Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Data Sources & Sinks

Lecture:
Introduction to Data Sources & Sinks

Introduction to Data Sources and Sinks

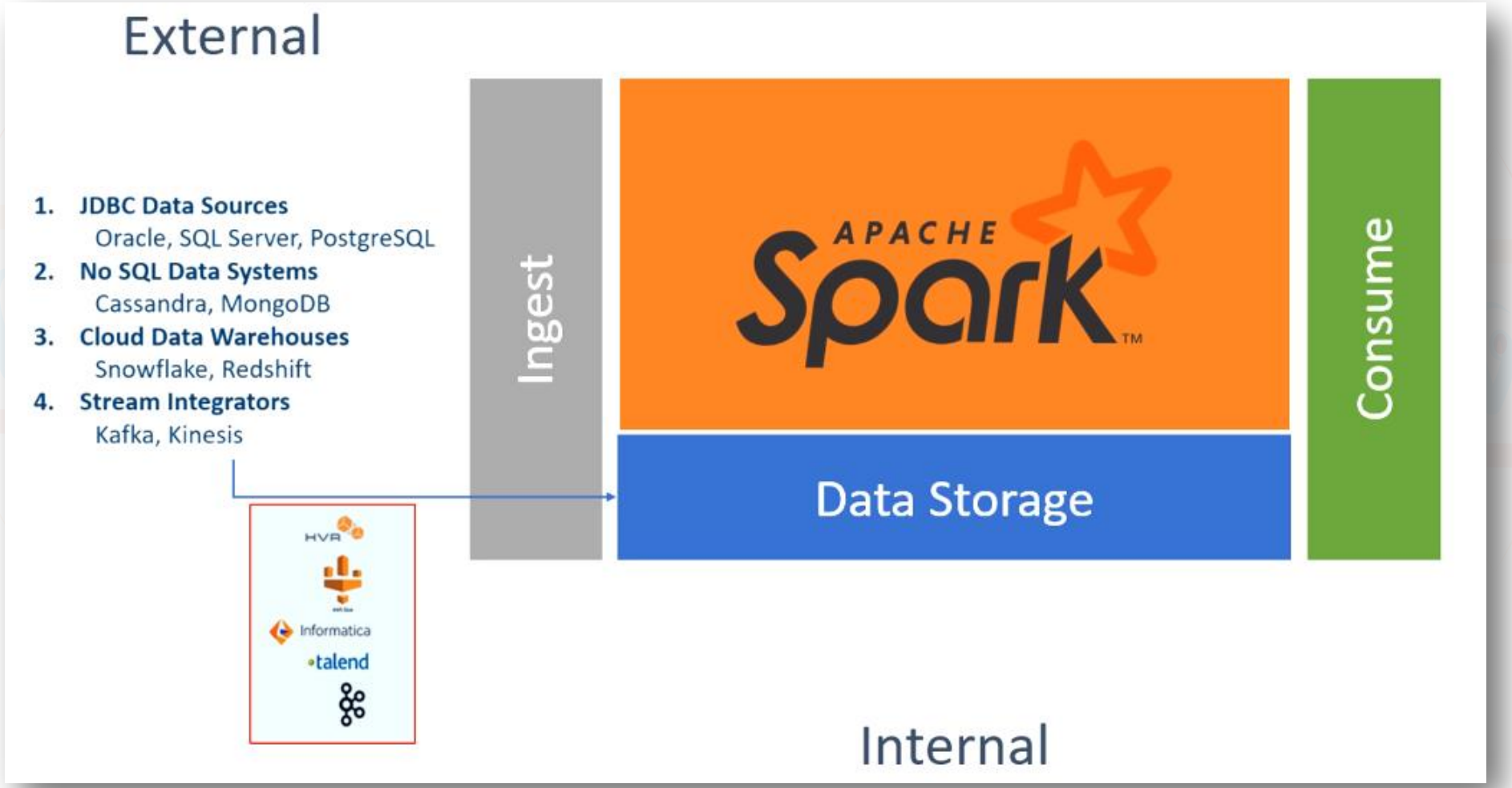
I already introduced you to the Data lake and Lakehouse concept, and you learned that Spark is used for processing large volumes of data.
However, any processing engine, including Spark, must read data from some data source.
And that's what we mean by Spark Data Sources.
These data sources can be further categorized into two groups:

1. External Data Sources
2. Internal Data Sources

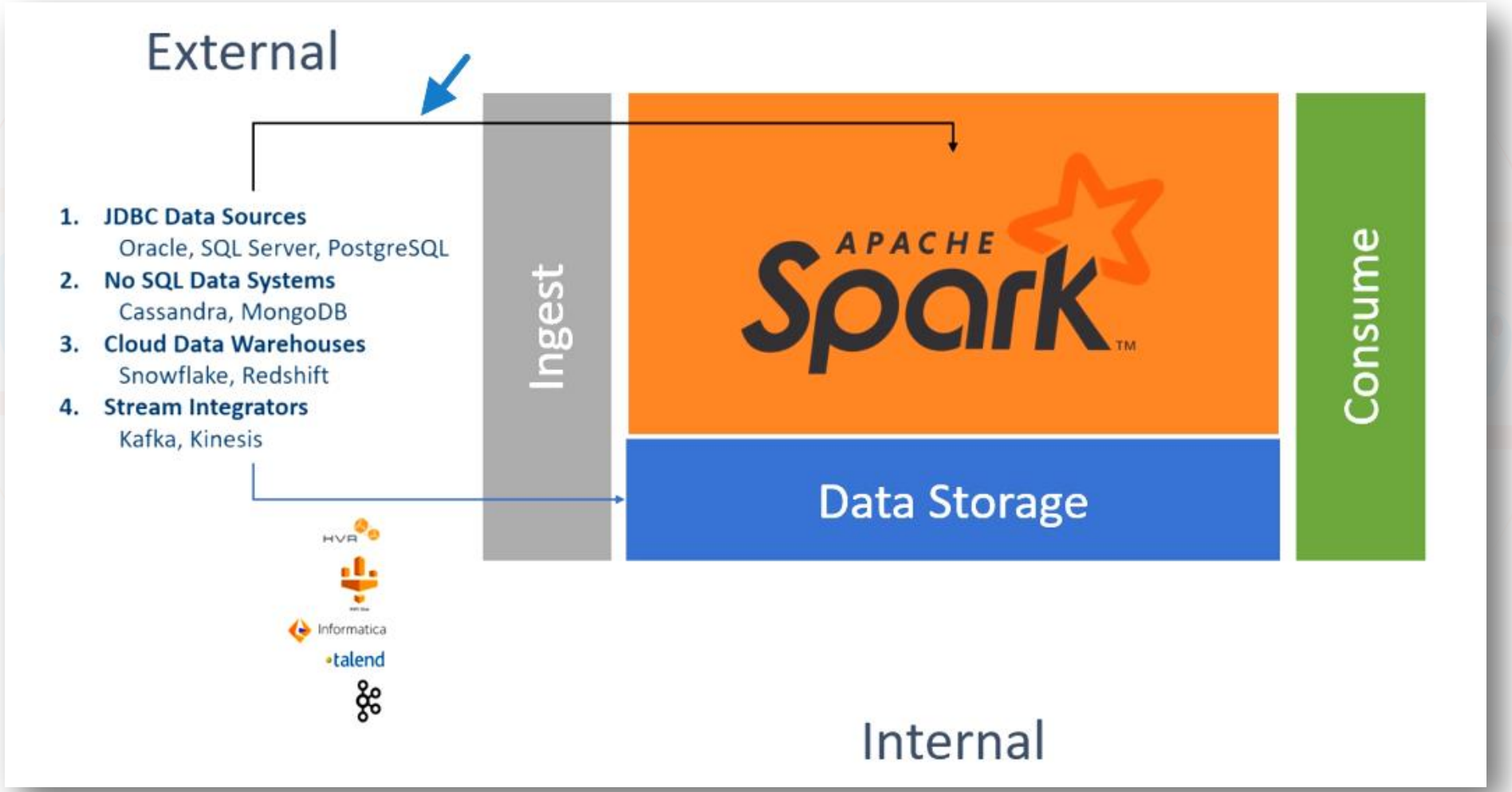
Your data might be stored in some source systems such as Oracle or SQL Server Databases. It might be stored at application servers, such as your logs. However, all these systems are external to your Data lake. You don't see them in this Data lake or Lakehouse conceptual diagram. So, we categorize such data sources as external data sources. The list of possible external data sources is too long. However, here are some notable systems listed below.

Now the question is this.
How can we read data from external sources to Spark?
You cannot process the data from these systems unless you read them and create a Spark Dataframe or a Dataset.
There are two approaches for this.

1. Bring your data to the Data lake and store them in your lake's distributed storage. How you bring it to the lake is your choice. The most commonly used approach is to use a suitable data integration tool as highlighted in the image below.

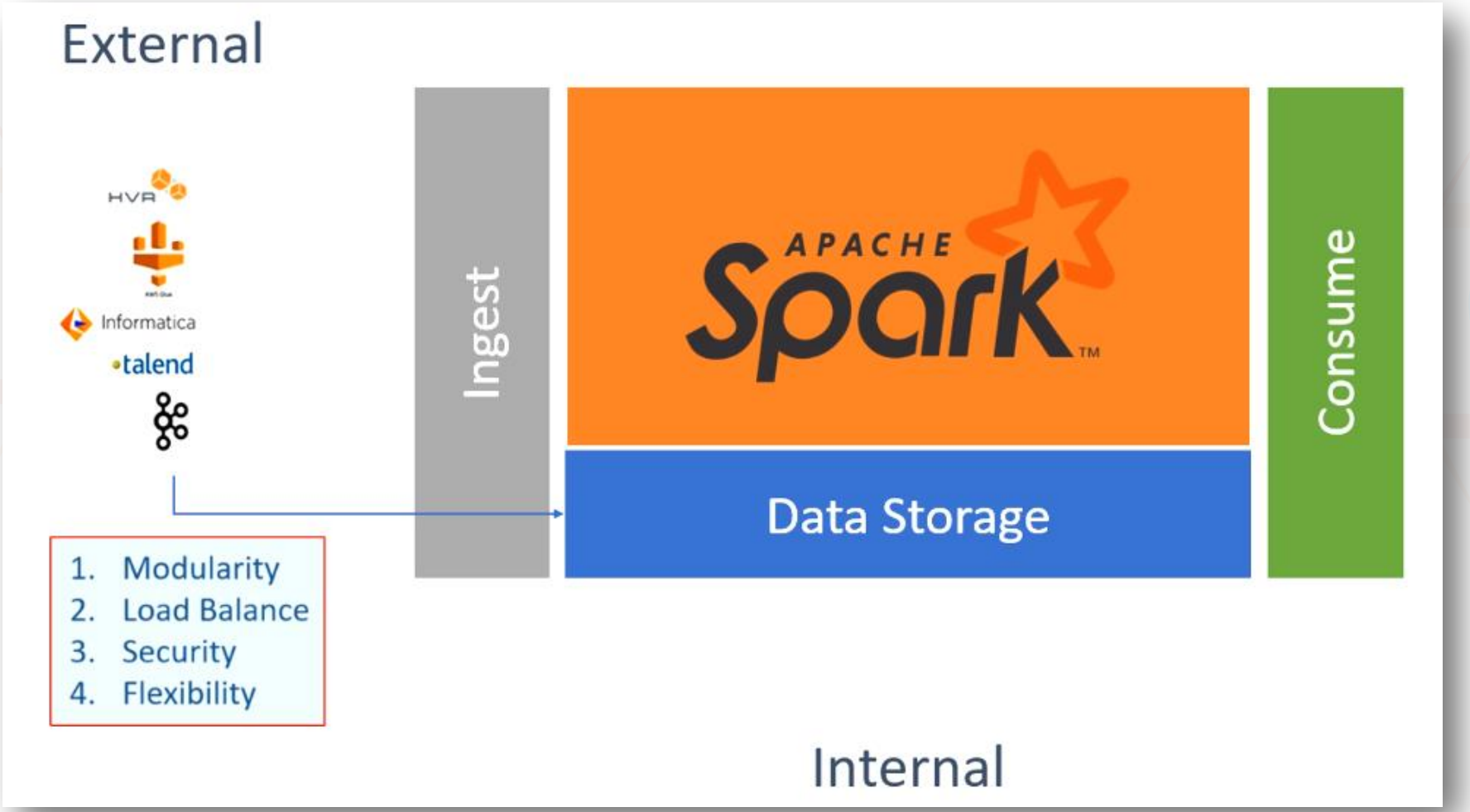


2. Use Spark Data Source API to connect with these external systems directly. And Spark allows you to do it for a variety of source systems. For example, you can connect to all of these 4 source systems which I have listed below.



So now you have two choices.
Which one do you prefer?
I prefer the first one for all my batch processing requirements and the second for all my stream processing requirements.
However, stream processing is part of the later course, so I leave that discussion for later.

Now let's assume that I have a batch processing requirement.
So I will prefer the first approach: Using a data integration tool to bring data to the distributed storage, and then we start processing it. We prefer this two step approach because of the four reasons highlighted below.



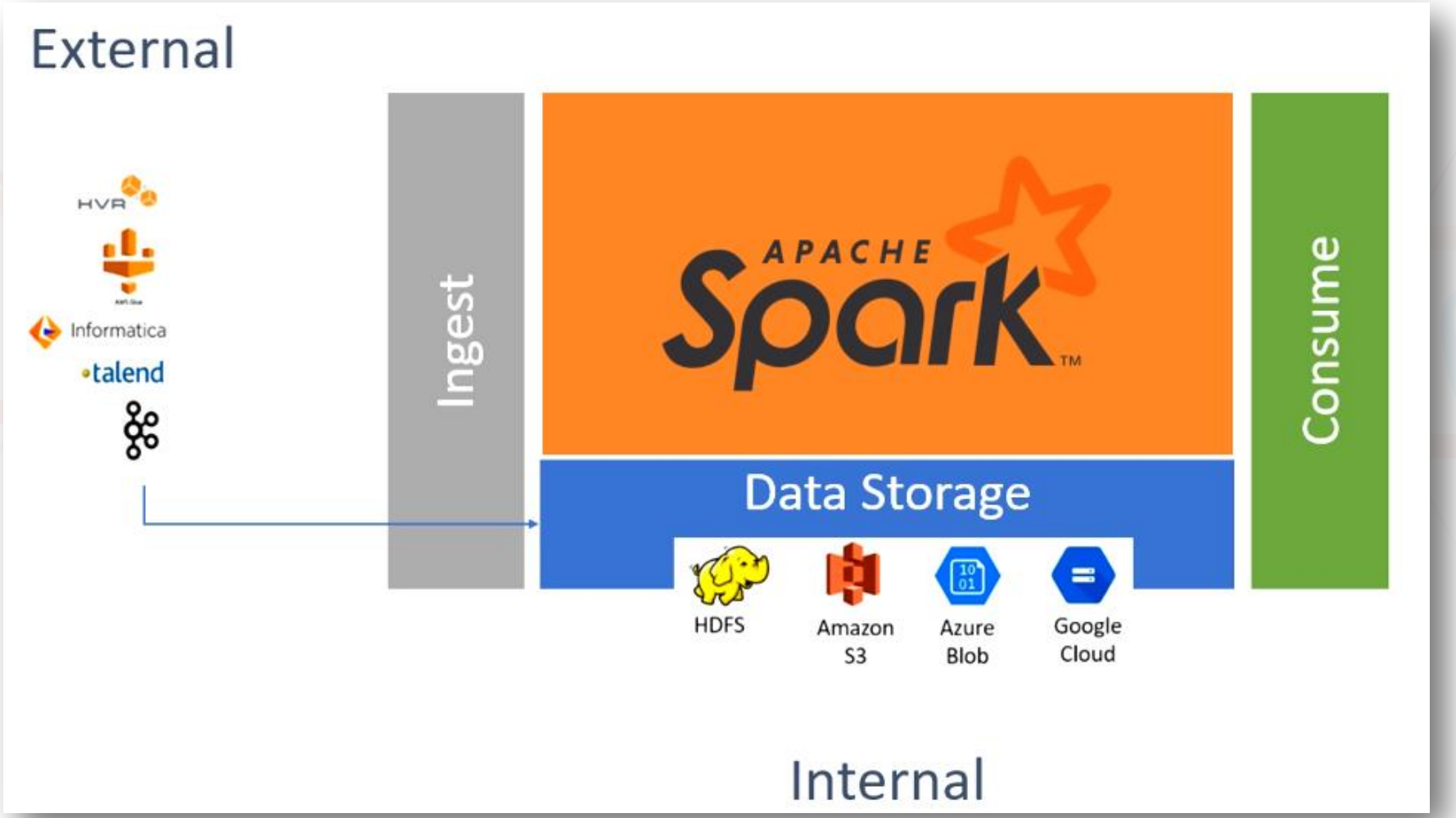
Here is the detailed explanation of the previous 4 factors influencing the two step approach:

1. Bringing data correctly and efficiently to your lake is a complex goal.
2. We want to decouple the ingestion from the processing to improve manageability.
3. Your source system would have been designed for some specific purpose. And the capacity of your source system is planned accordingly. Now, if you want to connect your Spark workload to these systems, you must replan your source system capacity and the security aspects of those systems.
4. We want to use the right tool for the right purpose. A spark is an excellent tool for data processing.

However, It wasn't designed to handle the complexities of Data Ingestion.

So, most of the well-designed real-life projects will not directly connect to the external systems even though we can do it.

Now let's come to the second category: Internal Data Sources.
So your internal data source is your distributed storage. It could be HDFS or cloud-based storage such as ADLS or Amazon S3. However, your data is stored in these systems as a data file.



The mechanics of reading data from HDFS or cloud storage are the same. However, the difference lies in the data file format. Here are some commonly used file formats listed below.

And other than these file formats, we also have two more options:

1. Hive Database Tables
2. Spark Database Tables

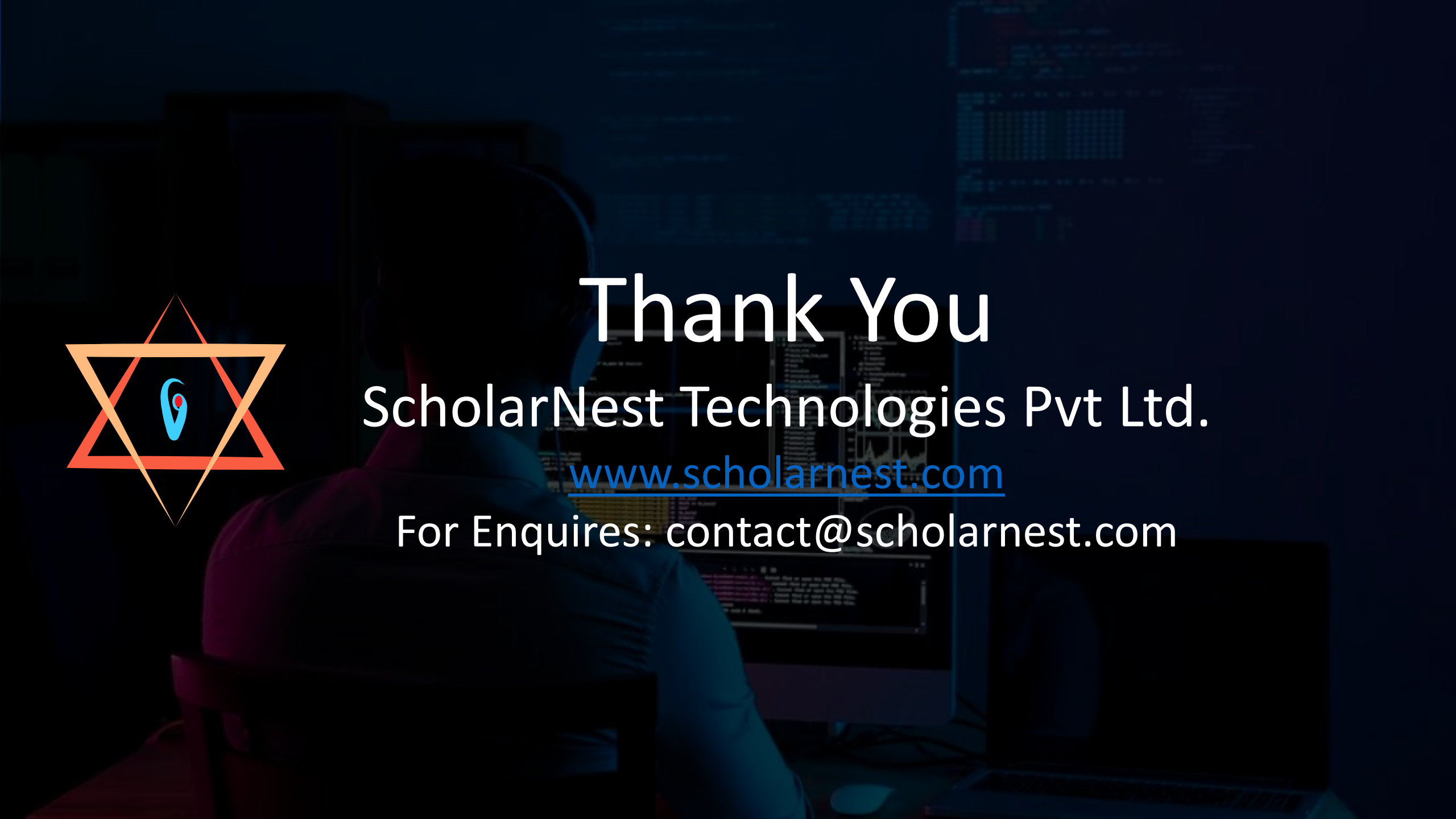
These two are also backed by data files. However, they also include some additional metadata and transaction logs stored outside the data file. So we do not categorize them as file sources.

So now you understand the Spark Data Sources part. Now let us move onto the next topic, that is Data Sink.
The data sinks are the final destination of the processed data.
So you are going to load the data from an internal or an external source.
Then you will be handling it using the Spark Dataframe APIs and Spark SQL.
Once your processing is complete, you want to save the outcome to an internal or an external system.
And these systems could be of three types:

1. A data file in your data lake or Lakehouse storage
2. A Spark Database Table
3. It could be an external system such as a JDBC or NoSQL database.

So the idea remains the same.
Working with the data source is all about reading the data, and working with the sink is about writing the data.
Like sources, Spark allows you to write the data in various file formats, Stark Database tables, and Hive Tables.
Spark also allows you to directly write the data to external sources such as JDBC databases, Cassandra, and MongoDB.

However, we again do not recommend directly writing data to the external systems for the same reasons we do not directly read from these external systems.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Data Sources & Sinks

Lecture:
Spark Dataframe Reader

Understanding Spark Dataframe Reader

Apache Spark offers a standardized API interface to work with Data Sources. These APIs have a well-defined format and a recommended pattern for usage. In this chapter, I am going to help you understand the basics of using Spark Data Sources.

Spark allows you to read data using the DataFrameReader API. You can access the DataFrameReader through the SparkSession via the read method.
Here is the general structure of the DataFrameReader API shown in the left side of the image below. And there is an example shown at the right side.

General structure

```
DataFrameReader
    .format(...)
    .option("key", "value")
    .schema(...)
    .load()
```

Indicative Example

```
spark.read
    .format("csv")
    .option)"header", "true")
    .option("path", "/data/mycsvfiles/")
    .option("mode", "FAILFAST")
    .schema(mySchema)
    .load()
```


You can get the DataFrameReader using the spark.read attribute where the spark is a SparkSession variable.

General structure

```
DataFrameReader
    .format(...)
    .option("key", "value")
    .schema(...)
    .load()
```

Indicative Example

```
spark.read
    .format("csv")
    .option)"header", "true")
    .option("path", "/data/mycsvfiles/")
    .option("mode", "FAILFAST")
    .schema(mySchema)
    .load()
```


Once you have the DataFrameReader, you can specify four main things.

General structure

```
DataFrameReader
    .format(...)
    .option("key", "value")
    .schema(...)
    .load()
```

Indicative Example

```
spark.read
    .format("csv")
    .option)"header", "true")
    .option("path", "/data/mycsvfiles/")
    .option("mode", "FAILFAST")
    .schema(mySchema)
    .load()
```


The first thing is the format of your data source. I am using CSV format because my data files are CSV files. Spark DataFrameReader reader supports several built-in formats such as CSV, JSON, Parquet, ORC, and JDBC. The DataFrameReader API is extended by the community developers and third-party vendors to support hundreds of external data sources.
And they offer a separate format for different sources such as Cassandra, MongoDB, XML, HBase, Redshift, and many more.

General structure

```
DataFrameReader
    .format(...)
    .option("key", "value")
    .schema(...)
    .load()
```

Indicative Example

```
spark.read
    .format("csv")
    .option)"header", "true")
    .option("path", "/data/mycsvfiles/")
    .option("mode", "FAILFAST")
    .schema(mySchema)
    .load()
```


The next thing is to specify the options.
Every data source has a specific set of options to determine how the DataFrameReader is going to read the data.
The header option in this example is specific to CSV format.
For other formats, you must look into the documentation of the data source.

General structure

```
DataFrameReader
    .format(...)
    .option("key", "value")
    .schema(...)
    .load()
```

Indicative Example

```
spark.read
    .format("csv")
    .option)"header", "true")
    .option("path", "/data/mycsvfiles/")
    .option("mode", "FAILFAST")
    .schema(mySchema)
    .load()
```


The third most important thing is the Read Mode.
You can specify the Mode via the option method itself.
But what is the Mode? Reading data from a source file, especially the file data sources such as CSV, JSON, and XML may encounter a corrupt or malformed record.
Read modes specify what will happen when Spark comes across a malformed record.

General structure

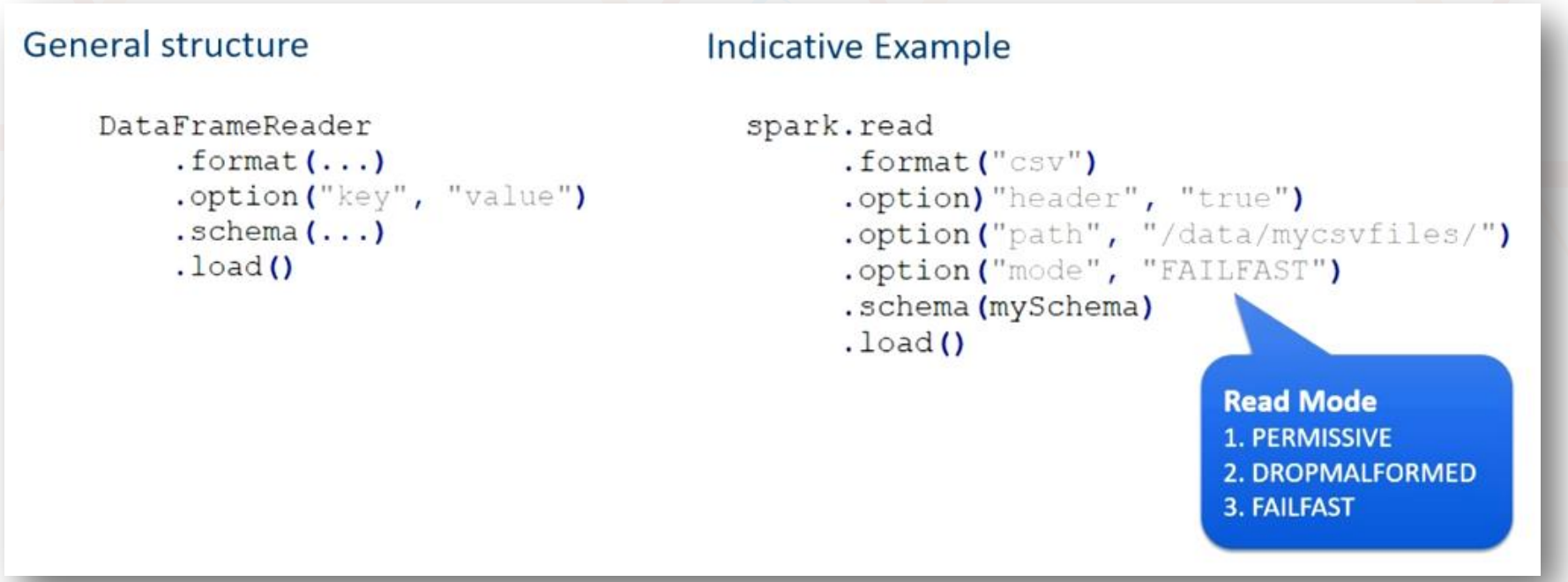
```
DataFrameReader
    .format(...)
    .option("key", "value")
    .schema(...)
    .load()
```

Indicative Example

```
spark.read
     .format("csv")
     .option)"header", "true")
     .option("path", "/data/mycsvfiles/")
     .option("mode", "FAILFAST")
     .schema(mySchema)
     .load()
```


Spark allows three read modes:

1. PERMISSIVE – It is the default option that sets all the fields to null when it encounters a corrupted record and places the corrupted records in a string column called _corrupt_record.
2. DROPMALFORMED – It is going to remove the malformed record. That means you are ignoring the malformed records and only loading the well-formed records.
3. FAIL FAST – It raises an exception and terminates immediately upon encountering a malformed record.



The last thing is the schema. The schema is optional for two reasons.

1. DataFrameReader allows you to infer the schema in many cases. So, if you are inferring the schema, then you do not provide an explicit schema. That's the first reason why schema is optional.
2. Some data sources, such as Parquet and AVRO, come with a well-defined schema inside the data source itself. So, in those cases, you do not need to specify a schema.

General structure

```
DataFrameReader
    .format(...)
    .option("key", "value")
    .schema(...)
    .load()
```

Indicative Example

```
spark.read
    .format("csv")
    .option)"header", "true")
    .option("path", "/data/mycsvfiles/")
    .option("mode", "FAILFAST")
    .schema(mySchema)
    .load()
```

Schema
1. Explicit
2. Infer Schema
3. Implicit

Once you are done setting the format, all necessary options, Mode, and schema, you can call the load() method to read the data and create a Dataframe.
This is a standard structure.

