Spark Azure Databricks

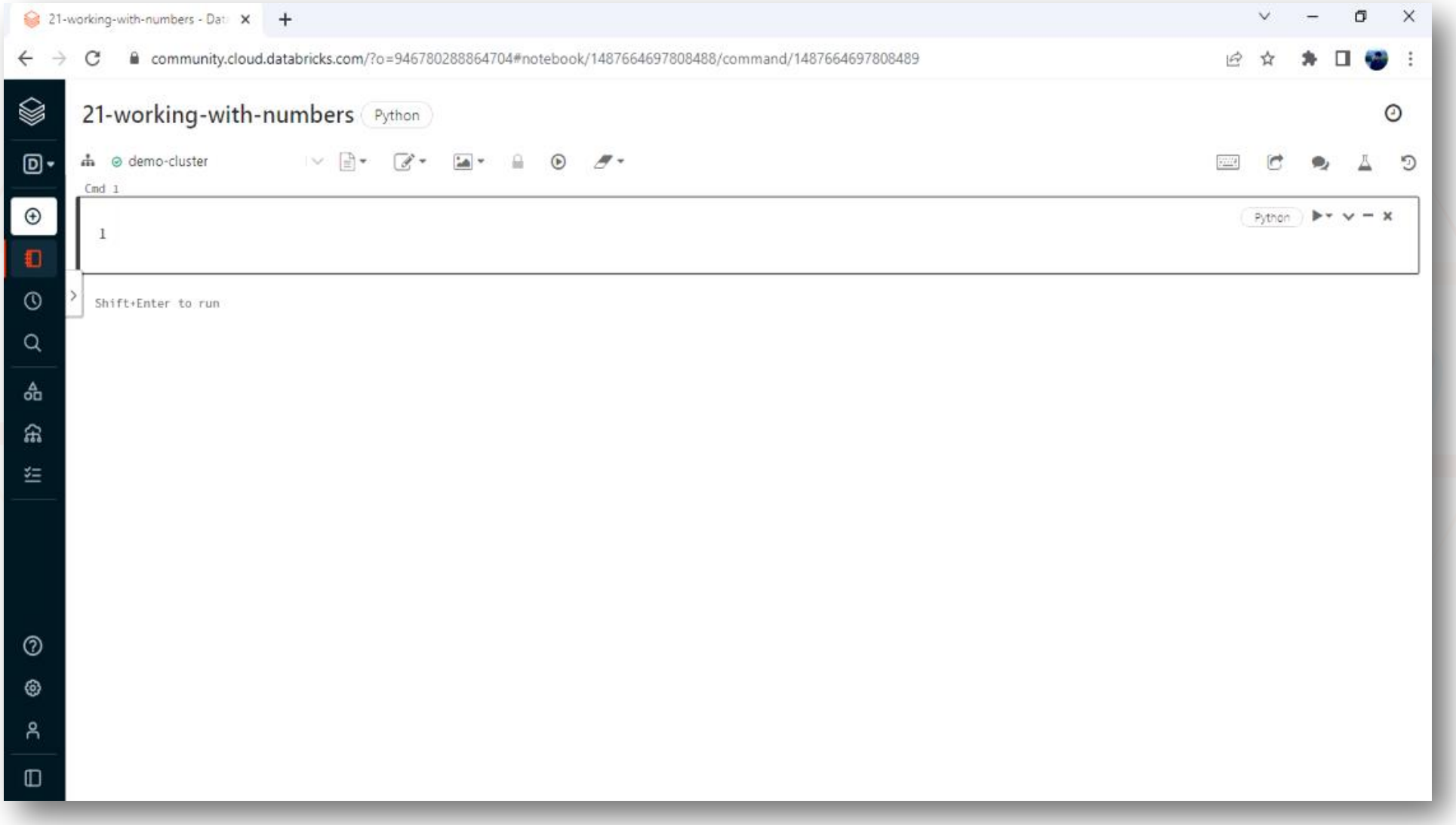
Databricks Spark Certification and Beyond

Module:
Working with Different Data Types

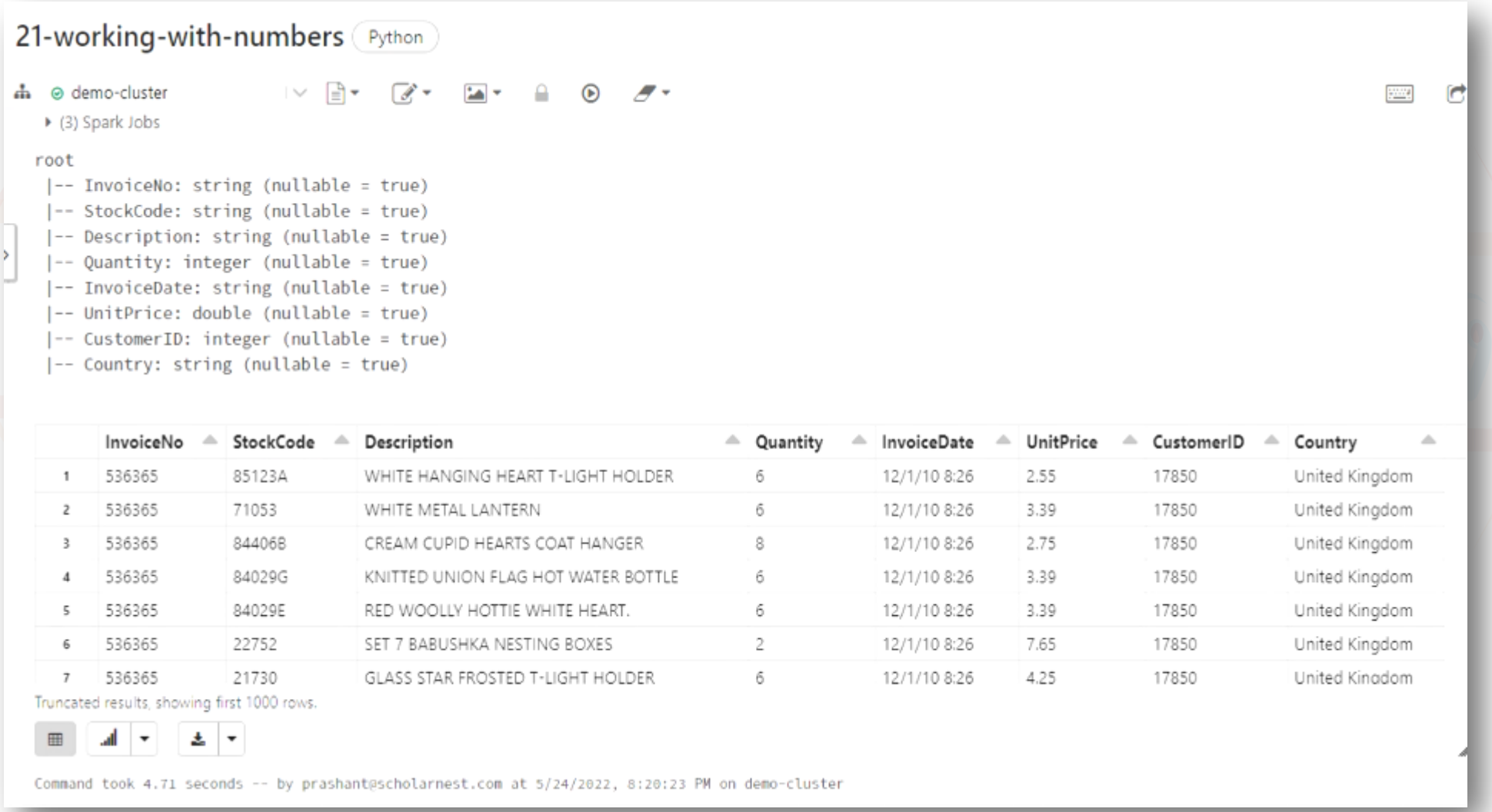
Lecture:
Working with Numbers

Working with Numbers

Go to your Databricks workspace and create a new notebook. **(Reference : 21-working-with-numbers)**

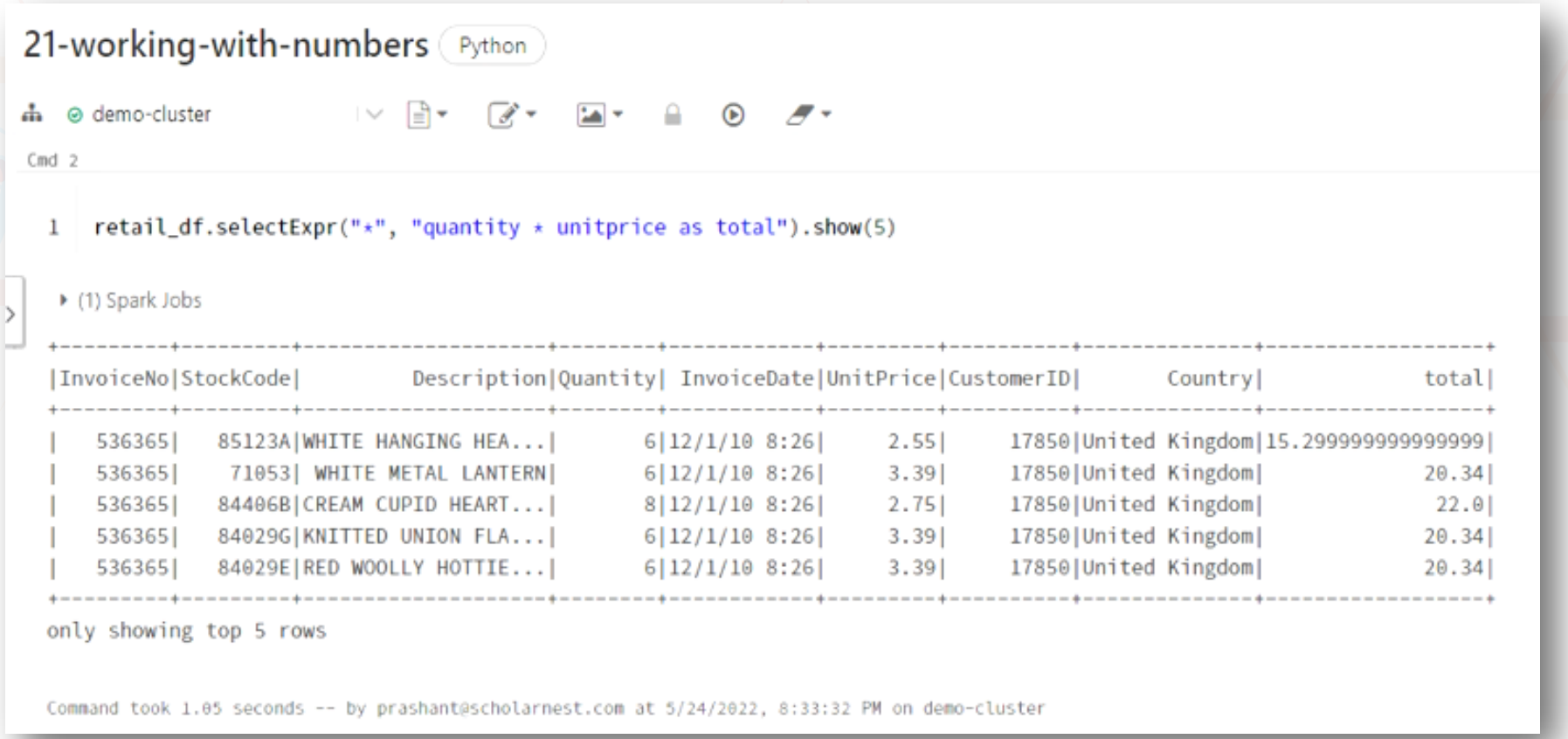


We need a Dataframe to look at some of the examples, so I have created a Dataframe below.
So we have two exciting fields here: quantity and unit price.
These two are numeric values, and I want to show you some examples using these fields.



	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/10 8:26	2.55	17850	United Kingdom
2	536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom
3	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/10 8:26	2.75	17850	United Kingdom
4	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/10 8:26	3.39	17850	United Kingdom
5	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	12/1/10 8:26	3.39	17850	United Kingdom
6	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	12/1/10 8:26	7.65	17850	United Kingdom
7	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	12/1/10 8:26	4.25	17850	United Kingdom

So let's start with a simple requirement to calculate the total sales value for each record.
We already have quantity and unit price.
Calculating total sales is as simple as multiplying the unit price with the quantity. You can easily do it using a SQL expression if you know SQL. And we have the code for the same given below.
I am using the selectExpr() method because I know I will run an expression. I am doing select *, so I get all existing columns. Then I use a SQL expression to multiply quantity and unit price and give it an alias.



You can always use the SQL approach. However, you should also know other ways to create Dataframe expressions. Because you might be reading code written by others and need to modify it.
The other developers might have used a different approach to creating expressions.
We do not want to get puzzled looking at existing code.
So it is always good to learn different approaches to creating expressions.
However, you are free to practice and use only one approach, which is more comfortable.
There is no performance impact, and Spark runs all the approaches with the same performance.

I used selectExpr() method in the previous approach. But you can also use the withColumn() transformation as shown below.

It is also same as the earlier expression. I want a new total column, so the column name is total. The second argument is an expression. You do not need to select all columns because the withColumn() adds or modifies only one column and keeps all earlier columns. I do not recommend using a long chain of withColumn() methods. However, using one is perfectly fine.

Cmd 3

```
1  from pyspark.sql.functions import expr
2
3  retail_df.withColumn("total", expr("quantity * unitprice")).show(5)
```

▸ (1) Spark Jobs

```
+---------+---------+--------------------+--------+-------------+---------+----------+--------------+------------------+
|InvoiceNo|StockCode|         Description|Quantity| InvoiceDate|UnitPrice|CustomerID|       Country|             total|
+---------+---------+--------------------+--------+-------------+---------+----------+--------------+------------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/10 8:26|     2.55|     17850|United Kingdom|15.299999999999999|
|   536365|    71053| WHITE METAL LANTERN|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|             20.34|
|   536365|   84406B|CREAM CUPID HEART...|       8|12/1/10 8:26|     2.75|     17850|United Kingdom|              22.0|
|   536365|   84029G|KNITTED UNION FLA...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|             20.34|
|   536365|   84029E|RED WOOLLY HOTTIE...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|             20.34|
+---------+---------+--------------------+--------+-------------+---------+----------+--------------+------------------+
only showing top 5 rows
```

Command took 0.49 seconds -- by prashant@scholarnest.com at 5/24/2022, 8:37:11 PM on demo-cluster

So, we have seen selectExpr() and withColumn() approach. You can also use the select() method as shown below.
So the select and selectExpr() are the same.
You will use expr() function explicitly in the select() method and the selectExpr() implicitly adds the expr() to each column.

Cmd 4

```
1  retail_df.select("*", expr("quantity * unitprice as total")).show(5)
```

▸ (1) Spark Jobs

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	total
536365	85123A	WHITE HANGING HEA...	6	12/1/10 8:26	2.55	17850	United Kingdom	15.299999999999999
536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84406B	CREAM CUPID HEART...	8	12/1/10 8:26	2.75	17850	United Kingdom	22.0
536365	84029G	KNITTED UNION FLA...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34

only showing top 5 rows

Command took 0.69 seconds -- by prashant@scholarnest.com at 5/24/2022, 8:43:09 PM on demo-cluster

Now let us move onto column expressions.
Here is an example of column expressions. I am using the quantity column multiplied by the unit price column. Most people do not prefer using column expressions like this. It always confuses many of us, and SQL developers hate this approach. However, we have an option, and many would like to use it.

Cmd 5

```
1  from pyspark.sql.functions import col
2
3  retail_df.select("*", col("quantity") * col("unitprice")).show(5)
```

▸ (1) Spark Jobs

```
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+----------------------+
|InvoiceNo|StockCode|         Description|Quantity| InvoiceDate|UnitPrice|CustomerID|       Country|(quantity * unitprice)|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+----------------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/10 8:26|     2.55|     17850|United Kingdom|    15.299999999999999|
|   536365|    71053| WHITE METAL LANTERN|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|                 20.34|
|   536365|   84406B|CREAM CUPID HEART...|       8|12/1/10 8:26|     2.75|     17850|United Kingdom|                  22.0|
|   536365|   84029G|KNITTED UNION FLA...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|                 20.34|
|   536365|   84029E|RED WOOLLY HOTTIE...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|                 20.34|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+----------------------+
only showing top 5 rows
```

Command took 0.47 seconds -- by prashant@scholarnest.com at 5/24/2022, 8:45:14 PM on demo-cluster

Now, we have a requirement here to give an alias to the new column highlighted below. You can see the column name in the result. It doesn't look nice. We wanted to rename this column as “total.”
How will you do that in the column expression?

Cmd 5

```
1  from pyspark.sql.functions import col
2
3  retail_df.select("*", col("quantity") * col("unitprice")).show(5)
```

▸ (1) Spark Jobs

```
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+-----------------------+
|InvoiceNo|StockCode|         Description|Quantity| InvoiceDate|UnitPrice|CustomerID|       Country|(quantity * unitprice)|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+-----------------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/10 8:26|     2.55|     17850|United Kingdom|     15.299999999999999|
|   536365|    71053| WHITE METAL LANTERN|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|                  20.34|
|   536365|   84406B|CREAM CUPID HEART...|       8|12/1/10 8:26|     2.75|     17850|United Kingdom|                   22.0|
|   536365|   84029G|KNITTED UNION FLA...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|                  20.34|
|   536365|   84029E|RED WOOLLY HOTTIE...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|                  20.34|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+-----------------------+
only showing top 5 rows
```

Command took 0.47 seconds -- by prashant@scholarnest.com at 5/24/2022, 8:45:14 PM on demo-cluster

We can place a parenthesis around the expression, and you can give it an alias as shown below. This worked because the multiplication result is a column, I placed it inside a parenthesis and used the column API.

Cmd 5

```
1  from pyspark.sql.functions import col
2
3  retail_df.select("*", (col("quantity") * col("unitprice")).alias("total")).show(5)
```

▸ (1) Spark Jobs

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	total
536365	85123A	WHITE HANGING HEA...	6	12/1/10 8:26	2.55	17850	United Kingdom	15.299999999999999
536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84406B	CREAM CUPID HEART...	8	12/1/10 8:26	2.75	17850	United Kingdom	22.0
536365	84029G	KNITTED UNION FLA...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34

only showing top 5 rows

Command took 0.37 seconds -- by prashant@scholarnest.com at 5/24/2022, 8:47:42 PM on demo-cluster

I have a small expression here. But if your expression is becoming too big, you can define an expression variable and use it later. Here is an example for the same shown in the screenshot below.
I defined the expression and stored it in a variable. Then I am using the variable inside the select().

Cmd 6

```
1  total_ex = col("quantity") * col("unitprice")
2  retail_df.select("*", total_ex).show(5)
```

▸ (1) Spark Jobs

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	(quantity * unitprice)
536365	85123A	WHITE HANGING HEA...	6	12/1/10 8:26	2.55	17850	United Kingdom	15.299999999999999
536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84406B	CREAM CUPID HEART...	8	12/1/10 8:26	2.75	17850	United Kingdom	22.0
536365	84029G	KNITTED UNION FLA...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34

only showing top 5 rows

Command took 0.67 seconds -- by prashant@scholarnest.com at 5/24/2022, 8:50:46 PM on demo-cluster

Aliasing is more intuitive in this case, you can simply add an alias to the total_ex.
I often see people confused and frustrated with the many ways to define spark expressions. But do not worry too much. Learn one approach and stick to it in your project. But knowing other approaches will help you interact with code written by others.

Cmd 6

```
1  total_ex = col("quantity") * col("unitprice")
2  retail_df.select("*", total_ex.alias("total")).show(5)
```

▸ (1) Spark Jobs

```
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+------------------+
|InvoiceNo|StockCode|         Description|Quantity| InvoiceDate|UnitPrice|CustomerID|       Country|             total|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+------------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/10 8:26|     2.55|     17850|United Kingdom|15.299999999999999|
|   536365|    71053| WHITE METAL LANTERN|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|             20.34|
|   536365|   84406B|CREAM CUPID HEART...|       8|12/1/10 8:26|     2.75|     17850|United Kingdom|              22.0|
|   536365|   84029G|KNITTED UNION FLA...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|             20.34|
|   536365|   84029E|RED WOOLLY HOTTIE...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|             20.34|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+------------------+
only showing top 5 rows
```

Command took 0.37 seconds -- by prashant@scholarnest.com at 5/24/2022, 8:51:23 PM on demo-cluster

We have a rounding problem here. The total is coming as 10-12 digits. It is a currency number.
We can safely round it for two digits.

Cmd 6

```
1  total_ex = col("quantity") * col("unitprice")
2  retail_df.select("*", total_ex.alias("total")).show(5)
```

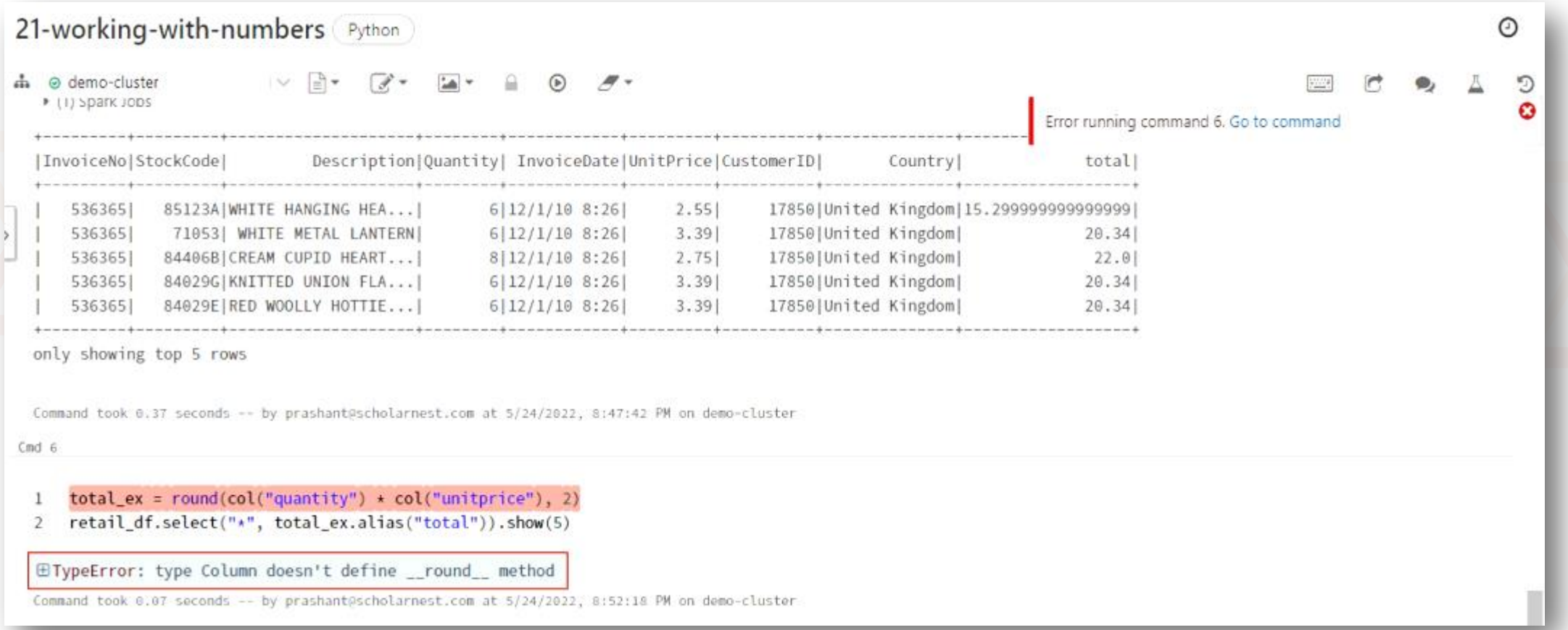
▸ (1) Spark Jobs

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	total
536365	85123A	WHITE HANGING HEA...	6	12/1/10 8:26	2.55	17850	United Kingdom	15.299999999999999
536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84406B	CREAM CUPID HEART...	8	12/1/10 8:26	2.75	17850	United Kingdom	22.0
536365	84029G	KNITTED UNION FLA...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34

only showing top 5 rows

Command took 0.37 seconds -- by prashant@scholarnest.com at 5/24/2022, 8:51:23 PM on demo-cluster

I can modify the expression and apply a round() function to the expression. But you can see that we got an error if we run the cell.
Why? Because I forgot to import the round function.



So far, we keep importing functions as and when we need them.
I need expr() function. So I will write code like this:

from pyspark.sql.functions import expr

If I need a round() function, I will add the round() function to the same list.

from pyspark.sql.functions import expr, round

Some people are tired of keeping on adding functions to this list. So they take a shortcut. They remove all the functions from the list and import *. You can see that this works, and you can see the output. But importing * is a bad habit, and it overwrites the python namespace.

Cmd 6

```
1  from pyspark.sql.functions import *
2
3  total_ex = round(col("quantity") * col("unitprice"), 2)
4  retail_df.select("*", total_ex.alias("total")).show(5)
```

▸ (1) Spark Jobs

```
+---------+---------+--------------------+--------+-------------+---------+----------+--------------+-----+
|InvoiceNo|StockCode|         Description|Quantity| InvoiceDate|UnitPrice|CustomerID|       Country|total|
+---------+---------+--------------------+--------+-------------+---------+----------+--------------+-----+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/10 8:26|     2.55|     17850|United Kingdom| 15.3|
|   536365|    71053| WHITE METAL LANTERN|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|20.34|
|   536365|   84406B|CREAM CUPID HEART...|       8|12/1/10 8:26|     2.75|     17850|United Kingdom| 22.0|
|   536365|   84029G|KNITTED UNION FLA...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|20.34|
|   536365|   84029E|RED WOOLLY HOTTIE...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|20.34|
+---------+---------+--------------------+--------+-------------+---------+----------+--------------+-----+
only showing top 5 rows
```

Command took 0.60 seconds -- by prashant@scholarnest.com at 5/24/2022, 8:54:11 PM on demo-cluster

You can see that I added another line of plain python code at the bottom of the cell. When I try running the cell, the last line throws an error stating 'Invalid argument.' So, you have two round() functions. One is given to you by the Python standard packages. And we have another one inside the *pyspark.sql.functions* package. If you import * from the *pyspark.SQL.functions*. The Python round() function is lost. It is not only the Python round() function but all the Python built-in functions are lost. You cannot use them. And that is why we import only what we need and do not import *.

However, importing one function is also not a good practice.
Why? Because we are still overwriting.
I mean, I am importing round() function from the *pyspark.sql.functions*.
So I am overwriting the Python round() function.
I am doing better by not overwriting all the functions.
But I am still overwriting the round() function.
So what is the best practice?

Import your pyspark.SQL.functions with an alias.

Go to the Clear menu and clear the notebook state. So whatever we imported so far is now clear, and we have a clean start. Now re-create the data frame because I cleared the state, so my Dataframe was also gone.

