ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Sampling and Splitting

Data analysis and Data scientists often want to do the following:

1. Take a random sample of a large data set
2. Randomly Split dataset to create training, validation, and test sets.
3. Combine data frames

1. **Take a random sample of a large data set**

   Assume you have a huge data frame of billions of records.
   You are developing some logic for data analysis.
   And you want to try and test your logic on the real data.
   But your Dataframe is huge.
   Testing your logic on billions of records will take time and resources.
   So you want to test it on a small portion of the real data frame.
   Spark allows you to take a random sample from your actual Dataframe and use it for testing.

2. **Randomly split dataset to create training, validation and data sets.**

   Machine learning and data science often need to split the Dataframe into two or more parts.

   They use one part of the data frame for training their models.

   Other parts may be used for validation and testing the models.

   So randomly splitting a Dataframe is also a common requirement.

   Spark allows you randomly split your Dataframes.

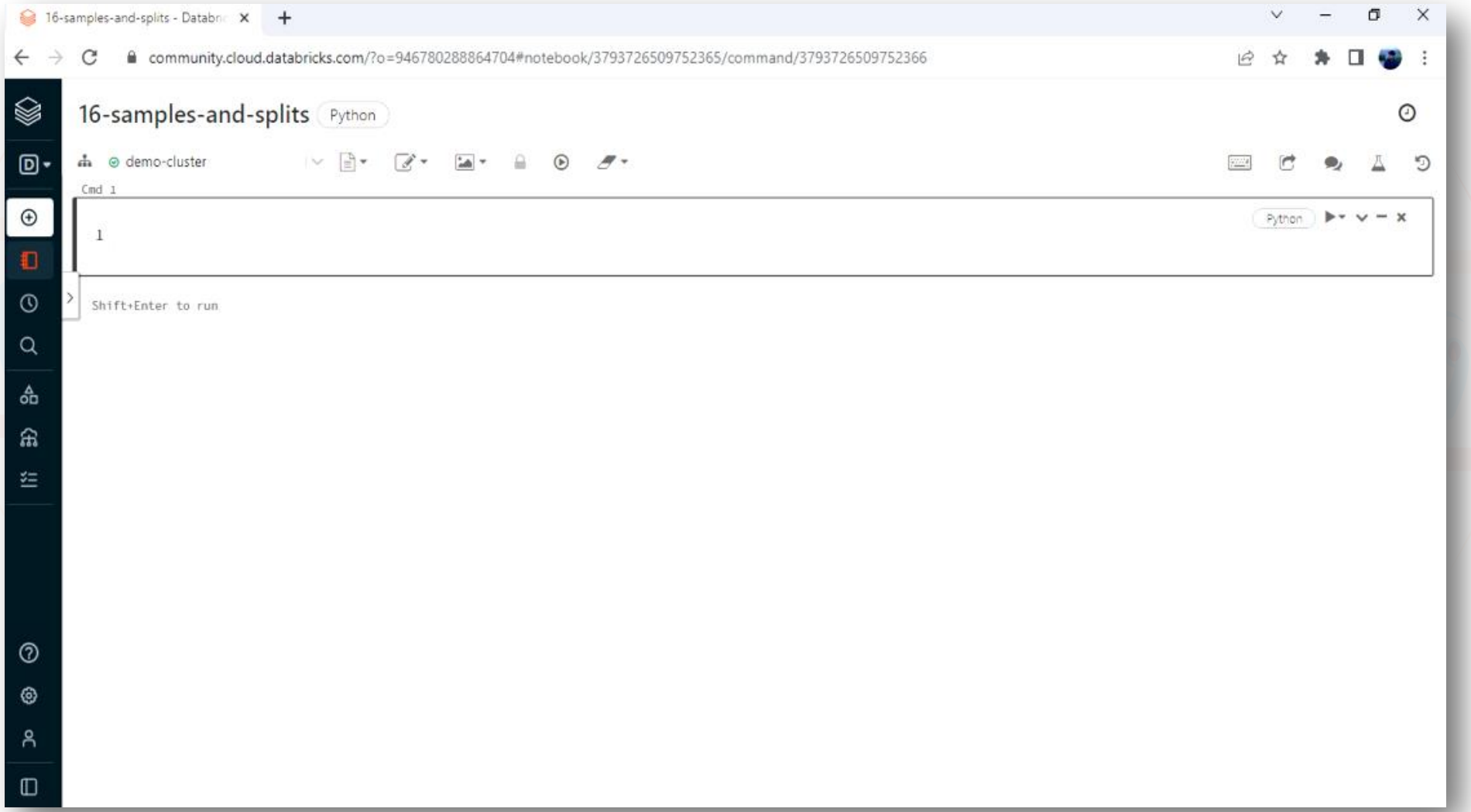
3. Combine Dataframes

When you learn to split a data frame into two or more parts, it is also essential to learn to combine two parts into a single Dataframe.

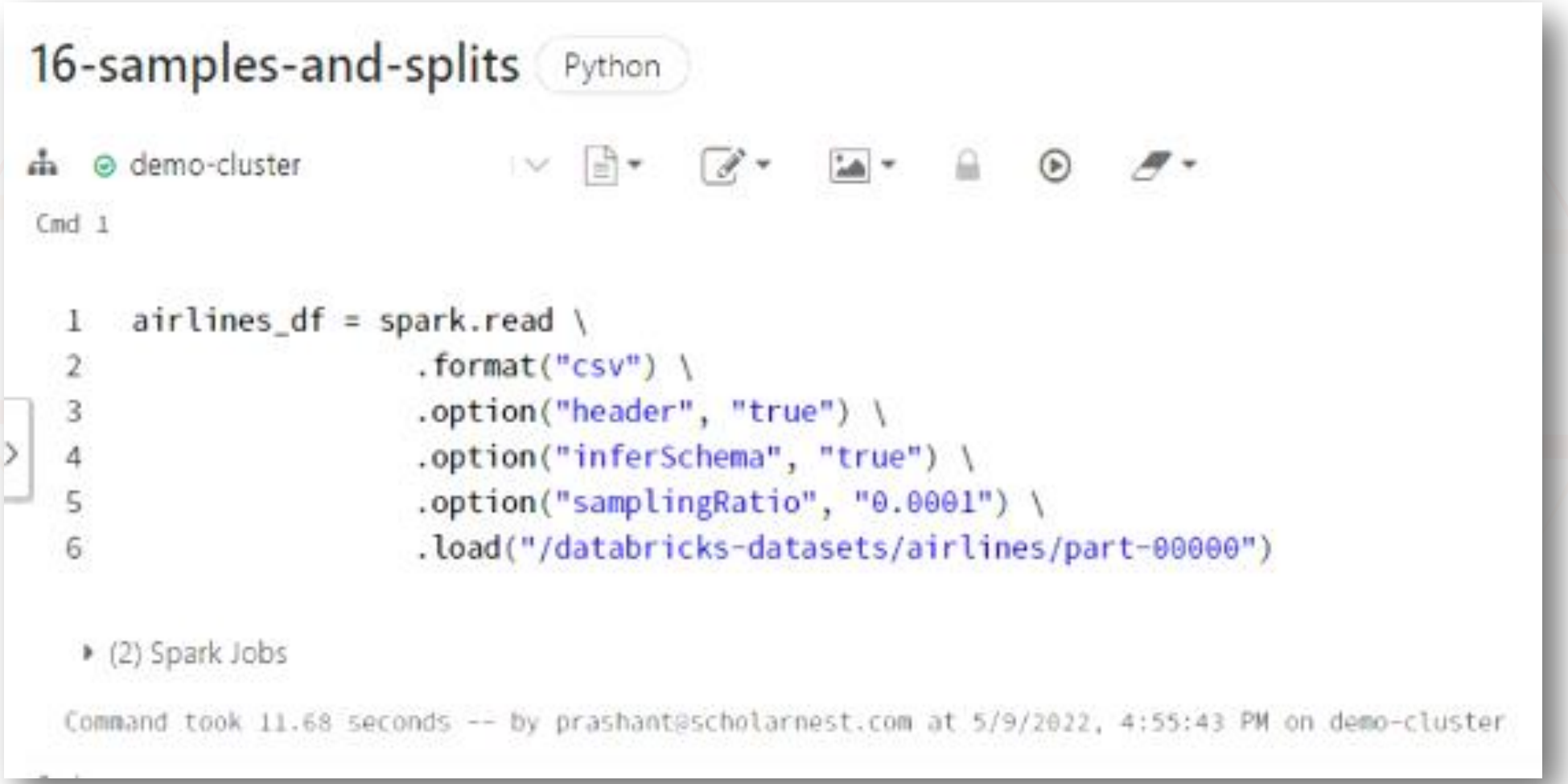
Spark offers two methods for sampling your Dataframe:

1. sample(withReplacement=None, fraction=None, seed=None)
2. sampleBy(col, fractions, seed=None)

Go to your Databricks workspace and create a new notebook.
(Reference : 16-samples-and-splits)



We need a large data frame so we can try sampling and splitting. So I have created one Dataframe using the airline sample data.

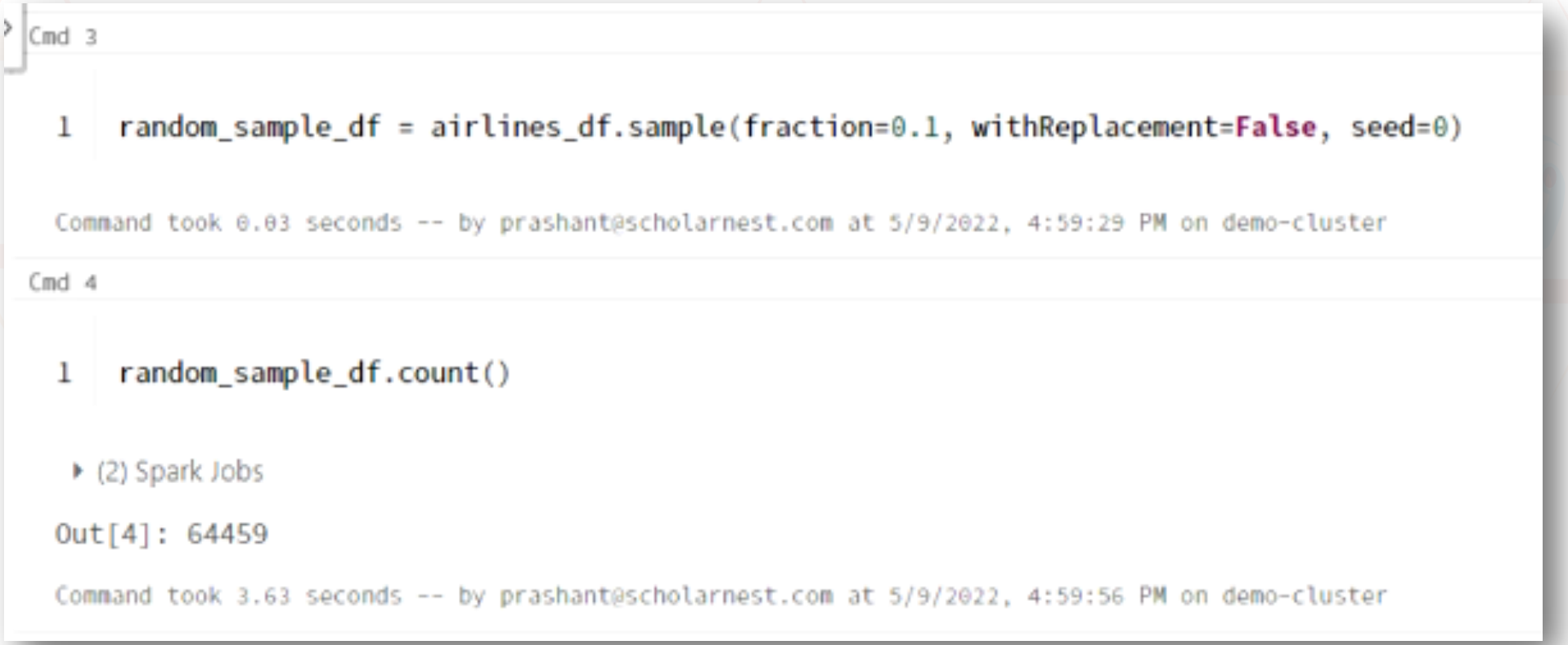


We have approximately six lakhs and forty-five thousand records. Now I want to take 10 thousand of these records. You can do a select * and limit 10000. But that will not be a proper mix or random sampling of records. When we want to take a sample, we mean that each record in the data frame should have an equal probability of being selected in the sample.

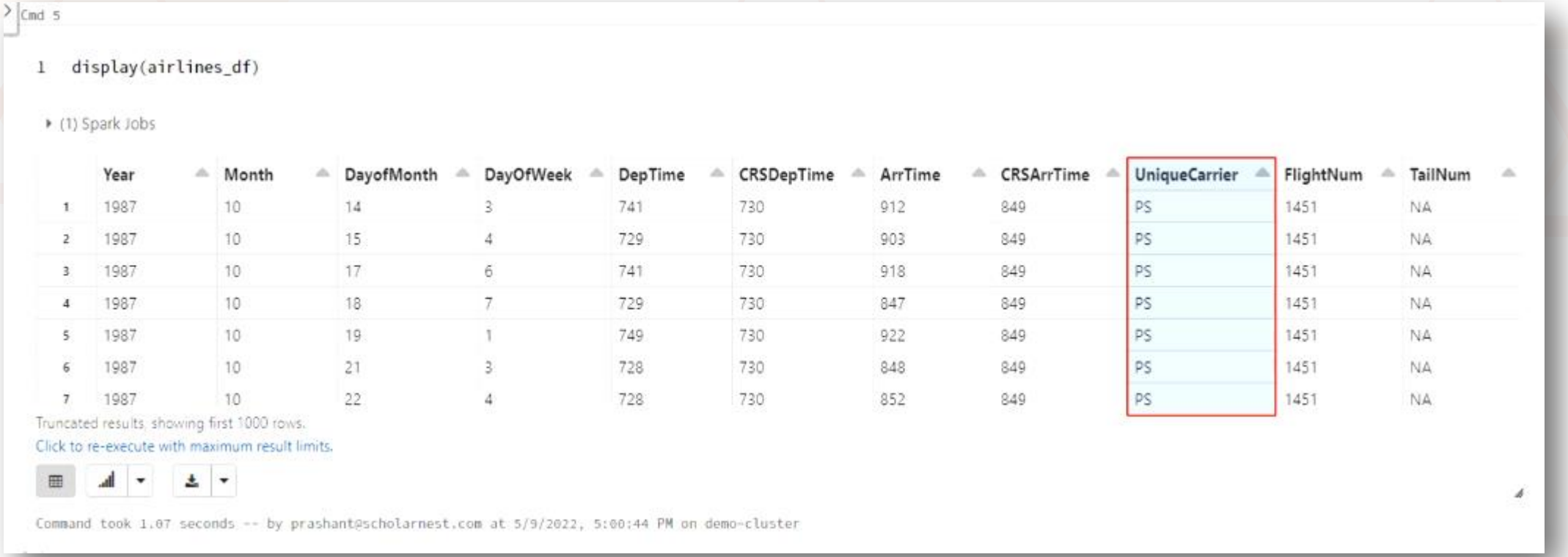
So I am sampling the *airlines_df* and creating *random_sample_df*. The sample() method takes three arguments.
The fraction is a required parameter, and it represents the sample percentage. I need 10%, so I am setting the fraction to 0.1. You can give any value from 0 to 1, representing the sample size in percentage.
The other two arguments are optional.
The withReplacement() take a true/false value. Can we?
If you want to take the same record twice, set the withReplacement to true. Otherwise, set it to false.
The seed is the seed of randomization.
You can use a seed to reproduce the same sample again.
It could be an integer value, and using the same seed across multiple runs will produce the same sample Dataframe.

```
Cmd 3

1  random_sample_df = airlines_df.sample(fraction=0.1, withReplacement=False, seed=0)

Command took 0.03 seconds -- by prashant@scholarnest.com at 5/9/2022, 4:59:29 PM on demo-cluster
```

You can see the count of the random sample shown below. The count may not be precisely 10% of your base data frame, but it will be close.
It happens because the sample() method does not give a guarantee to provide an exact fraction of the total count.
However, the result will be close to the fraction specified.



We have the airlines_df Dataframe shown below.
We have the UniqueCarrier field highlighted below.
Now let's assume you want to bucket your data by the UniqueCarrier and then take samples from each bucket.
Each bucket is known as strata, and the approach is called Stratification.
Sometimes, it is desired to take stratified samples.


Cmd 5

```
1  display(airlines_df)
```

(1) Spark Jobs

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

Truncated results, showing first 1000 rows.
Click to re-execute with maximum result limits.

Command took 1.07 seconds -- by prashant@scholarnest.com at 5/9/2022, 5:00:44 PM on demo-cluster

So I am filtering airlines_df and taking only three UniqueCarrier codes. This step is not necessary for generating stratified examples. But I am limiting the options so you can understand them easily.
Then I counted records for each carrier code, and the result is shown below. So in my base_df, I have 56091 records for AA, 63104 for DL, and so on.
Now I have a requirement to create two strata. One for AA and another for DL. I do not care about the PS or other carriers because that's not in my requirement. Once we have two strata, I want to take 10K records from each stratum and create a sample of 20K records.

Requirements:

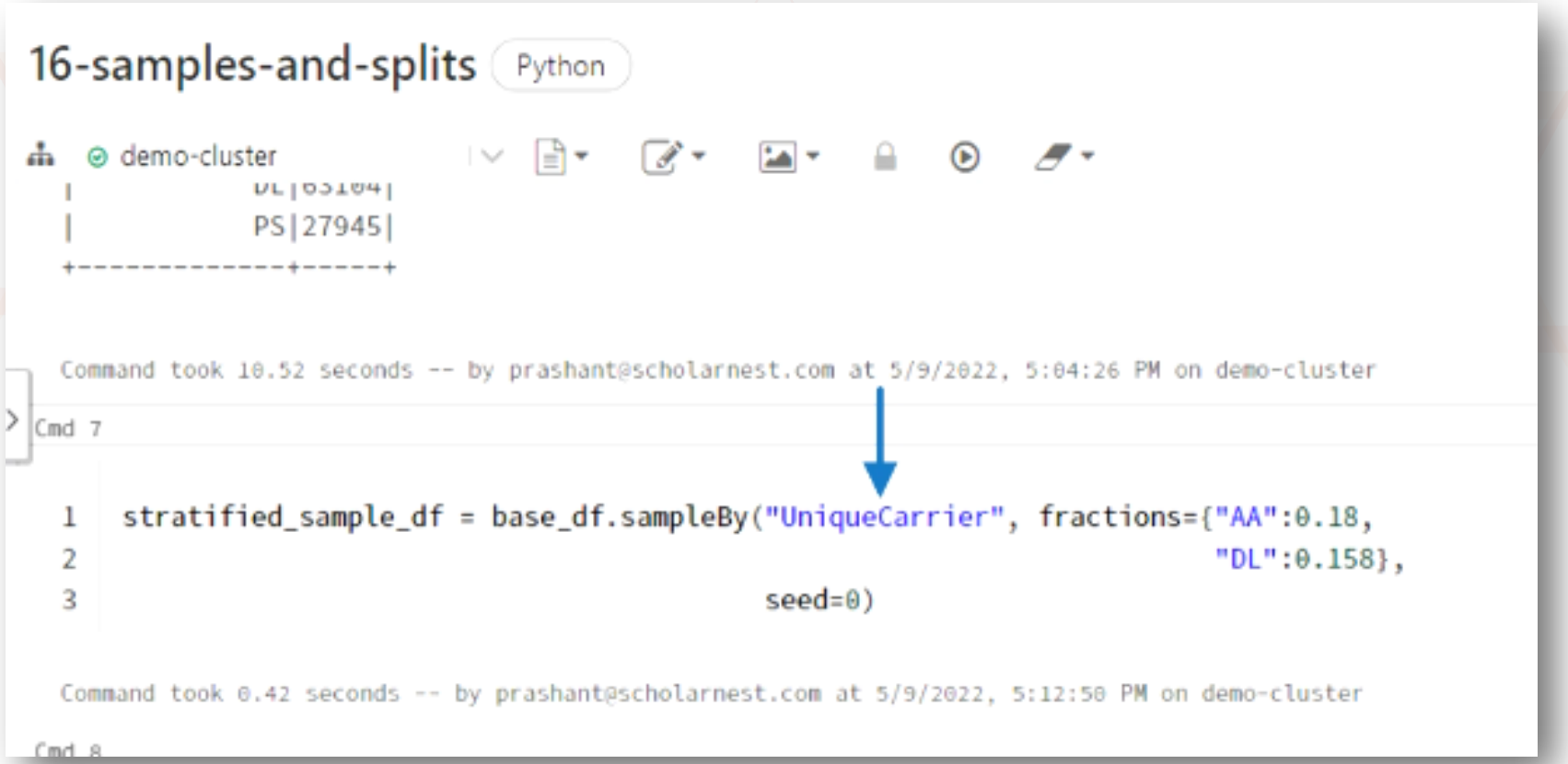
- 20K samples for AA and DL
- Random Sampling
- Each code should have approx. 10K samples

I need data from two Carrier codes. So I will create two strata. For selecting records from strata, you need a sampling fraction for the strata. And here is the calculation of sampling fraction.

Sampling Fraction Calculation:

- AA : 10,000/56091 = 0.18
- DL : 10,000/63104 = 0.158

I am using the sample() method on my base data frame to create a stratified sample. The sample() method takes three arguments. The first argument is the strata column name. So this example will create strata using the unique carrier codes. I have three unique carriers in my Dataframe, so the sample() will create three strata. A stratum is nothing but a kind of record bucket. So all the records for AA will go in one bucker. Similarly, all the records for DL will be in a different bucket, and so on. Each unique carrier code will become one bucket, and we call them strata.



The second argument is for strata fractions. I want to sample only AA and DL. So I will give only two fractions.
The fraction for AA is 0.18, and DL is 0.158. I am giving only two fractions. So Spark will sample records from only these two buckets.

The last argument is the randomization seed.
You can use a seed to reproduce the same sample again.
It could be an integer value, and using the same seed across multiple runs will produce the same sample Dataframe.

You can see the stratified Dataframe shown below, we got the desired output.
That's all about sampling data frames.
We have two options.

1. sample() method, which applies simple random sampling.
2. We also have a sampleBy() that applies stratified random sampling.

Now let’s move onto the next topic of Splitting a Dataframe.
Let's assume you have airlines_df. And you want to split this Dataframe into three parts. The split must be random. And the split ratio should be 1:2:1. This ratio is the same as 25%:50%:25%. Spark gives you the randomSplit() method for splitting a dataframe. So I can use the randomSplit() method on the airlines_df and split it into multiple parts as shown below. I am splitting it into three parts. But you can split and create 2 or 5 or whatever number. The randomSplit() takes two arguments. The first one is a list of weights for each split.
You can give weights in floating-point numbers. The weight must be a floating-point number, and the spark will normalize the values into a split percentage.

You can check the count of the three data frames as shown below.
The counts are in the given percentage. It may not be precisely 25%:50%:25%, but it should be close enough.
The randomization doesn't give a guarantee for an exact count.

The next topic is to combine Dataframes.
Can we combine smaller Dataframe to create one large Dataframe? Yes. We can if they have the same schema.
Spark allows you to combine two or more data frames as long as they have the same schema.
we have a union method to combine two data frames. If you have three data frames to combine, you can create a chain of union() methods and combine multiple data frames. And you can see that the Dataframe df4's count is equal to the sum of the count of Dataframes df1 and df3.

You should remember a few critical things about the union:

1. Both the combining Dataframe must have the same schema.
2. Union is applied by column order. So the first column of df2 combines with the first column of df1. So the column order is essential.
3. Union method is the same as UNION ALL in SQL. So it will not remove duplicates. If you have the same record in two data frames, the combined Dataframe will have both records. If you want to eliminate duplicates, you can use the distinct() method on the combined Dataframe.