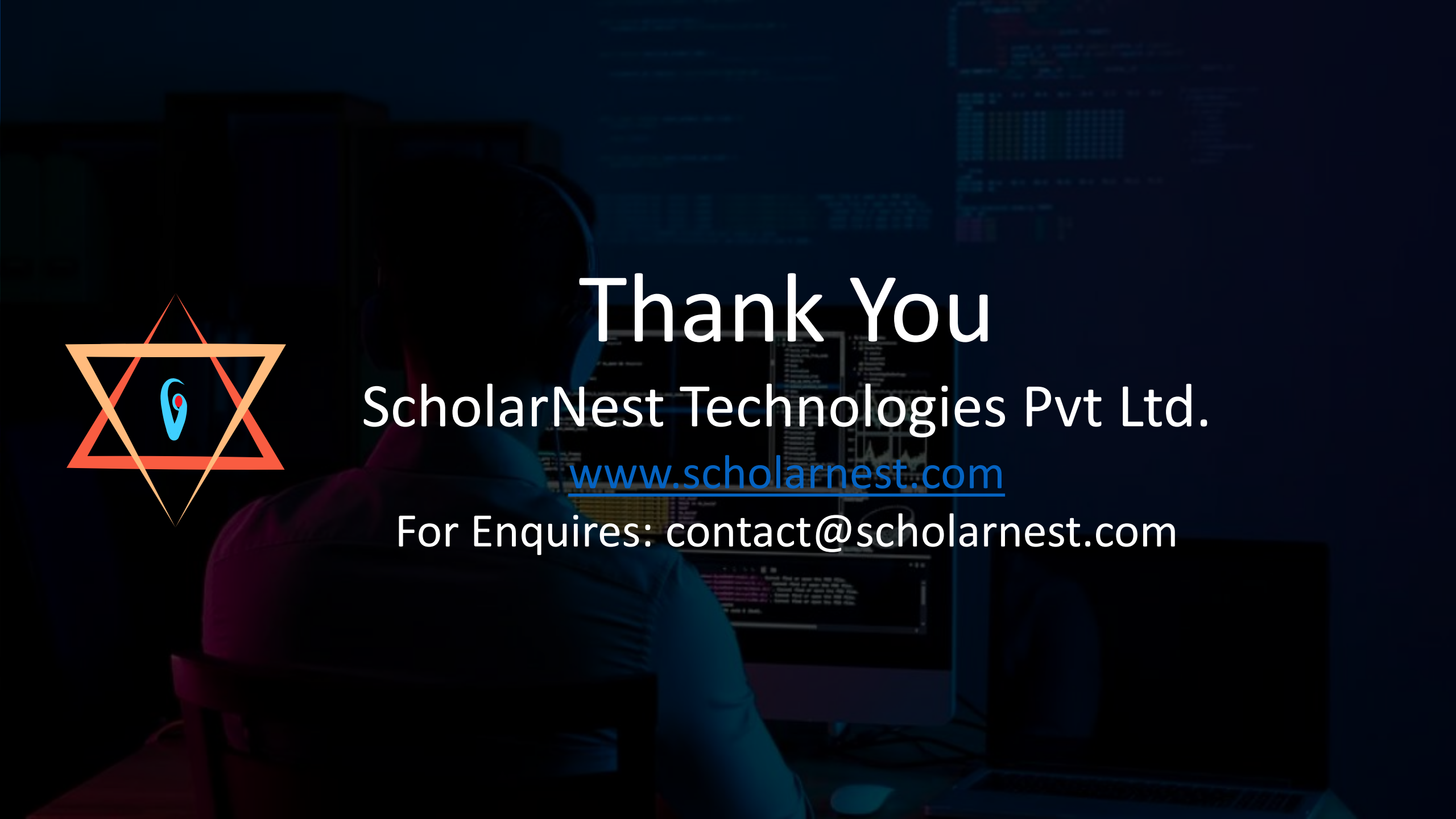
General structure

```
DataFrameReader
    .format(...)
    .option("key", "value")
    .schema(...)
    .load()
```

Indicative Example

```
spark.read
     .format("csv")
     .option)"header", "true")
     .option("path", "/data/mycsvfiles/")
     .option("mode", "FAILFAST")
     .schema(mySchema)
     .load()
```


However, like any other tool, DataFrameReader also comes with some shortcuts and variations. We have already seen one such shortcut in the earlier examples.
Do you remember?
Instead of using the load method, we used the CSV() method. Right?
That was a shortcut.
However, I recommend avoiding shortcuts and following the standard style.
Following the standard is going to add to your code maintainability.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

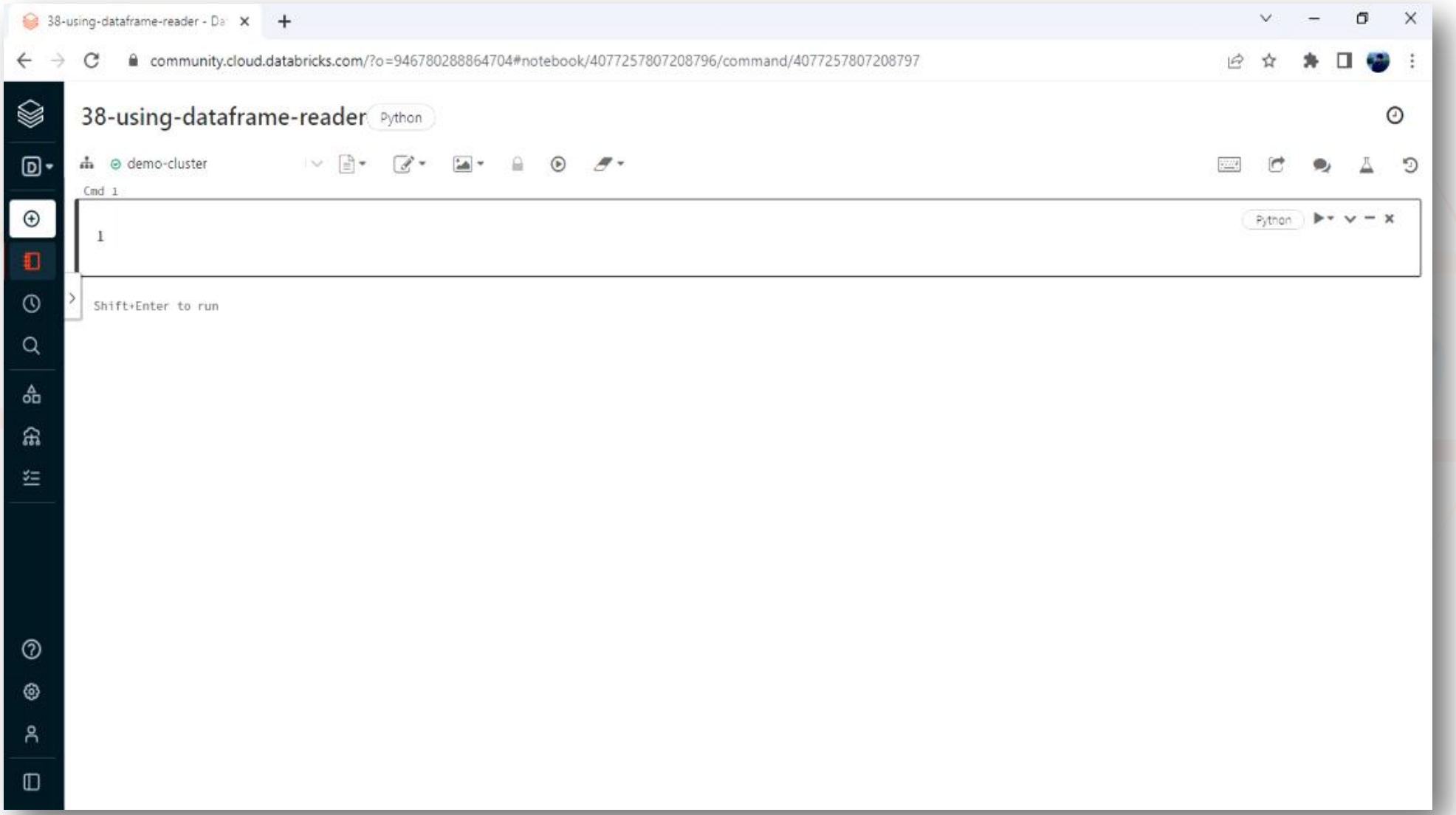
Databricks Spark Certification and Beyond

Module:
Data Sources & Sinks

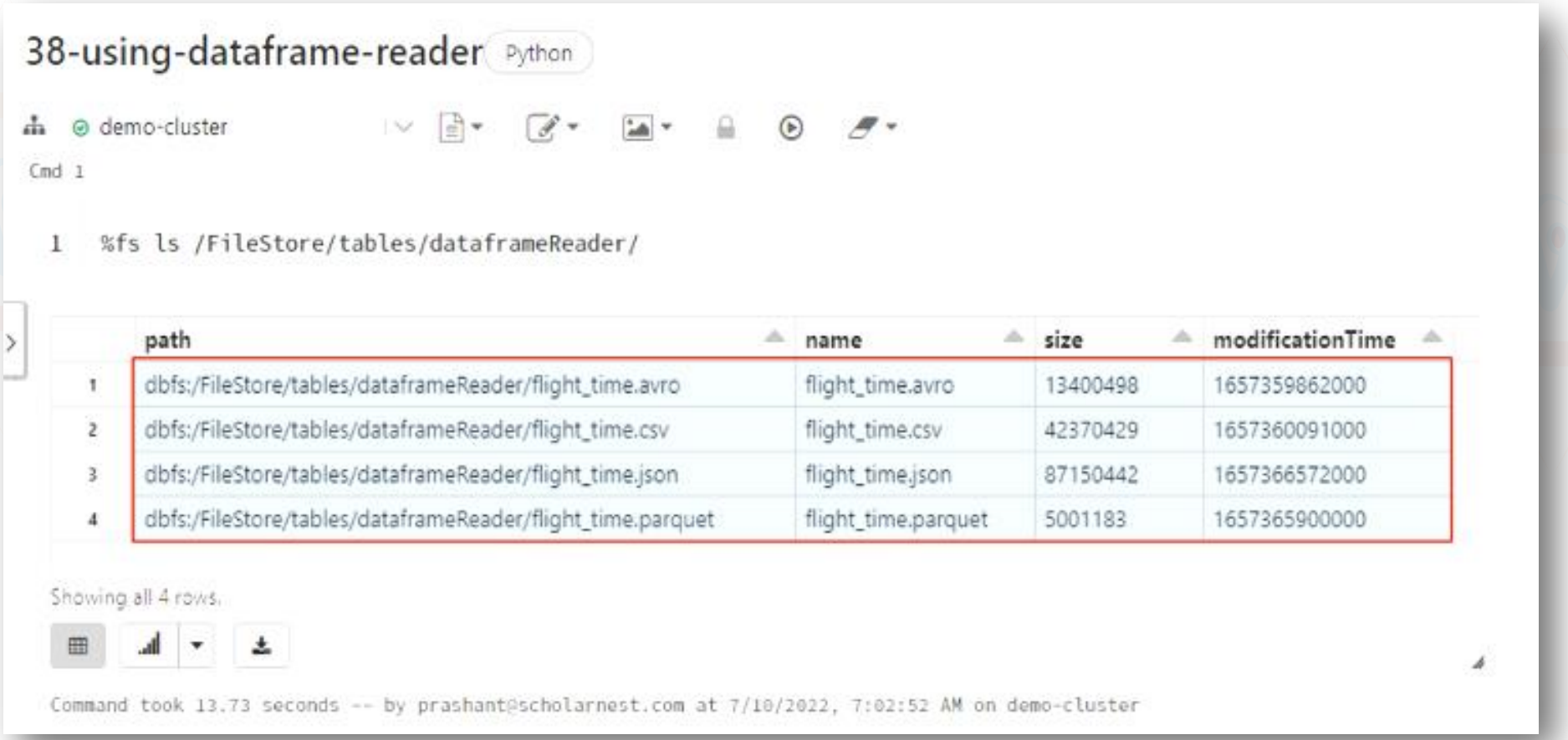
Lecture:
Reading Different Files

Reading CSV, JSON, Parquet, and AVRO File

Go to your Databricks workspace and create a new notebook.
(Reference: 38-using-dataframe-reader.ipynb)



We will use this notebook to read four different data files using the Dataframe reader API. These four files are highlighted below. The first one is a CSV file with more than 400K records of flight timing data. The second one is also the same data, but this one is a JSON file. Similarly, I have other versions of the same data, but those are parquet and Avro files.
(Reference: /data/dataframeReader/)



In this lecture, I want to show you the following:

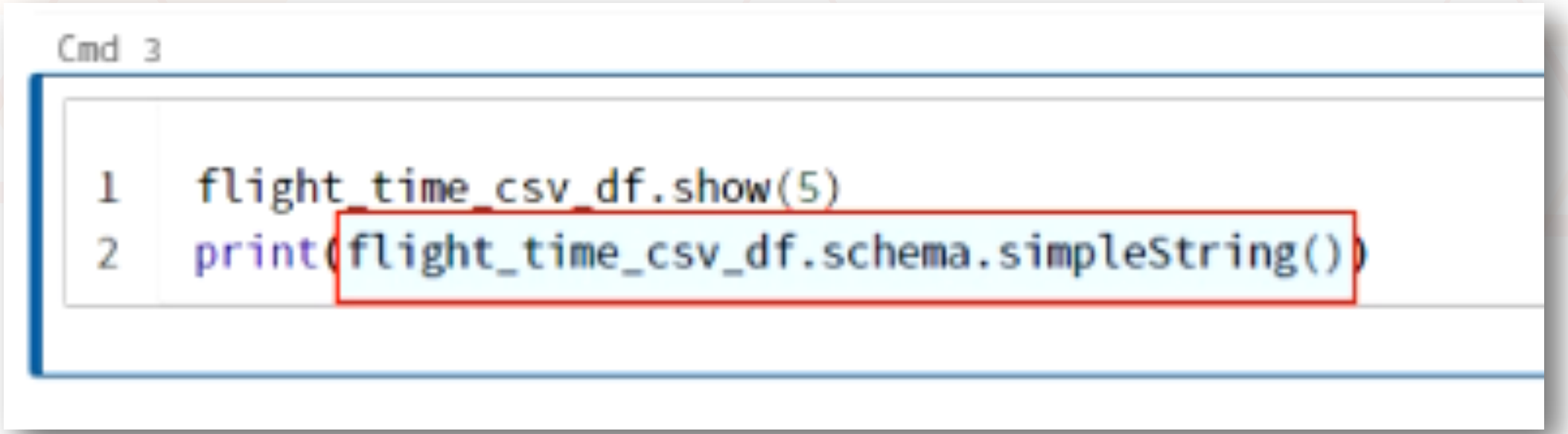
How to use DataFrameReader for CSV, JSON, Parquet, and Avro data sources?

And in the next lecture, I will refresh your memories about the following:

1. Spark Data Types
2. How to explicitly define a schema for your data and use the schema-on-read with the data frames?

Let me use DataFrameReader to read the CSV file as shown in the code below.
We start with the spark session and use the read method. The next thing is to specify the data source format. In my case, this is a CSV source. The next step is to give some necessary options. My CSV file comes with a header, so I am setting the header to true. Then I specify the file path. The path could be a file name, or it could be a directory location. Spark DataFrameReader is going to read all the files in the given directory. You can even use some wild cards (*) to target specific files, and that's what I am doing in this example.

So now we are ready to load the CSV file to a Dataframe. I am not giving any schema, and I am not even asking DataFrameReader to infer the schema.
So, let us see five records from the Dataframe, I also want to know the default Dataframe Schema. So I am trying to log the Dataframe schema converted to a simple string.



So, it looks like I am reading the data correctly. The field names are showing up correctly. But where do we get these field names? The Dataframe reader picks the column names from the header row. I didn't give the infer-schema option, but I gave the header option. The Dataframe reader is smart enough to take the column names from the header row.

```
1  flight_time_csv_df.show(5)
2  print(flight_time_csv_df.schema.simpleString())

▸ (1) Spark Jobs
+--------+----------+----------------+------+---------------+----+-------------+-----------+--------+---------+-------+-----------+--------+---------+--
------+
| FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|ORIGIN_CITY_NAME|DEST|DEST_CITY_NAME|CRS_DEP_TIME|DEP_TIME|WHEELS_ON|TAXI_IN|CRS_ARR_TIME|ARR_TIME|CANCELLED|DI
STANCE|
+--------+----------+----------------+------+---------------+----+-------------+-----------+--------+---------+-------+-----------+--------+---------+--
------+
|1/1/2000|        DL|            1451|   BOS|      Boston, MA| ATL|   Atlanta, GA|       1115|    1113|     1343|      5|       1400|    1348|        0|
946|
|1/1/2000|        DL|            1479|   BOS|      Boston, MA| ATL|   Atlanta, GA|       1315|    1311|     1536|      7|       1559|    1543|        0|
946|
|1/1/2000|        DL|            1857|   BOS|      Boston, MA| ATL|   Atlanta, GA|       1415|    1414|     1642|      9|       1721|    1651|        0|
946|
|1/1/2000|        DL|            1997|   BOS|      Boston, MA| ATL|   Atlanta, GA|       1715|    1720|     1955|     10|       2013|    2005|        0|
946|
|1/1/2000|        DL|            2065|   BOS|      Boston, MA| ATL|   Atlanta, GA|       2015|    2010|     2230|     10|       2300|    2240|        0|
946|
+--------+----------+----------------+------+---------------+----+-------------+-----------+--------+---------+-------+-----------+--------+---------+--
------+
only showing top 5 rows

struct<FL_DATE:string,OP_CARRIER:string,OP_CARRIER_FL_NUM:string,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:string,
DEP_TIME:string,WHEELS_ON:string,TAXI_IN:string,CRS_ARR_TIME:string,ARR_TIME:string,CANCELLED:string,DISTANCE:string>

Command took 0.56 seconds -- by prashant@scholarnest.com at 7/10/2022, 7:08:26 AM on demo-cluster
```


However, all of these fields are given a String Datatype.
Why? Because we didn't infer the schema.

```
1  flight_time_csv_df.show(5)
2  print(flight_time_csv_df.schema.simpleString())
```

(1) Spark Jobs

```
------+
| FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|ORIGIN_CITY_NAME|DEST|DEST_CITY_NAME|CRS_DEP_TIME|DEP_TIME|WHEELS_ON|TAXI_IN|CRS_ARR_TIME|ARR_TIME|CANCELLED|DI
STANCE|
+--------+----------+-----------------+------+---------------+----+-------------+------------+--------+---------+-------+------------+--------+---------+--
------+
|1/1/2000|        DL|             1451|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1115|    1113|     1343|      5|        1400|    1348|        0|
946|
|1/1/2000|        DL|             1479|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1315|    1311|     1536|      7|        1559|    1543|        0|
946|
|1/1/2000|        DL|             1857|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1415|    1414|     1642|      9|        1721|    1651|        0|
946|
|1/1/2000|        DL|             1997|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1715|    1720|     1955|     10|        2013|    2005|        0|
946|
|1/1/2000|        DL|             2065|   BOS|      Boston, MA| ATL|   Atlanta, GA|        2015|    2010|     2230|     10|        2300|    2240|        0|
946|
+--------+----------+-----------------+------+---------------+----+-------------+------------+--------+---------+-------+------------+--------+---------+--
------+
only showing top 5 rows
```

```
struct<FL_DATE:string,OP_CARRIER:string,OP_CARRIER_FL_NUM:string,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:string,
DEP_TIME:string,WHEELS_ON:string,TAXI_IN:string,CRS_ARR_TIME:string,ARR_TIME:string,CANCELLED:string,DISTANCE:string>
```

Command took 0.56 seconds -- by prashant@scholarnest.com at 7/10/2022, 7:08:26 AM on demo-cluster

So, I am setting the option to infer the schema here. Now let's re-run it.

Cmd 2

```
1  flight_time_csv_df = spark.read \
2                       .format("csv") \
3                       .option("header", "true") \
4                       .option("inferSchema", "true") \
5                       .load("/FileStore/tables/dataframeReader/flight*.csv")
```

▸ (1) Spark Jobs

Command took 1.22 seconds -- by prashant@scholarnest.com at 7/10/2022, 7:07:16 AM on demo-cluster

Now it is a little better.
My numeric fields are now inferred to be an integer. That's what I wanted. However, my date filed is still a string.

▸ (1) Spark Jobs

```
------+
| FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|ORIGIN_CITY_NAME|DEST|DEST_CITY_NAME|CRS_DEP_TIME|DEP_TIME|WHEELS_ON|TAXI_IN|CRS_ARR_TIME|ARR_TIME|CANCELLED|DI
STANCE|
+--------+----------+-----------------+------+---------------+----+-------------+------------+--------+---------+-------+------------+--------+---------+--
------+
|1/1/2000|        DL|             1451|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1115|    1113|     1343|      5|        1400|    1348|        0|
946|
|1/1/2000|        DL|             1479|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1315|    1311|     1536|      7|        1559|    1543|        0|
946|
|1/1/2000|        DL|             1857|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1415|    1414|     1642|      9|        1721|    1651|        0|
946|
|1/1/2000|        DL|             1997|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1715|    1720|     1955|     10|        2013|    2005|        0|
946|
|1/1/2000|        DL|             2065|   BOS|      Boston, MA| ATL|   Atlanta, GA|        2015|    2010|     2230|     10|        2300|    2240|        0|
946|
+--------+----------+-----------------+------+---------------+----+-------------+------------+--------+---------+-------+------------+--------+---------+--
------+
only showing top 5 rows
```

```
struct<FL_DATE:string,OP_CARRIER:string,OP_CARRIER_FL_NUM:int,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:int,DEP_TI
ME:int,WHEELS_ON:int,TAXI_IN:int,CRS_ARR_TIME:int,ARR_TIME:int,CANCELLED:int,DISTANCE:int>
```

Command took 0.93 seconds -- by prashant@scholarnest.com at 7/10/2022, 7:10:16 AM on demo-cluster

You cannot rely on the infer schema option. It doesn't always infer the schema correctly.
I have a date filed in this dataset, but the schema inference is showing it as a string.
So what options do we have? You have got only two options:

1. Explicitly specify a schema
2. Use a data file format that comes with the schema

Here is the code to read the JSON file. I have copied the code for CSV file and modified it.
I have changed the name of the Dataframe, and also the format to JSON.
I have removed the header option because the JSON doesn't come with a header row.
You may want to keep the inferSchema, but the inferSchema doesn't make any sense for JSON. Why? Because the Dataframe reader always infers the schema from JSON and you do not need to give this option. So I have removed the infer schema option as well. Finally, I changed the file name at the end, gave a print statement to see the schema.

Cmd 4

```
1  flight_time_json_df = spark.read \
2                        .format("json") \
3                        .load("/FileStore/tables/dataframeReader/flight*.json")
4
5  print(flight_time_json_df.schema.simpleString())
```


The JSON format also sorted the column names in alphabetical order. And that's fine.
However, I do see a similar behaviour for inference.
My integer columns are now *BigInt*, but my date column is still a string.
So we know schema inference is not a good option.
It didn't work well for CSV, and JSON is also not good at schema inference.

Cmd 4

```
1  flight_time_json_df = spark.read \
2                        .format("json") \
3                        .load("/FileStore/tables/dataframeReader/flight*.json")
4
5  print(flight_time_json_df.schema.simpleString())
```

▸ (1) Spark Jobs

```
struct<ARR_TIME:bigint,CANCELLED:bigint,CRS_ARR_TIME:bigint,CRS_DEP_TIME:bigint,DEP_TIME:bigint,DEST:string,DEST_CITY_NAME:string,DISTANCE:bigint,FL_DATE:str
ing,OP_CARRIER:string,OP_CARRIER_FL_NUM:bigint,ORIGIN:string,ORIGIN_CITY_NAME:string,TAXI_IN:bigint,WHEELS_ON:bigint>
```

Command took 8.07 seconds -- by prashant@scholarnest.com at 7/10/2022, 7:15:44 AM on demo-cluster

Now comes the second question:

How do we fix the schema problem?

And I already mentioned that you have two options:

1. Explicitly specify a schema
2. Use a data file format that comes with the schema.

I will come to the explicitly specifying schema in the next lecture.
For now, let's look at the second option of using a file format that comes with an in-built schema.

We also have a parquet data file.
The parquet is a binary file format, and it comes with the schema information already included in the data file.
So, you do not need to specify the schema explicitly.
The DataFrameReader will automatically load the schema information from the parquet file itself.
However, your parquet data file should contain the correct schema.
If your schema is incorrect in the file, you will load it incorrectly.

I created the parquet file purposefully with accurate schema information for all the fields.
Let's load out the parquet file and see if we get an appropriate schema.

I have taken the same code again and modified it according to parquet file requirements. You should change the Dataframe name at both places. I will also change the format and the file name. And we will again print the schema information.
You can see the output as well.
My parquet file contains the well-defined correct schema along with the data. So, the Dataframe reader for the parquet source loaded it with the same schema. Now you can see the int as well as the date field.

Cmd 5

```
1  flight_time_parquet_df = spark.read \
2                           .format("parquet") \
3                           .load("/FileStore/tables/dataframeReader/flight*.parquet")
4
5  print(flight_time_parquet_df.schema.simpleString())
```

▸ (1) Spark Jobs

```
struct<FL_DATE:date,OP_CARRIER:string,OP_CARRIER_FL_NUM:int,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:int,DEP_TIM
E:int,WHEELS_ON:int,TAXI_IN:int,CRS_ARR_TIME:int,ARR_TIME:int,CANCELLED:int,DISTANCE:int>
```

Command took 1.94 seconds -- by prashant@scholarnest.com at 7/10/2022, 7:18:50 AM on demo-cluster

You should prefer using the Parquet file format as long as it is possible.
And the parquet file format is the recommended and the default file format for Apache Spark.
Your data integration tools might bring the original data source in CSV or in JSON format.
However, if you have a choice, then you must try to get your data in the Parquet file format.
And most of the data integration tools are now supporting the creation of parquet files.
So, I recommend you stick to parquet files for your Spark data processing.

Assignment:

We are now left with one Avro file.

I have an Avro file to represent the same data.

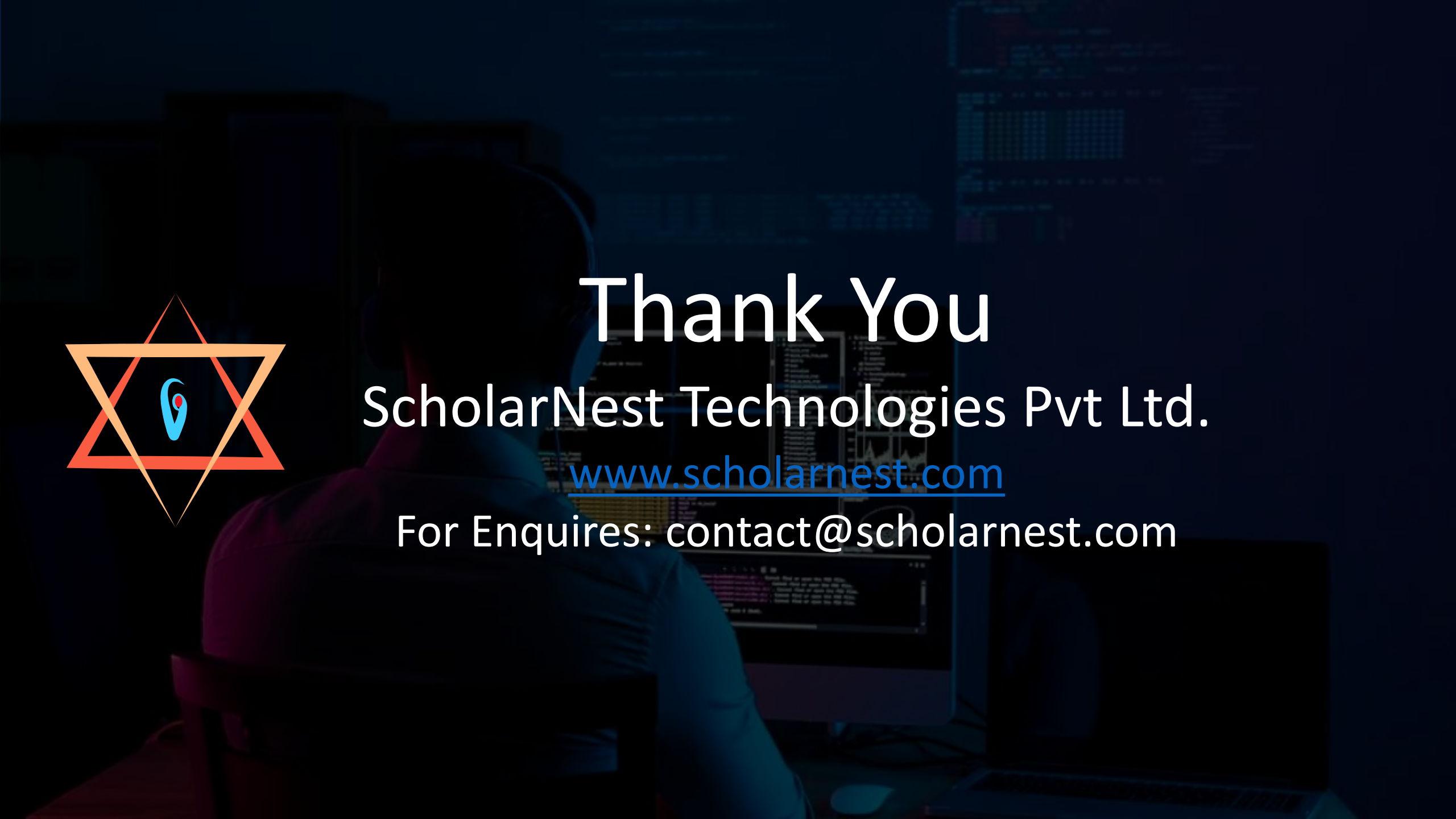
The Avro file is also a binary file format, and it comes with embedded schema information like parquet.

You can load the Avro data file in the same way you loaded the parquet.

The only change will be to change the format to Avro, and the rest of everything should work.

I leave it as an exercise for you to load the Avro data file and create a Spark Dataframe.

Check the schema information and let me know if you face some problems or see an issue.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Data Sources & Sinks

Lecture:
Schema On-Read

Reading data with schema-on-read

Dataframe schema is all about setting the column name and appropriate data types.
However, you should also know the Spark data types to define the schema correctly.

We already learned about the spark Data types and defining schema in the earlier lectures.
In this lecture, I will quickly recap those concepts and help you apply the schema-on-read with different types of file formats.

So, let's try to recap the Spark Types before discussing the schema. Apache Spark comes with its own data types. Here are the ten most commonly used types shown below.
However, you can find the complete list in the org.apache.spark.sql.types package.

Spark Data Types

S.No.	Spark Types	Spark SQL Types	Description
1.	IntegerType	INT, INTEGER	32 bit signed integer between -2147483648 to 2147483647
2.	LongType	LONG, BIGINT	64 bit signed integer between 9223372036854775808 to 9223372036854775807
3.	DoubleType	DOUBLE	64 bit double precision floating point number
4.	BooleanType	BOOLEAN	
5.	StringType	STRING	
6.	DateType	DATE	
7.	TimestampTyp	TIMESTAMP	
8.	ArrayType	ARRAY	
9.	MapType	MAP	
10.	StructType	STRUCT	

The Spark data types simply correspond to their equivalent Python types. However, we define the Spark Dataframe schema using the Spark Types. You might be wondering why Spark maintains its own types. Why don't we simply use the Python types? Well, there is a valid reason for this.
Spark is like a compiler.
It complies with the high-level Dataframe API code into low-level RDD operations.
But for now, you can consider Spark as a compiler. It compiles the Dataframe API into low-level RDD operation.
And during this compilation process, it generates different execution plans and also performs a bunch of optimizations.
This all is not at all possible for the Spark engine without maintaining its own type of information.

And this approach is not new.
Every SQL database would have a set of SQL data types. Similarly, Spark also works on Spark data types.

Now that you refreshed your memories about the Spark types.
Let's move on to the schema.
Spark allows you to define a schema in two ways.

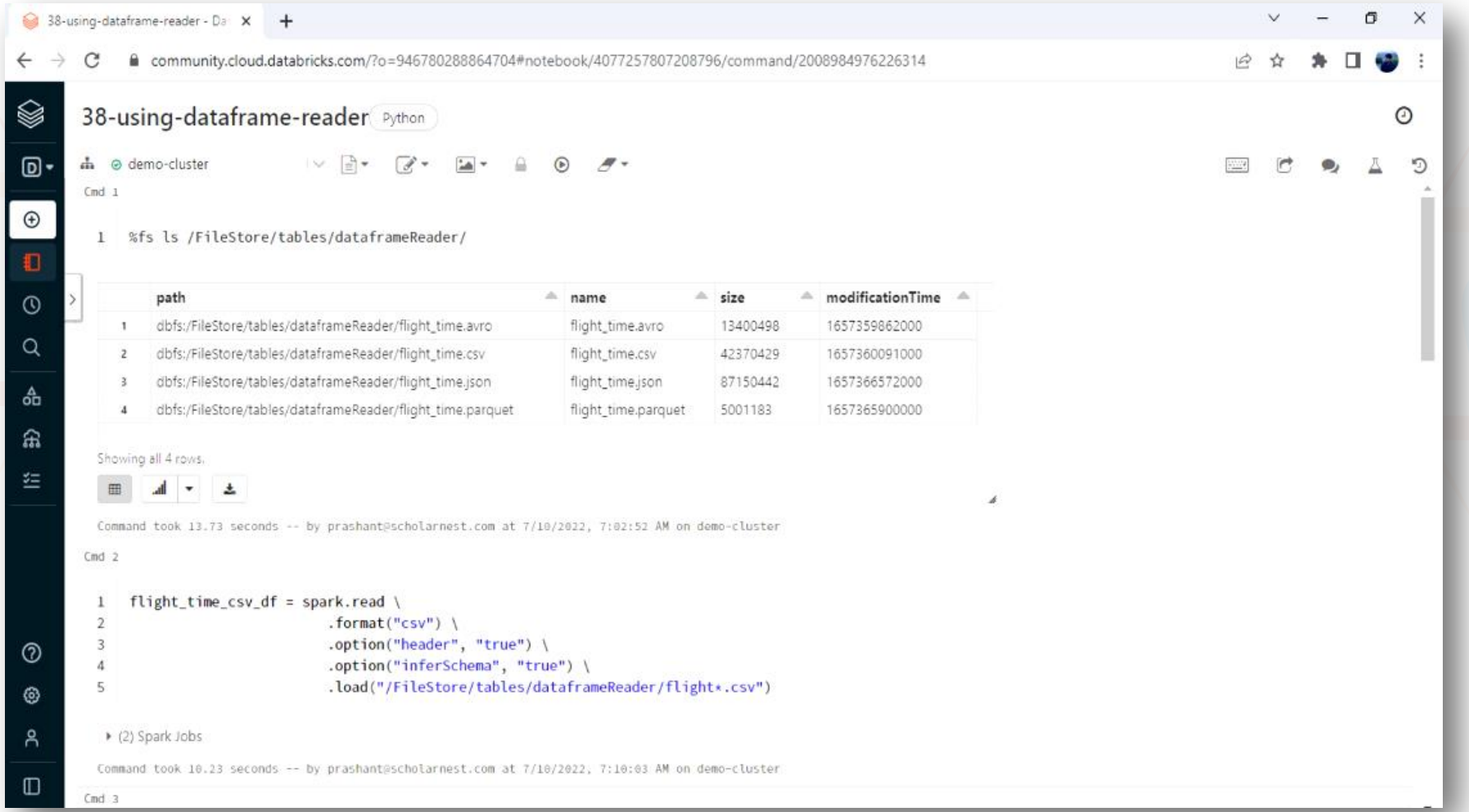
1. Programmatically
2 Using DDL String

The second method is much simpler and easier. We have already learned both methods.

