Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Aggregation

Lecture:
Simple
Aggregation

Simple Aggregations

Aggregations can be classified into three broad categories:

1. Simple Aggregations
2. Grouping Aggregations
3. Windowing Aggregations and summarization.

All aggregations in Spark are implemented via built-in functions.
And these aggregate functions can be categorized into three types:

1. Simple aggregation functions
2. Grouping aggregations
3. Windowing functions

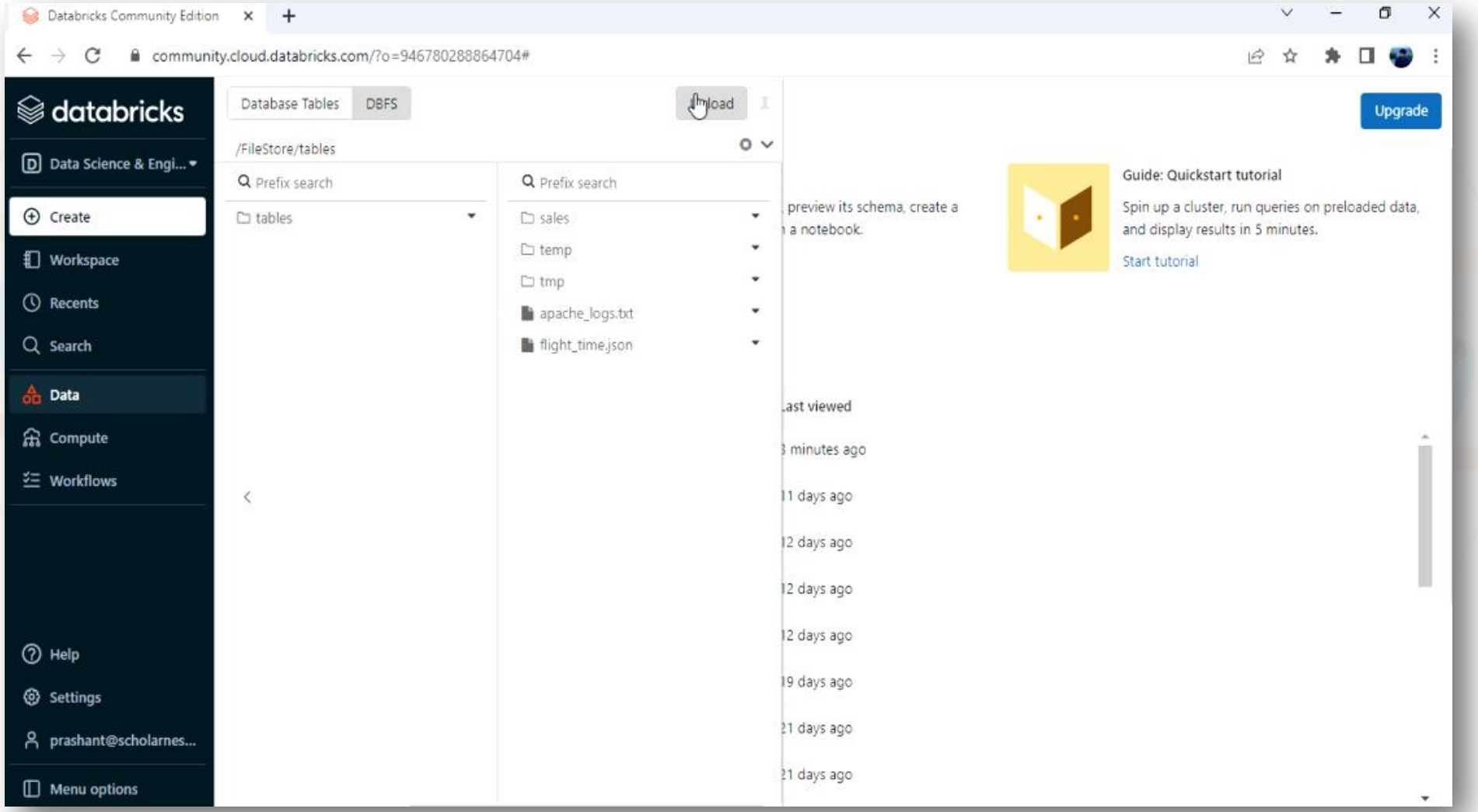
So the aggregations are of three types. And we also have three types of aggregate functions.

Here are some examples of the most commonly used simple aggregation functions:

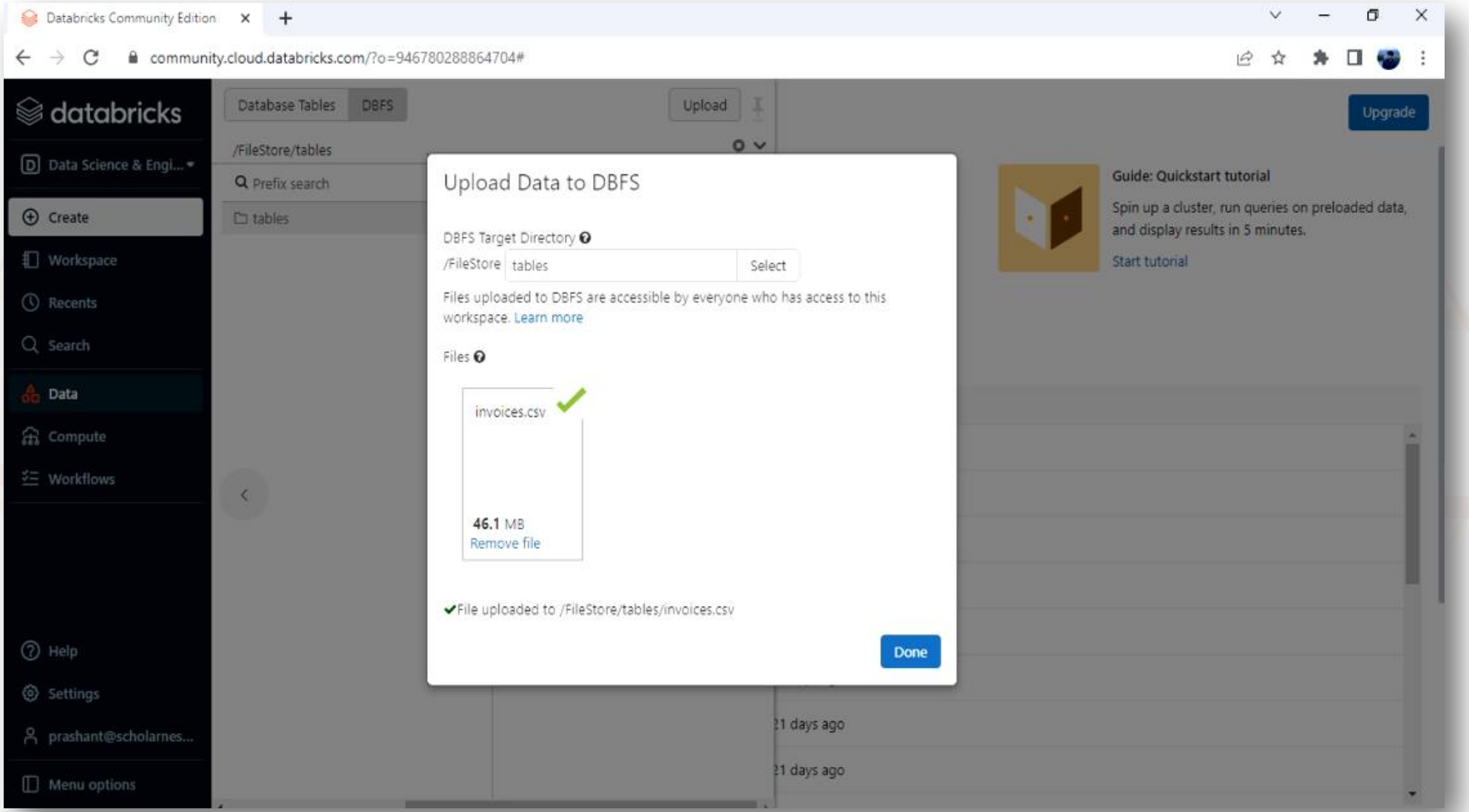
1. avg()
2. count()
3. countDistinct()
4. max()
5. min()
6. mean()
7. sum()
8. product()
9. stddev()
10. variance()

It is not an exhaustive list. We have many other statistical aggregation functions. But the list gives you a good indication of what is available under simple aggregation. Almost all these functions are also used with grouping aggregation.