21-working-with-numbers Python

demo-cluster

Cmd 1

```
1  retail_df = spark.read \
2                  .format("csv") \
3                  .option("header", "true") \
4                  .option("inferSchema", "true") \
5                  .load("/databricks-datasets/online_retail/data-001/data.csv")
6
7  retail_df.printSchema()
8  display(retail_df)
```

(3) Spark Jobs

```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: string (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: integer (nullable = true)
 |-- Country: string (nullable = true)
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/10 8:26	2.55	17850	United Kingdom
2	536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom
3	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/10 8:26	2.75	17850	United Kingdom

Here is the best practice. I removed the old import and changed it to import functions as sf. The SF stands for spark functions. You can give it whatever name, but I wanted to call it sf. This import statement will import all the spark functions. It is the same as import *. However, everything is imported under the SF namespace. And you can use this namespace in your code. So my spark round function becomes sf.round. Similarly, my spark col() function becomes sf.col().

Cmd 6

```
1  import pyspark.sql.functions as sf
2
3  total_ex = sf.round(sf.col("quantity") * sf.col("unitprice"), 2)
4  retail_df.select("*", total_ex.alias("total")).show(5)
5  print(round(5.20356))
```

▸ (1) Spark Jobs

```
+---------+---------+--------------------+--------+-------------+---------+----------+--------------+-----+
|InvoiceNo|StockCode|         Description|Quantity| InvoiceDate|UnitPrice|CustomerID|       Country|total|
+---------+---------+--------------------+--------+-------------+---------+----------+--------------+-----+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/10 8:26|     2.55|     17850|United Kingdom| 15.3|
|   536365|    71053| WHITE METAL LANTERN|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|20.34|
|   536365|   84406B|CREAM CUPID HEART...|       8|12/1/10 8:26|     2.75|     17850|United Kingdom| 22.0|
|   536365|   84029G|KNITTED UNION FLA...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|20.34|
|   536365|   84029E|RED WOOLLY HOTTIE...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|20.34|
+---------+---------+--------------------+--------+-------------+---------+----------+--------------+-----+
only showing top 5 rows
```


So, I am importing all my spark functions into the SF namespace. I wanted to use the spark round() function. And I can use it using the SF namespace. The last line wanted to use the standard python round() function. And we can use it without any namespace. So now, both the round functions will work. And you can see the output below the cell.
So I recommend importing your spark functions under a new namespace.

Cmd 6

```
1  import pyspark.sql.functions as sf
2
3  total_ex = sf.round(sf.col("quantity") * sf.col("unitprice"), 2)
4  retail_df.select("*", total_ex.alias("total")).show(5)
5  print(round(5.20356))
```

▸ (1) Spark Jobs

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	total
536365	85123A	WHITE HANGING HEA...	6	12/1/10 8:26	2.55	17850	United Kingdom	15.3
536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84406B	CREAM CUPID HEART...	8	12/1/10 8:26	2.75	17850	United Kingdom	22.0
536365	84029G	KNITTED UNION FLA...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/10 8:26	3.39	17850	United Kingdom	20.34

only showing top 5 rows

Now let me introduce you to the Spark stats package.
So you can see the summary of the Dataframe. The summary method() is quite handy in analysing your Dataframe. It gives you count, min, max, and percentiles which indicates the nature of your data.

Cmd 7

```
1  retail_df.summary().show()
```

▸ (2) Spark Jobs

summary	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
count	65499	65499	65333	65499	65499	65499	40218	65499
mean	539091.6921058759	29165.654135469875	null	8.366234599001512	null	5.857585611994696	15384.033517330548	null
stddev	1586.8350514333126	19298.622465903674	null	413.80812814338367	null	145.79596265581822	1766.8634991790627	null
min	536365	10002	4 PURPLE FLOCK D...	-74215	1/10/11 10:04	0.0	12346	Australia
25%	537646.0	21722.0	null	1	null	1.25	13999	null
50%	539118.0	22350.0	null	2	null	2.51	15358	null
75%	540508.0	22767.0	null	8	null	4.24	17019	null
max	C541694	m	reverse 21/5/10 a...	74215	12/9/10 9:49	16888.02	18283	United Kingdom

Command took 13.29 seconds -- by prashant@scholarnest.com at 5/26/2022, 4:56:16 PM on demo-cluster

For example, I know the invoice number has some string values. Look at the max value of the invoice number. The column looks like an integer, but we have string data.

Cmd 7

```
1  retail_df.summary().show()
```

▸ (2) Spark Jobs

summary	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
count	65499	65499	65333	65499	65499	65499	40218	65499
mean	539091.6921058759	29165.654135469875	null	8.366234599001512	null	5.857585611994696	15384.033517330548	null
stddev	1586.8350514333126	19298.622465903674	null	413.80812814338367	null	145.79596265581822	1766.8634991790627	null
min	536365	10002	4 PURPLE FLOCK D...	-74215	1/10/11 10:04	0.0	12346	Australia
25%	537646.0	21722.0	null	1	null	1.25	13999	null
50%	539118.0	22350.0	null	2	null	2.51	15358	null
75%	540508.0	22767.0	null	8	null	4.24	17019	null
max	C541694	m	reverse 21/5/10 a...	74215	12/9/10 9:49	16888.02	18283	United Kingdom

Similarly, look at the quantity. We have two outliers there. One is a negative number, and another is a large positive number. These guys indicate dirty data, and the summary is super helpful in identifying these types of problems.

Cmd 7

```
1  retail_df.summary().show()
```

▸ (2) Spark Jobs

summary	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
count	65499	65499	65333	65499	65499	65499	40218	65499
mean	539091.6921058759	29165.654135469875	null	8.366234599001512	null	5.857585611994696	15384.033517330548	null
stddev	1586.8350514333126	19298.622465903674	null	413.80812814338367	null	145.79596265581822	1766.8634991790627	null
min	536365	10002	4 PURPLE FLOCK D...	-74215	1/10/11 10:04	0.0	12346	Australia
25%	537646.0	21722.0	null	1	null	1.25	13999	null
50%	539118.0	22350.0	null	2	null	2.51	15358	null
75%	540508.0	22767.0	null	8	null	4.24	17019	null
max	C541694	m	reverse 21/5/10 a...	74215	12/9/10 9:49	16888.02	18283	United Kingdom

You can also take out specific stats if you do not want the full summary as shown below.
I am simply taking min, max, and 99th percentile of unit price. And it indicates a potential data problem. The 99th percentile of unit price is approximately 20 bugs, and the max value is more than 16K, which is odd.

Cmd 8

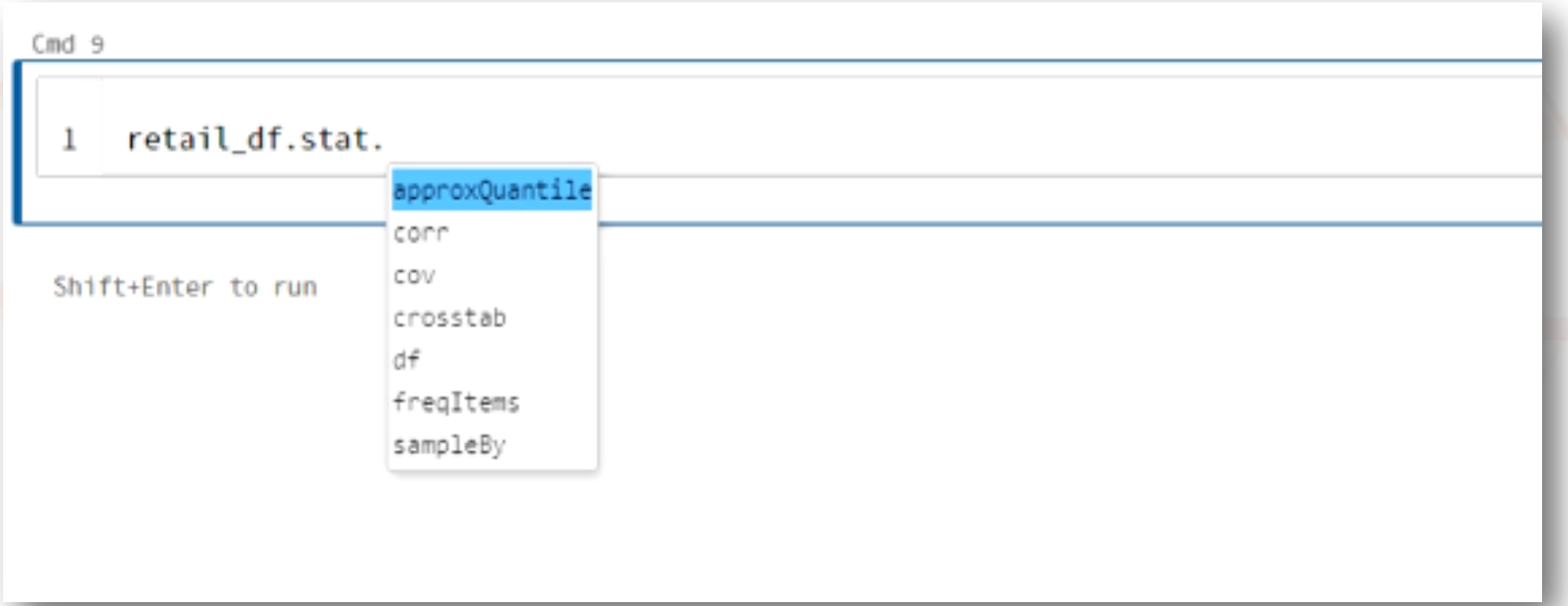
```
1  retail_df.select(sf.min("unitprice"), sf.max("unitprice"), sf.percentile_approx("unitprice", 0.99)).show()
```

▸ (2) Spark Jobs

```
+--------------+--------------+-----------------------------------------+
|min(unitprice)|max(unitprice)|percentile_approx(unitprice, 0.99, 10000)|
+--------------+--------------+-----------------------------------------+
|           0.0|      16888.02|                                    20.38|
+--------------+--------------+-----------------------------------------+
```

Command took 2.39 seconds -- by prashant@scholarnest.com at 5/26/2022, 5:00:23 PM on demo-cluster

Spark also offers you a bunch of stat functions. And those stat functions could be handy. You can see some stats functions shown below. Just type the Dataframe name dot stat and press tab. And you will see some stat methods.



I am using one of the stat functions, ‘freqItems.’
I am counting the frequency of country and description columns at 30%. It means, the frequency items will give me a list of countries that shows up 30% of the time in this Dataframe.
And you can see the result, 30% of the Dataframe is made up of only three countries.

```
Cmd 9

1  retail_df.stat.freqItems(["country"], 0.3).show(truncate=False)

▸ (1) Spark Jobs

+---------------------------------+
|country_freqItems                |
+---------------------------------+
|[France, Germany, United Kingdom]|
+---------------------------------+


Command took 1.53 seconds -- by prashant@scholarnest.com at 5/26/2022, 5:02:11 PM on demo-cluster
```


Now, let's assume I have a Dataframe like this.
The name is the customer name, and the item is the purchased item.
So Alice bought milk. Then Bob bought bread. Then Alice again purchased apples.
You can go on like this.
So this Dataframe represents a typical transaction table where you are interested in two columns.

name	item
Alice	milk
Bob	bread
Mike	butter
Alice	apples
Bob	oranges
Mike	milk

And now you want to create a cross tab summary of the columns as shown below. It indicates that Mike bought apples seven times, bread six times, butter and milk seven times, and oranges six times.

I have taken a super simple example Dataframe of only two columns.

And that's more than enough to give you an idea of what you can do with the crosstab summary. Calculating crosstab is super easy in Spark.

name_item	apples	bread	butter	milk	oranges
Mike	7	6	7	7	6
Alice	7	7	6	7	7
Bob	6	7	7	6	7

I have dynamically generated a data list of 100 records as shown below.
I am creating a list of three names.
Then I create another list of five items.
Finally, I am using Python list comprehension to create a list of hundred tuples. It is as simple as running a loop a hundred times.
And each time, I take one name and an item to create one record in the list. Like that, I will have a hundred records.

Cmd 10

```
1  names = ["Alice", "Bob", "Mike"]
2  items = ["milk", "bread", "butter", "apples", "oranges"]
3
4  data_list = [(names[i % 3], items[i % 5]) for i in range(100)]
```


Then I can take this data_list to create a Spark Dataframe as follows. In the earlier lessons, I typed each element of the Python list manually. But this time, I used Python code to generate it at runtime.

Cmd 10

```
1  names = ["Alice", "Bob", "Mike"]
2  items = ["milk", "bread", "butter", "apples", "oranges"]
3
4  data_list = [(names[i % 3], items[i % 5]) for i in range(100)]
5
6  sample_df = spark.createDataFrame(data_list).toDF("names", "items")
7  display(sample_df)
```

▸ (3) Spark Jobs

	names	items
1	Alice	milk
2	Bob	bread
3	Mike	butter
4	Alice	apples
5	Bob	oranges
6	Mike	milk
7	Alice	bread

Showing all 100 rows.

Command took 1.43 seconds -- by prashant@scholarnest.com at 5/26/2022, 5:15:14 PM on demo-cluster

Now you want to create a crosstab summary as shown below.
This crosstab summary is also known as a frequency table. It is sometimes also known as a pivot table. However, a pivot is more flexible. We will learn pivoting when talking about grouping and aggregation in the Spark. However, crosstab is a simple two-column frequency table.
And you can create it using the stat.crosstab() function.

name_item	apples	bread	butter	milk	oranges
Mike	7	6	7	7	6
Alice	7	7	6	7	7
Bob	6	7	7	6	7

Here is the code for the same, and you can see the desired output below the cell.
The crosstab() method takes only two arguments. The first column becomes the row values.
And the second column becomes the column names. And the values are the count.
The crosstab is not as flexible as pivoting, but it can create a frequency table more efficiently.

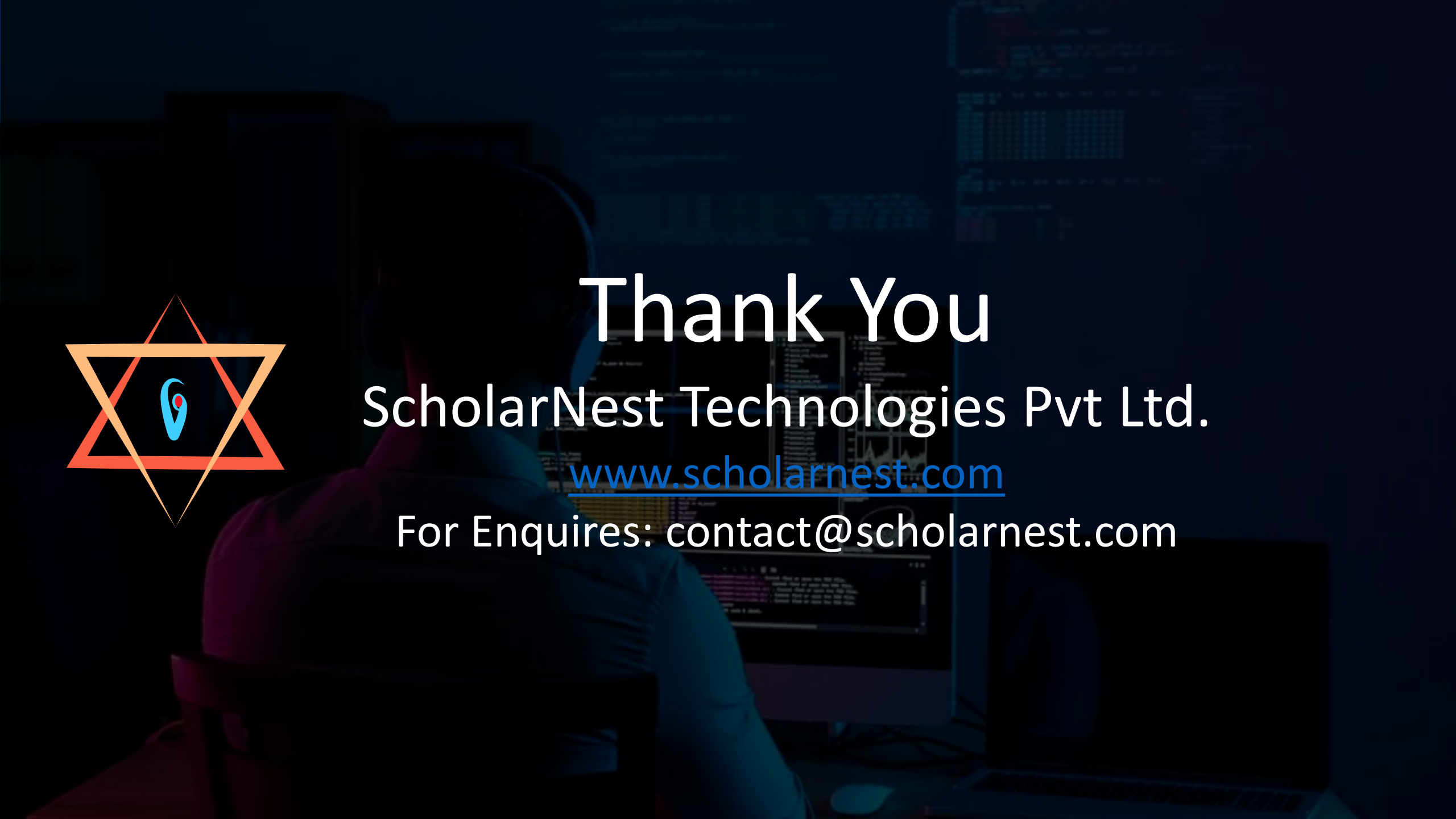
Cmd 11

```
1  sample_df.stat.crosstab("names", "items").show()
```

▸ (2) Spark Jobs

```
+-----------+------+-----+------+----+-------+
|names_items|apples|bread|butter|milk|oranges|
+-----------+------+-----+------+----+-------+
|       Mike|     7|    6|     7|   7|      6|
|      Alice|     7|    7|     6|   7|      7|
|        Bob|     6|    7|     7|   6|      7|
+-----------+------+-----+------+----+-------+
```

Command took 2.74 seconds -- by prashant@scholarnest.com at 5/26/2022, 5:17:18 PM on demo-cluster



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

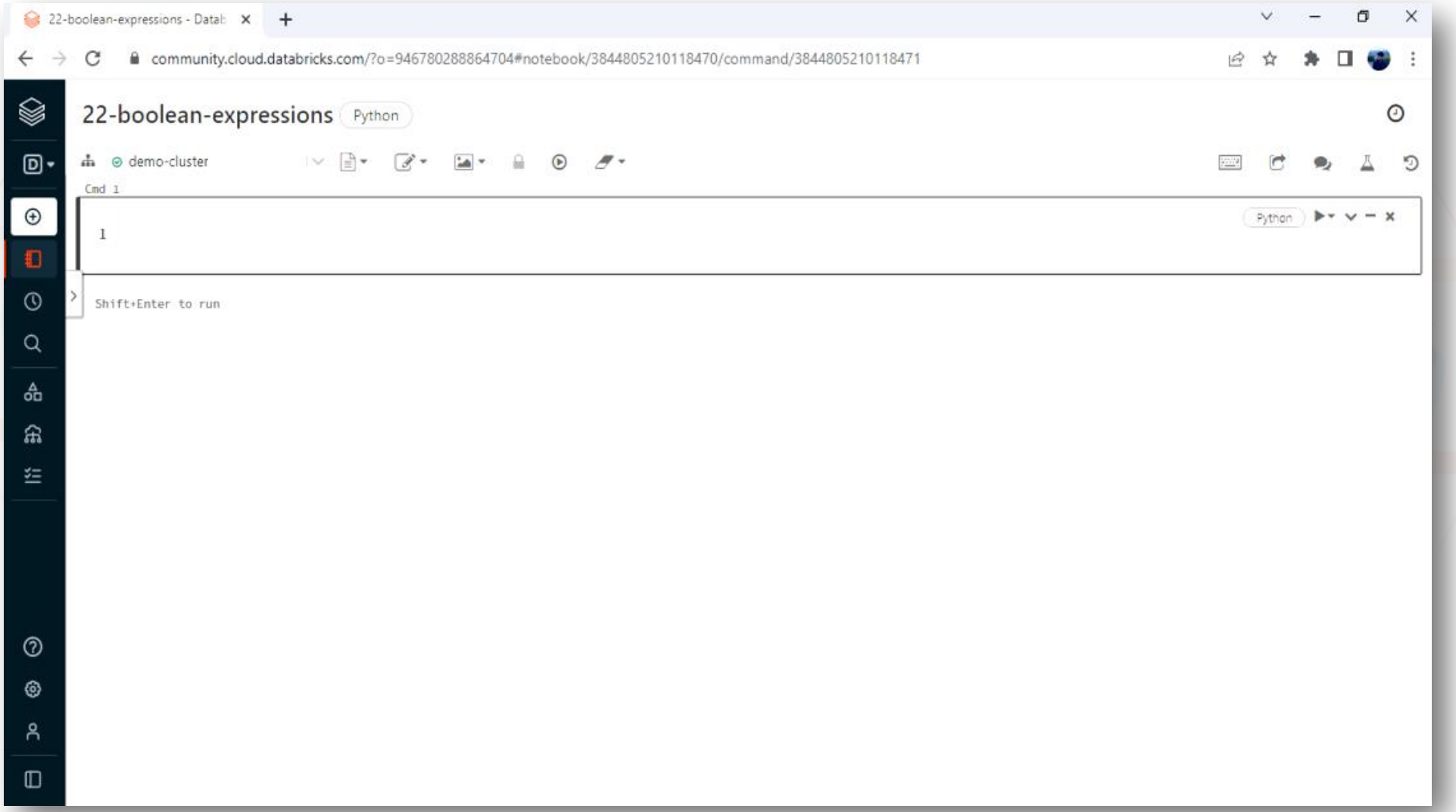
Databricks Spark Certification and Beyond

Module:
Working with Different Data Types

Lecture:
Boolean Expressions

Boolean Expressions

Go to your Databricks workspace and create a new notebook. **(Reference : 22-boolean-expressions)**



We need a Dataframe to look at some of the examples, so I have created a Dataframe below.

Cmd 1

```
1  retail_df = spark.read \
2              .format("csv") \
3              .option("header", "true") \
4              .option("inferSchema", "true") \
5              .load("/databricks-datasets/online_retail/data-001/data.csv")
6
7  display(retail_df)
```

▸ (3) Spark Jobs

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/10 8:26	2.55	17850	United Kingdom
2	536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom
3	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/10 8:26	2.75	17850	United Kingdom
4	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/10 8:26	3.39	17850	United Kingdom
5	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	12/1/10 8:26	3.39	17850	United Kingdom
6	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	12/1/10 8:26	7.65	17850	United Kingdom
7	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	12/1/10 8:26	4.25	17850	United Kingdom

Truncated results, showing first 1000 rows.

Command took 3.56 seconds -- by prashant@scholarnest.com at 5/26/2022, 5:30:40 PM on demo-cluster

Boolean expressions are primarily used in the where() method or filter() method. Here is an example to filter the given Dataframe and select records where InvoiceNo equals '536365.' You can use SQL-like expressions in the where() method to filter records. I used the equality operator to check if the InvoiceNo equals the given value.

Cmd 2

```
1  retail_df.where("InvoiceNo==536365").show()
```

▸ (2) Spark Jobs

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
536365	85123A	WHITE HANGING HEA...	6	12/1/10 8:26	2.55	17850	United Kingdom
536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom
536365	84406B	CREAM CUPID HEART...	8	12/1/10 8:26	2.75	17850	United Kingdom
536365	84029G	KNITTED UNION FLA...	6	12/1/10 8:26	3.39	17850	United Kingdom
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/10 8:26	3.39	17850	United Kingdom
536365	22752	SET 7 BABUSHKA NE...	2	12/1/10 8:26	7.65	17850	United Kingdom
536365	21730	GLASS STAR FROSTE...	6	12/1/10 8:26	4.25	17850	United Kingdom

You have a lot of operators and functions that help you create a variety of Boolean expressions. Here is a summary of those operators and functions.

S.No.	Purpose	SQL Operator	Column Operator
1.	Less than	<	<
2.	Less than or equal	<=	<=
3.	Greater than	>	>
4.	Greater than or equal	>=	>=
5.	Equal	=, ==	==
6.	Null safe equal	<=>	<=>
7.	Negation	!	!
8.	Null check	IS NULL, isnull()	IS NULL, isnull()
9.	Not Null check	IS NOT NULL, isnotnull()	IS NOT NULL, isnotnull()
10.	Logical AND	AND	&
11.	Logical OR	OR	|
12.	Inclusion	in()	isin()
13.	String match	LIKE	like(), contains(), startswith(), endswith()
14.	Between	BETWEEN	between()
15.	Regular expressions	rlike()	rlike()

So I found 15 items to include in the list. You already know these because you know SQL or Python programming. But I wanted you to notice and remember the difference between SQL expressions and Column expressions operators.

S.No.	Purpose	SQL Operator	Column Operator
1.	Less than	<	<
2.	Less than or equal	<=	<=
3.	Greater than	>	>
4.	Greater than or equal	>=	>=
5.	Equal	=, ==	==
6.	Null safe equal	<=>	<=>
7.	Negation	!	!
8.	Null check	IS NULL, isnull()	IS NULL, isnull()
9.	Not Null check	IS NOT NULL, isnotnull()	IS NOT NULL, isnotnull()
10.	Logical AND	AND	&
11.	Logical OR	OR	|
12.	Inclusion	in()	isin()
13.	String match	LIKE	like(), contains(), startswith(), endswith()
14.	Between	BETWEEN	between()
15.	Regular expressions	rlike()	rlike()

SQL expressions support single = and double = both for equality check. However, a column expression supports only == for the same.
Similarly, the logical AND and logical OR are different. The LIKE and BETWEEN are keywords in SQL expressions. However, they are functions in column expression.

S.No.	Purpose	SQL Operator	Column Operator
1.	Less than	<	<
2.	Less than or equal	<=	<=
3.	Greater than	>	>
4.	Greater than or equal	>=	>=
5.	Equal	=, ==	==
6.	Null safe equal	<=>	<=>
7.	Negation	!	!
8.	Null check	IS NULL, isnull()	IS NULL, isnull()
9.	Not Null check	IS NOT NULL, isnotnull()	IS NOT NULL, isnotnull()
10.	Logical AND	AND	&
11.	Logical OR	OR	|
12.	Inclusion	in()	isin()
13.	String match	LIKE	like(), contains(), startswith(), endswith()
14.	Between	BETWEEN	between()
15.	Regular expressions	rlike()	rlike()

Now come back to the notebook. So we applied an equality check using the == symbol. It is a SQL expression, so you can also use a single =, which will work.

Cmd 2

```
1  retail_df.where("InvoiceNo==536365").show()
```

▸ (2) Spark Jobs

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
536365	85123A	WHITE HANGING HEA...	6	12/1/10 8:26	2.55	17850	United Kingdom
536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom
536365	84406B	CREAM CUPID HEART...	8	12/1/10 8:26	2.75	17850	United Kingdom
536365	84029G	KNITTED UNION FLA...	6	12/1/10 8:26	3.39	17850	United Kingdom
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/10 8:26	3.39	17850	United Kingdom
536365	22752	SET 7 BABUSHKA NE...	2	12/1/10 8:26	7.65	17850	United Kingdom
536365	21730	GLASS STAR FROSTE...	6	12/1/10 8:26	4.25	17850	United Kingdom

1

Command took 1.74 seconds -- by prashant@scholarnest.com at 5/26/2022, 5:37:51 PM on demo-cluster

Cmd 3

You can see that we got the output using single = as well.

Cmd 2

```
1  retail_df.where("InvoiceNo=536365").show()
```

▸ (2) Spark Jobs

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
536365	85123A	WHITE HANGING HEA...	6	12/1/10 8:26	2.55	17850	United Kingdom
536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom
536365	84406B	CREAM CUPID HEART...	8	12/1/10 8:26	2.75	17850	United Kingdom
536365	84029G	KNITTED UNION FLA...	6	12/1/10 8:26	3.39	17850	United Kingdom
536365	84029E	RED WOOLLY HOTTIE...	6	12/1/10 8:26	3.39	17850	United Kingdom
536365	22752	SET 7 BABUSHKA NE...	2	12/1/10 8:26	7.65	17850	United Kingdom
536365	21730	GLASS STAR FROSTE...	6	12/1/10 8:26	4.25	17850	United Kingdom

1

Command took 1.58 seconds -- by prashant@scholarnest.com at 5/26/2022, 5:44:35 PM on demo-cluster

You can implement the same condition using a column expression as well.

Cmd 3

```
1  import pyspark.sql.functions as sf
2
3  retail_df.where(sf.col("InvoiceNo")==536365).show()
```

▸ (2) Spark Jobs

```
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+
|InvoiceNo|StockCode|         Description|Quantity| InvoiceDate|UnitPrice|CustomerID|       Country|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/10 8:26|     2.55|     17850|United Kingdom|
|   536365|    71053| WHITE METAL LANTERN|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|
|   536365|   84406B|CREAM CUPID HEART...|       8|12/1/10 8:26|     2.75|     17850|United Kingdom|
|   536365|   84029G|KNITTED UNION FLA...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|
|   536365|   84029E|RED WOOLLY HOTTIE...|       6|12/1/10 8:26|     3.39|     17850|United Kingdom|
|   536365|    22752|SET 7 BABUSHKA NE...|       2|12/1/10 8:26|     7.65|     17850|United Kingdom|
|   536365|    21730|GLASS STAR FROSTE...|       6|12/1/10 8:26|     4.25|     17850|United Kingdom|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+
```

1

Command took 2.51 seconds -- by prashant@scholarnest.com at 5/26/2022, 7:59:36 PM on demo-cluster

Now I want to filter records where InvoiceNo is 536365 and StockCode is 85123A. And we have the code for this given below.

Cmd 4

```
1  retail_df.where("InvoiceNo==536365 AND StockCode = '85123A'") \
2           .show()
```

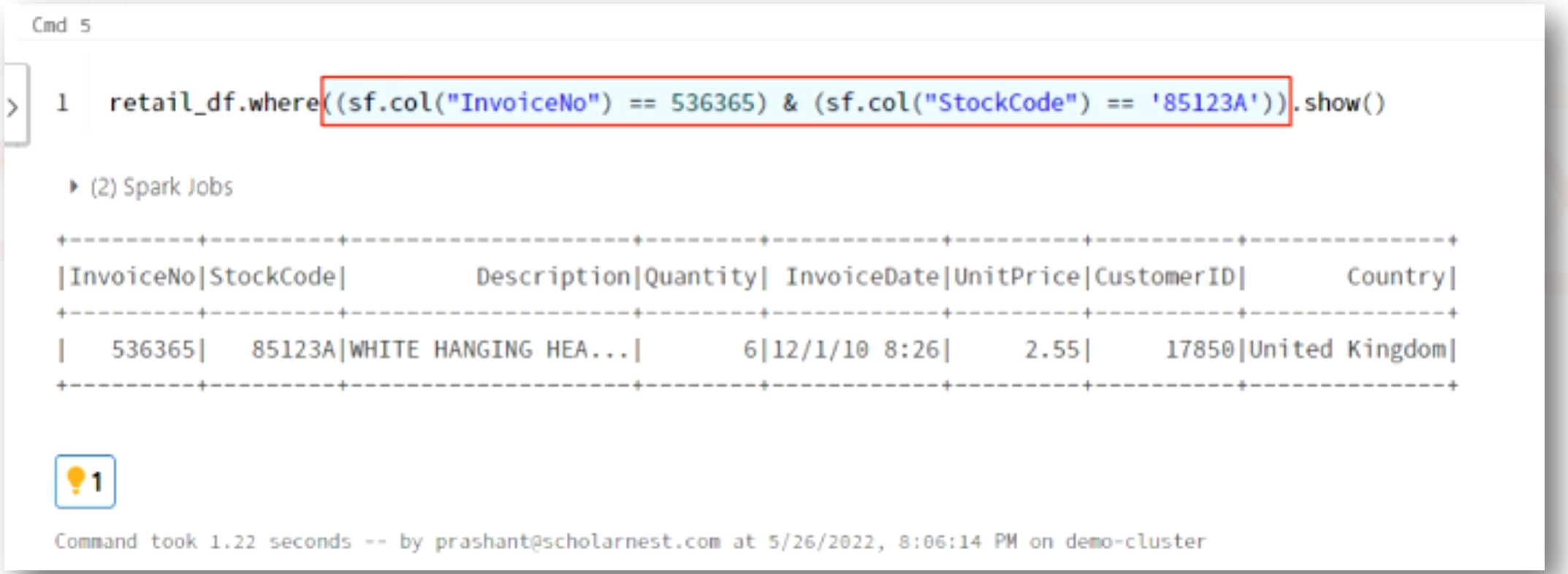
▸ (2) Spark Jobs

```
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+
|InvoiceNo|StockCode|         Description|Quantity| InvoiceDate|UnitPrice|CustomerID|       Country|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/10 8:26|     2.55|     17850|United Kingdom|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+
```

1

Command took 1.88 seconds -- by prashant@scholarnest.com at 5/26/2022, 8:01:05 PM on demo-cluster

Here is the same example using column expressions.
I am enclosing both the conditions in a pair of parenthesis. Don't forget to do it. You will see errors If you remove or forget the parenthesis. When combining multiple conditions in a column expression, it is a best practice to enclose them in a pair of parenthesis. You might see errors if you forget that.



If you have a long-expression, you can create variables and use them in the where method as shown below. So, what am I doing here?
I defined a filter condition for invoice number. Then I defined another condition for stock code. Finally, I applied both of these conditions using the AND operator.

Cmd 6