Re-open the notebook which we created in the previous lecture. **(Reference: 38-using-Dataframe-reader.ipynb)**

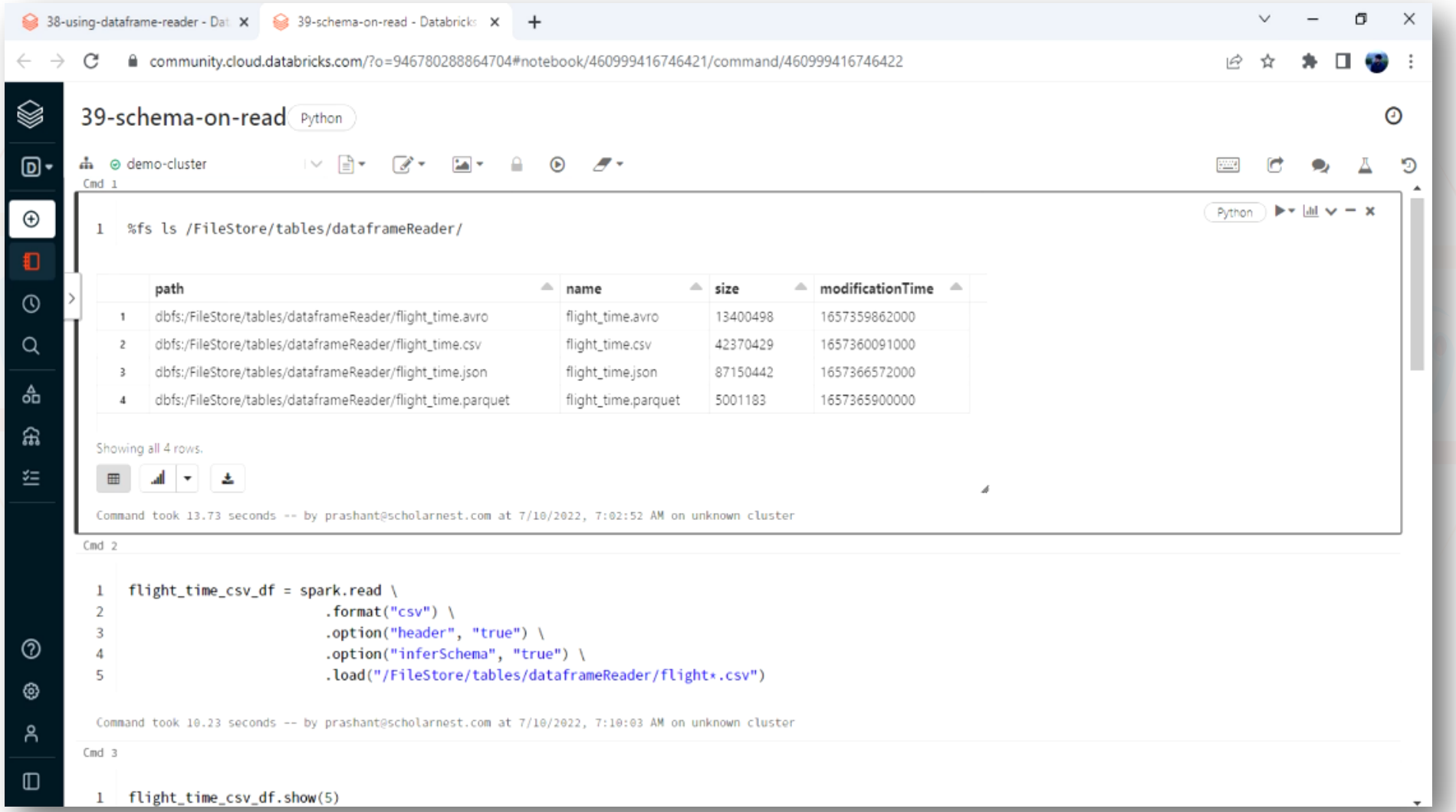
In this example, we read the same data using four different data file sources.

Now I want to define the schema for this data set and load the data using the schema-on-read option.



I cloned the previous notebook and give it a new name. **(Reference: 39-schema-on-read.ipynb)**

Also, delete the first cell.



Add a new cell at the top and define the schema using the programmatic method.
A Spark Dataframe Schema is a StructType that comprises a list of StructField.
The StructField takes two mandatory arguments.
The first one is a column name, and the second one is the data type.
You must be using one of the Spark Data Types.
These Spark types are defined in the pyspark.sql.types package, so I am also importing all the types.

Cmd 1

```
1  from pyspark.sql.types import *
2
3  flight_schema_struct = StructType([
4          StructField("FL_DATE", DateType())
5  ])
```


I have added more fields to complete the schema definition.
The StructType represents a DataFrame row structure, and the StructField is a column definition.

```
1  from pyspark.sql.types import *
2
3  flight_schema_struct = 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 7/10/2022, 5:01:00 PM on demo-cluster
```


Now you can use this schema to load your CSV file to a Dataframe and remove the inferSchema option.
However, we still keep the header option because we want to skip the first row in the CSV file.

Cmd 2

```
1  flight_time_csv_df = spark.read \
2                       .format("csv") \
3                       .option("header", "true") \
4                       .schema(flight_schema_struct) \
5                       .load("/FileStore/tables/dataframeReader/flight*.csv")
```

Command took 10.23 seconds -- by prashant@scholarnest.com at 7/10/2022, 7:10:03 AM on unknown cluster

If you look at the schema structure, I am using three Spark Types.
DateType, StringType, and IntegerType.
If the types in the data file do not match the schema at runtime, Spark should throw an error.
However, you must set the mode for getting the error.

Cmd 1

```
1  from pyspark.sql.types import *
2
3  flight_schema_struct = 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 7/10/2022, 5:01:00 PM on demo-cluster

1. DateType
2. StringType
3. IntegerType

I added a new option to set the mode.
I defined the schema correctly, and I am not expecting any error.
You can see that the spark.read method worked without any error.

Cmd 2

```
1  flight_time_csv_df = spark.read \
2                           .format("csv") \
3                           .option("header", "true") \
4                           .schema(flight_schema_struct) \
5                           .option("mode", "FAILFAST") \
6                           .load("/FileStore/tables/dataframeReader/flight*.csv")
```

Command took 10.23 seconds -- by prashant@scholarnest.com at 7/10/2022, 7:10:03 AM on unknown cluster

However, if I try to run the next cell and show the Dataframe, you will see that we got an exception. It says FileReadException: Error while reading the file.
The error message is not intuitive.

If you expand the error message, you will see that you do not see a clear error message indicating the main cause of the failure. However, I can see that the if isinstance() method call throws an error. And that gives me an indication of something wrong with the data type validation. The isinstance() is used for checking the data types.

So which field data type could be causing the error?
Close this error message, and let's look at our schema definition.
I have three data types in this schema:

1. Date
2. String
3. Integer

The string type is not going to cause a data type error. The integer and date types are the potential problems. I have one date type and many integer types.

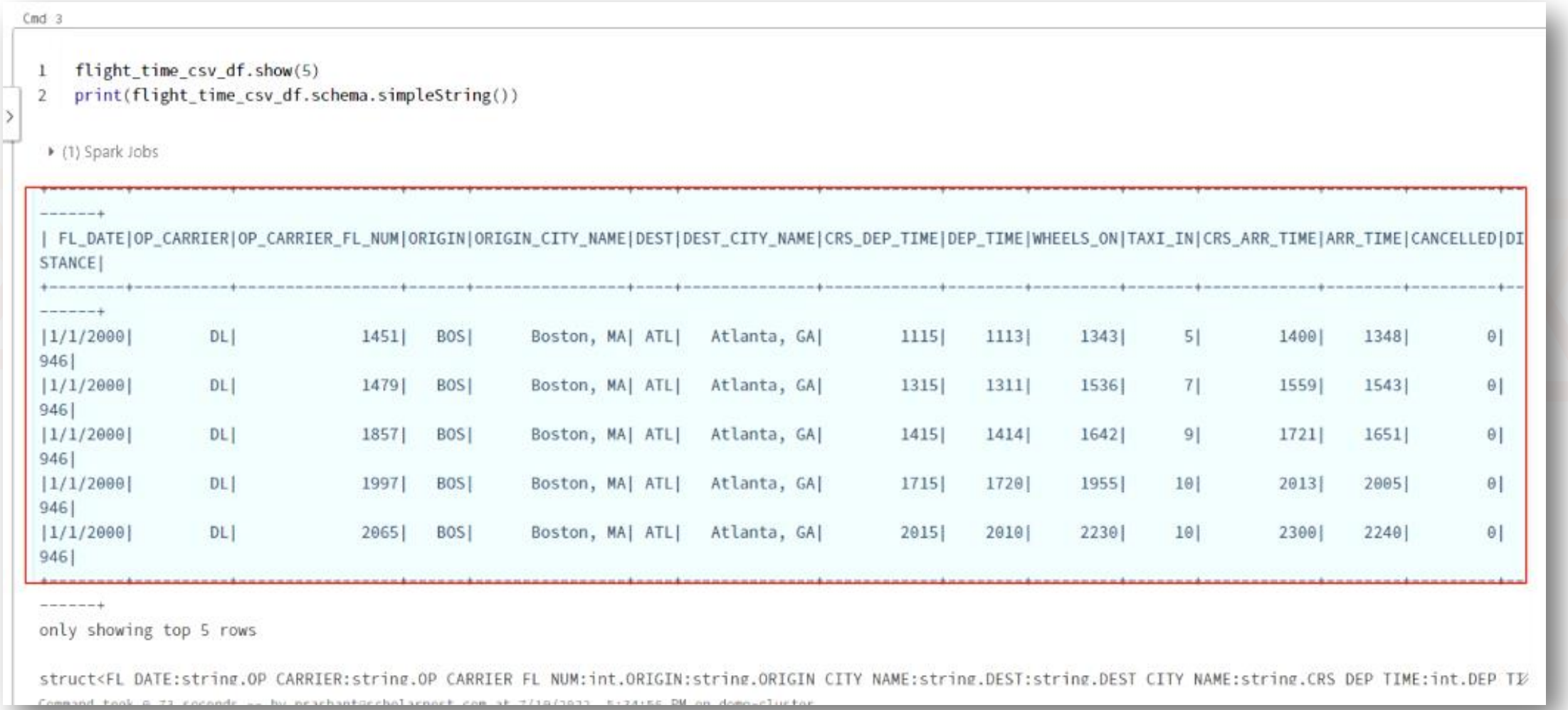
Check the date type first and see if that one is the culprit.
Change the DateType() to StringType() in the schema and start rerunning it from there.

Cmd 1

```
1  from pyspark.sql.types import *
2
3  flight_schema_struct = 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 7/10/2022, 5:01:00 PM on demo-cluster

Now it worked. I can see the data.
So now we know it was the FL_DATE causing the problem. I will fix it. But before that I will change the schema for the FL_DATE back to DateType() and rerun the cell.



Cmd 3

```
1  flight_time_csv_df.show(5)
2  print(flight_time_csv_df.schema.simpleString())
```

▸ (1) Spark Jobs

```
+--------+----------+----------------+------+---------------+----+-------------+-----------+--------+---------+------+-----------+--------+---------+--
------+
| FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|ORIGIN_CITY_NAME|DEST|DEST_CITY_NAME|CRS_DEP_TIME|DEP_TIME|WHEELS_ON|TAXI_IN|CRS_ARR_TIME|ARR_TIME|CANCELLED|DI
STANCE|
+--------+----------+----------------+------+---------------+----+-------------+-----------+--------+---------+------+-----------+--------+---------+--
------+
|1/1/2000|        DL|            1451|   BOS|      Boston, MA| ATL|   Atlanta, GA|       1115|    1113|     1343|     5|       1400|    1348|        0|
946|
|1/1/2000|        DL|            1479|   BOS|      Boston, MA| ATL|   Atlanta, GA|       1315|    1311|     1536|     7|       1559|    1543|        0|
946|
|1/1/2000|        DL|            1857|   BOS|      Boston, MA| ATL|   Atlanta, GA|       1415|    1414|     1642|     9|       1721|    1651|        0|
946|
|1/1/2000|        DL|            1997|   BOS|      Boston, MA| ATL|   Atlanta, GA|       1715|    1720|     1955|    10|       2013|    2005|        0|
946|
|1/1/2000|        DL|            2065|   BOS|      Boston, MA| ATL|   Atlanta, GA|       2015|    2010|     2230|    10|       2300|    2240|        0|
946|
+--------+----------+----------------+------+---------------+----+-------------+-----------+--------+---------+------+-----------+--------+---------+--
------+
only showing top 5 rows

struct<FL_DATE:string,OP_CARRIER:string,OP_CARRIER_FL_NUM:int,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:int,DEP_TI
```


Now we know that the Spark DataFrameReader cannot read and parse the FL_DATE field as DateType. But how do I fix it? Well, our CSV data source came with a month/day and year format. And the DataFrameReader doesn't know that information. So you can set another option to inform the DataFrameReader about the date format in the data file.

Cmd 1

```
1  from pyspark.sql.types import *
2
3  flight_schema_struct = 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 0.04 seconds -- by prashant@scholarnest.com at 7/10/2022, 5:35:36 PM on demo-cluster

I added a new option and informed the DataFrameReader that my date string can be defined using the "M/d/y".

Cmd 2

```
1  flight_time_csv_df = spark.read \
2                           .format("csv") \
3                           .option("header", "true") \
4                           .schema(flight_schema_struct) \
5                           .option("mode", "FAILFAST") \
6                           .option("dateFormat", "M/d/y") \
7                           .load("/FileStore/tables/dataframeReader/flight*.csv")
```

Command took 0.66 seconds -- by prashant@scholarnest.com at 7/12/2022, 8:46:19 PM on demo-cluster

We can see the Dataframe, and my data looks good. And the schema is also perfect.
The date field is now a Date.
So you learned to define and use the schema. We defined the schema using the StructType() and StructField().

Cmd 3

```
1  flight_time_csv_df.show(5)
2  print(flight_time_csv_df.schema.simpleString())
```

(1) Spark Jobs

```
--------+
|   FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|ORIGIN_CITY_NAME|DEST|DEST_CITY_NAME|CRS_DEP_TIME|DEP_TIME|WHEELS_ON|TAXI_IN|CRS_ARR_TIME|ARR_TIME|CANCELLED|
DISTANCE|
+----------+----------+-----------------+------+---------------+----+-------------+------------+--------+---------+-------+------------+--------+---------+
--------+
|2000-01-01|        DL|             1451|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1115|    1113|     1343|      5|        1400|    1348|        0|
946|
|2000-01-01|        DL|             1479|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1315|    1311|     1536|      7|        1559|    1543|        0|
946|
|2000-01-01|        DL|             1857|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1415|    1414|     1642|      9|        1721|    1651|        0|
946|
|2000-01-01|        DL|             1997|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1715|    1720|     1955|     10|        2013|    2005|        0|
946|
|2000-01-01|        DL|             2065|   BOS|      Boston, MA| ATL|   Atlanta, GA|        2015|    2010|     2230|     10|        2300|    2240|        0|
946|
+----------+----------+-----------------+------+---------------+----+-------------+------------+--------+---------+-------+------------+--------+---------+
--------+
only showing top 5 rows

struct<FL_DATE:date,OP_CARRIER:string,OP_CARRIER_FL_NUM:int,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:int,DEP_TIM
E:int,WHEELS_ON:int,TAXI_IN:int,CRS_ARR_TIME:int,ARR_TIME:int,CANCELLED:int,DISTANCE:int>

Command took 5.29 seconds -- by prashant@scholarnest.com at 7/13/2022, 9:08:21 AM on demo-cluster
```


Here I have defined the schema DDL. The Schema DDL is straightforward.
All you need is the column name and data type separated by a comma.
And you can use this schema in the same way as a Programmatically defined schema.
I have made a small change in the DDL schema. Look at the CANCELLED field. I defined it as a string. Earlier I defined it as an int. Let me show you the previous schema definition.
So the CANCELLED field was an IntegerType(). But in the DDL schema, I defined it as a String type. Why? Just an experiment to see schema-on-read in action.

```
Cmd 4

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 STRING, DISTANCE INT"""

Command took 0.05 seconds -- by prashant@scholarnest.com at 7/13/2022, 9:12:21 AM on demo-cluster
```


I wanted to show you that you can use your own choice of types in the schema.
The data in the file is an integer.
However, I want to read it as a string, and Spark doesn't complain.
Why? Because an integer can be easily converted into a string without an error.
And Spark will automatically do that conversion for us.
That's the power of schema-on-read.
I am defining a schema and reading the data using my schema.
The data could be different, and Spark will read it correctly as long as the data fits into my schema.
The integer easily fits into a string, so Spark will not complain.
The String data couldn't fit into the Date, so we saw an error.
However, as soon as we informed the date format, Spark could fit the string date into a date type, and it worked.

We are reading the JSON file in the next cell. So let me try the DDL schema for the JSON source. Do not forget to specify the date format. Without telling the date format, Spark will throw an error.
We already saw it in the CSV example. The JSON is also the same.
I am also setting the fail-fast mode, so we see an error in case of any problems.
You can see that this code worked, and look at the data type of the CANCELLED field. It is shown as a string.

Cmd 5

```
1  flight_time_json_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/dataframeReader/flight*.json")
7
8  print(flight_time_json_df.schema.simpleString())
```

```
struct<FL_DATE:date,OP_CARRIER:string,OP_CARRIER_FL_NUM:int,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:int,DEP_TIM
E:int,WHEELS_ON:int,TAXI_IN:int,CRS_ARR_TIME:int,ARR_TIME:int,CANCELLED:string,DISTANCE:int>
```

Command took 0.80 seconds -- by prashant@scholarnest.com at 7/13/2022, 9:15:35 AM on demo-cluster

So we learned to use the schema-on-read with the CSV and JSON files.
What about the Parquet and AVRO files?

The Parquet and Avro files are binary file formats and come with schema information embedded in the data file.
We do not need to set the schema with these files.
But what if I do set a schema?
Will it throw an error saying we already have a schema in the Parquet file?
No! It won't.
Why? Because Spark supports schema-on-read.
So even if we already have schema information inside the file, I can set a different schema and apply the schema-on-read concept.

I ran the next cell and to see what we get without explicitly setting the schema.
I can see the schema here highlighted below.
Let's focus on the data type of the CANCELLED field. It is an Integer. Why Integer? Because that's what is defined inside the Parquet file.
The Parquet file comes with schema information embedded in the file.
And that schema defines the CANCELLED filed as an integer.

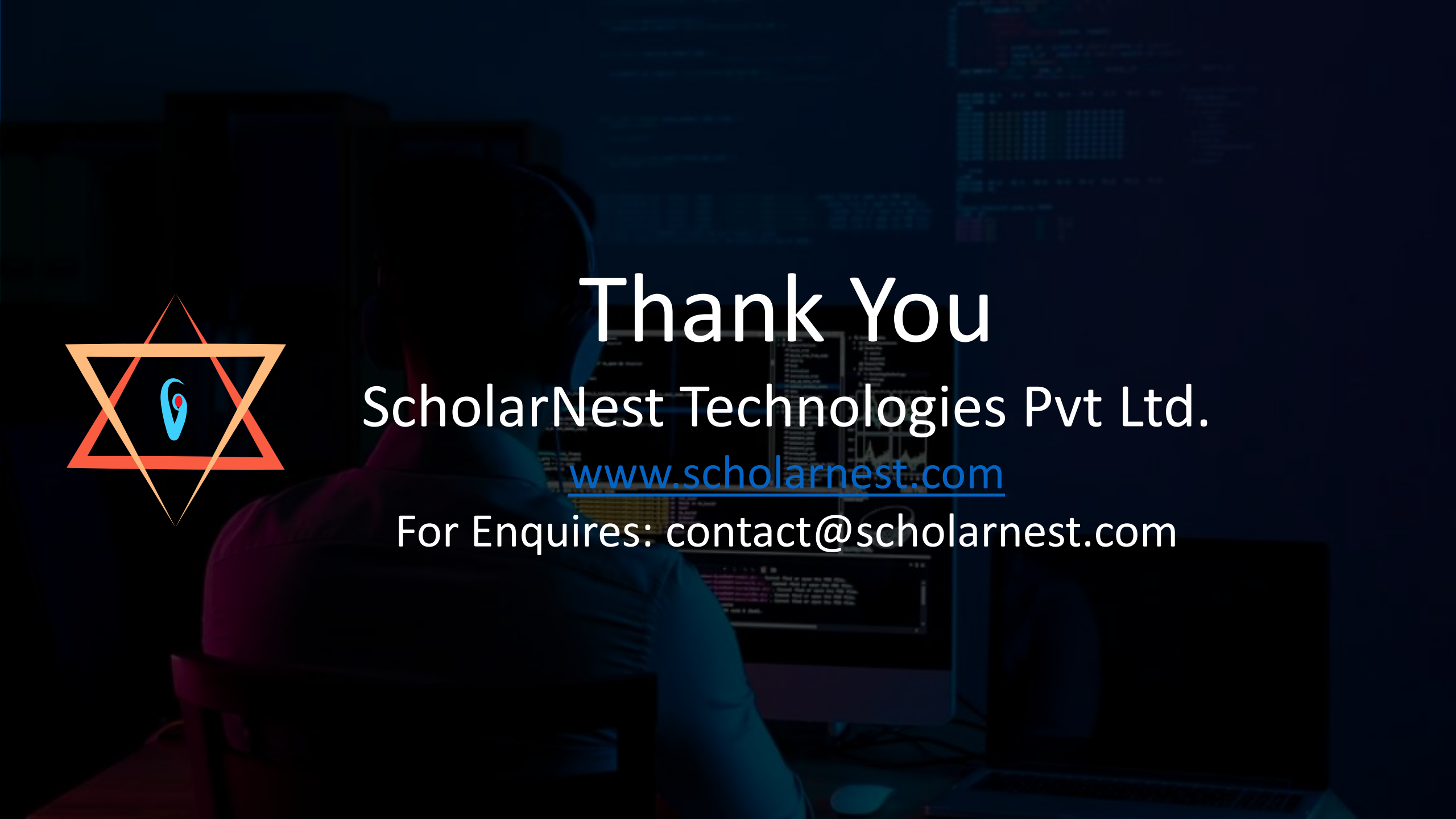
I am using the DDL schema.
And in this DDL schema, I defined the CANCELLED field as a string.
And you can see the same is replicated in the output below.
So, the Parquet and the Avro files do have an embedded schema inside the file.
So you do not need to specify the schema explicitly.
However, you can still supply your own schema and load the data with the new schema by applying the schema-on-read.

Cmd 6

```
1  flight_time_parquet_df = spark.read \
2                           .format("parquet") \
3                           .schema(flight_schema_ddl) \
4                           .load("/FileStore/tables/dataframeReader/flight*.parquet")
5
6  print(flight_time_parquet_df.schema.simpleString())
```

```
struct<FL_DATE:date,OP_CARRIER:string,OP_CARRIER_FL_NUM:int,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:int,DEP_TIM
E:int,WHEELS_ON:int,TAXI_IN:int,CRS_ARR_TIME:int,ARR_TIME:int,CANCELLED:string,DISTANCE:int>
```

Command took 0.60 seconds -- by prashant@scholarnest.com at 7/13/2022, 9:17:28 AM on demo-cluster



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Data Sources & Sinks

Lecture:
Dataframe Reader Options

DataFrameReader Options for CSV, JSON and Others

We learned to use the DataFrameReader API in the earlier lectures.
Here are some examples of using the Spark DataFrameReader API.

```
spark.read \
     .format("csv") \
     .option("header", "true") \
     .schema(flight_schema_struct) \
     .option("mode", "FAILFAST") \
     .option("dateFormat", "M/d/y") \
     .load("/FileStore/tables/dataframeReader/flight*.csv")
```

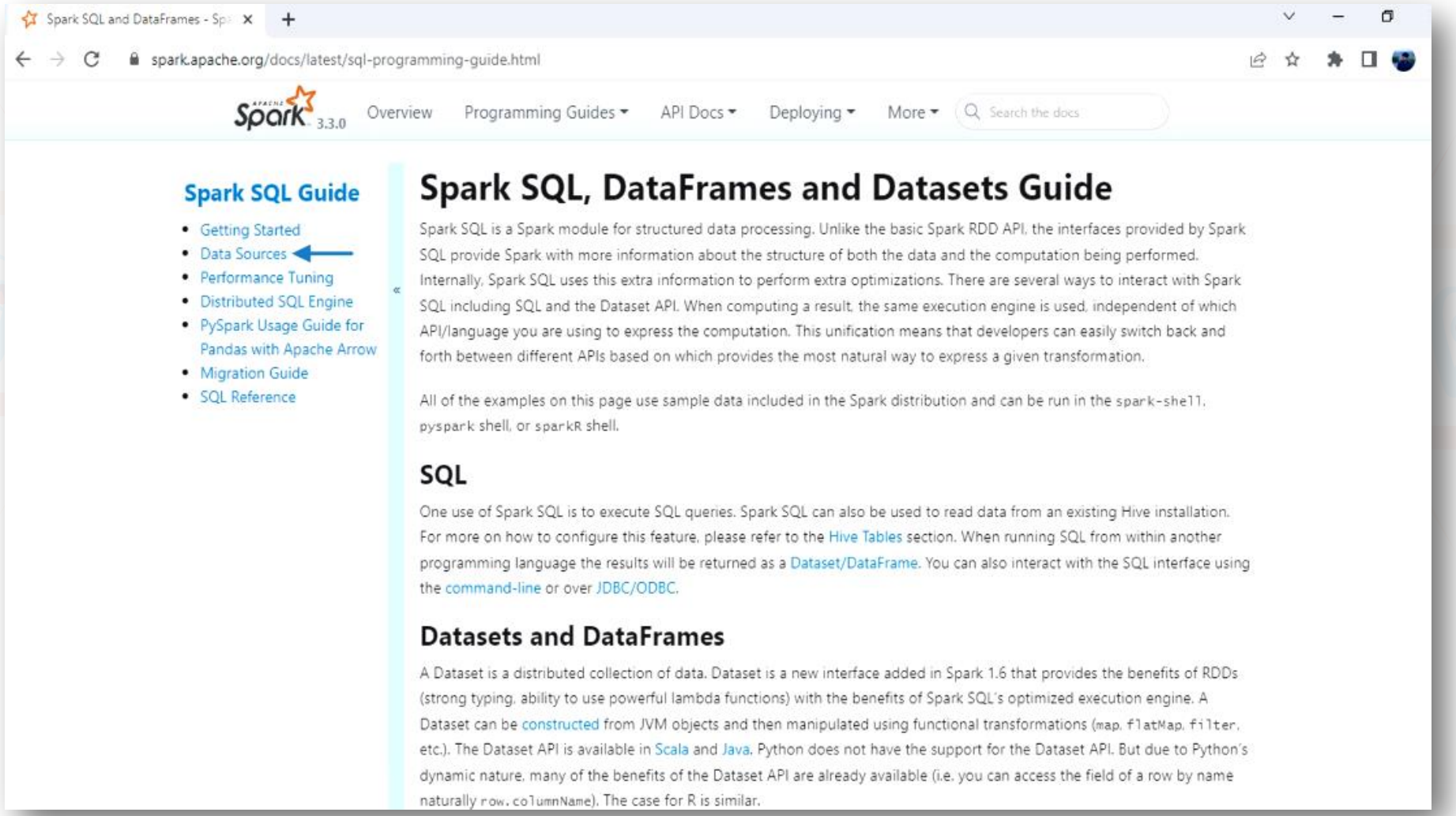
```
spark.read \
     .format("csv") \
     .option("header", "true") \
     .option("inferSchema", "true") \
     .load("/FileStore/tables/dataframeReader/flight*.csv")
```


The DataFrameReader API is super flexible, and all the flexibility is given through the options.
In these examples, we are using different options based on the data source format.
So these options are super important to handle different situations.
You might be wondering, is there a place I can get a list of all the available options. And that is the discussion for this chapter.

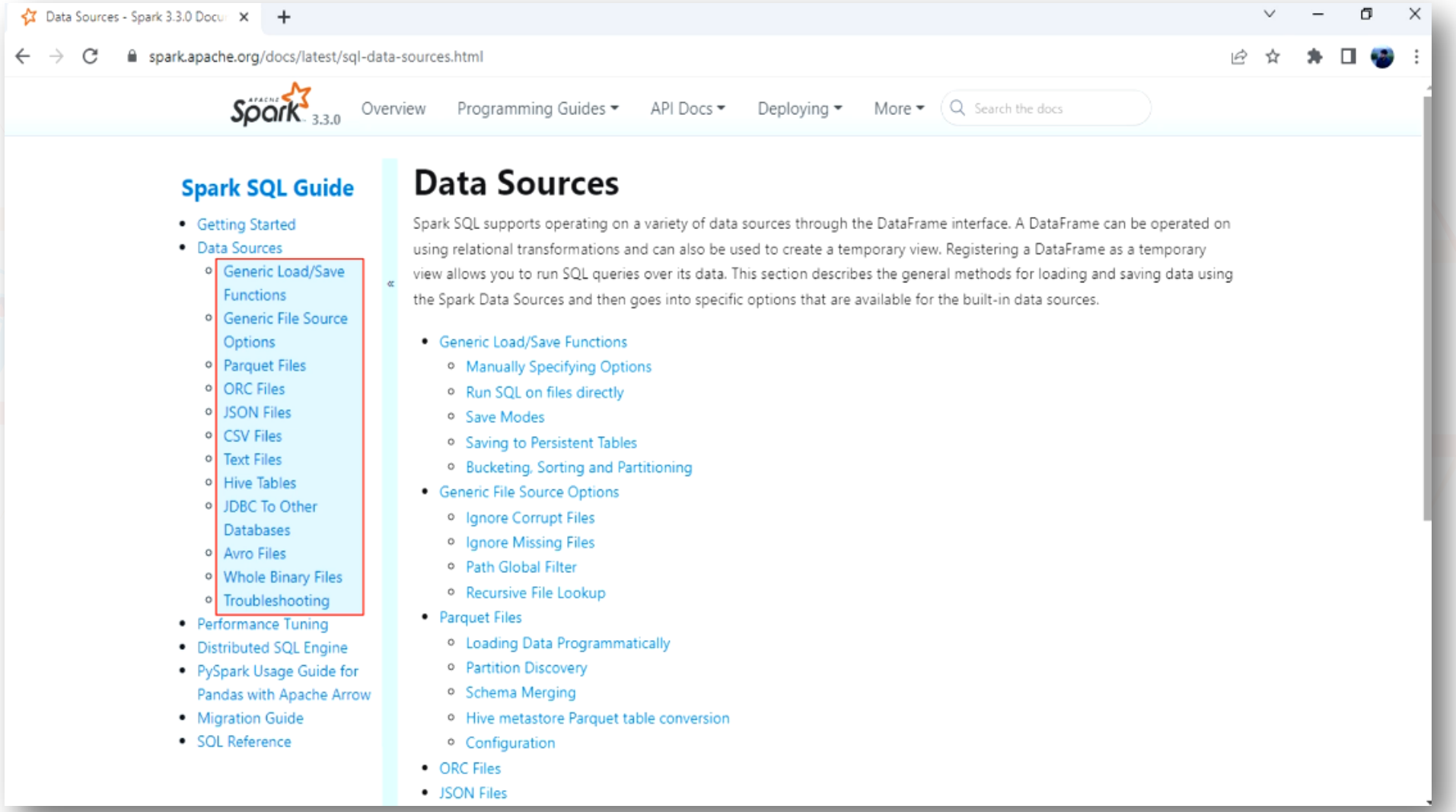
```
spark.read \
     .format("csv") \
     .option("header", "true") \
     .schema(flight_schema_struct) \
     .option("mode", "FAILFAST") \
     .option("dateFormat", "M/d/y") \
     .load("/FileStore/tables/dataframeReader/flight*.csv")
```