We need to create some examples. So let me go to the Databricks workspace and upload the retail invoices dataset under the DBFS tab for learning the aggregation.
(Reference: /data/invoices.csv)

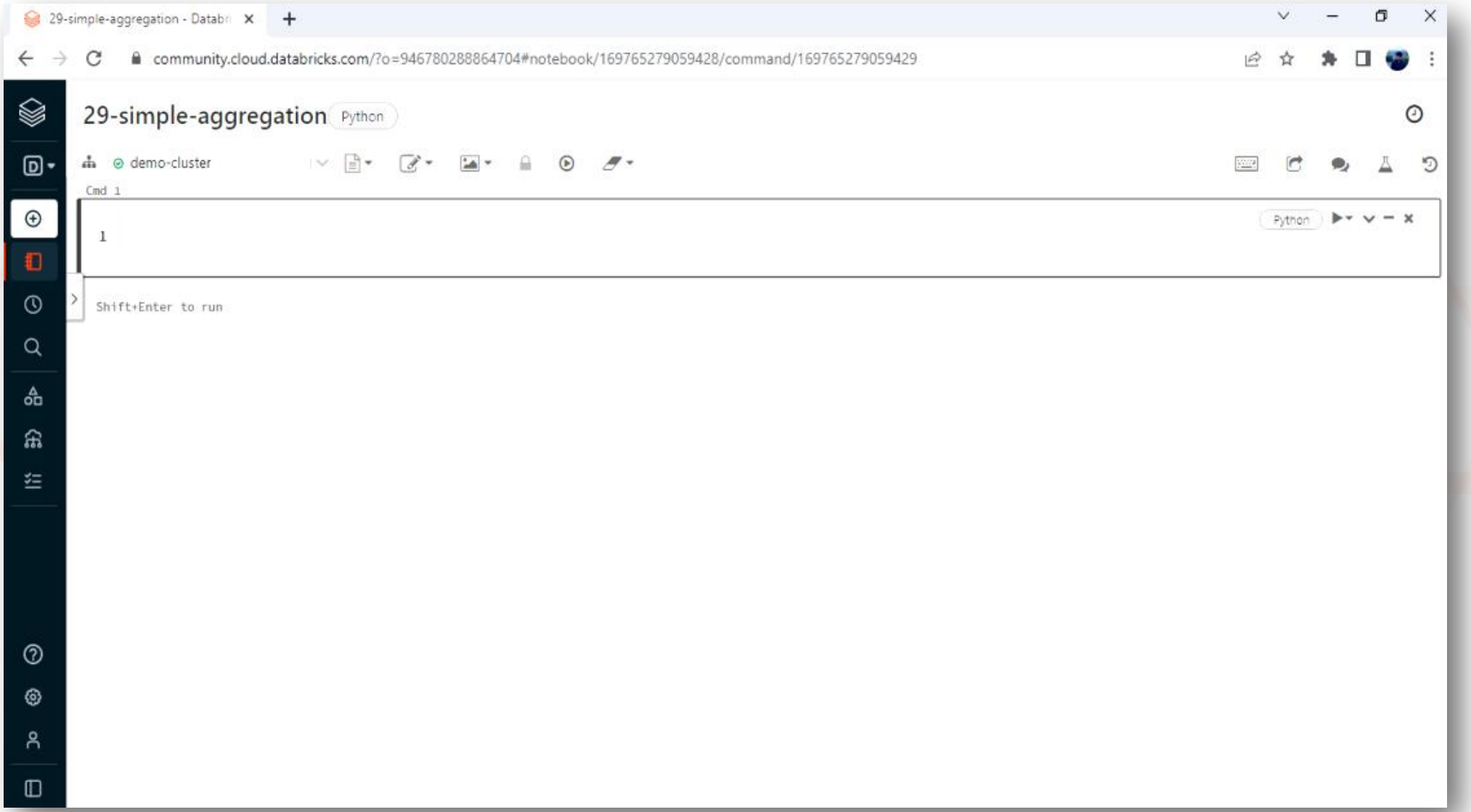


Once the file is uploaded you are all set to see the example in action.

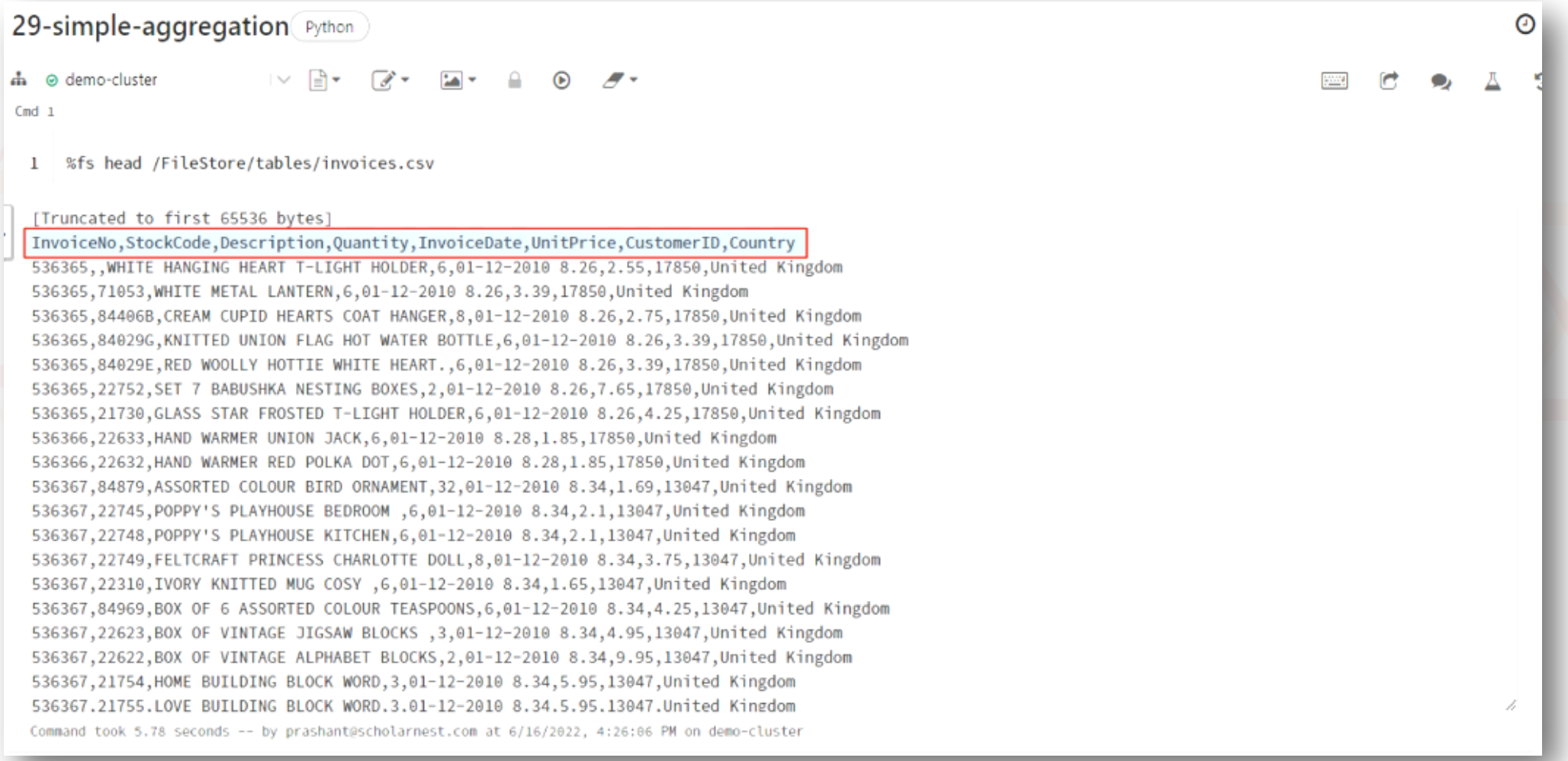


Once the data file is uploaded, you can create a new notebook.

(Reference: 29-simple-aggregation)



You can check the data file once as shown below.
This data file contains the list of invoice line items. We also have a header row consisting of invoice number, stock code, and stock item description. We also have other things like Quantity, Date, UnitPrice, CustomerID, and Country.



Invoice numbers are repeating, so all these items belong to the same invoice.

29-simple-aggregation Python

demo-cluster

Cmd 1

```
1  %fs head /FileStore/tables/invoices.csv
```

```
[Truncated to first 65536 bytes]
InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country
536365,,WHITE HANGING HEART T-LIGHT HOLDER,6,01-12-2010 8.26,2.55,17850,United Kingdom
536365,71053,WHITE METAL LANTERN,6,01-12-2010 8.26,3.39,17850,United Kingdom
536365,84406B,CREAM CUPID HEARTS COAT HANGER,8,01-12-2010 8.26,2.75,17850,United Kingdom
536365,84029G,KNITTED UNION FLAG HOT WATER BOTTLE,6,01-12-2010 8.26,3.39,17850,United Kingdom
536365,84029E,RED WOOLLY HOTTIE WHITE HEART.,6,01-12-2010 8.26,3.39,17850,United Kingdom
536365,22752,SET 7 BABUSHKA NESTING BOXES,2,01-12-2010 8.26,7.65,17850,United Kingdom
536365,21730,GLASS STAR FROSTED T-LIGHT HOLDER,6,01-12-2010 8.26,4.25,17850,United Kingdom
536366,22633,HAND WARMER UNION JACK,6,01-12-2010 8.28,1.85,17850,United Kingdom
536366,22632,HAND WARMER RED POLKA DOT,6,01-12-2010 8.28,1.85,17850,United Kingdom
536367,84879,ASSORTED COLOUR BIRD ORNAMENT,32,01-12-2010 8.34,1.69,13047,United Kingdom
536367,22745,POPPY'S PLAYHOUSE BEDROOM ,6,01-12-2010 8.34,2.1,13047,United Kingdom
536367,22748,POPPY'S PLAYHOUSE KITCHEN,6,01-12-2010 8.34,2.1,13047,United Kingdom
536367,22749,FELTCRAFT PRINCESS CHARLOTTE DOLL,8,01-12-2010 8.34,3.75,13047,United Kingdom
536367,22310,IVORY KNITTED MUG COSY ,6,01-12-2010 8.34,1.65,13047,United Kingdom
536367,84969,BOX OF 6 ASSORTED COLOUR TEASPOONS,6,01-12-2010 8.34,4.25,13047,United Kingdom
536367,22623,BOX OF VINTAGE JIGSAW BLOCKS ,3,01-12-2010 8.34,4.95,13047,United Kingdom
536367,22622,BOX OF VINTAGE ALPHABET BLOCKS,2,01-12-2010 8.34,9.95,13047,United Kingdom
536367,21754,HOME BUILDING BLOCK WORD,3,01-12-2010 8.34,5.95,13047,United Kingdom
536367.21755.LOVE BUILDING BLOCK WORD.3.01-12-2010 8.34.5.95.13047.United Kingdom
```