```
1  invoice_filter = sf.col("InvoiceNo") == 536365
2  stock_filter = retail_df.StockCode == '85123A'
3
4  retail_df.where(invoice_filter & stock_filter).show()
```

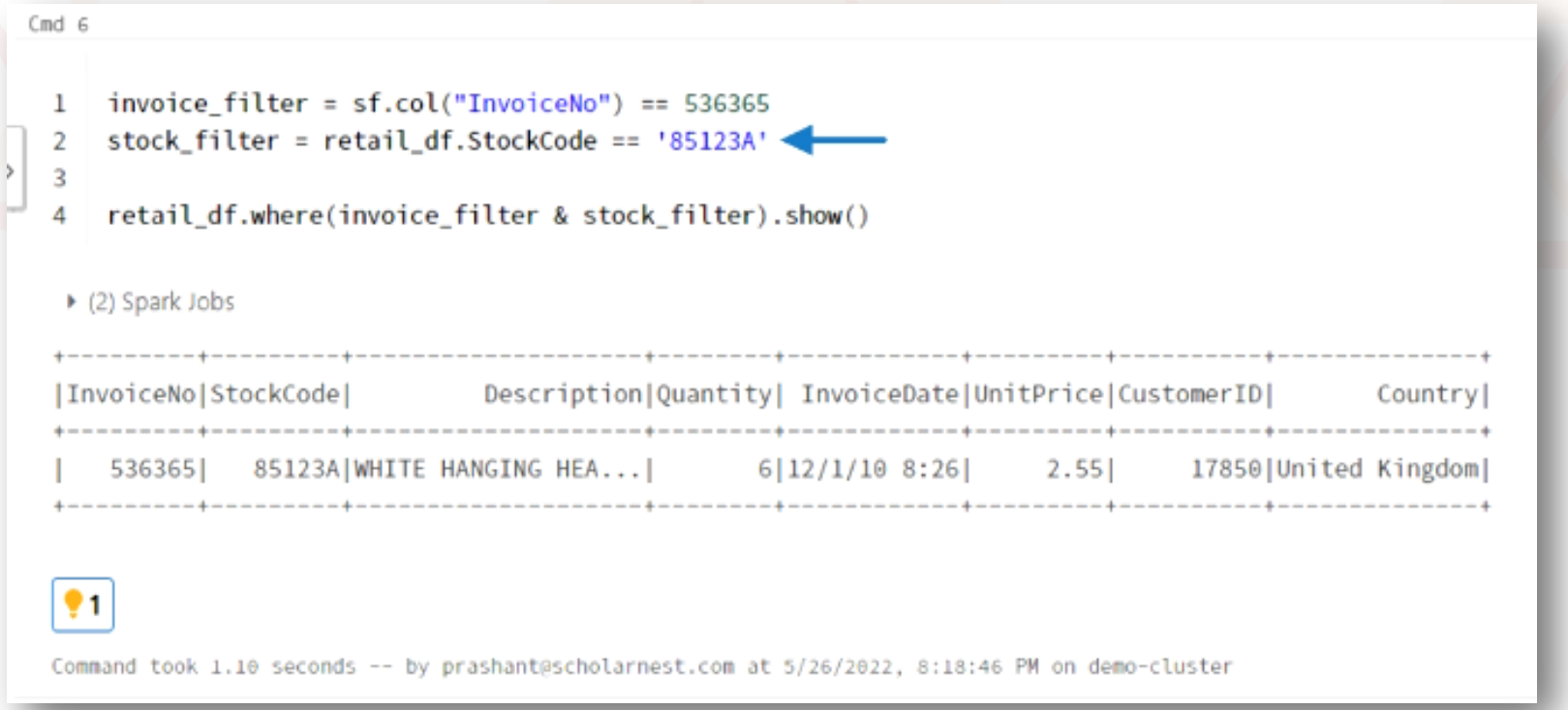
▸ (2) Spark Jobs

```
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+
|InvoiceNo|StockCode|         Description|Quantity| InvoiceDate|UnitPrice|CustomerID|       Country|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/10 8:26|     2.55|     17850|United Kingdom|
+---------+---------+--------------------+--------+-----------+---------+----------+--------------+
```

1

Command took 1.10 seconds -- by prashant@scholarnest.com at 5/26/2022, 8:18:46 PM on demo-cluster

I used the Dataframe dot column notation here. I told you multiple ways to refer to a dataframe column. We can use the col() function as I used here. We can also use dot notation to refer to a column name. Both are fine, and they work. However, we do not prefer using dot notation in practice. I always prefer using the col() function. Because the dot notation is case-sensitive. Look at this code, I wrote *retail_df.StockCode*. I am using the exact case for the stock code column defined in the Dataframe schema. If you change the upper case to lower, this code will throw an error. The col() function is great. The column name in the col() function is not case-sensitive. But the dot notation is case sensitive. And that is why I do not prefer using dot notation.



Look at the code below. I am chaining two where() methods. And Spark will apply both the conditions.
But we haven't specified how to combine these two conditions.
Will it be a logical AND? Will it be a logical OR?
That’s for you to brainstorm and find out.

Cmd 7

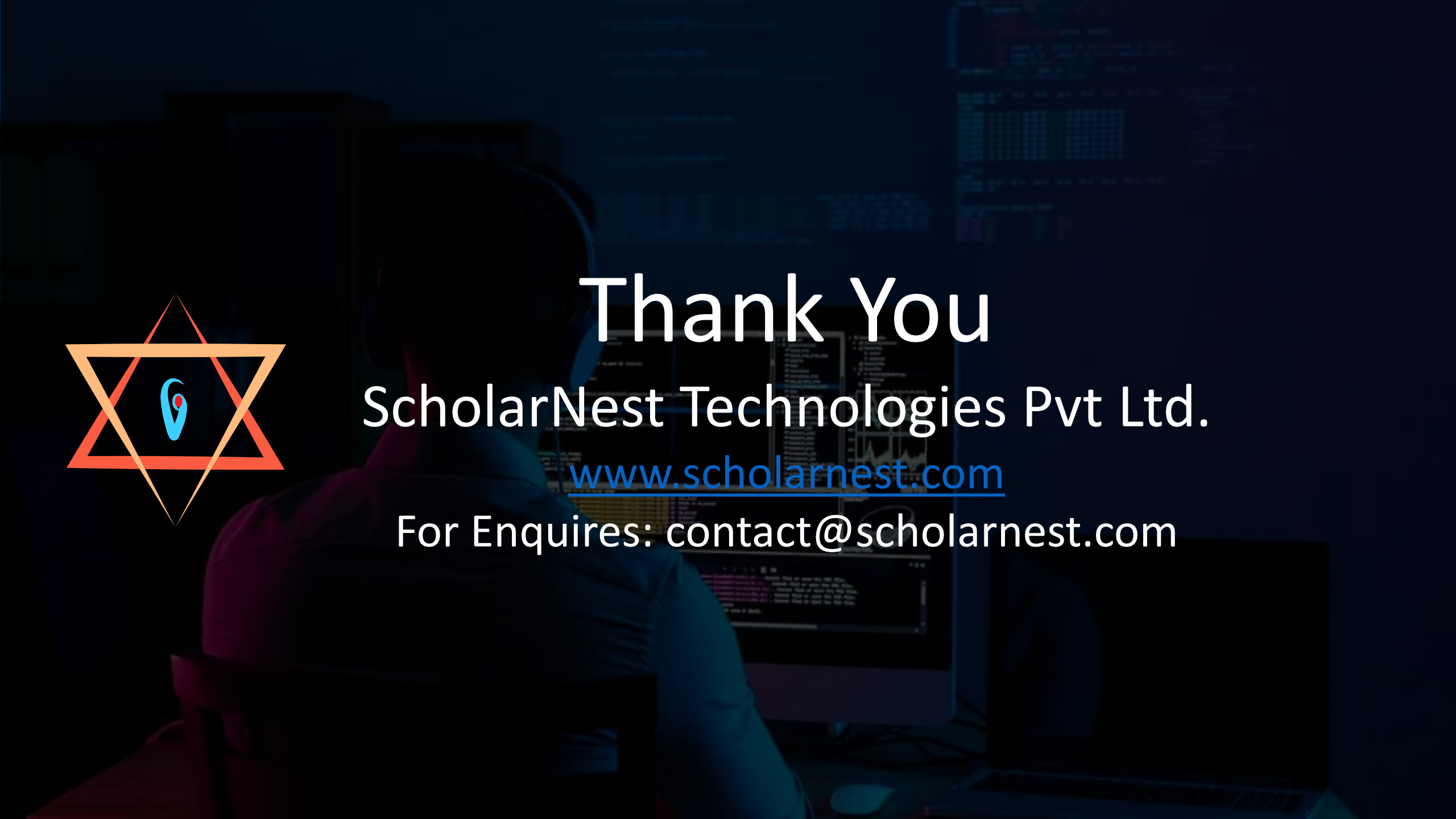
```
1  retail_df.where("InvoiceNo==536365") \
2           .where("StockCode=='85123A'").show()
```

▸ (2) Spark Jobs

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
536365	85123A	WHITE HANGING HEA...	6	12/1/10 8:26	2.55	17850	United Kingdom

1

Command took 0.79 seconds -- by prashant@scholarnest.com at 5/26/2022, 8:20:55 PM on demo-cluster



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

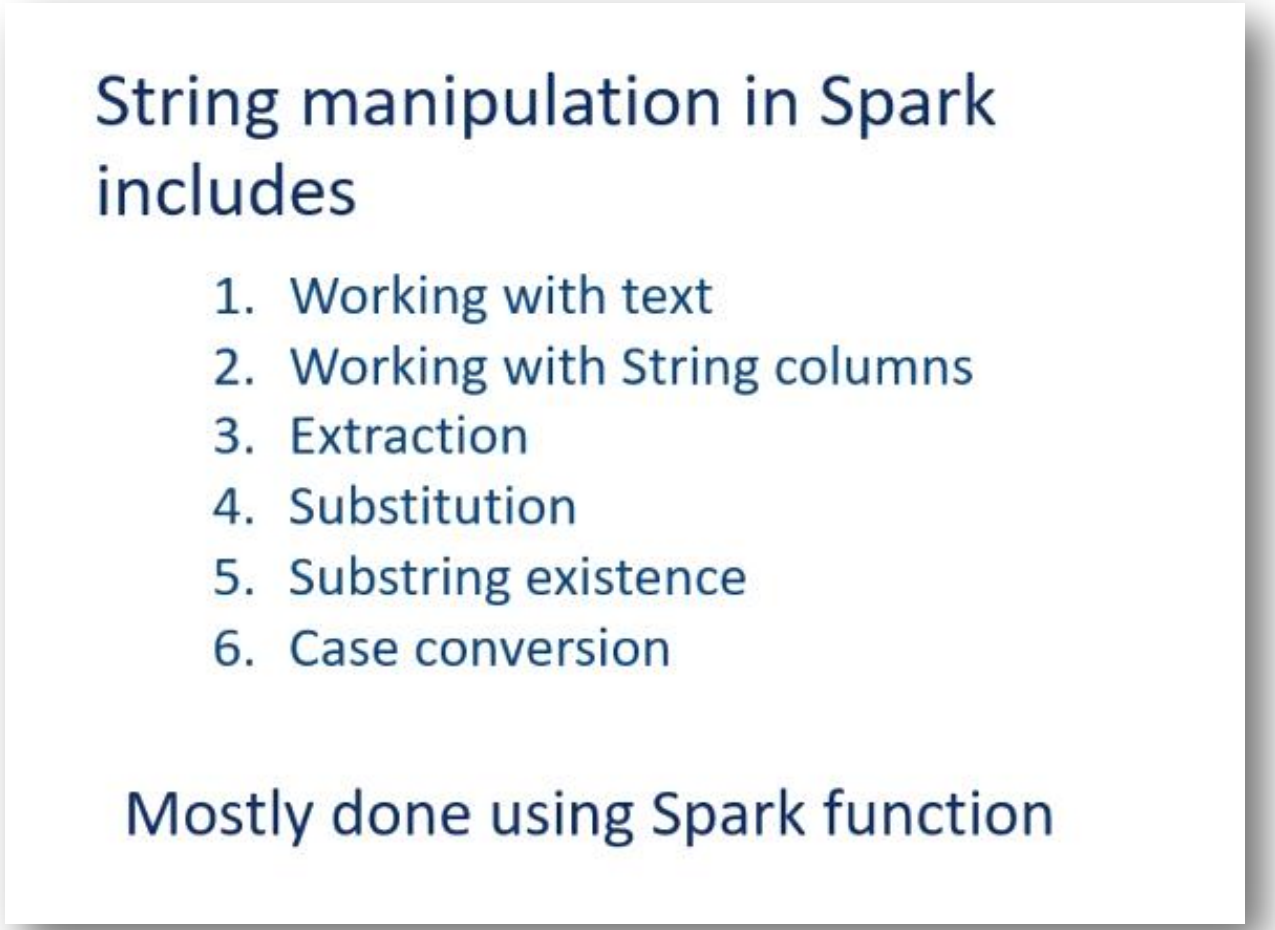
Databricks Spark Certification and Beyond

Module:
Working with Different Data Types

Lecture:
Manipulating Strings

Manipulating Strings

String manipulation is one of the most common requirements in any data system. You might be manipulating text data or string columns for various purposes listed below.
We have a lot of things to do with the strings. And almost all of those are done with the help of string manipulation functions.

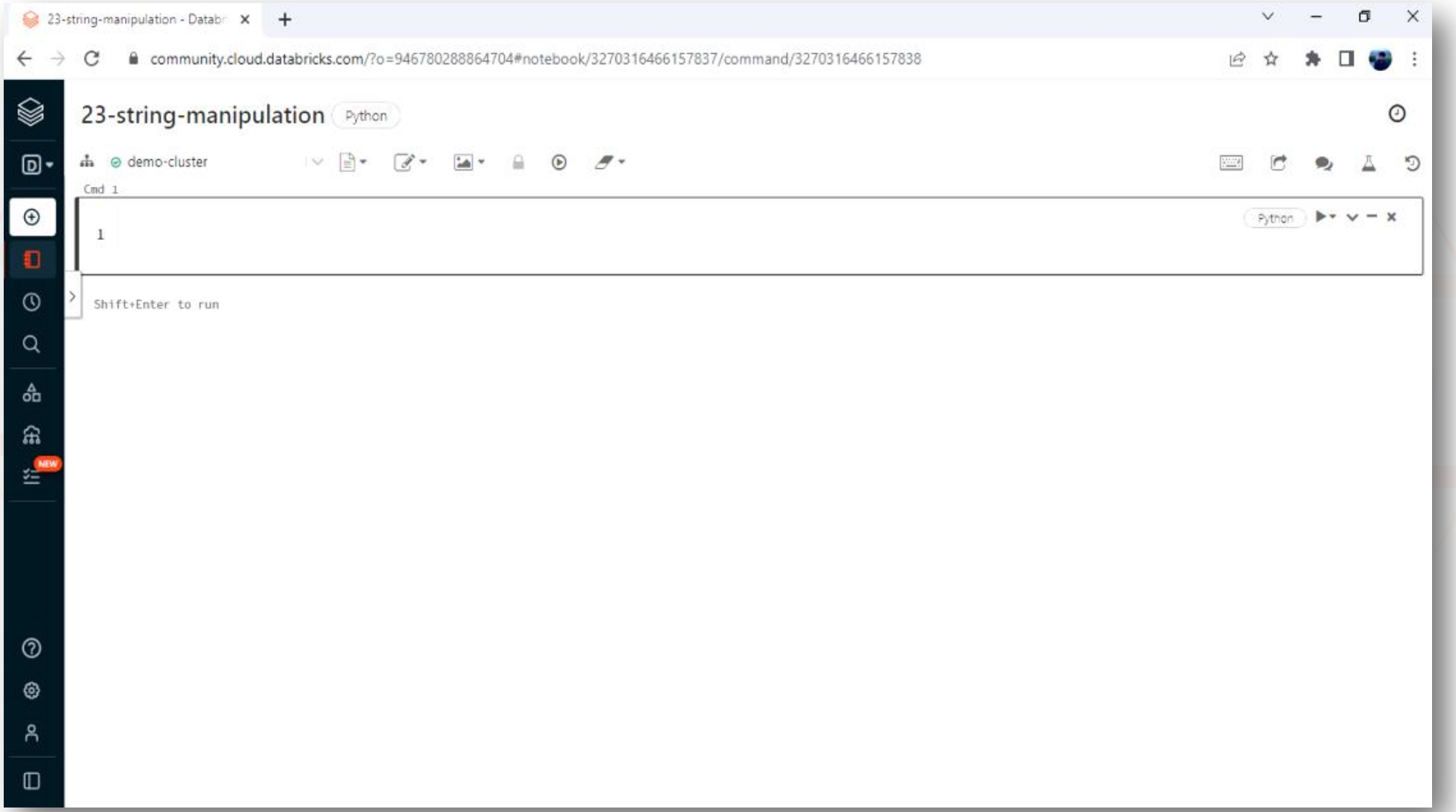


Spark offers you a wealth of string manipulation functions. Here is a list of the most common string functions listed below. I have listed eleven items here. All the functions listed here are available as dataframe functions, and they are also built-in SQL functions. So you can use them easily in Spark SQL and your Dataframe code.

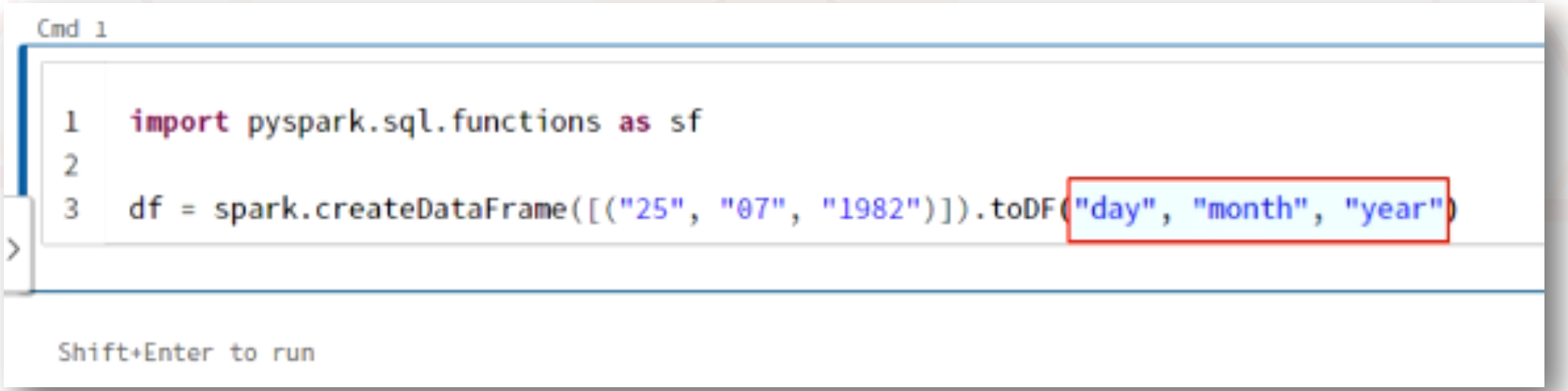
The first item is concat() and concat_ws() function. These guys allow you to concatenate multiple string columns together. The concat_ws() stands for concatenation with a separator.

S.No.	Function	Description
1.	concat(),concat_ws()	Concatenate multiple columns
2.	format_number(),	Format numbers
	format_string()	Format strings
3.	initcap(), lower(), upper()	Converting letters
4.	instr(),locate()	Locate the position of the of substr
5.	substring()	extracting substring
6.	length()	Character length including spaces
7.	lpad(), rpad()	For padding to a length
8.	ltrim(), rtrim(), trim()	Triming
9.	split(), sentences()	Splitting
10.	translate()	Character translation
11.	regexp_extract(), regexp_replace	Java regular expression

Go to your Databricks workspace and create a new notebook. **(Reference : 23-string-manipulation)**



I am creating a new Dataframe. Assume you have three columns: day, month and year. And you want to combine them to create a date column. You can easily use the concat() function for doing that.



Here is the code to concatenate year, month, and day. And you can see the output below.

Cmd 1

```
1  import pyspark.sql.functions as sf
2
3  df = spark.createDataFrame([("25", "07", "1982")]).toDF("day", "month", "year")
4  df.select(sf.concat_ws('/', df.year, df.month, df.day).alias("date")).show()
```

▸ (3) Spark Jobs

```
+----------+
|      date|
+----------+
|1982/07/25|
+----------+
```

Command took 5.02 seconds -- by prashant@scholarnest.com at 5/29/2022, 12:43:17 AM on demo-cluster

All the string manipulation functions are also available to you as built-in functions. So you can also use them to create SQL-like expressions as shown below.

Cmd 2

```
1  df.selectExpr("concat_ws('-', year, month, day) as date").show()
```

▸ (3) Spark Jobs

```
+----------+
|      date|
+----------+
|1982-07-25|
+----------+
```

Command took 0.92 seconds -- by prashant@scholarnest.com at 5/29/2022, 12:44:36 AM on demo-cluster

The next one in the list is format_number() and format_string().
You can use these guys to change the display format of numbers and strings.
Let us see some examples.

S.No.	Function	Description
1.	concat(),concat_ws()	Concatenate multiple columns
2.	format_number(),	Format numbers
	format_string()	Format strings
3.	initcap(), lower(), upper()	Converting letters
4.	instr(),locate()	Locate the position of the of substr
5.	substring()	extracting substring
6.	length()	Character length including spaces
7.	lpad(), rpad()	For padding to a length
8.	ltrim(), rtrim(), trim()	Triming
9.	split(), sentences()	Splitting
10.	translate()	Character translation
11.	regexp_extract(), regexp_replace	Java regular expression

Here is an example use of the format_number(). You can use it to apply separator and currency symbols.

Here is an example shown below for using format_string().
The format_string() is similar to printf() in C programming. You can use the format_string() to create a string combining multiple columns.

Cmd 4

```
1  df = spark.createDataFrame([("Sanjay", "Kalra", 25, "July", 1982)]) \
2          .toDF("first_name", "last_name", "day", "month", "year")
3
4  df.select(sf.format_string("Mr. %s %s was born on %dth %s of %d.",
5                             df.first_name, df.last_name,
6                             df.day, df.month, df.year).alias("fun_str")) \
7      .show(truncate=False)
```

▸ (3) Spark Jobs

```
+-----------------------------------------------+
|fun_str                                        |
+-----------------------------------------------+
|Mr. Sanjay Kalra was born on 25th July of 1982.|
+-----------------------------------------------+
```

Command took 1.07 seconds -- by prashant@scholarnest.com at 5/29/2022, 12:56:40 AM on demo-cluster

You have initcap(), lower(), and upper() functions to change the case.

S.No.	Function	Description
1.	concat(),concat_ws()	Concatenate multiple columns
2.	format_number(),	Format numbers
	format_string()	Format strings
3.	initcap(), lower(), upper()	Converting letters
4.	instr(),locate()	Locate the position of the of substr
5.	substring()	extracting substring
6.	length()	Character length including spaces
7.	lpad(), rpad()	For padding to a length
8.	ltrim(), rtrim(), trim()	Triming
9.	split(), sentences()	Splitting
10.	translate()	Character translation
11.	regexp_extract(), regexp_replace	Java regular expression

The instr() and locate() are the same. They allow you to search a substring inside a larger string.

S.No.	Function	Description
1.	concat(),concat_ws()	Concatenate multiple columns
2.	format_number(), format_string()	Format numbers Format strings
3.	initcap(), lower(), upper()	Converting letters
4.	instr(),locate()	Locate the position of the of substr
5.	substring()	extracting substring
6.	length()	Character length including spaces
7.	lpad(), rpad()	For padding to a length
8.	ltrim(), rtrim(), trim()	Triming
9.	split(), sentences()	Splitting
10.	translate()	Character translation
11.	regexp_extract(), regexp_replace	Java regular expression

Item 5,6,7 and 8 are also popular SQL functions. I hope you know when and how to use them.

S.No.	Function	Description
1.	concat(),concat_ws()	Concatenate multiple columns
2.	format_number(),	Format numbers
	format_string()	Format strings
3.	initcap(), lower(), upper()	Converting letters
4.	instr(),locate()	Locate the position of the of substr
5.	substring()	extracting substring
6.	length()	Character length including spaces
7.	lpad(), rpad()	For padding to a length
8.	ltrim(), rtrim(), trim()	Triming
9.	split(), sentences()	Splitting
10.	translate()	Character translation
11.	regexp_extract(), regexp_replace	Java regular expression

You have two functions to split a string. The split() function splits a string into words, and the sentences() is to split a string into sentences.
The result of the split comes in an array.

S.No.	Function	Description
1.	concat(),concat_ws()	Concatenate multiple columns
2.	format_number(), format_string()	Format numbers Format strings
3.	initcap(), lower(), upper()	Converting letters
4.	instr(),locate()	Locate the position of the of substr
5.	substring()	extracting substring
6.	length()	Character length including spaces
7.	lpad(), rpad()	For padding to a length
8.	ltrim(), rtrim(), trim()	Triming
9.	split(), sentences()	Splitting
10.	translate()	Character translation
11.	regexp_extract(), regexp_replace	Java regular expression

The next one is the translate() function. This guy allows you to translate one character into another character.

S.No.	Function	Description
1.	concat(),concat_ws()	Concatenate multiple columns
2.	format_number(), format_string()	Format numbers Format strings
3.	initcap(), lower(), upper()	Converting letters
4.	instr(),locate()	Locate the position of the of substr
5.	substring()	extracting substring
6.	length()	Character length including spaces
7.	lpad(), rpad()	For padding to a length
8.	ltrim(), rtrim(), trim()	Triming
9.	split(), sentences()	Splitting
10. →	translate()	Character translation
11.	regexp_extract(), regexp_replace	Java regular expression

Here is a new Dataframe.
You have some typo or some junk characters in the address field. You can use the translate() function to translate these characters to something else.

Cmd 5

```
1  df = spark.createDataFrame([("Benga`li Market", "110001"), ("Adu~godi", "560030")]) \
2          .toDF("address", "pin")
3
4  df.show()
```

▸ (3) Spark Jobs

```
+---------------+------+
|        address|   pin|
+---------------+------+
|Benga`li Market|110001|
|       Adu~godi|560030|
+---------------+------+
```

Command took 0.88 seconds -- by prashant@scholarnest.com at 5/29/2022, 1:07:44 AM on demo-cluster

I am translating both the characters to nothing. You can translate them to something else. But in my case, I simply want to remove them, so I am translating them to nothing. And you can see that the junk characters are removed and we got a clean output.

Cmd 6

```
1  df.withColumn("address", sf.translate(df.address, "~`", "")).show()
```

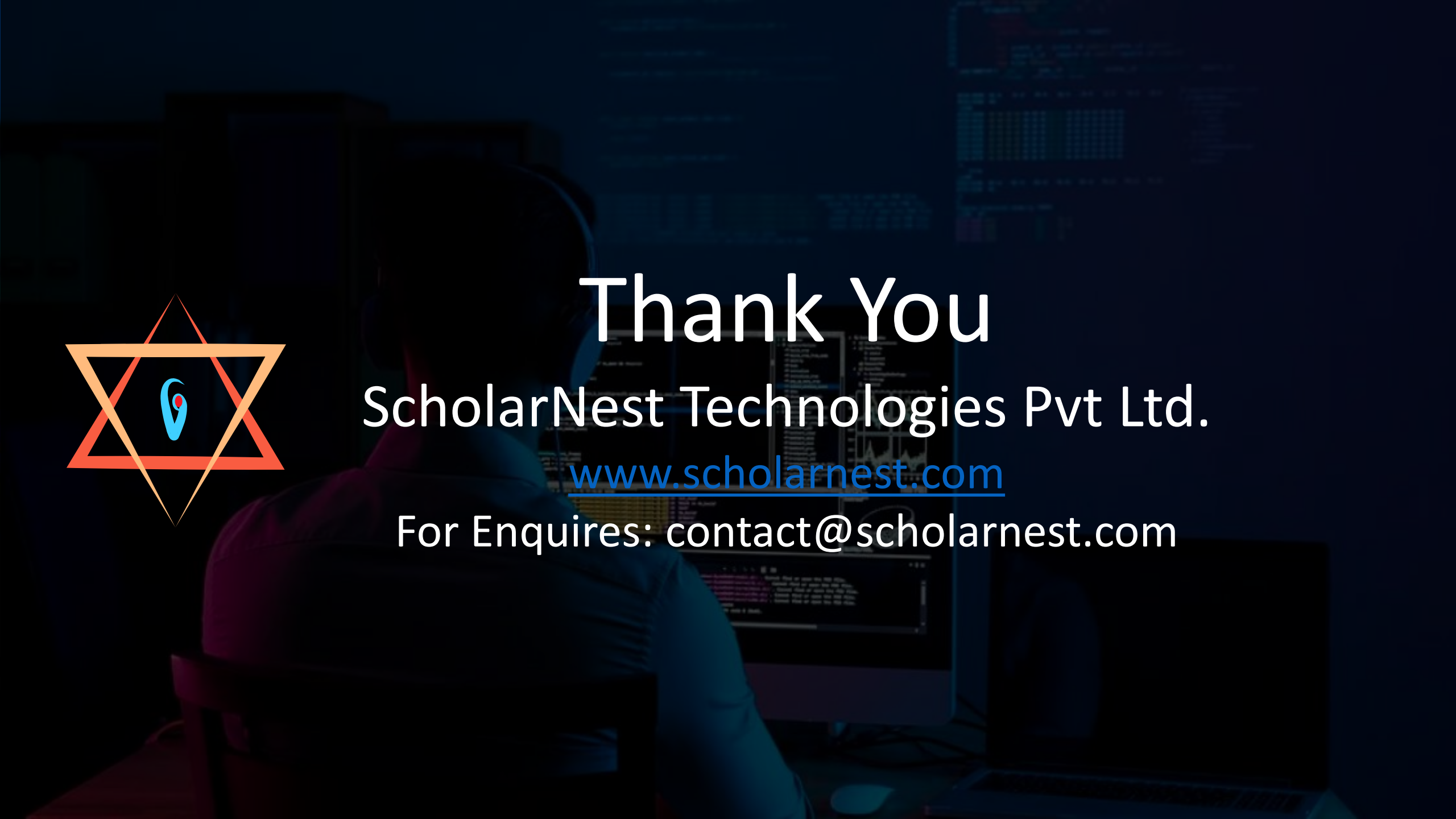
▸ (3) Spark Jobs