```
spark.read \
     .format("csv") \
     .option("header", "true") \
     .option("inferSchema", "true") \
     .load("/FileStore/tables/dataframeReader/flight*.csv")
```

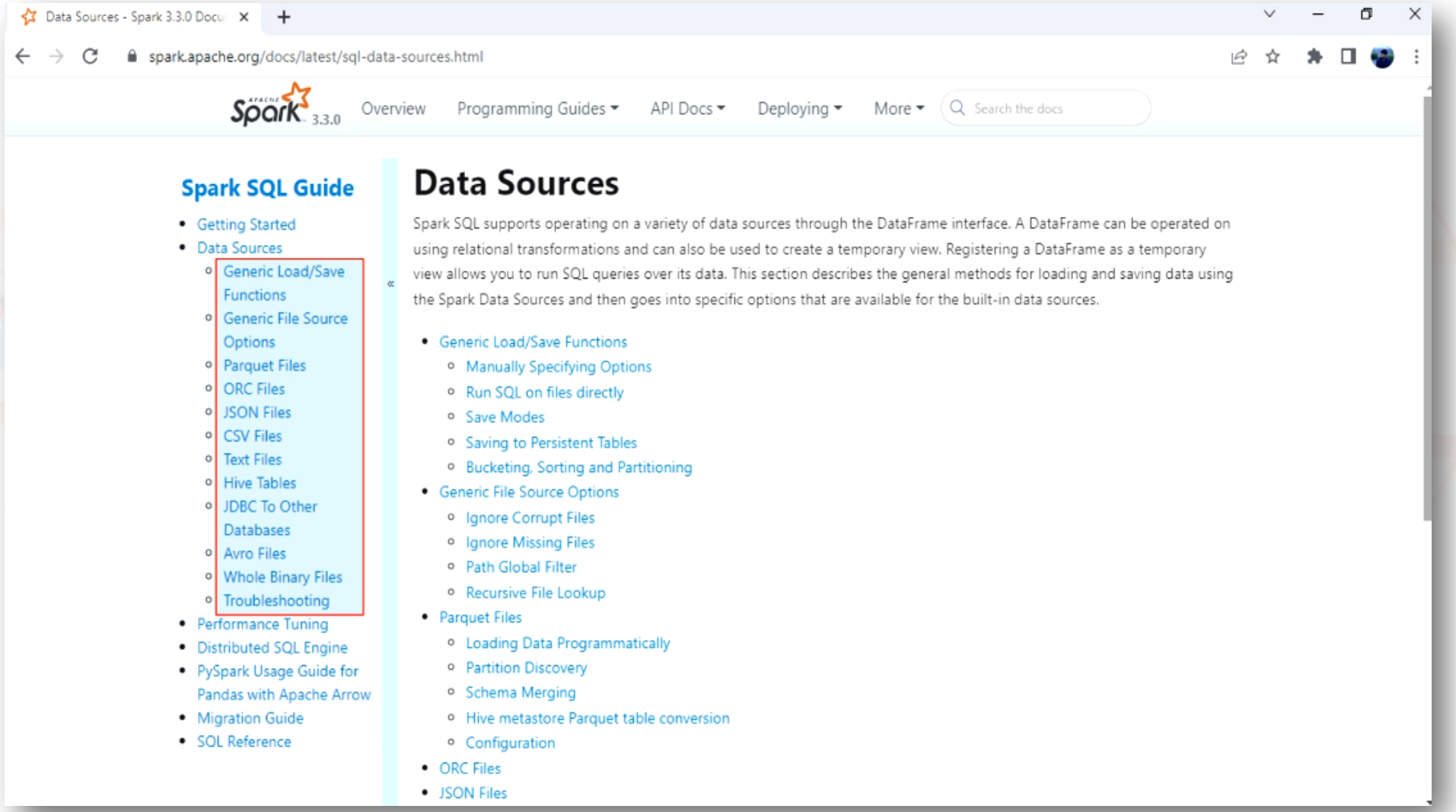

Go to the Spark SQL documentation.
(Reference: https://spark.apache.org/docs/latest/sql-programming-guide.html)
Do you see the link for Data Sources?
Click that.



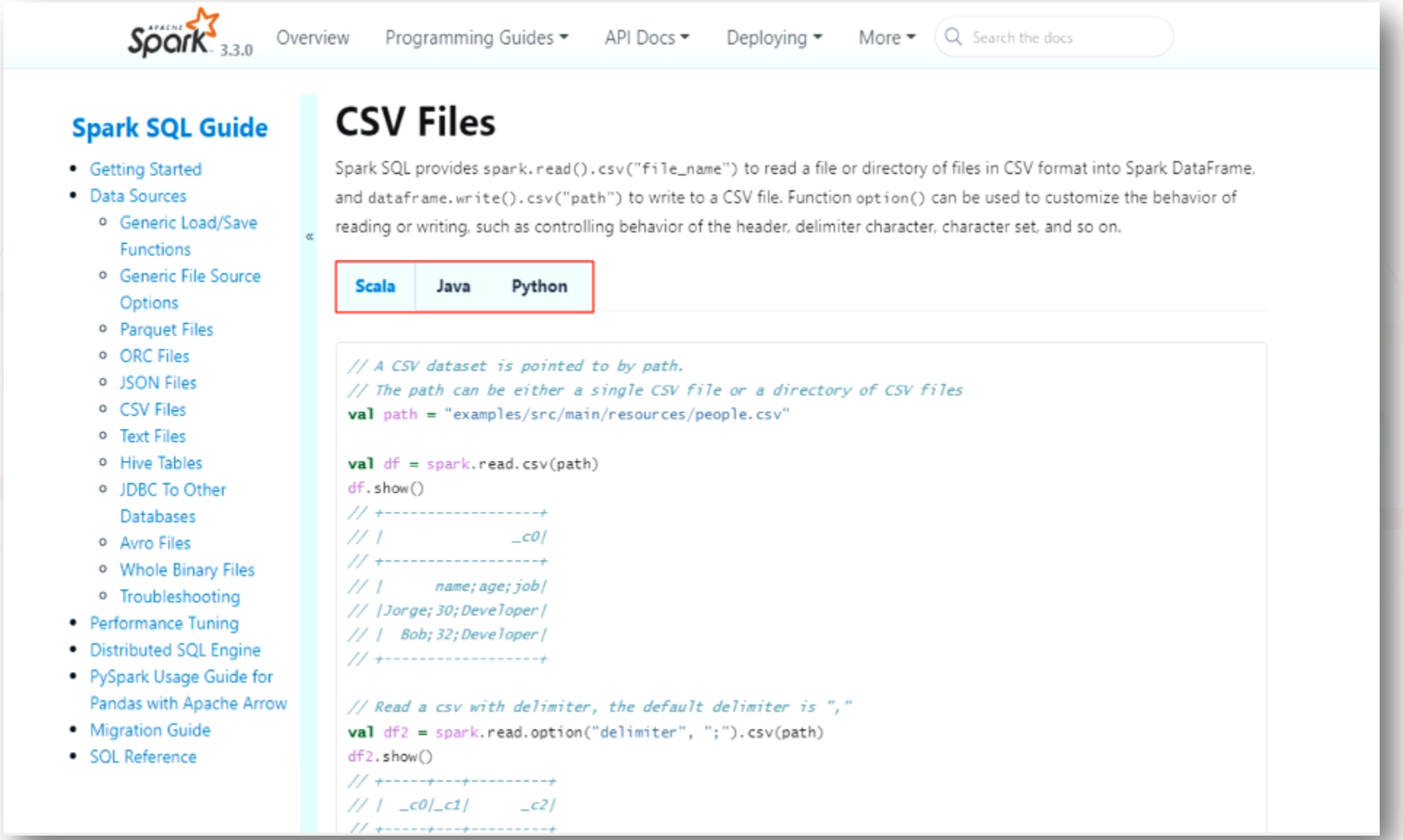
You will see a list of all Spark in-build data sources. You have Parquet, ORC, JSON, CSV, Text, Hive, JDBC, and Avro. This list documents all the Spark built-in data sources. You do not need any extra library or package to work with these data sources. And you can get the list of supported options here.



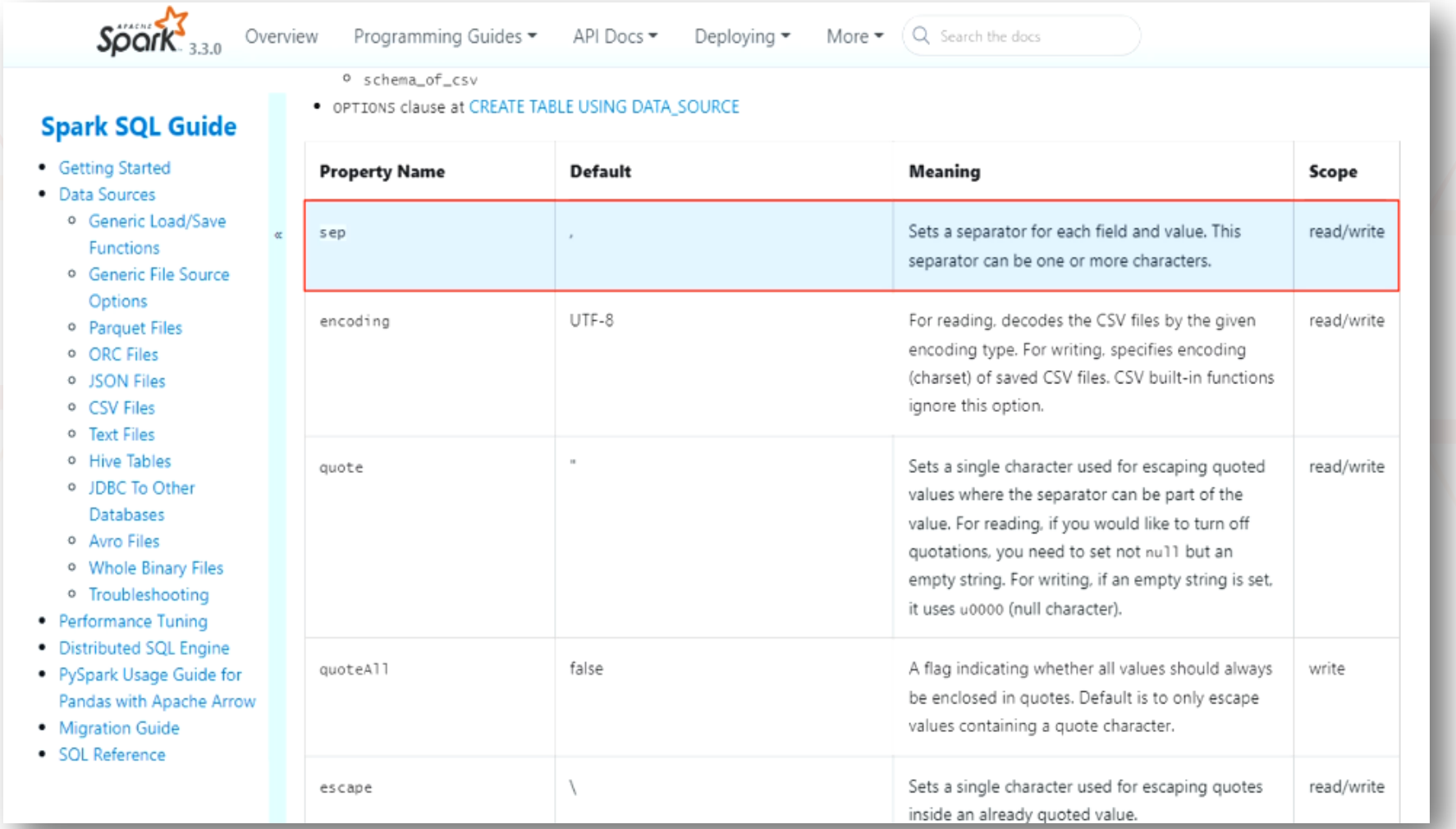
The DataFrameReader options are dependent on the format.
Some options make sense for the CSV format but are not relevant and support the JSON format.
And this is the place where you can get the complete list of all the supported format options.



Here is the option for CSV file. You can see Scala, Java, and Python examples.



If you scroll down, you will see a table of supported options. The first one is the separator option. The default value is a comma. So the CSN format allows you to load any delimited text file. It could be a comma-delimited file, tab-delimited, pipe-delimited, or whatever. You can set the separator option, inform the DataFrameReader about the correct delimiter, and read any delimited file.



Spark 3.3.0 Overview Programming Guides API Docs Deploying More Search the docs

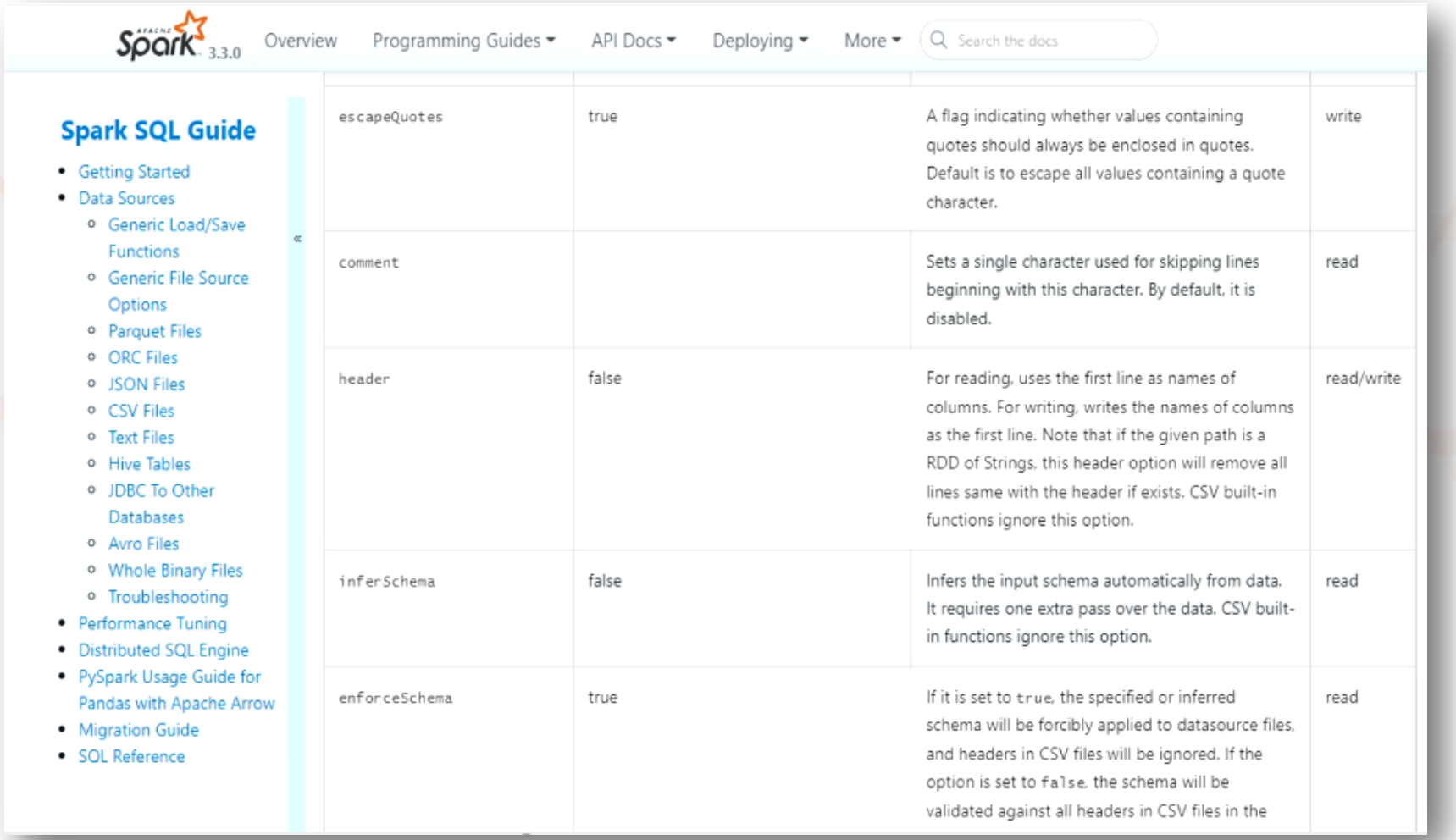
- schema_of_csv
- OPTIONS clause at CREATE TABLE USING DATA_SOURCE

Spark SQL Guide

- Getting Started
- Data Sources
  - Generic Load/Save Functions
  - Generic File Source Options
  - Parquet Files
  - ORC Files
  - JSON Files
  - CSV Files
  - Text Files
  - Hive Tables
  - JDBC To Other Databases
  - Avro Files
  - Whole Binary Files
  - Troubleshooting
- Performance Tuning
- Distributed SQL Engine
- PySpark Usage Guide for Pandas with Apache Arrow
- Migration Guide
- SQL Reference

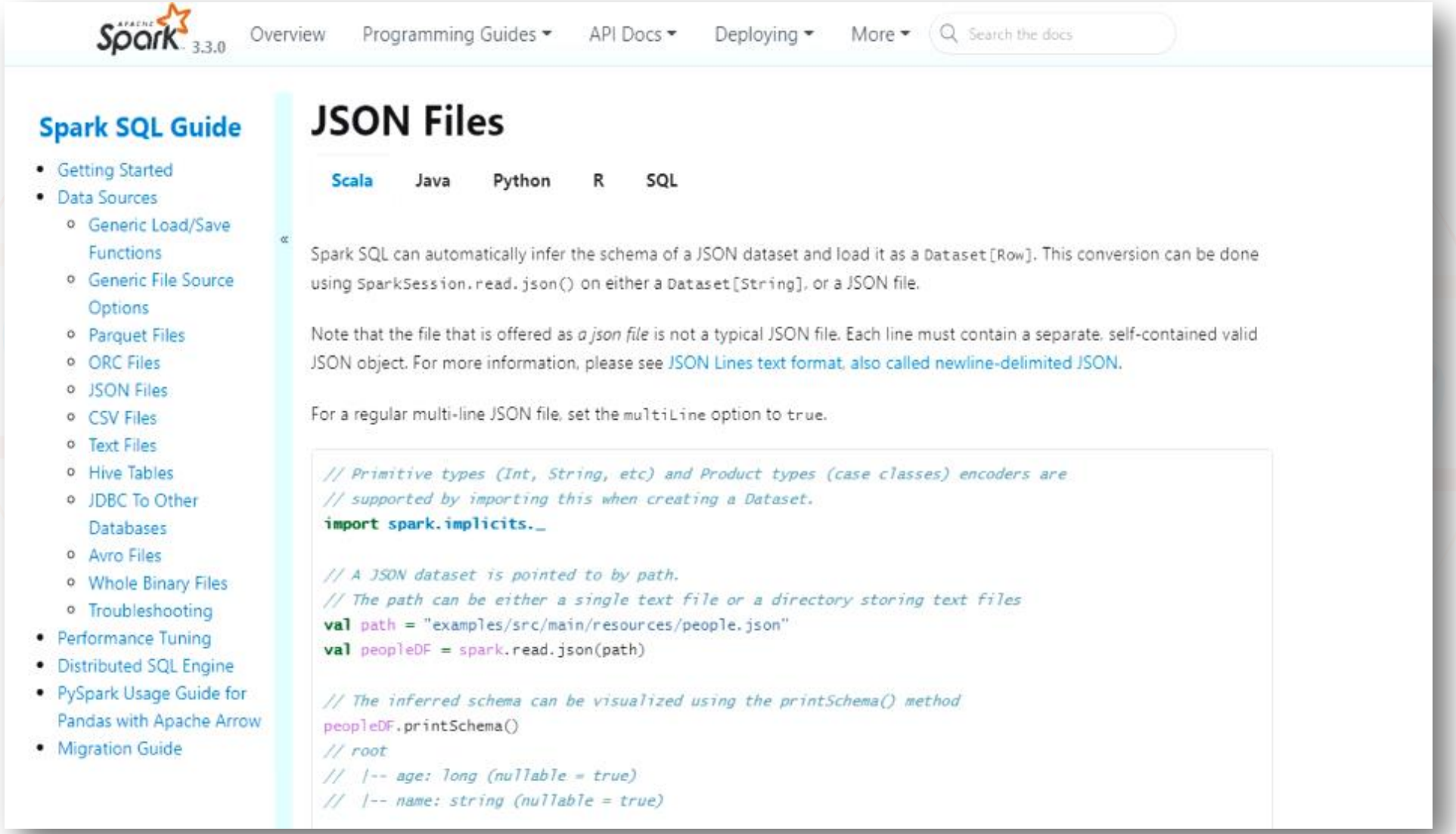
Property Name	Default	Meaning	Scope
sep	,	Sets a separator for each field and value. This separator can be one or more characters.	read/write
encoding	UTF-8	For reading, decodes the CSV files by the given encoding type. For writing, specifies encoding (charset) of saved CSV files. CSV built-in functions ignore this option.	read/write
quote	"	Sets a single character used for escaping quoted values where the separator can be part of the value. For reading, if you would like to turn off quotations, you need to set not null but an empty string. For writing, if an empty string is set, it uses u0000 (null character).	read/write
quoteAll	false	A flag indicating whether all values should always be enclosed in quotes. Default is to only escape values containing a quote character.	write
escape	\	Sets a single character used for escaping quotes inside an already quoted value.	read/write

We have many such options that make the CSV format reader super flexible. If you scroll down, you will see the header, inferSchema, dateFormat, etc. These are some options that we already used in our examples. However, if you are reading CSV or a delimited data file in your project and facing some data load problem or complexity, this is the place to come and look for suitable options to handle your requirement.

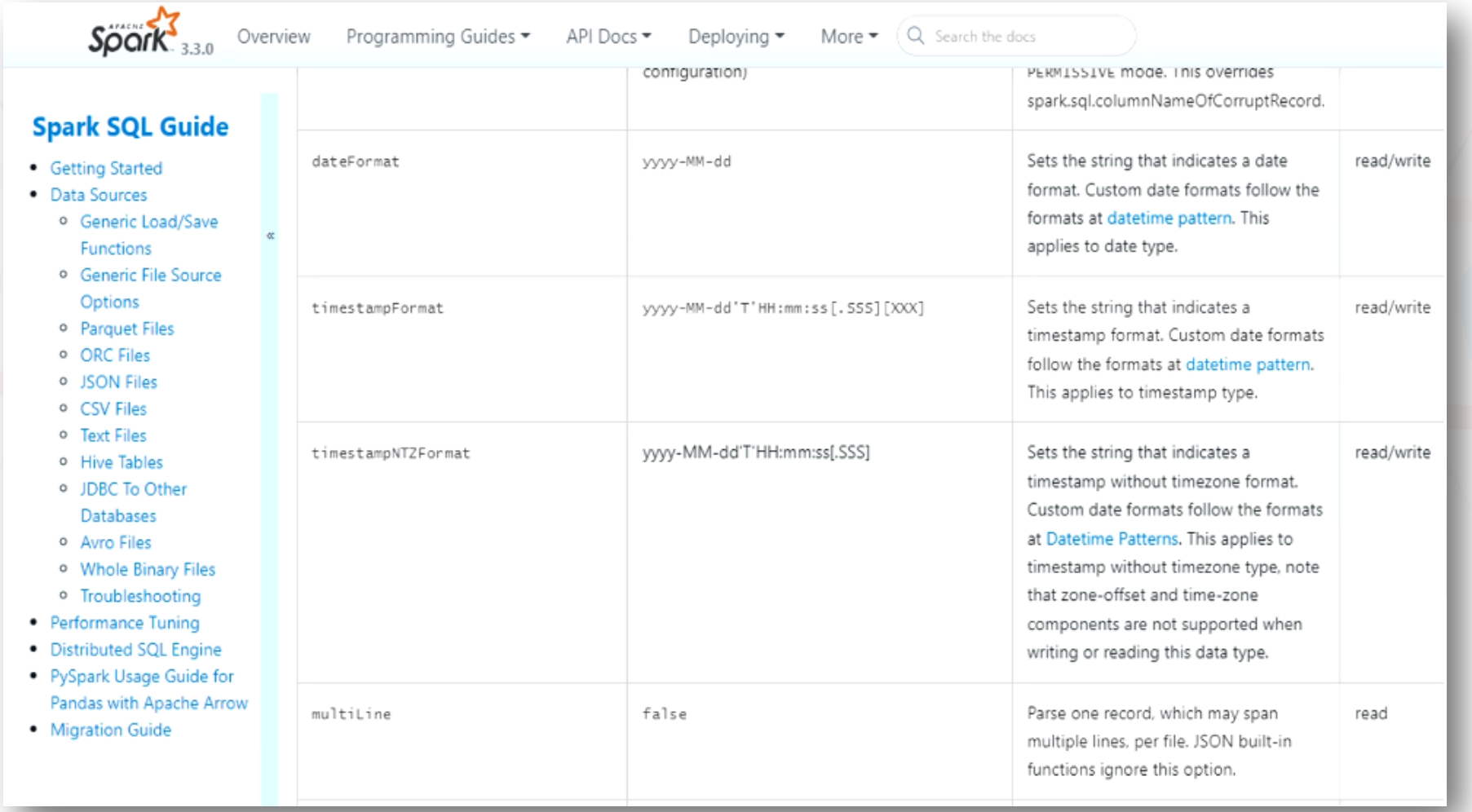


escapeQuotes	true	A flag indicating whether values containing quotes should always be enclosed in quotes. Default is to escape all values containing a quote character.	write
comment		Sets a single character used for skipping lines beginning with this character. By default, it is disabled.	read
header	false	For reading, uses the first line as names of columns. For writing, writes the names of columns as the first line. Note that if the given path is a RDD of Strings, this header option will remove all lines same with the header if exists. CSV built-in functions ignore this option.	read/write
inferSchema	false	Infers the input schema automatically from data. It requires one extra pass over the data. CSV built-in functions ignore this option.	read
enforceSchema	true	If it is set to `true`, the specified or inferred schema will be forcibly applied to datasource files, and headers in CSV files will be ignored. If the option is set to `false`, the schema will be validated against all headers in CSV files in the	read

Now let us see the JSON file. Scroll down, to see the list of options.



So you can see the timezone, which defaults to the spark.sql.session.timeZone. Similarly, you have dateFormat, timestamp format, and multiLine. A good JSON file comes with one JSON record on a single line. However, you may also get a JSON data file that comes with well-formatted JSON, and a single record spans multiple lines. You can read that file by setting the multiLine option.

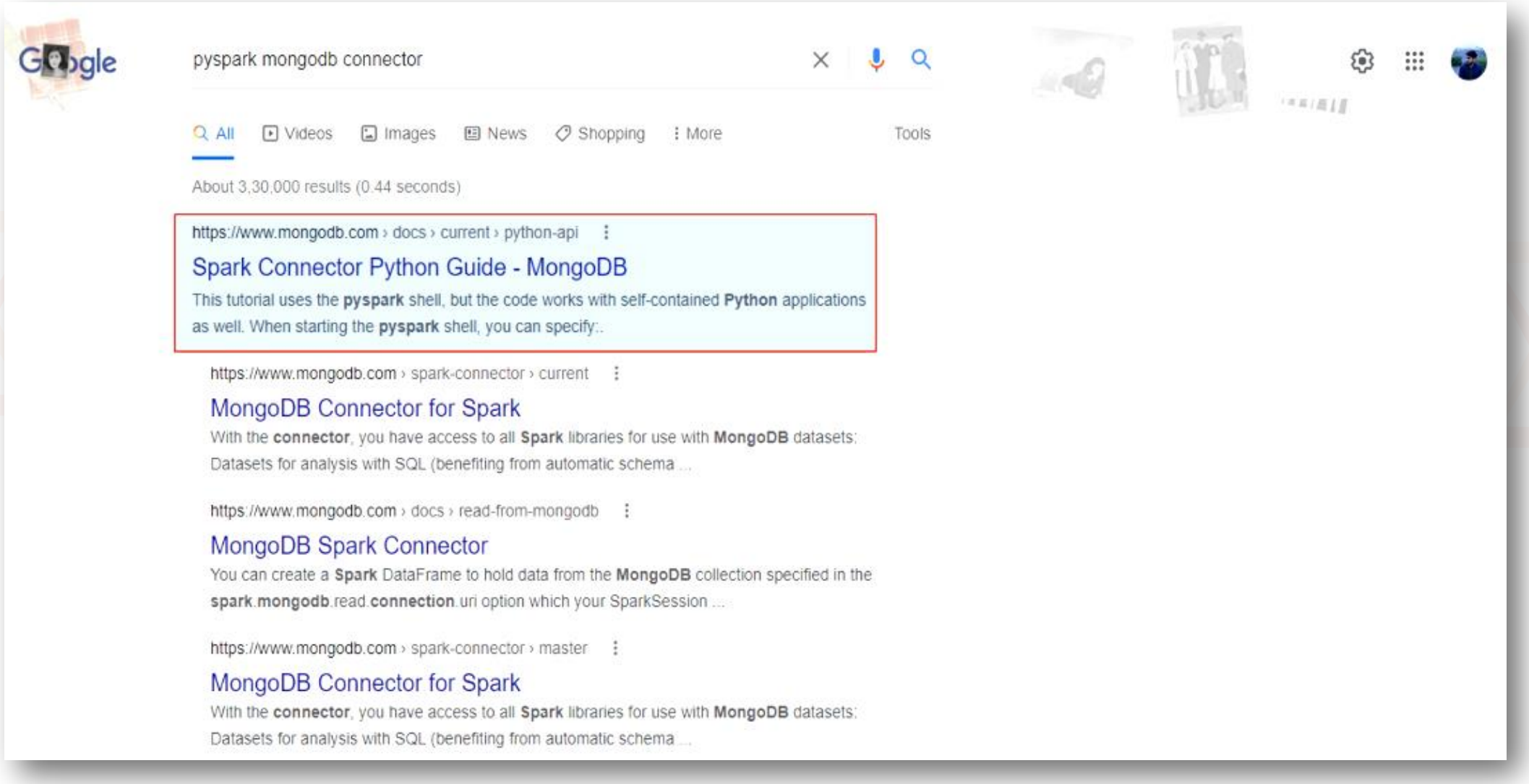


So now you know the place to look for the Spark Data source options.
However, this document lists options only for the Sark built-in data sources.

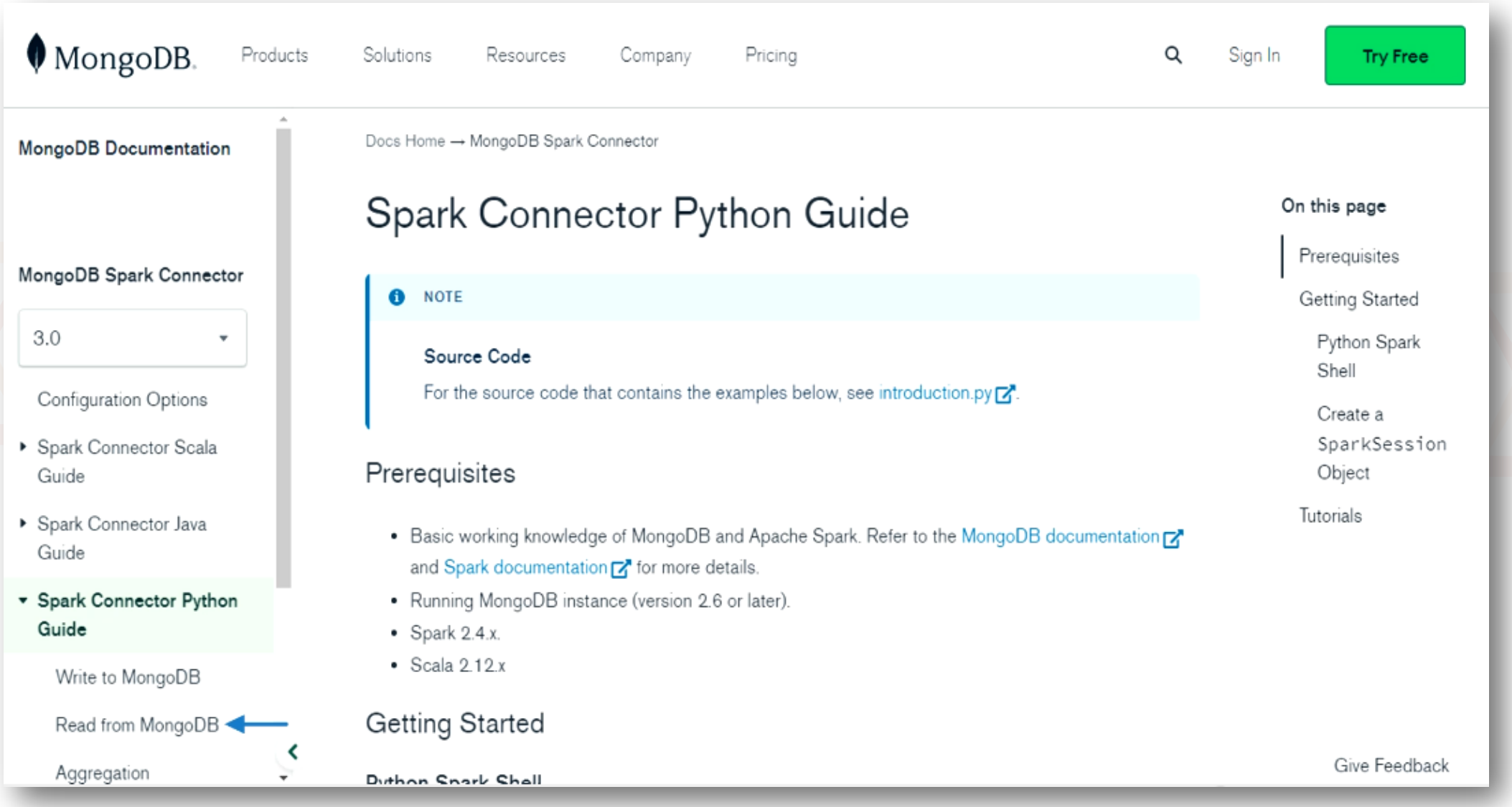
Spark also supports many external data sources; you can use the same DataFrameReader API to read data from external sources.
And the list of supported options will be available on their respective documentation.

For example, Spark supports MongoDB as a DataFrameReader source.
So let me search for the pyspark MongoDB connector, and hopefully, I will find something meaningful.

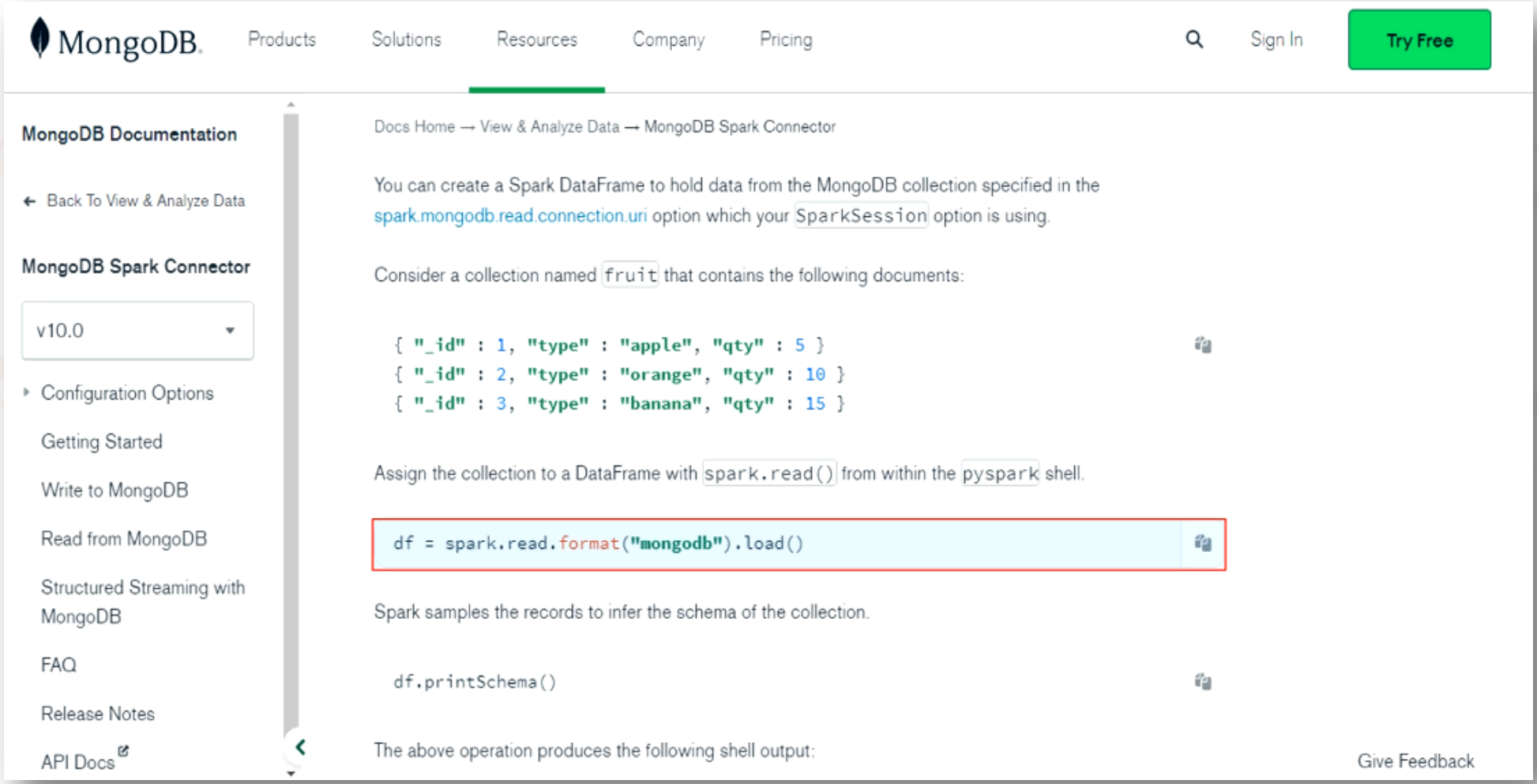
So I got a link to mongodb.com, which says Spark Connector Python Guide.
That's what I was looking for. Click the link, and you will see the documentation.



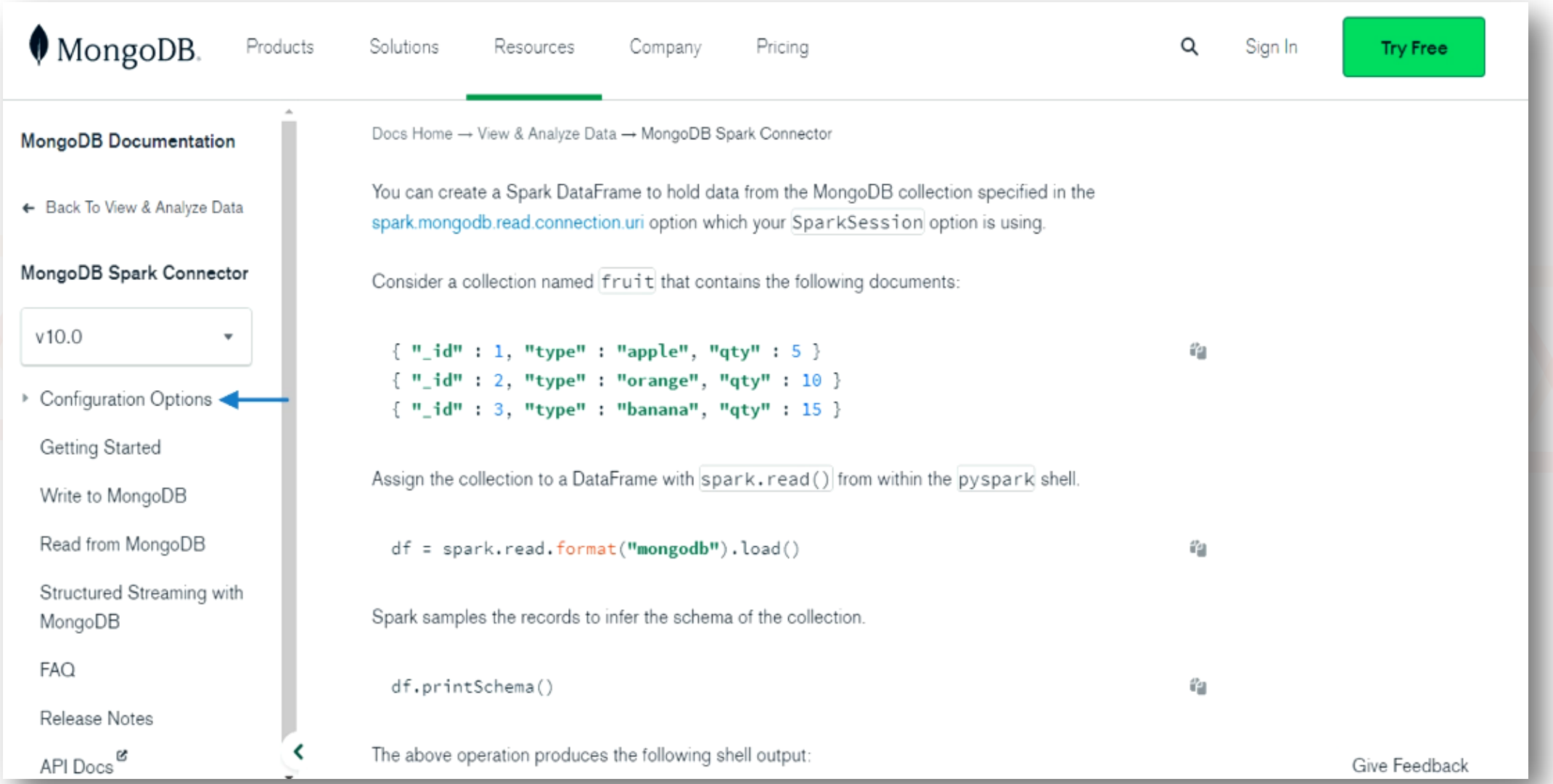
So I have a section here: Read from MongoDB
Let me check that out.



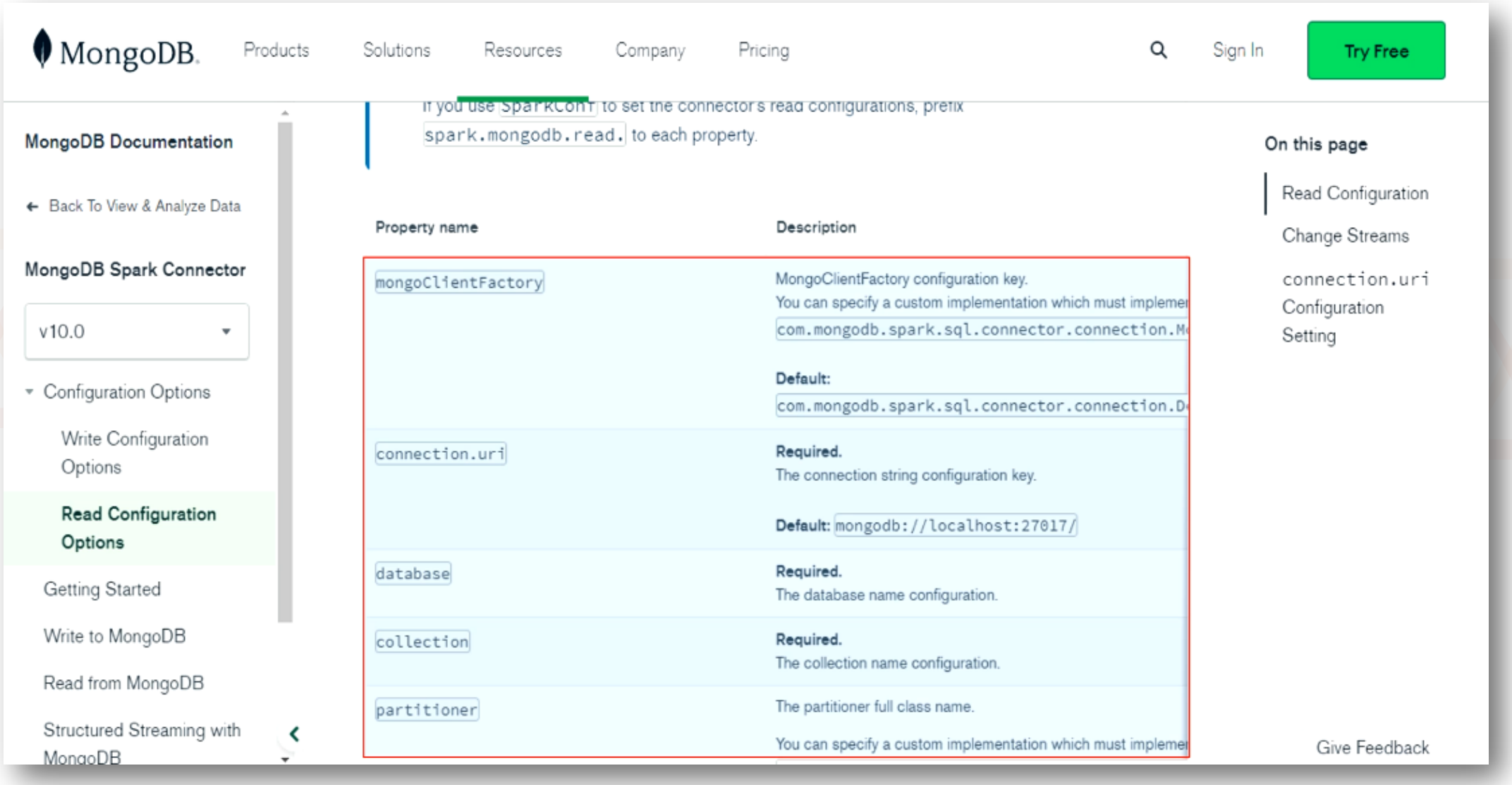
Look at this spark.read.format(). The spark.read gives you the DataFrameReader.
And you can read from MongoDB by setting the format to MongoDB. It is as simple as that.
But you may have to install and configure some packages and learn some MongoDB basics to work with MongoDB and Spark.



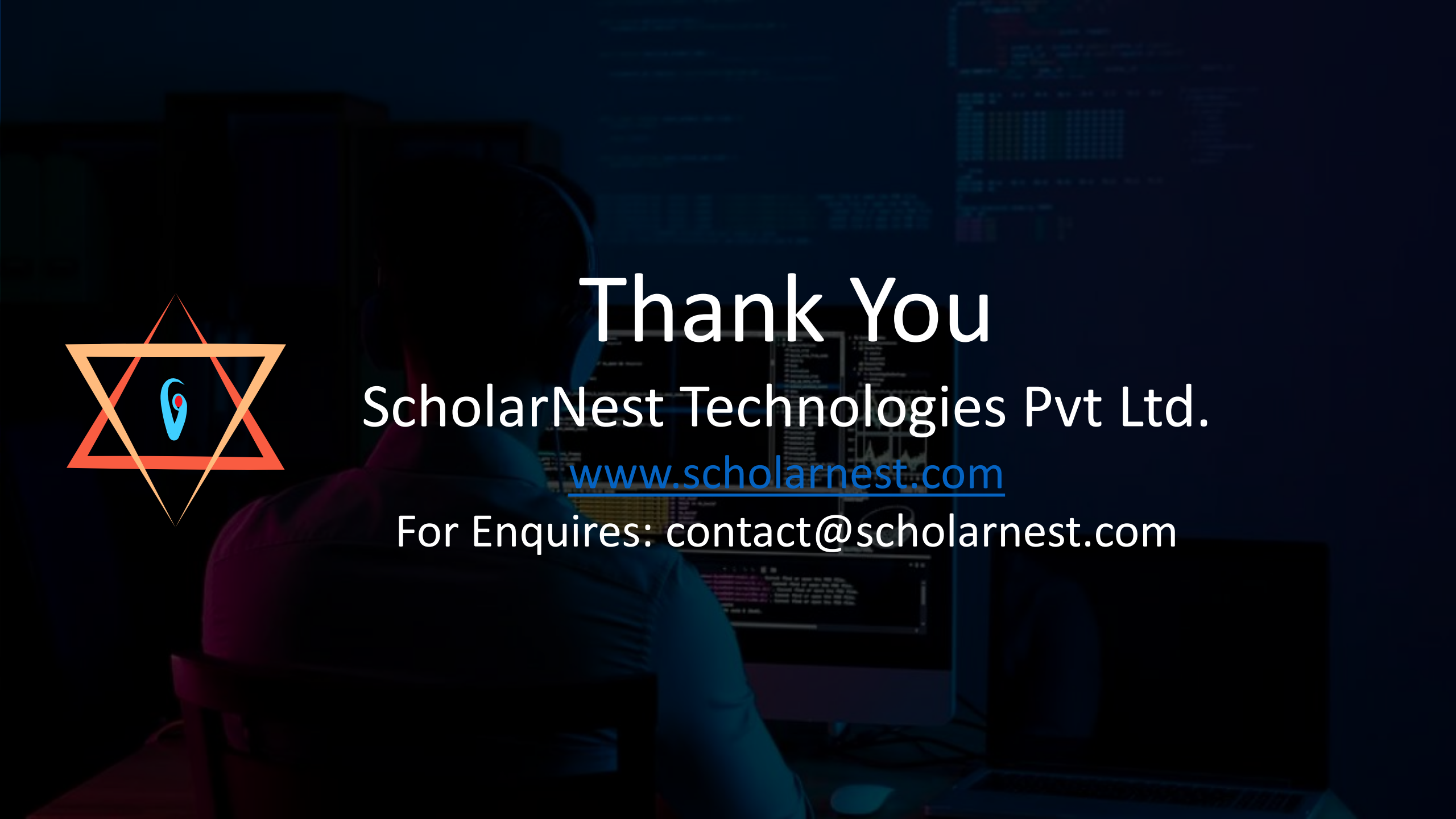
Do you see this link for the Configuration Options? Click that and go to read configuration options.



Here is the list of options. You can use these in the DataFrameReader options to configure the DataFrameReader for the MongoDB source.



Spark DataFrameReader and Spark DataframeWriter are standard across many sources and sinks.
You can use the same API to work with Spark's built-in sources and sinks.
And you can also use the same API to work with external sources and sinks.
However, for using external sources and sinks, you may have to install and configure some extra packages.
And we will learn that in the Advanced Spark course.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Data Sources & Sinks

Lecture:
Spark Dataframe Writer

Understanding Spark Dataframe Writer

Spark allows you to write data using DataFrameWriter API. The DataFrameWriter is a standardized API to work with various internal and external data sources. Here is the general structure of the DataFrameWriter API on the left side of the image shown below. And here is an example on the right side.

General structure

