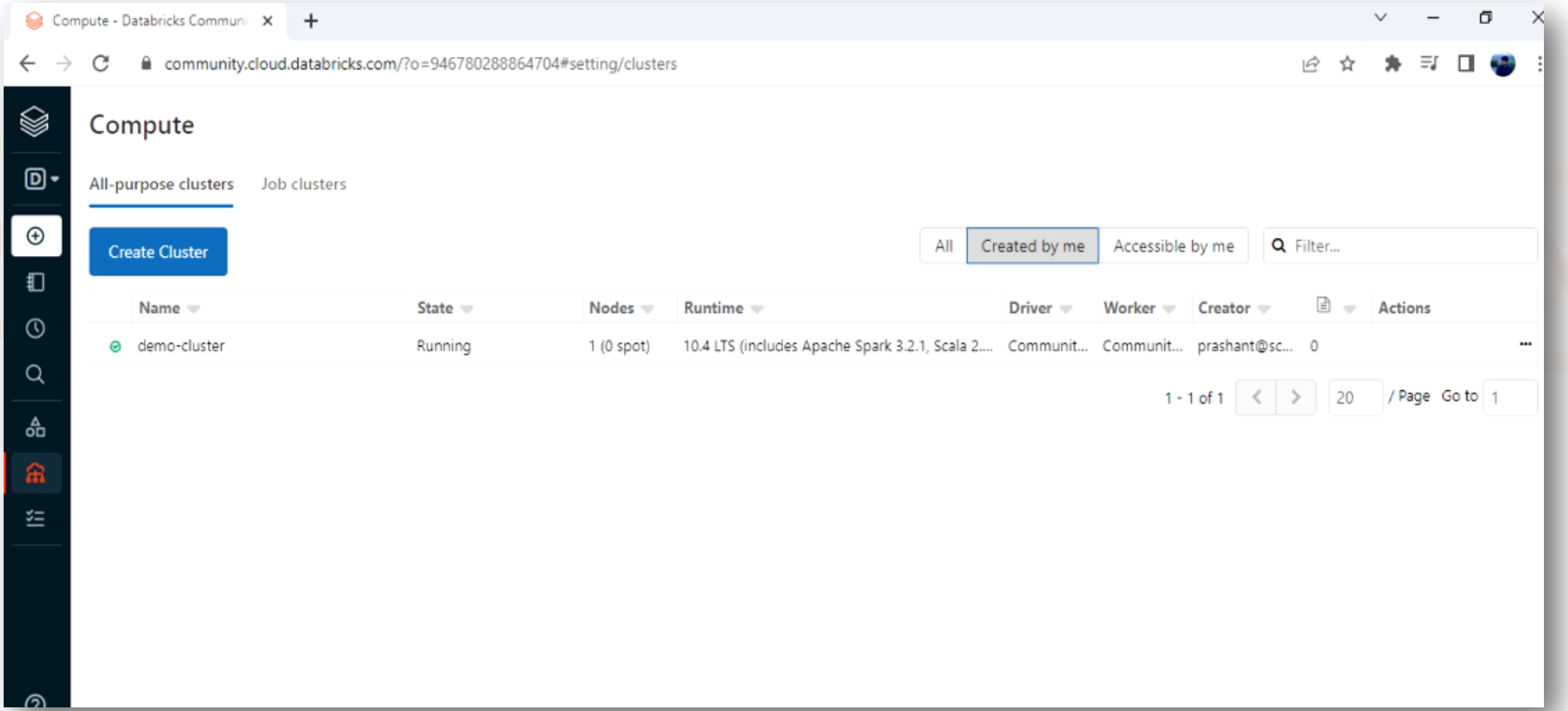
Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Spark Web UI - Introduction

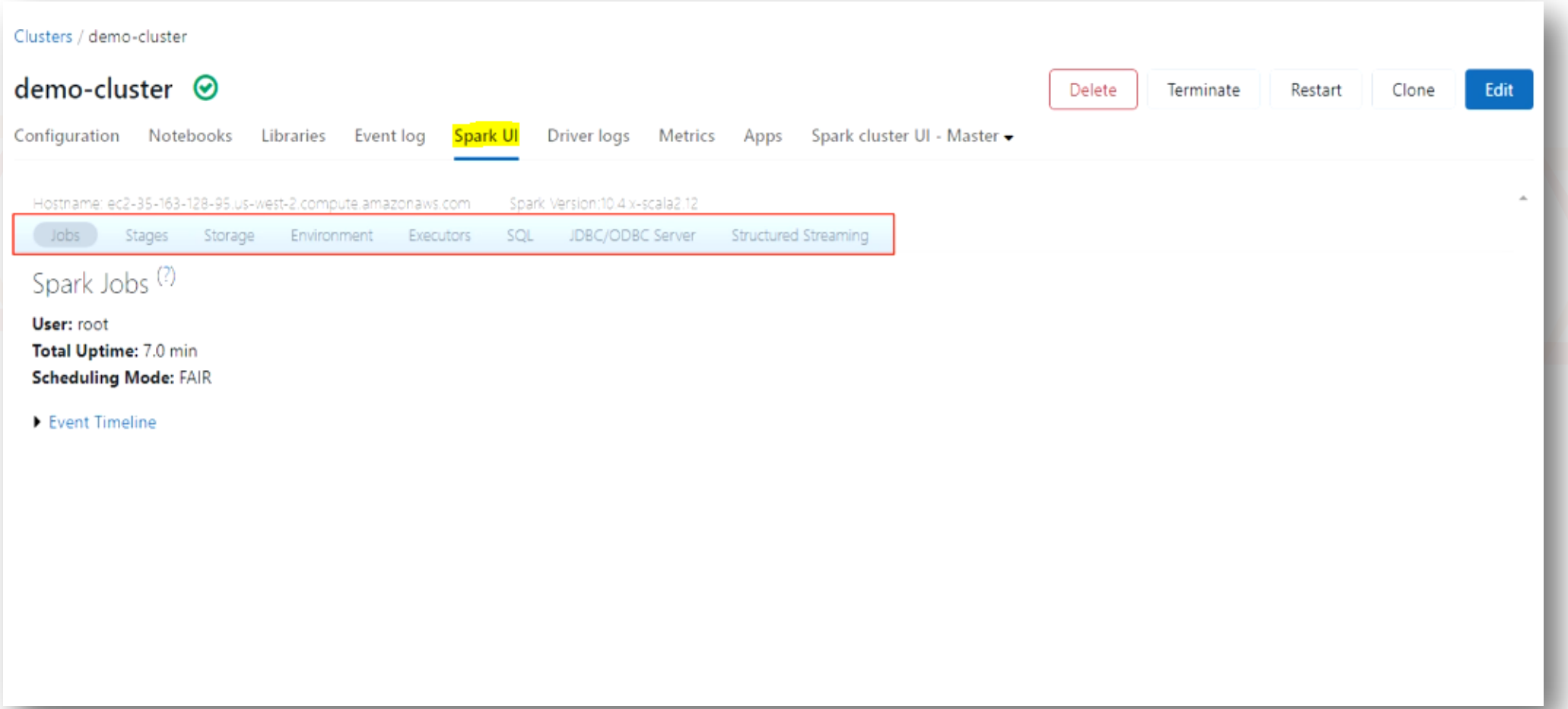
Spark Web UI

The go-to place to monitor the status and resource consumption of your Spark cluster