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

Absolute Beginner to Specialization in Apache Spark and Azure Databricks

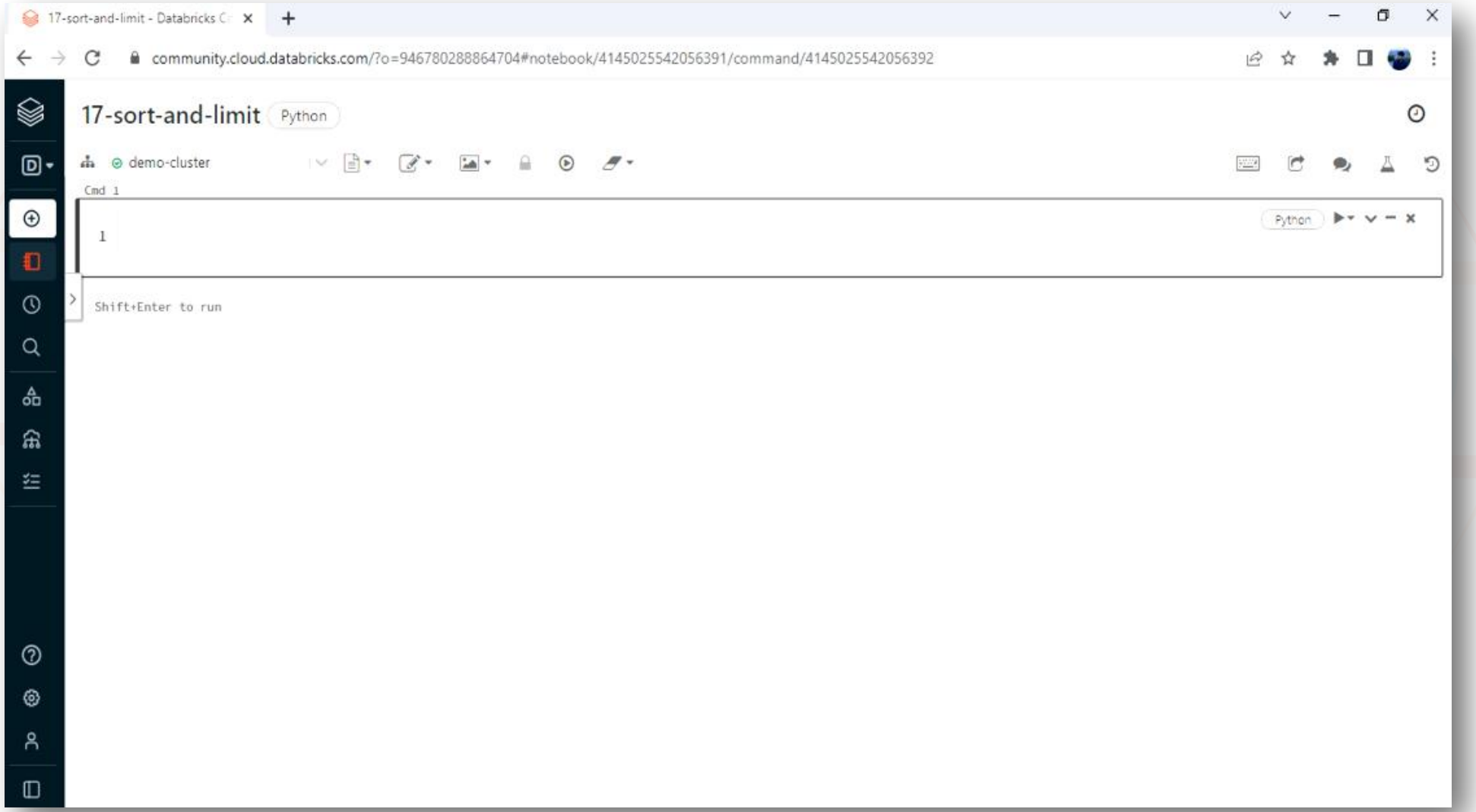
Sorting and Limiting

When we sort the Dataframe, we always want to specify two things:

1. Sorting columns
2. Sorting order

We may want to sort the Dataframe with one column or multiple columns. And sorting order could be ascending or descending. Finally, you may also have a requirement to limit the top 5 or top 10 records.

Go to your Databricks workspace and create a new notebook.
(Reference : 17-sort-and-limit)



We need a Dataframe for applying various sorting techniques. So I have created a Dataframe and displayed it as shown below.

17-sort-and-limit Python

demo-cluster

Cmd 1

```
1  airlines_df = spark.read \
2                 .format("csv") \
3                 .option("header", "true") \
4                 .option("inferSchema", "true") \
5                 .option("samplingRatio", "0.0001") \
6                 .load("/databricks-datasets/airlines/part-00000")
7
8  display(airlines_df)
```

(3) Spark Jobs

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

Truncated results, showing first 1000 rows.

Click to re-execute with maximum result limits.

Command took 4.17 seconds -- by prashant@scholarnest.com at 5/9/2022, 7:57:00 PM on demo-cluster

Now let's assume I want to summarize the count of airline data over the following fields:

1. UniqueCarrier
2. Origin
3. Dest
4. FlightNum

In order to achieve this requirement, we will group by the Dataframe over these fields and calculate the count.

We need a Dataframe for applying various sorting techniques. So I have created a Dataframe and displayed it as shown below.

17-sort-and-limit Python

demo-cluster

Cmd 1

```
1  airlines_df = spark.read \
2                 .format("csv") \
3                 .option("header", "true") \
4                 .option("inferSchema", "true") \
5                 .option("samplingRatio", "0.0001") \
6                 .load("/databricks-datasets/airlines/part-00000")
7
8  display(airlines_df)
```

(3) Spark Jobs

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

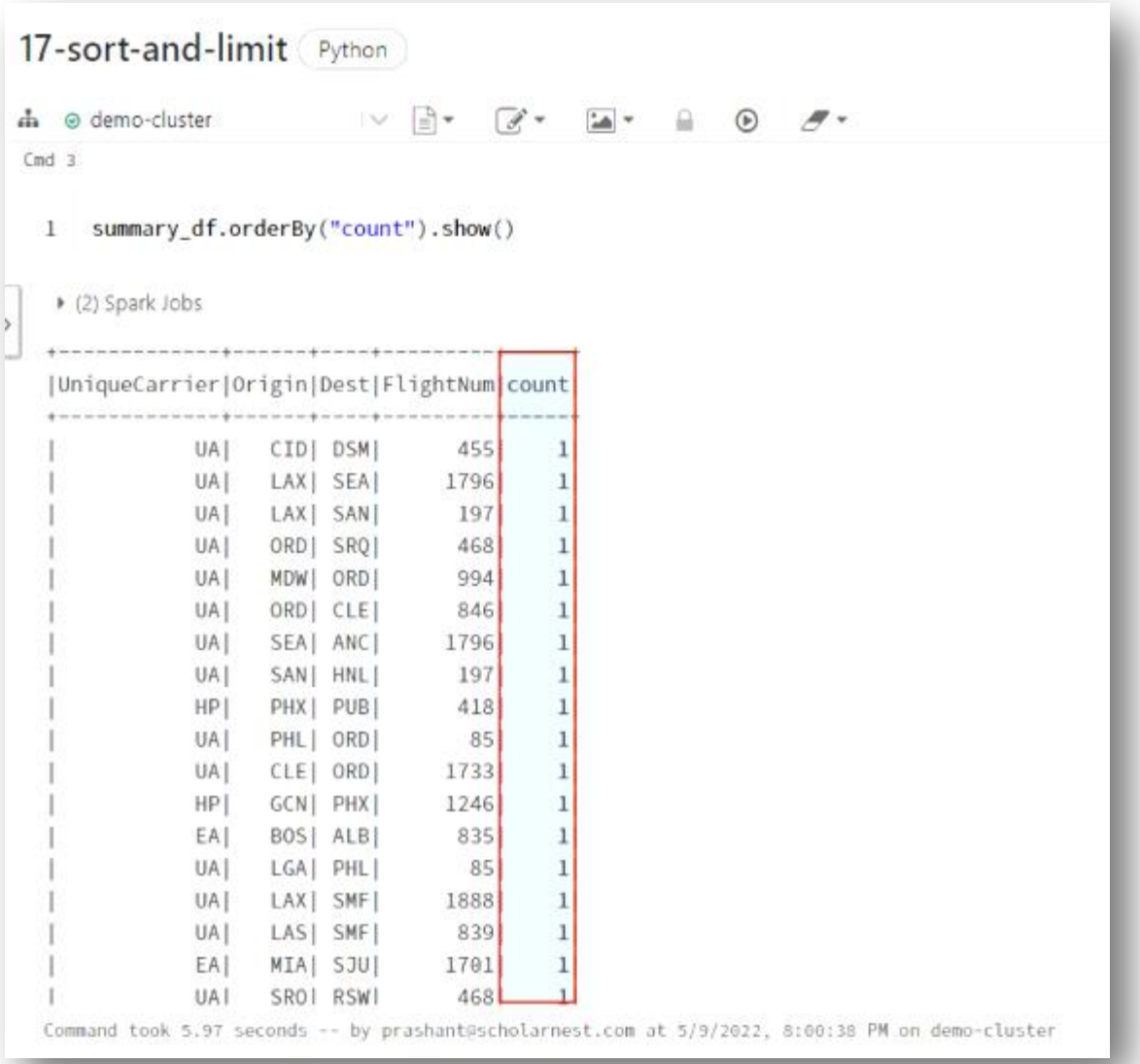
Truncated results, showing first 1000 rows.

Click to re-execute with maximum result limits.

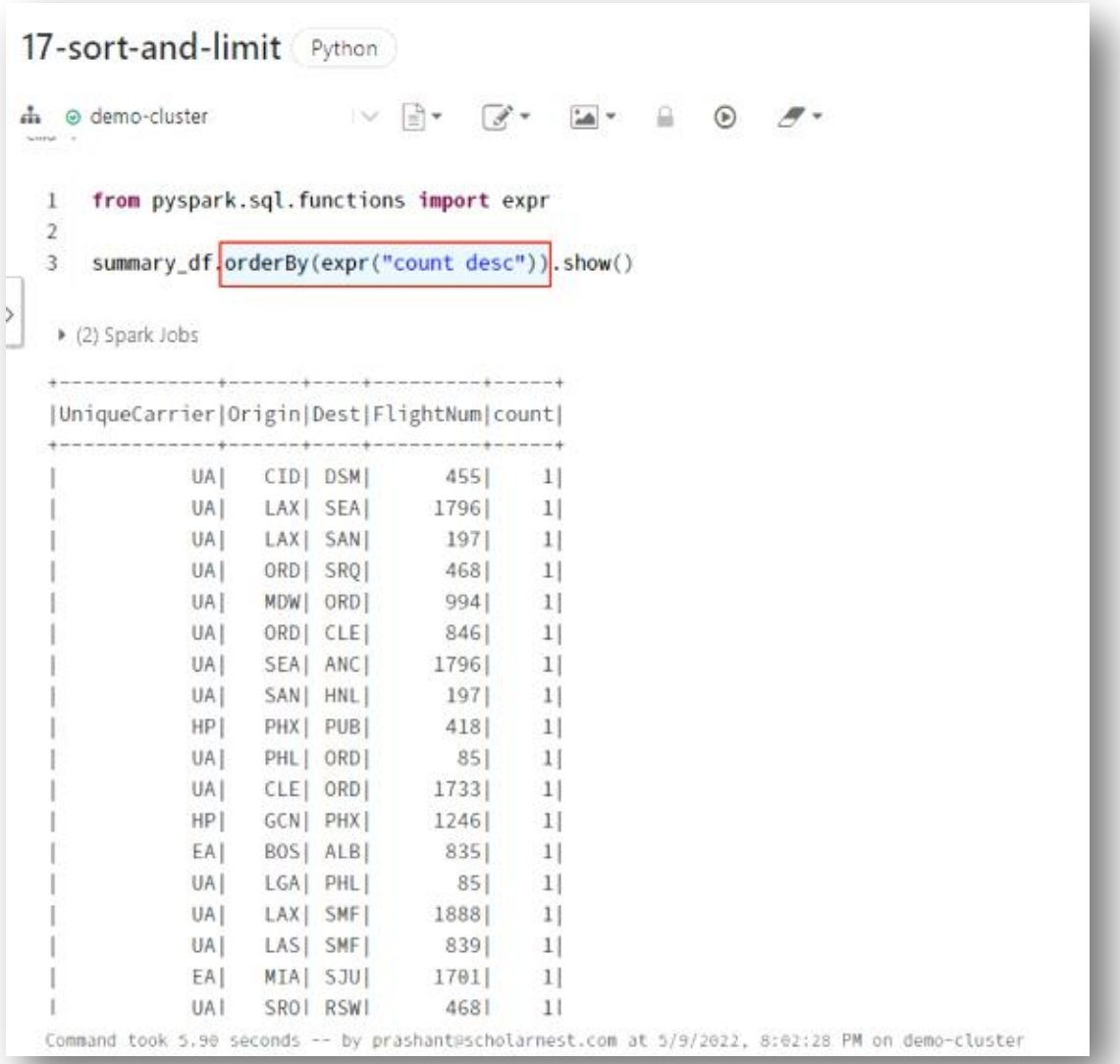
Command took 4.17 seconds -- by prashant@scholarnest.com at 5/9/2022, 7:57:00 PM on demo-cluster

Here is my summary Dataframe shown below. Now let's assume you want to sort this Dataframe over the count. Spark gives you two methods for sorting your Dataframe: sort() and orderBy()
Both the methods are the same. There is no difference, and you can use any one of these. We will be using orderBy() in all our examples. But you can replace the orderBy() call with the sort() method, and the code should work without any problems.

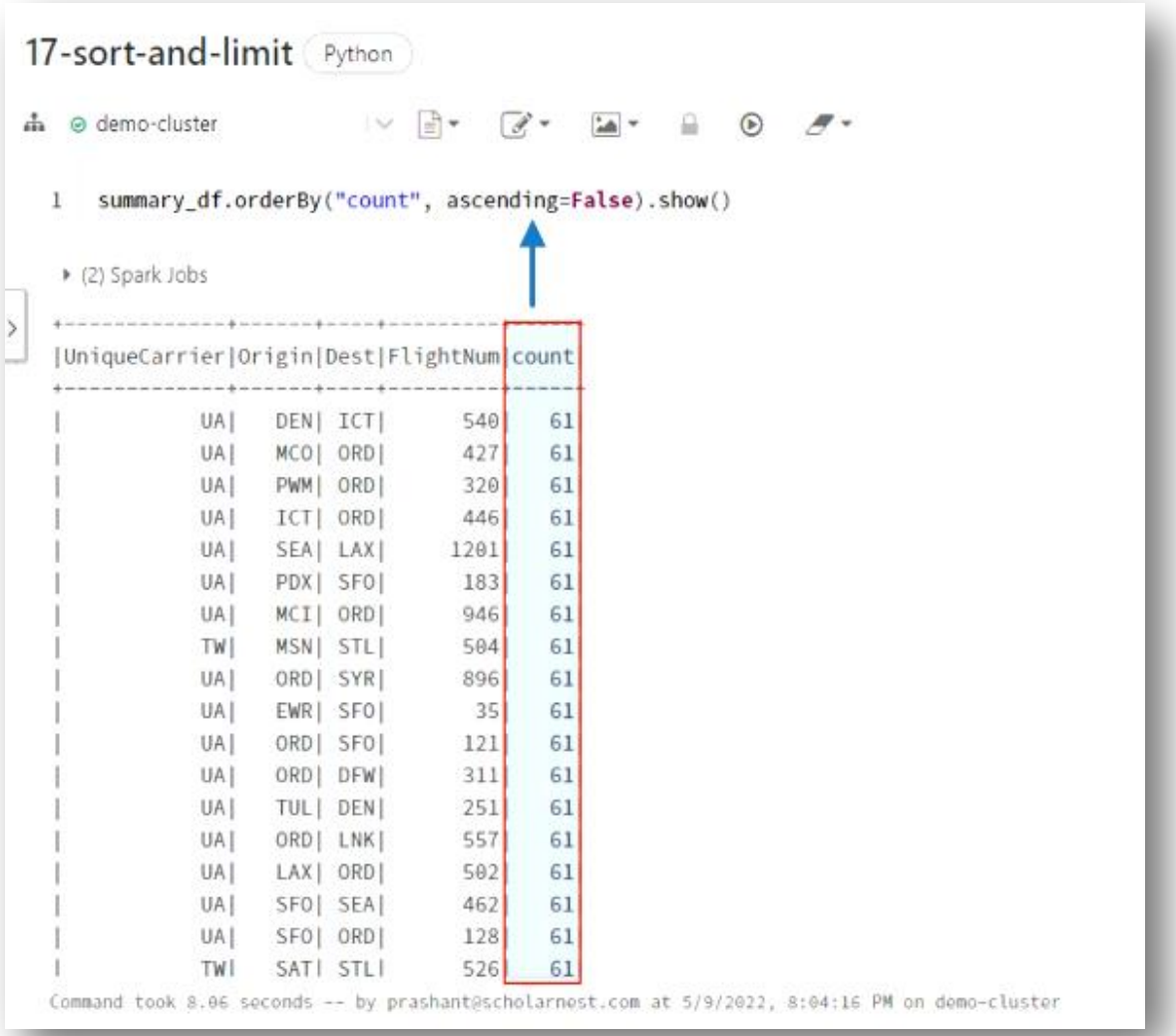
Here is the code to sort the Dataframe on count. And this one is a single-column count. The default order is ascending. So you will see the smallest count first, and the largest value comes in the end.



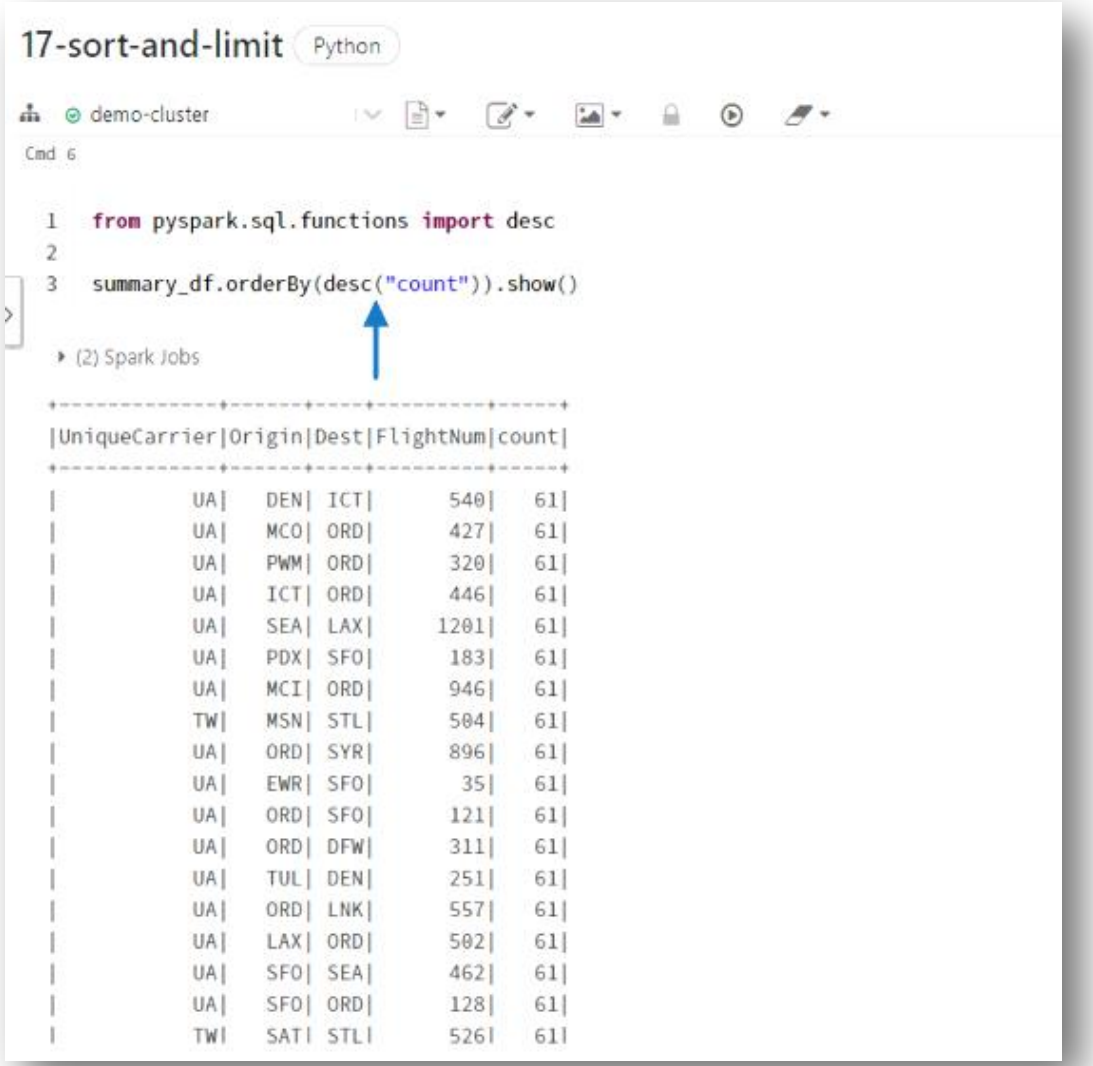
Here I changed the order by column name to a SQL-like expression and added a desc clause, but if you see the output, you will observe that this does not work. So be careful with this common mistake.



The orderBy() method takes another argument for specifying the order. The second argument sets ascending to false, and the orderBy() method starts showing results in descending order.
The default value of ascending is true, so you can ignore the second argument if you want to sort your result in ascending order.



So you can import the desc() function and apply it to the column. This is another way to sort the order in descending order.



We have another alternative to sort the columns in descending order.
We can import the col() method and apply it to the column name to convert it into a column object.
Once we have a column object, we can apply column APIs.