```
DataFrameWriter
    .format(...)
    .option(...)
    .partitionBy(...)
    .bucketBy(...)
    .sortBy(...)
    .save()
```

Indicative Example

```
DataFrame.write
        .format("parquet")
        .mode(saveMode)
        .option("path", "/data/flights/")
        .save()
```


So, you can get the DataFrameWriter using the write method on any of your Dataframe.

General structure

```
DataFrameWriter
    .format(...)
    .option(...)
    .partitionBy(...)
    .bucketBy(...)
    .sortBy(...)
    .save()
```

Indicative Example

```
DataFrame.write
         .format("parquet")
         .mode(saveMode)
         .option("path", "/data/flights/")
         .save()
```


Once you have the DataFrameWriter, you can specify four main things.

General structure

```
DataFrameWriter
    .format(...)
    .option(...)
    .partitionBy(...)
    .bucketBy(...)
    .sortBy(...)
    .save()
```

Indicative Example

```
DataFrame.write
    .format("parquet")
    .mode(saveMode)
    .option("path", "/data/flights/")
    .save()
```


The first and the most important thing is the output format.
Spark supports several internal file formats such as CSV, JSON, Parquet, AVRO, ORC, etc.
However, Parquet is the default file format for Apache Spark. So, DataFrameWriter assumes Parquet when you do not specify the format. These are the internal Spark file formats bundled with your Spark.
However, you also have a bunch of formats offered by the community developers and third-party vendors. Some of the popular ones are Cassandra, MongoDB, and Kafka.

General structure

```
DataFrameWriter
    .format(...)
    .option(...)
    .partitionBy(...)
    .bucketBy(...)
    .sortBy(...)
    .save()
```

Indicative Example

```
DataFrame.write
    .format("parquet")
    .mode(saveMode)
    .option("path", "/data/flights/")
    .save()
```


The next thing is to specify the options.
Every data sink has a specific set of options to determine how the DataFrameWriter will write the data.
However, at a minimum, you must specify the target of your DataFrame.
For a file-based data sink like Parquet, it should be a directory location.

General structure

```
DataFrameWriter
    .format(...)
    .option(...)
    .partitionBy(...)
    .bucketBy(...)
    .sortBy(...)
    .save()
```

Indicative Example

```
DataFrame.write
         .format("parquet")
         .mode(saveMode)
         .option("path", "/data/flights/")
         .save()
```


The next one is the save mode.
Save modes specify the behaviour when Spark finds existing data at the specified location.

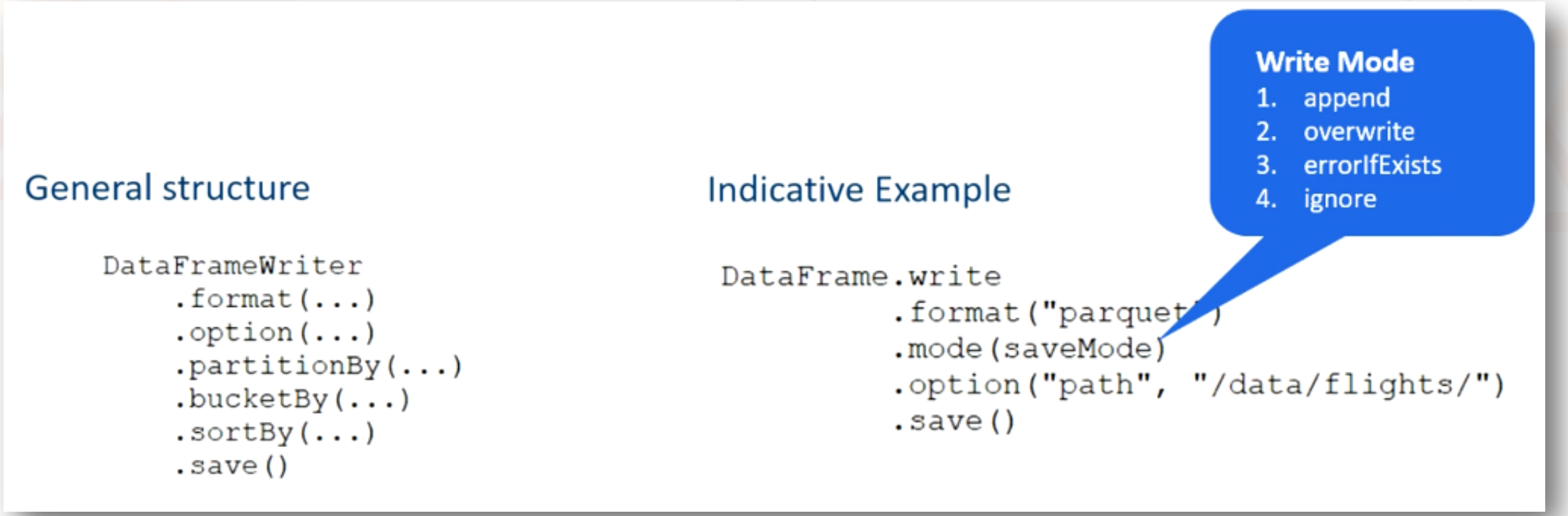
General structure

```
DataFrameWriter
    .format(...)
    .option(...)
    .partitionBy(...)
    .bucketBy(...)
    .sortBy(...)
    .save()
```

Indicative Example

```
DataFrame.write
         .format("parquet")
         .mode(saveMode)
         .option("path", "/data/flights/")
         .save()
```


We have four valid modes.
The append mode will create new files without touching the existing data at the given location.
The overwrite will remove the existing data files and create new files.
The "errorIfExists" will throw an error when you already have some data at the specified location.
The ignore will write the data if and only if the target directory is empty and do nothing if some files already exist at the location.



Finally, the most important thing is to control the layout of your output data.
What does it mean? It means to control the following items:

1. Organizing your output in partitions and buckets
2. Storing sorted data

General structure

```
DataFrameWriter
    .format(...)
    .option(...)
    .partitionBy(...)
    .bucketBy(...)
    .sortBy(...)
    .save()
```

Data Layout
1. partitions
2. Buckets

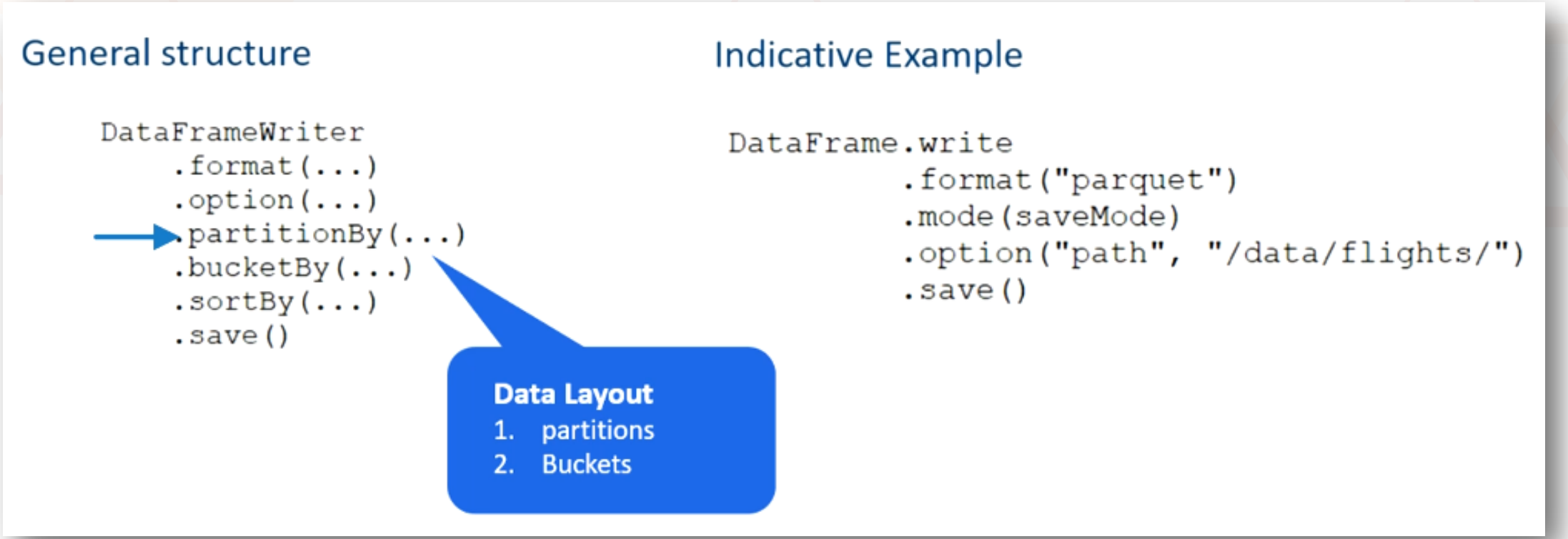
Indicative Example