Command took 5.78 seconds -- by prashant@scholarnest.com at 6/16/2022, 4:26:06 PM on demo-cluster

We have created a Dataframe for the data file. I am reading the data file to a DataFrame and displaying it here. I also imported all spark functions here because aggregation is all about using aggregate and windowing functions.

```
1  from pyspark.sql import functions as sf
2
3  invoices_df = spark.read \
4                  .format("csv") \
5                  .option("header", "true") \
6                  .option("inferSchema", "true") \
7                  .load("/FileStore/tables/invoices.csv")
8
9  display(invoices_df)
```

▸ (3) Spark Jobs

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536365	null	WHITE HANGING HEART T-LIGHT HOLDER	6	01-12-2010 8.26	2.55	17850	United Kingdom
2	536365	71053	WHITE METAL LANTERN	6	01-12-2010 8.26	3.39	17850	United Kingdom
3	536365	844068	CREAM CUPID HEARTS COAT HANGER	8	01-12-2010 8.26	2.75	17850	United Kingdom
4	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	01-12-2010 8.26	3.39	17850	United Kingdom
5	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	01-12-2010 8.26	3.39	17850	United Kingdom
6	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	01-12-2010 8.26	7.65	17850	United Kingdom
7	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	01-12-2010 8.26	4.25	17850	United Kingdom

Truncated results, showing first 1000 rows.

Click to re-execute with maximum result limits.

Command took 18.05 seconds -- by prashant@scholarnest.com at 6/16/2022, 4:31:02 PM on demo-cluster

The simplest form of aggregation is to summarize the complete Data Frame, and it is going to give you a single row in the result.
For example, you can count the number of records in this Dataframe, and it will return you a single row with the count of records as shown below.
We start with the Dataframe, use the select() method, and apply the count(*) function.

```
Cmd 3

1  invoices_df.select(sf.count("*") \
2                     ).show()

▸ (2) Spark Jobs

+--------+
|count(1)|
+--------+
|  541909|
+--------+

Command took 5.40 seconds -- by prashant@scholarnest.com at 6/16/2022, 4:35:17 PM on demo-cluster
```


But remember! this code is different than invoice_df.count().
You can see below that, both the codes are giving me the same result. However, the first one is using an aggregation function inside the select method. The select method will return a Dataframe. So the first approach is a transformation approach. The second one is using a Dataframe count() action. It simply returns the count value. So the second approach is the Dataframe action approach.

```
1  invoices_df.select(sf.count("*") \
2                     ).show()

▸ (2) Spark Jobs

+--------+
|count(1)|
+--------+
|  541909|
+--------+

Command took 5.40 seconds -- by prashant@scholarnest.com at 6/16/2022, 4:35:17 PM on demo-cluster

Cmd 4

1  invoices_df.count()

▸ (2) Spark Jobs

Out[4]: 541909

Command took 2.79 seconds -- by prashant@scholarnest.com at 6/16/2022, 4:36:07 PM on demo-cluster
```


Now the next point of discussion is the difference between count(*) and count(1).
The difference between them is - Nothing.
In Spark, they are the same. Spark automatically converts count(*) into count(1).
And you can see it here in the screenshot below. Do you see the column name of the result?
It says count(1). I ran count(*), but it says count(1). Because Spark converts the count(*) into count(1).
So we do not have any difference in count(*) and count(1).

However, counting a column field has a difference.
The count(*) will count all the rows even if you have null values in all the columns. However, counting a field such as StockCode will not count the null values. The count of StockCode is one less than the count(*) because we have one null StockCode.

Cmd 3

```
1  invoices_df.select(sf.count("*"),
2                     sf.count("StockCode") \
3                        ).show()
```

▸ (2) Spark Jobs

```
+--------+----------------+
|count(1)|count(StockCode)|
+--------+----------------+
|  541909|          541908|
+--------+----------------+
```

Command took 5.37 seconds -- by prashant@scholarnest.com at 6/16/2022, 4:47:08 PM on demo-cluster

You can also give an alias to the columns using the column API, and you can also add as many columns as you want. I have added columns for sum, average, and a *countDistinct* function to count the unique values of an invoice number. I also gave aliases to all of these columns. And this example is the first type of aggregation - the simple aggregation. A simple aggregation will summarize the whole Dataframe and get a single row.

Cmd 3

```
1  invoices_df.select(sf.count("*").alias("count_all"),
2                     sf.count("StockCode").alias("count_stock_code"),
3                     sf.sum("Quantity").alias("TotalQuantity"),
4                     sf.avg("UnitPrice").alias("AvgPrice"),
5                     sf.countDistinct("InvoiceNo").alias("CountDistinct") \
6                        ).show()
```

▸ (3) Spark Jobs

```
+---------+----------------+-------------+-----------------+-------------+
|count_all|count_stock_code|TotalQuantity|         AvgPrice|CountDistinct|
+---------+----------------+-------------+-----------------+-------------+
|   541909|          541908|      5176450|4.611113626088477|        25900|
+---------+----------------+-------------+-----------------+-------------+
```

Command took 7.17 seconds -- by prashant@scholarnest.com at 6/16/2022, 4:53:02 PM on demo-cluster

These aggregates are nothing but functions. You can use them in column object expressions and you can also use them in SQL like string expressions as shown below. I used the selectExpr(), because I do not want to use the expr() function everywhere.
All these function calls are expressions. So I must use the expr() function to evaluate them.
Instead of using expr() function for each expression, I used the selectExpr().

Cmd 4

```
1  invoices_df.selectExpr(
2          "count(*) as count_all",
3          "count(StockCode) as count_stock_code",
4          "sum(Quantity) as TotalQuantity",
5          "avg(UnitPrice) as AvgPrice"
6      ).show()
```

▸ (2) Spark Jobs

```
+---------+----------------+-------------+------------------+
|count_all|count_stock_code|TotalQuantity|         AvgPrice|
+---------+----------------+-------------+------------------+
|   541909|          541908|      5176450|4.611113626089747|
+---------+----------------+-------------+------------------+
```

Command took 3.56 seconds -- by prashant@scholarnest.com at 6/16/2022, 4:56:42 PM on demo-cluster

I could not use the countDistinct() here in the SQL approach. In the earlier example, I also calculated the countDistinct(). But I am not using it here, because countDistinct() is a Dataframe function and is not available as a built-in SQL function.
So this approach of using SQL-like expressions cannot use Dataframe functions.

Cmd 4

```
1  invoices_df.selectExpr(
2          "count(*) as count_all",
3          "count(StockCode) as count_stock_code",
4          "sum(Quantity) as TotalQuantity",
5          "avg(UnitPrice) as AvgPrice"
6      ).show()
```

▸ (2) Spark Jobs

```
+---------+----------------+-------------+------------------+
|count_all|count_stock_code|TotalQuantity|         AvgPrice|
+---------+----------------+-------------+------------------+
|   541909|          541908|      5176450|4.611113626089747|
+---------+----------------+-------------+------------------+
```

Command took 3.56 seconds -- by prashant@scholarnest.com at 6/16/2022, 4:56:42 PM on demo-cluster

The simple aggregation will always give you a one-line summary. However, you may want a more detailed summary. For example, I am looking to summarize my Dataframe as shown below.
We still want to compute two aggregates. The first one is the sum(Quantity). And the second one is the sum(Quantity * UnitPrice). However, instead of summarizing them for the whole Dataframe, we want to group them into the country and invoice number. And then apply these two aggregates. And that is the second type of aggregation: The Grouping aggregation.