17-sort-and-limit Python

demo-cluster

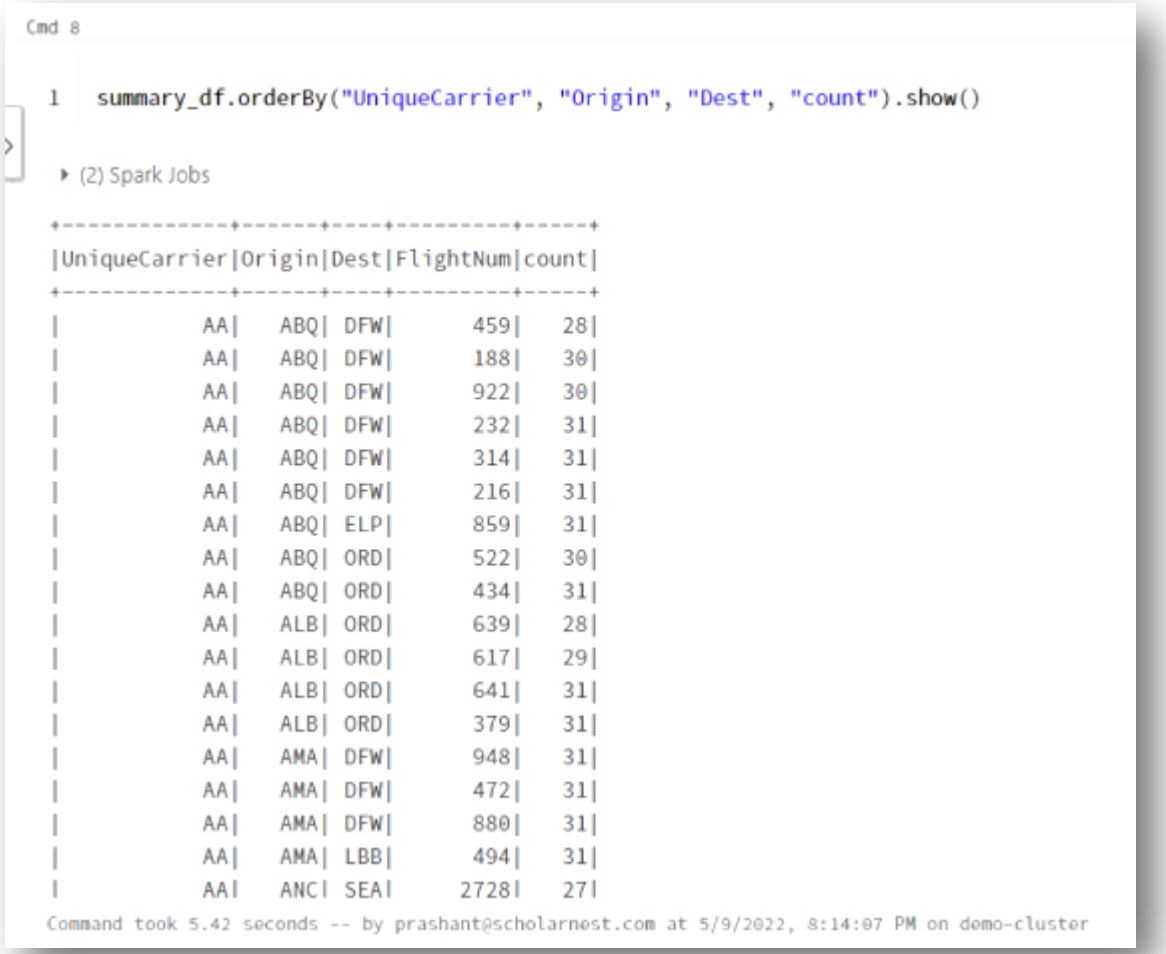
Cmd 7

```
1  from pyspark.sql.functions import col
2
3  summary_df.orderBy(col("count").desc()).show()
```

(2) Spark Jobs

```
+-------------+------+----+---------+-----+
|UniqueCarrier|Origin|Dest|FlightNum|count|
+-------------+------+----+---------+-----+
|           UA|   DEN| ICT|      540|   61|
|           UA|   MCO| ORD|      427|   61|
|           UA|   PWM| ORD|      320|   61|
|           UA|   ICT| ORD|      446|   61|
|           UA|   SEA| LAX|     1201|   61|
|           UA|   PDX| SFO|      183|   61|
|           UA|   MCI| ORD|      946|   61|
|           TW|   MSN| STL|      504|   61|
|           UA|   ORD| SYR|      896|   61|
|           UA|   EWR| SFO|       35|   61|
|           UA|   ORD| SFO|      121|   61|
|           UA|   ORD| DFW|      311|   61|
|           UA|   TUL| DEN|      251|   61|
|           UA|   ORD| LNK|      557|   61|
|           UA|   LAX| ORD|      502|   61|
|           UA|   SFO| SEA|      462|   61|
|           UA|   SFO| ORD|      128|   61|
|           TW|   SAT| STL|      526|   61|
```

Now let us look how to sort multiple columns.
I want to sort the Dataframe by carrier code, origin, destination, and count. By default, all the columns are sorted in ascending order. Inside the same carrier code, it is sorted by the source. Similarly, it is then ordered by the destination code and finally by the count. I didn't sort it by flight number, so the flight number appears random.



I want to keep the first three fields sorted in ascending order, but the count must be in descending order. And you can see the code and output for this requirement shown below.

17-sort-and-limit Python

demo-cluster

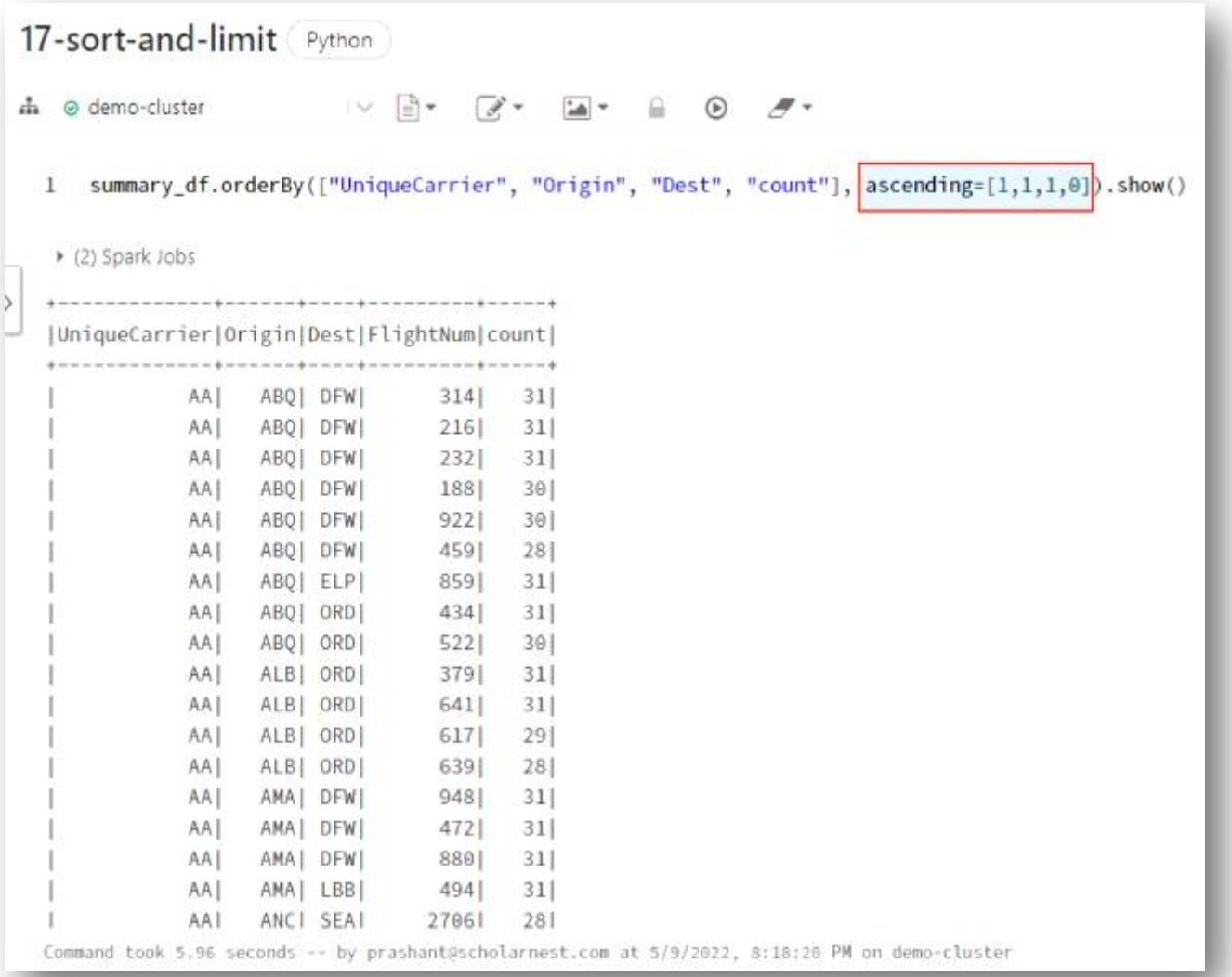
Cmd 8

```
1  from pyspark.sql.functions import desc
2
3  summary_df.orderBy("UniqueCarrier", "Origin", "Dest", desc("count")).show()
```

(2) Spark Jobs

```
+-------------+------+----+---------+-----+
|UniqueCarrier|Origin|Dest|FlightNum|count|
+-------------+------+----+---------+-----+
|           AA|   ABQ| DFW|      314|   31|
|           AA|   ABQ| DFW|      216|   31|
|           AA|   ABQ| DFW|      232|   31|
|           AA|   ABQ| DFW|      188|   30|
|           AA|   ABQ| DFW|      922|   30|
|           AA|   ABQ| DFW|      459|   28|
|           AA|   ABQ| ELP|      859|   31|
|           AA|   ABQ| ORD|      434|   31|
|           AA|   ABQ| ORD|      522|   30|
|           AA|   ALB| ORD|      379|   31|
|           AA|   ALB| ORD|      641|   31|
|           AA|   ALB| ORD|      617|   29|
|           AA|   ALB| ORD|      639|   28|
|           AA|   AMA| DFW|      948|   31|
|           AA|   AMA| DFW|      472|   31|
|           AA|   AMA| DFW|      880|   31|
|           AA|   AMA| LBB|      494|   31|
|           AA|   ANC| SEA|     2706|   28|
```

Spark also offers you another method to do it as shown below. We pass a list of sorting columns. Then we pass a list of Boolean values. In this example, the ascending flag for the first three columns is true, and the last one is false. So the orderBy() will sort the Dataframe in ascending order for the first three columns, and the last one is descending.



In some cases, you may want to see only top N records. Spark allows you to do that using the limit() method.
You can see in the screenshot below that I can limit this Dataframe to return only the top-five results.
So we got only the top five records. Limit is beneficial with single-column sorting requirements.

Cmd 10

```
1  from pyspark.sql.functions import desc
2
3  summary_df.orderBy("UniqueCarrier", "Origin", "Dest", desc("count")).limit(5).show()
```

▸ (2) Spark Jobs

```
+-------------+------+----+---------+-----+
|UniqueCarrier|Origin|Dest|FlightNum|count|
+-------------+------+----+---------+-----+
|           AA|   ABQ| DFW|      314|   31|
|           AA|   ABQ| DFW|      216|   31|
|           AA|   ABQ| DFW|      232|   31|
|           AA|   ABQ| DFW|      188|   30|
|           AA|   ABQ| DFW|      922|   30|
+-------------+------+----+---------+-----+
```

Command took 5.82 seconds -- by prashant@scholarnest.com at 5/9/2022, 8:20:26 PM on demo-cluster

Here is another example.
I am grouping the Dataframe records by FlightNum and taking sum(distance). Then I sorted it in descending order to the largest travelled distance comes at the top. Finally, I am limiting the Dataframe to five records. The result of the set of transformations gave me the top five flights that travelled most.

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Nulls in Apache Spark – Part 1

In any data processing system, a null represents the absence of a data item. It indicates that there is no valid data available for this field, or it is unknown.

And that's a big problem for data processing.

You do not know what to do with the null.

So it is critical that you learn the following things:

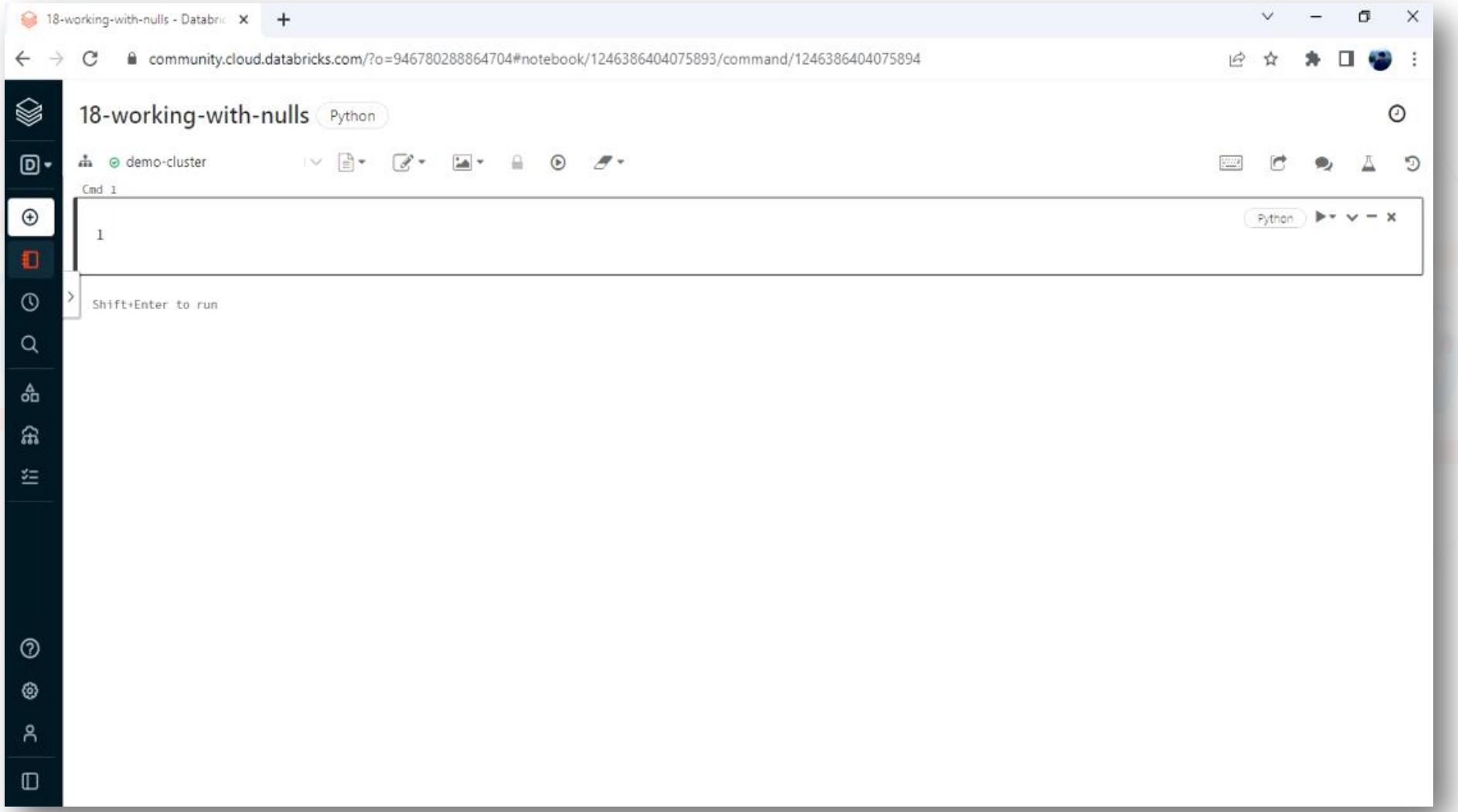
1. What are some common and complex null-related problems?
2. How Spark deals with Nulls?
3. How you can deal with Nulls?

Here are some of the common and complex null-related problems.

What are some common and complex Null problems?

1. Null in the Join Column
2. Null value Data Skew
3. Null in the Partition Column
4. Null in Spark Functions
5. Null in UDFs

Go to your Databricks workspace and create a new notebook.
(Reference : 18-working-with-nulls)



I am creating two Dataframes. The first one is an employee Dataframe, and the second data frame is for departments. The employee Dataframe has got some nulls in the department column. Now I want to join these two data frames using the department column.

18-working-with-nulls Python

demo-cluster

Cmd 1

```
1  data_list1 = [(100, "Prashant", "Software"),
2                (101, "David", None),
3                (102, "Sushant", None),
4                (103, "Abdul", "Account"),
5                (104, "Shruti", "Software")]
6
7  data_list2 = [(501, "Software"),
8                (502, "Account")]
9
10 employee_df = spark.createDataFrame(data_list1).toDF("id", "first_name", "department")
11 department_df = spark.createDataFrame(data_list2).toDF("id", "department")
```

Command took 0.40 seconds -- by prashant@scholarnest.com at 5/18/2022, 11:34:22 AM on demo-cluster

I am starting with the employee Dataframe and joining it with department_df using the department field. This expression uses default inner join. You can see the result highlighted below. You do not see any null records. The records for David and Sushant from the employee Dataframe are left aside. They didn't participate in the inner Join.

The join operation in spark is a complex operation. It distributes data across the cluster over the network, and the distribution process is known as shuffle. But for now, you can assume that the shuffle is a time-consuming process. Most of the join operations in Spark suffers from performance problem, and the reason is a high volume of shuffle records.

In this example, we had two null records. They are not going to be part of the join results. But they will participate in the shuffle and slow down your join operation. The records with null values in the join column may not participate in the join operation. But they consume resources and slow down your join operation. If you eliminate these null records before the Join, you can tune your join operation to work faster.

18-working-with-nulls Python

demo-cluster

Cmd 1

```
1  data_list1 = [(100, "Prashant", "Software"),
2                (101, "David", None),
3                (102, "Sushant", None),
4                (103, "Abdul", "Account"),
5                (104, "Shruti", "Software")]
6
7  data_list2 = [(501, "Software"),
8                (502, "Account")]
9
10 employee_df = spark.createDataFrame(data_list1).toDF("id", "first_name", "department")
11 department_df = spark.createDataFrame(data_list2).toDF("id", "department")
```

Command took 0.40 seconds -- by prashant@scholarnest.com at 5/18/2022, 11:34:22 AM on demo-cluster

Cmd 2

```
1  employee_df.join(department_df, "department").select("first_name", "department").show()
```

(3) Spark Jobs

```
+----------+----------+
|first_name|department|
+----------+----------+
|     Abdul|   Account|
|  Prashant|  Software|
|    Shruti|  Software|
+----------+----------+
```


You have two tables, as shown below. You want to join these tables using a department.
But how does Join happen?

	id	first_name	department
1	100	Prashant	Software
2	101	David	null
3	102	Sushant	null
4	103	Abdul	Account
5	104	Shruti	Software

	id	department
1	501	Software
2	502	Account

Spark will take both the tables and break them into Join key partitions. So the employee table will break into three partitions, as shown below. One partition for each unique join key.
We have three unique values in the employee table.

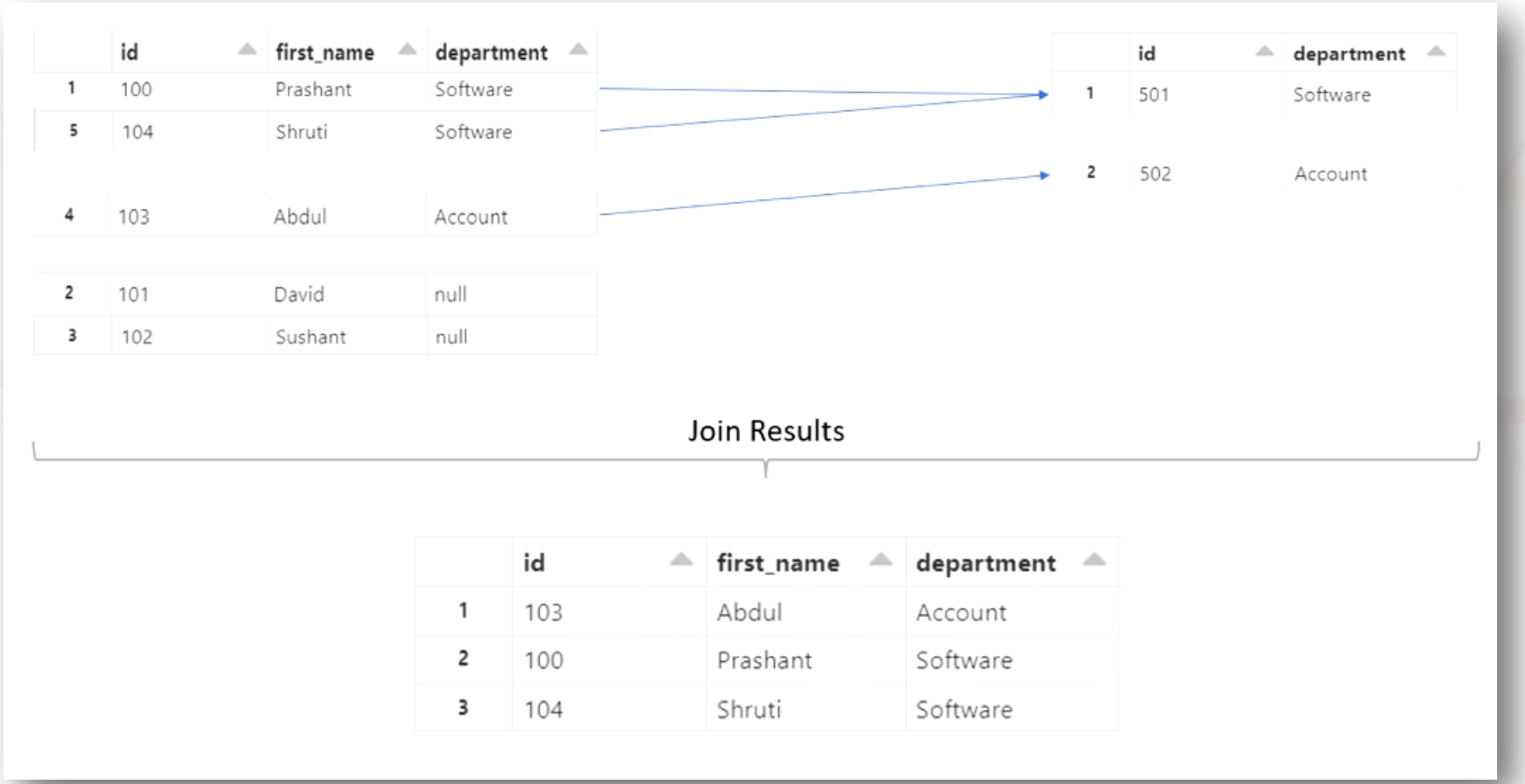
1. Software
2. Account
3. Null

So the employee table will create three partitions.
Similarly, the department table will create two partitions because we have two unique keys.