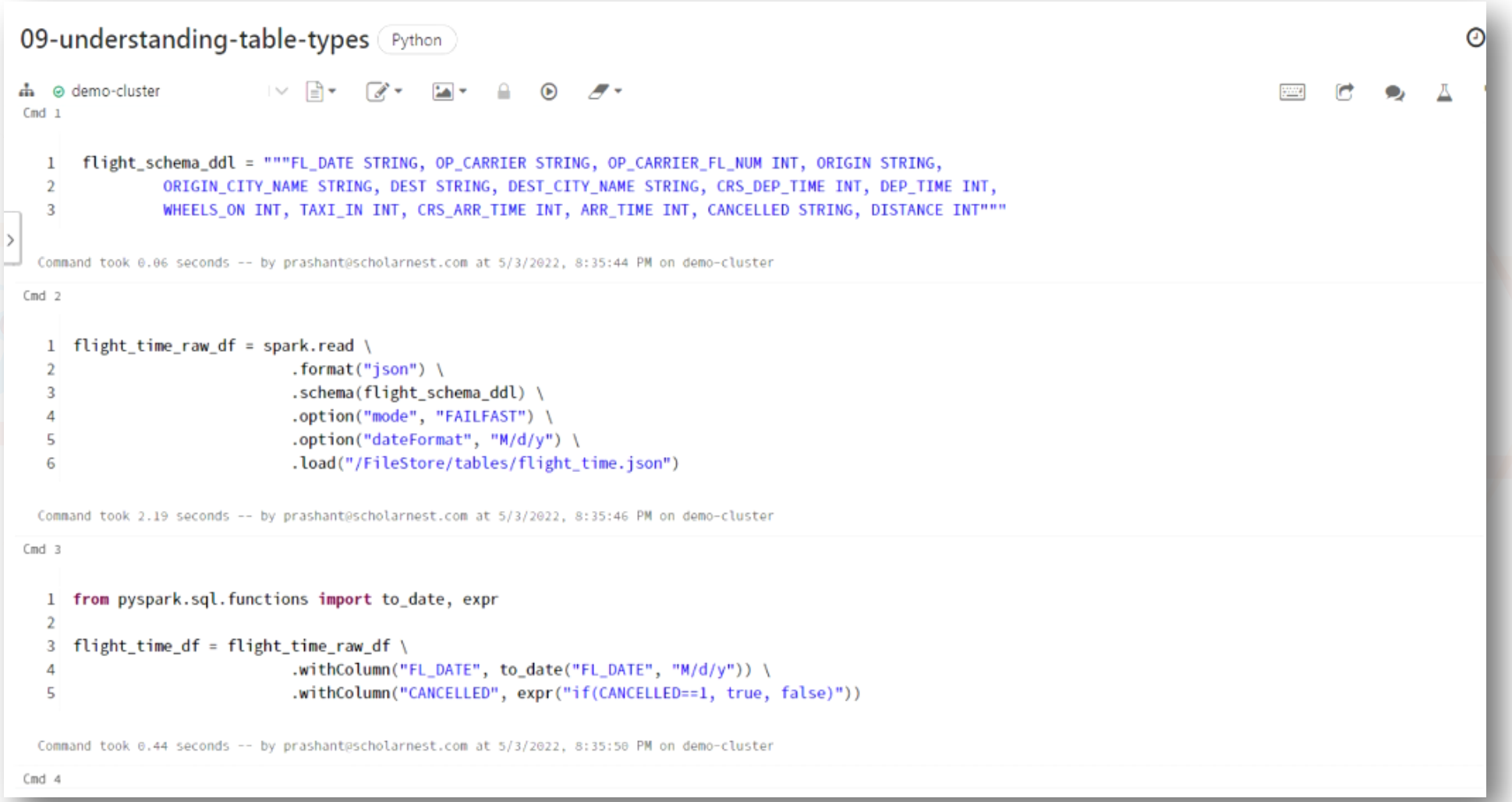
Go to your Databricks workspace.
Now, go to compute menu and start a cluster.
I already have one cluster running.
Click the cluster and go to the cluster details page.



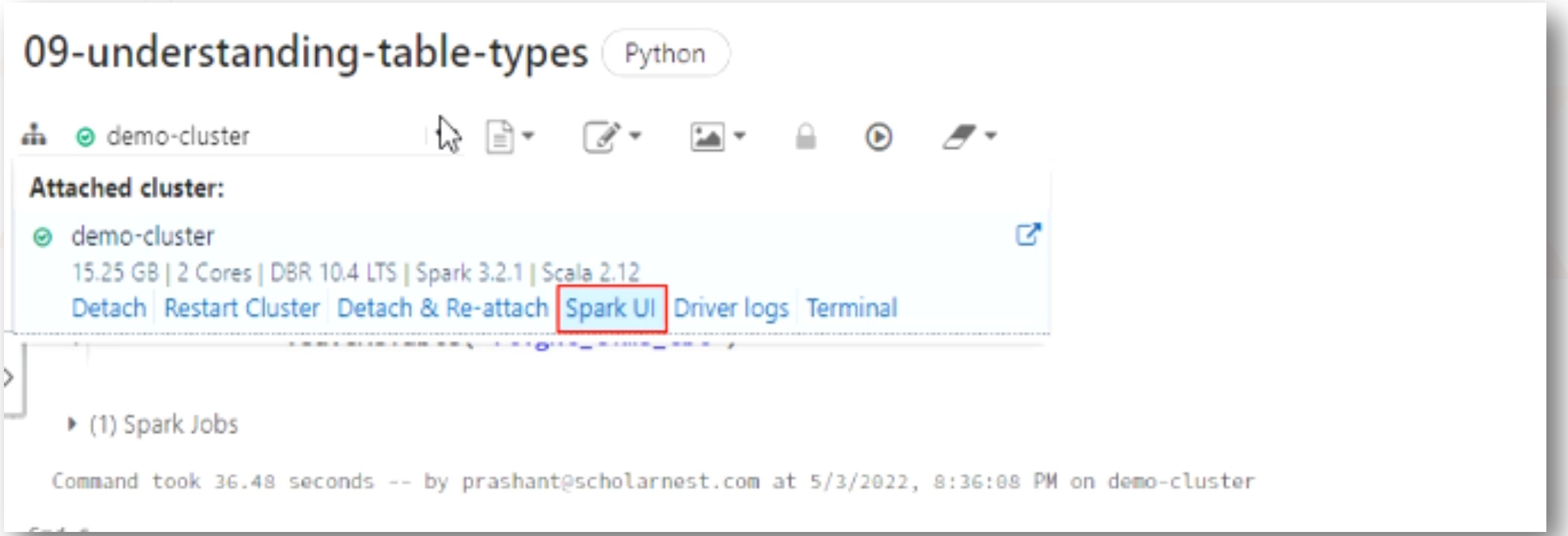
Then, navigate to your Spark Web UI. You will see some tabs such as Jobs, Stages, storage, etc. Spark UI is your go-to place for monitoring your Spark applications and understanding many other details about your application.
You do not see anything here because I haven't run anything on this cluster.
Things will start appearing here as you start running your application or the Spark code.



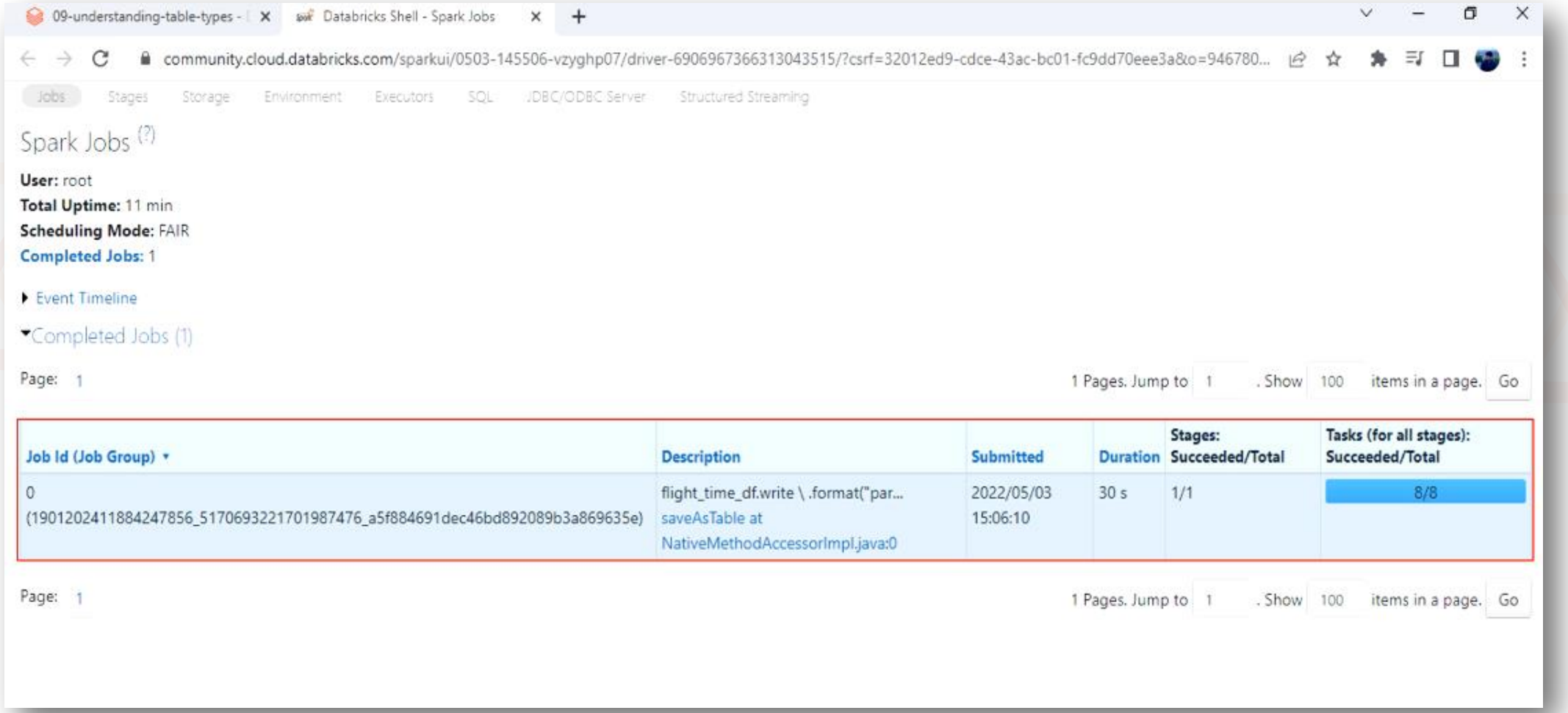
Open a notebook from the previous lecture. **(Reference - 09-understanding-table-types)** Attach a cluster to this notebook. And run all the cells up to where we are writing a Dataframe.