Country	InvoiceNo	sum(Quantity)	sum(Quantity * UnitPrice)
United Kingdom	536446	329	440.89
United Kingdom	536508	216	155.52
Australia	543376	21	67.95
Australia	546135	90	210.9
Australia	545065	217	412.55
Austria	547493	929	1542.08
Austria	543027	145	153.76
Austria	559707	100	484.86
Austria	545570	53	166.04
Bahrain	553900	20	89.0
Bahrain	552449	240	459.4
Belgium	547061	215	226.3
Belgium	555928	20	99.45

If you have a good background in SQL, you know to achieve the requirement given in the previous slide you have to use the GroupBy SQL expression.
And you can quickly write a SQL expression for this aggregation, as shown below.
Surprisingly, if you can create a SQL expression, you can run it in Spark. You can use your Dataframe to create a temp view. And then, you can use Spark SQL.

```
SELECT Country,
        InvoiceNo,
        sum(Quantity) as TotalQuantity,
        round(sum(Quantity*UnitPrice), 2) as InvoiceValue
FROM sales
GROUP BY Country, InvoiceNo
```


Here I have created a temp view in the first line of code. Now I can run a SQL on this temp view. Next, we want to run a SQL expression and get the results in a Dataframe. You can use the spark.sql() method to get the desired result as shown below.

Cmd 5

```
1  invoices_df.createOrReplaceTempView("sales")
2
3  summary_sql = spark.sql("""
4              SELECT Country, InvoiceNo,
5                      sum(Quantity) as TotalQuantity,
6                      round(sum(Quantity*UnitPrice), 2) as InvoiceValue
7               FROM sales
8               GROUP BY Country, InvoiceNo""")
9
10 display(summary_sql)
```

▸ (2) Spark Jobs

	Country	InvoiceNo	TotalQuantity	InvoiceValue
1	United Kingdom	536446	329	440.89
2	United Kingdom	536508	216	155.52
3	United Kingdom	537018	-3	0
4	United Kingdom	537401	-24	0
5	United Kingdom	537811	74	268.86
6	United Kingdom	C537824	-2	-14.9
7	United Kingdom	538895	370	247.38

Truncated results, showing first 1000 rows.

Click to re-execute with maximum result limits.

The SQL expressions will return a DataFrame, and you can use it in whatever way you want.
So you can use your SQL skills and write grouping aggregations. Spark SQL is an excellent and easy method to run your grouping aggregations. There is no harm in taking the SQL approach for complex grouping aggregations.

Cmd 5

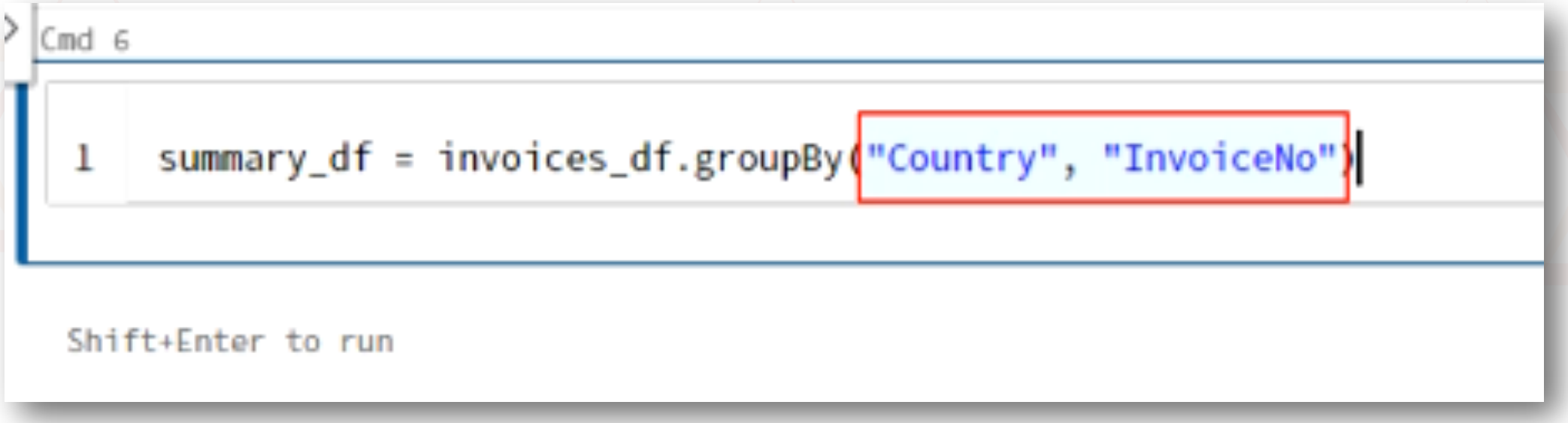
```
1  invoices_df.createOrReplaceTempView("sales")
2
3  summary_sql = spark.sql("""
4              SELECT Country, InvoiceNo,
5                     sum(Quantity) as TotalQuantity,
6                     round(sum(Quantity*UnitPrice), 2) as InvoiceValue
7              FROM sales
8              GROUP BY Country, InvoiceNo""")
9
10 display(summary_sql)
```

▸ (2) Spark Jobs

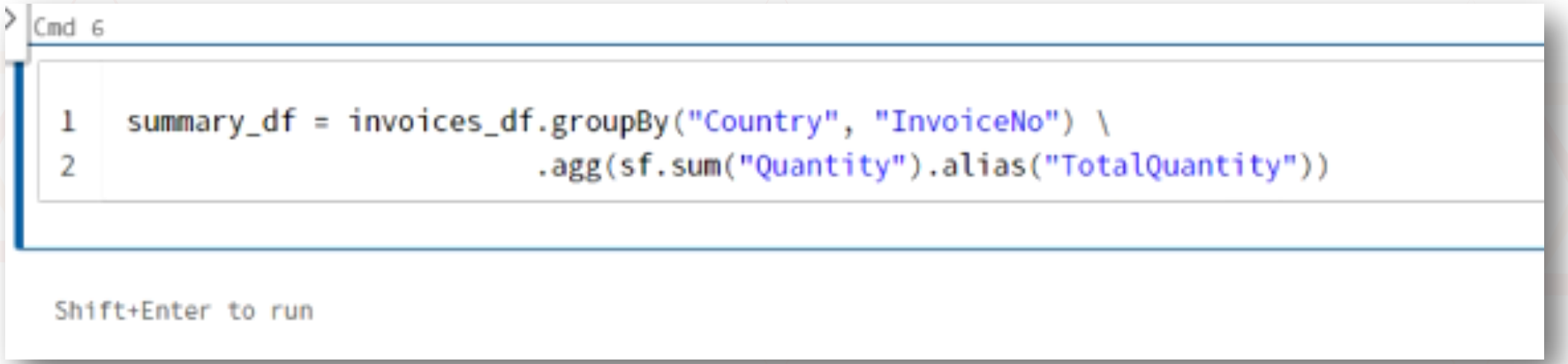
	Country	InvoiceNo	TotalQuantity	InvoiceValue
1	United Kingdom	536446	329	440.89
2	United Kingdom	536508	216	155.52
3	United Kingdom	537018	-3	0
4	United Kingdom	537401	-24	0
5	United Kingdom	537811	74	268.86
6	United Kingdom	C537824	-2	-14.9
7	United Kingdom	538895	370	247.38

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

However, you can do the same using Dataframe expressions also, as shown below.
We start with the DataFrame and use the groupBy() to group the records. Same as the SQL, I am grouping it on the country and the Invoice number.



Then we have to apply the sum() functions. So, you can use the agg() transformation. The agg()transformation is specifically designed to take a list of aggregation functions.
So my first aggregation is the sum(Quantity) and I have also given an alias to the sum function.



The next aggregation is a little complex.

*round(sum(Quantity*UnitPrice),2) as InvoiceValue*

We have a sum() function, then we have a round() function, and finally, an alias.

But the "Quantity * UnitPrice" is not a single column name. It is an expression. So, we wrap it around the expr(), and we will get the sum() of the given expression. Then we can wrap the entire thing inside the round, and it will round the value to two digits. And finally, you can give an alias to the result of the round function.

This is how we create column expressions.

Cmd 6

```
1   summary_df = invoices_df.groupBy("Country", "InvoiceNo") \
2                           .agg(sf.sum("Quantity").alias("TotalQuantity"),
3                                sf.round(sf.sum(sf.expr("Quantity * UnitPrice")), 2).alias("InvoiceValue"))
```

Command took 0.12 seconds -- by prashant@scholarnest.com at 6/16/2022, 5:15:27 PM on demo-cluster

You can use the SQL approach if you feel more comfortable with SQL-like string expressions. I have taken the same code written in the previous cell and renamed the *summary_df* to *summary_df1*. Then I have removed the second expression which was using Dataframe column expression in the previous cell, and I have used a SQL-like expression here.

Cmd 7

```
1   summary_df_1 = invoices_df.groupBy("Country", "InvoiceNo") \
2                             .agg(sf.sum("Quantity").alias("TotalQuantity"),
3                                  sf.expr("round(sum(Quantity*UnitPrice), 2) as InvoiceValue"))
```

Command took 0.09 seconds -- by prashant@scholarnest.com at 6/16/2022, 5:23:07 PM on demo-cluster

So, both approaches given below are the same.