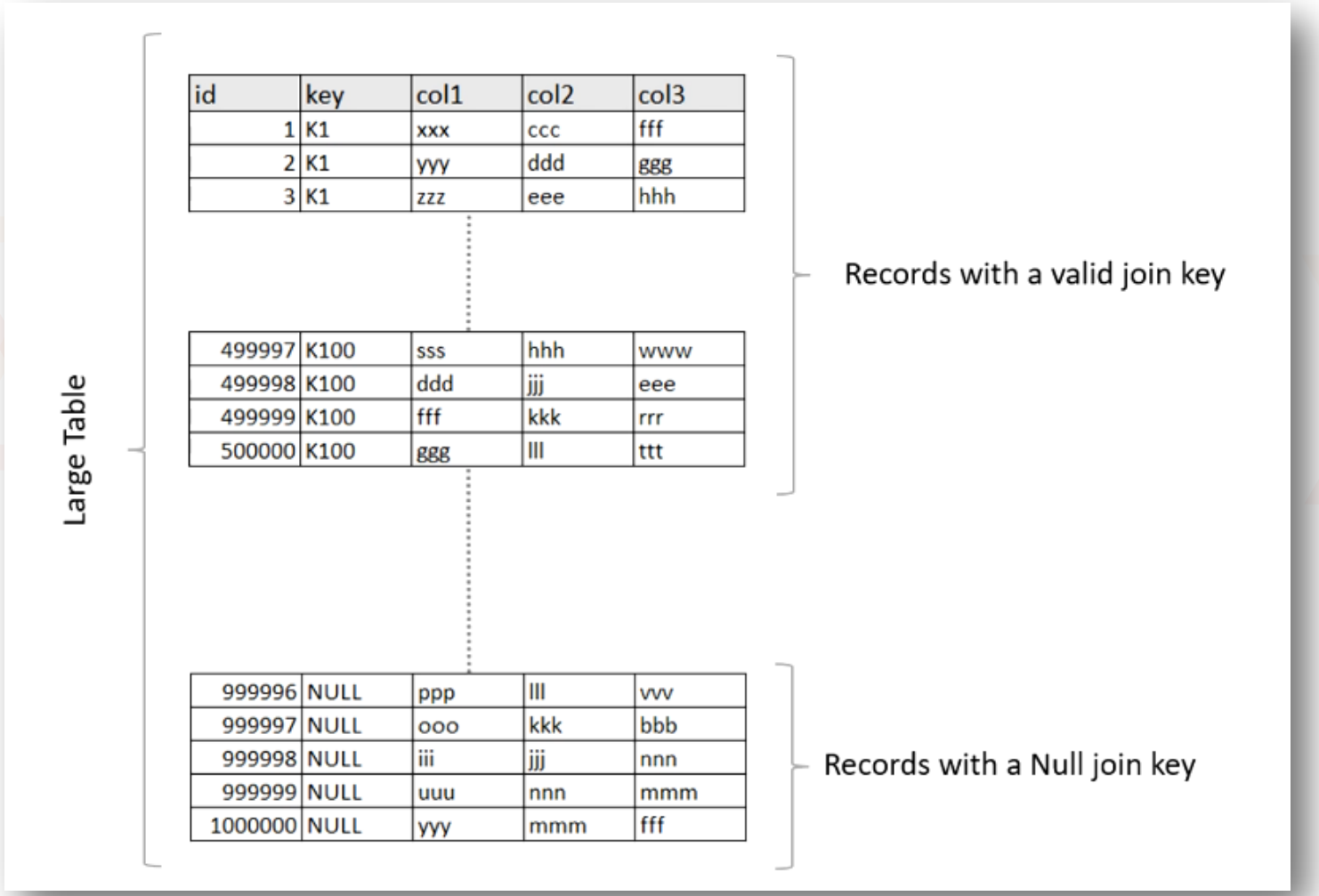
	id	first_name	department
1	100	Prashant	Software
5	104	Shruti	Software
4	103	Abdul	Account
2	101	David	null
3	102	Sushant	null

	id	department
1	501	Software
2	502	Account

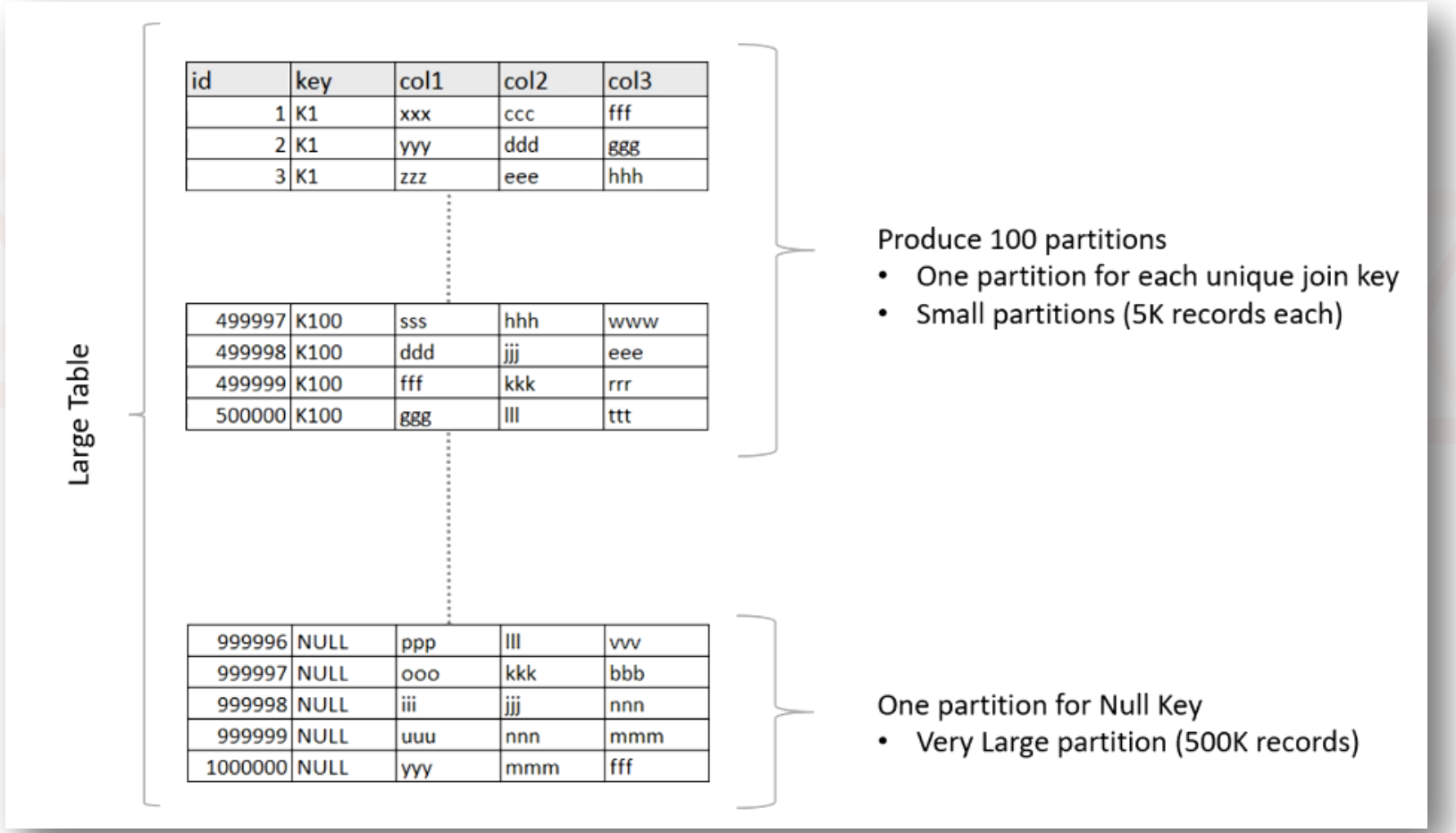
Now, the spark will combine partitions for the same key to apply the Join. So two employee records combine with the software department. The next employee record combines with the account department.
But what about the null department? If you are doing an inner join, this one is left aside, and it does not combine with any partition. But this null partition still participates in the shuffle and consumes memory and CPU.



Now, let's scale the previous scenario. Assume you have a table of one million records. But you have only a hundred unique join keys. Each unique key has approximately 5K records, so you have half a million records with non-null join keys. The remaining half a million records are null.



If you try to join this table with another table using the join key. Spark will split this table into key-wise partitions. You will get 100 partitions for each unique key. And you will also get one partition for all the null values. These hundred partitions will have approximately 5K records in each partition. But the null partition will get 500K records.



This problem is called the data skew problem. Spark is distributed processing system.
It partitions your data on keys, and that happens in many scenarios.
Join operation is one of the most common partitioning scenario.
So whenever you are breaking your table into key-based partitions, you have a danger of creating a data skew.
In the data skew situation, all other partitions are of small size, but one or more partitions are disproportionate and very large compared to others.
These skewed partitions are massive performance problems in Spark.
Due to these skewed partitions, your application may hang or even fail with an out-of-memory error.
And the null values may easily cause skewed partitions.
Null values in the join key are a massive performance risk.
So you must be aware of this problem and take care of it in your code.

A similar problem happens when you want to write partitioned data.
Suppose you want to create a spark table and partition it by a location.
However, the location column is null for 30% of your records.
What will happen?
You will create a table with a null partition having 30% of the records in a single partition.
That's another data skew problem.
So, Null values are often a reason for data skew when partitioning is used on the null value column.

We also have other null problems.
Many machine-learning Spark APIs do not handle null values, and they throw an exception if they see nulls in the Dataframe column.
For example, VectorAssembler is used by data scientists for combining features into a single vector and applying logistic regression and decision trees.
However, the VectorAssembler throws an exception if you have nulls in the Dataframe.
We are not learning machine learning here. However, I wanted to point out that some Spark functions may not handle null columns and throw an exception.
So you should know how to handle nulls.

Another common source of the null problem is associated with the user-defined functions.
Spark allows you to create user-defined functions, and you can use those functions in Dataframe and Spark SQL.
But when you code a user-defined function, you must ensure that your UDF handles null values appropriately.
I often see UDFs throwing exceptions when you pass null values to them.

Now let us see how Spark deals with Nulls.
Here is a Person Dataframe. We have some records where the age is null.

18-working-with-nulls Python

demo-cluster

Command took 3.62 seconds -- by prashant@scholarnest.com at 5/18/2022, 11:35:22 AM on demo-cluster

Cmd 3

```
1  person_list = [(100, "Prashant", 30),
2                 (101, "David", None),
3                 (102, "Sushant", None),
4                 (103, "Abdul", 45),
5                 (104, "Shruti", 28)]
6
7  person_df = spark.createDataFrame(person_list).toDF("id", "name", "age")
8  display(person_df)
```

(3) Spark Jobs

	id	name	age
1	100	Prashant	30
2	101	David	null
3	102	Sushant	null
4	103	Abdul	45
5	104	Shruti	28

Showing all 5 rows.

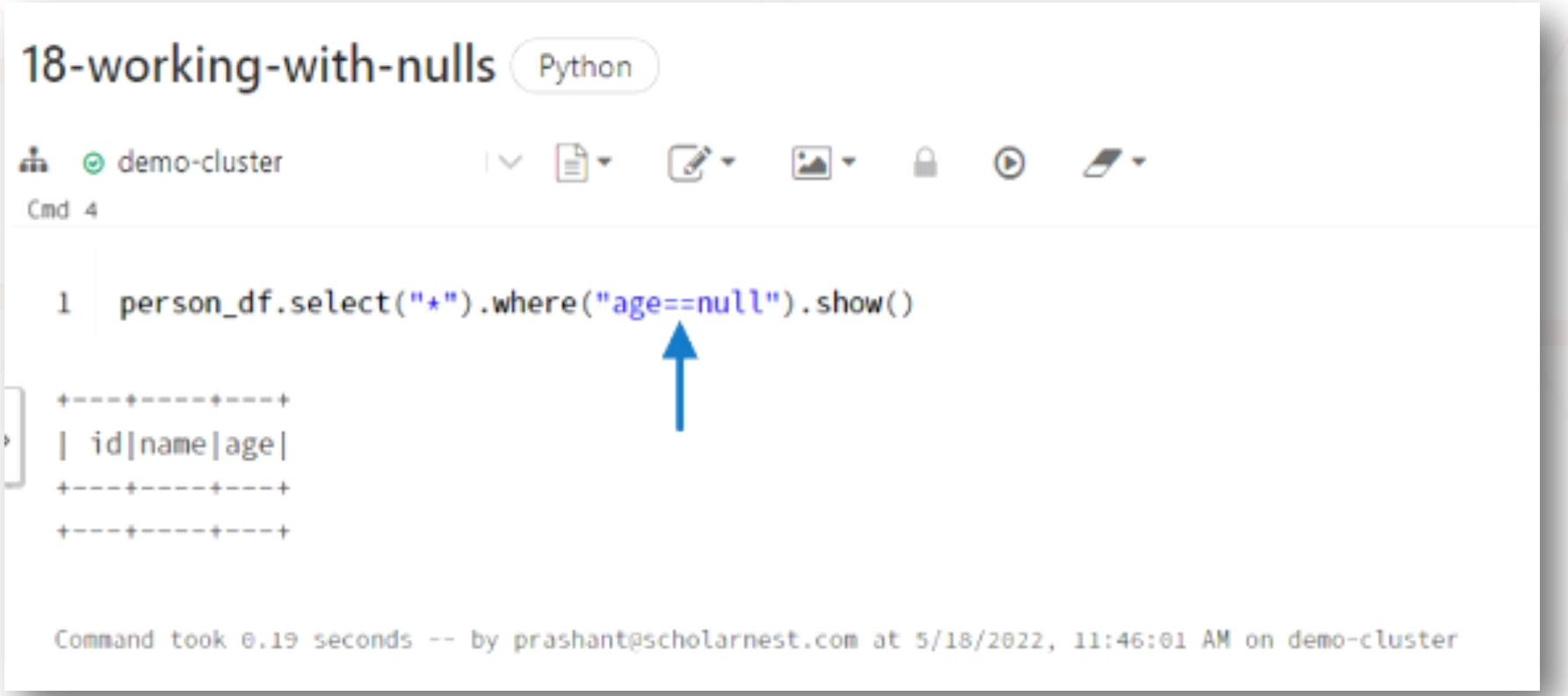
Command took 0.91 seconds -- by prashant@scholarnest.com at 5/18/2022, 11:44:52 AM on demo-cluster

Now I want to tell you the following rules:

1. Null equality is tested using the "is null" operator
2. Comparison operator on null value is also a null.
3. Logical AND operator on a null value is also a null
4. Spark aggregations ignore the null value records
5. groupBy() will group all the null values in one bucket

Let us see some examples to understand these rules.

I want to select all records where the age equals null. I tried using the code below and get the desired output, but it didn’t work. I was expecting two records.
But I didn't get anything.
Why? Because null equality is tested using the "is null" operator.



Now you see two records.
So remember to use the "is null" operator if you want to test something equal to null. The == operator doesn't work with nulls.

You can see it here that comparison operator on null value is also a null.
I compared age > 29, and it returns true or false for non-null columns.
But the result of a comparison is null for the null value column.

Now, we want to select all the records where the age is greater than 29.
We have the following code to achieve this requirement. This expression will be true for only two records. So you will see only two records in the result. The other three records are not included because either the value is false or it is null.

Cmd 6

```
1  person_df.select("*").where("age > 29").show()
```

▸ (3) Spark Jobs

```
+---+--------+---+
| id|    name|age|
+---+--------+---+
|100|Prashant| 30|
|103|   Abdul| 45|
+---+--------+---+
```

Command took 0.72 seconds -- by prashant@scholarnest.com at 5/18/2022, 4:19:42 PM on demo-cluster

Spark handles null values differently. Nulls are special cases in any expression, comparison, grouping, and aggregations. If you know SQL, you already know how databases handle nulls. Spark also deals with nulls in the same way.

I recommend that you go through the following documentation page once and check out some SQL examples of how spark treats nulls. *(Reference - https://spark.apache.org/docs/latest/sql-ref-null-semantics.html)*

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Nulls in Apache Spark – Part 2

In the previous session, we discussed about the following problems in Nulls, and also discussed the first and second problem in detail:

1. What are some common and complex null-related problems?
2. How Spark deals with Nulls?
3. How you can deal with Nulls?

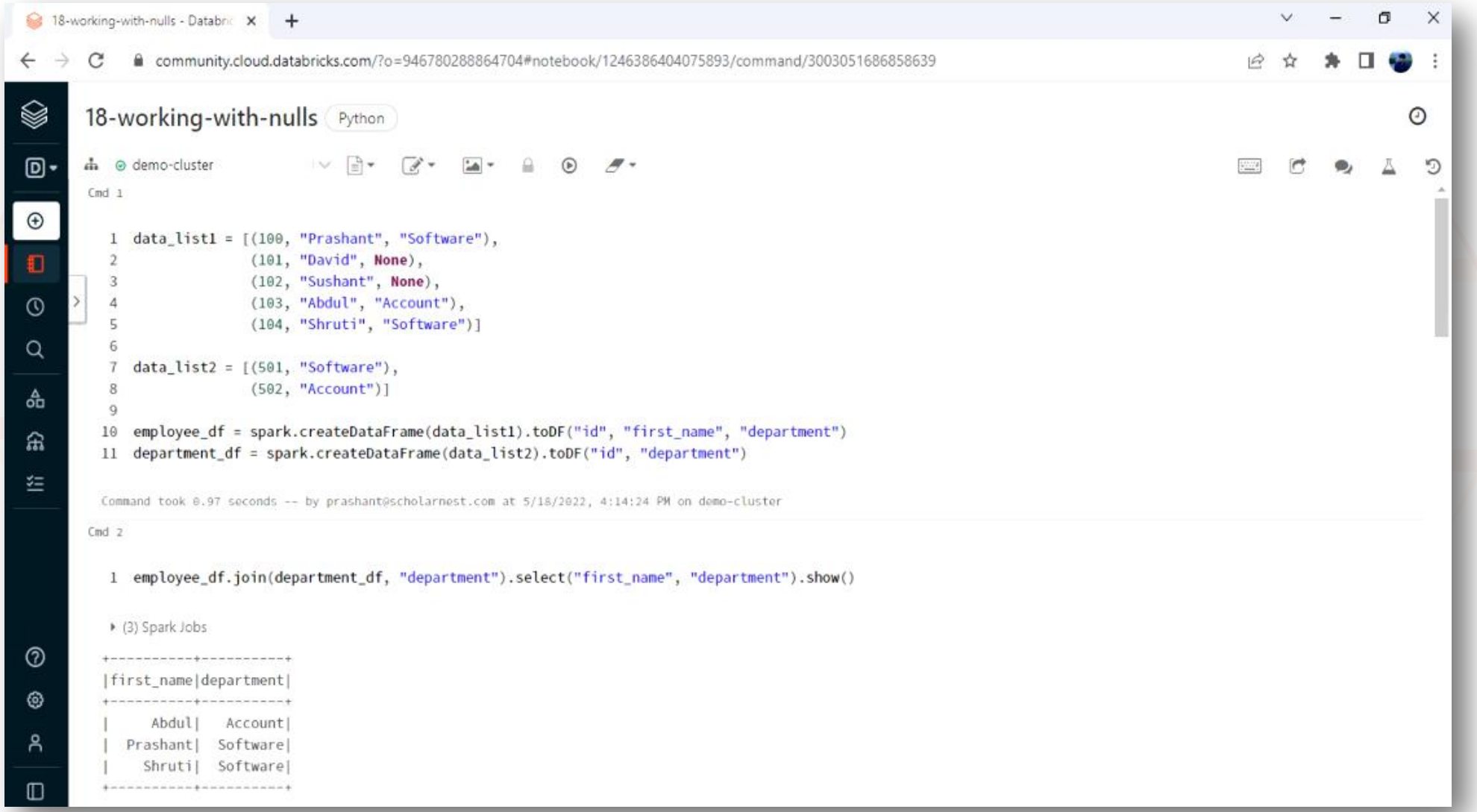
How can we deal with Nulls?
You have three options:

1. Ignore them
2. Remove the null records
3. Fill the null values with some alternative or the best possible value

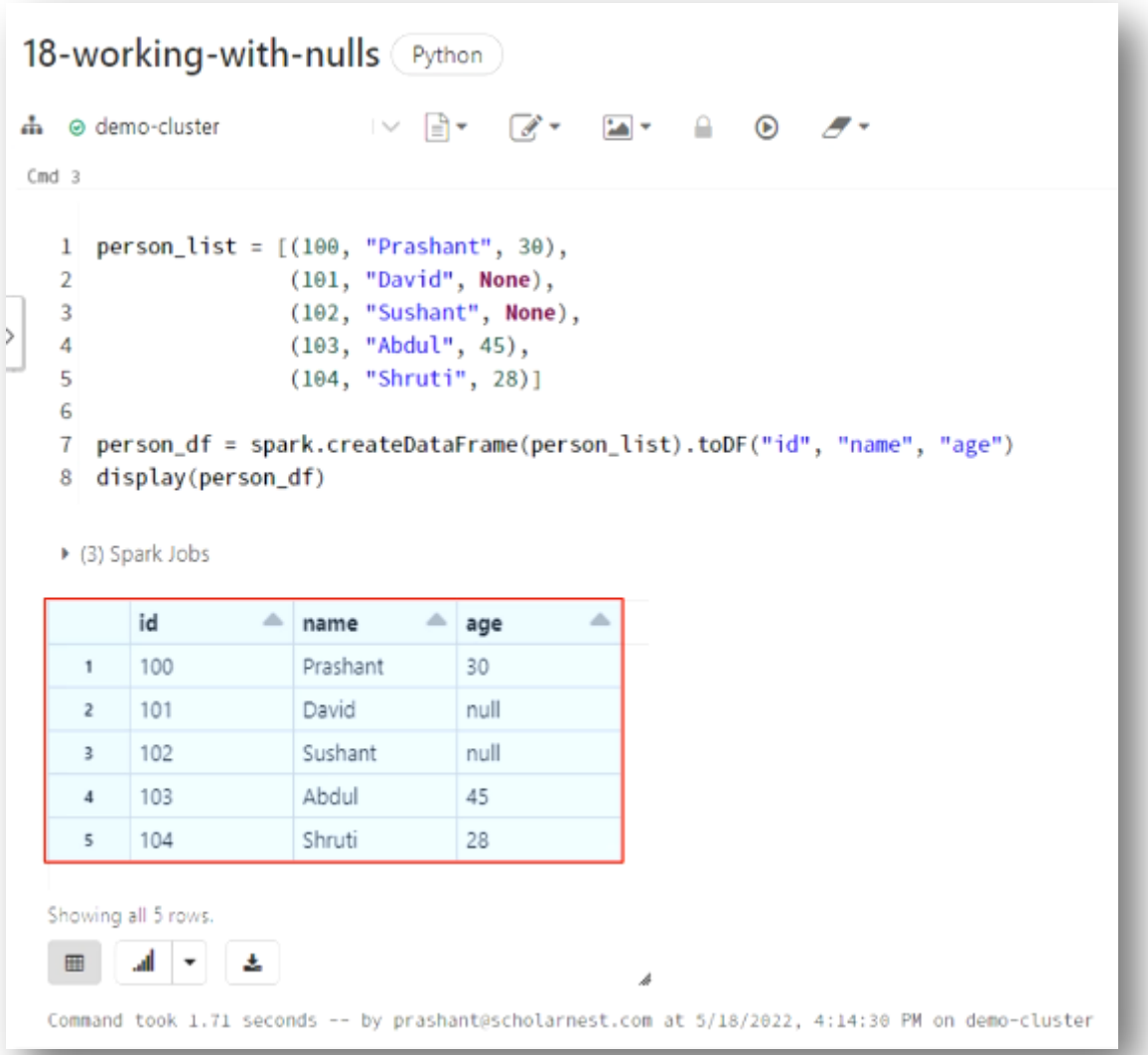
None of the options are good choices for data analysis.
However, we cannot do anything else, so you must choose one of the three.

The default action is to ignore them and not worry at all until they cause a problem.
Many people take the ignorance approach, and it works up to some extent.
Why? Why does it work? Because most of the spark transformations and functions are designed to deal with nulls in a particular way.
For example, aggregations such as sum(), avg(), min(), max(), etc., leaves the null values aside and do not take them into account.
These functions treat the null records like they do not exist.