```
+--------------+------+
|       address|   pin|
+--------------+------+
|Bengali Market|110001|
|       Adugodi|560030|
+--------------+------+
```

Command took 0.84 seconds -- by prashant@scholarnest.com at 5/29/2022, 1:09:09 AM on demo-cluster

The last item is about two regular expression functions. Regular expressions are handy for manipulating text and string data. If you are working with unstructured text or log files, these two guys could help you find structure within the text data. I already showed an example in the earlier lectures for using them. However, creating regular expressions is a different skill. Spark supports Java regular expressions. So make sure you are using Java-compliant regular expressions. If you are new to regular expressions and want to learn, I recommend the following documentation link. (https://docs.oracle.com/javase/tutorial/essential/regex/index.html)

S.No.	Function	Description
1.	concat(),concat_ws()	Concatenate multiple columns
2.	format_number(), format_string()	Format numbers Format strings
3.	initcap(), lower(), upper()	Converting letters
4.	instr(),locate()	Locate the position of the of substr
5.	substring()	extracting substring
6.	length()	Character length including spaces
7.	lpad(), rpad()	For padding to a length
8.	ltrim(), rtrim(), trim()	Triming
9.	split(), sentences()	Splitting
10.	translate()	Character translation
11.	regexp_extract(), regexp_replace	Java regular expression



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Working with Different Data Types

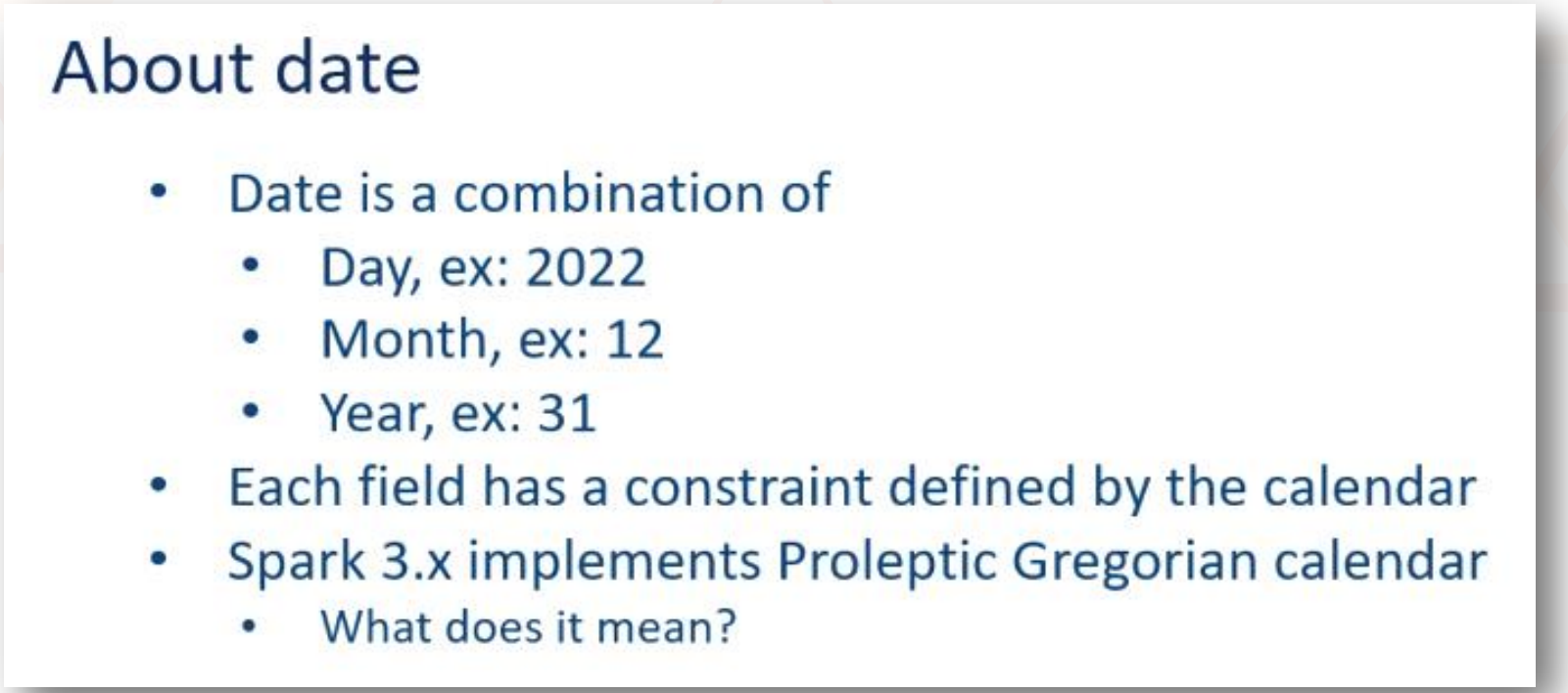
Lecture:
Dates and Timestamps

Dates and Timestamps

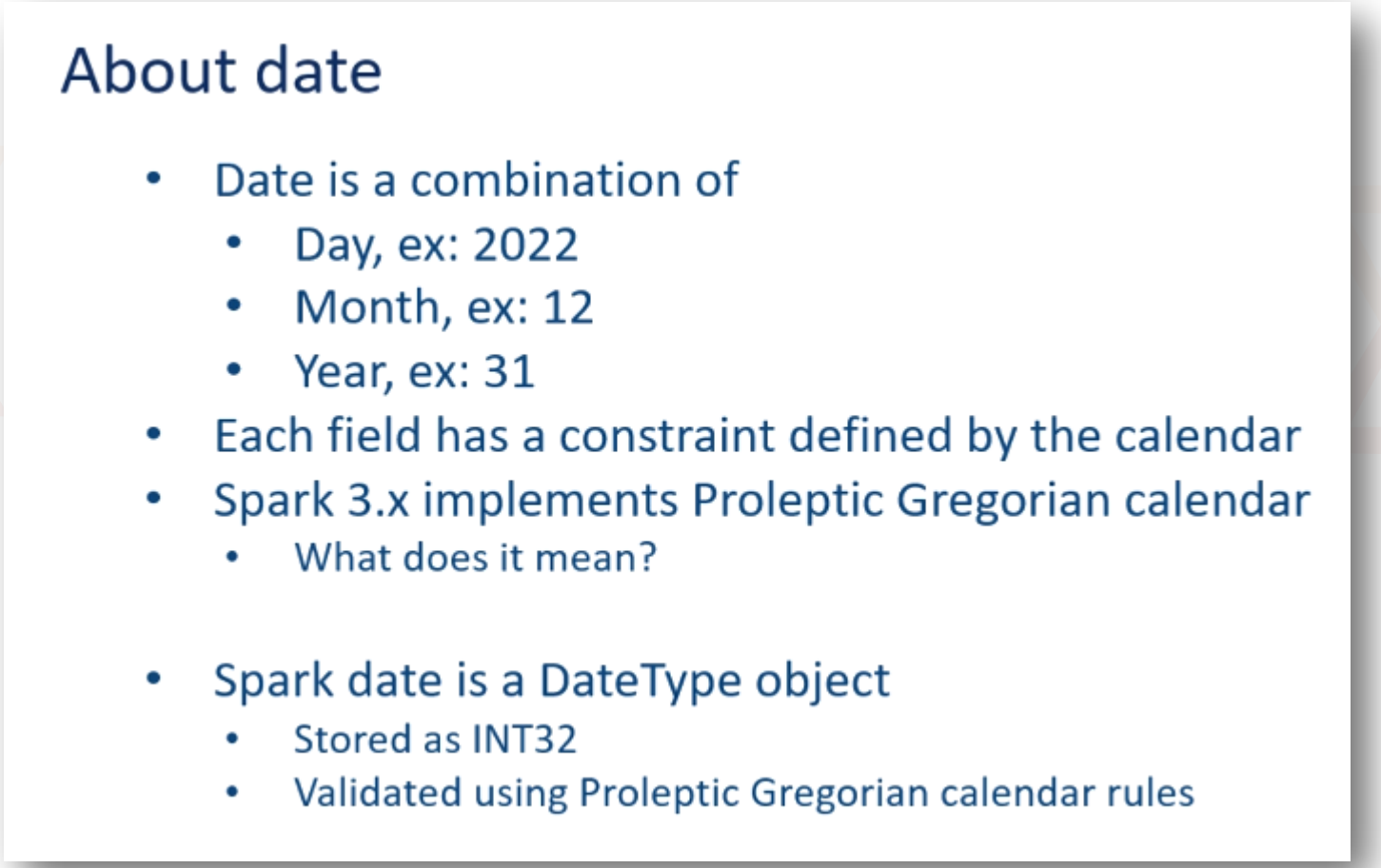
In this chapter, we will learn the following things:

1. How does Spark validate and store a date?
2. How does Spark handle invalid dates?
3. How does Spark stores and validate a timestamp?
4. What is Spark timestamp precision?
5. How does Spark store and handles time zone?

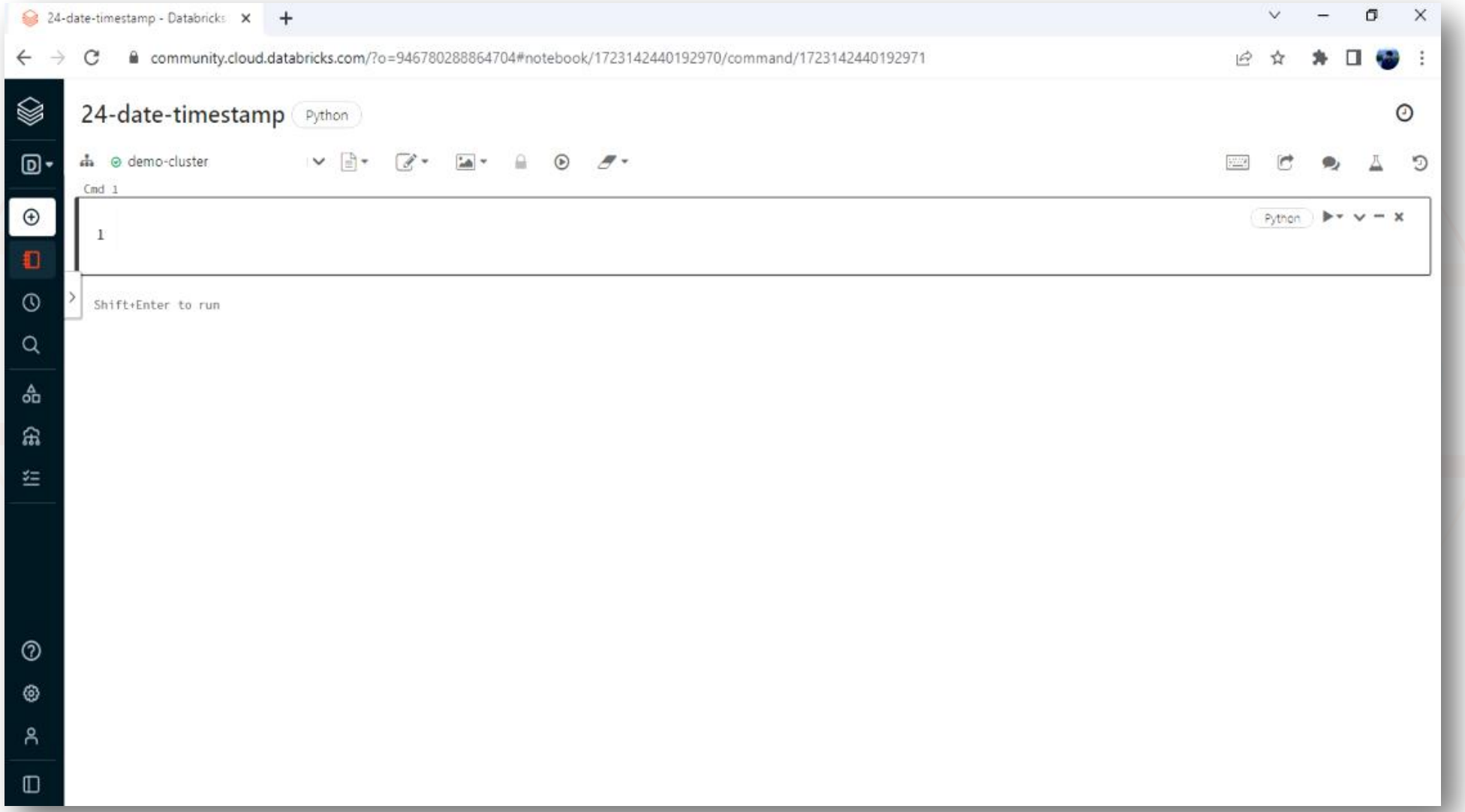
The definition of date is super simple. It is a combination of day, month, and year.
However, the values of days, months, and years have constraints. And these constraints are rules of a real-world calendar.
For example, the month must be between 1 and 12.
Depending on the year and month, the day must be between 1 and 28/29/30/31.
And for spark, these constraints are validated by the Proleptic Gregorian calendar.



Spark stores date in DateType(). The DateType internally stores dates in a 4-byte integer, known as INT32. The dataType() doesn't have a time component. It only represents a day, month and year validated against the Proleptic Gregorian calendar. So any date which is not a valid date according to the Proleptic Gregorian calendar is an invalid date for Apache Spark.



Go to your Databricks workspace and create a new notebook. **(Reference : 24-date-timestamp)**



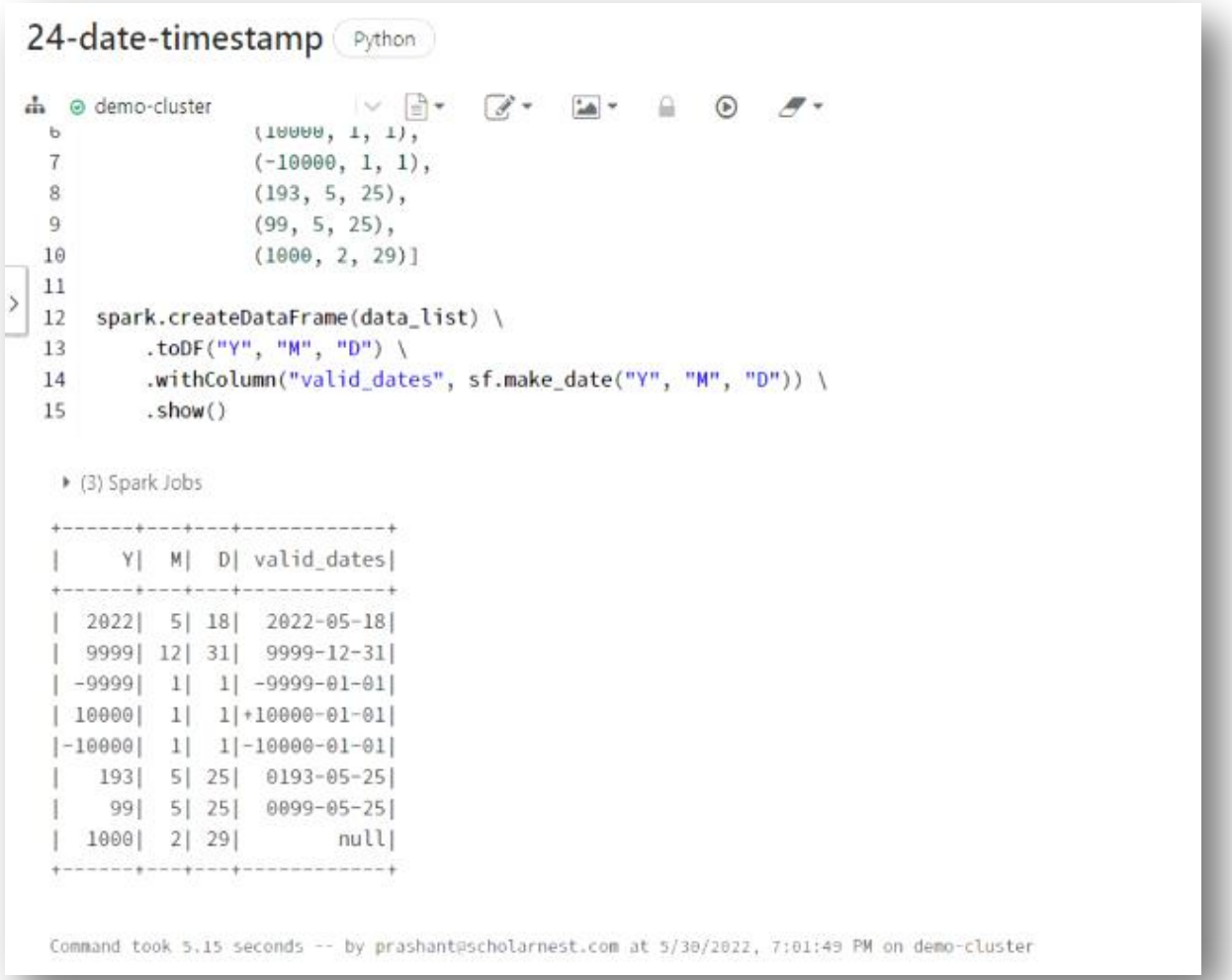
Here is a data list of some random year, month, and day fields.
Then I am creating a Dataframe and using the make_date() function to try making a valid date from Year, Month, and Day fields. Most of the years are weird. The first one is the year 2022, which is fine. But the other years are in positive, negative, and less than four-digit values.

Cmd 1

```
1   import pyspark.sql.functions as sf
2
3   data_list = [(2022, 5, 18),
4                (9999, 12, 31),
5                (-9999, 1, 1),
6                (10000, 1, 1),
7                (-10000, 1, 1),
8                (193, 5, 25),
9                (99, 5, 25),
10               (1000, 2, 29)]
11
12  spark.createDataFrame(data_list) \
13      .toDF("Y", "M", "D") \
14      .withColumn("valid_dates", sf.make_date("Y", "M", "D")) \
15      .show()
```

Shift+Enter to run

If you run the cell, you can see that all the values turned out to be a valid dates except one.
The negative years are BC, and the positive values are AD. But both are valid for the Proleptic Gregorian calendar. Two-digit and three-digit years are prefixed with zeros.
The last one is an invalid date because the year 1000 is not a leap year using the Proleptic Gregorian calendar.

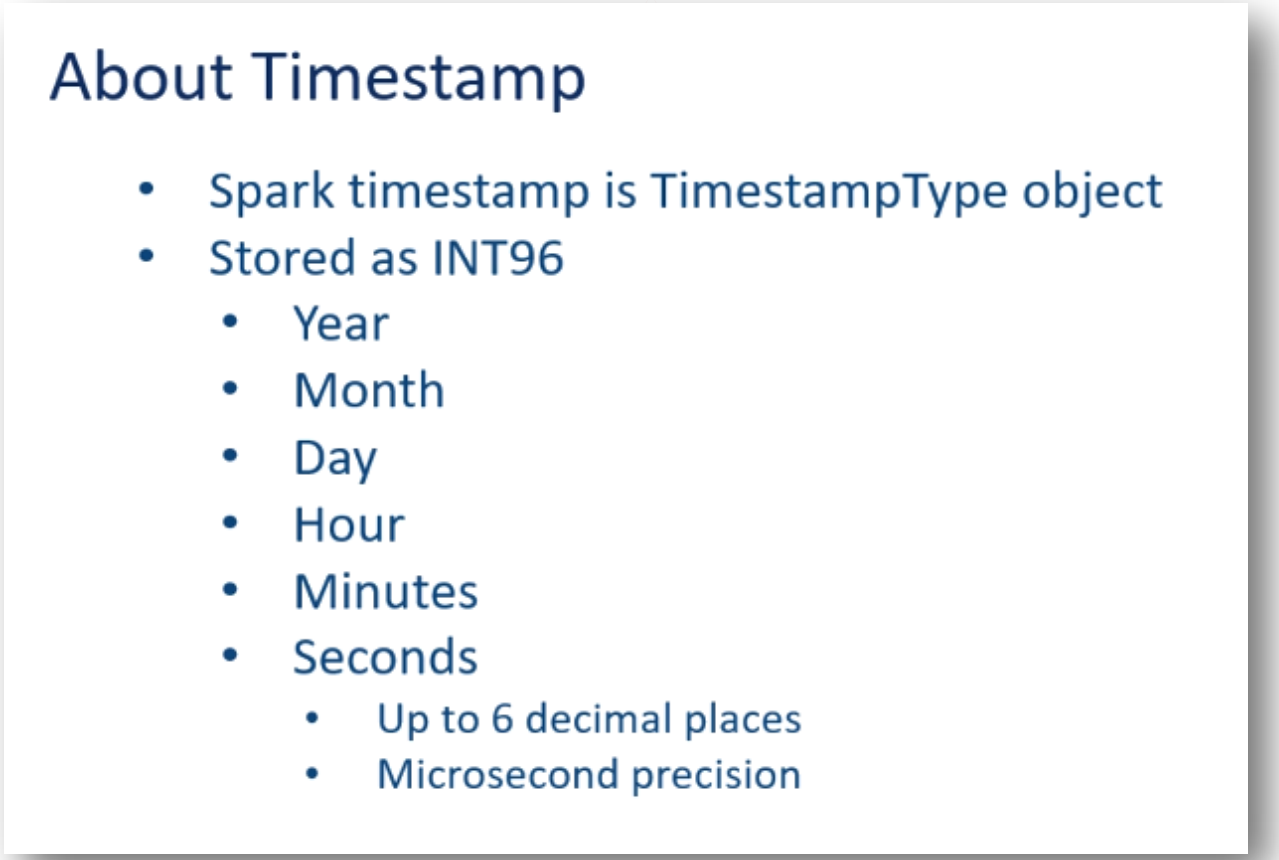


The formula for calculating leap year depends on the calendar system. So other calendars might show the year 1000 as a leap year, but the Proleptic Gregorian calendar does not show the year 1000 as a leap year.
Spark will try to make dates using the values. However, it will become null if it is an invalid date or Spark couldn't interpret it correctly.

So remember two basics about the dates in Spark:

1. Spark tries to interpret the date using the given format and validates it against the Proleptic Gregorian calendar.
2. Valid dates are taken, and invalid dates are turned null.

Now let's move on to the timestamp basics.
Spark stores timestamp in the TimestampType() field. The time is internally stored as a 12-byte integer known as INT96. Timestamp is made up of six fields given in the screenshot below. All the fields are integer, but the second is a decimal up to six decimal places. Spark will simply ignore it If you try using more than six decimal places in the seconds. It won't even round, and simply chop it off. So, spark timestamp can go up to microsecond precision. You cannot go beyond microseconds.



Before we proceed, let us look at the questions we discussed earlier:

1. How does Spark validate and store a date?
2. How does Spark handle invalid dates?
3. How does Spark stores and validate a timestamp?
4. What is Spark timestamp precision?
5. How does Spark store and handles time zone?

And I hope you learned the answer to the first four questions. Let me quickly answer these again, so you get the point.

1. Spark validates the data against the Proleptic Gregorian calendar. And it stores valid date using INT32 representation.
2. Invalid dates are converted to null.
3. Spark validates the timestamp against a time in the Proleptic Gregorian calendar. And it stores the valid timestamp using INT96 representation.
4. Spark 3.x supports microsecond precision.

I have a data file. **(Reference: data/machine-events-no-tz.csv)**
You can see the contents of the file in the screenshot below. I have an events data file. We have only three fields. Component name, event time, and the reading. You can think of it as events sent by an IoT device. They send events every 10-15 seconds. And we have the timestamp for each event. The first event is sent at 06:14:10. The next one is sent at 06:14:25. But we do not know the timezone.
Does this time represents an IST or EST, or GMT, or is it UTC? I want to load this data into Spark, but Spark doesn't have any way to know the timezone of this time field. And that may be a problem in some cases.

The missing timezone could be a problem, or it might not be a problem. It depends on your requirement and what you want to do with your data.

Let me give you two example requirements:

1. Draw a 24-hour line chart to present how they change or fluctuate over time.
2. Monitor these events in real-time and identify malfunction of the device. Also, generate malfunction notification for the filed officer.

Now think about these two requirements.

Timezone is not critical for the first requirement. You can draw a 24-hour timeline even if you do not know the timezone. However, the timezone is critical for the second requirement. You are asked to notify the field officer and inform him that the device started malfunctioning at 10.15 AM. The field officer must get this message in the local time zone, and when you say 10.15, he will assume 10.15 in his local time zone. It doesn't make sense to inform the field officer that the device started to malfunction at 10:15 today. But we do not know whether it is 10:15 US time or Asia time, or he should look at the London clock tower.

So the time zone is often critical. The best practice is to add timezone information to your data even if you do not need the timezone. Missing timezone or incorrect timezone is ambiguous data. Your project must do everything possible to fix this problem.

Here is another data file. **(Reference: data/machine-events-with-tz.csv)**
This guy is precise data. We have timezone information with the timestamp.
Here the AXT594 is sending event time in the UTC timezone. The device could be located in India or Japan. We don't know that. But the time is in the UTC timezone, and we know that. Similarly, AXT595 sends data event time in the IST format. And the AXT596 is sending data using the London time.

Cmd 3

```
1  %fs head /FileStore/tables/tmp/machine_events_with_tz.csv
```

```
component,event_time,reading
AXT594,17-05-2022 06:14:10.359+0000,23
AXT595,17-05-2022 06:14:25.380+0530,22
AXT596,17-05-2022 06:14:35.346+0100,24
AXT594,17-05-2022 06:14:45.381+0000,21
AXT595,17-05-2022 06:14:55.356+0530,23
AXT596,17-05-2022 06:15:05.372+0100,22
AXT594,17-05-2022 06:15:15.355+0000,21
AXT595,17-05-2022 06:16:25.326+0530,25
AXT596,17-05-2022 06:17:35.345+0100,22
AXT594,17-05-2022 06:18:45.365+0000,21
```

Command took 0.68 seconds -- by prashant@scholarnest.com at 5/30/2022, 7:21:45 PM on demo-cluster

Here is the previous file, it is all about AXT594. I discussed it with the team that owns and manages this data. They are providing this dataset to me for ingesting into the data lake. So the team knows the timezone.
We do not have any ambiguous data, and the files we provide are complete and valid. In some cases, you will see timezone information in the data file. When you do not see it, you can assume the UTC timezone.

Cmd 2

```
1  %fs head /FileStore/tables/tmp/machine_events_no_tz.csv
```

```
component,event_time,reading
AXT594,17-05-2022 06:14:10.359,23
AXT594,17-05-2022 06:14:25.380,25
AXT594,17-05-2022 06:14:35.346,21
AXT594,17-05-2022 06:14:45.381,22
AXT594,17-05-2022 06:14:55.356,25
AXT594,17-05-2022 06:15:05.372,23
AXT594,17-05-2022 06:15:15.355,24
AXT594,17-05-2022 06:16:25.326,24
AXT594,17-05-2022 06:17:35.345,21
AXT594,17-05-2022 06:18:45.365,22
```

Command took 12.78 seconds -- by prashant@scholarnest.com at 5/30/2022, 7:16:58 PM on demo-cluster

Data capture might happen in different ways.
They might capture the timezone with each record or normalize the time to a default timezone. I have data files for both scenarios.

The first file is normalized to the UTC timezone, and all the timestamps are UTC. The second file is not standardized, and time zones are captured with each record.

So we learned two common timezone scenarios.
Now I have two more questions:

1. How do we correctly load this data in Spark?
2. How does Spark handles the timezones?

Spark comes with a default timezone setting. And you can use the code shown below to get the default timezone for your session.
The default timezone for my session is UTC.

You can change the timezone as per your requirement.
In the code shown below, I am changing it to IST, and rechecking it.

```
Cmd 5

1  spark.conf.set("spark.sql.session.timeZone", "IST")
2  spark.conf.get("spark.sql.session.timeZone")

Out[15]: 'IST'

Command took 0.04 seconds -- by prashant@scholarnest.com at 5/30/2022, 7:57:28 PM on demo-cluster
```

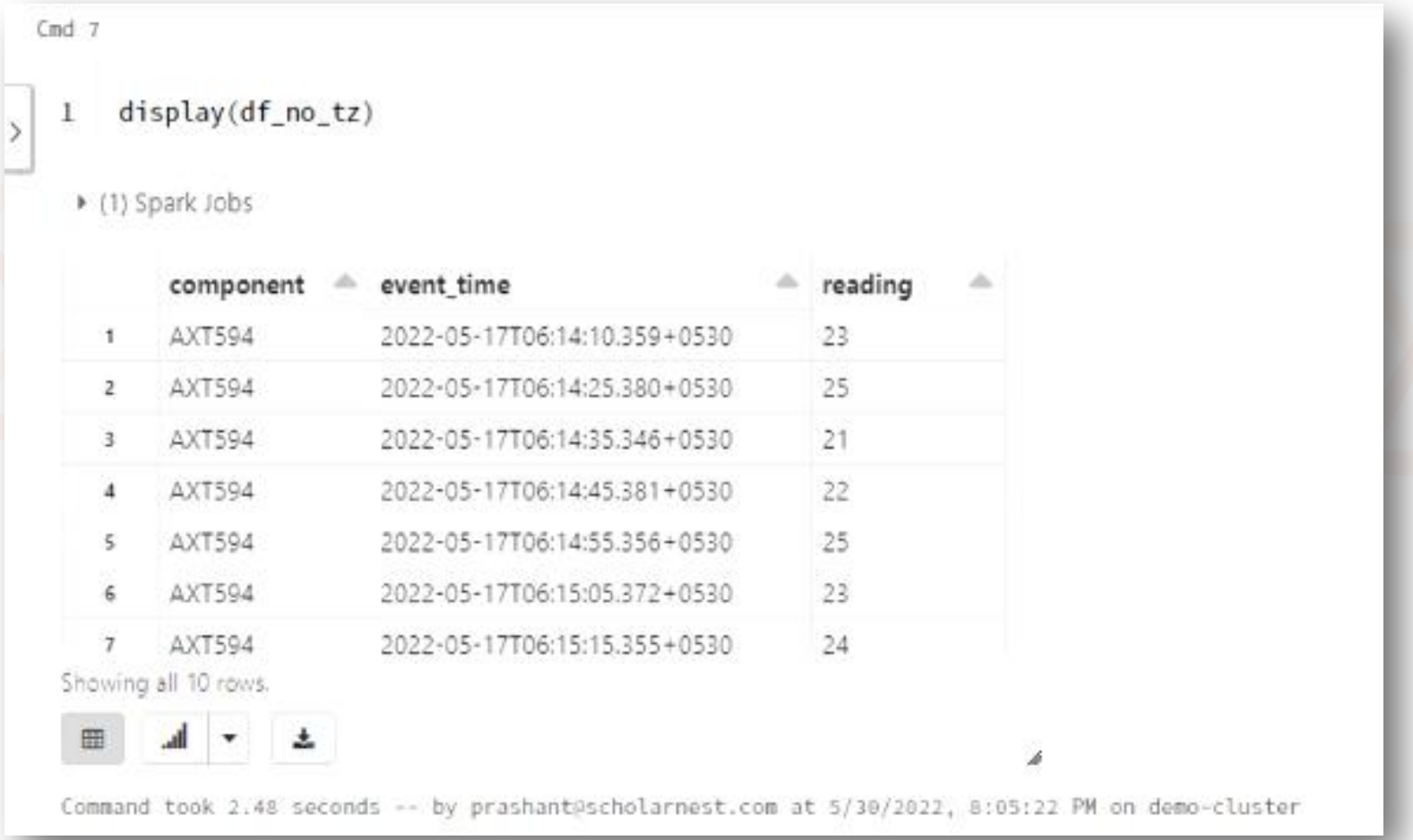

Here in this code, I am loading the first data file.
I am defining the schema for the file. I do not want to load the event_time as a timestamp. So I defined it as a string. I will load it as a string and fix it later. Then I load the data. I am setting the header to skip the first row. Then I am setting the schema and load the file. The last line is to fix the event_time data type. So I am using the to_timestamp() built-in function to convert the string into a timestamp.
You might be wondering about the format string here. You can refer to the Spark SQL documentation (https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html) to understand how to create a format string. This page comes with a list of codes. You can use them within Spark SQL functions to format the date and timestamp fields.

Cmd 6

```
1  schema = "component string, event_time string, reading int"
2
3  df_no_tz = spark.read \
4                  .format("csv") \
5                  .option("header", "true") \
6                  .schema(schema) \
7                  .load("/FileStore/tables/tmp/machine_events_no_tz.csv") \
8                  .withColumn("event_time", sf.expr("to_timestamp(event_time, 'dd-MM-yyyy HH:mm:ss.SSS')"))
```

Command took 0.75 seconds -- by prashant@scholarnest.com at 5/30/2022, 8:04:11 PM on demo-cluster

Here is the Dataframe for the first data file.
You can see the time for the first record is still 6:14:10, but Spark added the IST timeZone to each record. Where is this timezone coming from? It is coming from the spark.sql.session.timeZone, remember we set it to IST earlier.


Cmd 7

```
1  display(df_no_tz)
```

(1) Spark Jobs

	component	event_time	reading
1	AXT594	2022-05-17T06:14:10.359+0530	23
2	AXT594	2022-05-17T06:14:25.380+0530	25
3	AXT594	2022-05-17T06:14:35.346+0530	21
4	AXT594	2022-05-17T06:14:45.381+0530	22
5	AXT594	2022-05-17T06:14:55.356+0530	25
6	AXT594	2022-05-17T06:15:05.372+0530	23
7	AXT594	2022-05-17T06:15:15.355+0530	24

Showing all 10 rows.

Command took 2.48 seconds -- by prashant@scholarnest.com at 5/30/2022, 8:05:22 PM on demo-cluster

So, If you do not specify the timezone, Spark assumes that the time is set in the *spark.sql.session.timeZone*. That's your default timezone. So if the timezone is missing, Spark will automatically add the default session timezone.
Spark is very strict about the timezone. Every timestamp in Spark must have a timezone. You cannot leave it. If you leave it, Spark will add the default session timezone to your data.
And that's perfectly fine because we have only two scenarios:

1. Your data is standardized to a timezone.
2. Your data comes with a timezone.

You can handle the first scenario by setting your Spark session timezone. Spark will automatically add the timezone to your data. I assumed that my file came with the IST timezone. So I changed my session timezone to IST and loaded the file. Spark will automatically add the IST timezone to my data. And that is what I wanted.
The second scenario doesn't have any problem. Your data comes with a timezone. So Spark will take it.

Here is the Dataframe for the second data file.
The code is the same as earlier. I just changed the file name here. And I also changed the format to add the zone specifier.

Cmd 8

```
1  schema = "component string, event_time string, reading int"
2
3  df_with_tz = spark.read \
4                  .format("csv") \
5                  .option("header", "true") \
6                  .schema(schema) \
7                  .load("/FileStore/tables/tmp/machine_events_with_tz.csv") \
8                  .withColumn("event_time", sf.expr("to_timestamp(event_time, 'dd-MM-yyyy HH:mm:ss.SSSZ')"))
```


Let's look at the output. My current session timezone is IST, so the display method shows everything in IST. And that makes sense. My default timezone is IST, so I see everything in IST. But the time is correct.

Cmd 8

```
1   schema = "component string, event_time string, reading int"
2
3   df_with_tz = spark.read \
4                   .format("csv") \
5                   .option("header", "true") \
6                   .schema(schema) \
7                   .load("/FileStore/tables/tmp/machine_events_with_tz.csv") \
8                   .withColumn("event_time", sf.expr("to_timestamp(event_time, 'dd-MM-yyyy HH:mm:ss.SSSZ')"))
9
10  display(df_with_tz)
```

▸ (1) Spark Jobs

	component	event_time	reading
1	AXT594	2022-05-17T11:44:10.359+0530	23
2	AXT595	2022-05-17T06:14:25.380+0530	22
3	AXT596	2022-05-17T10:44:35.346+0530	24
4	AXT594	2022-05-17T11:44:45.381+0530	21
5	AXT595	2022-05-17T06:14:55.356+0530	23
6	AXT596	2022-05-17T10:45:05.372+0530	22
7	AXT594	2022-05-17T11:45:15.355+0530	21

Showing all 10 rows.

Look at the record times.
AXT594 record was loaded in UTC 6:14. Spark converted it to IST for showing it to me. So the time is 11:44, which is five and a half hours ahead of UTC. UTC 6:14 is 11:44 in IST. And that's what Spark is showing

```
1  schema = "component string, event_time string, reading int"
2
3  df_with_tz = spark.read \
4                  .format("csv") \
5                  .option("header", "true") \
6                  .schema(schema) \
7                  .load("/FileStore/tables/tmp/machine_events_with_tz.csv") \
8                  .withColumn("event_time", sf.expr("to_timestamp(event_time, 'dd-MM-yyyy HH:mm:ss.SSSZ')"))
9
10 display(df_with_tz)
```

▸ (1) Spark Jobs

	component	event_time	reading
1	AXT594	2022-05-17T11:44:10.359+0530	23
2	AXT595	2022-05-17T06:14:25.380+0530	22
3	AXT596	2022-05-17T10:44:35.346+0530	24
4	AXT594	2022-05-17T11:44:45.381+0530	21
5	AXT595	2022-05-17T06:14:55.356+0530	23
6	AXT596	2022-05-17T10:45:05.372+0530	22
7	AXT594	2022-05-17T11:45:15.355+0530	21

Showing all 10 rows.

Similarly, AXT595 was in IST. So it is still showing 6:14 IST. No conversion was required.