```
DataFrame.write
        .format("parquet")
        .mode(saveMode)
        .option("path", "/data/flights/")
        .save()
```


We create Spark Dataframe reading one or more files.
Similarly, when you write a Dataframe, you can write it as one or more files.

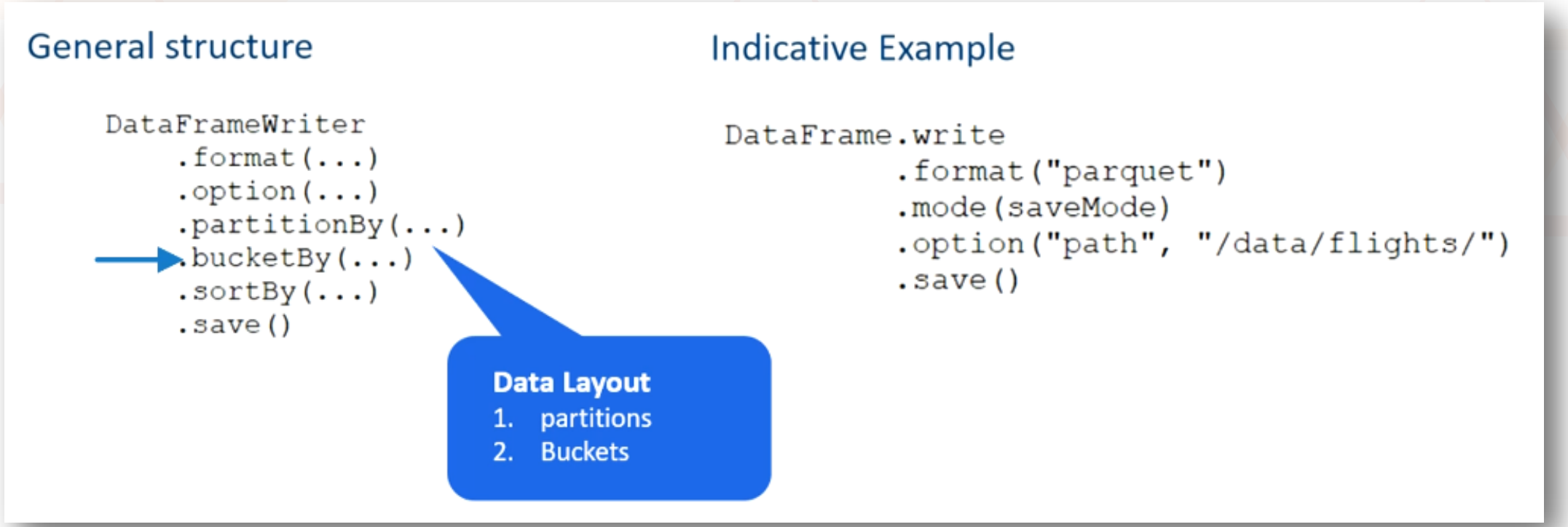
So the DataFrameWriter gives us options to break out and organize your data into logical partitions and buckets.
Partitioning or bucketing are powerful techniques to organize your tables and data files logically and speed up the query performance.

We have two options.
The first option is to use a partitionBy() method. The partitionBy() method will partition your data using a key column. You can use a single column key such as country code or a composite column key such as country code plus state code.
Key-based partitioning is a powerful method to break your data logically.
It also helps improve your Spark SQL performance using the partition pruning technique.



The second option is partitioning your data into a fixed number of predefined buckets.
And this is known as bucketing.

You can use the bucketBy() method.
However, the bucketBy() is only available on Spark-managed tables.

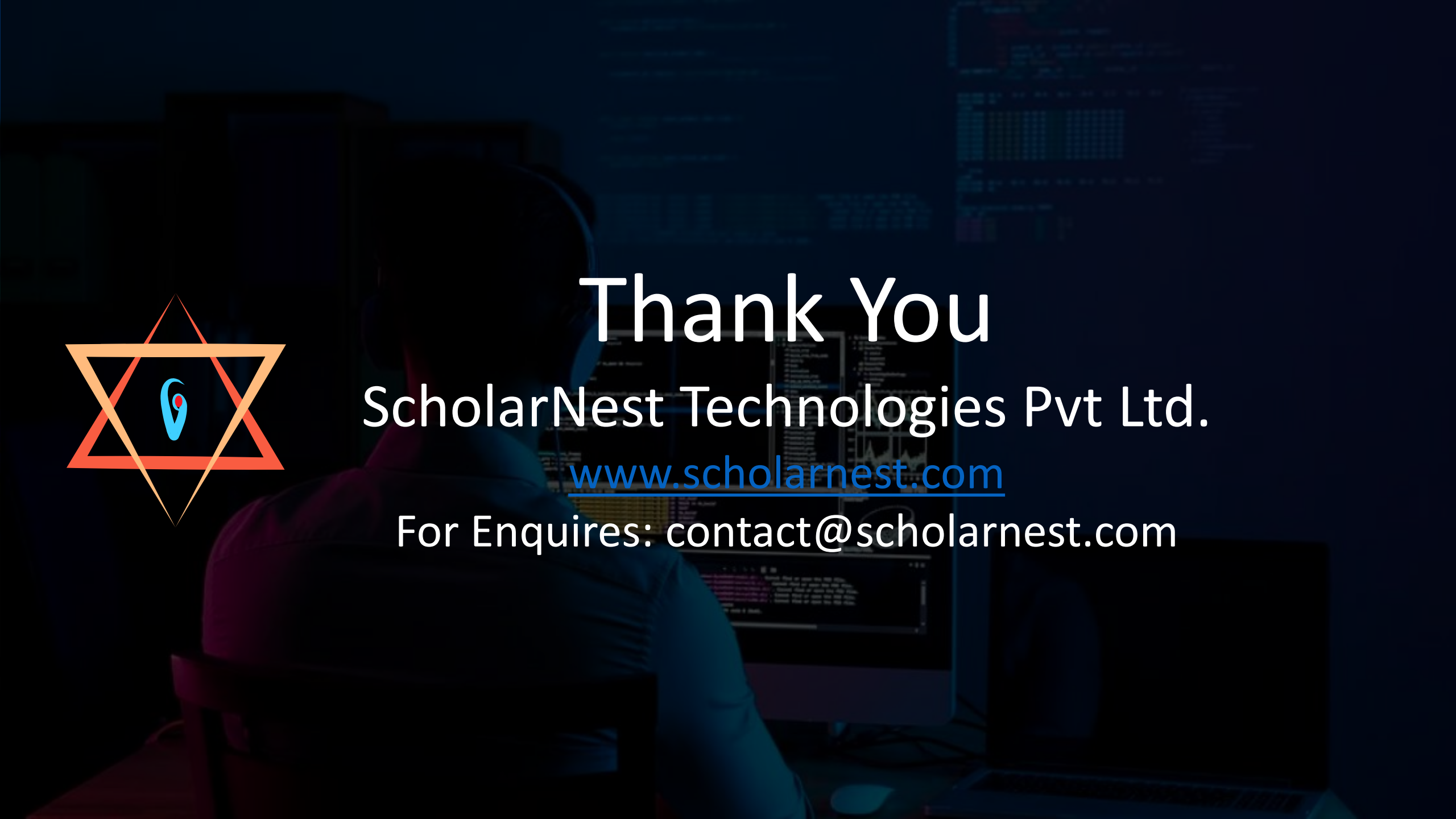


As of now, you know that we have two logical partitioning options.

1. partitionBy()
2. bucketBy()

We have one more suitable option to learn hereand that is sortBy().

The sortBy() method is commonly used with the bucketBy().
This option allows you to create sorted buckets.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

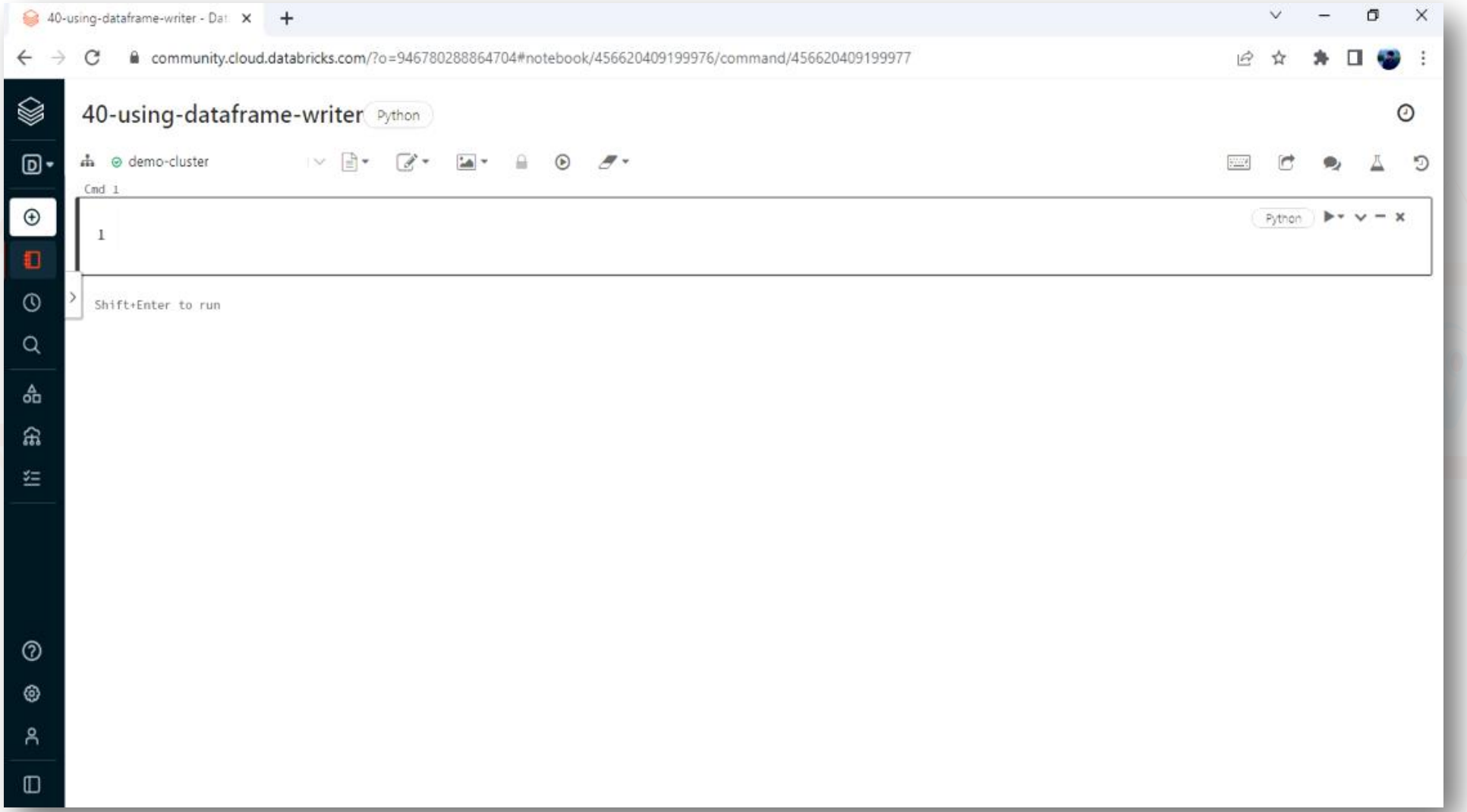
Databricks Spark Certification and Beyond

Module:
Data Sources & Sinks

Lecture:
Using Dataframe Writer

Writing CSV JSON Parquet and AVRO Files

Go to your Databricks workspace and create a new notebook. **(Reference: 40-using-dataframe-writer.ipynb)**



For writing a Dataframe, we need to have a Dataframe. So I have created a Dataframe below.
So I am reading JSON data and creating a Dataframe.
Now, what do I want to do? I will not do any data processing in this example.
We will simply write this Dataframe using the DataFrameWriter API and learn the mechanics.
However, in a real scenario, you will be processing your source data, and then you are going to write the output to a sink.

Cmd 1

```
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 STRING, DISTANCE INT"""
4
5  flight_time_json_df = spark.read \
6                            .format("json") \
7                            .schema(flight_schema_ddl) \
8                            .option("mode", "FAILFAST") \
9                            .option("dateFormat", "M/d/y") \
10                           .load("/FileStore/tables/dataframeReader/flight*.json")
```

Command took 0.78 seconds -- by prashant@scholarnest.com at 7/12/2022, 8:53:05 PM on demo-cluster

Now we want to write the Dataframe we created as an avro output.
So, we start with the Dataframe and use the write() method to get the DataFrameWriter. Then we set the format. And then the save mode. I am going to use the overwrite mode.
The next most important thing is to set the output directory path.
Finally, we call the save() method.

Cmd 2

```
1  flight_time_json_df.write \
2                     .format("avro") \
3                     .mode("overwrite") \
4                     .option("path", "/FileStore/tables/temp/") \
5                     .save()
```


What do you expect when I execute it?
This code will create one or more Avro files in the given directory. However, before writing to the directory location, this API call will also clean up the target directory because we are working in overwrite mode. The save() method in an overwrite mode is a two-step process. Clean the directory and write the new files. So, be careful while using the overwrite mode.

Cmd 2

```
1  flight_time_json_df.write \
2                     .format("avro") \
3                     .mode("overwrite") \
4                     .option("path", "/FileStore/tables/temp/") \
5                     .save()
```


I have one more question. How many Avro files do you expect? One, two, or ten?
Well, Spark is a distributed processing engine, and your Dataframe is a distributed data structure.
Every Dataframe is internally broken into one or more partitions.
Why? Because Spark wants to process it in parallel.
So it should partition the Dataframe and process each partition in parallel.
In the next section, we will learn about the Dataframe partitions and parallel processing, where I will talk about the Spark internals.
But for now, you can think of a Spark Dataframe made up of multiple partitions.
So when you write your Dataframe, each partition is written as a single file.
So, I am writing the Dataframe as Avro files. How many Avro files do you expect? One, two, or ten?
It depends upon the number of Dataframe partitions.
So, if this Dataframe has five partitions, you can expect five output files, each of which will be written in parallel by different executers.

So to answer our previous question, I must know how many partitions we have in this Dataframe. So let me add a new cell above the current cell and check the number of partitions, as shown below.
We convert the Dataframe to an RDD and get the number of partitions.
So now we know the number of partitions in this Dataframe and how many files I expect.

Here is my output directory.
You can see two types of files here.
The files starting with an underscore are hidden control files.

Cmd 4

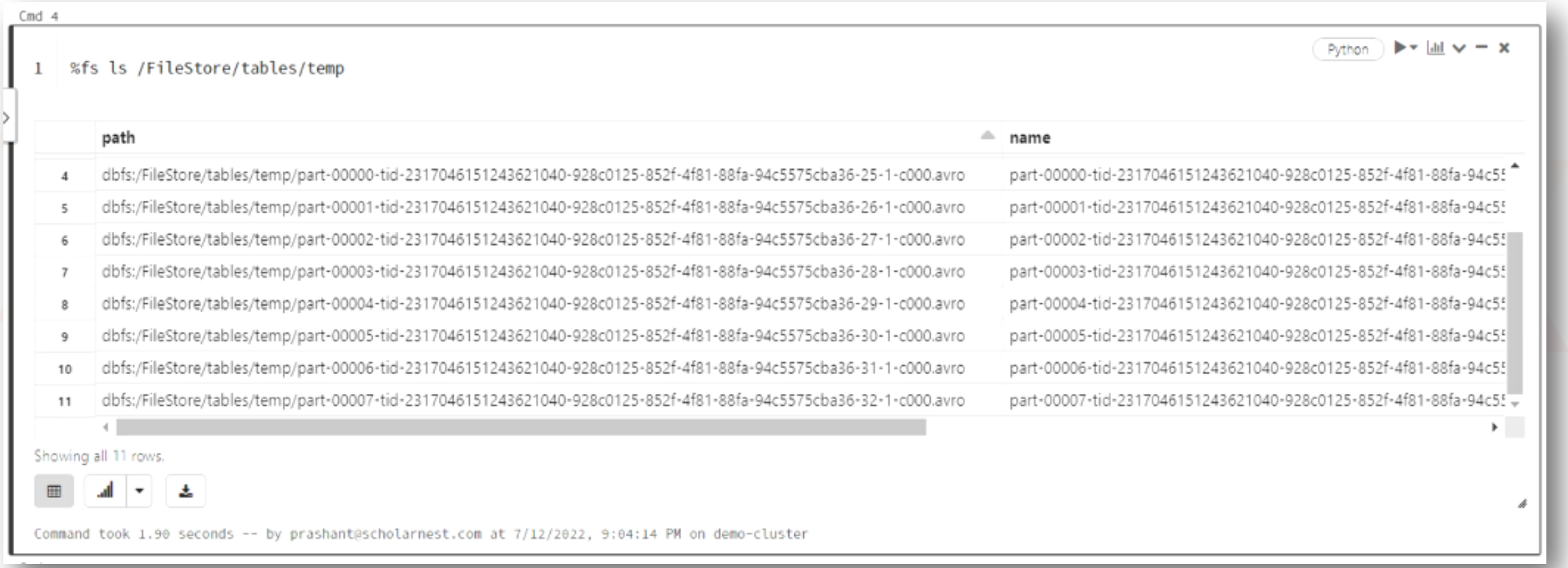
```
1  %fs ls /FileStore/tables/temp
```

	path	name
1	dbfs:/FileStore/tables/temp/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/temp/_committed_2317046151243621040	_committed_2317046151243621040
3	dbfs:/FileStore/tables/temp/_started_2317046151243621040	_started_2317046151243621040
4	dbfs:/FileStore/tables/temp/part-00000-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575cba36-25-1-c000.avro	part-00000-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575
5	dbfs:/FileStore/tables/temp/part-00001-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575cba36-26-1-c000.avro	part-00001-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575
6	dbfs:/FileStore/tables/temp/part-00002-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575cba36-27-1-c000.avro	part-00002-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575
7	dbfs:/FileStore/tables/temp/part-00003-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575cba36-28-1-c000.avro	part-00003-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575
8	dbfs:/FileStore/tables/temp/part-00004-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575cba36-29-1-c000.avro	part-00004-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575
9	dbfs:/FileStore/tables/temp/part-00005-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575cba36-30-1-c000.avro	part-00005-tid-2317046151243621040-928c0125-852f-4f81-88fa-94c5575

Showing all 11 rows.

Command took 1.90 seconds -- by prashant@scholarnest.com at 7/12/2022, 9:04:14 PM on demo-cluster

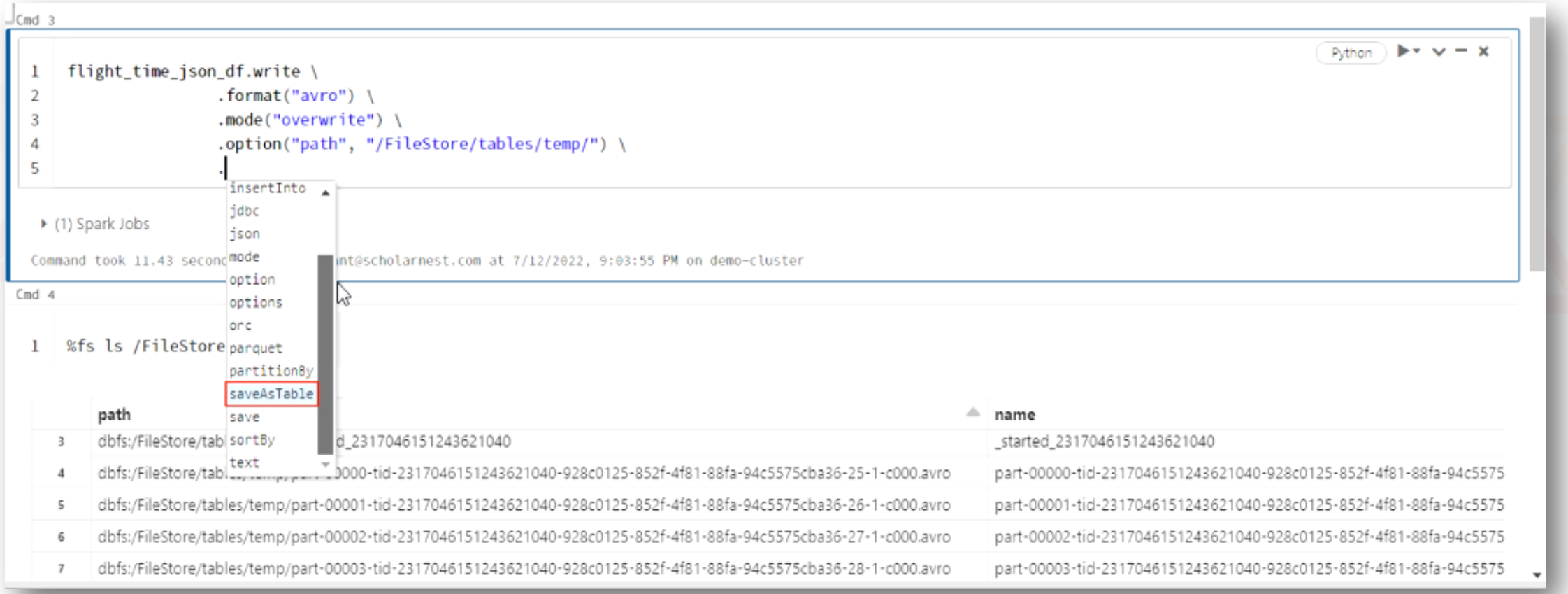
You can also see some AVRO data files as well in the output directory.
How many are there? They are the same as the number of partitions. And that's what we expected.



So you learned how to use DataFrameWriter API to write your dataframe to an output sink.
I showed this example to write your data to the AVRO file sink.
However, you can use the same approach to write data to various sinks.

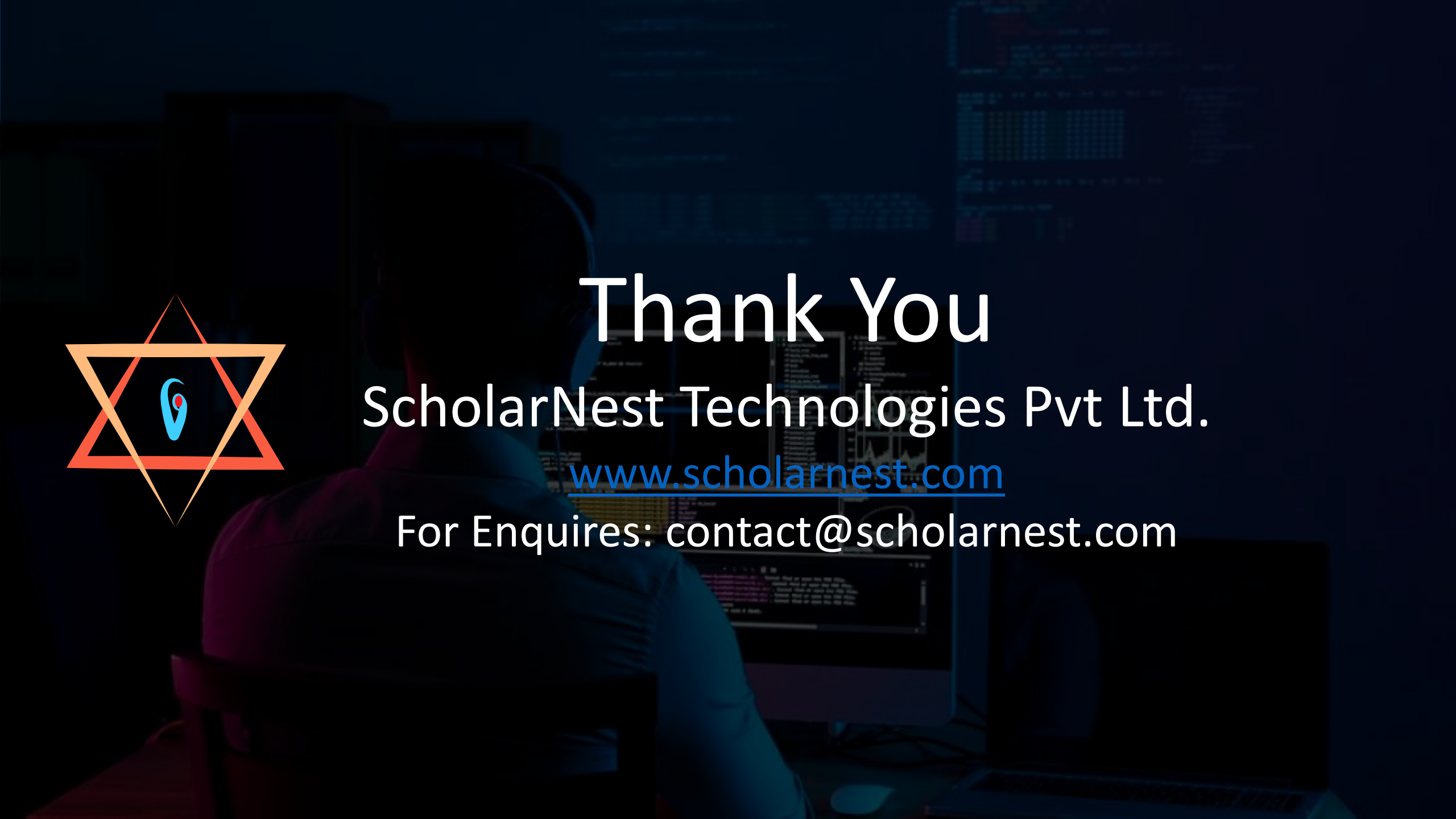
You can write it to Spark's built-in sinks such as CSV, JSON, and Parquet.
You can also write it to external data sinks such as Cassandra or MongoDB.
We will see some examples of working with external data sources in the advanced spark course.

You can go back to the cell where we have the code for writing the Dataframe. So, we are using the save() method in the end. Delete the save() method but leave the dot symbol. Now press the tab, and you will see the available methods. Choose the saveAsTable() method from the list.



The save() method will save your Dataframe to a sink. The format determines the type of sink. But the saveAsTable() method will save your Dataframe to a Spark database table.

The DataFrameWriter is the standard approach for saving your Dataframe.
You can use the same DataFrameWriter API to save your data frames to various sinks.
And that includes Spark built-in sinks, Spark Tables, and external data sinks.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Data Sources & Sinks

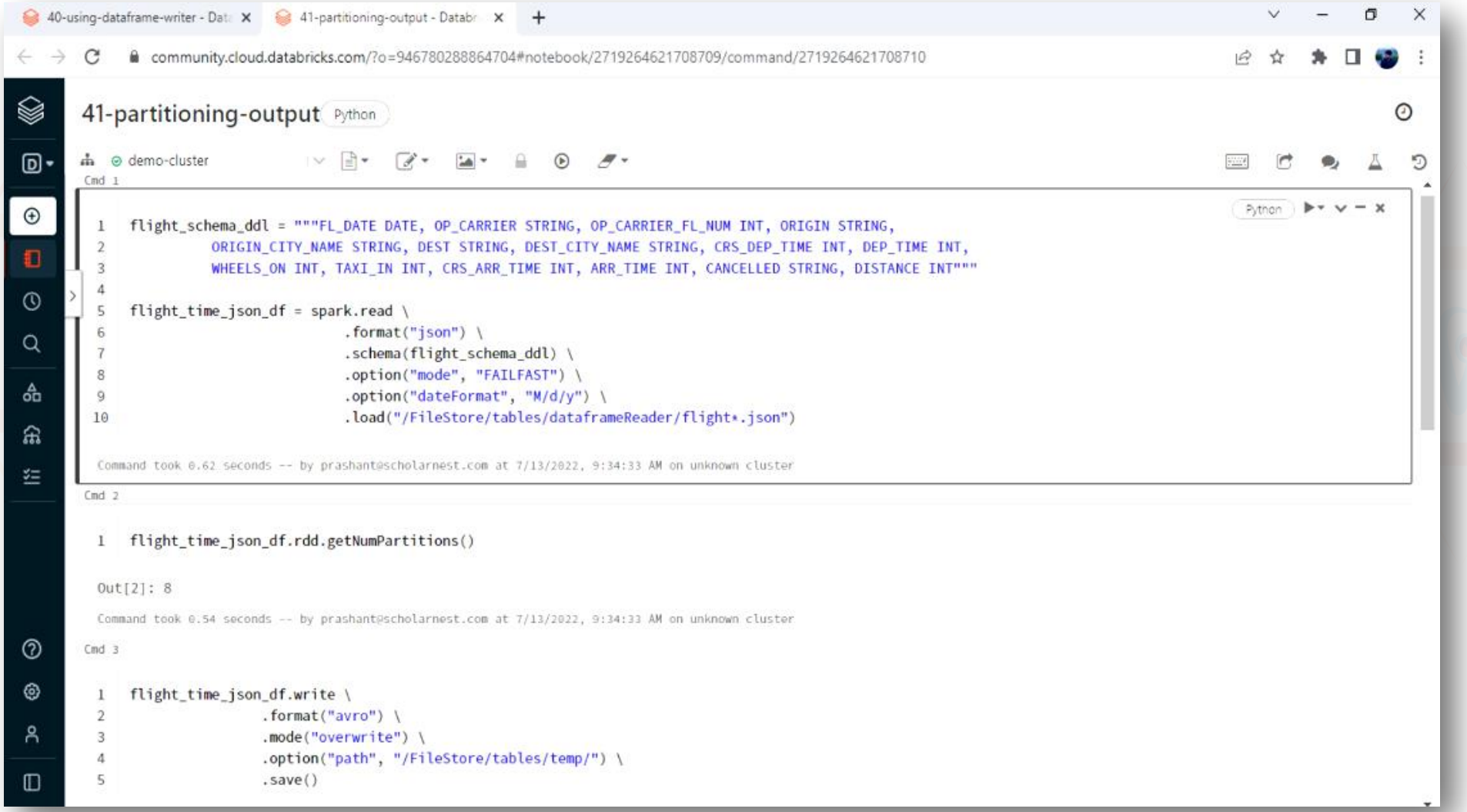
Lecture:
Using Dataframe Writer

Writing CSV JSON Parquet and AVRO Files

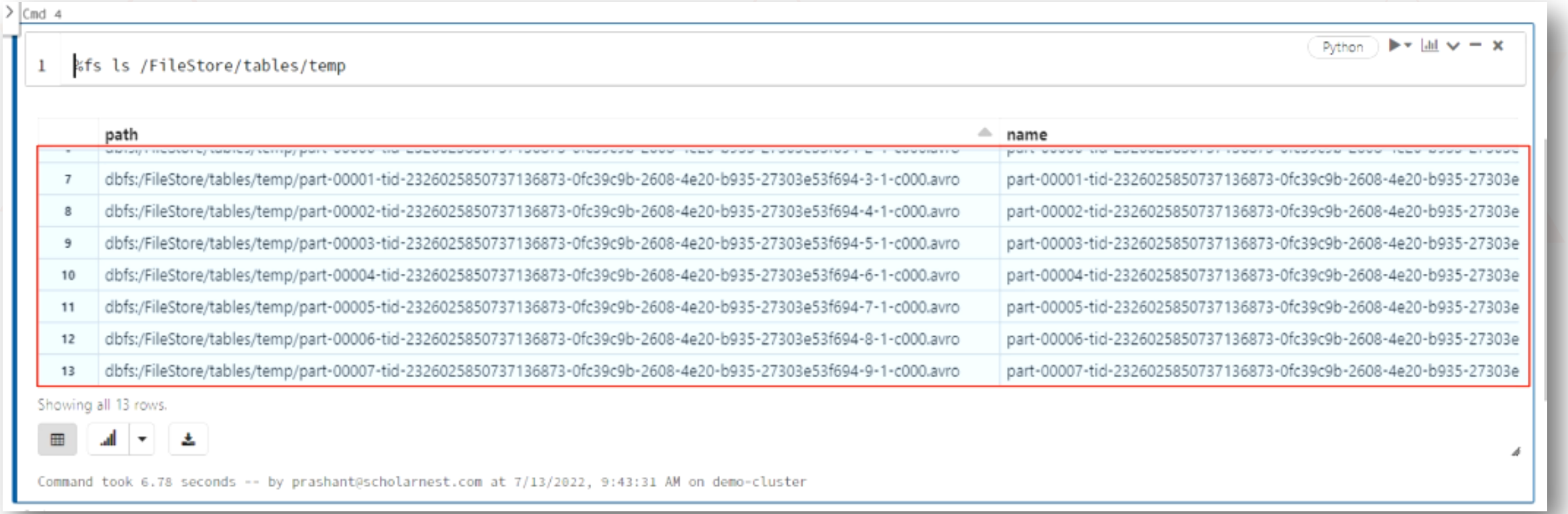
Go to your Databricks workspace and clone the earlier notebook.

(Reference: 41-partitioning-output.ipynb)

Also, give a new name to your clone



Run the last cell to check your output directory.
We have eight Avro files, and you can see them highlighted in the image below. I also explained that the Dataframe is internally partitioned so Spark can process the Dataframe partitions in parallel. So when you write the data frame, it creates one file per partition.
In our example, we got eight files because we had eight partitions.



But in most cases, you do not want to dump your data in a single directory as a bunch of files, as shown in this example. I dumped my Dataframe as a set of eight files in a single directory.

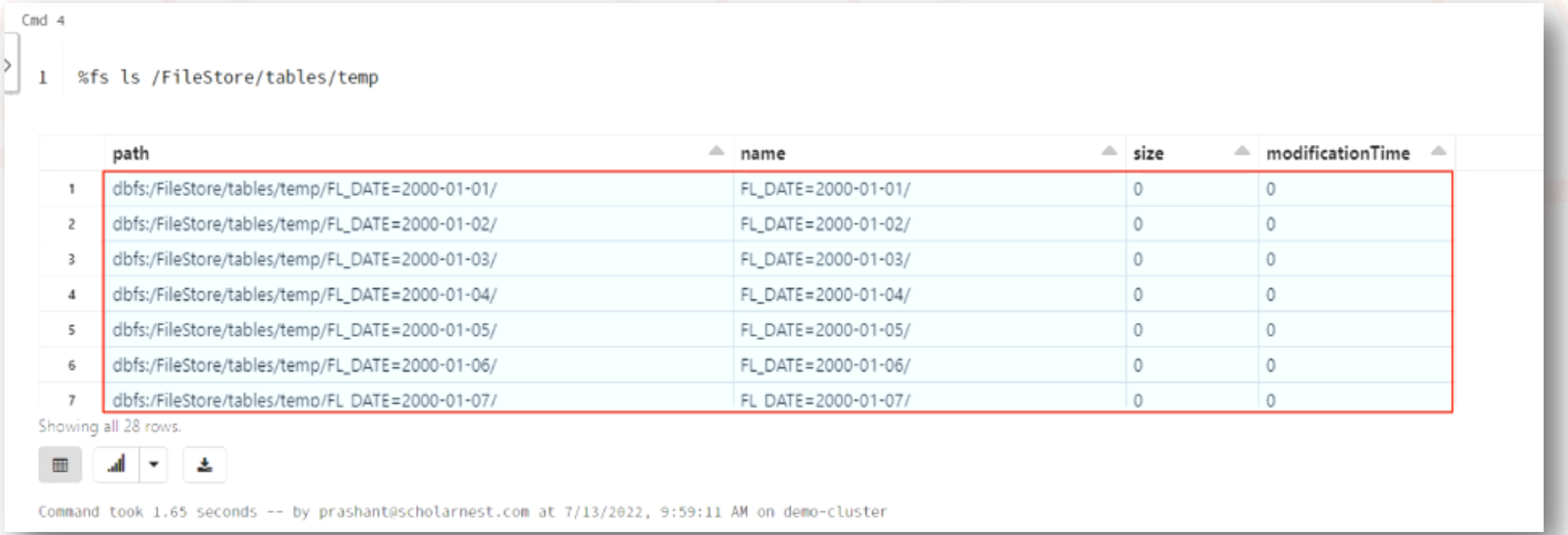
Sooner or later, I will need to read this data back and run queries on the same.
And those queries are not going to be efficient. They will not perform well.
Why? Because I haven't organized my data according to my requirement and queries.
The current directory looks like a dumping place.
How will Spark efficiently search data in this directory?

So the best practice is to design your data layout according to your future queries.
For example, I can organize the flight time dataset using the flight date.

And that is where DataFrameWriter gives you the partitionBy() method.
You can use the partitionBy() method to organize and structure your data in a subdirectory structure.

I added a partitionBy() method in our previous code, and I want to partition the output into two columns. The first column is the flight date, and the second column is the flight carrier. Why am I partitioning my data on these two columns? Because I know I will be querying this data using the flight date. So It makes good sense to partition my data on the flight date. Some of my queries will also use flight carriers along with the flight date. So I will add a second level of partitioning using the OP_CAREER column.

Now if you check the output directory, you can see that we have directories for each date. I partitioned my output using the FL_DATE column, and you can see the directory names here. DataFrameWriter created so many directories. The directory name includes the column name and the column value. The first directory name is FL_DATE=2000-01-01, which means that this directory contains data only where FL_DATE == 2000-01-01. Similarly, the second directory name is FL_DATE=2000-01-02. So the second directory will contain records where FL_DATE == 2000-01-02.



Now think about a SQL query like this:

SELECT * FROM fire_tbl WHERE FL_DATE == 2000-01-02

We want to see all the data where FL_DATE == 2000-01-02.
To answer this query, Spark can easily look into the second directory and return all the data stored inside this directory.
We do not even need to look into the other directories. Why? Because we know we already organized our data into date-wise directories. We don't need to look into other directories.

Structuring and organizing your data into appropriate partitions can give a massive boost to your queries.
That's why the partitionBy() method of the DataFrameWriter is super important.

You can go inside any of these date-wise directories.
Inside the date, we have directories for each OP_CARRIER.
Why? Because we made OP_CARRIER the second level of the partition.
So each date is further partitioned into OP_CARRIER.
And you will see directories like OP_CARRIER=AA and OP_CARRIER=AS.

Cmd 5