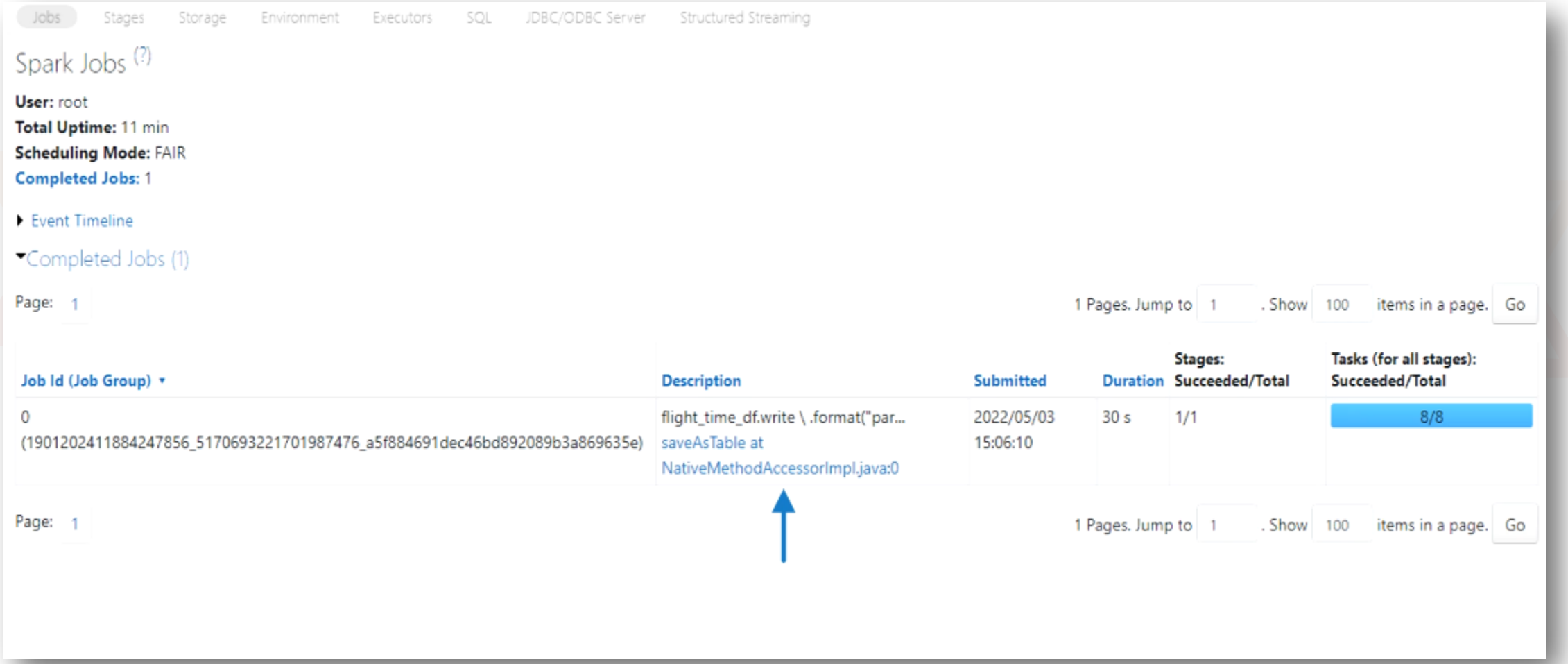
Click the demo-cluster drop down, and you will see a direct link for the Spark UI.
Let me go there.



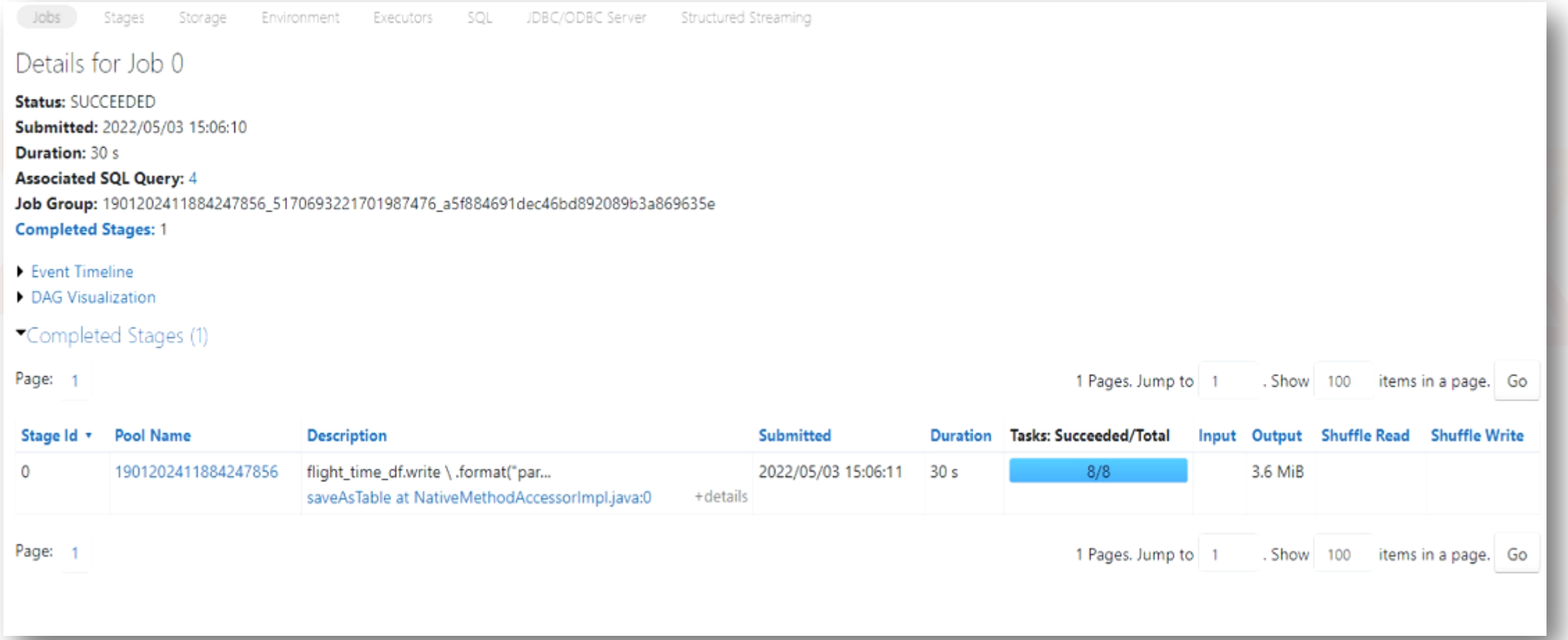
You can see some information here. We are currently in the Jobs tab of the Spark UI. The Jobs tab displays a summary page of all jobs in the Spark application. The summary page shows high-level information, such as the status, duration, and progress bar.



You can click on a job description on the summary page to see the details.



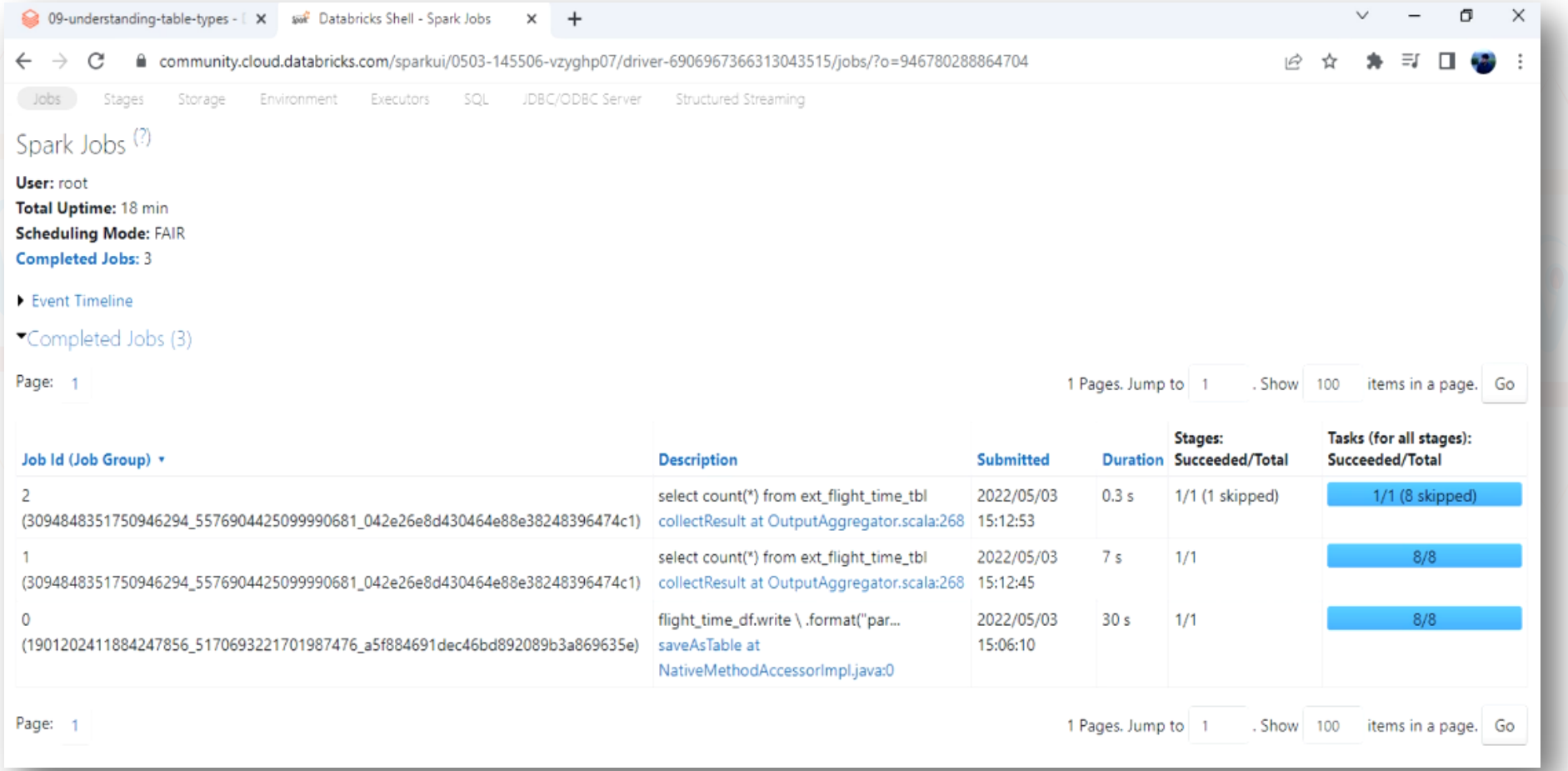
Here we are on the Job details for job id zero.
The Job details page shows you the stage-wise breakup of your Job. I have only one stage in this Job. The details page also shows some other information about the Job stages, such as duration and task progress bar.