Cmd 6

```
summary_df = invoices_df.groupBy("Country", "InvoiceNo") \
                        .agg(sf.sum("Quantity").alias("TotalQuantity"),
                             sf.round(sf.sum(sf.expr("Quantity * UnitPrice")), 2).alias("InvoiceValue"))
```

Command took 0.12 seconds -- by prashant@scholarnest.com at 6/16/2022, 5:15:27 PM on demo-cluster

Cmd 7

```
summary_df_1 = invoices_df.groupBy("Country", "InvoiceNo") \
                          .agg(sf.sum("Quantity").alias("TotalQuantity"),
                               sf.expr("round(sum(Quantity*UnitPrice), 2) as InvoiceValue"))
```

Command took 0.09 seconds -- by prashant@scholarnest.com at 6/16/2022, 5:23:07 PM on demo-cluster

Assignment:

I am considering the same invoice data file.

Can you create a Dataframe that aggregates to show the following?

I have five columns in the result. The first two are grouping columns, and the remaining three are aggregations. So, we want to group it by country and the week number of the year.

You already have a country column in the input Dataframe. However, you don't have the week number. But you have the Invoice Date, and you can transform the data to a week number. We have a built-in function for this.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
Spain	49	1	67	174.72
Germany	48	11	1795	3309.75
Lithuania	48	3	622	1598.06
Germany	49	12	1852	4521.39
Bahrain	51	1	54	205.74

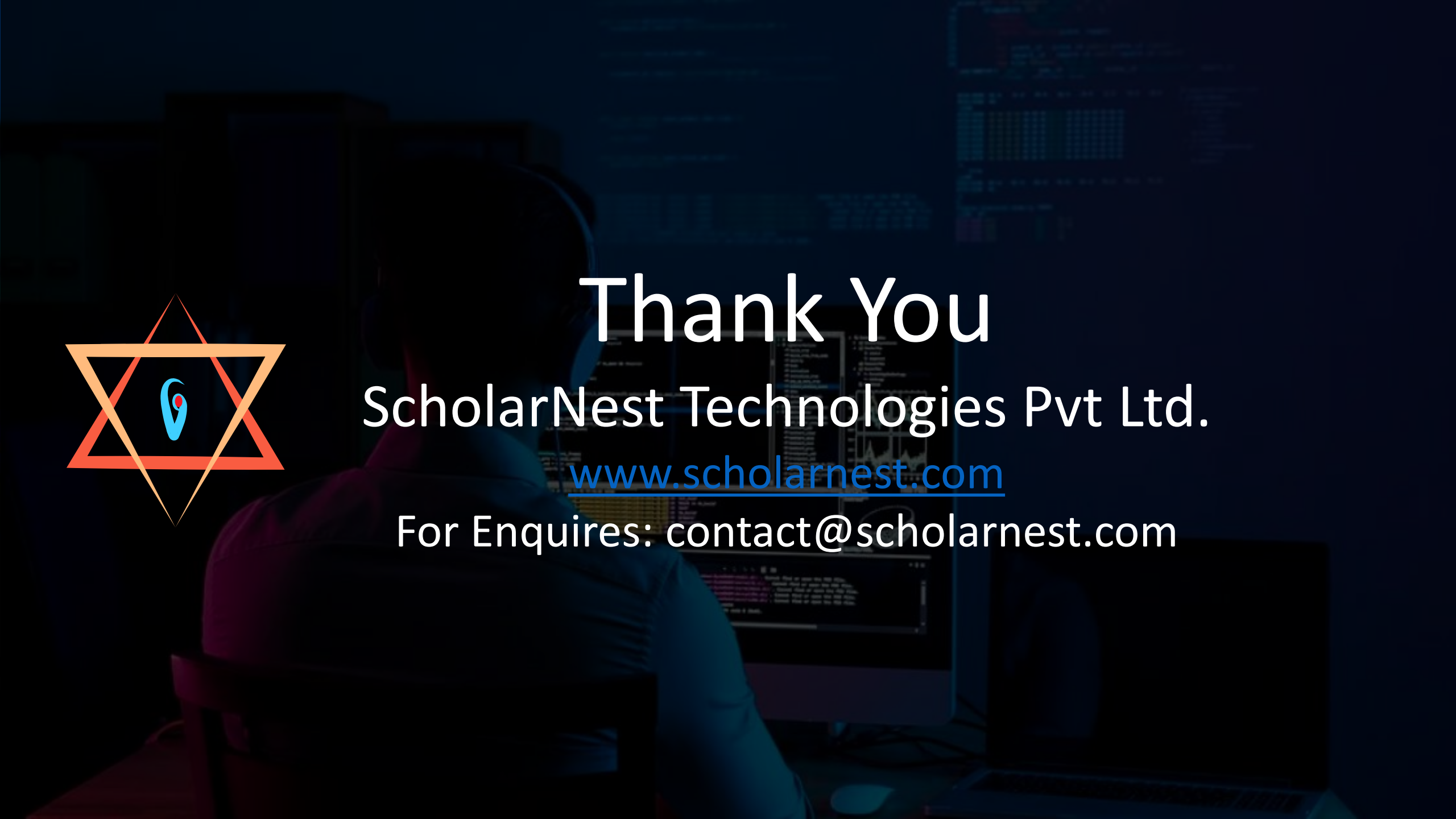
Assignment:

Now let's come to the aggregates. So the first aggregate is the number of unique invoices. You already learned the countDistinct(). You can get this quickly.

The total quantity is the sum("Quantity"), and the total value is the sum(Quantity * Unit Price).

Take the challenge and do it yourself.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
Spain	49	1	67	174.72
Germany	48	11	1795	3309.75
Lithuania	48	3	622	1598.06
Germany	49	12	1852	4521.39
Bahrain	51	1	54	205.74



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

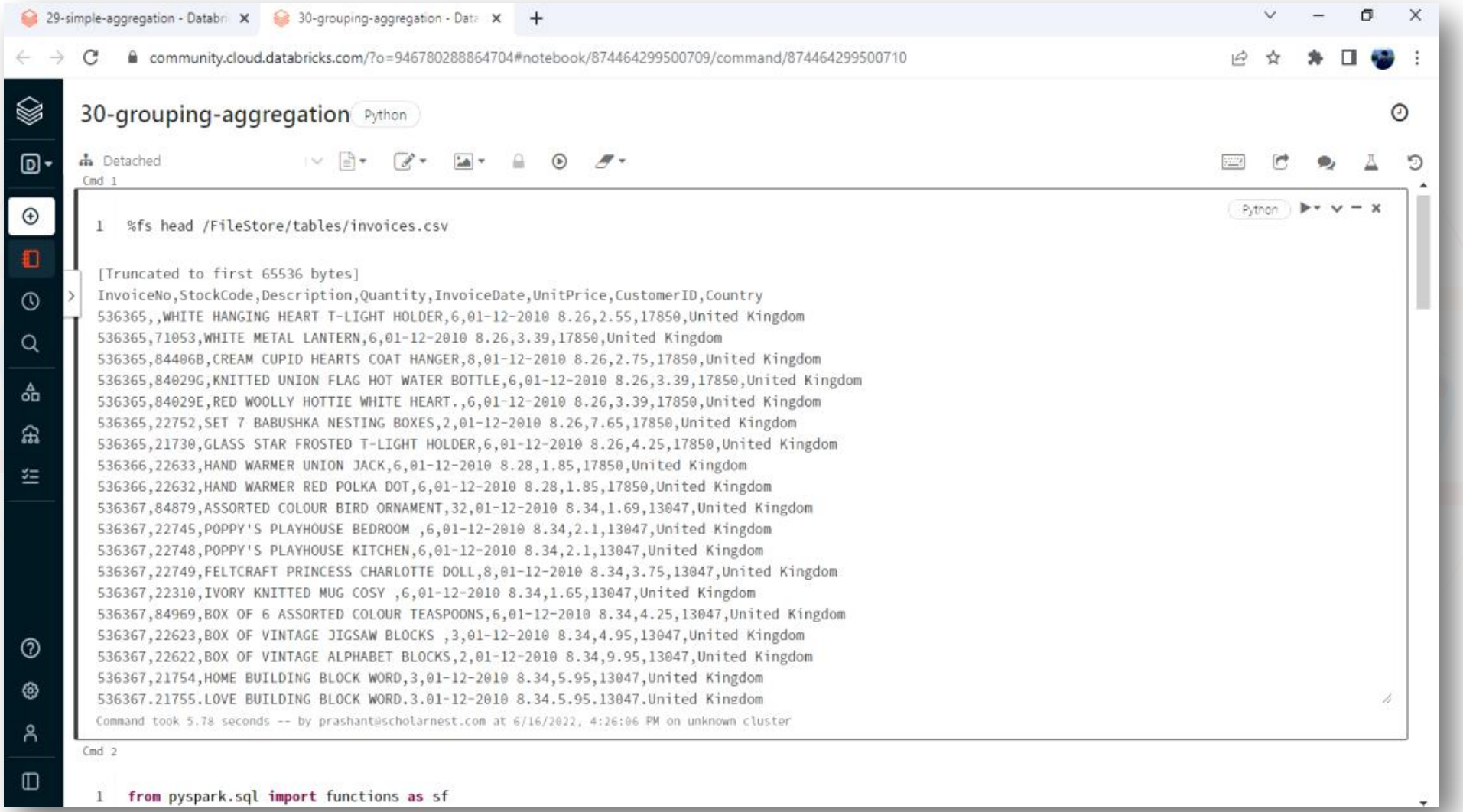
Databricks Spark Certification and Beyond

Module:
Aggregation

Lecture:
Grouping
Aggregation

Grouping Aggregations

Go to the Databricks workspace and clone the previous lecture's notebook (29-simple-aggregation). Also, give a new name to your clone. **(Reference: 30-grouping-aggregation)**



Remove everything from the clone and only keep the invoice_df.
Also, run it once, so you know it is working.