```
1  %fs ls /FileStore/tables/temp/FL_DATE=2000-01-02/
```

	path	name	size	modificationTime
1	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=AA/	OP_CARRIER=AA/	0	0
2	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=AS/	OP_CARRIER=AS/	0	0
3	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=CO/	OP_CARRIER=CO/	0	0
4	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=DL/	OP_CARRIER=DL/	0	0
5	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=HP/	OP_CARRIER=HP/	0	0
6	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=NW/	OP_CARRIER=NW/	0	0
7	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=TW/	OP_CARRIER=TW/	0	0

Showing all 10 rows.

Command took 0.75 seconds -- by prashant@scholarnest.com at 7/13/2022, 10:00:59 AM on demo-cluster

Now the thing about the following query.

SELECT * FROM fire_tbl WHERE FL_DATE == 2000-01-02 AND OP_CARRIER==AS

Answering this query is again super quick.
Spark will look into the FL_DATE=2000-01-02/OP_CARRIER=AS directory and give you everything.
So this structure will benefit two types of SQLs:

1. WHERE FL_DATE == 2000-01-02
2. WHERE FL_DATE == 2000-01-02 AND OP_CARRIER==AS

If your SQL has a where clause on the FL_DATE, Spark will read one FL_DATE directory and all its subdirectories to give you the result.
But if you have a filter on FL_DATE and OP_CARRIER both, then Spark will read one FL_DATE directory and only one OP_CARRIER subdirectory to give you the result.

Now if I look inside one of the OP_CARRIER directory as shown below.
You will see that I have some hidden control files and one data file.
And this file contains data only for the FL_DATE=2000-01-02 and OP_CARRIER=AS.

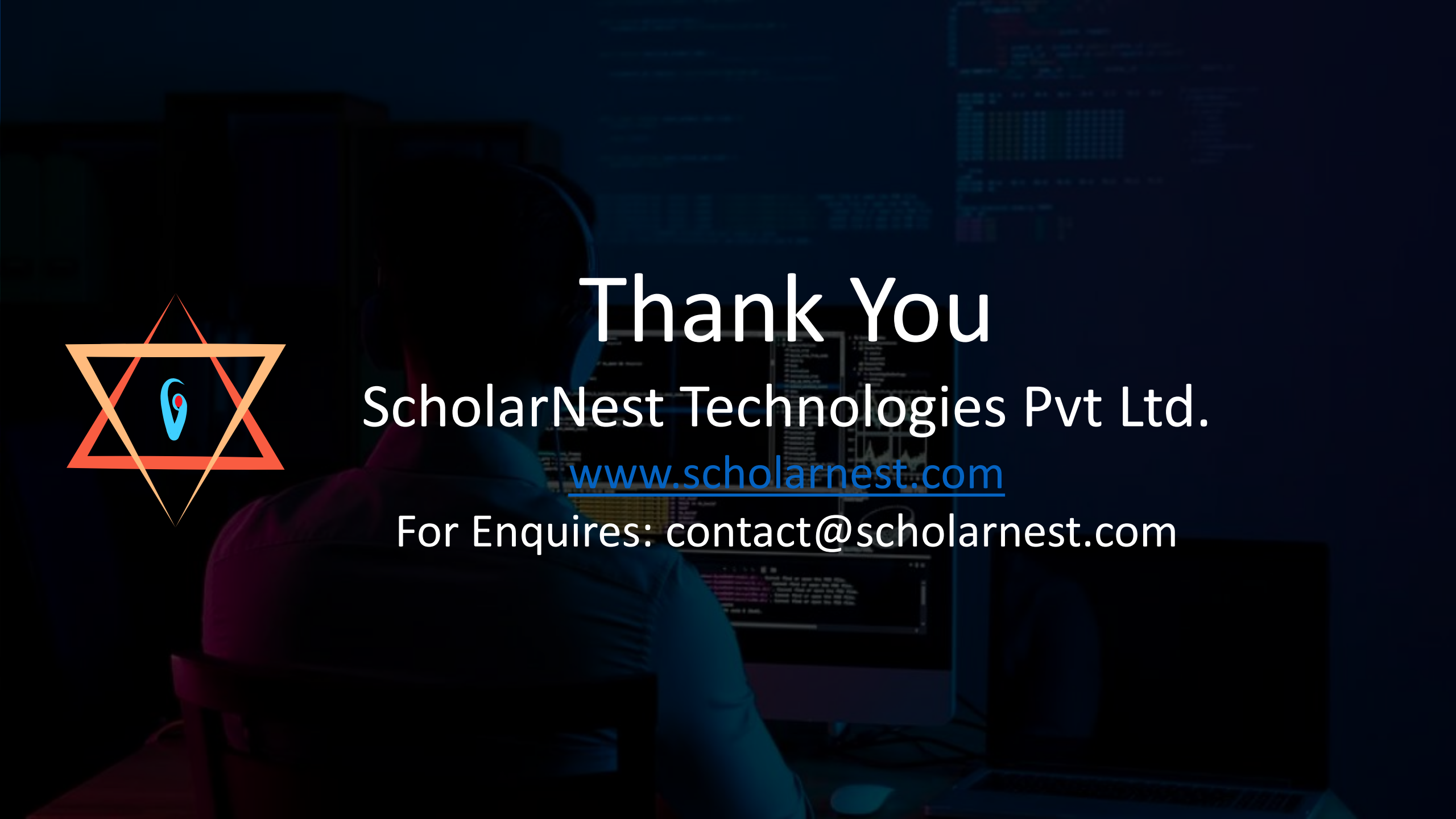
Cmd 6

```
1  %fs ls /FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=AS/
```

	path	name
1	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=AS/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=AS/_committed_2223203446639625875	_committed_2223203446639625875
3	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=AS/_started_2223203446639625875	_started_2223203446639625875
4	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/OP_CARRIER=AS/part-00000-tid-2223203446639625875-ade5fec6-de3f-4273-9776-2709e6ae28f5-18-12.c000.avro	part-00000-tid-2223203446639625875-ade5fec6-de3f-4273-9776-2709e6a

Showing all 4 rows.

Command took 0.91 seconds -- by prashant@scholarnest.com at 7/13/2022, 10:02:09 AM on demo-cluster



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

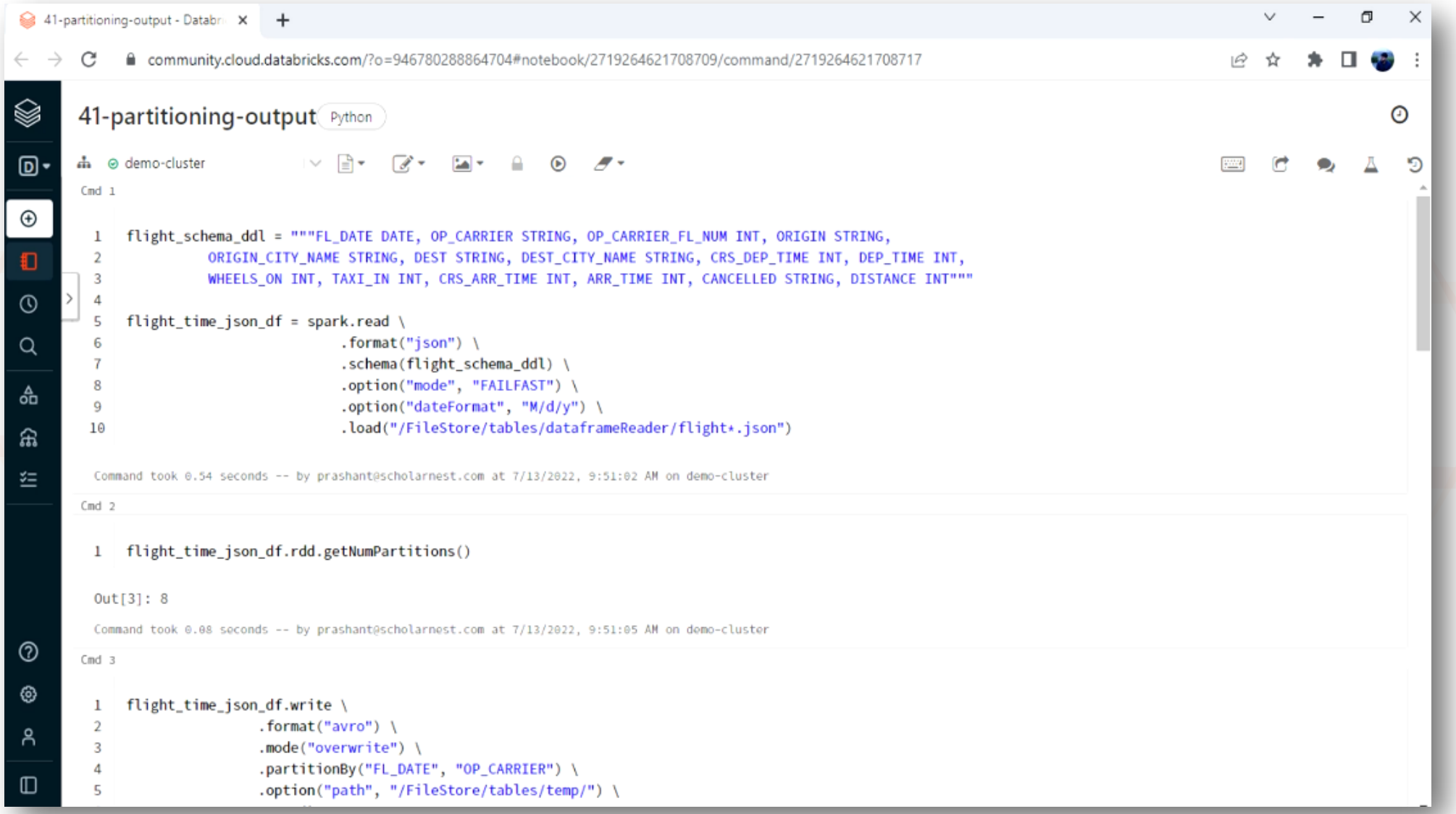
Databricks Spark Certification and Beyond

Module:
Data Sources & Sinks

Lecture:
Reading Partitioned Dataframes

Reading Partitioned Data Frames

Go to your Databricks workspace and open the previous lecture notebook. **(Reference: 41-partitioning-output.ipynb)**



We created flight_time_json_df Dataframe, and then we wrote DataFrameWriter code to save it to a new location. What was the location? “/FileStore/tables/temp/” Right?

Cmd 3

```
1  flight_time_json_df.write \
2                  .format("avro") \
3                  .mode("overwrite") \
4                  .partitionBy("FL_DATE", "OP_CARRIER") \
5                  .option("path", "/FileStore/tables/temp/") \
6                  .save()
```

▸ (1) Spark Jobs

Command took 1.33 minutes -- by prashant@scholarnest.com at 7/13/2022, 9:57:37 AM on demo-cluster

And we used the partitionBy() method on the DataFrameWriter. The partitionBy() will partition our data on FL_DATE and then create sub-partitions using the OP_CAREER field. These partitions are nothing but directories. Each partition is one directory, and sub partitions are subdirectories inside the parent partition directory.

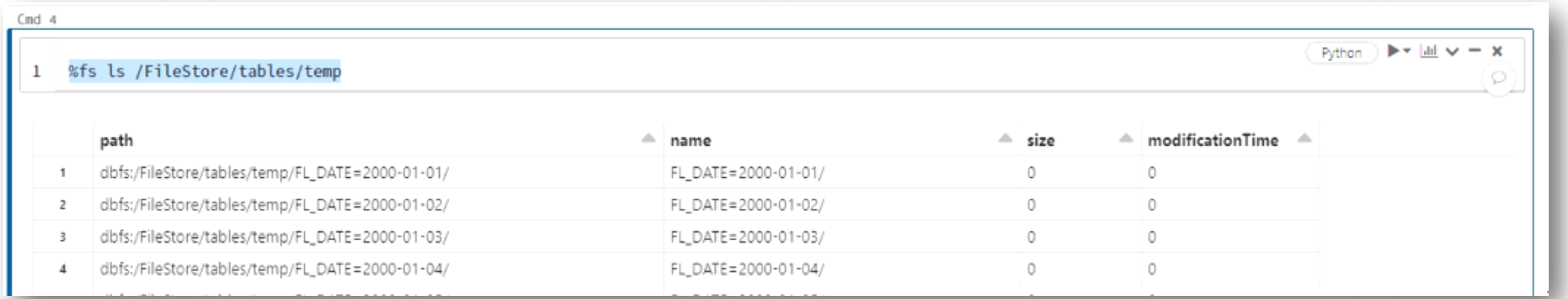
Cmd 3

```
1  flight_time_json_df.write \
2                  .format("avro") \
3                  .mode("overwrite") \
4                  .partitionBy("FL_DATE", "OP_CARRIER") \
5                  .option("path", "/FileStore/tables/temp/") \
6                  .save()
```

▸ (1) Spark Jobs

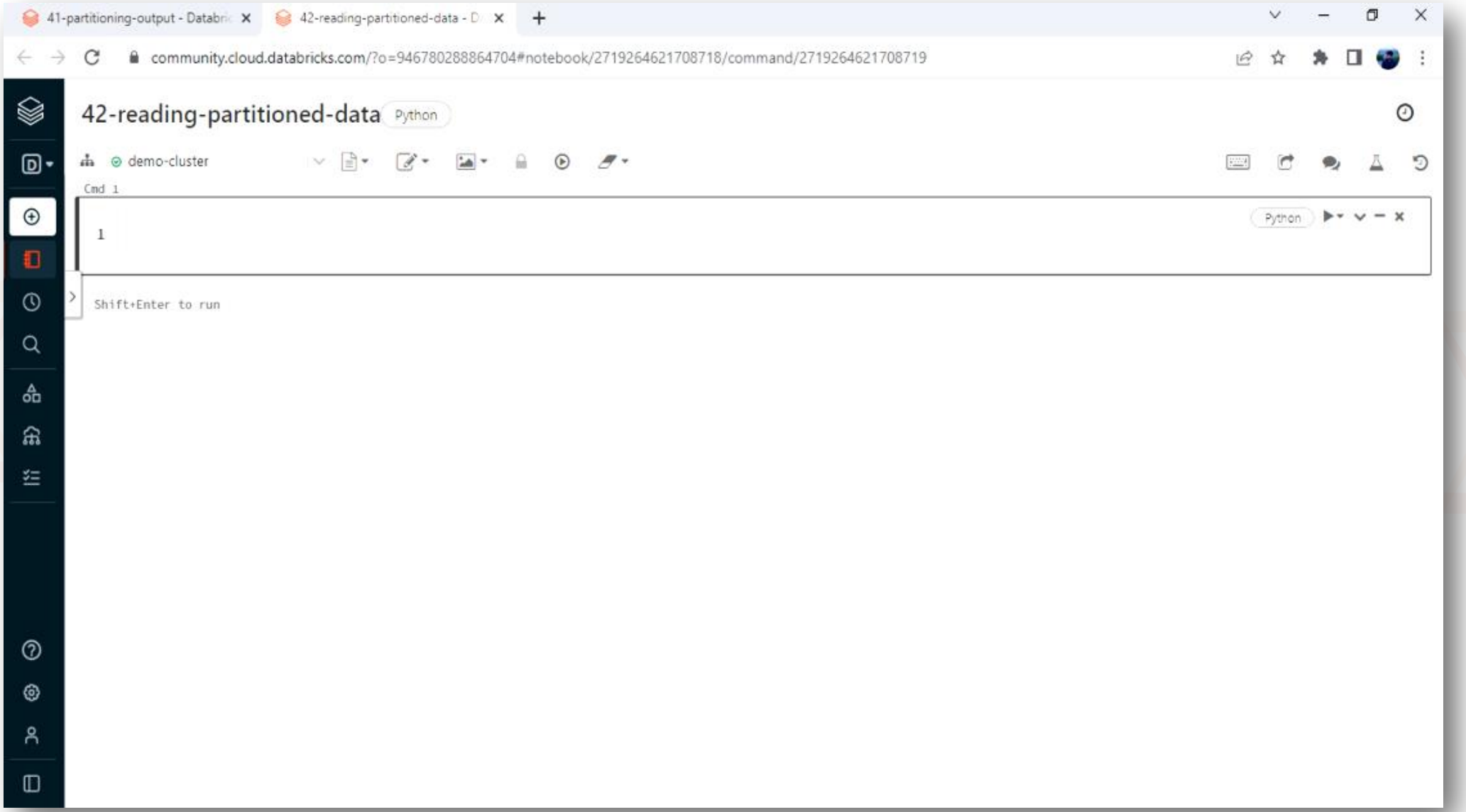
Command took 1.33 minutes -- by prashant@scholarnest.com at 7/13/2022, 9:57:37 AM on demo-cluster

Let me copy this ls command, and run this command from the new notebook.

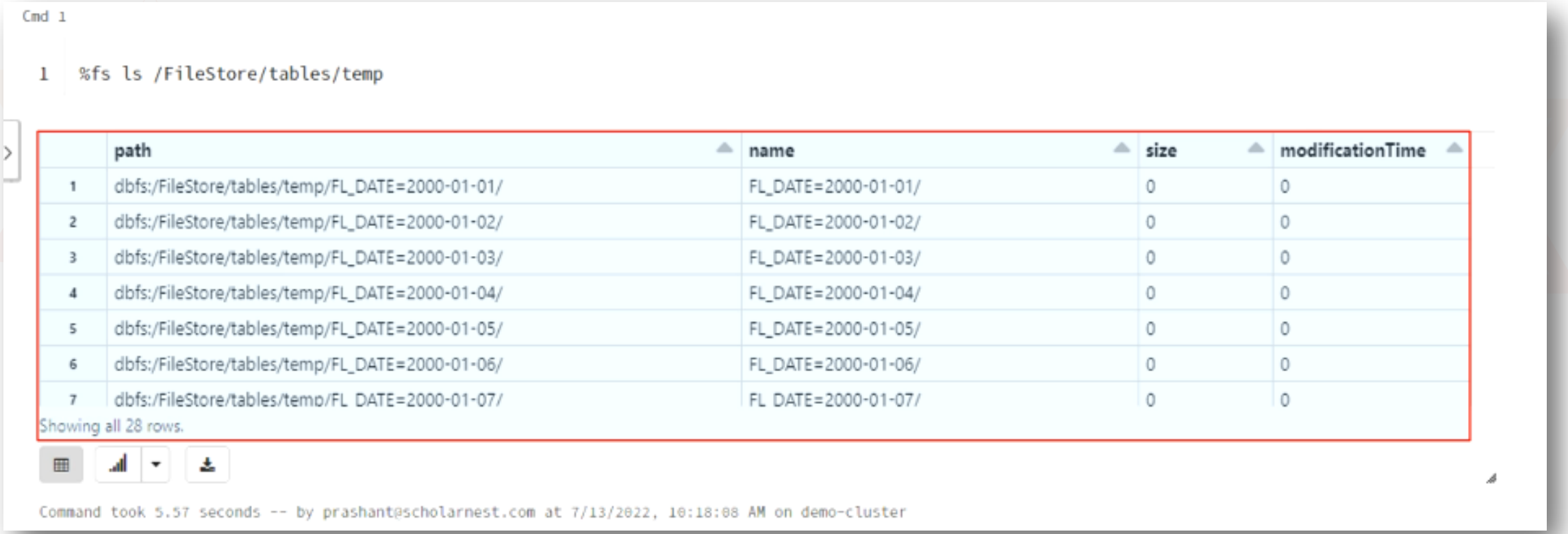


	path	name	size	modificationTime
1	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-01/	FL_DATE=2000-01-01/	0	0
2	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/	FL_DATE=2000-01-02/	0	0
3	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-03/	FL_DATE=2000-01-03/	0	0
4	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-04/	FL_DATE=2000-01-04/	0	0

Let me create a new notebook. **(Reference: 42-reading-partitioned-data.ipynb)**



Now let me paste the ls command and run it here in the new notebook as shown below. You can see these FL_DATE directories. These are nothing but my FL_DATE partitions. You can go inside any one partition and see the OP_CAREER sub partitions.



	path	name	size	modificationTime
1	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-01/	FL_DATE=2000-01-01/	0	0
2	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-02/	FL_DATE=2000-01-02/	0	0
3	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-03/	FL_DATE=2000-01-03/	0	0
4	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-04/	FL_DATE=2000-01-04/	0	0
5	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-05/	FL_DATE=2000-01-05/	0	0
6	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-06/	FL_DATE=2000-01-06/	0	0
7	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-07/	FL_DATE=2000-01-07/	0	0

So here are my OP_CAREER partitions.
You can go inside the one OP_CAREER partitions and see data files.

Cmd 2