Click the DAG visualization link to see the DAG for the selected Job.
You might not understand these things yet because I didn't cover Spark Jobs, Stages, and Tasks. However, the information shown over the Spark UI is crucial for understanding your application behavior and resources and identifying tuning and optimization opportunities.

I have only one Job as of now. It makes sense to run a few more jobs to understand how this page looked when we ran multiple jobs on the same cluster. So come back to your notebook and run a few more commands. Run the create external table command, come down to the select count(*) from ext_flight_time_tbl, and run that as well.
Now, look at the output. It says 2 Spark Jobs and also shows the count. So the select statement triggered two Spark jobs.

Go back to the Spark UI once again and refresh the page, and you will see three jobs. We can see Job IDs in the first column. Job-0 was triggered by the saveAsTable() method. The saveAsTable() is a DataFramewriter method. Job-1 and Job-2 were triggered by the select count(*) SQL.

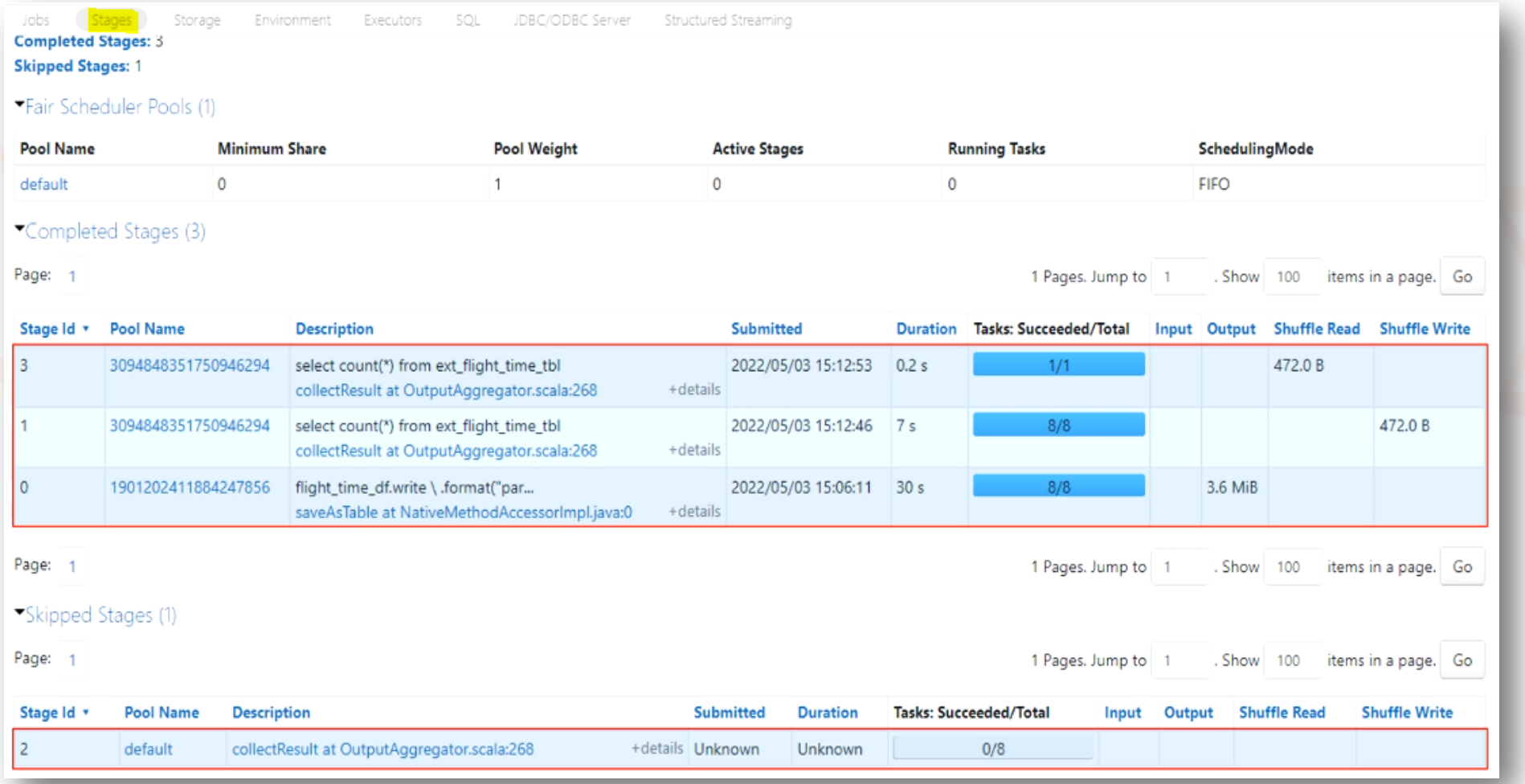


Now, go to the next tab that shows the stage summary page.

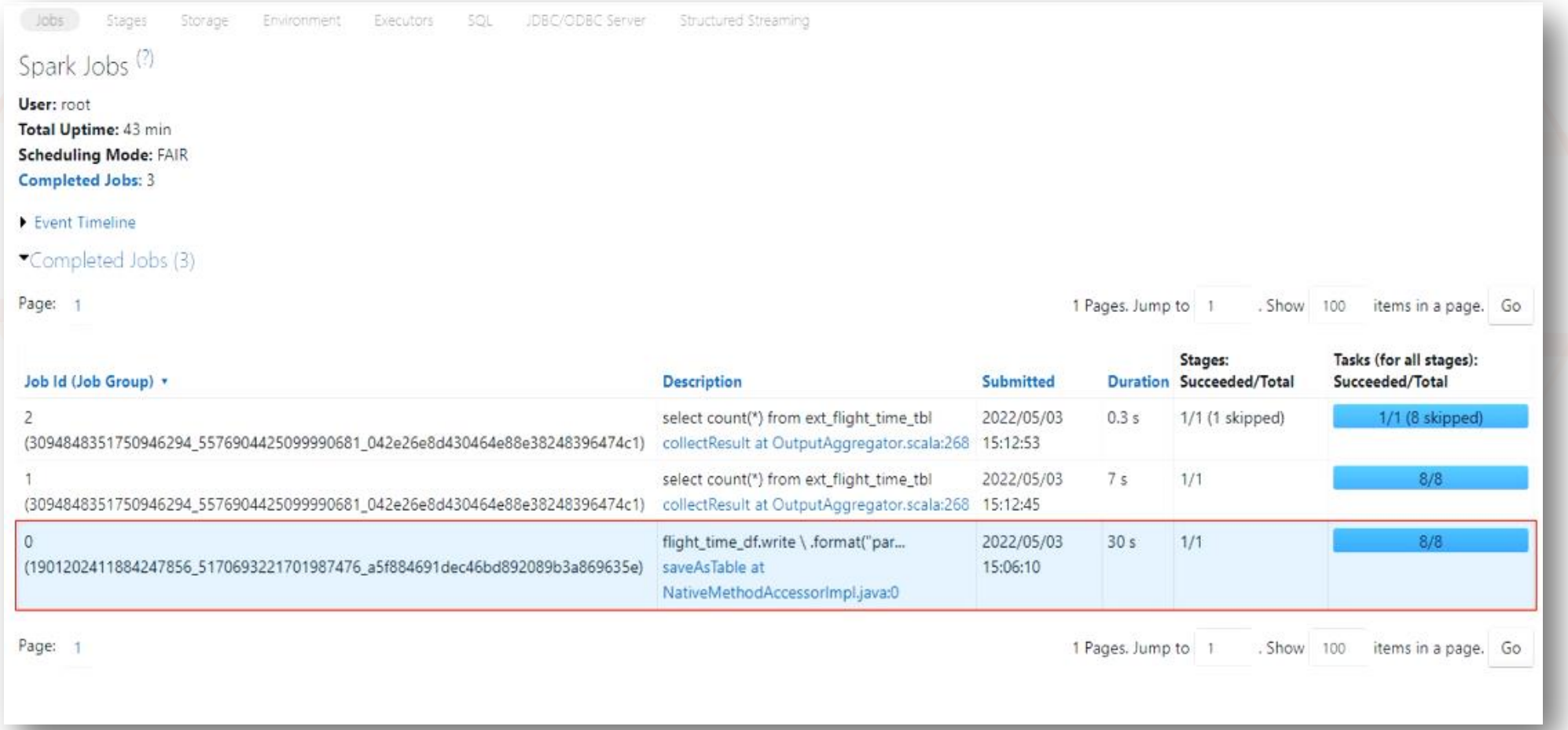
So the stage summary page shows all the stages across the Jobs.