```
1  schema = "component string, event_time string, reading int"
2
3  df_with_tz = spark.read \
4                  .format("csv") \
5                  .option("header", "true") \
6                  .schema(schema) \
7                  .load("/FileStore/tables/tmp/machine_events_with_tz.csv") \
8                  .withColumn("event_time", sf.expr("to_timestamp(event_time, 'dd-MM-yyyy HH:mm:ss.SSSZ')"))
9
10 display(df_with_tz)
```

▸ (1) Spark Jobs

	component	event_time	reading
1	AXT594	2022-05-17T11:44:10.359+0530	23
2	AXT595	2022-05-17T06:14:25.380+0530	22
3	AXT596	2022-05-17T10:44:35.346+0530	24
4	AXT594	2022-05-17T11:44:45.381+0530	21
5	AXT595	2022-05-17T06:14:55.356+0530	23
6	AXT596	2022-05-17T10:45:05.372+0530	22
7	AXT594	2022-05-17T11:45:15.355+0530	21

Showing all 10 rows.

Command took 0.78 seconds -- by prashant@scholarnest.com at 5/30/2022, 8:10:30 PM on demo-cluster

Finally, the AXT596 was using London time. You can try converting 6:14 from the London clock to the India clock, and it will be 10:44, as shown here.

```
1  schema = "component string, event_time string, reading int"
2
3  df_with_tz = spark.read \
4                  .format("csv") \
5                  .option("header", "true") \
6                  .schema(schema) \
7                  .load("/FileStore/tables/tmp/machine_events_with_tz.csv") \
8                  .withColumn("event_time", sf.expr("to_timestamp(event_time, 'dd-MM-yyyy HH:mm:ss.SSSZ')"))
9
10 display(df_with_tz)
```

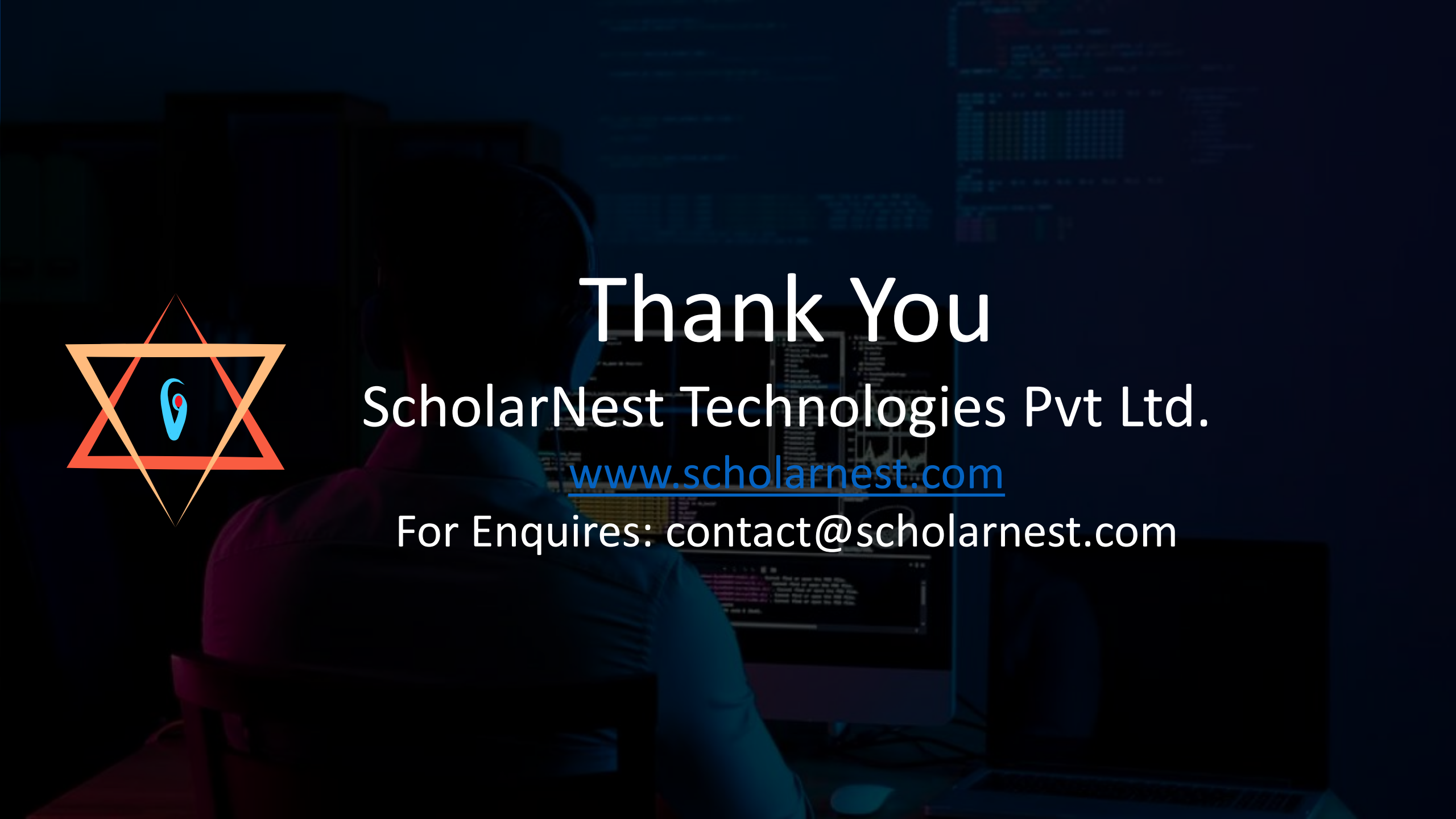
▸ (1) Spark Jobs

	component	event_time	reading
1	AXT594	2022-05-17T11:44:10.359+0530	23
2	AXT595	2022-05-17T06:14:25.380+0530	22
3	AXT596	2022-05-17T10:44:35.346+0530	24
4	AXT594	2022-05-17T11:44:45.381+0530	21
5	AXT595	2022-05-17T06:14:55.356+0530	23
6	AXT596	2022-05-17T10:45:05.372+0530	22
7	AXT594	2022-05-17T11:45:15.355+0530	21

Showing all 10 rows.

Command took 0.78 seconds -- by prashant@scholarnest.com at 5/30/2022, 8:10:30 PM on demo-cluster

So, If your data comes with a timezone, Spark will read it correctly. But it will show you the output in your local session timezone. Spark will automatically convert the timezone correctly.
If your data doesn't come with the timezone, Spark will assume that the data is given in the local session time zone and add the timezone automatically.
There are some date/time functions listed in the documentation.
(https://spark.apache.org/docs/latest/sql-ref-functions-builtin.html#date-and-timestamp-functions)
We use those functions for date arithmetic and other types of operations.
However, most of those were inherited from SQL.
So if you know SQL, you should be familiar with those functions.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Working with Different Data Types

Lecture:
Working with Struct

Complex Types – StructType

Apache Spark offers you three complex data types.

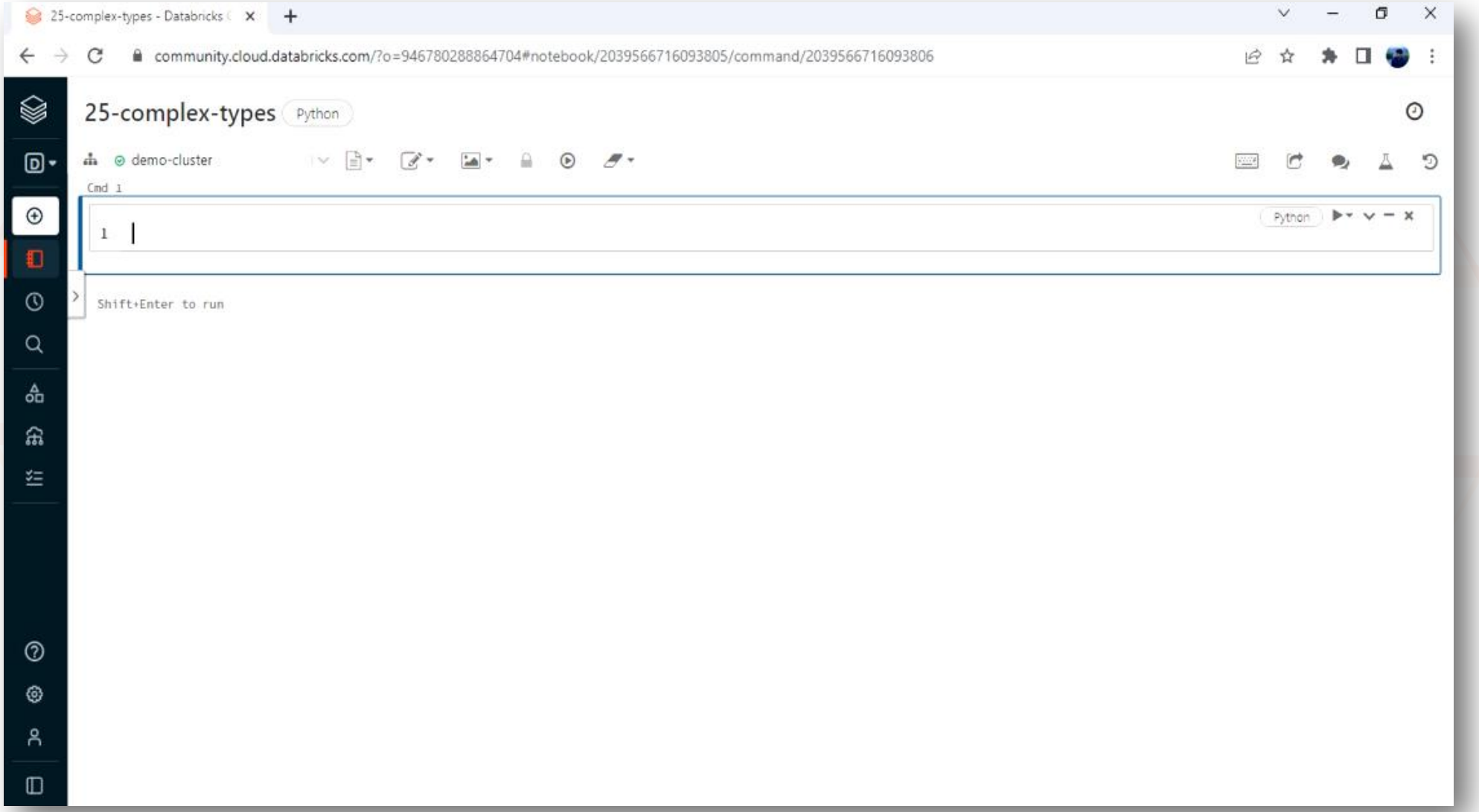
1. StructType()
2. ArrayType()
3. Maptype()

Let's start with the StructType() in this lecture.

The StructType() is the default type for a Dataframe schema.

It represents a record of columns.

Go to your Databricks workspace and create a new notebook. **(Reference : 25-complex-types)**



I have a JSON data file. **(Reference: data/struct-record.json)**
You can see the contents of the file in the output below. The JSON data file comes with only three fields: ID, FirstName, and LastName. Why am I using JSON for this example, and why not CSV? Because the CSV file doesn't support complex data types. CSV is more of structured data. And complex types are specifically designed to work with semi-structured data. The CSV file format doesn't support storing semi-structured data. So we are using a JSON file for this example. You can even use a Parquet file, but JSON is human-readable, so I decided to use JSON so you can see the data inside the file.

So we have the data file, and now I want to read it into a Dataframe as shown below.
As we learned earlier, we have two standard steps to read a data file.
Define a schema for the data file and use the DataFrameReader API to read the file.

Cmd 2

```
1  ddl_schema = "ID string, FirstName string, LastName string"
2
3  df = spark.read \
4          .format("json") \
5          .schema(ddl_schema) \
6          .load("/FileStore/tables/tmp/struct_record.json")
7
8  df.printSchema()
9  display(df)
```

Shift+Enter to run

You can see the schema as well as the Dataframe in the output below.
The printSchema() method shows the schema in a more readable format. However, I want to see the technical definition of the schema.

```
1  ddl_schema = "ID string, FirstName string, LastName string"
2
3  df = spark.read \
4          .format("json") \
5          .schema(ddl_schema) \
6          .load("/FileStore/tables/tmp/struct_record.json")
7
8  df.printSchema()
9  display(df)
```

▸ (1) Spark Jobs

```
root
 |-- ID: string (nullable = true)
 |-- FirstName: string (nullable = true)
 |-- LastName: string (nullable = true)
```

	ID	FirstName	LastName
1	101	Prashant	Pandey
2	102	David	Turner
3	103	Katie	Mcloskey
4	104	Nasima	Khatun
5	105	Pritam	Jain

Here is the dataframe schema to get the technical value of the schema.
The dataframe schema is an attribute that returns the technical definition of the schema.
We gave a DDL schema string. However, Spark converts it to a proper StructType() schema definition. And you can see the output. So the DataFrame record is a StructType(). We do not specify the StructType() in a schema DDL. However, it is a StructType().
So, DataFrame record is a StructType(). And this struct type is made up of three StructFileds: ID, FirstName, and LastName.

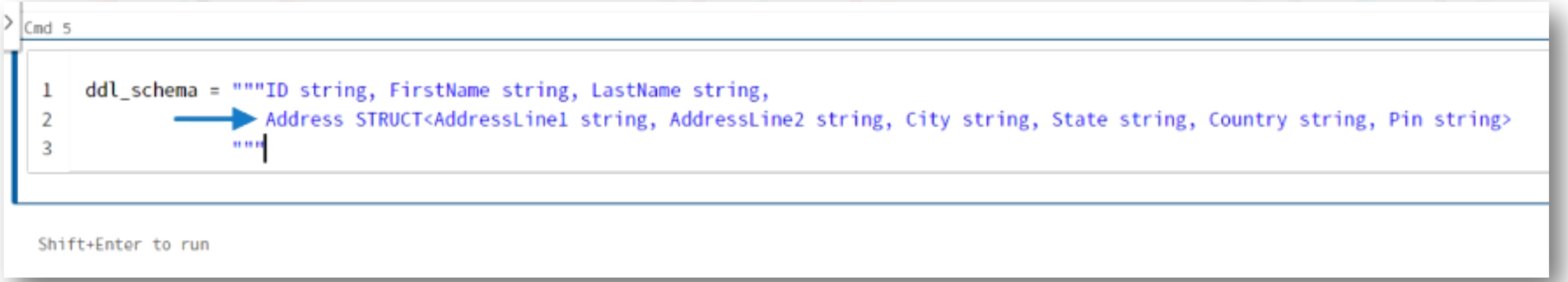
I have another JSON data file. **(Reference: data/struct-of-struct.json)**
So this file is an extension of the earlier file. We have four fields in this file: ID, FirstName, LastName, and Address. But the Address is a record in itself. We have a parent record and a child record. You can see it as a record containing another record. In more technical terms, the address field is also a StructType().

```
1   %fs head /FileStore/tables/tmp/struct_of_struct.json

{"ID":"101","FirstName":"Prashant","LastName":"Pandey","Address":{"AddressLine1":"D104 Gopalan Squire","AddressLine2":"Whitefield","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"}}
{"ID":"102","FirstName":"David","LastName":"Turner","Address":{"AddressLine1":"109 Park Street","AddressLine2":"Richmond","City":"London","State":"London","C
ountry":"Engaland","Pin":"EC1A"}}
{"ID":"103","FirstName":"Katie","LastName":"Mcloskey","Address":{"AddressLine1":"9th Avenue","AddressLine2":"Dorsy Road","City":"Belfast","State":"Belfas
t","Country":"Northern Ireland","Pin":"BT1 1BG"}}
{"ID":"104","FirstName":"Nasima","LastName":"Khatun","Address":{"AddressLine1":"G105 MG Tower","AddressLine2":"Bregade Road","City":"Kolkata","State":"West B
engal","Country":"India","Pin":"7000001"}}
{"ID":"105","FirstName":"Pritam","LastName":"Jain","Address":{"AddressLine1":"M206 Richmond Tower","AddressLine2":"Electronic City","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"}}


Command took 0.73 seconds -- by prashant@scholarnest.com at 6/2/2022, 7:48:06 PM on demo-cluster
Cmd 5
```


Here is the schema definition of the second data file.
The schema up to the LastName field is the same as earlier. Then we added the Address field.
But the type of the Address field is a struct. Then we define the fields of the Address struct.
It looks complex. But it is super simple. The Address field is a struct in itself.
And the Address struct is made up of six fields: AddressLine1, AddressLine2, City, State, Country, and Pin.



Now, I can use the schema definition to load the data as shown in the code given below.

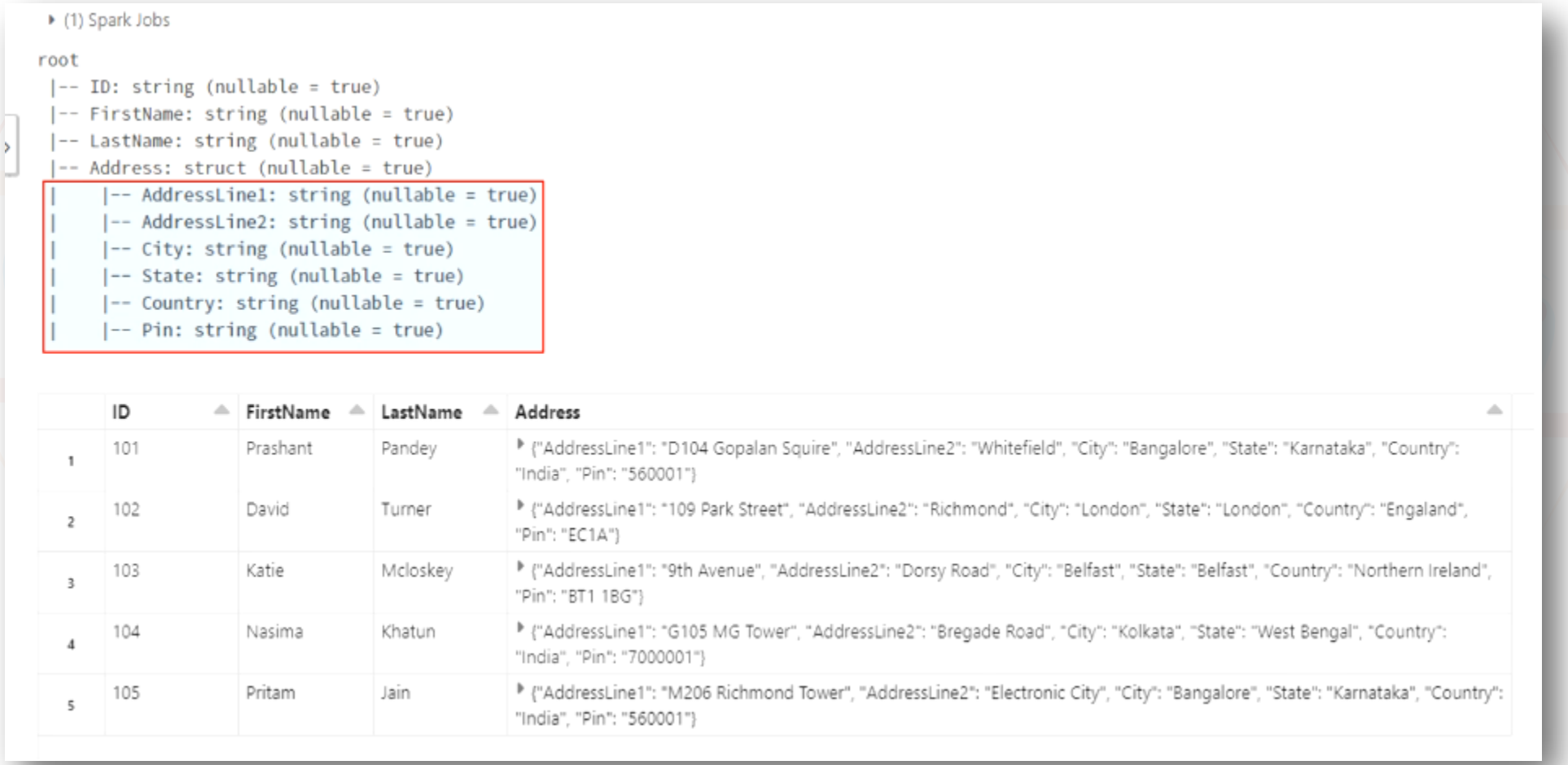
Cmd 5

```
ddl_schema = """ID string, FirstName string, LastName string,
                Address STRUCT<AddressLine1 string, AddressLine2 string, City string, State string, Country string, Pin string>
              """

df = spark.read \
        .format("json") \
        .schema(ddl_schema) \
        .load("/FileStore/tables/tmp/struct_of_struct.json")

df.printSchema()
display(df)
```


Here is the output of the cell.
You can see the printSchema() output. The ID, FirstName, and LastName are plain fields.
The Address is a complex struct field. And the structure of the Address is highlighted below.



▸ (1) Spark Jobs

```
root
 |-- ID: string (nullable = true)
 |-- FirstName: string (nullable = true)
 |-- LastName: string (nullable = true)
 |-- Address: struct (nullable = true)
 |    |-- AddressLine1: string (nullable = true)
 |    |-- AddressLine2: string (nullable = true)
 |    |-- City: string (nullable = true)
 |    |-- State: string (nullable = true)
 |    |-- Country: string (nullable = true)
 |    |-- Pin: string (nullable = true)
```

	ID	FirstName	LastName	Address
1	101	Prashant	Pandey	▸ {"AddressLine1": "D104 Gopalan Squire", "AddressLine2": "Whitefield", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}
2	102	David	Turner	▸ {"AddressLine1": "109 Park Street", "AddressLine2": "Richmond", "City": "London", "State": "London", "Country": "Engaland", "Pin": "EC1A"}
3	103	Katie	Mcloskey	▸ {"AddressLine1": "9th Avenue", "AddressLine2": "Dorsy Road", "City": "Belfast", "State": "Belfast", "Country": "Northern Ireland", "Pin": "BT1 1BG"}
4	104	Nasima	Khatun	▸ {"AddressLine1": "G105 MG Tower", "AddressLine2": "Bregade Road", "City": "Kolkata", "State": "West Bengal", "Country": "India", "Pin": "7000001"}
5	105	Pritam	Jain	▸ {"AddressLine1": "M206 Richmond Tower", "AddressLine2": "Electronic City", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}

So now you know how to read a complex struct type data from a data source.
We are reading it from a JSON file.
But you can read it from any source in the same way.
All you need to do is define an appropriate schema for reading the data correctly.

So now that we learned how to read StructType data. Let us see how to access it.

So you can access the StructType() using the dot notation in the parent field.
For example, you can read the city name using the address dot city, as shown in the code below.

You can also use the * to select all the fields as shown below.

Cmd 7

```
1  address_df = df.select("Address.*")
2  display(address_df)
```

▸ (1) Spark Jobs

	AddressLine1	AddressLine2	City	State	Country	Pin
1	D104 Gopalan Squire	Whitefield	Bangalore	Karnataka	India	560001
2	109 Park Street	Richmond	London	London	Engaland	EC1A
3	9th Avenue	Dorsy Road	Belfast	Belfast	Northern Ireland	BT1 1BG
4	G105 MG Tower	Bregade Road	Kolkata	West Bengal	India	7000001
5	M206 Richmond Tower	Electronic City	Bangalore	Karnataka	India	560001

Showing all 5 rows.

Command took 0.36 seconds -- by prashant@scholarnest.com at 6/4/2022, 12:01:41 PM on demo-cluster

Assignment:

Here is the Dataframe we created, this one is a kind of denormalized Dataframe.

I mean, you have a personal detail, and you also have a personal address embedded in the same Dataframe. Can you separate the details?

Cmd 8