```
1  %fs ls /FileStore/tables/temp/FL_DATE=2000-01-01/
```

	path	name	size	modificationTime
1	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-01/OP_CARRIER=AA/	OP_CARRIER=AA/	0	0
2	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-01/OP_CARRIER=AS/	OP_CARRIER=AS/	0	0
3	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-01/OP_CARRIER=CO/	OP_CARRIER=CO/	0	0
4	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-01/OP_CARRIER=DL/	OP_CARRIER=DL/	0	0
5	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-01/OP_CARRIER=HP/	OP_CARRIER=HP/	0	0
6	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-01/OP_CARRIER=NW/	OP_CARRIER=NW/	0	0
7	dbfs:/FileStore/tables/temp/FL_DATE=2000-01-01/OP_CARRIER=TW/	OP_CARRIER=TW/	0	0

Showing all 10 rows.

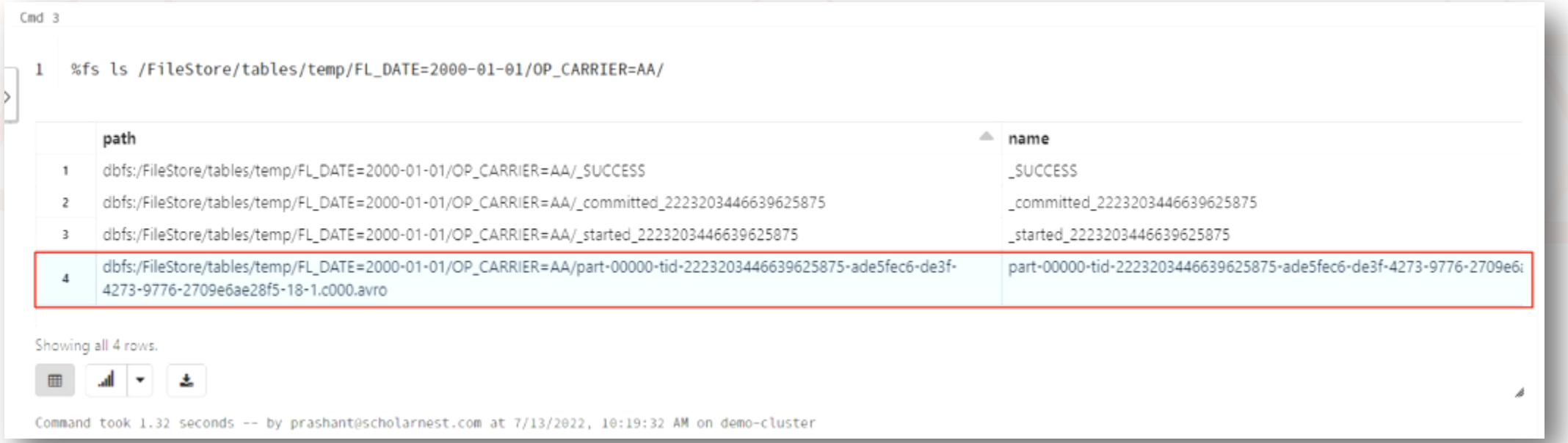
Command took 0.53 seconds -- by prashant@scholarnest.com at 7/13/2022, 10:18:54 AM on demo-cluster

Here is one AVRO data file.

All of this is great. But I have a question:

How do I read this data file?

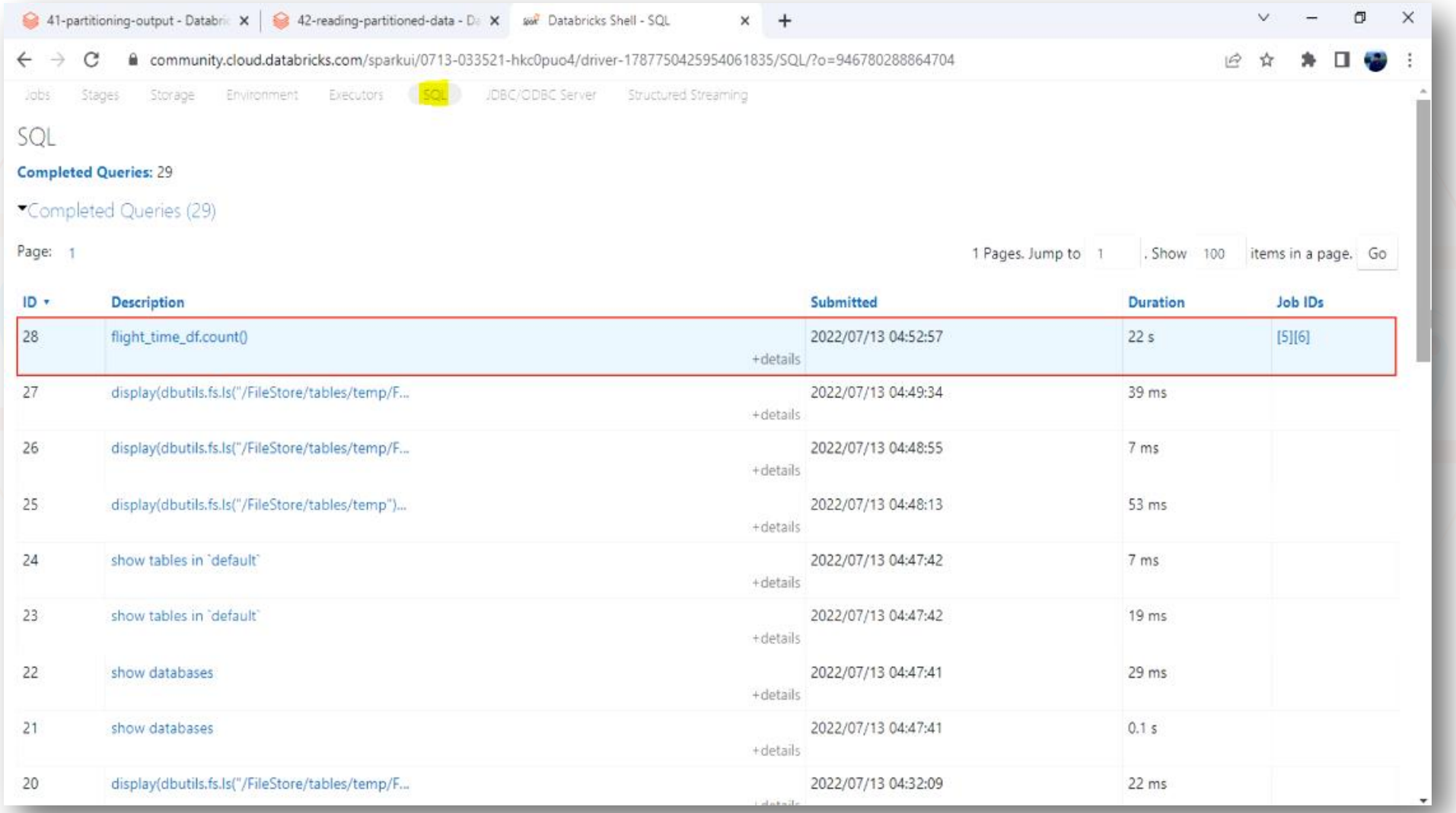
I mean, we use the spark DataFrameReader API to read the data from a given location.



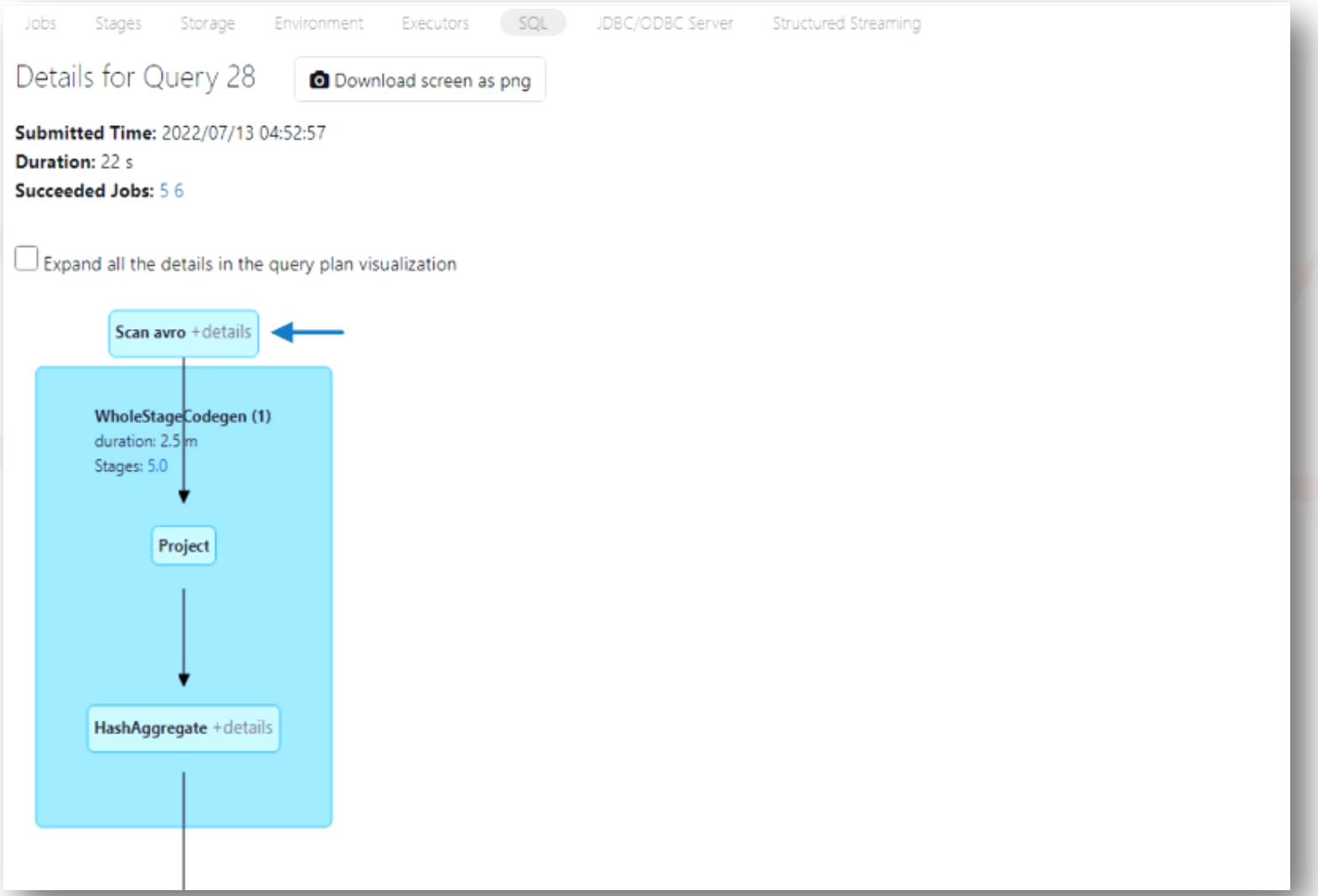
Here is the code for reading the Dataframe. But for reading a data file, I must give a location in the load method. The load method takes two types of inputs.
I can give only the directory location, and it will read all the data files from the given directory.
I can also give directory location and the file name, so the DataFrameReader reads only a given data file.
But I created partition directories, and now I have so many directories.
Which directory should I give to the load method?
The answer is super simple.
You will give the base directory name.

So I am giving only the base directory name where I created partitioned data.
And the Spark DataFrameReader is smart enough to read all the partition directories and the sub partition directories.
I ran the cell and took the Dataframe count in the next cell. You do not see any errors. That means the DataFrameReader reads all the data files inside the base directory. It will automatically navigate the subdirectories, find the data files, and read them all. You can see that count here as well.

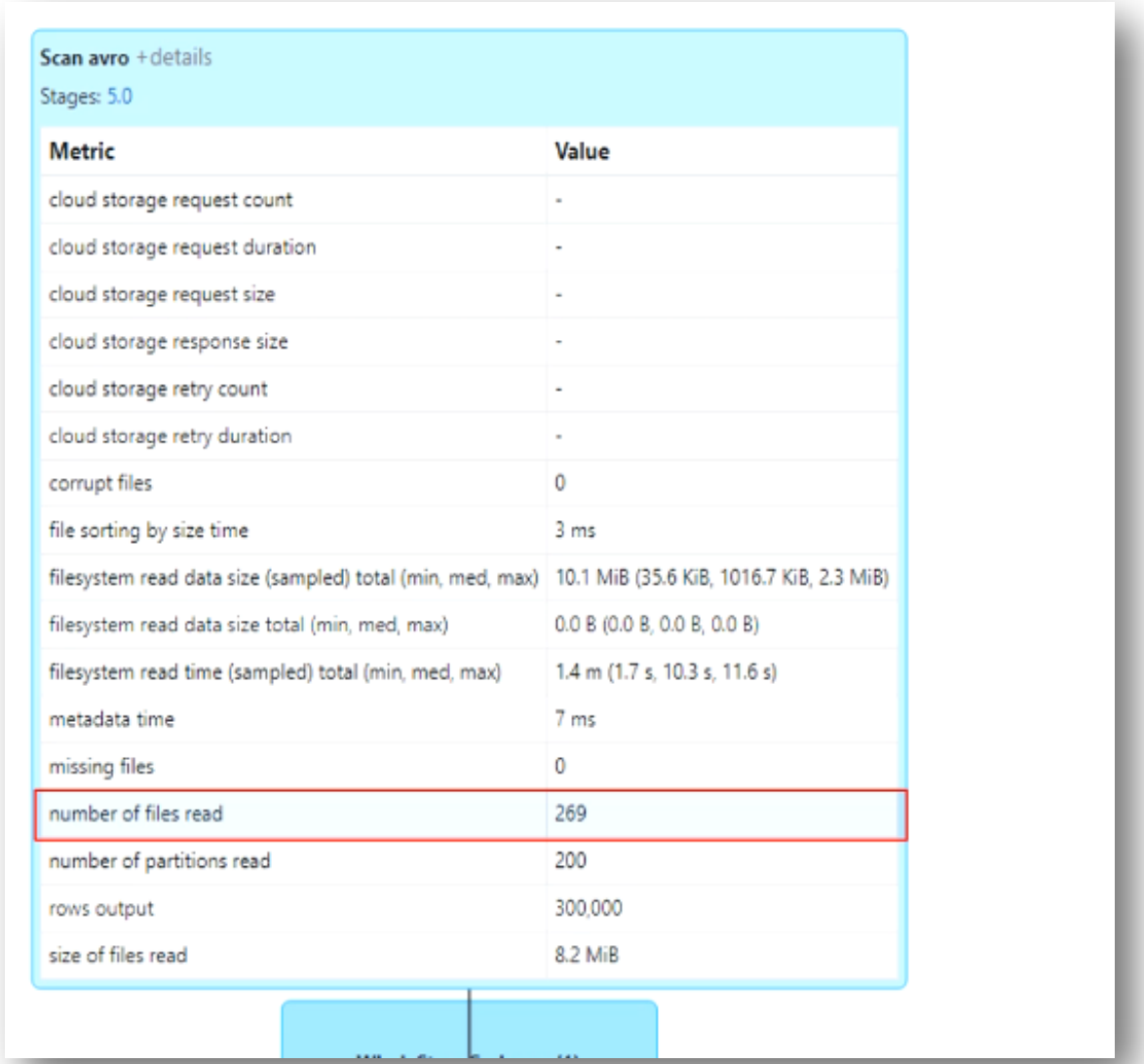
Now go to the Spark UI, and navigate to the SQL tab.
The top most query is your flight_time_df.count(). Click that, and it will show you more details.



Do you see this DAG and the Scan Avro at the top? That's the first thing your code will do. So when you run flight_time_df.count(), it will scan the Avro files. Reading data files is the first thing. Then it will count the records. Click the details next to the scan Avro.



Scroll down, and you will see "number of files read"
Look at this count. Now you know Spark DataFrameReader is reading these many files to give you the count.



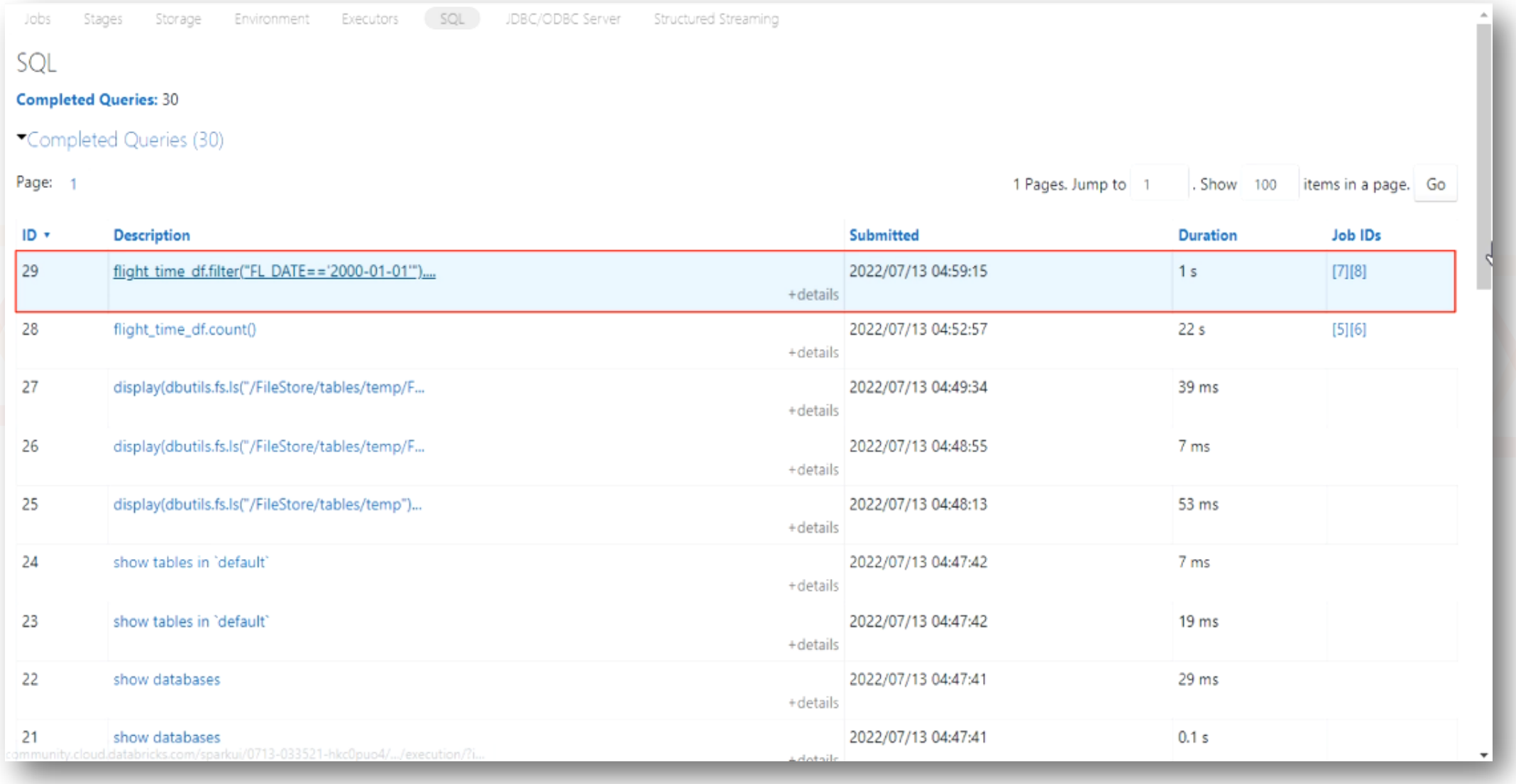
Scan avro +details

Stages: 5.0

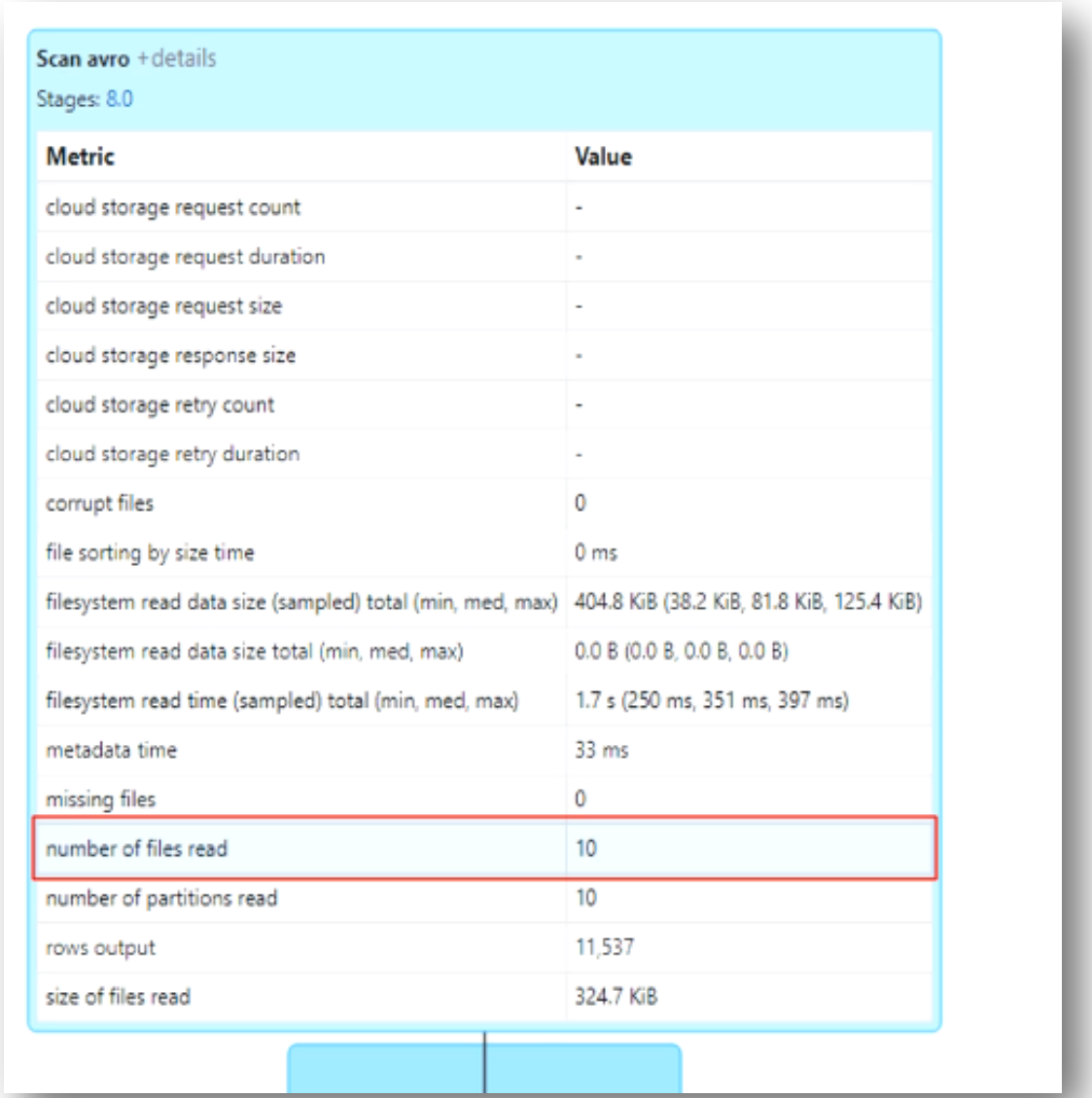
Metric	Value
cloud storage request count	-
cloud storage request duration	-
cloud storage request size	-
cloud storage response size	-
cloud storage retry count	-
cloud storage retry duration	-
corrupt files	0
file sorting by size time	3 ms
filesystem read data size (sampled) total (min, med, max)	10.1 MiB (35.6 KiB, 1016.7 KiB, 2.3 MiB)
filesystem read data size total (min, med, max)	0.0 B (0.0 B, 0.0 B, 0.0 B)
filesystem read time (sampled) total (min, med, max)	1.4 m (1.7 s, 10.3 s, 11.6 s)
metadata time	7 ms
missing files	0
number of files read	269
number of partitions read	200
rows output	300,000
size of files read	8.2 MiB

Now to understand the benefit of creating partitions, go back to the notebook and apply a filter condition on the Dataframe and count it again.
Now I want to count only the number of records for 1st January 2020.

Go to the Spark UI once again. And again navigate to the SQL tab and check the most recent command.



You can again navigate to the Scan Avro step. Click the details, and let's see how many files we read. This is less. And that's why we partition our data.



Scan avro +details

Stages: 8.0

Metric	Value
cloud storage request count	-
cloud storage request duration	-
cloud storage request size	-
cloud storage response size	-
cloud storage retry count	-
cloud storage retry duration	-
corrupt files	0
file sorting by size time	0 ms
filesystem read data size (sampled) total (min, med, max)	404.8 KiB (38.2 KiB, 81.8 KiB, 125.4 KiB)
filesystem read data size total (min, med, max)	0.0 B (0.0 B, 0.0 B, 0.0 B)
filesystem read time (sampled) total (min, med, max)	1.7 s (250 ms, 351 ms, 397 ms)
metadata time	33 ms
missing files	0
number of files read	10
number of partitions read	10
rows output	11,537
size of files read	324.7 KiB

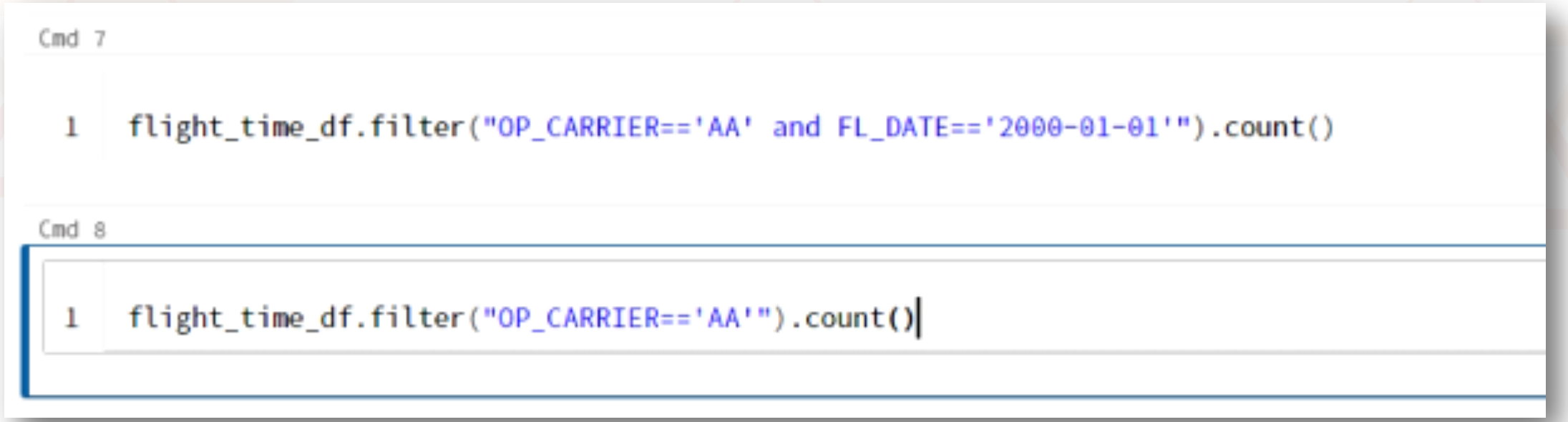
Partition your data using the filter columns.
You should know what queries you are planning to fire on your data.
You will realize that most of your future queries will be filtering on some commonly used columns.

Once you know that, you can partition your data on those columns.
This partitioning will give a massive boost to your queries. Why? Because Spark will be reading fewer data.
Reading fewer data takes less time, less memory, and less computation.
So everything is less, and your queries perform faster. RIght?

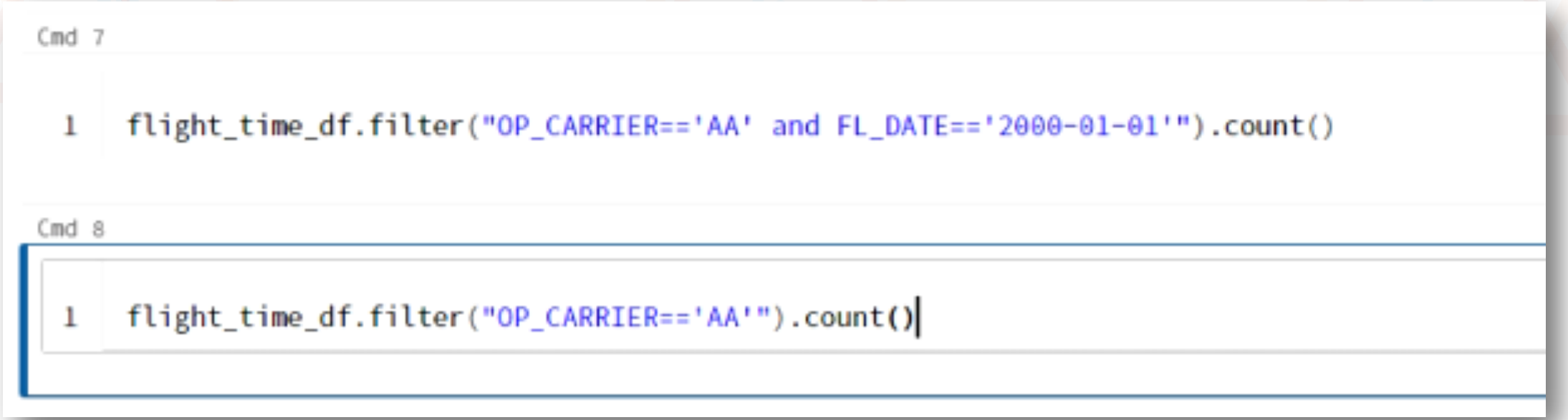
Great! I hope you got the following things:

1. How to partition your data and Spark tables.
2. How to read the partitioned tables?
3. Why you should partition your tables?

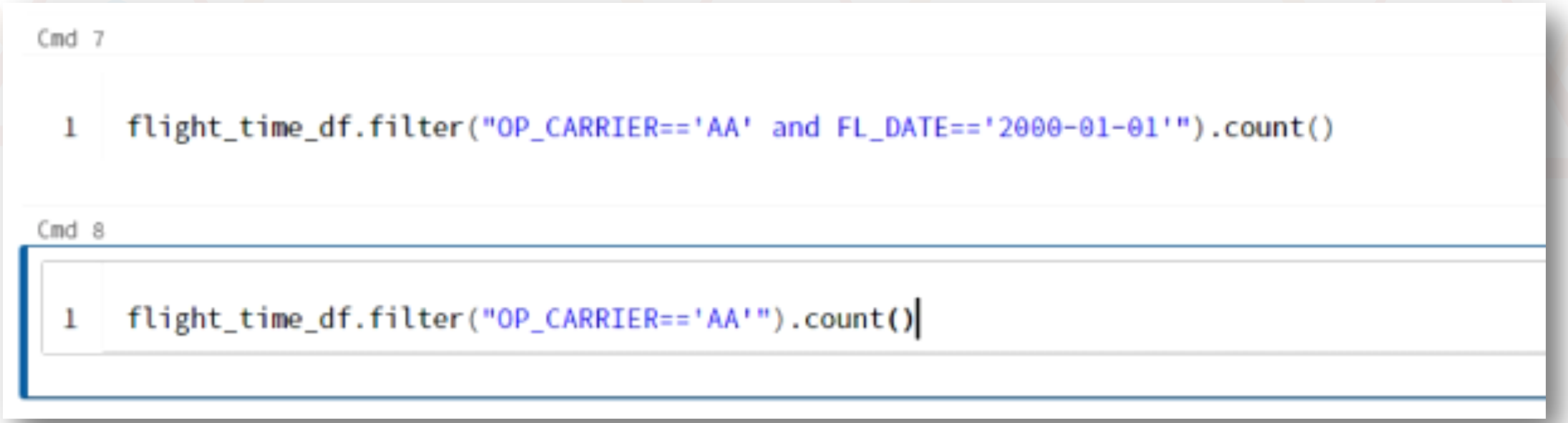
I have written two more queries here.
Now, let me ask a question.
Which one of these will benefit from the partitioning that we already created?
Take a pause and think about it.



The first one is filtering for two partition columns. So this one will read less number of files. I guess it will read only one file because we know we have only one data file per sub partition.
One more thing. I gave the condition in the wrong order.
I mean, the FL_DATE is the top-level partition, and the OP_CAREER is the sub partition.
I am applying the OP_CAREER filter first and then applying FL_DATE.
But that doesn't matter. Spark is smart enough to correct the order.
So this query is expected to read only one sub partition and hence only one file.



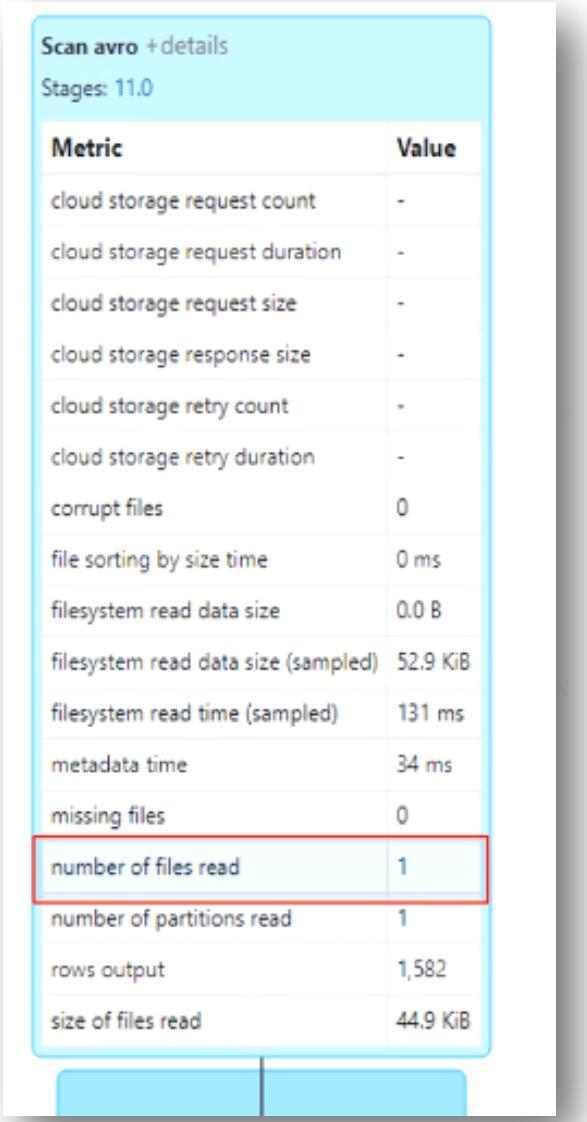
What about the next one?
This query is filtering only on the sub partition. I didn't apply the filter on the FL_DATE, which is the top-level partition. Will this query benefit from the partitioning?
Yes, it will. This query will still benefit from the partitioning. It will go into all top-level partitions and read only one sub partition from each top-level partition.
Benefits are less because we go in all top-level partitions, but it is still much better than reading all the 200+ files.



Now let me run both of these and check the number of files read from the Spark UI.

```
Cmd 7

1  flight_time_df.filter("OP_CARRIER=='AA' and FL_DATE=='2000-01-01'").count()

▸ (2) Spark Jobs

Out[4]: 1582

Command took 0.76 seconds -- by prashant@scholarnest.com at 7/13/2022, 10:34:01 AM on demo-cluster

Cmd 8

1  flight_time_df.filter("OP_CARRIER=='AA'").count()

▸ (2) Spark Jobs

Out[5]: 39322

Command took 2.44 seconds -- by prashant@scholarnest.com at 7/13/2022, 10:34:05 AM on demo-cluster
```


Here is the result for the double filter query. There is only one number of file here.

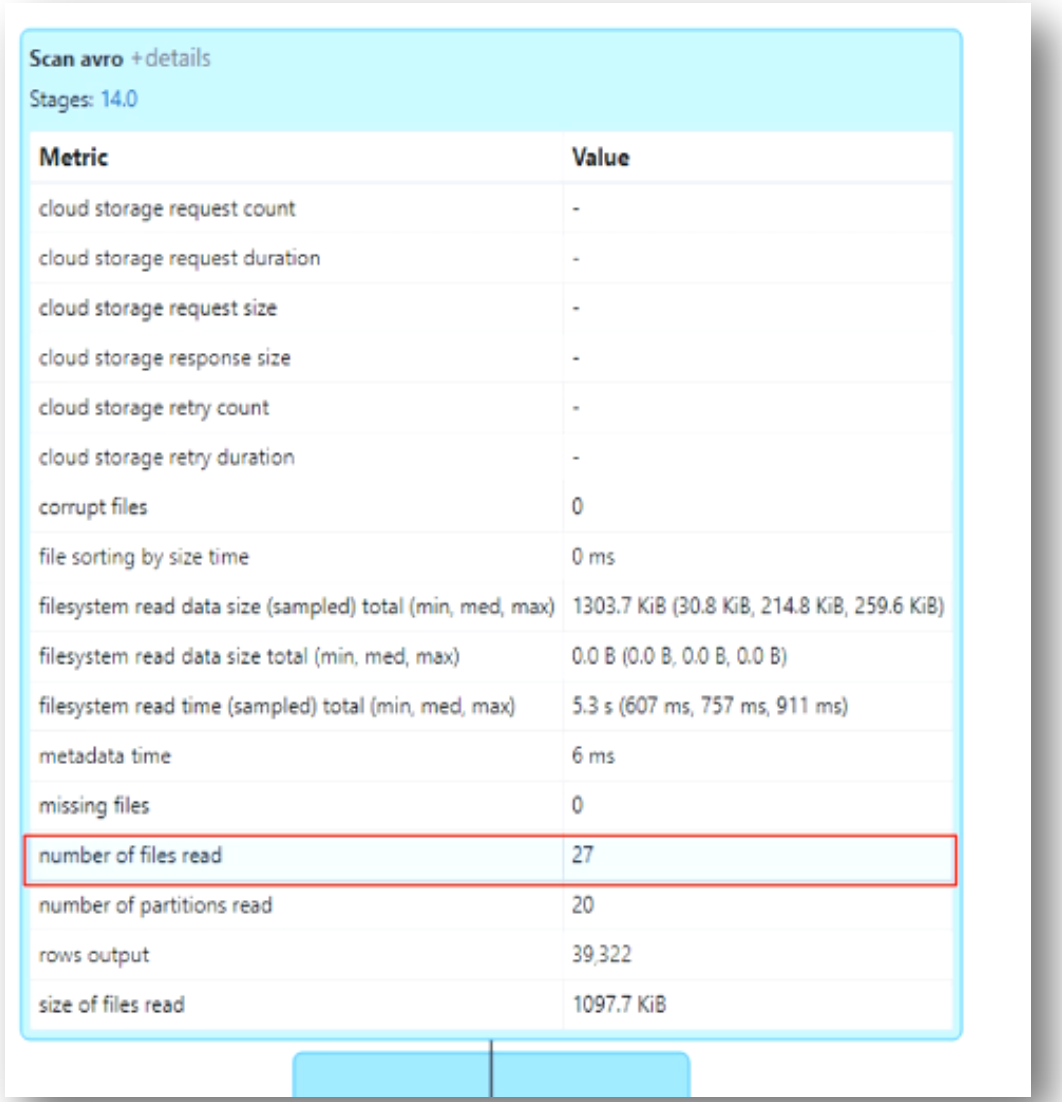


Scan avro +details

Stages: 11.0

Metric	Value
cloud storage request count	-
cloud storage request duration	-
cloud storage request size	-
cloud storage response size	-
cloud storage retry count	-
cloud storage retry duration	-
corrupt files	0
file sorting by size time	0 ms
filesystem read data size	0.0 B
filesystem read data size (sampled)	52.9 KiB
filesystem read time (sampled)	131 ms
metadata time	34 ms
missing files	0
number of files read	1
number of partitions read	1
rows output	1,582
size of files read	44.9 KiB

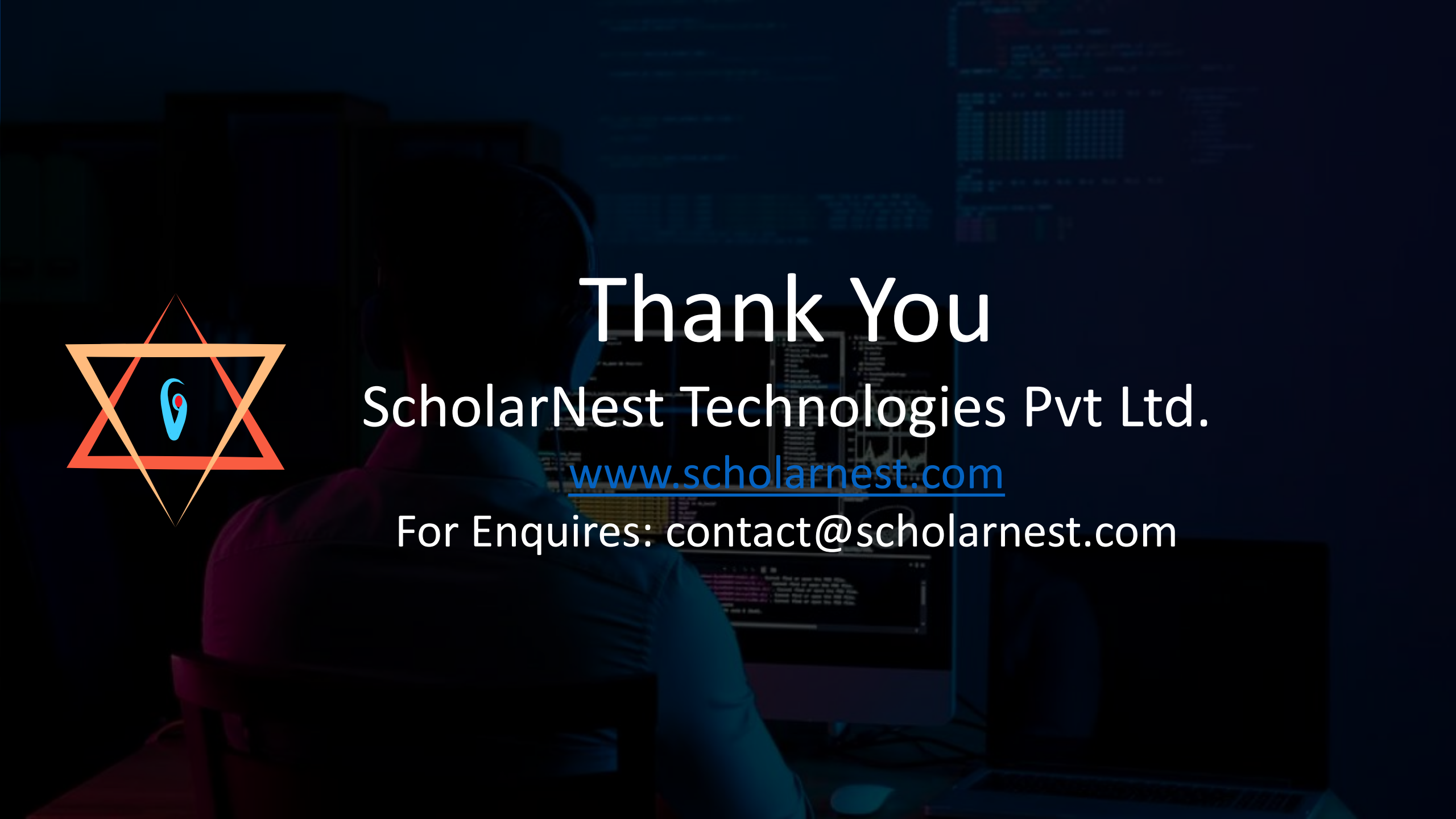
Now if you see the results for the OP_CARRIER filter, you will see that this one is a little more. But still a lot less than 250+. My dataset is stored as 250+ files. But this query is reading approximately only 10% of the total.



Scan avro +details

Stages: 14.0

Metric	Value
cloud storage request count	-
cloud storage request duration	-
cloud storage request size	-
cloud storage response size	-
cloud storage retry count	-
cloud storage retry duration	-
corrupt files	0
file sorting by size time	0 ms
filesystem read data size (sampled) total (min, med, max)	1303.7 KiB (30.8 KiB, 214.8 KiB, 259.6 KiB)
filesystem read data size total (min, med, max)	0.0 B (0.0 B, 0.0 B, 0.0 B)
filesystem read time (sampled) total (min, med, max)	5.3 s (607 ms, 757 ms, 911 ms)
metadata time	6 ms
missing files	0
number of files read	27
number of partitions read	20
rows output	39,322
size of files read	1097.7 KiB



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com