We ran three jobs, and we saw them on the Jobs summary page.

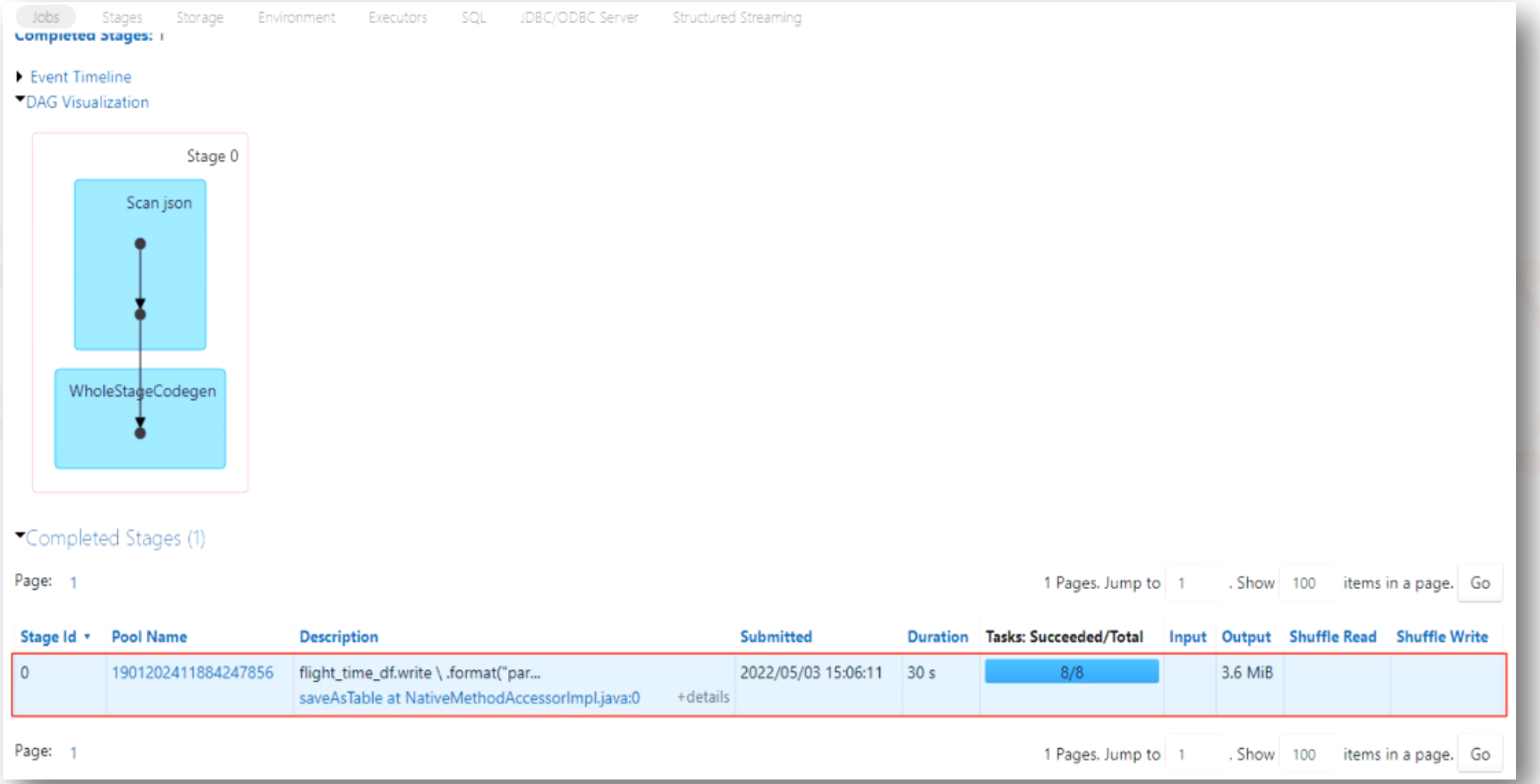
Altogether, those four jobs ran in four stages, and we can see four stages listed here on this stages summary page.



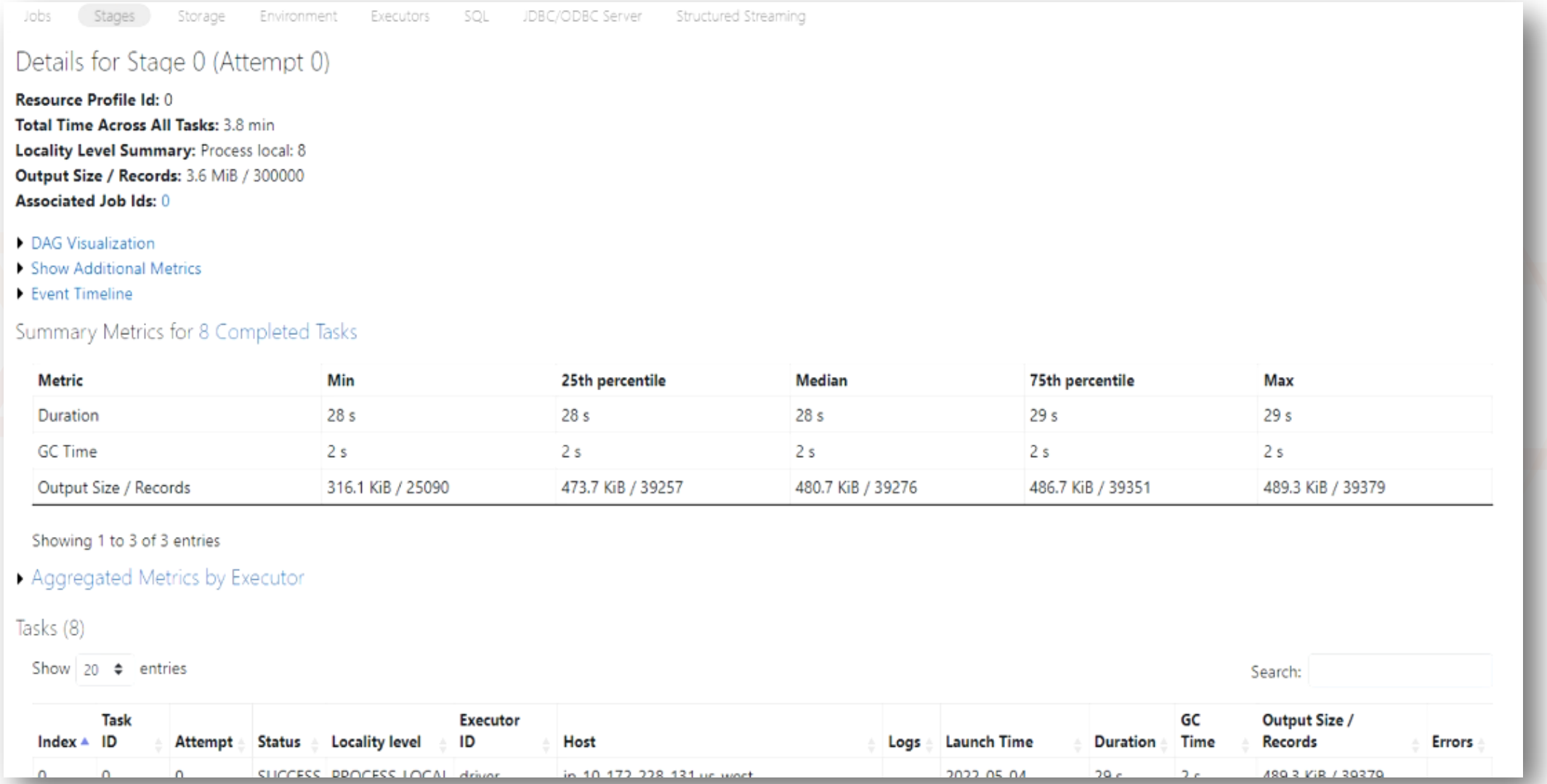
Go to the Jobs tab.
I have three jobs, and I know Job-0 is saving my Dataframe to a Spark table.
I also know that Job-1 and Job-2 give me counts from the external table.
The Job-0 took 30 seconds, so I want to investigate it further.
So I will click the Job description and reach the Job details page.