```
1  display(df)
```

▸ (1) Spark Jobs

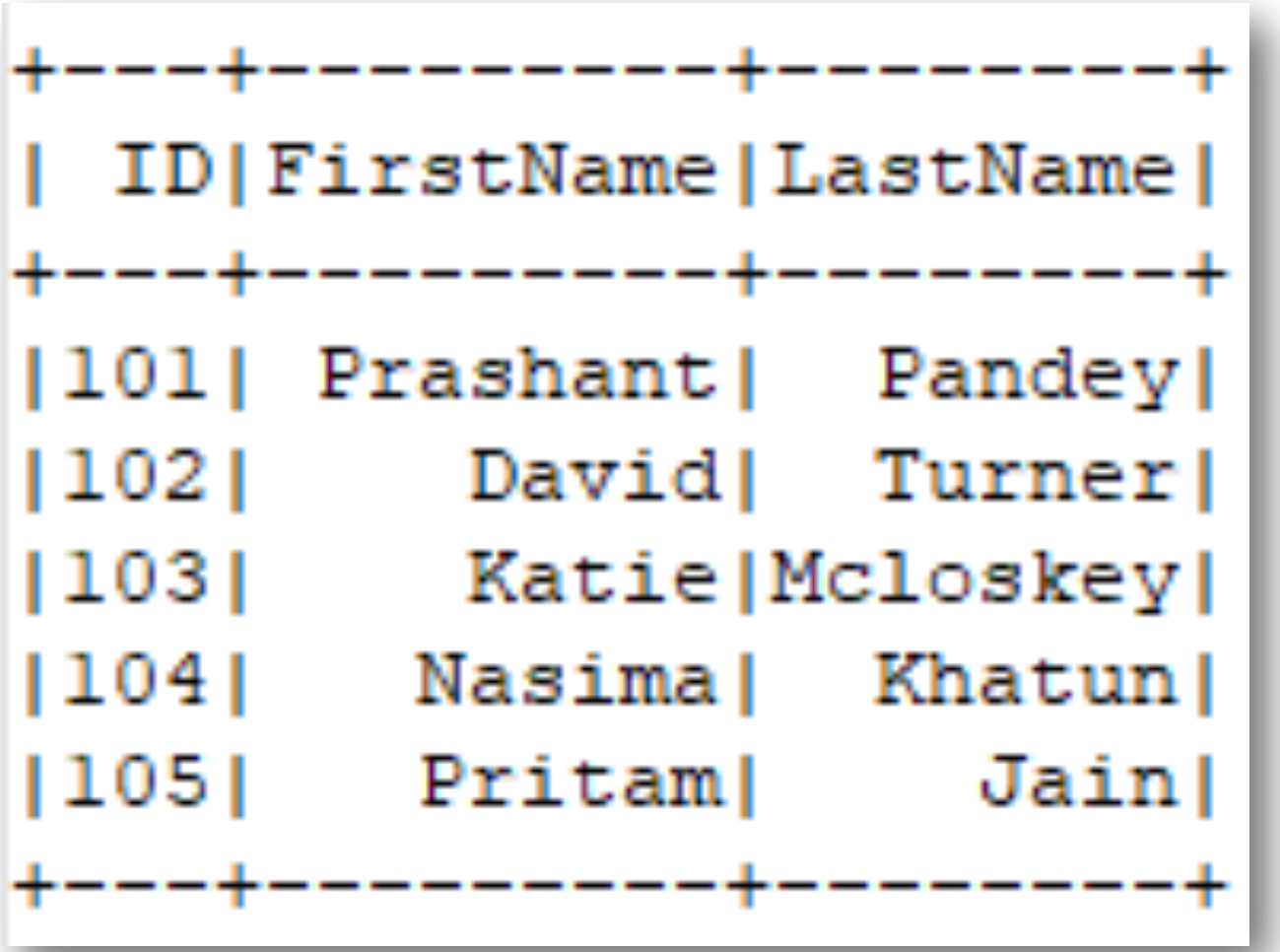
	ID	FirstName	LastName	Address
1	101	Prashant	Pandey	▸ {"AddressLine1": "D104 Gopalan Squire", "AddressLine2": "Whitefield", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}
2	102	David	Turner	▸ {"AddressLine1": "109 Park Street", "AddressLine2": "Richmond", "City": "London", "State": "London", "Country": "Engaland", "Pin": "EC1A"}
3	103	Katie	Mcloskey	▸ {"AddressLine1": "9th Avenue", "AddressLine2": "Dorsy Road", "City": "Belfast", "State": "Belfast", "Country": "Northern Ireland", "Pin": "BT1 1BG"}
4	104	Nasima	Khatun	▸ {"AddressLine1": "G105 MG Tower", "AddressLine2": "Bregade Road", "City": "Kolkata", "State": "West Bengal", "Country": "India", "Pin": "7000001"}
5	105	Pritam	Jain	▸ {"AddressLine1": "M206 Richmond Tower", "AddressLine2": "Electronic City", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}

Showing all 5 rows.

Command took 0.29 seconds -- by prashant@scholarnest.com at 6/4/2022, 12:02:08 PM on demo-cluster

Assignment:

I mean, I want a person data frame that looks like this.



ID	FirstName	LastName
101	Prashant	Pandey
102	David	Turner
103	Katie	Mcloskey
104	Nasima	Khatun
105	Pritam	Jain

Assignment:

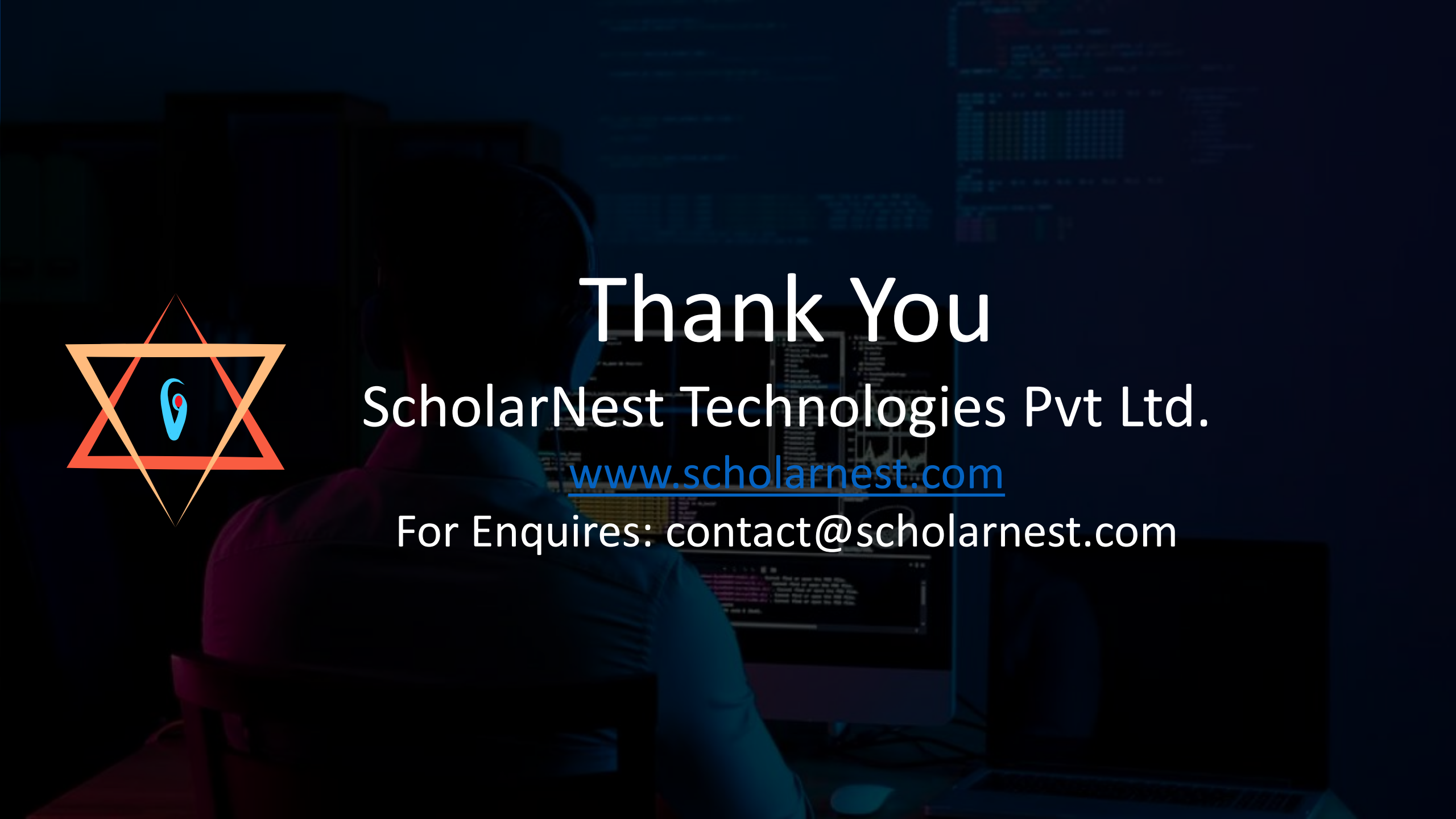
And I want a person_address Dataframe that looks like this.

ID	AddressLine1	AddressLine2	City	State	Country	Pin
101	D104 Gopalan Squire	Whitefield	Bangalore	Karnataka	India	560001
102	109 Park Street	Richmond	London	London	Engaland	EC1A
103	9th Avenue	Dorsy Road	Belfast	Belfast	Northern Ireland	BT1 1BG
104	G105 MG Tower	Bregade Road	Kolkata	West Bengal	India	7000001
105	M206 Richmond Tower	Electronic City	Bangalore	Karnataka	India	560001

I leave it as an exercise for you.

However, I have already written the code and kept it in the notebook.

You can download it from your course material and refer to it.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Working with Different Data Types

Lecture:
Working with Arrays

Complex Types – Arrays

Apache Spark offers you three complex data types.

1. StructType()
2. ArrayType()
3. Maptype()

We already covered StructType() and learned to work with it.

This lecture will extend the earlier example to explore the ArrayType().

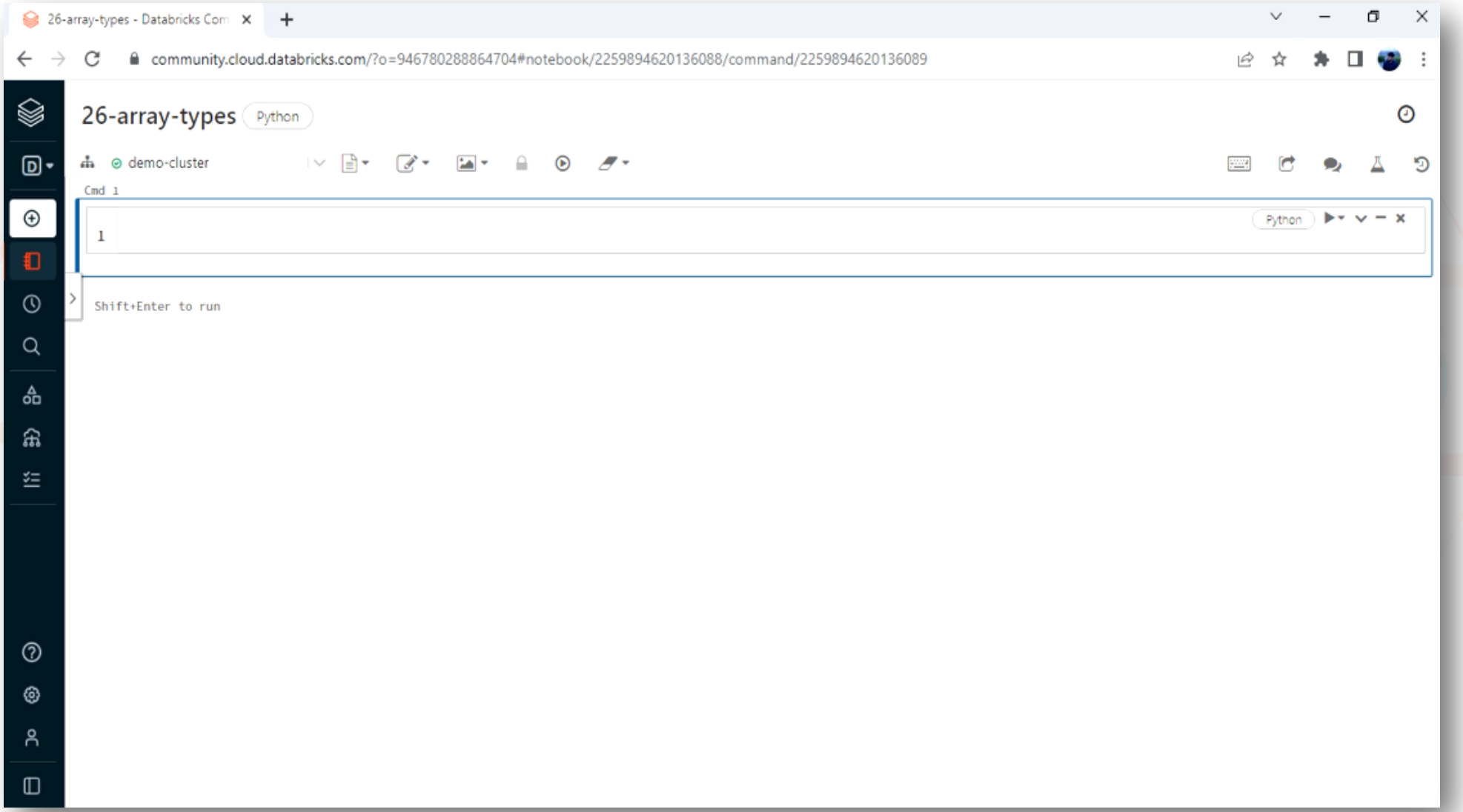
I showed you a JSON data file in the earlier lecture.
I am showing you a record from the data file, so that you understand the record structure. It represents four fields about a person: ID, FirstName, LastName, and Address.
The address is a complex type, and we learned that the address is a struct of AddressLine1, AddressLine2, City, and so on.

```json
{
  "ID": "101",
  "FirstName": "Prashant",
  "LastName": "Pandey",
  "Address": {
    "AddressLine1": "D104 Gopalan Squire",
    "AddressLine2": "Whitefield",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  }
}
```


I have added another field to the same record, skills.
The skills column is also a complex type. But it is a little more complex than the struct. It is an array of a struct. I mean, the address is just one struct. But the Skills is an array of structs. You have two child records in this array. However, you may have 3 or 10 or even more. Now let us look at how do we load this record into the Spark Dataframe.

```
{
  "ID": "101",
  "FirstName": "Prashant",
  "LastName": "Pandey",
  "Address": {
    "AddressLine1": "D104 Gopalan Squire",
    "AddressLine2": "Whitefield",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  },
  "Skills": [
    { "Skill": "Apache Spark", "YearsOfExperience": "5" },
    { "Skill": "Apache Kafka", "YearsOfExperience": "6" }
  ]
}
```

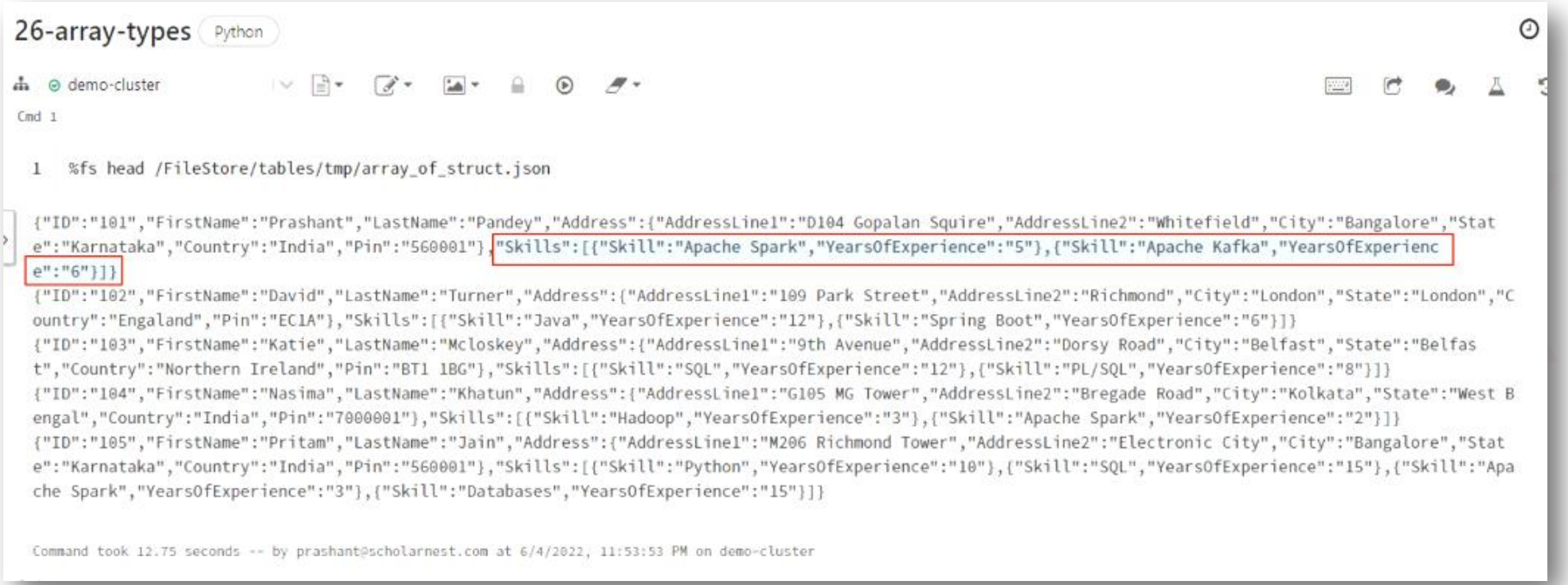

Go to your Databricks workspace and create a new notebook. **(Reference : 26-array-types)**



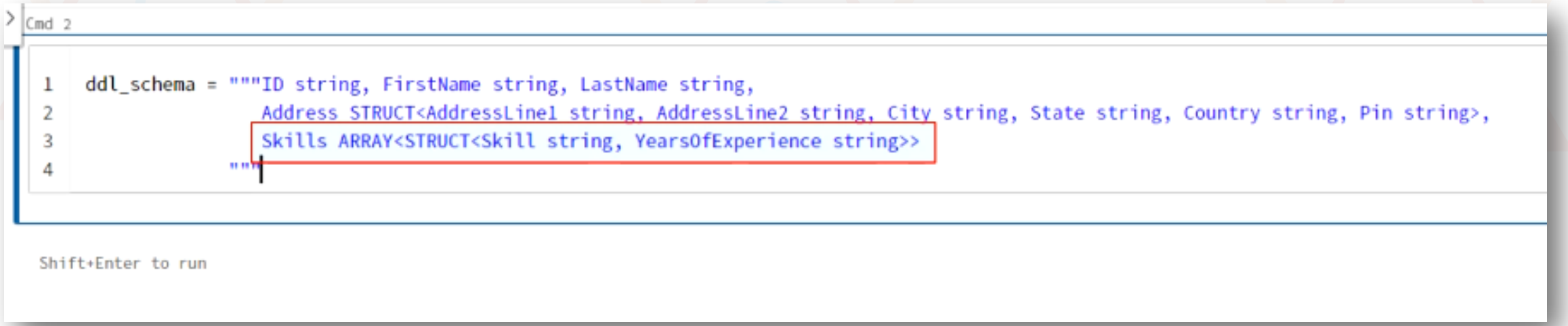
I modified the earlier data file and created a new one with an array column.

(Reference: data/array-of-struct.json)

You can see the skills field here. The first person has two skills: Apache Spark and Apache Kafka.



Here is the schema definition for the record.
You already learned about the first two lines in the earlier lecture. I added a third line to define the skills column. The column name is Skills, and the column type is ARRAY. We define the array element type inside the angle brackets. And the array element is a STRUCT. It could be a simple string or a long value. But in our case, it is a struct. Then we define the details of the struct.



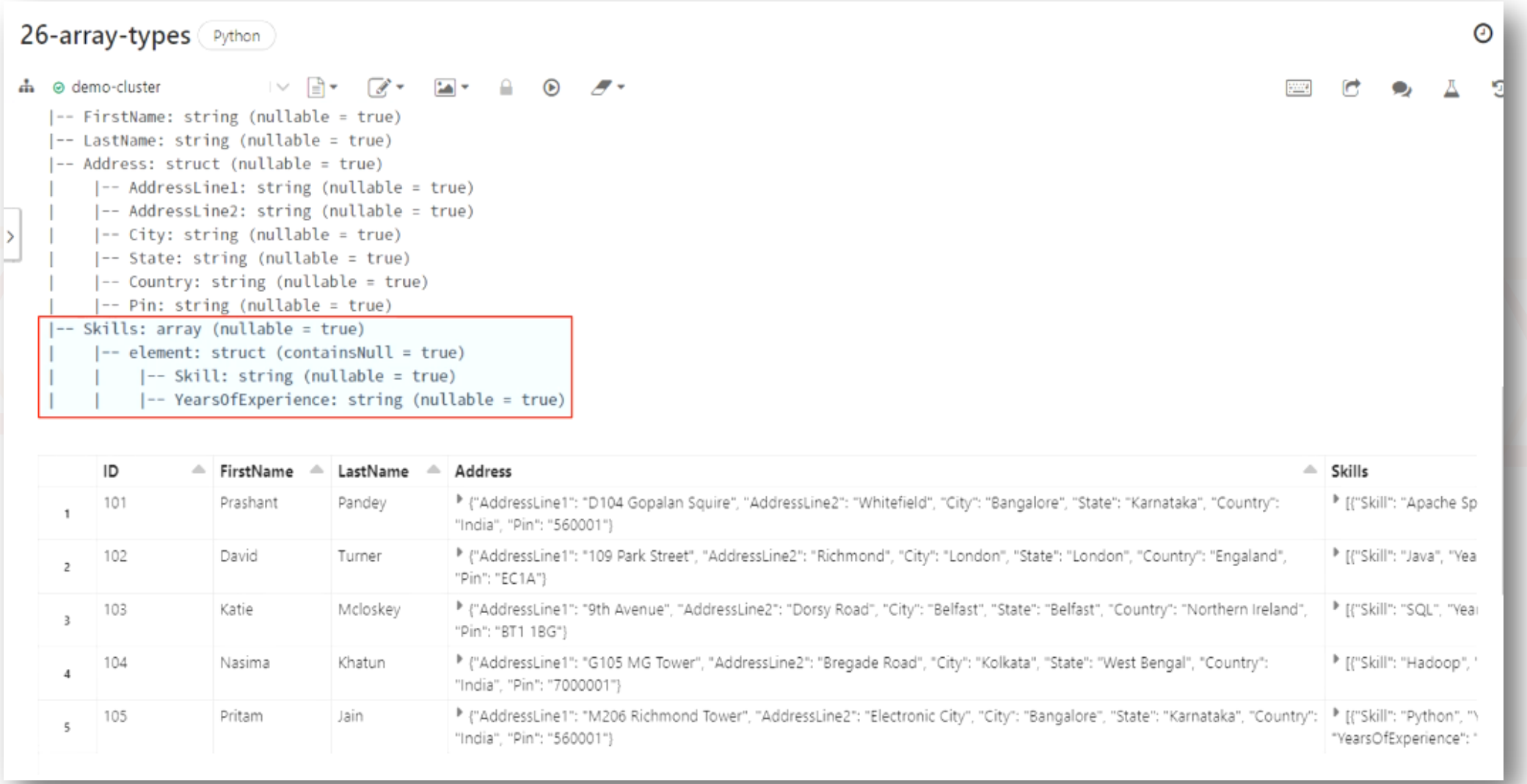
Once you know how to define array columns in your Dataframe, you can easily load the data file into a Dataframe as shown below.

Cmd 2

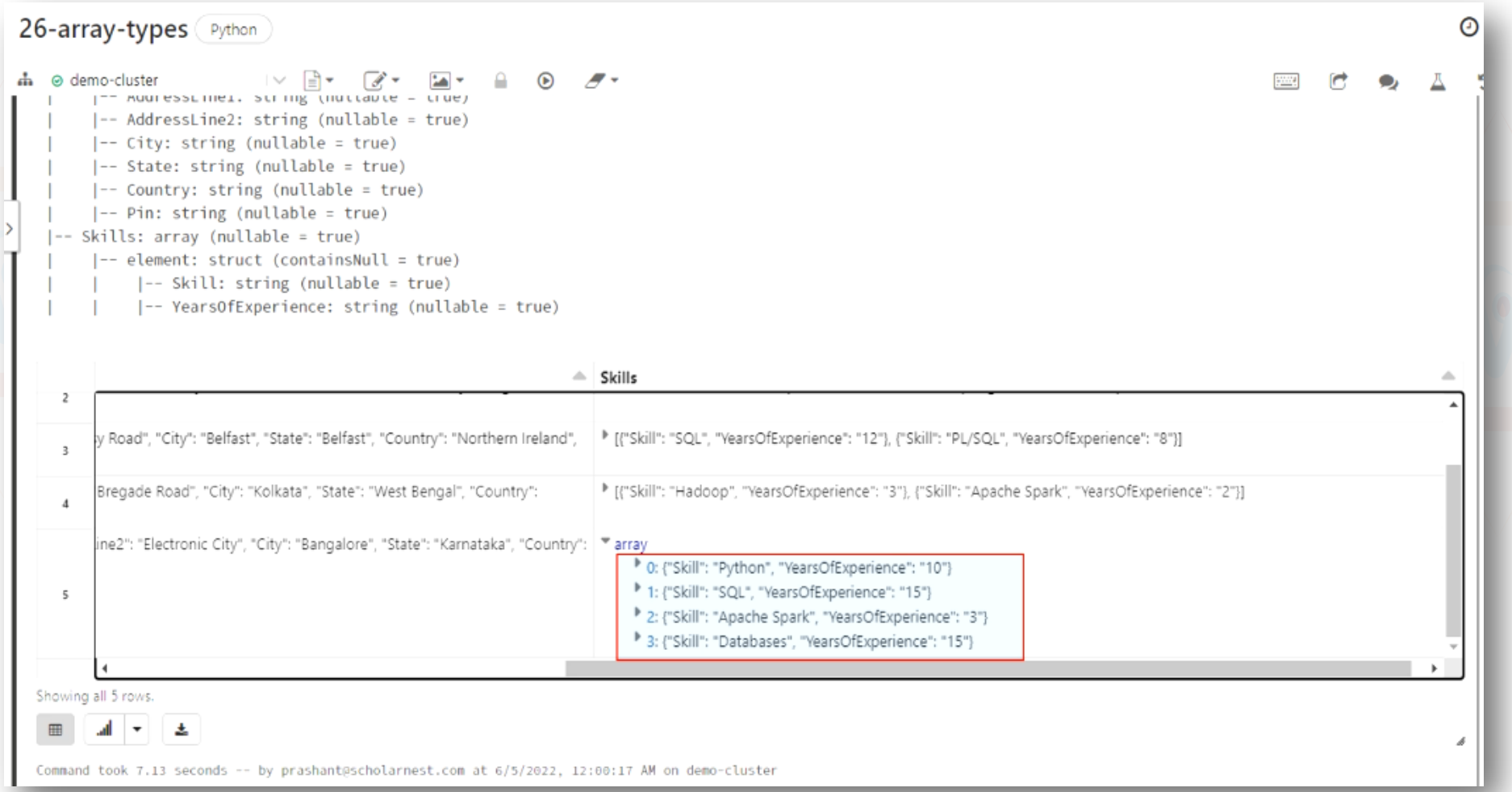
```
1  ddl_schema = """ID string, FirstName string, LastName string,
2                  Address STRUCT<AddressLine1 string, AddressLine2 string, City string, State string, Country string, Pin string>,
3                  Skills ARRAY<STRUCT<Skill string, YearsOfExperience string>>
4                """
5
6  df = spark.read \
7          .format("json") \
8          .schema(ddl_schema) \
9          .load("/FileStore/tables/tmp/array_of_struct.json")
10
11 df.printSchema()
12 display(df)
```

Shift+Enter to run

You can see the schema highlighted below. It clearly shows that the Skills are an array. The array element is a struct. And then we have two struct fields.



Look at the Dataframe display.
If you see the skills column of the last record, you can see that we have an array of four elements. The array index starts from zero and goes up to four. Each array element contains a struct of two fields: Skill name and years of experience.



You can access the full array using the column name as shown below.

```
Cmd 3

1   df.select("ID", "Skills").show(truncate=False)

▸ (1) Spark Jobs

+---+-----------------------------------------------------------------+
|ID |Skills                                                           |
+---+-----------------------------------------------------------------+
|101|[{Apache Spark, 5}, {Apache Kafka, 6}]                           |
|102|[{Java, 12}, {Spring Boot, 6}]                                   |
|103|[{SQL, 12}, {PL/SQL, 8}]                                         |
|104|[{Hadoop, 3}, {Apache Spark, 2}]                                 |
|105|[{Python, 10}, {SQL, 15}, {Apache Spark, 3}, {Databases, 15}]    |
+---+-----------------------------------------------------------------+

Command took 1.15 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:02:32 AM on demo-cluster
```


If you want to access the individual elements of the array, you can use array subscript, as given in the screenshot below.

So you can access the first element, the second element, and so on. But remember, accessing an array element is an expression. So you can use it from selectExpr(), or you must evaluate it using the expr() function.

Cmd 4

```
1  df.selectExpr("ID",
2                "Skills[0] as first_skill",
3                "Skills[1] as second_skill",
4                "Skills[2] as third_skill").show()
```

▸ (1) Spark Jobs

```
+---+----------------+----------------+----------------+
| ID|      first_skill|     second_skill|      third_skill|
+---+----------------+----------------+----------------+
|101|{Apache Spark, 5}|{Apache Kafka, 6}|            null|
|102|       {Java, 12}| {Spring Boot, 6}|            null|
|103|        {SQL, 12}|      {PL/SQL, 8}|            null|
|104|      {Hadoop, 3}|{Apache Spark, 2}|            null|
|105|     {Python, 10}|        {SQL, 15}|{Apache Spark, 3}|
+---+----------------+----------------+----------------+
```

Command took 0.98 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:03:50 AM on demo-cluster

In this example, each array element is a struct. So you can also use dot notation after the array subscript as shown below.

Cmd 5

```
1  df.selectExpr("ID",
2                "Skills[0].Skill as primary_skill",
3                "Skills[1].YearsOfExperience as experience").show()
```

▸ (1) Spark Jobs

```
+---+-------------+----------+
| ID|primary_skill|experience|
+---+-------------+----------+
|101| Apache Spark|         6|
|102|         Java|         6|
|103|          SQL|         8|
|104|       Hadoop|         2|
|105|       Python|        15|
+---+-------------+----------+
```

Command took 0.95 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:05:58 AM on demo-cluster

You can also explode() the array and convert it into rows.
So, I took the ID so I could identify the person. Then I exploded the array separating each element of the array and converting it into a row. Person 101 had two skills, so we see two records for 101. Similarly, person 105 had four skills, and we see four records. The explode() is a super useful function for working with ArrayType().

Cmd 6

```
1  df.selectExpr("ID", "explode(Skills) as exploded_skills") \
2      .show()
```

▸ (1) Spark Jobs

```
+---+----------------+
| ID|  exploded_skills|
+---+----------------+
|101|{Apache Spark, 5}|
|101|{Apache Kafka, 6}|
|102|       {Java, 12}|
|102| {Spring Boot, 6}|
|103|        {SQL, 12}|
|103|      {PL/SQL, 8}|
|104|      {Hadoop, 3}|
|104|{Apache Spark, 2}|
|105|     {Python, 10}|
|105|        {SQL, 15}|
|105|{Apache Spark, 3}|
|105|  {Databases, 15}|
+---+----------------+
```

Command took 1.08 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:07:09 AM on demo-cluster

Person 101 had two skills, so we see two records for 101.

Cmd 6

```
1  df.selectExpr("ID", "explode(Skills) as exploded_skills") \
2      .show()
```

▸ (1) Spark Jobs

```
+---+----------------+
| ID|  exploded_skills|
+---+----------------+
|101|{Apache Spark, 5}|
|101|{Apache Kafka, 6}|
|102|       {Java, 12}|
|102| {Spring Boot, 6}|
|103|        {SQL, 12}|
|103|      {PL/SQL, 8}|
|104|      {Hadoop, 3}|
|104|{Apache Spark, 2}|
|105|     {Python, 10}|
|105|        {SQL, 15}|
|105|{Apache Spark, 3}|
|105|  {Databases, 15}|
+---+----------------+
```

Command took 1.08 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:07:09 AM on demo-cluster

Similarly, person 105 had four skills, and we see four records.
So, the explode() is a super useful function for working with ArrayType().

Cmd 6

```
1  df.selectExpr("ID", "explode(Skills) as exploded_skills") \
2      .show()
```

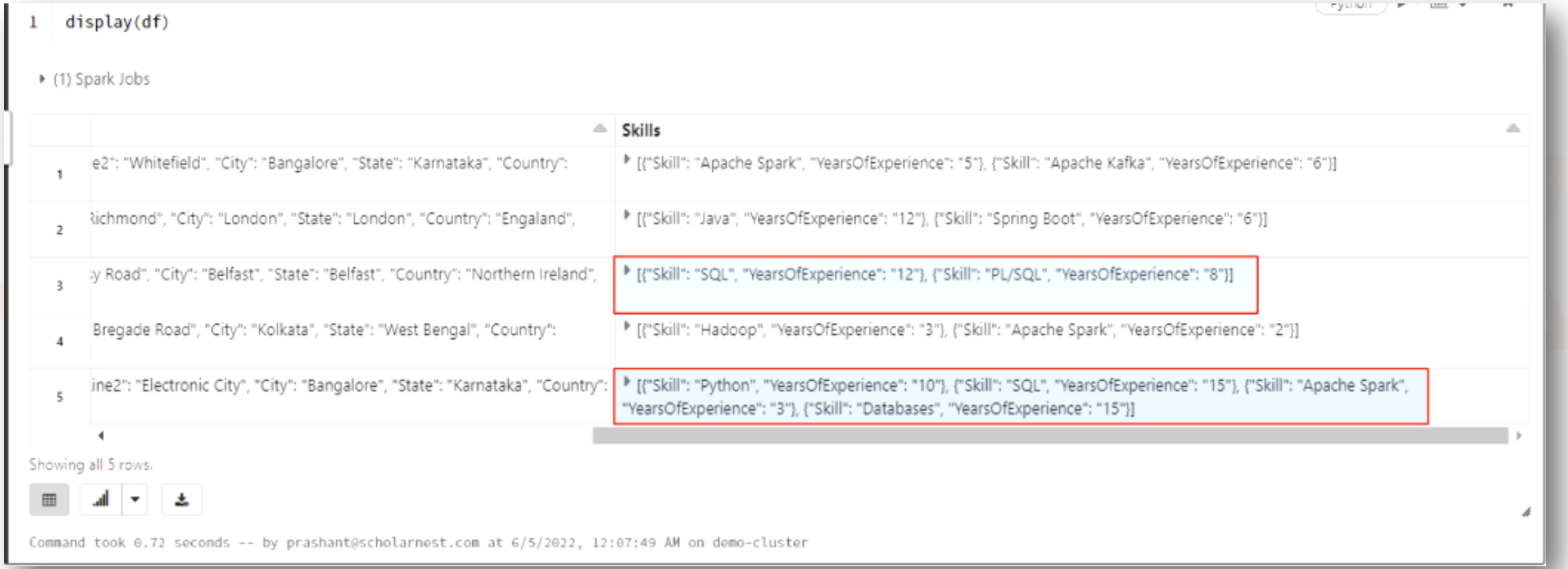
▸ (1) Spark Jobs