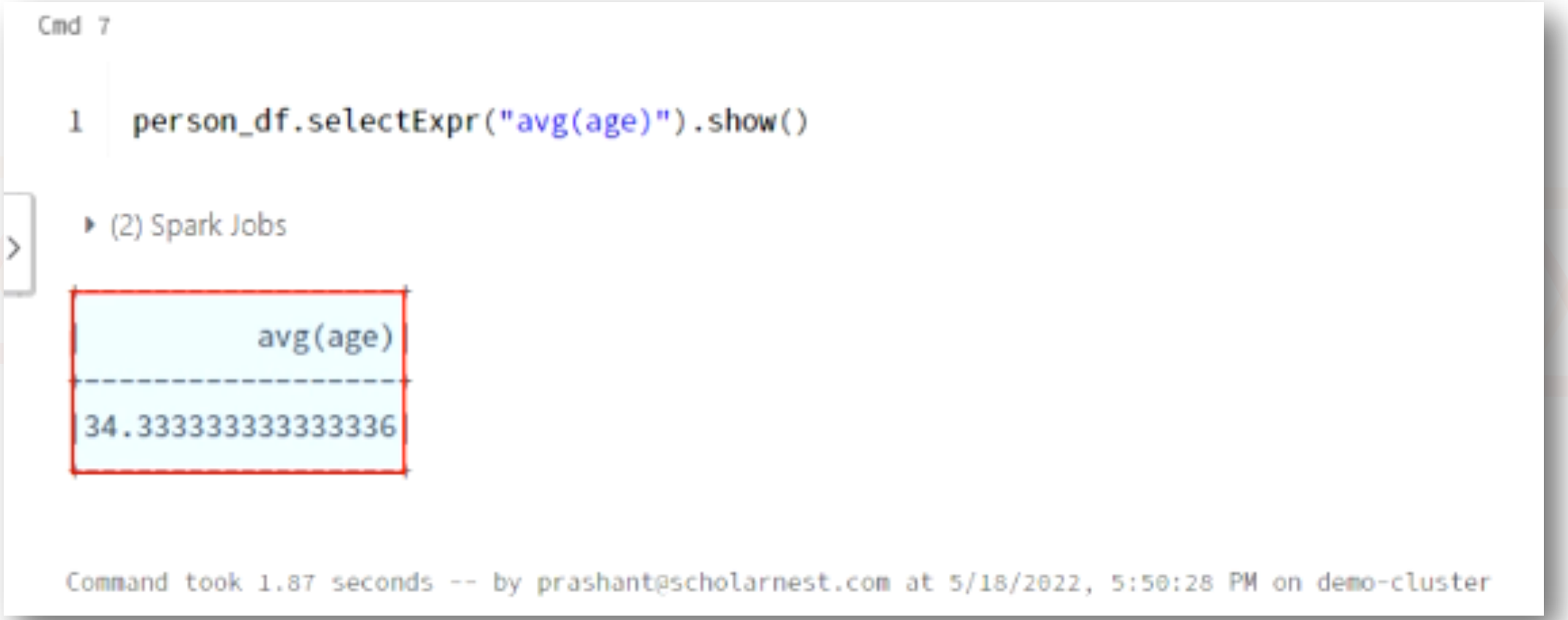
Go to your data bricks workspace and open the notebook from the previous lesson. **(Reference : 18-working-with-nulls)**



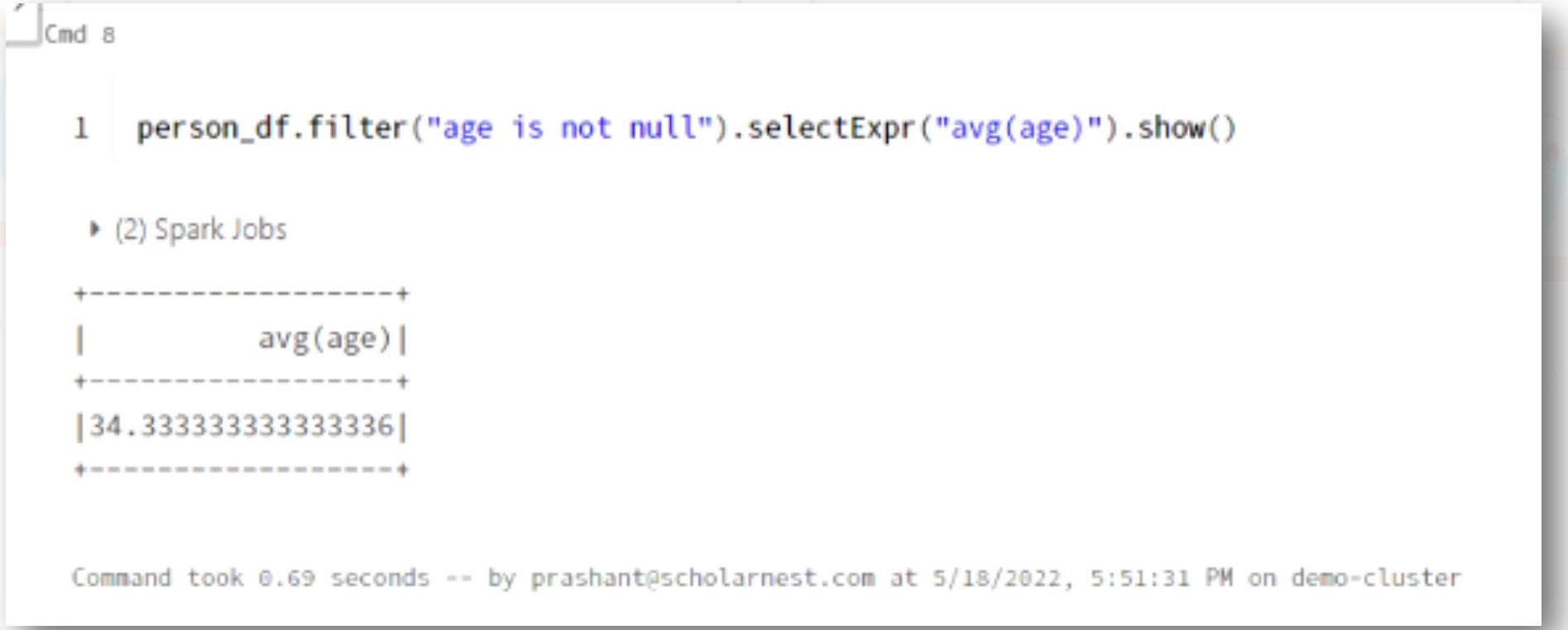
We have this person's Dataframe.
Now let me calculate the average age from the Dataframe.



The average age is coming to 34.33.



Now I am removing the null records and then calculating the average age again, and you can see that it is the same as earlier.
So, you can keep the null records in your Dataframe or remove the null records. The avg() function returns the same value in both cases. Why? Because the aggregation function in the Dataframe is designed to correct your ignorance and remove the null value records from the computation. And that is why ignoring null works in a lot of cases.



As discussed earlier, you have three options to work with nulls.

1. Ignore
2. Remove
3. Fill

The ignore will take default implicit action according to the functions and expressions.

But that is not a best practice. You must explicitly remove or fill the null values to avoid unknown problems.

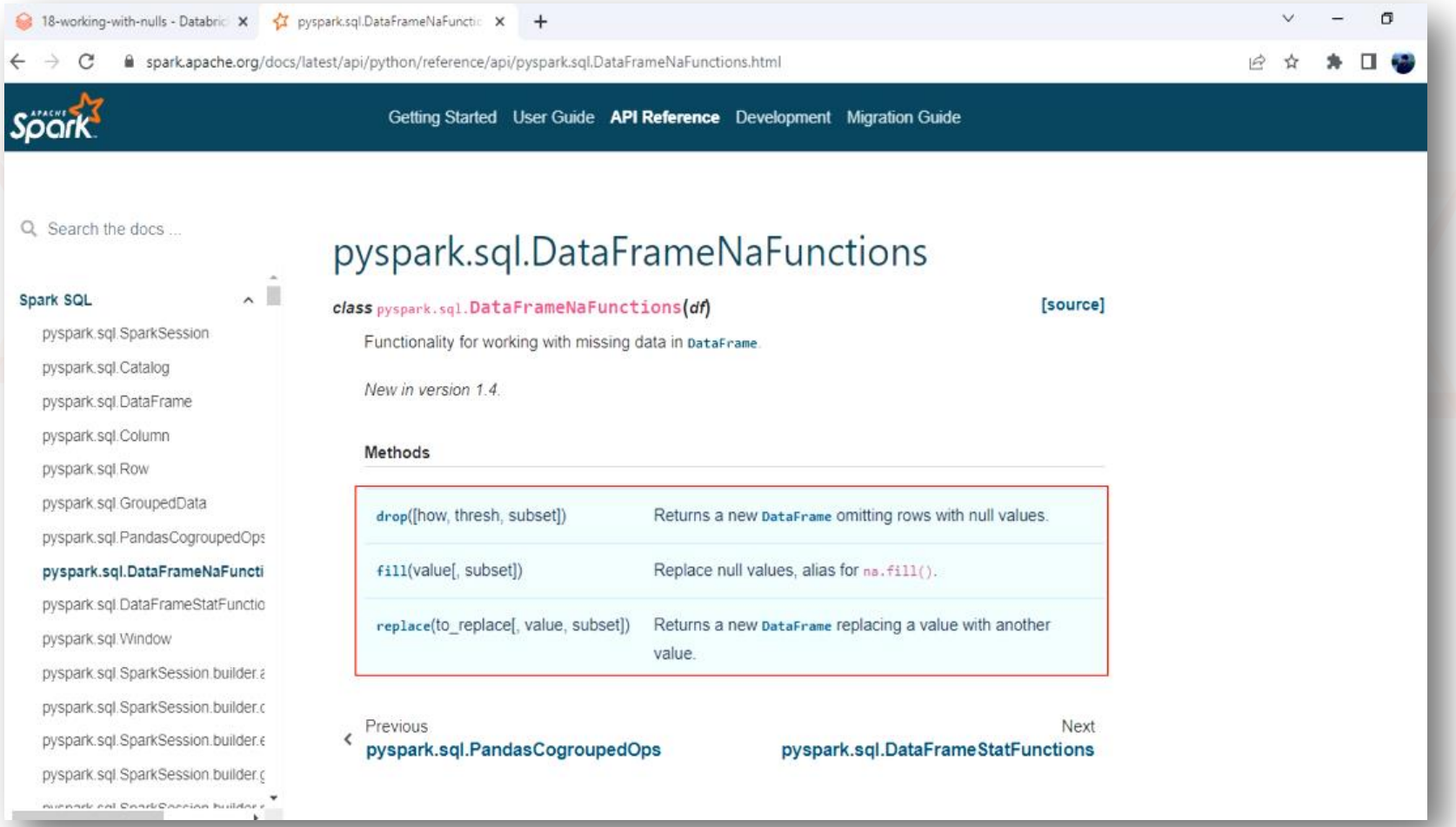
Now let's focus on how to remove or fill the null values. Let's start with the removal.
We have two approaches:

1. filter()
2. na.drop()

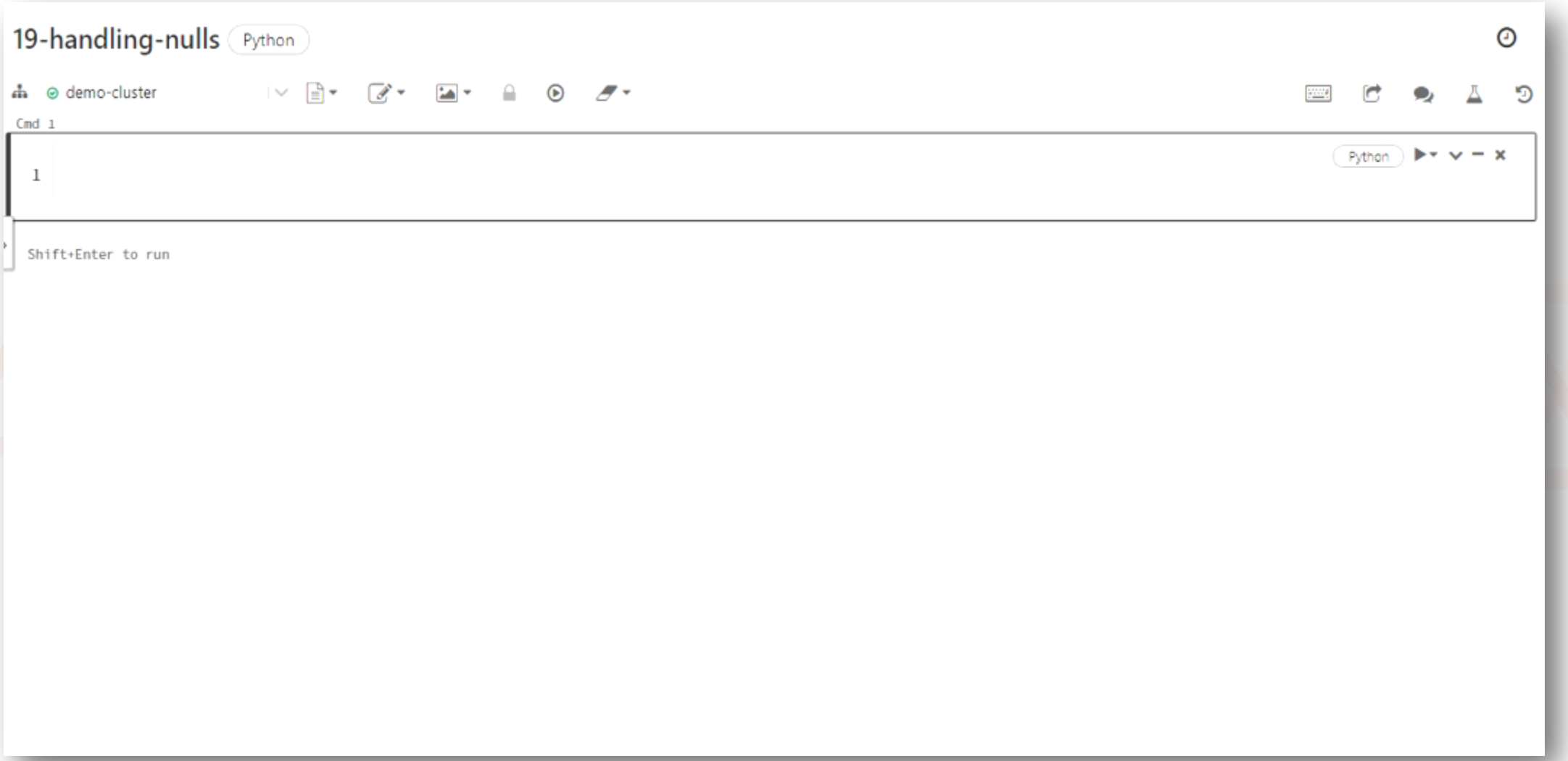
I used a filter condition on the person data frame and took only those records where age is not null. So you can filter out null column records. However, we have more flexible options to drop the null records. The drop options are available via the NA package.

Go to Apache Spark API reference page and search for pyspark.sql.DataFrameNaFunctions, and you will see the below page. *(Reference : https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrameNaFunctions.html)*

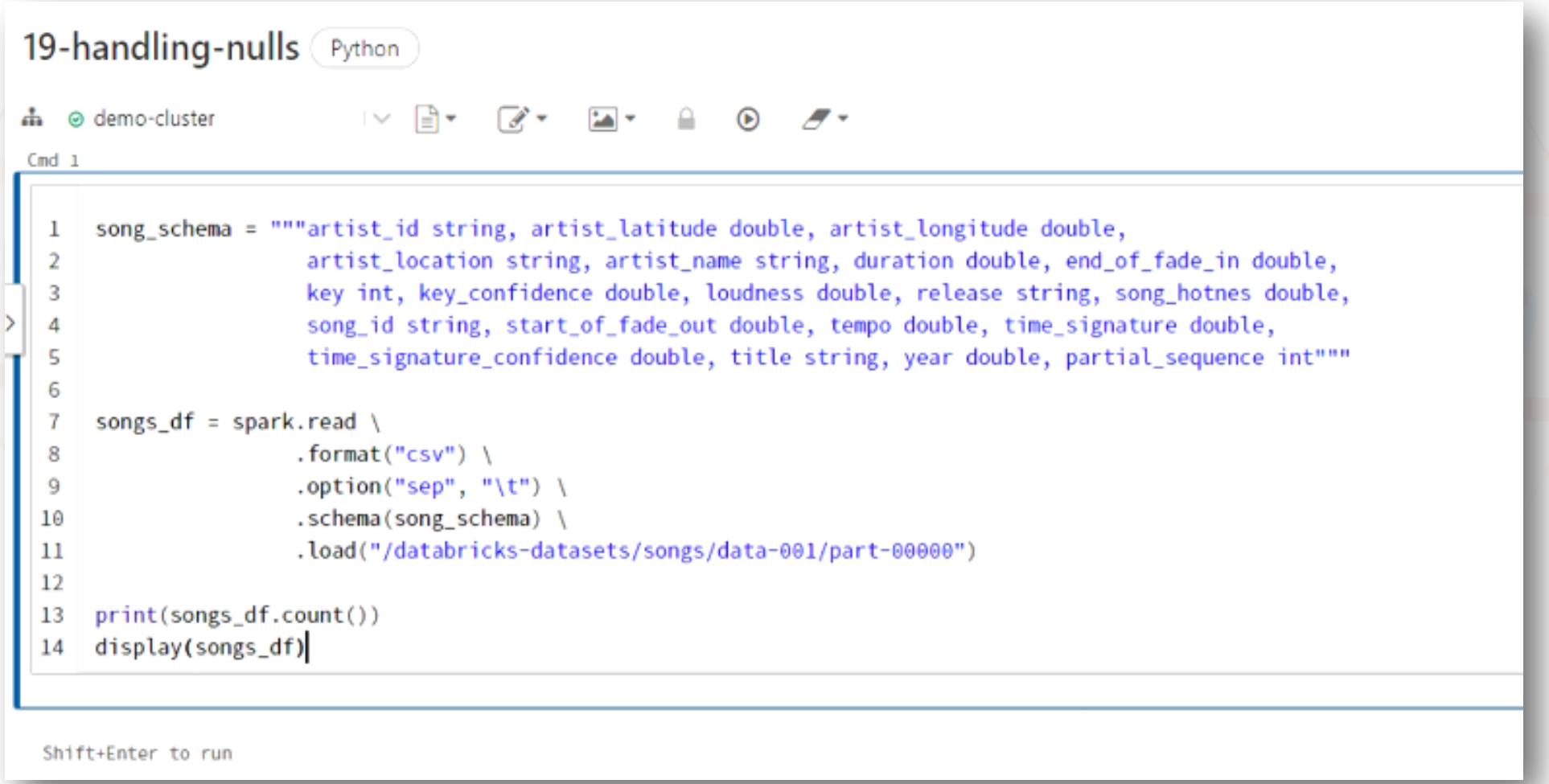
You will see three functions highlighted below. Now let us use these functions.



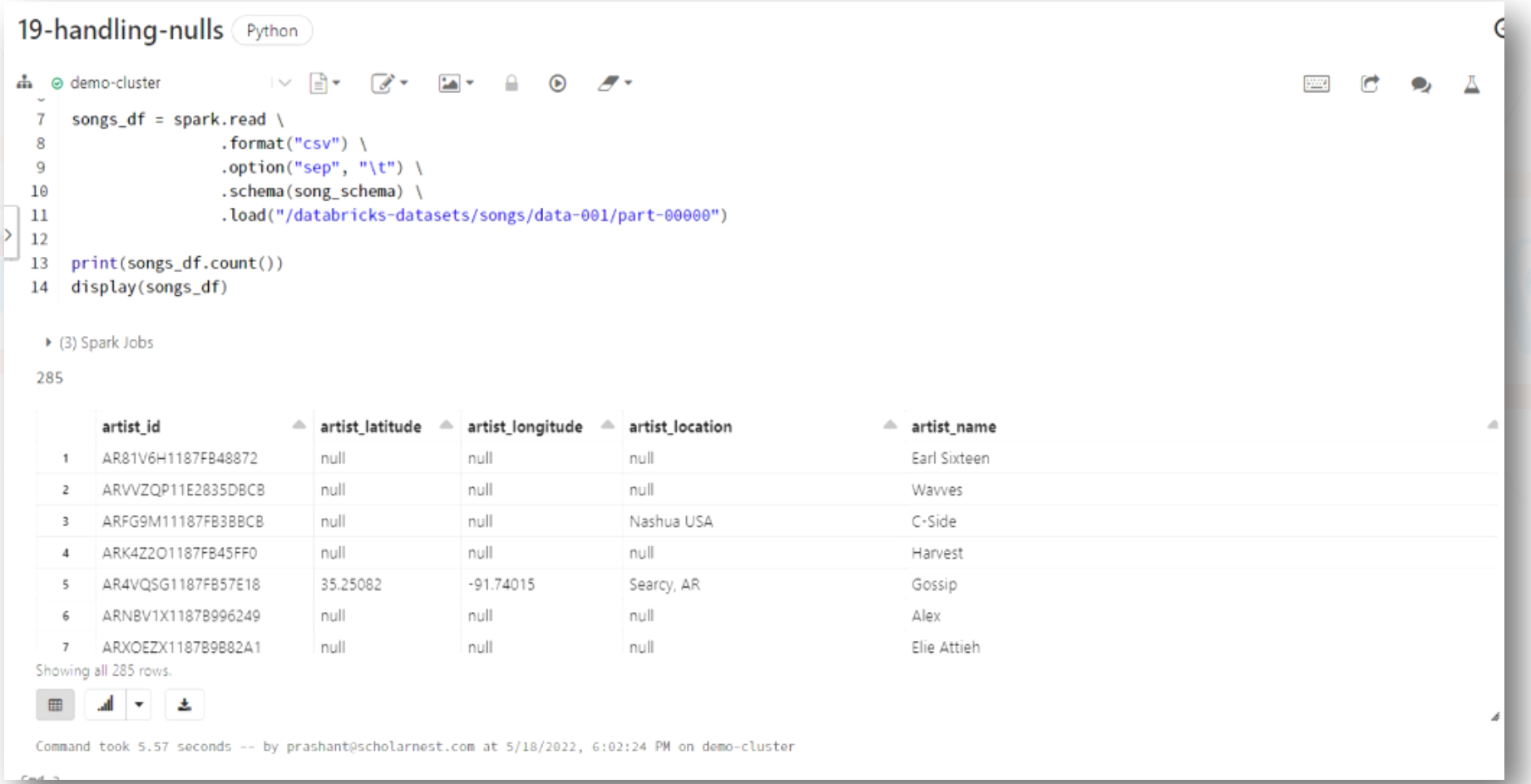
I will create a new notebook to explore these functions **(Reference : 19.handling-nulls)**



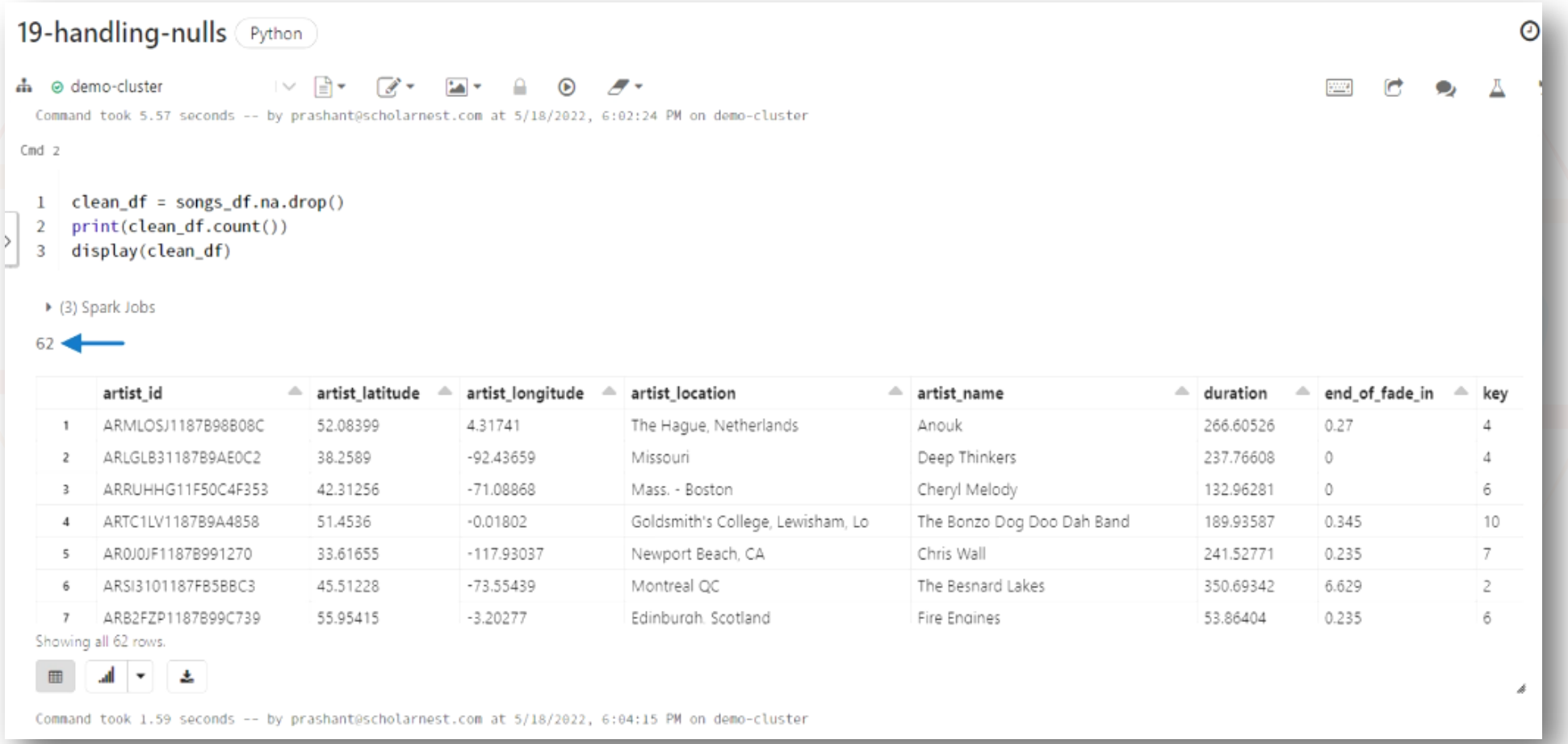
I have loaded a sample dataset and created a Dataframe. I am loading a CSV data file. In fact, the data file is tab-separated. So it is not a CSV file but a TSV file. However, Spark supports CSV, TSV, and similar files under the CSV format. So you use the CSV format and set an option for the separator. I am loading a tab-separated file. The format is CSV, and the separator is a tab character.