30-grouping-aggregation Python

demo-cluster

Cmd 1

```
1  from pyspark.sql import functions as sf
2
3  invoices_df = spark.read \
4                  .format("csv") \
5                  .option("header", "true") \
6                  .option("inferSchema", "true") \
7                  .load("/FileStore/tables/invoices.csv")
8
9  display(invoices_df)
```

(3) Spark Jobs

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536365	null	WHITE HANGING HEART T-LIGHT HOLDER	6	01-12-2010 8.26	2.55	17850	United Kingdom
2	536365	71053	WHITE METAL LANTERN	6	01-12-2010 8.26	3.39	17850	United Kingdom
3	536365	844068	CREAM CUPID HEARTS COAT HANGER	8	01-12-2010 8.26	2.75	17850	United Kingdom
4	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	01-12-2010 8.26	3.39	17850	United Kingdom
5	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	01-12-2010 8.26	3.39	17850	United Kingdom
6	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	01-12-2010 8.26	7.65	17850	United Kingdom
7	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	01-12-2010 8.26	4.25	17850	United Kingdom

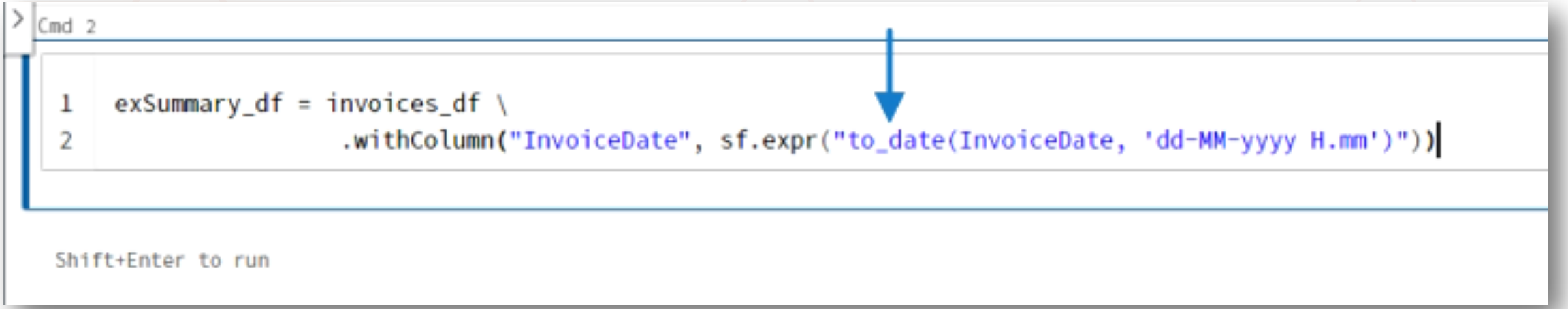
Truncated results, showing first 1000 rows.

Click to re-execute with maximum result limits.

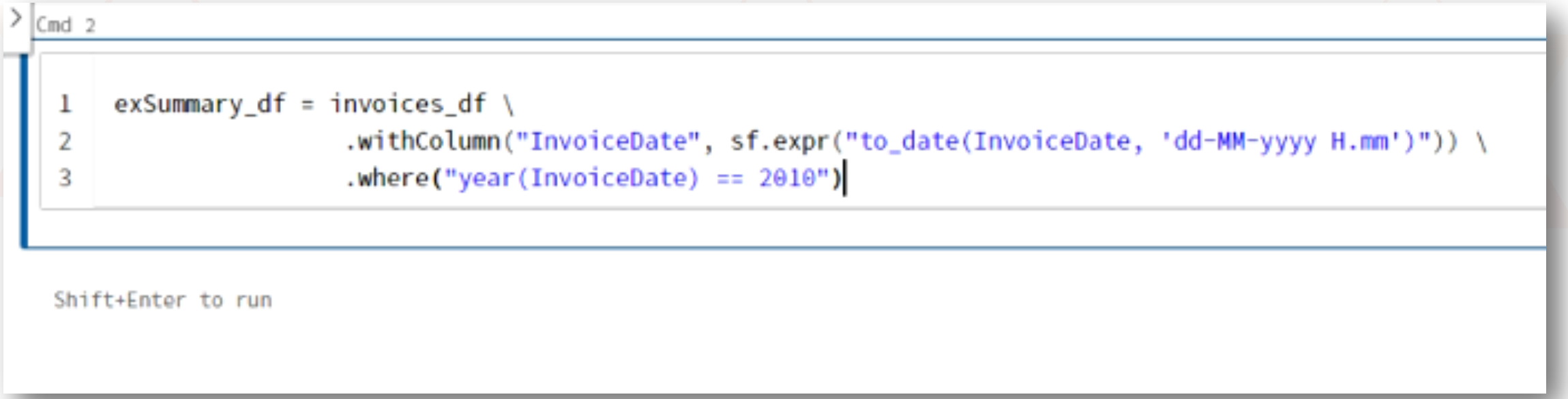
Now we want to compute the grouping aggregates as per the table shown here.
We want to group the initial raw Dataframe into two columns: Country and Week Number.
We already have a country field in the source Dataframe. However, we do not have the week number field. But we can extract the week number from the *InvoiceDate*. We have a built-in function called *weekofyear()*, which will give us the week number. However, the *weekofyear()* expects a Date field, and the invoice date is a string in our data frame. So, the first thing is to convert the Invoice Date to a proper date field.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
Spain	49	1	67	174.72
Germany	48	11	1795	3309.75
Lithuania	48	3	622	1598.06
Germany	49	12	1852	4521.39
Bahrain	51	1	54	205.74

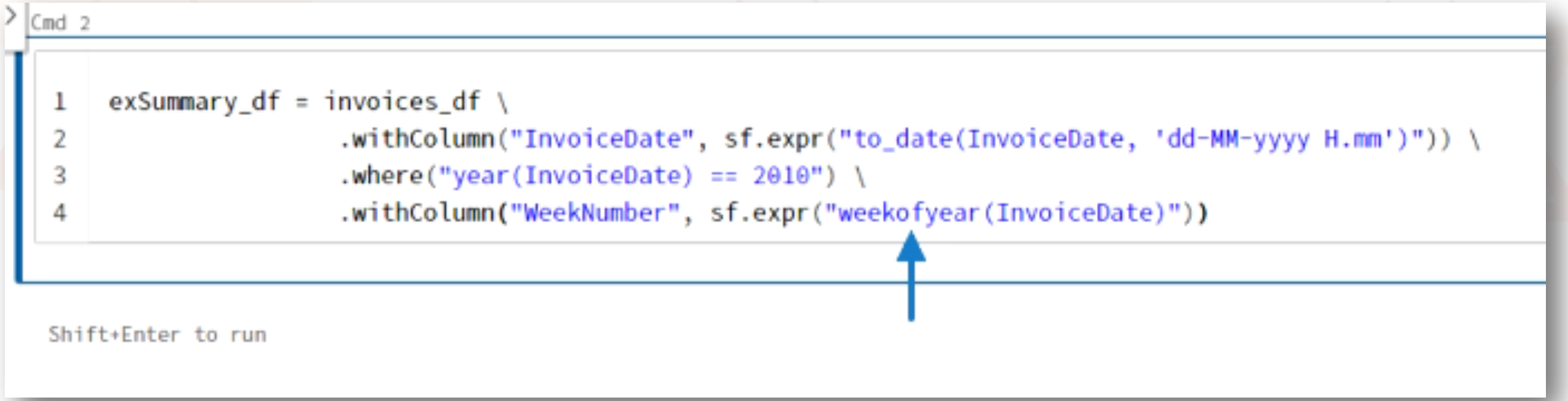
We start with the invoice data frame, and we will be using the *withColumn()* transformation to convert the *InvoiceDate*. All you need to do is to apply the *to_date()* function.
Now the *InvoiceDate* will change to a proper date type, and we can easily get the week number.



The week number doesn't make any sense without the year. I mean, when we say week number 12, we must say week number 12 of the year 2010. Otherwise, we do not know what we are talking about. Is it week 12 of 2010 or week 12 of 2011? So let me limit the data to the year 2010 only as shown below.



Now, you can again use the *withColumn()* to add a new column for the week number.
Extracting the week number is as simple as using the *weekofyear()* function.
I keep on using the SQL-like expressions and evaluate them with the expr() function.
But you can try creating the same expression using the col() function, Dataframe functions, and column API.



Now we are ready to do a groupBy() for the country and then the week number, as given below.