```
+---+----------------+
| ID|  exploded_skills|
+---+----------------+
|101|{Apache Spark, 5}|
|101|{Apache Kafka, 6}|
|102|       {Java, 12}|
|102| {Spring Boot, 6}|
|103|        {SQL, 12}|
|103|      {PL/SQL, 8}|
|104|      {Hadoop, 3}|
|104|{Apache Spark, 2}|
|105|     {Python, 10}|
|105|        {SQL, 15}|
|105|{Apache Spark, 3}|
|105|  {Databases, 15}|
+---+----------------+
```

Command took 1.08 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:07:09 AM on demo-cluster

Assignment:

Here is our Dataframe. I want you to find out the person with SQL skills. You can see the third and fifth persons have SQL skills. But how will you query it?

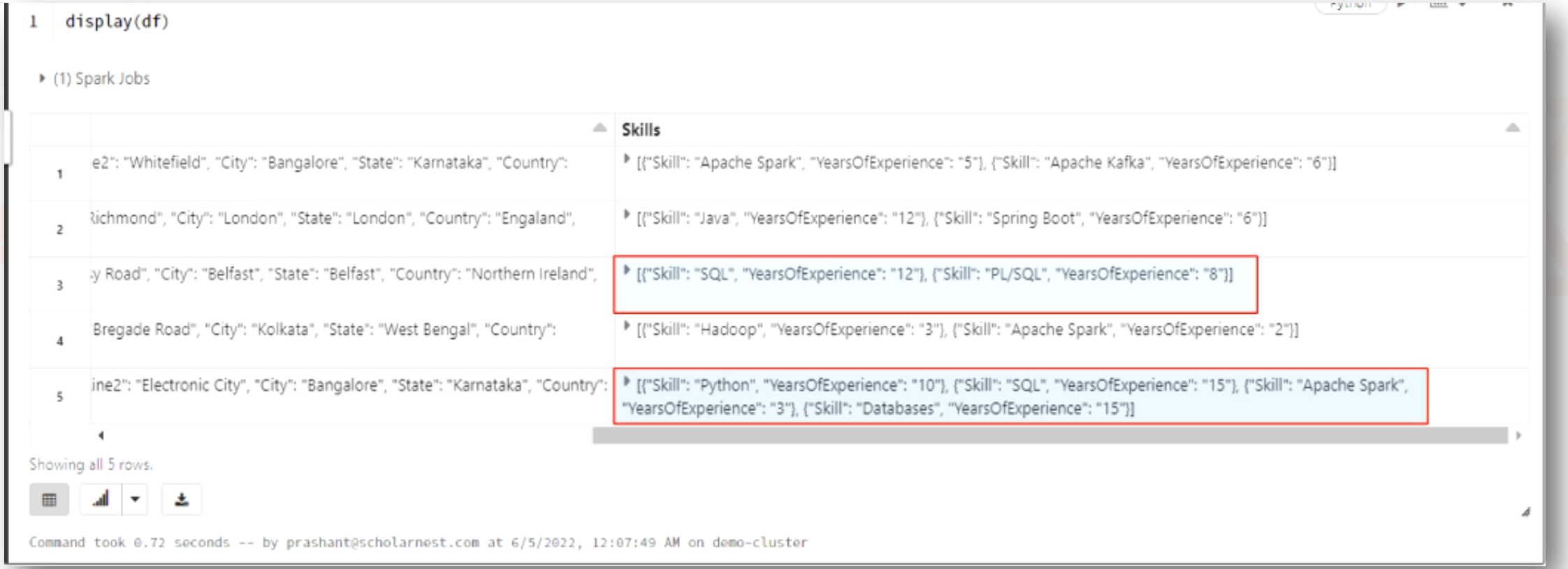


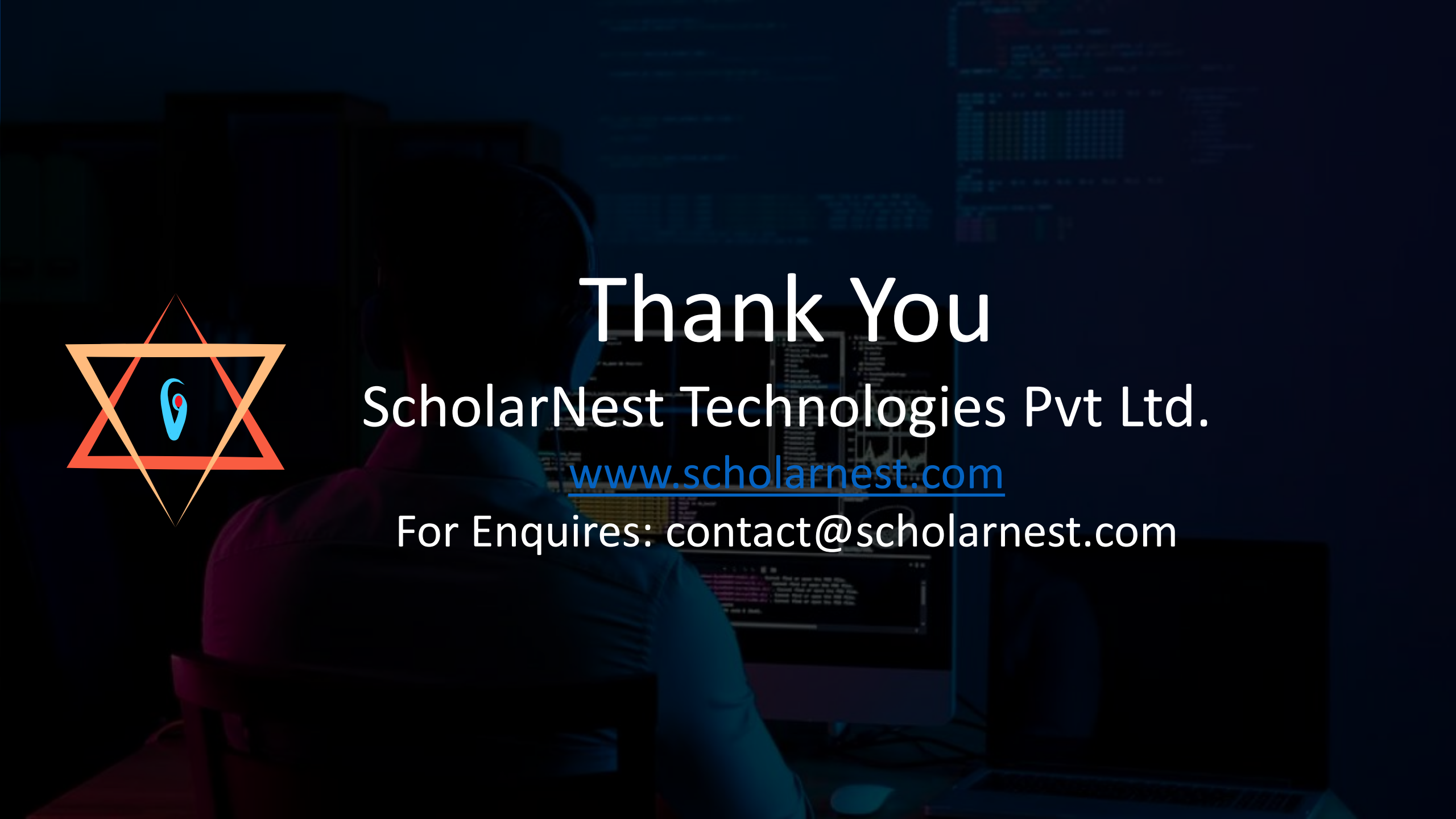
Assignment:

Can you write a Dataframe expression to answer the following question:

Q. Find out the list of person ids having SQL skills?

Try it yourself. I'll give you a hint. Explore the Skills column, and you can quickly write a where condition. The answer is given in your downloadable notebook.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Working with Different Data Types

Lecture:
Working with Maps

Complex Types – MapType

Apache Spark offers you three complex data types.

1. StructType()
2. ArrayType()
3. Maptype()

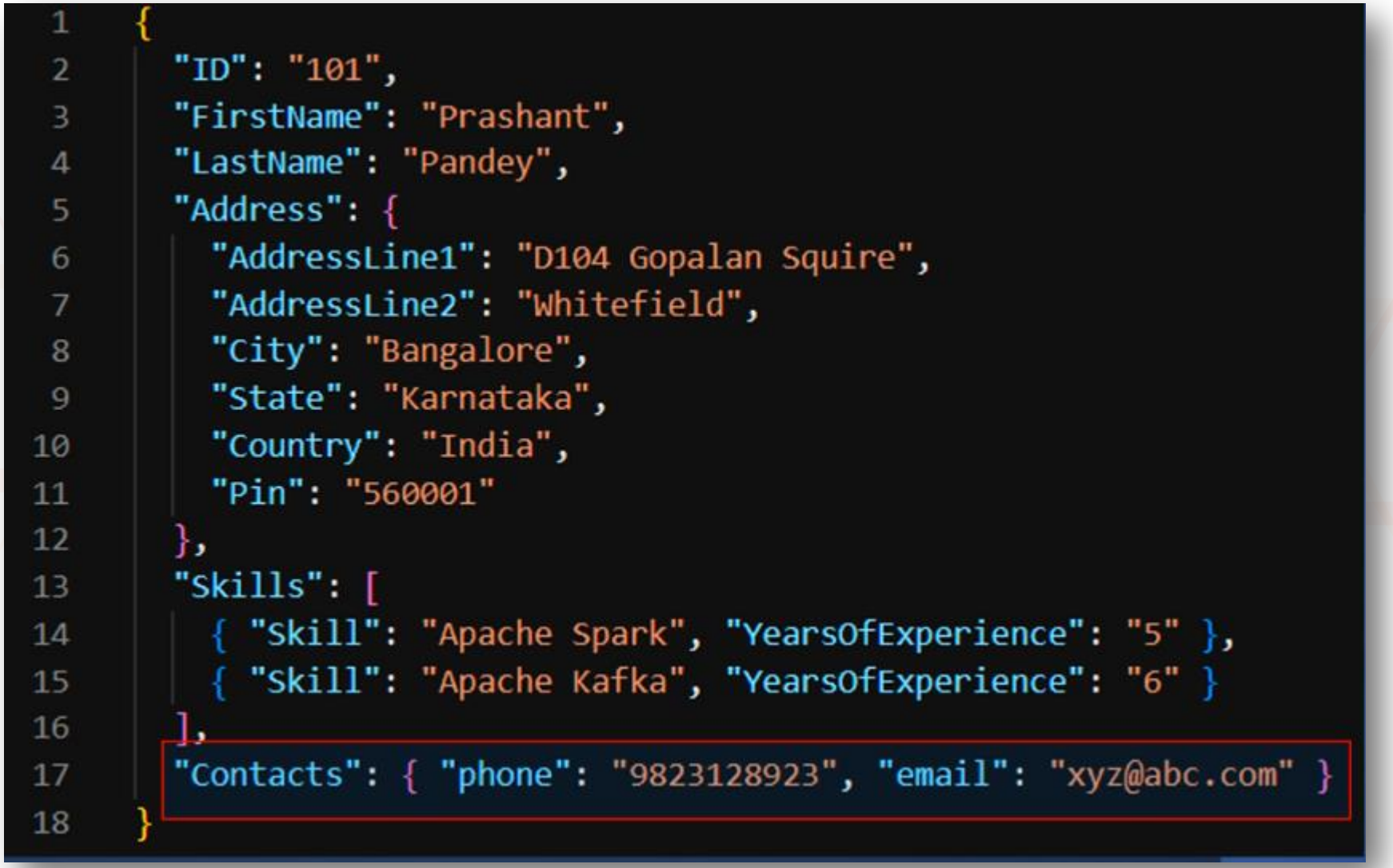
We already covered StructType() and ArrayType() in the earlier lectures.

This lecture will extend the earlier example to explore MapType().

I showed you a JSON data file in the earlier lecture. Here is a record from the data file.
It represents five fields about a person: ID, FirstName, LastName, Address, and Skills.
The address is a complex type, and we learned that the address is a struct of AddressLine1, AddressLine2, City, and so on. The Skills is an Array of StructType(). We learned to define the schema and load such data records.

```json
{
  "ID": "101",
  "FirstName": "Prashant",
  "LastName": "Pandey",
  "Address": {
    "AddressLine1": "D104 Gopalan Squire",
    "AddressLine2": "Whitefield",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  },
  "Skills": [
    { "Skill": "Apache Spark", "YearsOfExperience": "5" },
    { "Skill": "Apache Kafka", "YearsOfExperience": "6" }
  ]
}
```

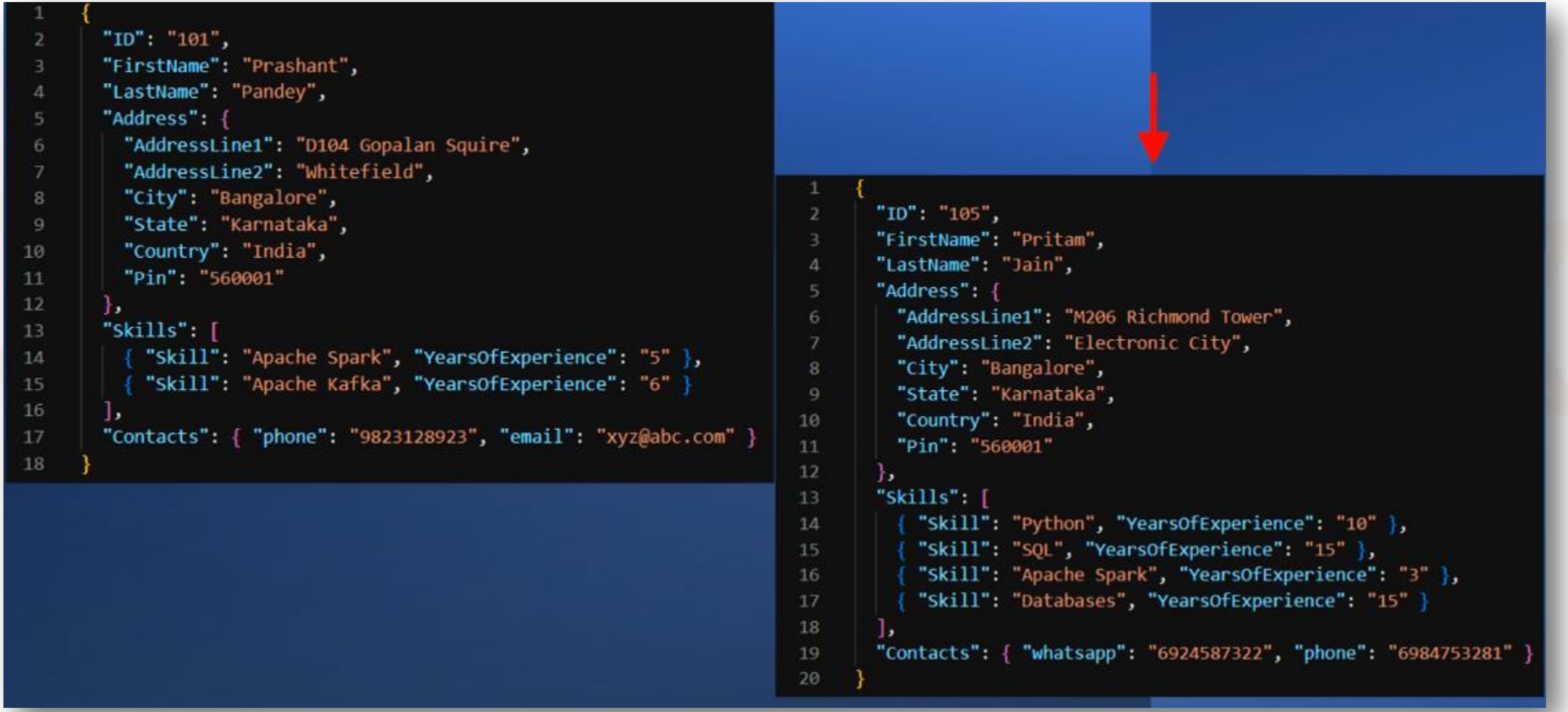

I have added another field to the previous record, the Contacts field. Now, look at the value of the contacts. Does it look like a struct? Yes. It does.



```
{
  "ID": "101",
  "FirstName": "Prashant",
  "LastName": "Pandey",
  "Address": {
    "AddressLine1": "D104 Gopalan Squire",
    "AddressLine2": "Whitefield",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  },
  "Skills": [
    { "Skill": "Apache Spark", "YearsOfExperience": "5" },
    { "Skill": "Apache Kafka", "YearsOfExperience": "6" }
  ],
  "Contacts": { "phone": "9823128923", "email": "xyz@abc.com" }
}
```


Try comparing it with the Address field. The address field is a struct, and it is designed as a column name and column value. For example, the AddressLine1 is the column name, and we also have a value of the AddressLine1. The Contact is also similar. The phone is the column name, and we have a value.

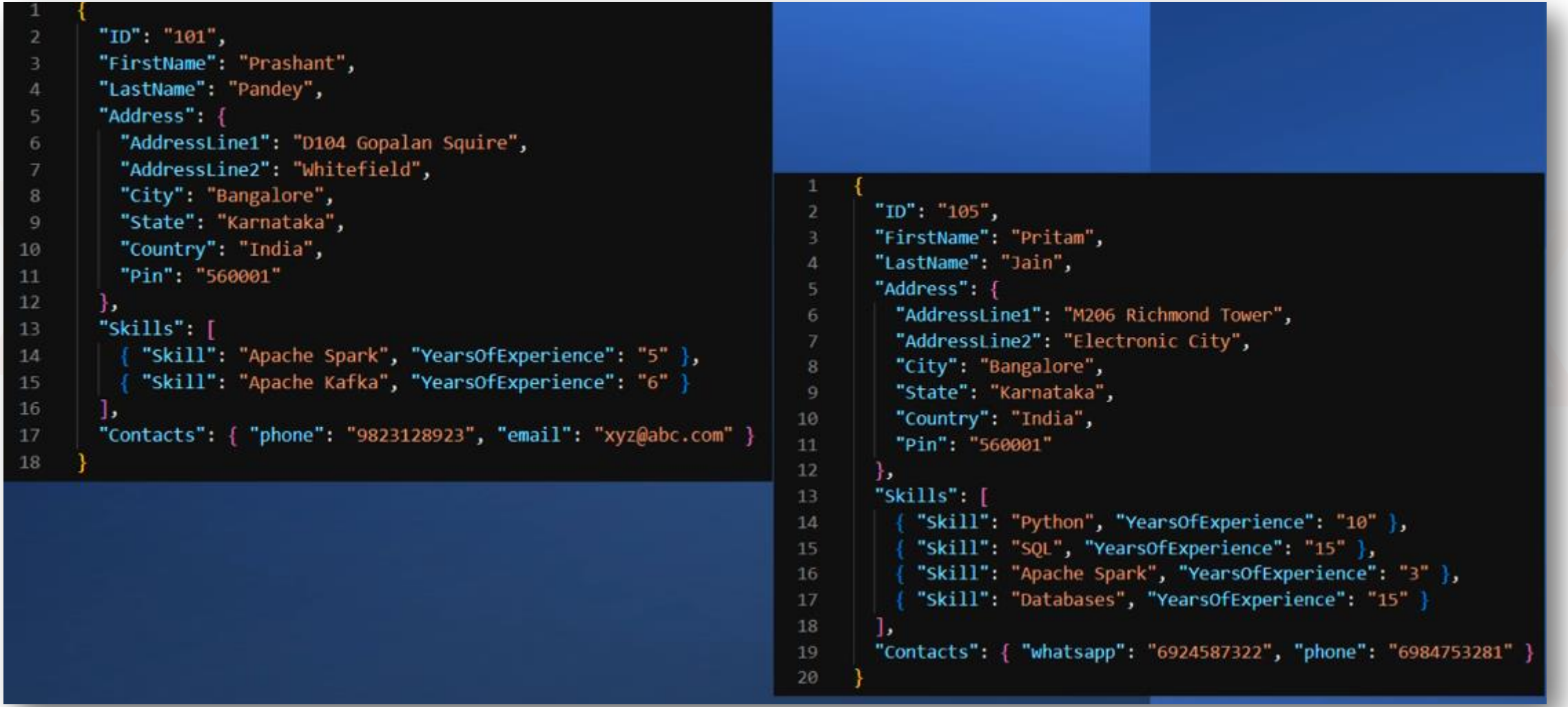
Here is another record from the data file. This guy is also given in the same dataset. The Contact in this record is different than the earlier one. The previous record comes with phone and email fields. But this guy is WhatsApp and phone.



```
{
  "ID": "101",
  "FirstName": "Prashant",
  "LastName": "Pandey",
  "Address": {
    "AddressLine1": "D104 Gopalan Squire",
    "AddressLine2": "Whitefield",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  },
  "Skills": [
    { "Skill": "Apache Spark", "YearsOfExperience": "5" },
    { "Skill": "Apache Kafka", "YearsOfExperience": "6" }
  ],
  "Contacts": { "phone": "9823128923", "email": "xyz@abc.com" }
}
```

```
{
  "ID": "105",
  "FirstName": "Pritam",
  "LastName": "Jain",
  "Address": {
    "AddressLine1": "M206 Richmond Tower",
    "AddressLine2": "Electronic City",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  },
  "Skills": [
    { "Skill": "Python", "YearsOfExperience": "10" },
    { "Skill": "SQL", "YearsOfExperience": "15" },
    { "Skill": "Apache Spark", "YearsOfExperience": "3" },
    { "Skill": "Databases", "YearsOfExperience": "15" }
  ],
  "Contacts": { "whatsapp": "6924587322", "phone": "6984753281" }
}
```

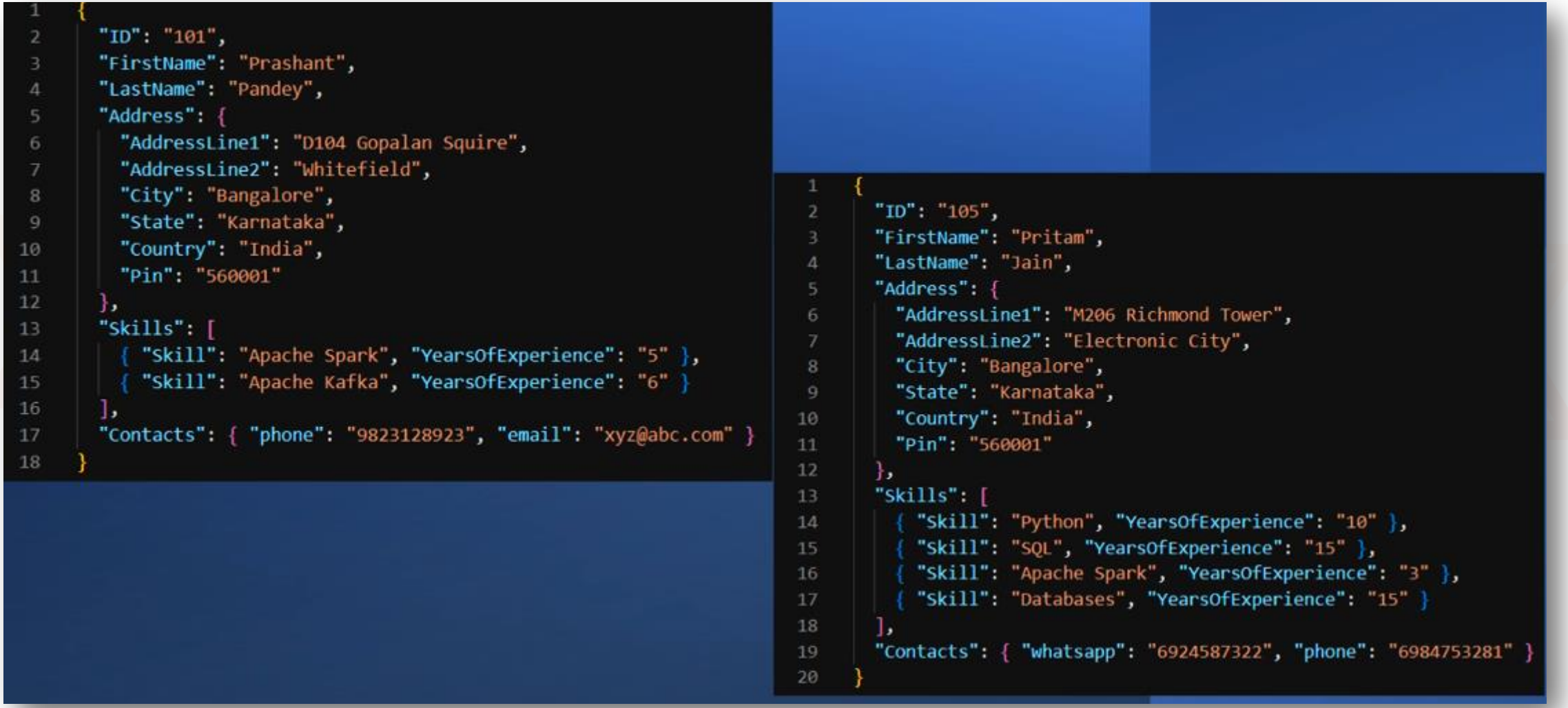

So, the column names are different. And that's a problem.
You cannot define it as a struct because the struct should have fixed and well-defined column names.
But in this case, we do not have a fixed list of columns.



```
{
  "ID": "101",
  "FirstName": "Prashant",
  "LastName": "Pandey",
  "Address": {
    "AddressLine1": "D104 Gopalan Squire",
    "AddressLine2": "Whitefield",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  },
  "Skills": [
    { "Skill": "Apache Spark", "YearsOfExperience": "5" },
    { "Skill": "Apache Kafka", "YearsOfExperience": "6" }
  ],
  "Contacts": { "phone": "9823128923", "email": "xyz@abc.com" }
}
```

```
{
  "ID": "105",
  "FirstName": "Pritam",
  "LastName": "Jain",
  "Address": {
    "AddressLine1": "M206 Richmond Tower",
    "AddressLine2": "Electronic City",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  },
  "Skills": [
    { "Skill": "Python", "YearsOfExperience": "10" },
    { "Skill": "SQL", "YearsOfExperience": "15" },
    { "Skill": "Apache Spark", "YearsOfExperience": "3" },
    { "Skill": "Databases", "YearsOfExperience": "15" }
  ],
  "Contacts": { "whatsapp": "6924587322", "phone": "6984753281" }
}
```

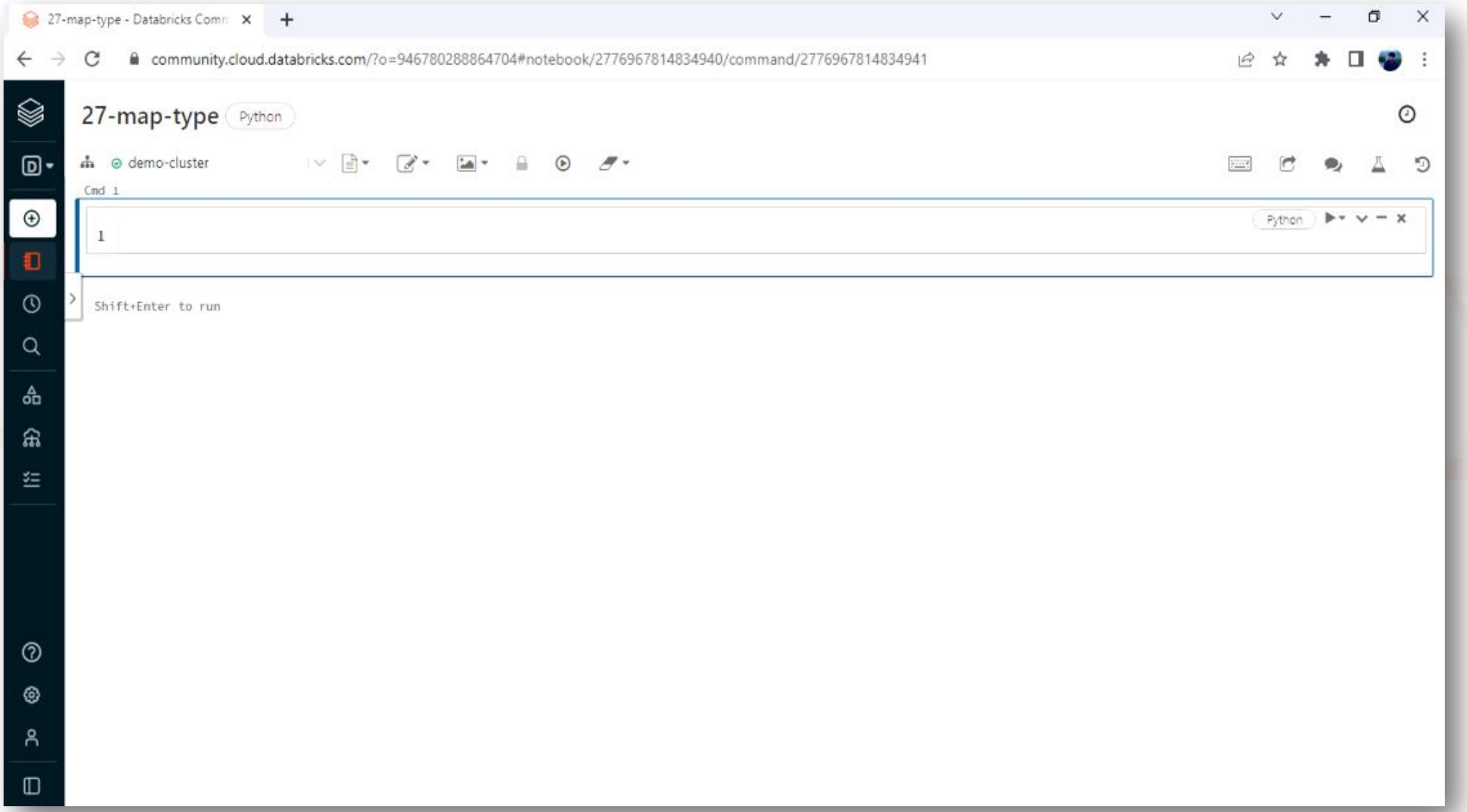

So how do we handle this type of problems? Spark gives you Map type() to handle it. The MapType() is a list of key/value pairs. And we use it when we do not have a fixed column name.