Here is the output Dataframe.
We have 285 records. If you look at the data you can see we have lots of nulls. The artist_latitude and longitude values are null for many records. We want to remove these records from the Dataframe. If you try using the filter method, it would be a difficult task as it would require a long filter condition. I mean, filter where artist_latitude is not null, then filter where artist_longitude is not null. Similarly, you may have to add multiple filter conditions.



The NA Functions are available with the dot NA notation. The drop() function removes all the rows where any column has a null value. So, the new Dataframe has got only 62 rows. We removed all the rows where any of the columns are null.



	artist_id	artist_latitude	artist_longitude	artist_location	artist_name	duration	end_of_fade_in	key
1	ARMLOSJ1187B98B08C	52.08399	4.31741	The Hague, Netherlands	Anouk	266.60526	0.27	4
2	ARLGLB31187B9AE0C2	38.2589	-92.43659	Missouri	Deep Thinkers	237.76608	0	4
3	ARRUHHG11F50C4F353	42.31256	-71.08868	Mass. - Boston	Cheryl Melody	132.96281	0	6
4	ARTC1LV1187B9A4858	51.4536	-0.01802	Goldsmith's College, Lewisham, Lo	The Bonzo Dog Doo Dah Band	189.93587	0.345	10
5	AR0J0JF1187B991270	33.61655	-117.93037	Newport Beach, CA	Chris Wall	241.52771	0.235	7
6	ARSI3101187FB5BBC3	45.51228	-73.55439	Montreal QC	The Besnard Lakes	350.69342	6.629	2
7	ARB2FZP1187B99C739	55.95415	-3.20277	Edinburgh, Scotland	Fire Engines	53.86404	0.235	6

The drop() takes three arguments.

1. how -> any or all
2. thresh -> integer
3. subset -> list of column names to consider

The "how" can take any or all. The default value is any.
So the drop() function will remove a record if any of the column values are null.
You can try to set it to all, and the drop() function will remove a record only if all the records are null.

I have changed the code and tried it with all values. And you can see that we got 285 records. So nothing is removed because we do not have any record where all the columns are null.

Cmd 2

```
1  clean_df = songs_df.na.drop(how="all")
2  print(clean_df.count())
3  display(clean_df)
```

▸ (3) Spark Jobs

285

	artist_id	artist_latitude	artist_longitude	artist_location	artist_name
1	AR81V6H1187FB48872	null	null	null	Earl Sixteen
2	ARVVZQP11E2835DBCB	null	null	null	Wavves
3	ARFG9M11187FB3BBCB	null	null	Nashua USA	C-Side
4	ARK4Z2O1187FB45FF0	null	null	null	Harvest
5	AR4VQSG1187FB57E18	35.25082	-91.74015	Searcy, AR	Gossip
6	ARNBV1X1187B996249	null	null	null	Alex
7	ARXOEZX1187B9B82A1	null	null	null	Elie Attieh

Showing all 285 rows.

Command took 1.29 seconds -- by prashant@scholarnest.com at 5/18/2022, 6:06:40 PM on demo-cluster

Now in this code we are using subset of columns. This example is to remove a record if all of these given columns are null. You can also change the how parameter to any, and it will remove a record if any of the given three columns are null. So you can combine the arguments as per your requirement. You can see the output that we are left with the 143 records. So you have all and any options. And you can combine it with the list of columns.

Cmd 2

```
1  clean_df = songs_df.na.drop(how="all", subset=['artist_latitude', 'artist_longitude', 'artist_location'])
2  print(clean_df.count())
3  display(clean_df)
```

▸ (3) Spark Jobs

143

	artist_id	artist_latitude	artist_longitude	artist_location	artist_name	duration	end_of_fac
1	ARFG9M11187FB3BBCB	null	null	Nashua USA	C-Side	247.32689	0
2	AR4VQSG1187FB57E18	35.25082	-91.74015	Searcy, AR	Gossip	430.23628	0
3	ARXPUIA1187B9A32F1	null	null	Rome, Italy	Simone Cristicchi	220.00281	2.119
4	ARNPPTH1187B9AD429	51.4855	-0.37196	Heston, Middlesex, England	Jimmy Page	156.86485	0.334
5	ARHAMGQ1187B9B6491	30.66121	-93.89386	Kirbyville, TX	Ivory Joe Hunter	170.39628	0.166
6	ARORFTD1187B99F7D4	null	null	台北, Taiwan	Epitaph	215.95383	5.045
7	AR8UA7Z1187B9AD9C9	48.10751	-1.68447	Rennes	Étienne Daho	303.62077	0

Showing all 143 rows.

Command took 1.33 seconds -- by prashant@scholarnest.com at 5/18/2022, 6:08:16 PM on demo-cluster

We also have a thresh hold option. You can remove the how and use the thresh hold option, and that is what is shown in the screenshot below. So this one says. Look for the given list of columns and remove a record if we have less than two non-null values. The condition is confusing. But it simply means keep the record if we have at least two non-null values amongst these given columns.

Cmd 2

```
1  clean_df = songs_df.na.drop(thresh=2, subset=['artist_latitude', 'artist_longitude', 'artist_location'])
2  print(clean_df.count())
3  display(clean_df)
```

▸ (3) Spark Jobs

103

	artist_id	artist_latitude	artist_longitude	artist_location	artist_name	duration	end_of_fad
1	AR4VQSG1187FB57E18	35.25082	-91.74015	Searcy, AR	Gossip	430.23628	0
2	ARNPPTH1187B9AD429	51.4855	-0.37196	Heston, Middlesex, England	Jimmy Page	156.86485	0.334
3	ARHAMGQ1187B9B6491	30.66121	-93.89386	Kirbyville, TX	Ivory Joe Hunter	170.39628	0.166
4	AR8UA7Z1187B9AD9C9	48.10751	-1.68447	Rennes	Étienne Daho	303.62077	0
5	ARMLOSJ1187B98B08C	52.08399	4.31741	The Hague, Netherlands	Anouk	266.60526	0.27
6	ARLGLB31187B9AE0C2	38.2589	-92.43659	Missouri	Deep Thinkers	237.76608	0
7	ARRUHHG11F50C4F353	42.31256	-71.08868	Mass. - Boston	Cheryl Melody	132.96281	0

Showing all 103 rows.

Command took 0.97 seconds -- by prashant@scholarnest.com at 5/18/2022, 6:09:43 PM on demo-cluster

So we have a super flexible na.drop() function. You can use this function to drop null records explicitly.
However, you do not want to drop your records for having nulls in some scenarios.
Instead, you want to fill those null values and try to repair them.
We have two NA functions for doing that.

1. fill()
2. replace()

Here is an example that fills the artist_latitude and longitude with zeros.
It doesn't make much sense to fill the lat/long values with zeros, but you have an option to fill the nulls with whatever you want.

Cmd 3

```
1  filled_df = songs_df.na.fill({'artist_latitude':0, 'artist_longitude':0})
2  print(filled_df.count())
3  display(filled_df)
```

▸ (3) Spark Jobs

285

	artist_id	artist_latitude	artist_longitude	artist_location	artist_name
1	AR81V6H1187FB48872	0	0	null	Earl Sixteen
2	ARVVZQP11E2835DBCB	0	0	null	Wavves
3	ARFG9M11187FB3BBCB	0	0	Nashua USA	C-Side
4	ARK4Z2O1187FB45FF0	0	0	null	Harvest
5	AR4VQSG1187FB57E18	35.25082	-91.74015	Searcy, AR	Gossip
6	ARNBV1X1187B996249	0	0	null	Alex
7	ARXOEZX1187B9B82A1	0	0	null	Elie Attieh

Showing all 285 rows.

Command took 1.14 seconds -- by prashant@scholarnest.com at 5/18/2022, 6:11:47 PM on demo-cluster

We also have a replace() method. But the replace() method is for a different purpose. Sometimes, you get data that is already filled with some values that are not meaningful.
Here is a base_df shown below. Let's assume you loaded a Dataframe from a CSV file, and the artist_location has got UnKnown values. These are not null values. But it is not meaningful, and they represent a lack of know value. You want to change these values with something else. You cannot use the fill() method because the fill() method fills a null column.

We can use the replace() to change the values as shown below, we look for the UnKnown in the artist location and replace it with the USA.

Cmd 5

```
1  replaced_df = base_df.na.replace({"UnKnown":"USA"}, ["artist_location"])
2  display(replaced_df)
```

▸ (1) Spark Jobs

	artist_id	artist_latitude	artist_longitude	artist_location	artist_name
1	AR81V6H1187FB48872	null	null	USA	Earl Sixteen
2	ARVVZQP11E2835DBCB	null	null	USA	Wavves
3	ARFG9M11187FB3BBCB	null	null	Nashua USA	C-Side
4	ARK4Z2O1187FB45FF0	null	null	USA	Harvest
5	AR4VQSG1187FB57E18	35.25082	-91.74015	Searcy, AR	Gossip
6	ARNBV1X1187B996249	null	null	USA	Alex
7	ARXOEZX1187B9B82A1	null	null	USA	Elie Attieh

Showing all 285 rows.

Command took 0.49 seconds -- by prashant@scholarnest.com at 5/18/2022, 6:16:46 PM on demo-cluster

You have three NA functions:

1. drop()
2. fill()
3. replace()

These functions can help you to deal with the null columns. The drop() allows you to drop the null column records. The fill() is to change the null values with some other values. The replace() is to change null or other similar values. These three functions are super flexible and take 2-3 parameters. You can customize the use of these functions using those parameters and meet most of your requirements. I cannot create and explain all possible combinations and scenarios. However, you should understand the purpose of these functions and try using them when you have appropriate requirements.

Besides these NA functions, you also have some Spark SQL built-in functions that allow you to change the null values.
Here are some examples:

COALESCE(), NULLIF(), IFNULL(), NVL(), NVL2(), ISNULL(), ISNOTNULL()

These are SQL built-in functions.
You can refer to the documentation and learn some details about these functions.
(Reference : https://spark.apache.org/docs/latest/api/sql/index.html)

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

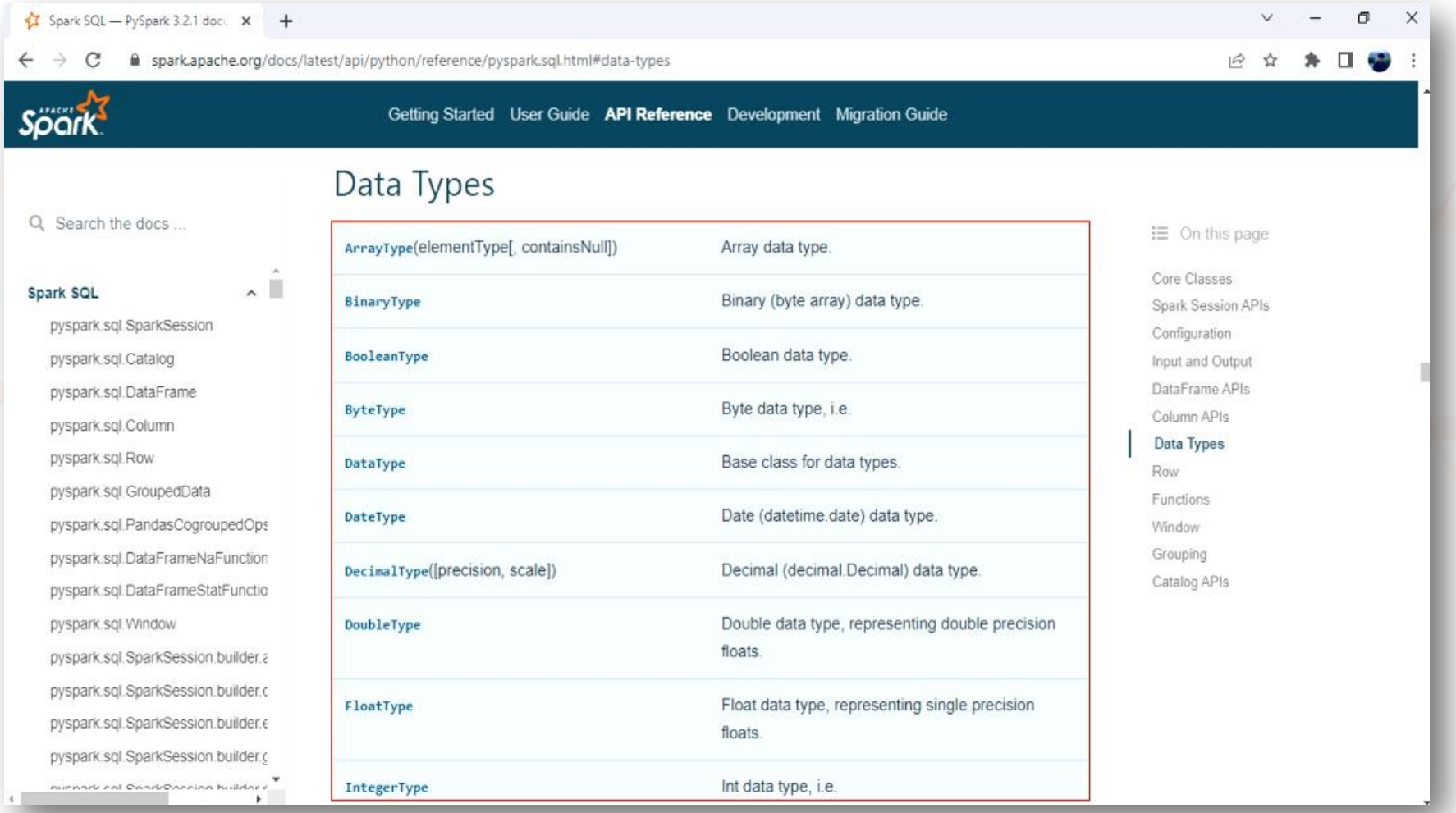
Instructor: Prashant Kumar Pandey

Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Spark Data Types

Every data processing system defines a set of data types.
If you have worked with relational databases, you know it already.
A database defines a list of data types, and you can use those types to define table schema.
Similarly, Spark also comes with a list of types.

Go to the latest Pyspark documentation page, and navigate to the Data Types page via the API reference option. (Reference : https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql.html#data-types)
You will see a list of Spark data types.



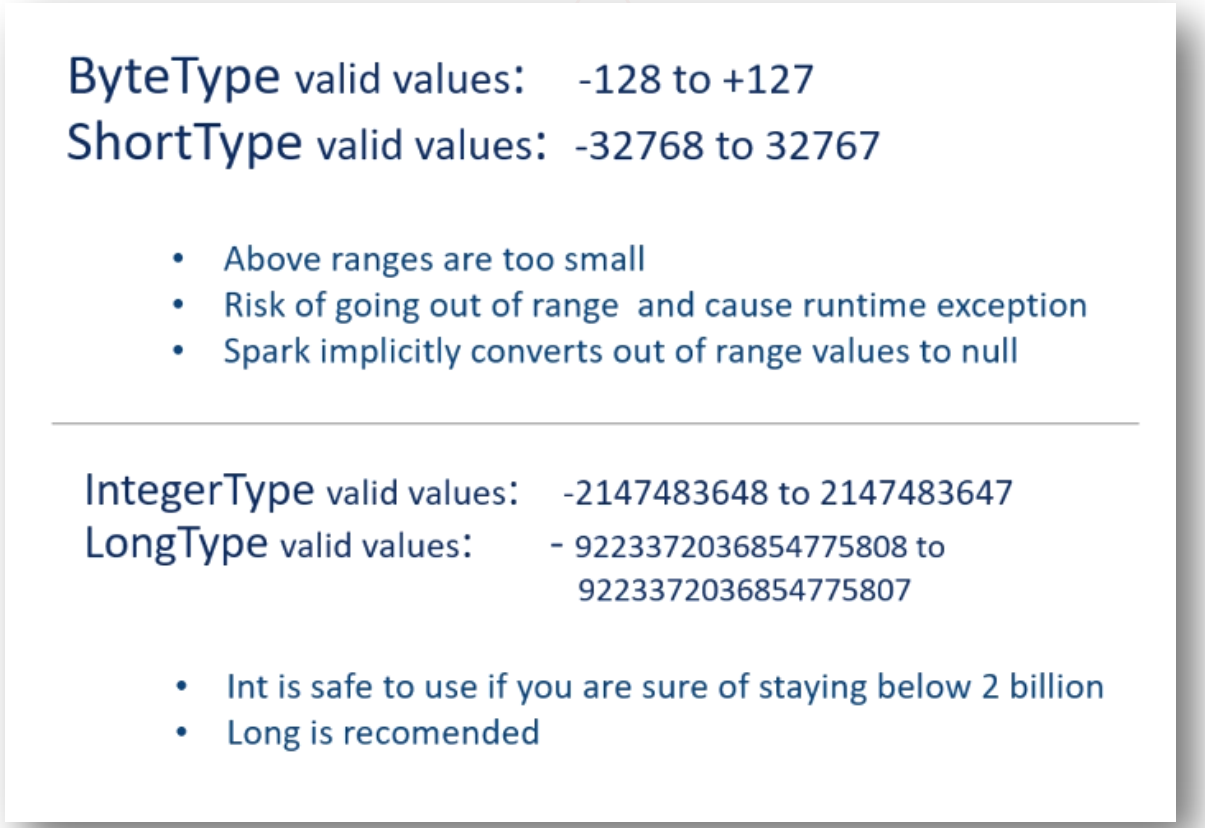
Here is a summarized list of all the Spark types.

S.No.	Spark Types	Spark SQL Types	Description
1.	DataType	NA	Base class for all Spark types
2.	ByteType	BYTE, TINYINT	8 bit signed integer between -128 to 127
3.	ShortType	SHORT, SMALLINT	16 bit signed integer between -32768 to 32767
4.	IntegerType	INT, INTEGER	32 bit signed integer between -2147483648 to 2147483647
5.	LongType	LONG, BIGINT	64 bit signed integer between 9223372036854775808 to 9223372036854775807
6.	FloatType	FLOAT, REAL	32 bit single precision floating point number
7.	DoubleType	DOUBLE	64 bit double precision floating point number
8.	DecimalType	DECIMAL, DEC, NUMERIC	Maximum precision
9.	BooleanType	BOOLEAN	
10.	StringType	STRING	
11.	DateType	DATE	
12.	TimestampType	TIMESTAMP	
13.	BinaryType	BINARY	
14.	ArrayType	ARRAY	
15.	MapType	MAP	
16.	StructType	STRUCT	
17.	StructField		
18.	NullType		
19.	YearMonthIntervalType	INTERVAL YEAR, INTERVAL YEAR TO MONTH, INTERVAL MONTH	
20.	DayTimeIntervalType	INTERVAL DAY, INTERVAL DAY TO HOUR, INTERVAL DAY TO MINUTE, INTERVAL DAY TO SECOND, INTERVAL HOUR, INTERVAL HOUR TO MINUTE, INTERVAL HOUR TO SECOND,	

So we have twenty types. The first four types are usually not used much in PySpark.
The first one is the DataType class. This guy is the base class of all the Spark Data types. You can ignore it because you are not likely to use it anywhere. The other three are ByteType, ShortType, and the IntegerType.
Spark defined them as 8-bit, 16-bit, and 32-bit signed integers. However, we do not recommend using these types. The best practice is to use the LongType for integers.

S.No.	Spark Types	Spark SQL Types	Description
1.	DataType	NA	Base class for all Spark types
2.	ByteType	BYTE, TINYINT	8 bit signed integer between -128 to 127
3.	ShortType	SHORT, SMALLINT	16 bit signed integer between -32768 to 32767
4.	IntegerType	INT, INTEGER	32 bit signed integer between -2147483648 to 2147483647
5.	LongType	LONG, BIGINT	64 bit signed integer between 9223372036854775808 to 9223372036854775807
6.	FloatType	FLOAT, REAL	32 bit single precision floating point number
7.	DoubleType	DOUBLE	64 bit double precision floating point number
8.	DecimalType	DECIMAL, DEC, NUMERIC	Maximum precision
9.	BooleanType	BOOLEAN	
10.	StringType	STRING	
11.	DateType	DATE	
12.	TimestampType	TIMESTAMP	
13.	BinaryType	BINARY	
14.	ArrayType	ARRAY	
15.	MapType	MAP	
16.	StructType	STRUCT	
17.	StructField		
18.	NullType		
19.	YearMonthIntervalType	INTERVAL YEAR, INTERVAL YEAR TO MONTH, INTERVAL MONTH	
20.	DayTimeIntervalType	INTERVAL DAY, INTERVAL DAY TO HOUR, INTERVAL DAY TO MINUTE, INTERVAL DAY TO SECOND, INTERVAL HOUR, INTERVAL HOUR TO MINUTE, INTERVAL HOUR TO SECOND,	

The Byte, Short, and Integers are designed in a data system to lower the memory footprint of the data. Spark performs in-memory computations. So using right-sized data types might help to use less memory. But saving memory over data types is not significant for real-life scenarios.
The IntegerType is 32-bit, and that guy is reasonably large. It can hold numbers a little larger than two billion. So we can use the IntegerType if we know we are not going to go above a couple of billion.
The long goes over nine quintillions, and that guy is relatively safe. You might not even reach quintillions. So one reason for recommending LongType and ignoring other integer types is to avoid the risk of crossing the limits.



Now, look at the savings.
IntegerType is 4 bytes, and LongType is 8 bytes. So for each column, you can save 4 bytes of memory. Let's assume you have 1 million records and one IntegerType column. You could have taken LongType, but you took IntegerType to save 4 bytes per record. You can see the saving in the table below.
Is it worth trying to save memory by choosing IntegerType instead of LongType if you do not have billions of records? And you keep on increasing the risk of crossing the limits.
So, LongType is recommended for integers unless you are too sure about the limits.

IntegerType: 4 Bytes
LongType: 8 Bytes

Long Size	Int Size	Net Savings	No of Columns	Number of Records	Total Savings	Savings in MB
8	4	4	1	1	4	0.000004
8	4	4	1	1000000	4000000	4
8	4	4	10	1000000	40000000	40
8	4	4	10	10000000	400000000	400
8	4	4	1	2000000000	8000000000	8000

The following three types are for decimal numbers. The float type is a 32-bit single-precision data type, and the double is 64-bit double precision. Most languages recommend using 64-bit double, and Spark is no different. We recommend using double for most of your floating-point numbers.
You can use DecimalType if you need maximum precision. However, most of your requirements are met by the double, and that one is recommended. So we have four integer types and three decimal types in Spark. However, you will be primarily using LongType and DoubleType in your application. These guys reduce the risk and fit most of your requirements.