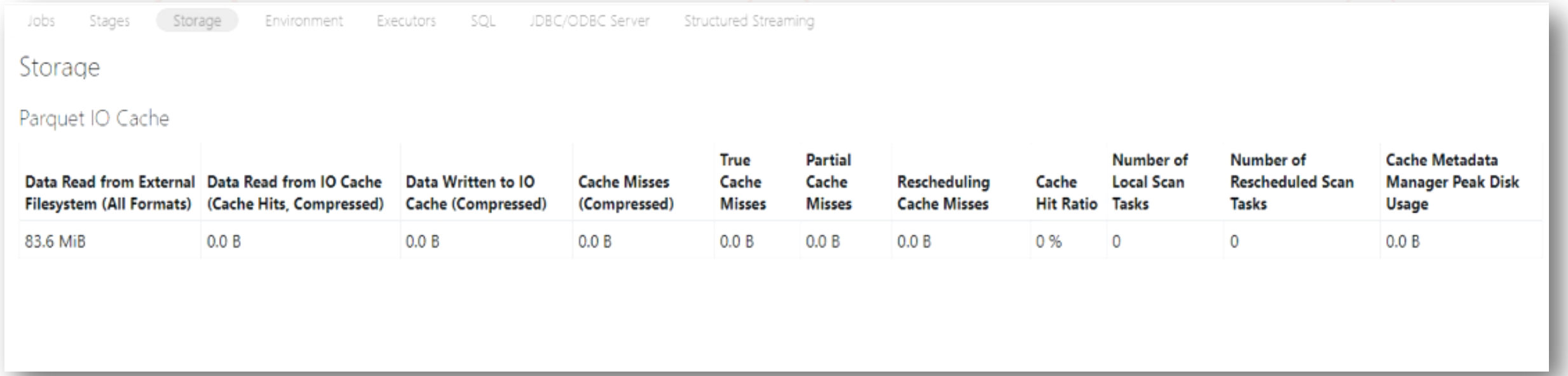
I see only one stage here, and I know this stage belongs to Job-0.
I want to look into the stage details, so I will click the Stage description and go to the stage details page.



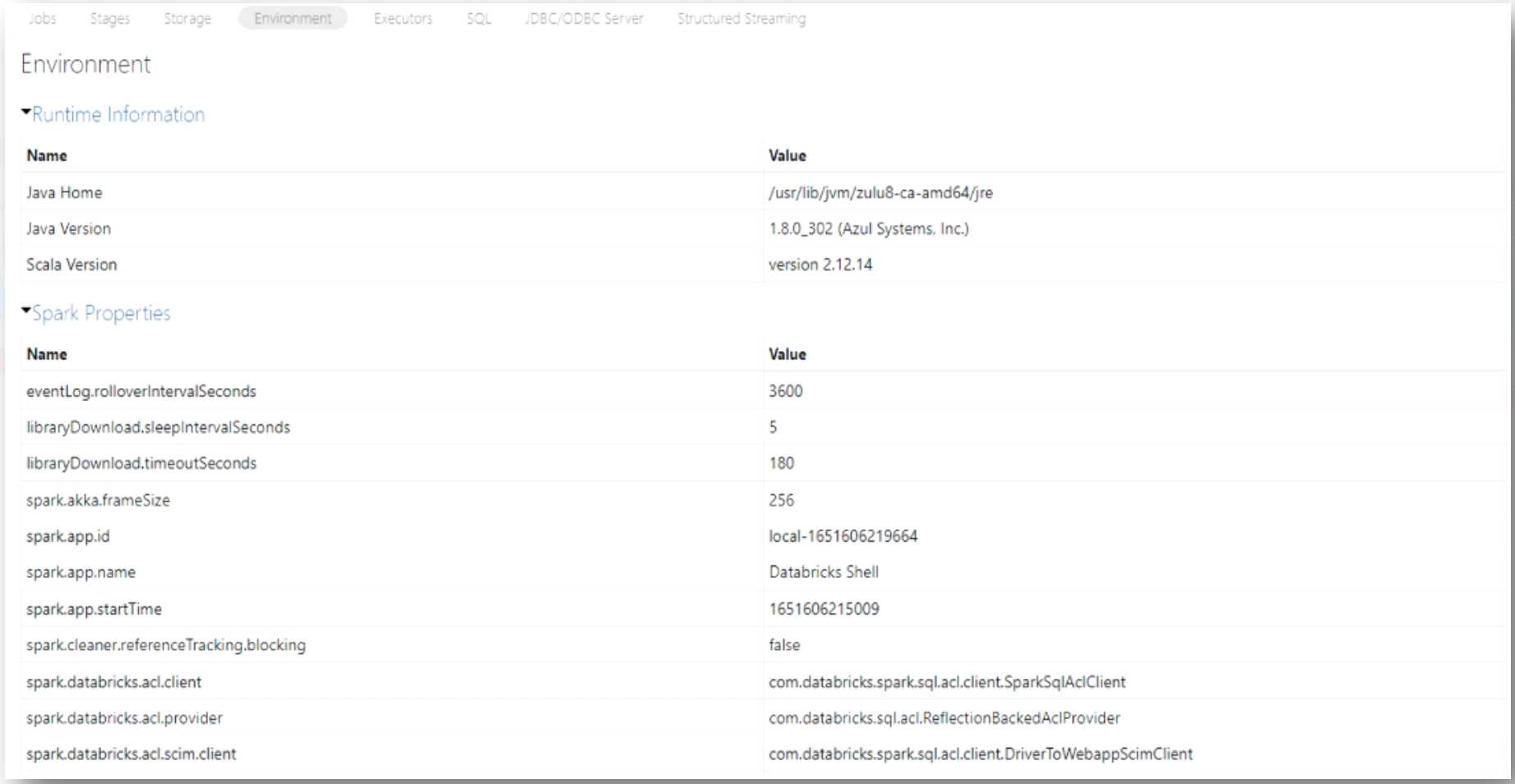
Here I am in the stage details page. I can see the stage details of my Job-0.
You can come here from the stage summary page also.



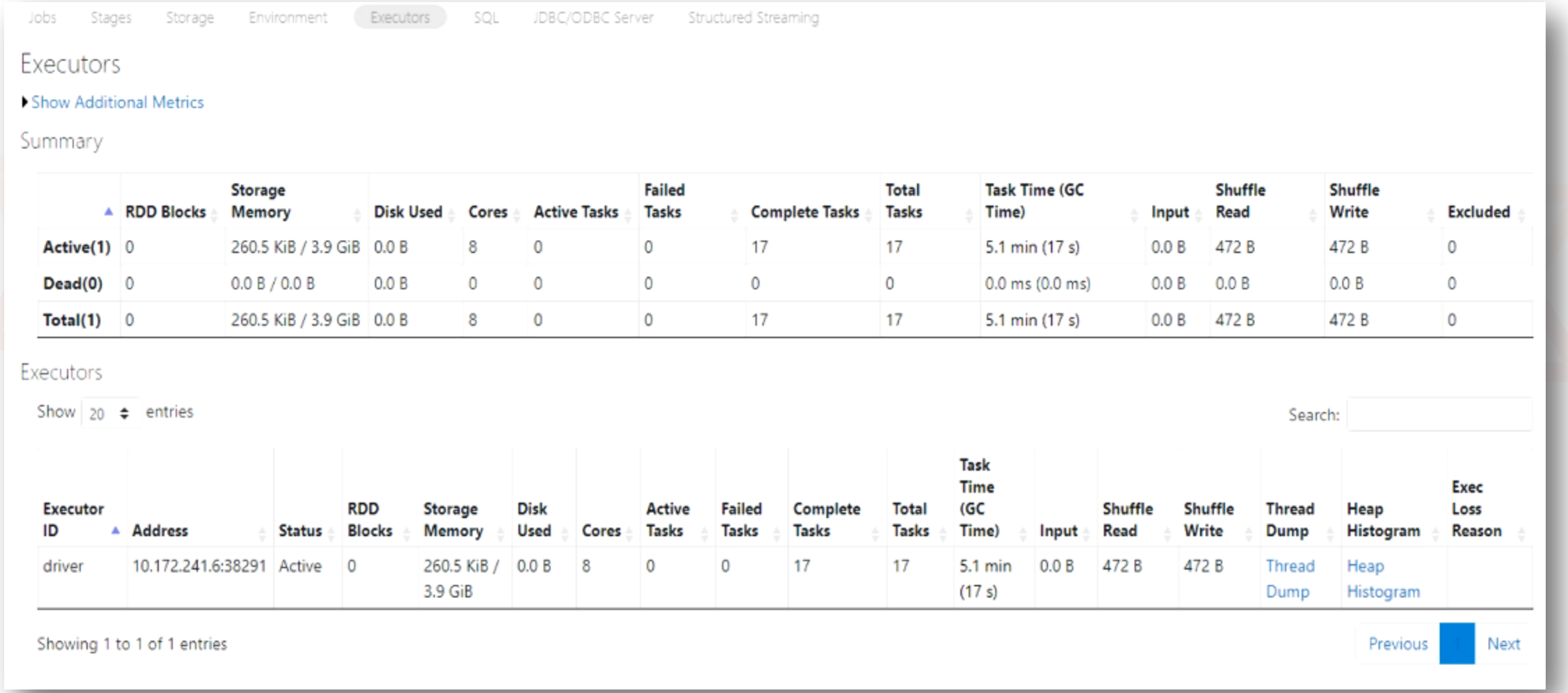
We also have a storage tab.
We do not see enough information here, but the storage page is your gateway to see the cache information.
This page shows information when you cache something in Spark memory



The next page is for the environment.
The Environment tab displays the values for the different environment and configuration variables, including JVM, Spark, and system properties.
We refer to this page to understand the runtime environment and see if some tuning is required.