Cmd 2

```
1  exSummary_df = invoices_df \
2                  .withColumn("InvoiceDate", sf.expr("to_date(InvoiceDate, 'dd-MM-yyyy H.mm')")) \
3                  .where("year(InvoiceDate) == 2010") \
4                  .withColumn("WeekNumber", sf.expr("weekofyear(InvoiceDate)")) \
5                  .groupBy("Country", "WeekNumber")
```

Shift+Enter to run

Now we are ready for the aggregation.
The first aggregation is as simple as using the countDistinct() for InvoiceNo, and we have also given an alias to it.
You can build your expression here in the agg() method. Or, you can define this aggregation as a Python variable and use the variable in the agg() method.

Cmd 2

```
1  exSummary_df = invoices_df \
2                  .withColumn("InvoiceDate", sf.expr("to_date(InvoiceDate, 'dd-MM-yyyy H.mm')")) \
3                  .where("year(InvoiceDate) == 2010") \
4                  .withColumn("WeekNumber", sf.expr("weekofyear(InvoiceDate)")) \
5                  .groupBy("Country", "WeekNumber") \
6                  .agg(sf.countDistinct("InvoiceNo").alias("NumInvoices"))
```

Shift+Enter to run

Here on the top of the cell I have defined a variable, *NumInvoices*. . After defining the variable, I have initialized it with the expression. And now I can use this variable inside the *agg()* function as shown below.

Cmd 2

```
1  NumInvoices = sf.countDistinct("InvoiceNo").alias("NumInvoices")
2
3  exSummary_df = invoices_df \
4                  .withColumn("InvoiceDate", sf.expr("to_date(InvoiceDate, 'dd-MM-yyyy H.mm')")) \
5                  .where("year(InvoiceDate) == 2010") \
6                  .withColumn("WeekNumber", sf.expr("weekofyear(InvoiceDate)")) \
7                  .groupBy("Country", "WeekNumber") \
8                  .agg(NumInvoices)
```

Shift+Enter to run

I have added a couple of more variables for *TotalQuantity* and *InvoiceValue*, and also added those variables in the *agg()* function. And you can see that the Dataframe is displayed below.

```
1  NumInvoices = sf.countDistinct("InvoiceNo").alias("NumInvoices")
2  TotalQuantity = sf.sum("Quantity").alias("TotalQuantity")
3  InvoiceValue = sf.expr("round(sum(Quantity * UnitPrice), 2) as InvoiceValue")
4
5  exSummary_df = invoices_df \
6                  .withColumn("InvoiceDate", sf.expr("to_date(InvoiceDate, 'dd-MM-yyyy H.mm')")) \
7                  .where("year(InvoiceDate) == 2010") \
8                  .withColumn("WeekNumber", sf.expr("weekofyear(InvoiceDate)")) \
9                  .groupBy("Country", "WeekNumber") \
10                 .agg(NumInvoices, TotalQuantity, InvoiceValue)
11
12 display(exSummary_df)
```

▸ (3) Spark Jobs

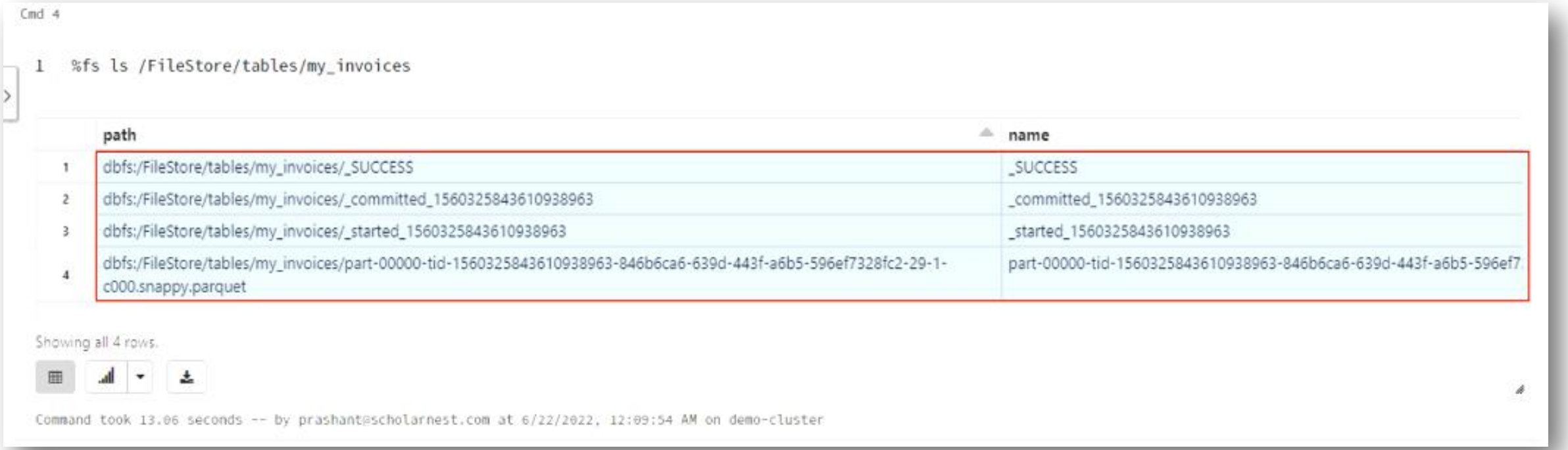
	Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
1	Spain	49	1	67	174.72
2	Germany	48	11	1795	3309.75
3	Lithuania	48	3	622	1598.06
4	Germany	49	12	1852	4521.39
5	Bahrain	51	1	54	205.74
6	Iceland	49	1	319	711.79
7	EIRE	51	5	95	276.84

Showing all 51 rows.

Now I want to save the data frame as a parquet file and use it for my next example.
We have already learned to save Dataframe as tables. But this time, I want to save it as a file.
There isn't much difference between saving as a file vs. saving as a table. When you save a Dataframe as a table, you will also create table metadata so other users can access it from Spark SQL.
But saving it as a file will not create metadata, and you cannot access it from Spark SQL. However, you can still load the file in a Dataframe as we do with the other data files.

I am using *dataframe.write* to get the Dataframe writer.
Then I am setting the format as parquet.
The mode is the overwrite mode, so I can run it as many times as I want.
And finally, I am telling the directory name for the output file.
If you run the cell, you will see that the code worked.

Now if you check the output directory, you can see that there are four files created by the Dataframe write operation.



Cmd 4

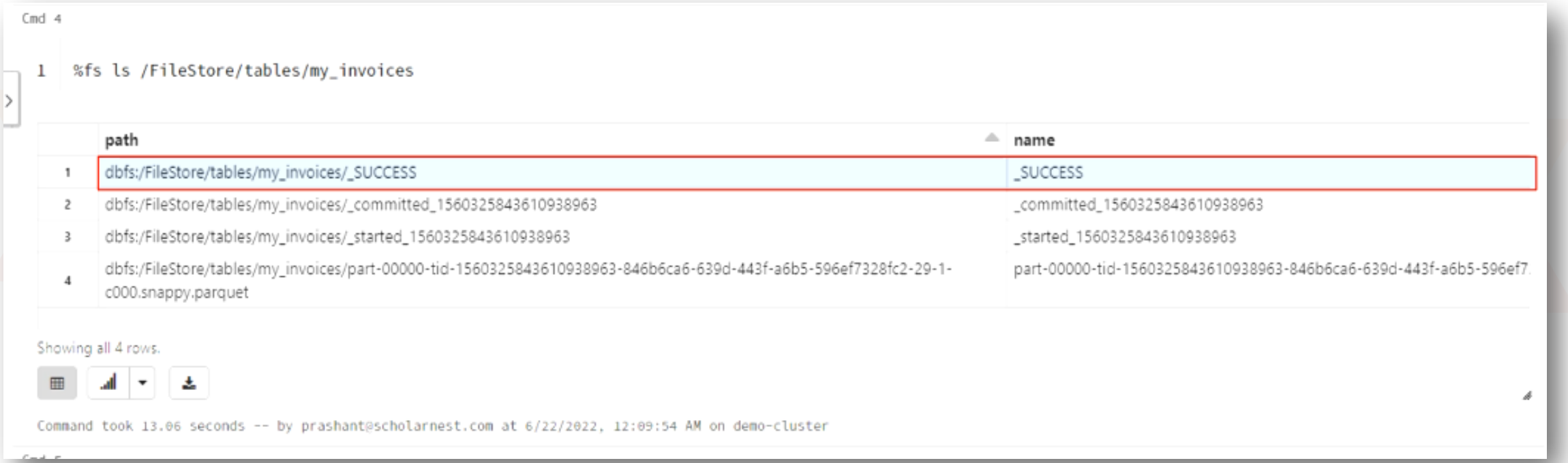
```
1  %fs ls /FileStore/tables/my_invoices
```

	path	name
1	dbfs:/FileStore/tables/my_invoices/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/my_invoices/_committed_1560325843610938963	_committed_1560325843610938963
3	dbfs:/FileStore/tables/my_invoices/_started_1560325843610938963	_started_1560325843610938963
4	dbfs:/FileStore/tables/my_invoices/part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7328fc2-29-1-c000.snappy.parquet	part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7

Showing all 4 rows.