S.No.	Spark Types	Spark SQL Types	Description
1.	DataType	NA	Base class for all Spark types
2.	ByteType	BYTE, TINYINT	8 bit signed integer between -128 to 127
3.	ShortType	SHORT, SMALLINT	16 bit signed integer between -32768 to 32767
4.	IntegerType	INT, INTEGER	32 bit signed integer between -2147483648 to 2147483647
5.	LongType	LONG, BIGINT	64 bit signed integer between 9223372036854775808 to 9223372036854775807
6.	FloatType	FLOAT, REAL	32 bit single precision floating point number
7.	DoubleType	DOUBLE	64 bit double precision floating point number
8.	DecimalType	DECIMAL, DEC, NUMERIC	Maximum precision
9.	BooleanType	BOOLEAN	
10.	StringType	STRING	
11.	DateType	DATE	
12.	TimestampType	TIMESTAMP	
13.	BinaryType	BINARY	
14.	ArrayType	ARRAY	
15.	MapType	MAP	
16.	StructType	STRUCT	

The following five data types are straightforward. The BooleanType is for true/false boolean values. StringType is to store string values.
The Datetype stores date values, and TimestampType is for a date with a time component. And the BinamyType is for raw bytes.

S.No.	Spark Types	Spark SQL Types	Description
1.	DataType	NA	Base class for all Spark types
2.	ByteType	BYTE, TINYINT	8 bit signed integer between -128 to 127
3.	ShortType	SHORT, SMALLINT	16 bit signed integer between -32768 to 32767
4.	IntegerType	INT, INTEGER	32 bit signed integer between -2147483648 to 2147483647
5.	LongType	LONG, BIGINT	64 bit signed integer between 9223372036854775808 to 9223372036854775807
6.	FloatType	FLOAT, REAL	32 bit single precision floating point number
7.	DoubleType	DOUBLE	64 bit double precision floating point number
8.	DecimalType	DECIMAL, DEC, NUMERIC	Maximum precision
9.	BooleanType	BOOLEAN	
10.	StringType	STRING	
11.	DateType	DATE	
12.	TimestampType	TIMESTAMP	
13.	BinaryType	BINARY	
14.	ArrayType	ARRAY	
15.	MapType	MAP	
16.	StructType	STRUCT	

Then we have three complex data types: ArrayType, MapType and StructType.

S.No.	Spark Types	Spark SQL Types	Description
1.	DataType	NA	Base class for all Spark types
2.	ByteType	BYTE, TINYINT	8 bit signed integer between -128 to 127
3.	ShortType	SHORT, SMALLINT	16 bit signed integer between -32768 to 32767
4.	IntegerType	INT, INTEGER	32 bit signed integer between -2147483648 to 2147483647
5.	LongType	LONG, BIGINT	64 bit signed integer between 9223372036854775808 to 9223372036854775807
6.	FloatType	FLOAT, REAL	32 bit single precision floating point number
7.	DoubleType	DOUBLE	64 bit double precision floating point number
8.	DecimalType	DECIMAL, DEC, NUMERIC	Maximum precision
9.	BooleanType	BOOLEAN	
10.	StringType	STRING	
11.	DateType	DATE	
12.	TimestampType	TIMESTAMP	
13.	BinaryType	BINARY	
14.	ArrayType	ARRAY	
15.	MapType	MAP	
16.	StructType	STRUCT	

This sheet also shows the SQL keywords for these data types. For example, we use LongType in PySpark, and we can use LONG or BIGINT in Spark SQL. The LONG and BIGINT are the same in Spark SQL, and they internally represent LongType in Spark.
Similarly, you can use StringType in PySpark and STRING in Spark SQL. They mean the same thing.

S.No.	Spark Types	Spark SQL Types	Description
1.	DataType	NA	Base class for all Spark types
2.	ByteType	BYTE, TINYINT	8 bit signed integer between -128 to 127
3.	ShortType	SHORT, SMALLINT	16 bit signed integer between -32768 to 32767
4.	IntegerType	INT, INTEGER	32 bit signed integer between -2147483648 to 2147483647
5.	LongType	LONG, BIGINT	64 bit signed integer between 9223372036854775808 to 9223372036854775807
6.	FloatType	FLOAT, REAL	32 bit single precision floating point number
7.	DoubleType	DOUBLE	64 bit double precision floating point number
8.	DecimalType	DECIMAL, DEC, NUMERIC	Maximum precision
9.	BooleanType	BOOLEAN	
10.	StringType	STRING	
11.	DateType	DATE	
12.	TimestampType	TIMESTAMP	
13.	BinaryType	BINARY	
14.	ArrayType	ARRAY	
15.	MapType	MAP	
16.	StructType	STRUCT	

We are using Spark SQL data types to define the table schema. That's how you can use the Spark SQL data types.

```
create table if not exists demo_db.fire_service_calls_tbl(
  CallNumber integer,
  UnitID string,
  IncidentNumber integer,
  CallType string,
  CallDate string,
  WatchDate string,
  CallFinalDisposition string,
  AvailableDtTm string,
  Address string,
  City string,
  Zipcode integer,
  Battalion string,
  StationArea string,
  Box string,
  OriginalPriority string,
  Priority string,
  FinalPriority integer,
  ALSUnit boolean,
  CallTypeGroup string,
  NumAlarms integer,
  UnitType string,
  UnitSequenceInCallDispatch integer,
  FirePreventionDistrict string,
  SupervisorDistrict string,
  Neighborhood string,
  Location string,
  RowID string,
  Delay float
) using parquet
```

Here is how we use Dataframe Schema in PySpark. We learned two approaches. The first one shown above uses using DDL schema definition, and the second one shown below uses PySpark syntax. And this is where we use Spark data types.

```
flight_schema_ddl = """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_TIME INT,
                       WHEELS_ON INT, TAXI_IN INT, CRS_ARR_TIME INT, ARR_TIME INT, CANCELLED STRING, DISTANCE INT"""
```

```
flight_schema = StructType([
    StructField("FL_DATE", DateType()),
    StructField("OP_CARRIER", StringType()),
    StructField("OP_CARRIER_FL_NUM", IntegerType()),
    StructField("ORIGIN", StringType()),
    StructField("ORIGIN_CITY_NAME", StringType()),
    StructField("DEST", StringType()),
    StructField("DEST_CITY_NAME", StringType()),
    StructField("CRS_DEP_TIME", IntegerType()),
    StructField("DEP_TIME", IntegerType()),
    StructField("WHEELS_ON", IntegerType()),
    StructField("TAXI_IN", IntegerType()),
    StructField("CRS_ARR_TIME", IntegerType()),
    StructField("ARR_TIME", IntegerType()),
    StructField("CANCELLED", IntegerType()),
    StructField("DISTANCE", IntegerType())
])
```

We have four more types in Spark listed below.
The StructField is not an actual data type. We use it to define a dataframe column in PySpark. We use this guy to define a schema. This class is defined in the Spark types package, but it is not an actual data type. It is a utility class.
The NullType represents no data type. You are not likely to use this class. Spark assigns a NullType to a column when it cannot infer the schema for a dataframe column. But we do not recommend using schema inference. So the NullType is not used anywhere if you are not using schema inference.
The remaining two types are to represent intervals. These types are not yet available to PySpark. These guys are only available in Scala, Java, and Spark SQL. PySpark doesn't have anything for these two guys. So you cannot explicitly use them in PySpark, but we can use them in Spark SQL. However, future versions might bring these to PySpark as well.

```
17.     StructField
18.     NullType
19.     YearMonthIntervalType       INTERVAL YEAR,
                                    INTERVAL YEAR TO MONTH,
                                    INTERVAL MONTH
20.     DayTimeIntervalType         INTERVAL DAY,
                                    INTERVAL DAY TO HOUR,
                                    INTERVAL DAY TO MINUTE,
                                    INTERVAL DAY TO SECOND,
                                    INTERVAL HOUR,
                                    INTERVAL HOUR TO MINUTE,
                                    INTERVAL HOUR TO SECOND,
```


Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

ScholarNest

Spark Azure Databricks

Databricks Spark Certification and beyond

Instructor: Prashant Kumar Pandey

Absolute Beginner to Specialization in Apache Spark and Azure Databricks

Casting and Aliases

Let's assume you are given a data file. You are told that data is dirty. So you must clean it up and save it as a Spark table. Once the data is clean, we will be using it for analysis. Right?
That's a common requirement.

You want to start working on this requirement.
So you asked for the following things:

1. Sample data file
2. File schema
3. Known data problems

You need a sample data file to test your code using the sample data and verify if your logic is working.
And you got a directory location where some data files are there. You have 2TB of data.
That's too big, and you cannot use such volumes for development and testing.
But that's fine. You at least have data to start the work. Then you asked for file schema so you can read the given data, and you got the schema as shown below.

Requirement:

Data Cleaning and Preparation

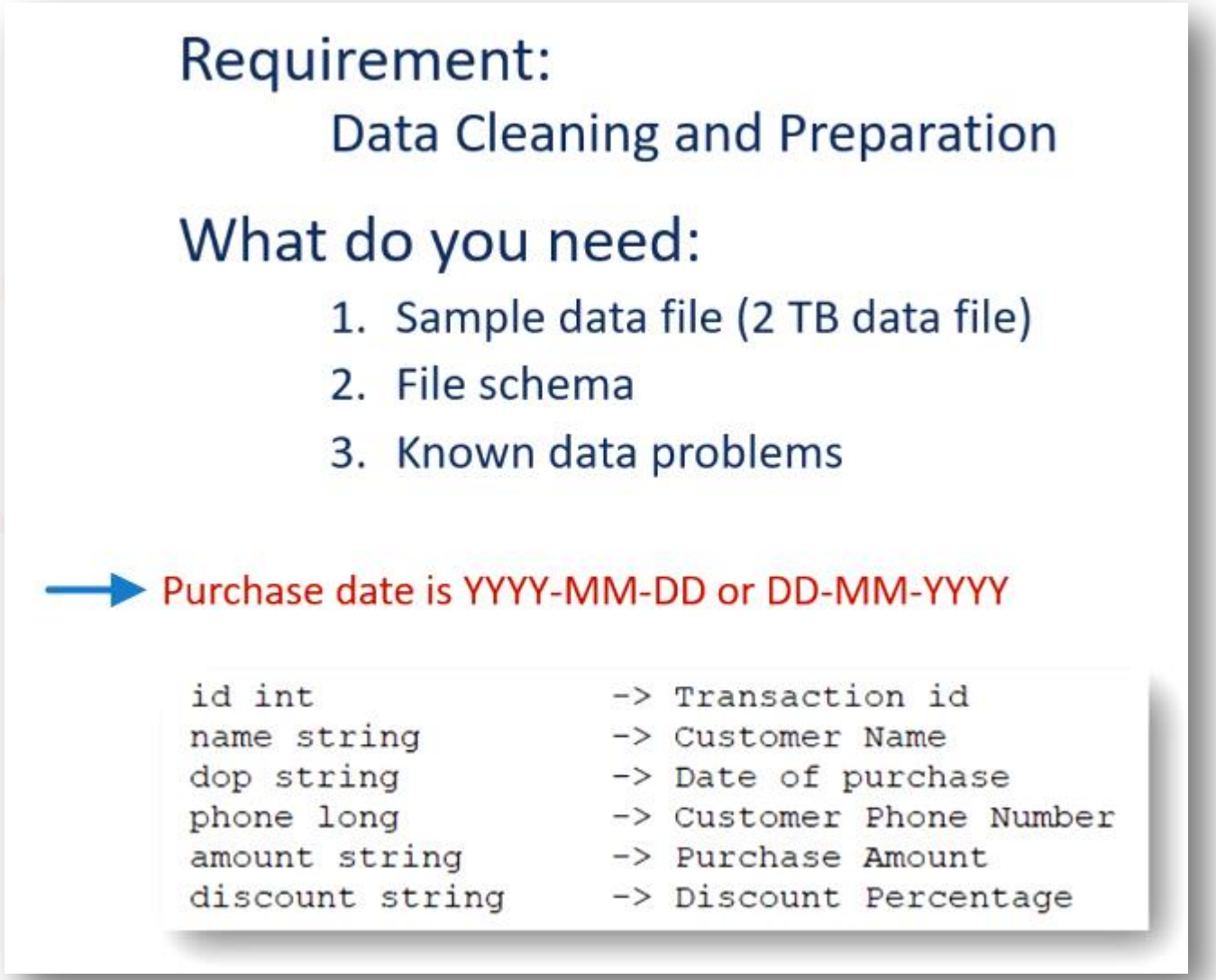
What do you need:

1. Sample data file (2 TB data file)
2. File schema
3. Known data problems

Purchase date is in YYYY-MM-DD or DD-MMM-YYYY

```
id int              -> Transaction id
name string         -> Customer Name
dop string          -> Date of purchase
phone long          -> Customer Phone Number
amount string       -> Purchase Amount
discount string     -> Discount Percentage
```

And they give you only one know problem.
Some date of purchase values are in YYYY-MM-DD format, but we also have values in DD-MM-YYYY format.

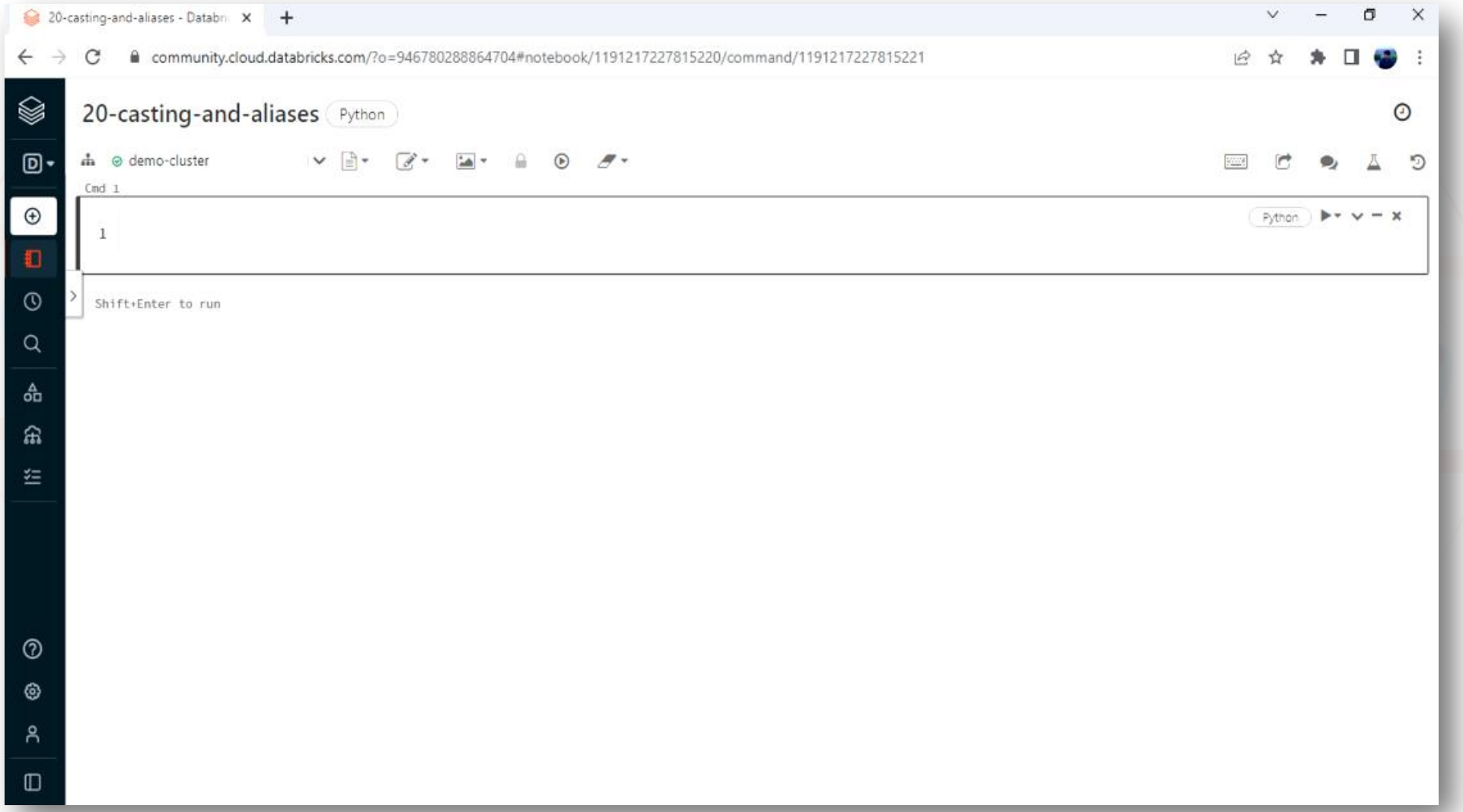


The first thing is to formulate the solution or make a list of things we want to do. I prepared the following list:

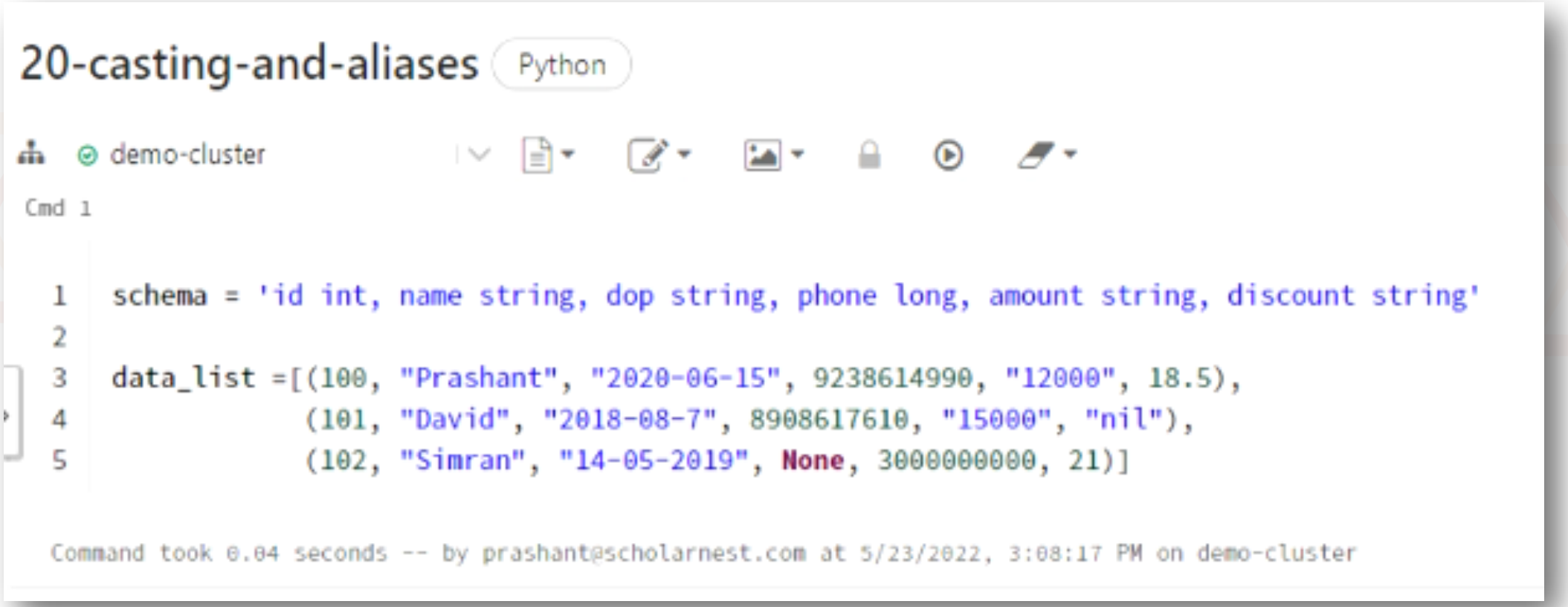
1. Create a small but good quality sample dataset
2. Analyse dataset to identify data problems
3. Ensure the sample includes problem records
4. Develop logic to correct and segregate
5. Save the correct records in a Spark table
6. Save the incorrect records to an error table

You need a small sample dataset for development and testing. You cannot work with a 2TB dataset for development and testing. That's too expensive and time-consuming also.
So the first step is to read the given data and create a sample Dataframe.
You can read the given data and take 10K samples from it. We have a known problem with different date formats. So you wanted to make sure that you are sampling data from YYYY-MM-DD and DD-MM-YYYY formats. So you can also go for a stratified sample for the date of purchase column.

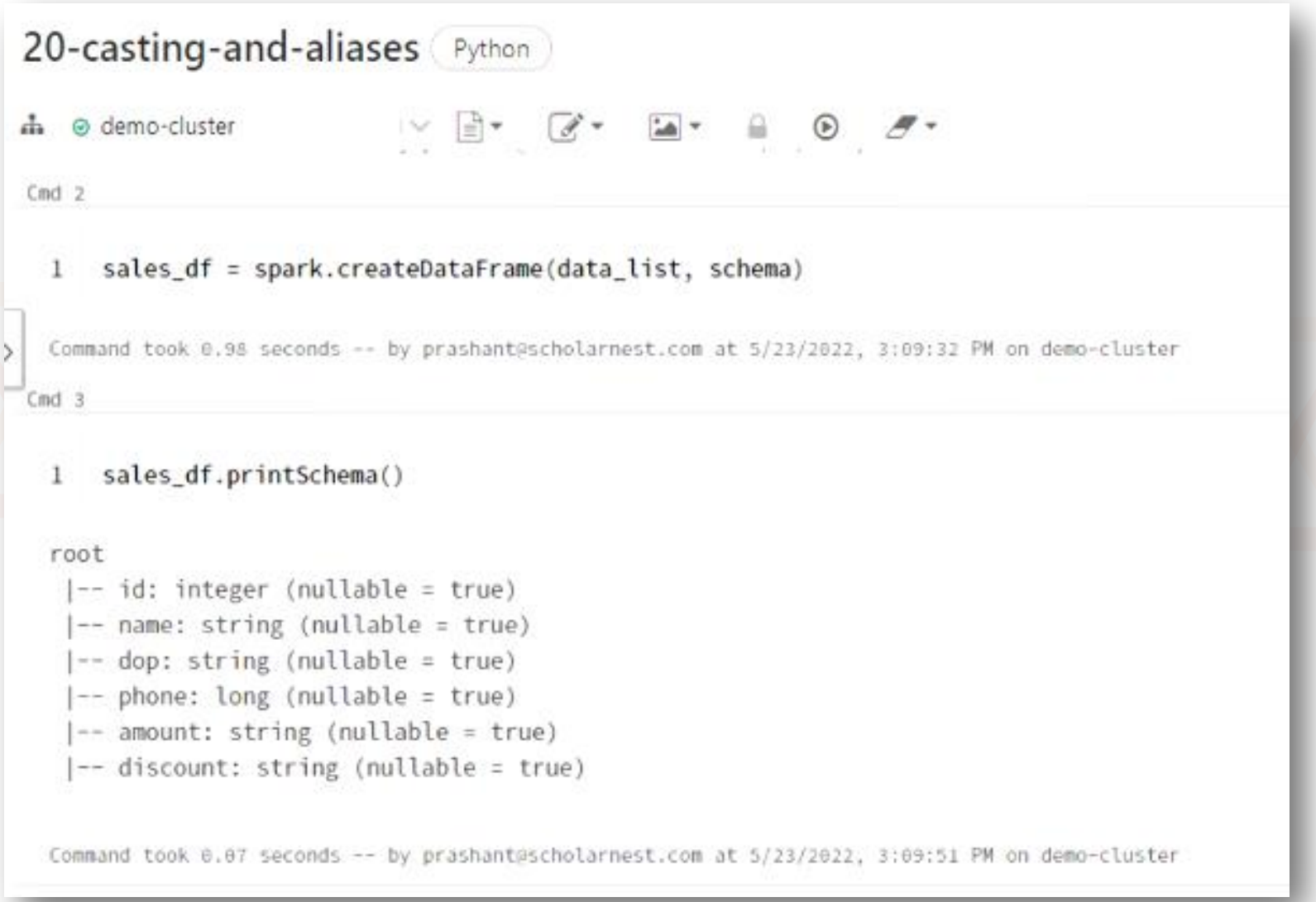
Go to your Databricks workspace and create a new notebook. **(Reference : 20-casting-and-aliases)**



I want to create a sample Dataframe for the given requirement manually. So, I have defined a data list and a schema for my requirement.