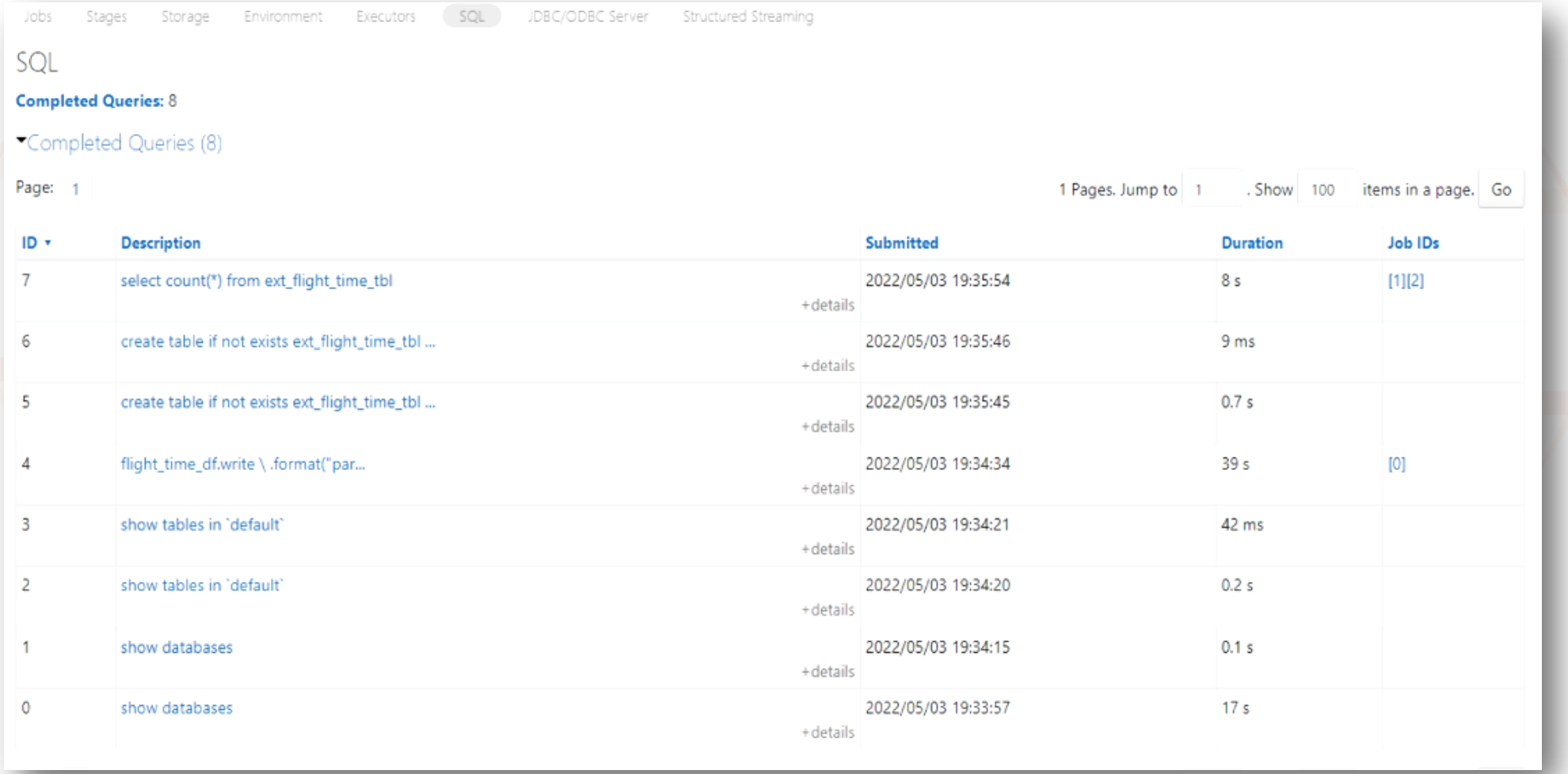
The next tab is the executor tab.
This guy displays summary information about the executors, including memory, cores, disk usage, and shuffle information each executor uses.



	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(1)	0	260.5 KiB / 3.9 GiB	0.0 B	8	0	0	17	17	5.1 min (17 s)	0.0 B	472 B	472 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1)	0	260.5 KiB / 3.9 GiB	0.0 B	8	0	0	17	17	5.1 min (17 s)	0.0 B	472 B	472 B	0

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump	Heap Histogram	Exec Loss Reason
driver	10.172.241.6:38291	Active	0	260.5 KiB / 3.9 GiB	0.0 B	8	0	0	17	17	5.1 min (17 s)	0.0 B	472 B	472 B	Thread Dump	Heap Histogram	

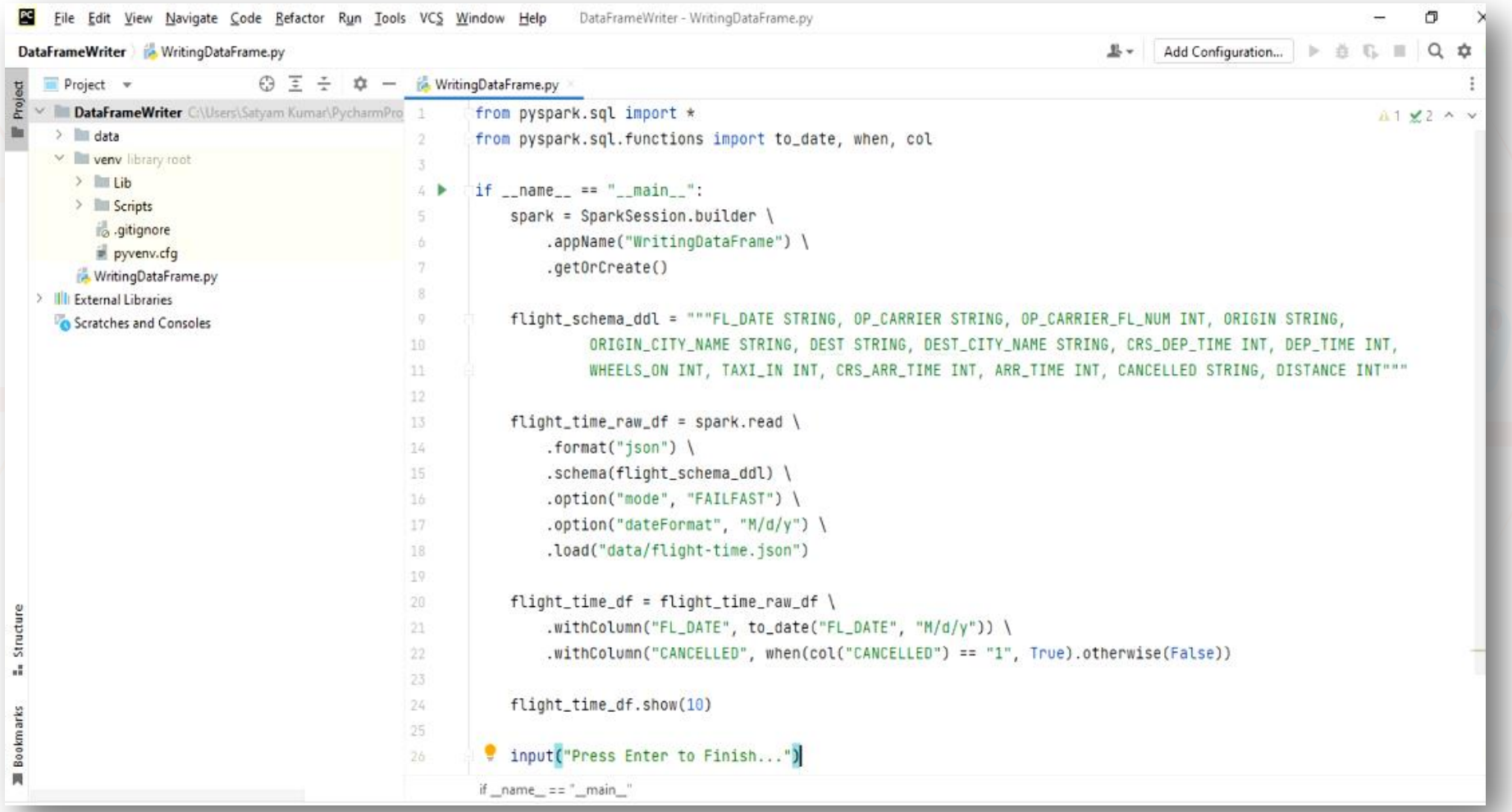
The next page is for SQL.
This page shows the list of SQL statements executed on your cluster.
You can see the time taken by each SQL. It is a summary page, and you go to the SQL details page, clicking any SQL.