```
{
  "ID": "101",
  "FirstName": "Prashant",
  "LastName": "Pandey",
  "Address": {
    "AddressLine1": "D104 Gopalan Squire",
    "AddressLine2": "Whitefield",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  },
  "Skills": [
    { "Skill": "Apache Spark", "YearsOfExperience": "5" },
    { "Skill": "Apache Kafka", "YearsOfExperience": "6" }
  ],
  "Contacts": { "phone": "9823128923", "email": "xyz@abc.com" }
}
```

```
{
  "ID": "105",
  "FirstName": "Pritam",
  "LastName": "Jain",
  "Address": {
    "AddressLine1": "M206 Richmond Tower",
    "AddressLine2": "Electronic City",
    "City": "Bangalore",
    "State": "Karnataka",
    "Country": "India",
    "Pin": "560001"
  },
  "Skills": [
    { "Skill": "Python", "YearsOfExperience": "10" },
    { "Skill": "SQL", "YearsOfExperience": "15" },
    { "Skill": "Apache Spark", "YearsOfExperience": "3" },
    { "Skill": "Databases", "YearsOfExperience": "15" }
  ],
  "Contacts": { "whatsapp": "6924587322", "phone": "6984753281" }
}
```


Go to your Databricks workspace and create a new notebook. **(Reference : 27-map-type)**



I modified the earlier data file and created a new one with the Contacts column.
(Reference: data/map-type-records.json)
Look at the Contacts in these records. The first record shows phone and email.

Cmd 1

```
%fs head /FileStore/tables/tmp/map_type_records.json
```


```
{"ID":"101","FirstName":"Prashant","LastName":"Pandey","Address":{"AddressLine1":"D104 Gopalan Squire","AddressLine2":"Whitefield","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"},"Skills":[{"Skill":"Apache Spark","YearsOfExperience":"5"},{"Skill":"Apache Kafka","YearsOfExperienc
e":"6"}],"Contacts":{"phone":"9823128923","email":"xyz@abc.com"}}
{"ID":"102","FirstName":"David","LastName":"Turner","Address":{"AddressLine1":"109 Park Street","AddressLine2":"Richmond","City":"London","State":"London","C
ountry":"Engaland","Pin":"EC1A"},"Skills":[{"Skill":"Java","YearsOfExperience":"12"},{"Skill":"Spring Boot","YearsOfExperience":"6"}],"Contacts":{"phone":"98
73145698"}}
{"ID":"103","FirstName":"Katie","LastName":"Mcloskey","Address":{"AddressLine1":"9th Avenue","AddressLine2":"Dorsy Road","City":"Belfast","State":"Belfas
t","Country":"Northern Ireland","Pin":"BT1 1BG"},"Skills":[{"Skill":"SQL","YearsOfExperience":"12"},{"Skill":"PL/SQL","YearsOfExperience":"8"}],"Contacts":
{"email":"ert89@abc.com"}}
{"ID":"104","FirstName":"Nasima","LastName":"Khatun","Address":{"AddressLine1":"G105 MG Tower","AddressLine2":"Bregade Road","City":"Kolkata","State":"West B
engal","Country":"India","Pin":"7000001"},"Skills":[{"Skill":"Hadoop","YearsOfExperience":"3"},{"Skill":"Apache Spark","YearsOfExperience":"2"}],"Contacts":
{"email":"magt23@abc.com","office":"7896524689"}}
{"ID":"105","FirstName":"Pritam","LastName":"Jain","Address":{"AddressLine1":"M206 Richmond Tower","AddressLine2":"Electronic City","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"},"Skills":[{"Skill":"Python","YearsOfExperience":"10"},{"Skill":"SQL","YearsOfExperience":"15"},{"Skill":"Apa
che Spark","YearsOfExperience":"3"},{"Skill":"Databases","YearsOfExperience":"15"}],"Contacts":{"whatsapp":"6924587322","phone":"6984753281"}}
```


Command took 5.26 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:45:24 AM on demo-cluster

The second record has a phone only.

Cmd 1

```
%fs head /FileStore/tables/tmp/map_type_records.json
```


```
{"ID":"101","FirstName":"Prashant","LastName":"Pandey","Address":{"AddressLine1":"D104 Gopalan Squire","AddressLine2":"Whitefield","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"},"Skills":[{"Skill":"Apache Spark","YearsOfExperience":"5"},{"Skill":"Apache Kafka","YearsOfExperienc
e":"6"}],"Contacts":{"phone":"9823128923","email":"xyz@abc.com"}}
{"ID":"102","FirstName":"David","LastName":"Turner","Address":{"AddressLine1":"109 Park Street","AddressLine2":"Richmond","City":"London","State":"London","C
ountry":"Engaland","Pin":"EC1A"},"Skills":[{"Skill":"Java","YearsOfExperience":"12"},{"Skill":"Spring Boot","YearsOfExperience":"6"}],"Contacts":{"phone":"98
73145698"}}
{"ID":"103","FirstName":"Katie","LastName":"Mcloskey","Address":{"AddressLine1":"9th Avenue","AddressLine2":"Dorsy Road","City":"Belfast","State":"Belfas
t","Country":"Northern Ireland","Pin":"BT1 1BG"},"Skills":[{"Skill":"SQL","YearsOfExperience":"12"},{"Skill":"PL/SQL","YearsOfExperience":"8"}],"Contacts":
{"email":"ert89@abc.com"}}
{"ID":"104","FirstName":"Nasima","LastName":"Khatun","Address":{"AddressLine1":"G105 MG Tower","AddressLine2":"Bregade Road","City":"Kolkata","State":"West B
engal","Country":"India","Pin":"7000001"},"Skills":[{"Skill":"Hadoop","YearsOfExperience":"3"},{"Skill":"Apache Spark","YearsOfExperience":"2"}],"Contacts":
{"email":"magt23@abc.com","office":"7896524689"}}
{"ID":"105","FirstName":"Pritam","LastName":"Jain","Address":{"AddressLine1":"M206 Richmond Tower","AddressLine2":"Electronic City","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"},"Skills":[{"Skill":"Python","YearsOfExperience":"10"},{"Skill":"SQL","YearsOfExperience":"15"},{"Skill":"Apa
che Spark","YearsOfExperience":"3"},{"Skill":"Databases","YearsOfExperience":"15"}],"Contacts":{"whatsapp":"6924587322","phone":"6984753281"}}
```


Command took 5.26 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:45:24 AM on demo-cluster

The third one is email only.

Cmd 1

```
%fs head /FileStore/tables/tmp/map_type_records.json
```


```
{"ID":"101","FirstName":"Prashant","LastName":"Pandey","Address":{"AddressLine1":"D104 Gopalan Squire","AddressLine2":"Whitefield","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"},"Skills":[{"Skill":"Apache Spark","YearsOfExperience":"5"},{"Skill":"Apache Kafka","YearsOfExperienc
e":"6"}],"Contacts":{"phone":"9823128923","email":"xyz@abc.com"}}
{"ID":"102","FirstName":"David","LastName":"Turner","Address":{"AddressLine1":"109 Park Street","AddressLine2":"Richmond","City":"London","State":"London","C
ountry":"Engaland","Pin":"EC1A"},"Skills":[{"Skill":"Java","YearsOfExperience":"12"},{"Skill":"Spring Boot","YearsOfExperience":"6"}],"Contacts":{"phone":"98
73145698"}}
{"ID":"103","FirstName":"Katie","LastName":"Mcloskey","Address":{"AddressLine1":"9th Avenue","AddressLine2":"Dorsy Road","City":"Belfast","State":"Belfas
t","Country":"Northern Ireland","Pin":"BT1 1BG"},"Skills":[{"Skill":"SQL","YearsOfExperience":"12"},{"Skill":"PL/SQL","YearsOfExperience":"8"}],"Contacts":
{"email":"ert89@abc.com"}}
{"ID":"104","FirstName":"Nasima","LastName":"Khatun","Address":{"AddressLine1":"G105 MG Tower","AddressLine2":"Bregade Road","City":"Kolkata","State":"West B
engal","Country":"India","Pin":"7000001"},"Skills":[{"Skill":"Hadoop","YearsOfExperience":"3"},{"Skill":"Apache Spark","YearsOfExperience":"2"}],"Contacts":
{"email":"magt23@abc.com","office":"7896524689"}}
{"ID":"105","FirstName":"Pritam","LastName":"Jain","Address":{"AddressLine1":"M206 Richmond Tower","AddressLine2":"Electronic City","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"},"Skills":[{"Skill":"Python","YearsOfExperience":"10"},{"Skill":"SQL","YearsOfExperience":"15"},{"Skill":"Apa
che Spark","YearsOfExperience":"3"},{"Skill":"Databases","YearsOfExperience":"15"}],"Contacts":{"whatsapp":"6924587322","phone":"6984753281"}}
```


Command took 5.26 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:45:24 AM on demo-cluster

So the column names are not consistent.
We could read the Contacts filed as a struct. However, varying column names is putting us in a difficult situation. However, I can handle this complexity using MapType().

Cmd 1

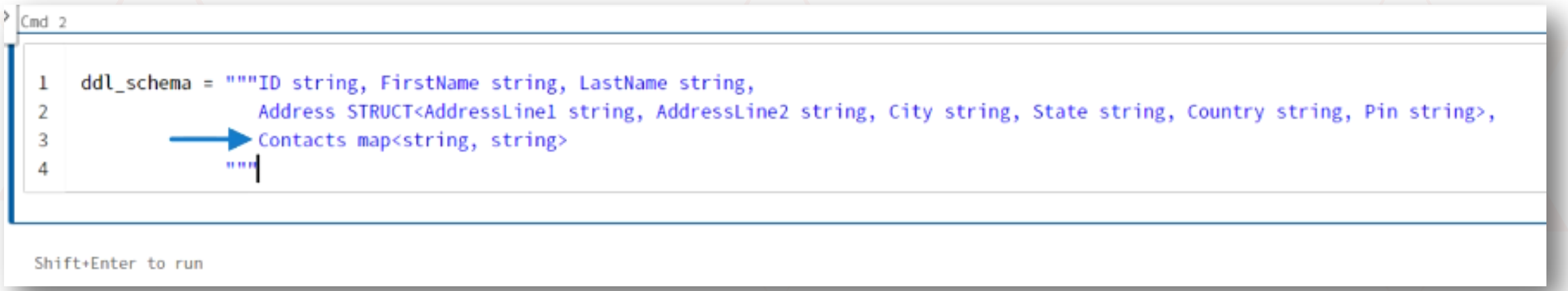
```
%fs head /FileStore/tables/tmp/map_type_records.json
```


```
{"ID":"101","FirstName":"Prashant","LastName":"Pandey","Address":{"AddressLine1":"D104 Gopalan Squire","AddressLine2":"Whitefield","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"},"Skills":[{"Skill":"Apache Spark","YearsOfExperience":"5"},{"Skill":"Apache Kafka","YearsOfExperienc
e":"6"}],"Contacts":{"phone":"9823128923","email":"xyz@abc.com"}}
{"ID":"102","FirstName":"David","LastName":"Turner","Address":{"AddressLine1":"109 Park Street","AddressLine2":"Richmond","City":"London","State":"London","C
ountry":"Engaland","Pin":"EC1A"},"Skills":[{"Skill":"Java","YearsOfExperience":"12"},{"Skill":"Spring Boot","YearsOfExperience":"6"}],"Contacts":{"phone":"98
73145698"}}
{"ID":"103","FirstName":"Katie","LastName":"Mcloskey","Address":{"AddressLine1":"9th Avenue","AddressLine2":"Dorsy Road","City":"Belfast","State":"Belfas
t","Country":"Northern Ireland","Pin":"BT1 1BG"},"Skills":[{"Skill":"SQL","YearsOfExperience":"12"},{"Skill":"PL/SQL","YearsOfExperience":"8"}],"Contacts":
{"email":"ert89@abc.com"}}
{"ID":"104","FirstName":"Nasima","LastName":"Khatun","Address":{"AddressLine1":"G105 MG Tower","AddressLine2":"Bregade Road","City":"Kolkata","State":"West B
engal","Country":"India","Pin":"7000001"},"Skills":[{"Skill":"Hadoop","YearsOfExperience":"3"},{"Skill":"Apache Spark","YearsOfExperience":"2"}],"Contacts":
{"email":"magt23@abc.com","office":"7896524689"}}
{"ID":"105","FirstName":"Pritam","LastName":"Jain","Address":{"AddressLine1":"M206 Richmond Tower","AddressLine2":"Electronic City","City":"Bangalore","Stat
e":"Karnataka","Country":"India","Pin":"560001"},"Skills":[{"Skill":"Python","YearsOfExperience":"10"},{"Skill":"SQL","YearsOfExperience":"15"},{"Skill":"Apa
che Spark","YearsOfExperience":"3"},{"Skill":"Databases","YearsOfExperience":"15"}],"Contacts":{"whatsapp":"6924587322","phone":"6984753281"}}
```


Command took 5.26 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:45:24 AM on demo-cluster

Here is the schema definition given below.

So I defined the Contacts as a map. The map is a key/value data type. The key is a string, and the value is also a string.



Once you have defined the schema, you can load the data using this schema.

Cmd 2

```
1  ddl_schema = """ID string, FirstName string, LastName string,
2                  Address STRUCT<AddressLine1 string, AddressLine2 string, City string, State string, Country string, Pin string>,
3                  Contacts map<string, string>
4              """
5
6  df = spark.read \
7          .format("json") \
8          .schema(ddl_schema) \
9          .load("/FileStore/tables/tmp/map_type_records.json")
10
11 df.printSchema()
12 display(df)
```

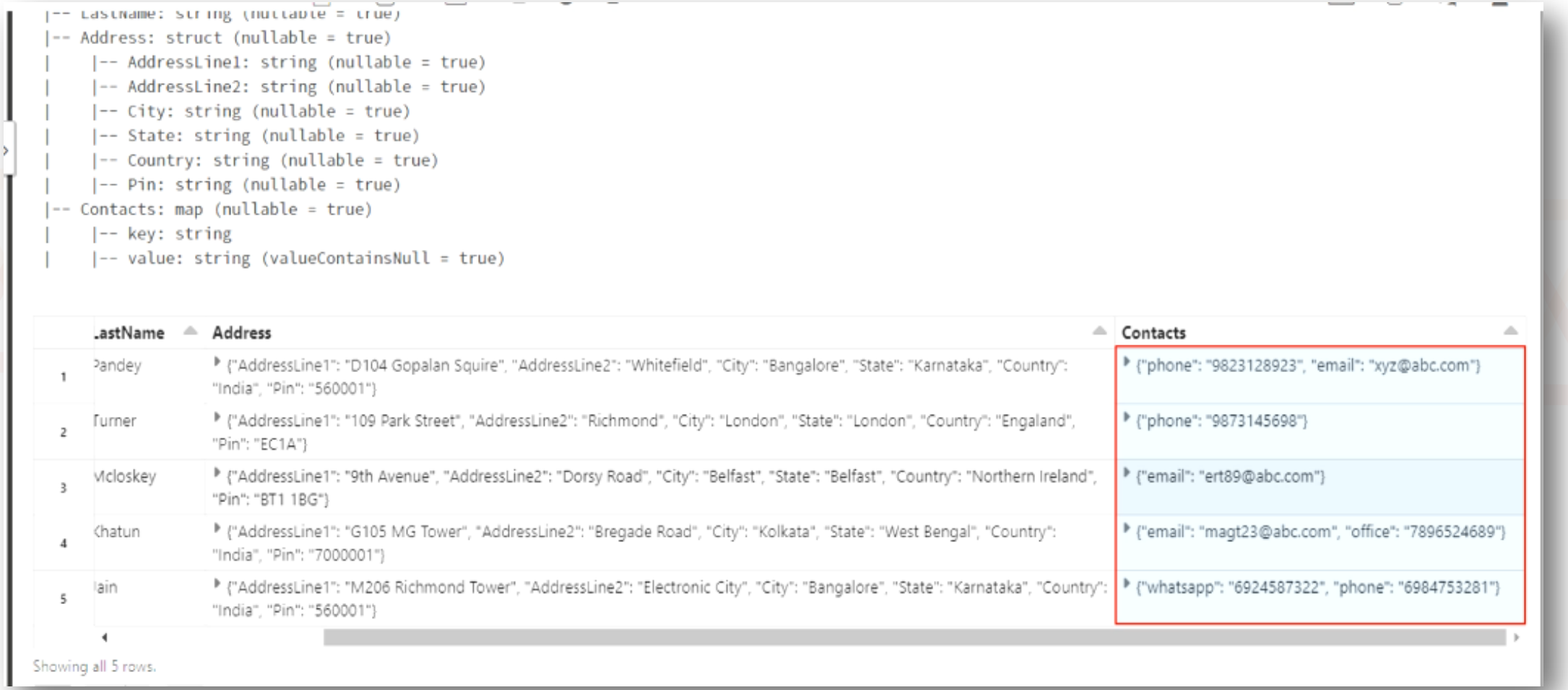
Shift+Enter to run

Here is the output of the printSchema(). Do you see the schema of the Contacts?
It is a map. And we have two fields: key and value. The MapType is defined with a key/value.
And everything is mapped as a key/value pair.

```
|-- LastName: string (nullable = true)
|-- Address: struct (nullable = true)
|    |-- AddressLine1: string (nullable = true)
|    |-- AddressLine2: string (nullable = true)
|    |-- City: string (nullable = true)
|    |-- State: string (nullable = true)
|    |-- Country: string (nullable = true)
|    |-- Pin: string (nullable = true)
|-- Contacts: map (nullable = true)
|    |-- key: string
|    |-- value: string (valueContainsNull = true)
```

	ID	FirstName	LastName	Address	Contacts
1	101	Prashant	Pandey	{"AddressLine1": "D104 Gopalan Squire", "AddressLine2": "Whitefield", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}	{"phone": "9823128
2	102	David	Turner	{"AddressLine1": "109 Park Street", "AddressLine2": "Richmond", "City": "London", "State": "London", "Country": "Engaland", "Pin": "EC1A"}	{"phone": "9873145
3	103	Katie	Mcloskey	{"AddressLine1": "9th Avenue", "AddressLine2": "Dorsy Road", "City": "Belfast", "State": "Belfast", "Country": "Northern Ireland", "Pin": "BT1 1BG"}	{"email": "ert89@ab
4	104	Nasima	Khatun	{"AddressLine1": "G105 MG Tower", "AddressLine2": "Bregade Road", "City": "Kolkata", "State": "West Bengal", "Country": "India", "Pin": "7000001"}	{"email": "magt23@
5	105	Pritam	Jain	{"AddressLine1": "M206 Richmond Tower", "AddressLine2": "Electronic City", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}	{"whatsapp": "69245

If you look at the contacts field in the Dataframe output. You can see that we are reading everything correctly. The first record has got phone and email. The fourth record has email and office. So, everything is loaded correctly.



```
|-- LastName: string (nullable = true)
|-- Address: struct (nullable = true)
|    |-- AddressLine1: string (nullable = true)
|    |-- AddressLine2: string (nullable = true)
|    |-- City: string (nullable = true)
|    |-- State: string (nullable = true)
|    |-- Country: string (nullable = true)
|    |-- Pin: string (nullable = true)
|-- Contacts: map (nullable = true)
|    |-- key: string
|    |-- value: string (valueContainsNull = true)
```

	LastName	Address	Contacts
1	Pandey	{"AddressLine1": "D104 Gopalan Squire", "AddressLine2": "Whitefield", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}	{"phone": "9823128923", "email": "xyz@abc.com"}
2	Turner	{"AddressLine1": "109 Park Street", "AddressLine2": "Richmond", "City": "London", "State": "London", "Country": "Engaland", "Pin": "EC1A"}	{"phone": "9873145698"}
3	Mcloskey	{"AddressLine1": "9th Avenue", "AddressLine2": "Dorsy Road", "City": "Belfast", "State": "Belfast", "Country": "Northern Ireland", "Pin": "BT1 1BG"}	{"email": "ert89@abc.com"}
4	Khatun	{"AddressLine1": "G105 MG Tower", "AddressLine2": "Bregade Road", "City": "Kolkata", "State": "West Bengal", "Country": "India", "Pin": "7000001"}	{"email": "magt23@abc.com", "office": "7896524689"}
5	Jain	{"AddressLine1": "M206 Richmond Tower", "AddressLine2": "Electronic City", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}	{"whatsapp": "6924587322", "phone": "6984753281"}

Showing all 5 rows.

You can access the MapType using the key. So you can take out the phone number using the Contacts and pass the field name. Spark will return null when you do not have a phone number.

Cmd 3

```
1  df.selectExpr("ID",
2                "Contacts['phone'] as phone",
3                "Contacts['email'] as email",
4                "Contacts['whatsapp'] as whatsapp").show()
```

▸ (1) Spark Jobs

```
+---+----------+--------------+----------+
| ID|     phone|         email|  whatsapp|
+---+----------+--------------+----------+
|101|9823128923|   xyz@abc.com|      null|
|102|9873145698|          null|      null|
|103|      null| ert89@abc.com|      null|
|104|      null|magt23@abc.com|      null|
|105|6984753281|          null|6924587322|
+---+----------+--------------+----------+
```

Command took 0.95 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:52:01 AM on demo-cluster

You can also use the explode() function on the maps. The explode will separate the map column into key and value.
So, I took the ID so I could identify the person. Then I exploded the Contacts map separating each key and value into rows.

Cmd 4

```
1  df.selectExpr("ID", "explode(Contacts)") \
2      .show()
```

▸ (1) Spark Jobs

```
+---+--------+--------------+
| ID|     key|         value|
+---+--------+--------------+
|101|   phone|    9823128923|
|101|   email|   xyz@abc.com|
|102|   phone|    9873145698|
|103|   email| ert89@abc.com|
|104|   email|magt23@abc.com|
|104|  office|    7896524689|
|105|whatsapp|    6924587322|
|105|   phone|    6984753281|
+---+--------+--------------+
```

Command took 0.66 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:52:56 AM on demo-cluster

Person 101 had two contacts, so we see two records for 101.

Cmd 4

```
1  df.selectExpr("ID", "explode(Contacts)") \
2      .show()
```

▸ (1) Spark Jobs

```
+---+--------+--------------+
| ID|     key|         value|
+---+--------+--------------+
|101|   phone|    9823128923|
|101|   email|   xyz@abc.com|
|102|   phone|    9873145698|
|103|   email| ert89@abc.com|
|104|   email|magt23@abc.com|
|104|  office|    7896524689|
|105|whatsapp|    6924587322|
|105|   phone|    6984753281|
+---+--------+--------------+
```

Command took 0.66 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:52:56 AM on demo-cluster

Similarly, 102 and 103 have shared only one Contact, so we have only one row for those.

Cmd 4

```
1  df.selectExpr("ID", "explode(Contacts)") \
2      .show()
```

▸ (1) Spark Jobs

```
+---+--------+--------------+
| ID|     key|         value|
+---+--------+--------------+
|101|   phone|    9823128923|
|101|   email|   xyz@abc.com|
|102|   phone|    9873145698|
|103|   email| ert89@abc.com|
|104|   email|magt23@abc.com|
|104|  office|    7896524689|
|105|whatsapp|    6924587322|
|105|   phone|    6984753281|
+---+--------+--------------+
```

Command took 0.66 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:52:56 AM on demo-cluster

Assignment:

You can see the Dataframe we created given below.

Can you write Dataframe code to show you the person's id, FirstName, and WhatsApp number?

We do not want to see the person on the list if they do not share a WhatsApp number.

```
1  display(df)
```

▸ (1) Spark Jobs

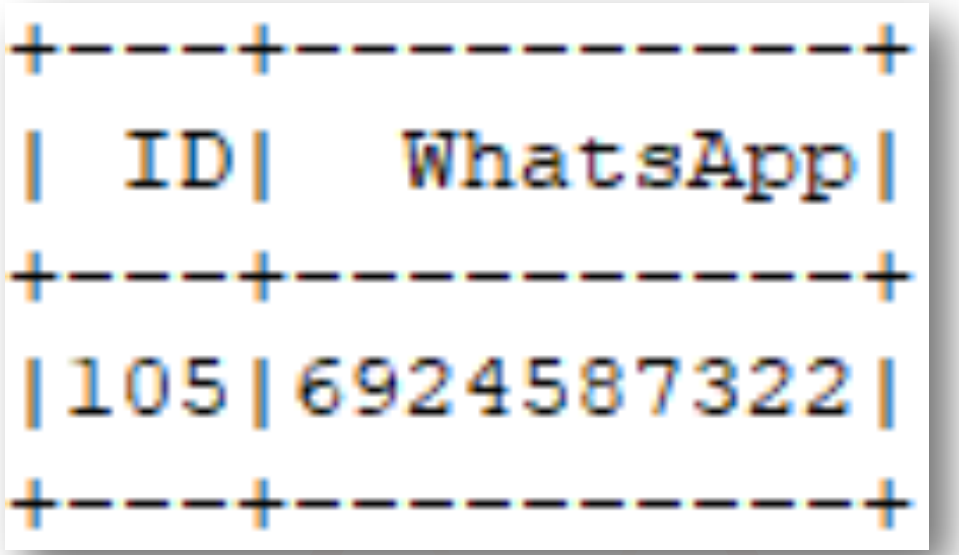
	ID	FirstName	LastName	Address	Contacts
1	101	Prashant	Pandey	{"AddressLine1": "D104 Gopalan Squire", "AddressLine2": "Whitefield", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}	{"phone": "9823128
2	102	David	Turner	{"AddressLine1": "109 Park Street", "AddressLine2": "Richmond", "City": "London", "State": "London", "Country": "Engaland", "Pin": "EC1A"}	{"phone": "9873145
3	103	Katie	Mcloskey	{"AddressLine1": "9th Avenue", "AddressLine2": "Dorsy Road", "City": "Belfast", "State": "Belfast", "Country": "Northern Ireland", "Pin": "BT1 1BG"}	{"email": "ert89@ab
4	104	Nasima	Khatun	{"AddressLine1": "G105 MG Tower", "AddressLine2": "Bregade Road", "City": "Kolkata", "State": "West Bengal", "Country": "India", "Pin": "7000001"}	{"email": "magt23@
5	105	Pritam	Jain	{"AddressLine1": "M206 Richmond Tower", "AddressLine2": "Electronic City", "City": "Bangalore", "State": "Karnataka", "Country": "India", "Pin": "560001"}	{"whatsapp": "6924

Showing all 5 rows.

Command took 0.44 seconds -- by prashant@scholarnest.com at 6/5/2022, 12:53:23 AM on demo-cluster

Assignment:

The output should give only one record. Here is the final output.

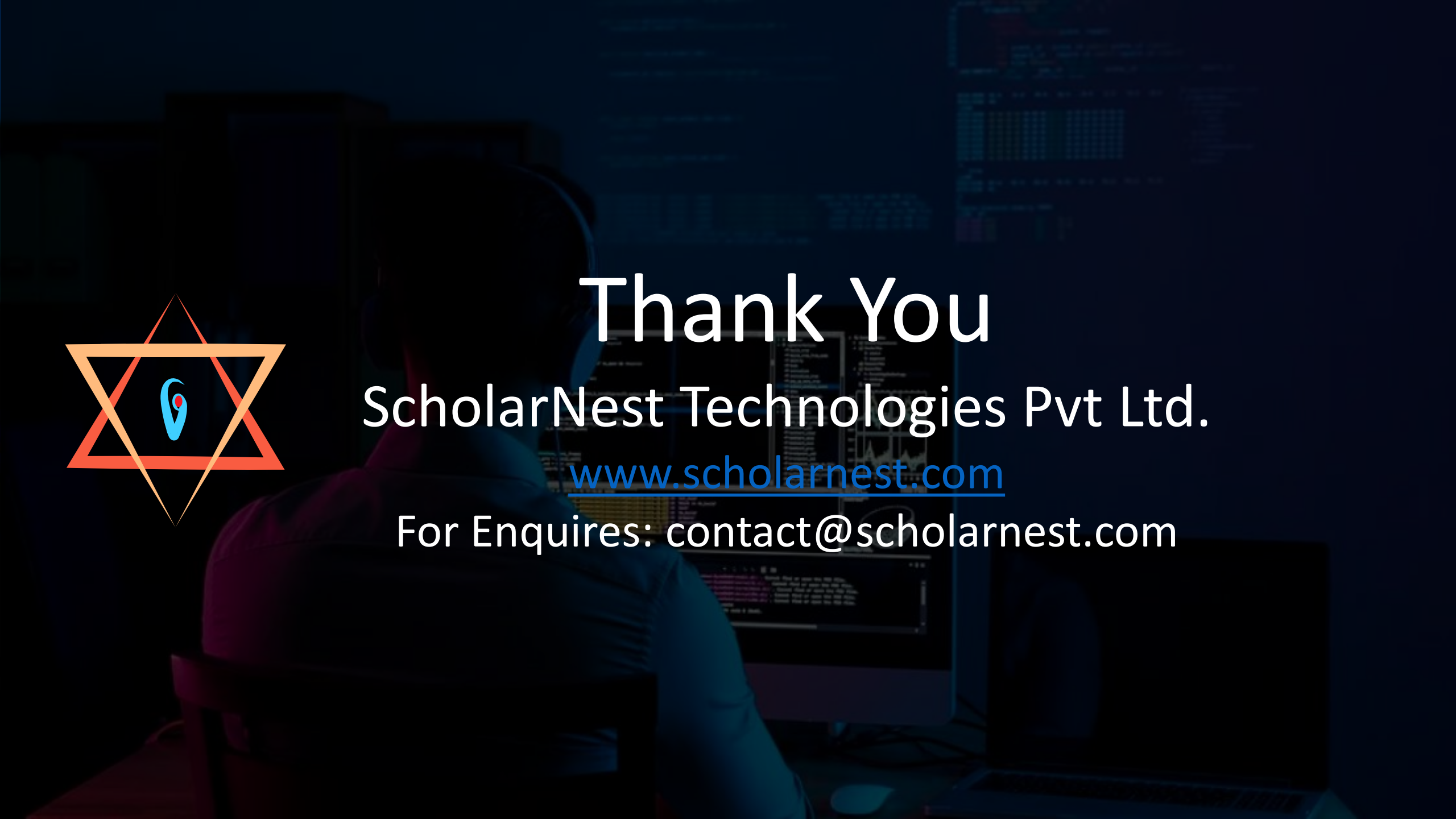


```
+---+----------+
| ID|  WhatsApp|
+---+----------+
|105|6924587322|
+---+----------+
```

Can you write code to produce this output?

Try it yourself.

The answer is given in your downloadable notebook.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com