Then we are creating a Dataframe and printing the schema of the Dataframe.



Let's analyze our sample and list down some problems.
I already know some schema problems. Look at the schema below.
The id is an integer, the date of purchase is a string, the phone is a long, amount and discount are strings. That's not correct.

Cmd 3

```
1  sales_df.printSchema()
```

```
root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- dop: string (nullable = true)
 |-- phone: long (nullable = true)
 |-- amount: string (nullable = true)
 |-- discount: string (nullable = true)
```

Command took 0.07 seconds -- by prashant@scholarnest.com at 5/23/2022, 3:09:51 PM on demo-cluster

I want to change the column names and data types to make them standard. Here is the new schema which I want to achieve. So I will be changing column names and some data types.
For example, transaction_id is changed to string. Similarly, date_of_purchase is a date field.
The purchase_amount is an integer, and the discount could be a double value. The best practice is to use long and avoid using integers. But I know it is purchase_amount, and it could not go bigger than two billion. If it is, that's a data problem for sure. I feel safe using an integer for purchase_amount, so let me keep it as an integer.

```
root
 |-- transaction_id: string (nullable = true)
 |-- customer_name: string (nullable = true)
 |-- date_of_purchase: date (nullable = true)
 |-- customer_phone: string (nullable = true)
 |-- purchase_amount: integer (nullable = true)
 |-- applied_discount: double (nullable = true)
```

Now let me analyze the data for new problems. Here is the sales_df Dataframe shown below. Eyeballing is one easy way to analyze and identify problems in a Dataframe.

Cmd 4

```
1  display(sales_df)
```

▸ (3) Spark Jobs

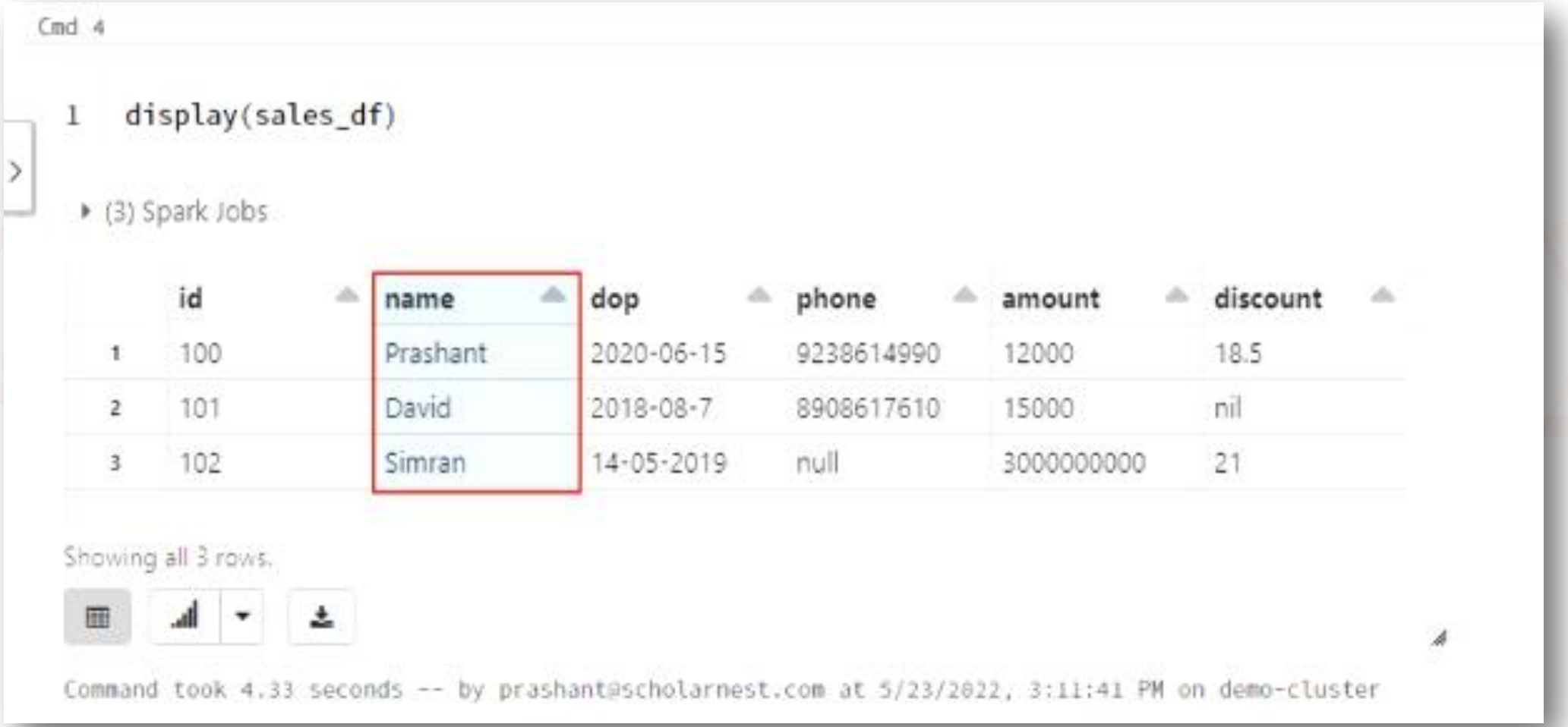
	id	name	dop	phone	amount	discount
1	100	Prashant	2020-06-15	9238614990	12000	18.5
2	101	David	2018-08-7	8908617610	15000	nil
3	102	Simran	14-05-2019	null	3000000000	21

Showing all 3 rows.

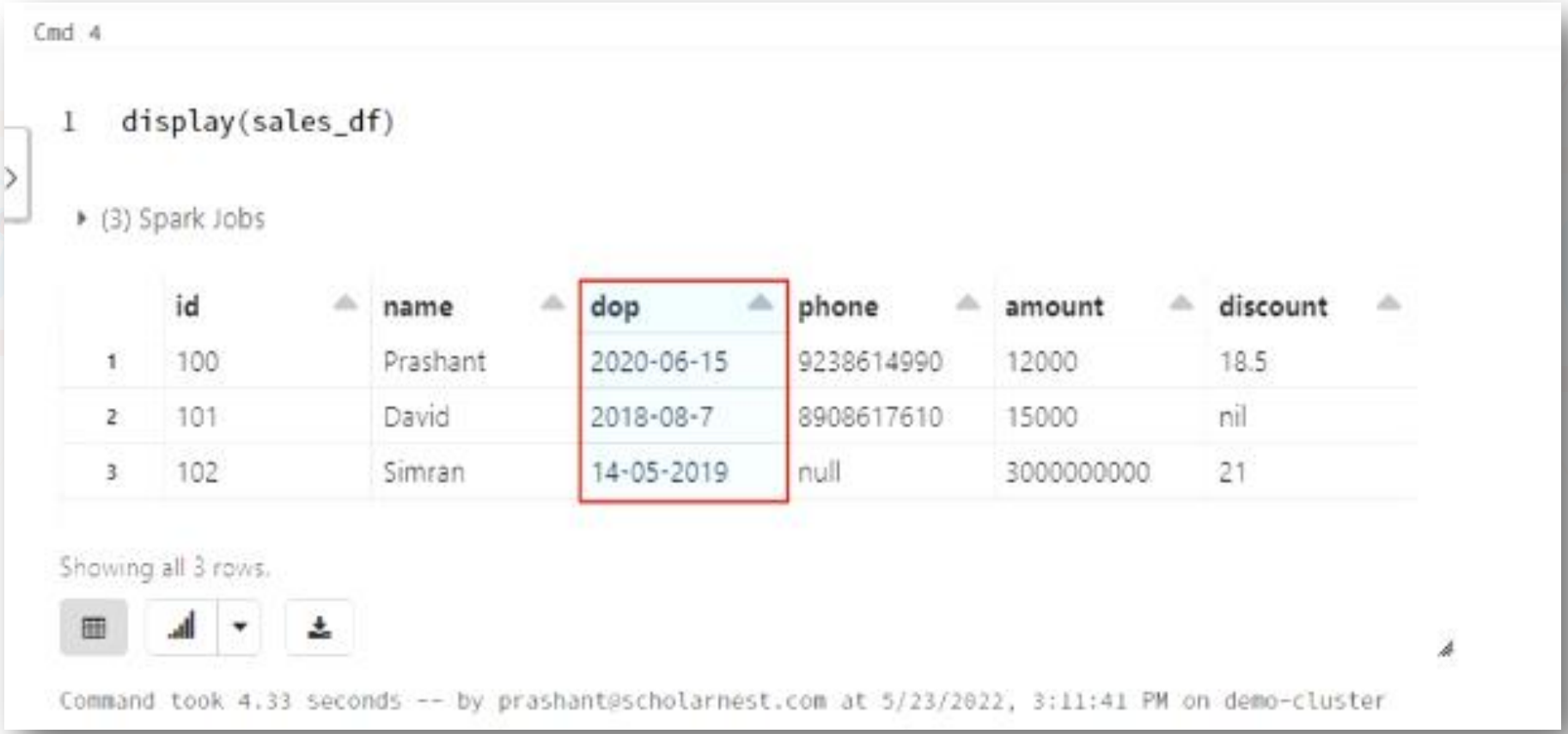
Command took 4.33 seconds -- by prashant@scholarnest.com at 5/23/2022, 3:11:41 PM on demo-cluster

So the id column is fine. It looks like integer values, and we want to convert it to string. That's a schema change, and I already listed it earlier.

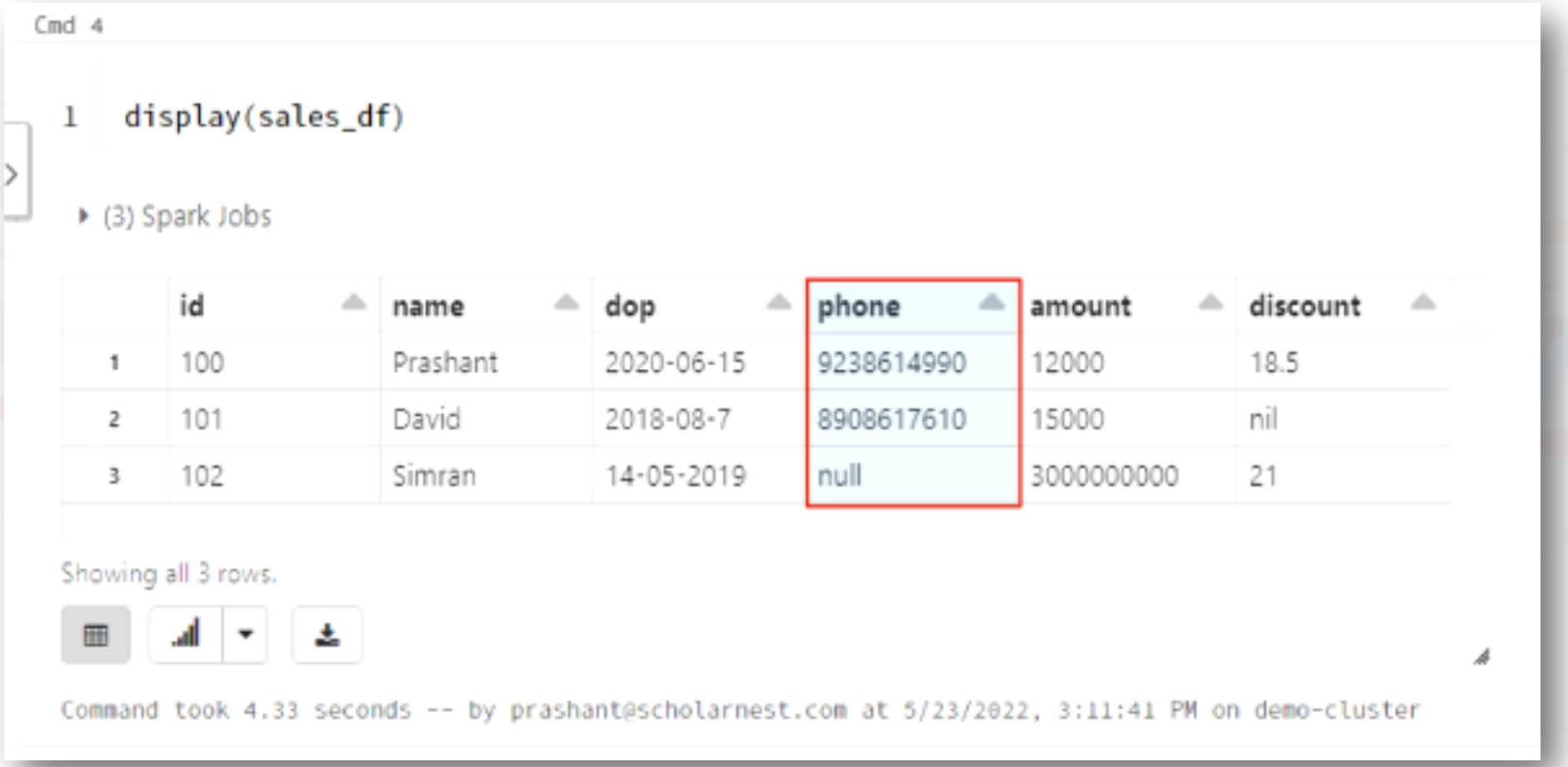
The name column values are also acceptable, and we only want to change the column name to customer_name.



I can see the date in two formats. Some records are in YYYY-MM-DD, and I have a record in DD-MM-YYYY. That was a known problem, and I already knew it. However, one of the dates is 2018-08-7. The date component is a single digit. Ideally, it should be a double-digit number as 07. But it shows single digit 7. Let me note them down as new problems, and we will see how to fix them.



The phone number is also acceptable. It is null in some places, but that's not a data problem. The number is unknown, and null is the best value representing the missing number.



	id	name	dop	phone	amount	discount
1	100	Prashant	2020-06-15	9238614990	12000	18.5
2	101	David	2018-08-7	8908617610	15000	nil
3	102	Simran	14-05-2019	null	3000000000	21

The amount looks good, with one exception. I assumed the purchase_amount to be a smaller number. But I can see a value larger than 2 billion. This record is incorrect for sure. So I have two options.

1. Fix the purchase_amount value
2. Separate this record into an error table.

I do not know how to fix the purchase_amount. I mean, I can set it to zero or null. But that will be wrong. I cannot assume any other value because we do not know the actual value. So it is best to separate this record into an error table. Even if I have nulls for the purchase_amount, those records should also go to the error table.

Now let's see the discount column. Some values are nil. That's a problem. I want to change these values to zero and convert the column to a double.

	id	name	dop	phone	amount	discount
1	100	Prashant	2020-06-15	9238614990	12000	18.5
2	101	David	2018-08-7	8908617610	15000	nil
3	102	Simran	14-05-2019	null	3000000000	21

I used eyeballing to identify the problems. But that's not a recommended approach. I mean, you cannot identify data problems eyeballing a million records.
Data engineers use various techniques for analyzing and identifying data problems. However, statistical methods are the most commonly used approach. In this method, I am selecting the columns dop, amount, and discount. Then I am using describe() method to show some default statistical analysis of the given columns. The describe() is a Dataframe transformation.
It can perform fundamental statistical analysis on a Dataframe such as count, min, max, percentile, and mean values.

You can see in the output below, the max value of the discount column is nil. And you know that's a problem. You were expecting decimal values in this column, but you have some strings.
Similarly, the mean and max values of the amount are too high. You were not expecting the amount to be in billions. So analysing your data and identifying data problems, outliers, unexpected values, etc., is a common activity for data engineers.
If you know your data, Spark methods, and their purpose, you can easily do the analysis.

Cmd 5

```
1  sales_df.select("dop", "amount", "discount").describe().show()
```

▸ (2) Spark Jobs

```
+-------+----------+--------------------+------------------+
|summary|       dop|              amount|          discount|
+-------+----------+--------------------+------------------+
|  count|         3|                   3|                 3|
|   mean|      null|          1.000009E9|             19.75|
| stddev|      null|1.7320430133408928E9|1.7677669529663689|
|    min|14-05-2019|               12000|              18.5|
|    max|2020-06-15|          3000000000|               nil|
+-------+----------+--------------------+------------------+
```

Command took 4.40 seconds -- by prashant@scholarnest.com at 5/23/2022, 3:14:01 PM on demo-cluster

So we learned to identify problems, and I have the following list:

1. Convert id from integer to string and rename it as transaction_id.
2. Rename the name column to customer_name.
3. Convert the dop to date format and rename the column to date_of_purchase.
4. Rename the phone column to customer_phone
5. Convert the amount to an Integer value and filter out nulls and outlier values. Rename the column to purchase_amount
6. Convert amount to double, converting nil and null values to zero. rename the column to applied_discount

We have requirements to rename almost all the columns. And we also have a requirement to convert data types. You are also supposed to filter out some dirty records.

You can rename columns. I showed you two approaches:

1. withColumnRenamed()
2. alias()

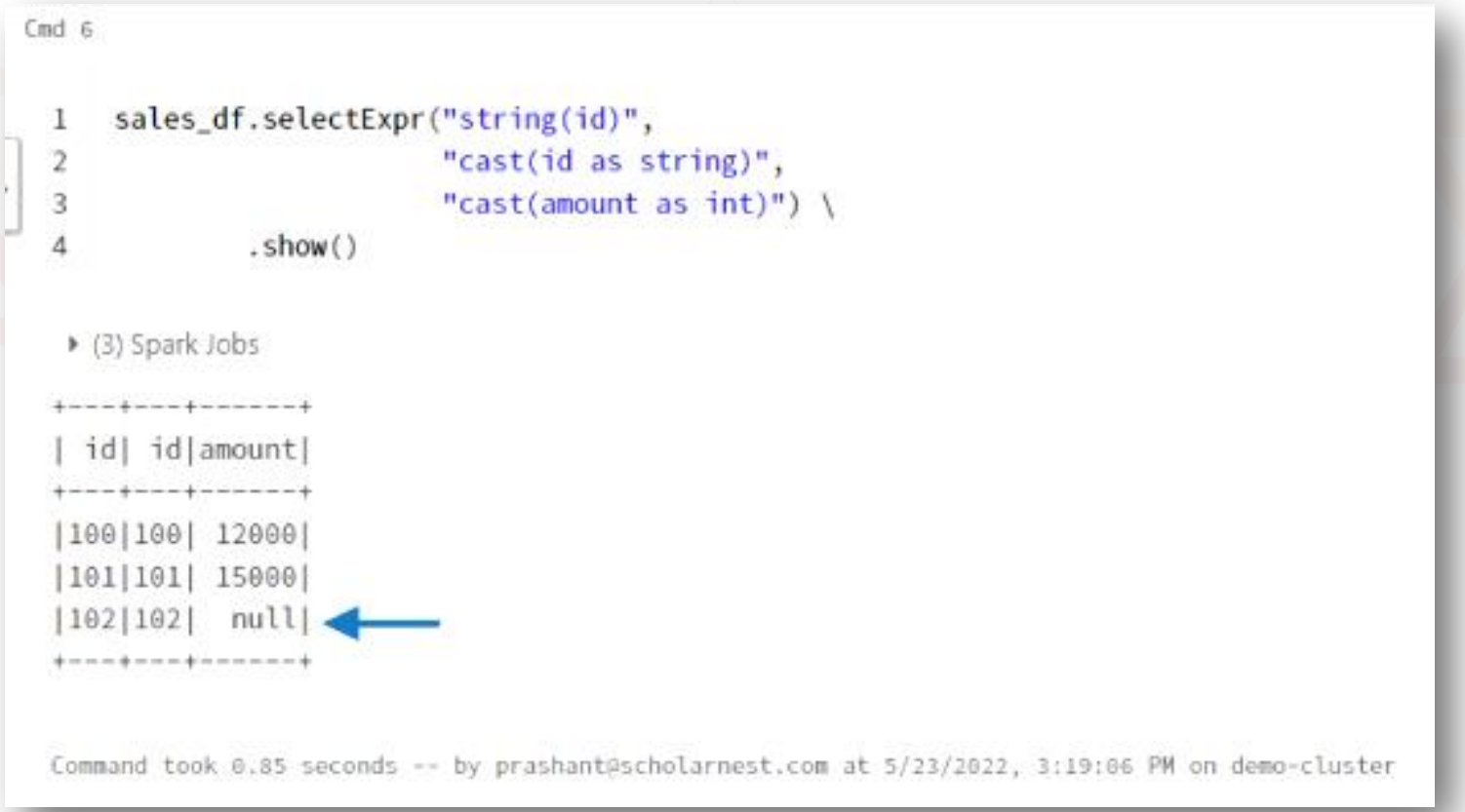
If you rename multiple columns, we recommend using the alias and avoid withColumnRenamed().
Why? Because each withColumnRenamed() adds a projection to your SQL plan and slows down your execution. So renaming multiple columns should be done using an alias in a single select() expression.

You already learned one approach to converting data types. We have two approaches to converting data types:

1. Using data type conversion functions such as to_date(), string(), int() etc.
2. Using cast() function

The id column is an integer. I want to convert it to a string. I told you about two approaches in the earlier slide, and here is the code using both approaches shown below.
You can use the string() built-in function to convert a column to a string. You can also use the cast() built-in function to convert a column to a string. The string() function is specific for converting a value to a string. However, the cast function is generic. You can use the cast function to convert a value to a string, integer, double, date, or whatever you want. The cast function supports all the Spark and Spark SQL datatypes.

You can cast values as a different data type. The cast function will convert the value if it can. But if it cannot convert the value to a given data type, it will convert it to a null. You can see it here in the output below.
You can see the amount for 102 is now set to null.
We have a huge number that couldn't fit into an integer. So the casting method could not convert it to an integer, so this value becomes null. That's how all the data type conversion approaches work. You can try the case() function or try some other method to convert a data type. If the value is not good to be converted, it will become null.



Assignment:

You have the following requirements:

1. Convert id from integer to string and rename it as transaction_id.
2. Rename the name column to customer_name.
3. Convert the dop to date format and rename the column to date_of_purchase.
4. Rename the phone column to customer_phone and convert it to string
5. Convert the amount to an Integer value and filter out nulls and outlier values. Rename the column to purchase_amount
6. Convert discount to double, converting nil and null values to zero. Rename the column to applied_discount

You need the following tools to meet these requirements:

1. Rename columns
2. Convert Data types
3. nvl() function
4. Filter Records

You can use an alias to rename a column. You can use the cast() function to convert data types.
You can also use the to_date() function to convert a string date to a date type. You can use the nvl() function to check the null value and take an alternative value. You can use the where() method to filter unwanted records.

I leave you here so you can write code to meet the given requirements.
Here is the screenshot of the final result shown below. You can compare your solution output with the one shown here. The solution to this problem is given in the notebook. So you can download your study material and refer to the solution.

Expected Output

```
root
 |-- transaction_id: string (nullable = true)
 |-- customer_name: string (nullable = true)
 |-- date_of_purchase: date (nullable = true)
 |-- customer_phone: string (nullable = true)
 |-- purchase_amount: integer (nullable = true)
 |-- applied_discount: double (nullable = false)

+--------------+-------------+----------------+--------------+---------------+----------------+
|transaction_id|customer_name|date_of_purchase|customer_phone|purchase_amount|applied_discount|
+--------------+-------------+----------------+--------------+---------------+----------------+
|           100|     Prashant|      2020-06-15|    9238614990|          12000|            18.5|
|           101|        David|      2018-08-07|    8908617610|          15000|             0.0|
+--------------+-------------+----------------+--------------+---------------+----------------+
```

Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com