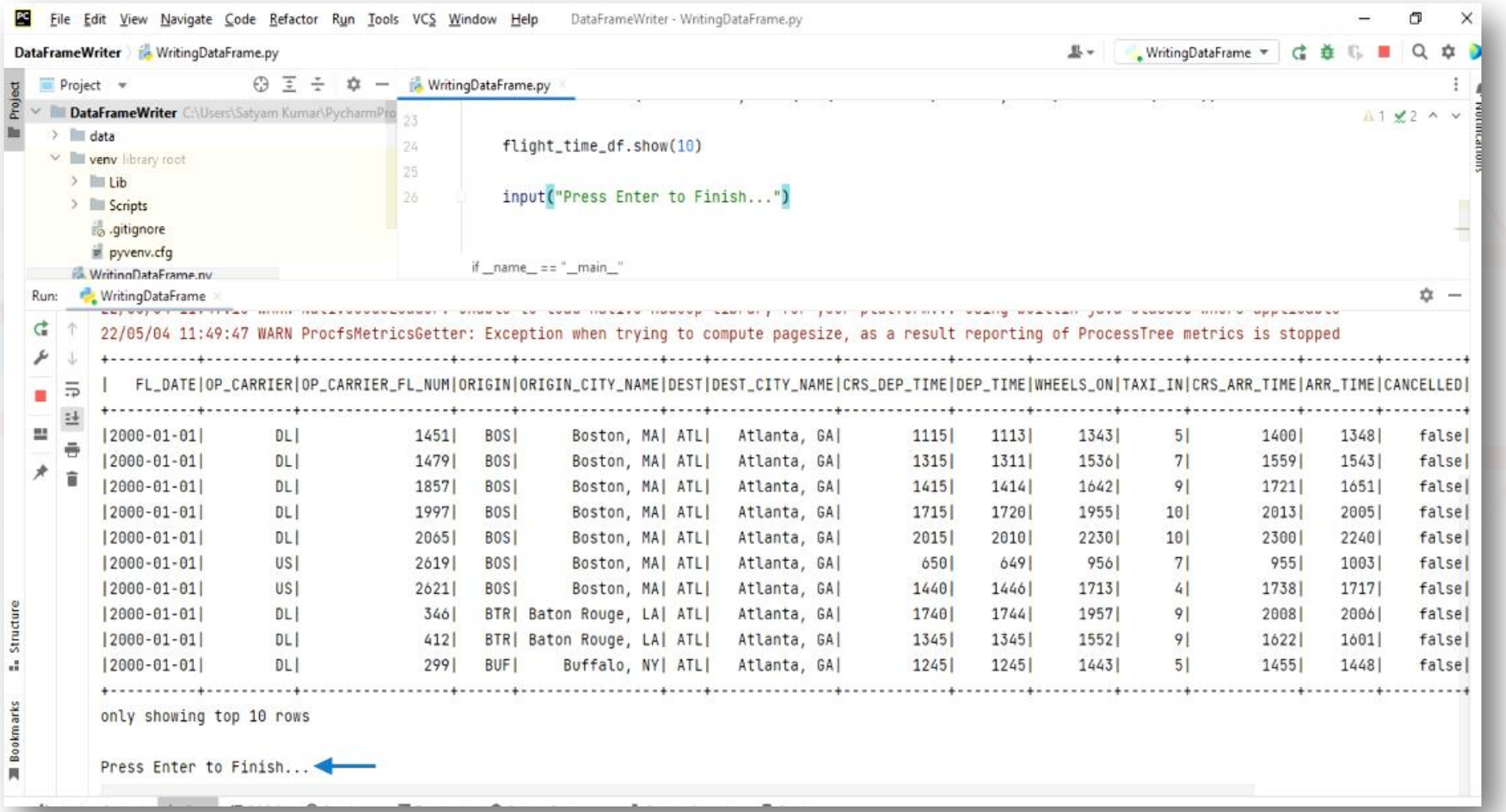
You also have JDBC/ODBC server tab.
This tab is visible only when your Spark cluster also runs a Thrift server and allows JDBC/ODBC connections.
We do not see anything here because we do not have any connections.
The last tab is for Spark Structured Streaming.
This tab shows information about streaming queries and micro-batches.
We will cover this tab in the Spark streaming course.

You create and test the Spark application on your local machine. And you can still access the Spark UI. Here is my PyCharm IDE. I created a new project and wrote some code here. **(Reference - DataFramewriter)**

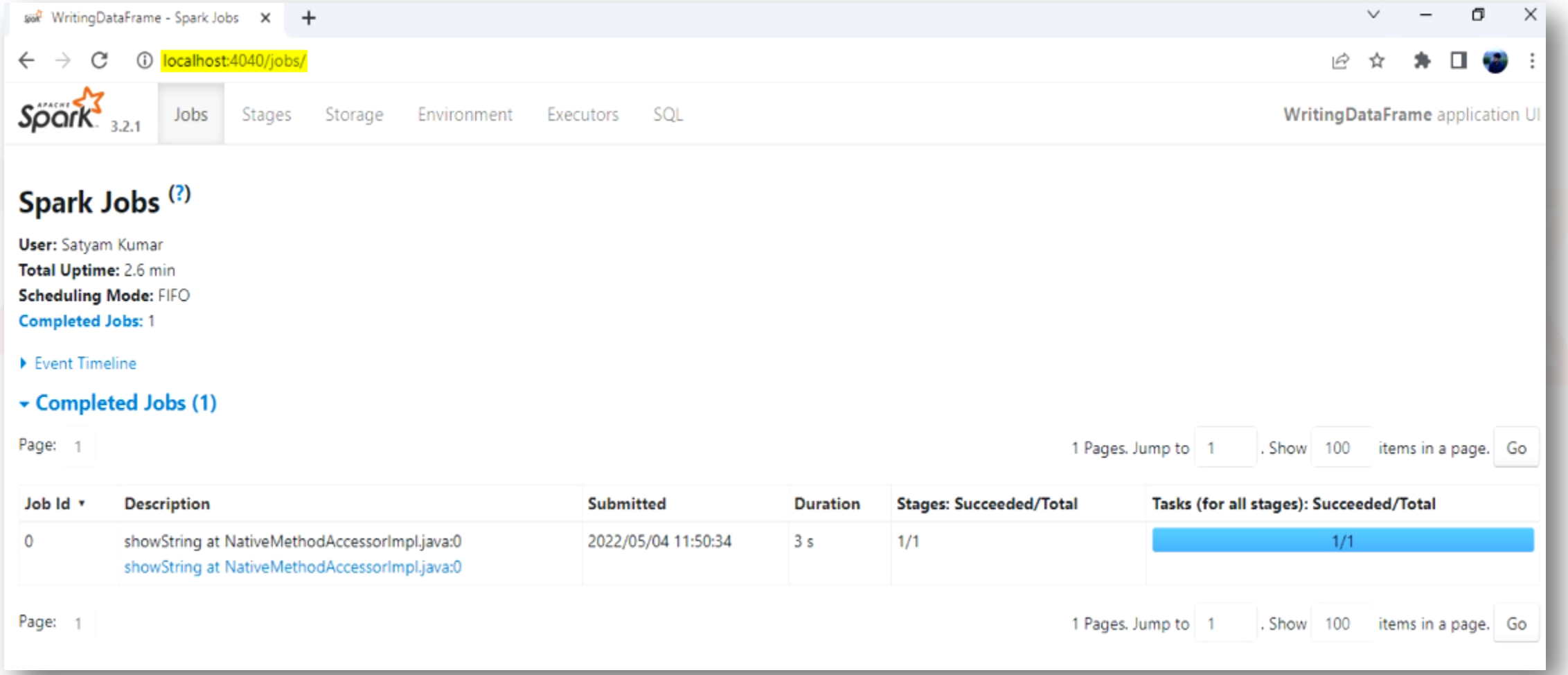
It is the same code that we saw in the notebook.



You can run this application, and it will work. I added a Python input method, so my application is not terminating, and it is waiting for me. You can see in the output, there is a message “press enter to finish” don’t press enter yet. I am holding my application from termination to see the Spark UI.



Spark UI is available to your localhost:4040 port. So let me start the browser and visit localhost:4040. You can see the same Spark UI on your local machine. This UI is only available if your Spark application is running. As soon as your application finishes, the Spark UI goes away.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com