Command took 13.06 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:09:54 AM on demo-cluster

The first one is the _SUCCESS file, indicating the write operation is successful.



Cmd 4

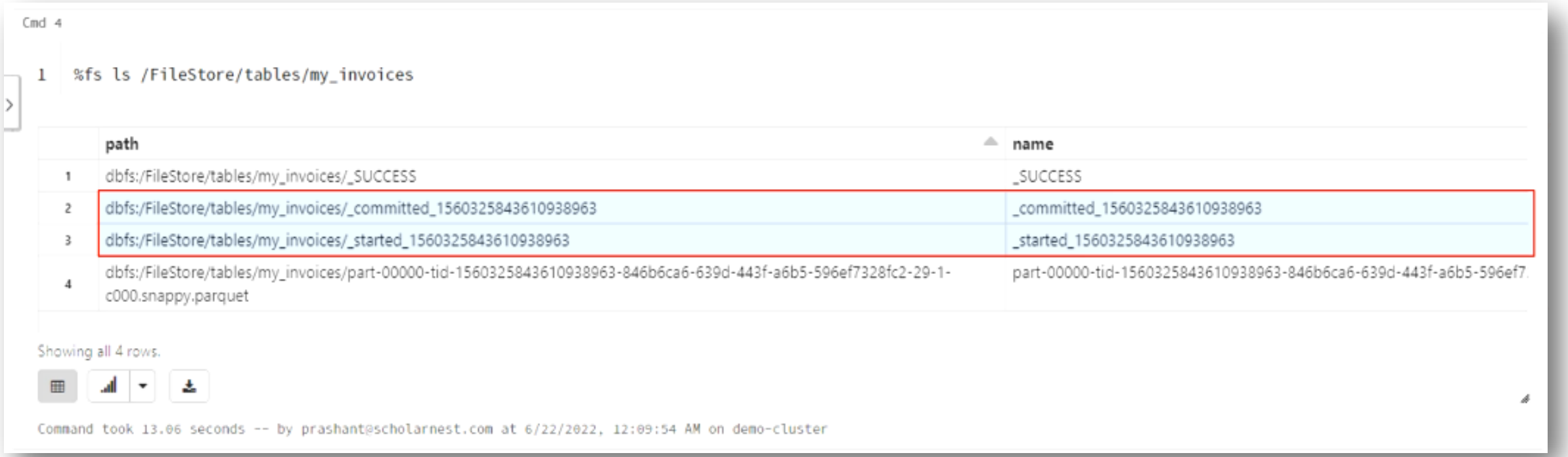
```
1  %fs ls /FileStore/tables/my_invoices
```

	path	name
1	dbfs:/FileStore/tables/my_invoices/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/my_invoices/_committed_1560325843610938963	_committed_1560325843610938963
3	dbfs:/FileStore/tables/my_invoices/_started_1560325843610938963	_started_1560325843610938963
4	dbfs:/FileStore/tables/my_invoices/part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7328fc2-29-1-c000.snappy.parquet	part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7

Showing all 4 rows.

Command took 13.06 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:09:54 AM on demo-cluster

We have two more files: started and committed. These two are the result of the overwrite mode. But we do not care about all these files. These are tiny files, and they are internal to Spark.



Cmd 4

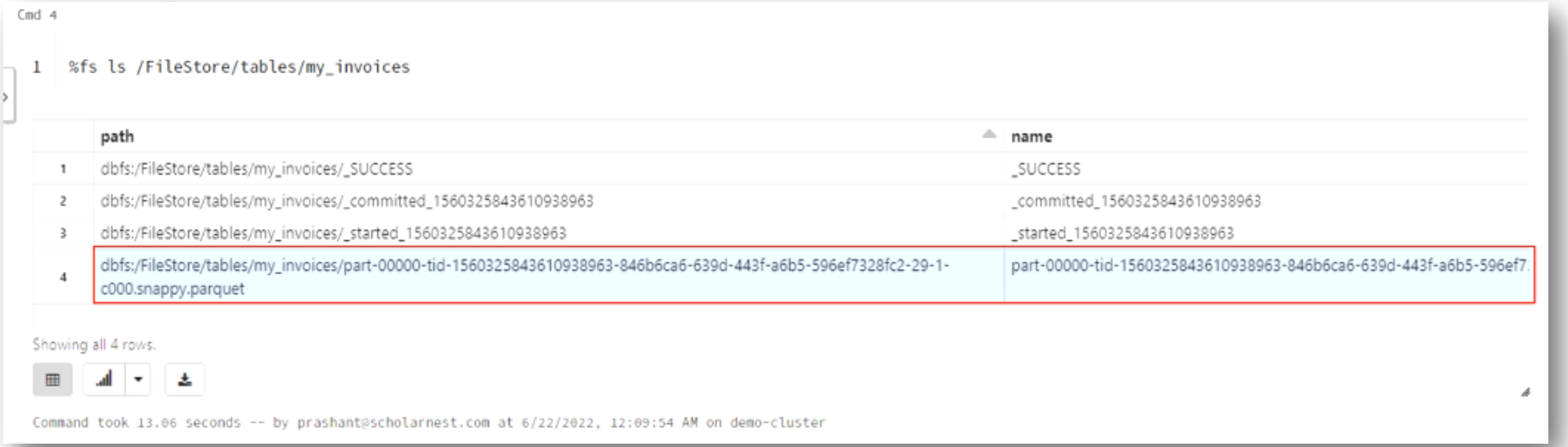
```
1  %fs ls /FileStore/tables/my_invoices
```

	path	name
1	dbfs:/FileStore/tables/my_invoices/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/my_invoices/_committed_1560325843610938963	_committed_1560325843610938963
3	dbfs:/FileStore/tables/my_invoices/_started_1560325843610938963	_started_1560325843610938963
4	dbfs:/FileStore/tables/my_invoices/part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7328fc2-29-1-c000.snappy.parquet	part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7

Showing all 4 rows.

Command took 13.06 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:09:54 AM on demo-cluster

All we care about is the data file. You can see one *snappy.parquet* file. So the write operation with parquet format creates a snappy compressed parquet data file.



Cmd 4

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

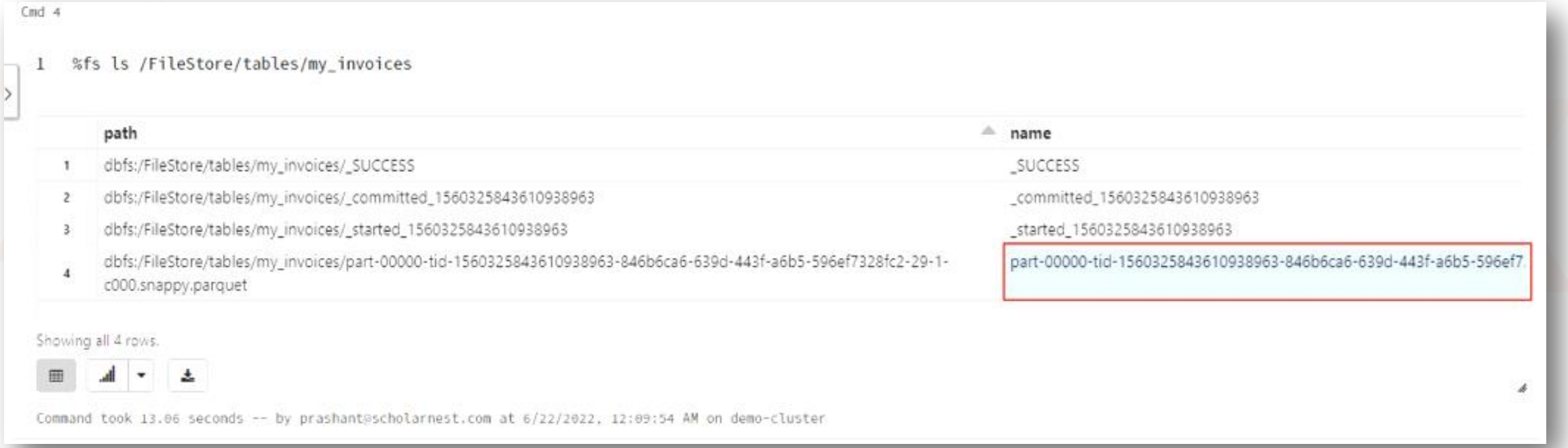
	path	name
1	dbfs:/FileStore/tables/my_invoices/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/my_invoices/_committed_1560325843610938963	_committed_1560325843610938963
3	dbfs:/FileStore/tables/my_invoices/_started_1560325843610938963	_started_1560325843610938963
4	dbfs:/FileStore/tables/my_invoices/part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7328fc2-29-1-c000.snappy.parquet	part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7.

Showing all 4 rows.

Command took 13.06 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:09:54 AM on demo-cluster

But how do we read such files?

I mean, we have been reading data files using the file location and the file name. But look at the file name highlighted below. It is automatically generated by the Spark write operation. How do I know the file name if I want to read this data file and create a data frame?



Cmd 4

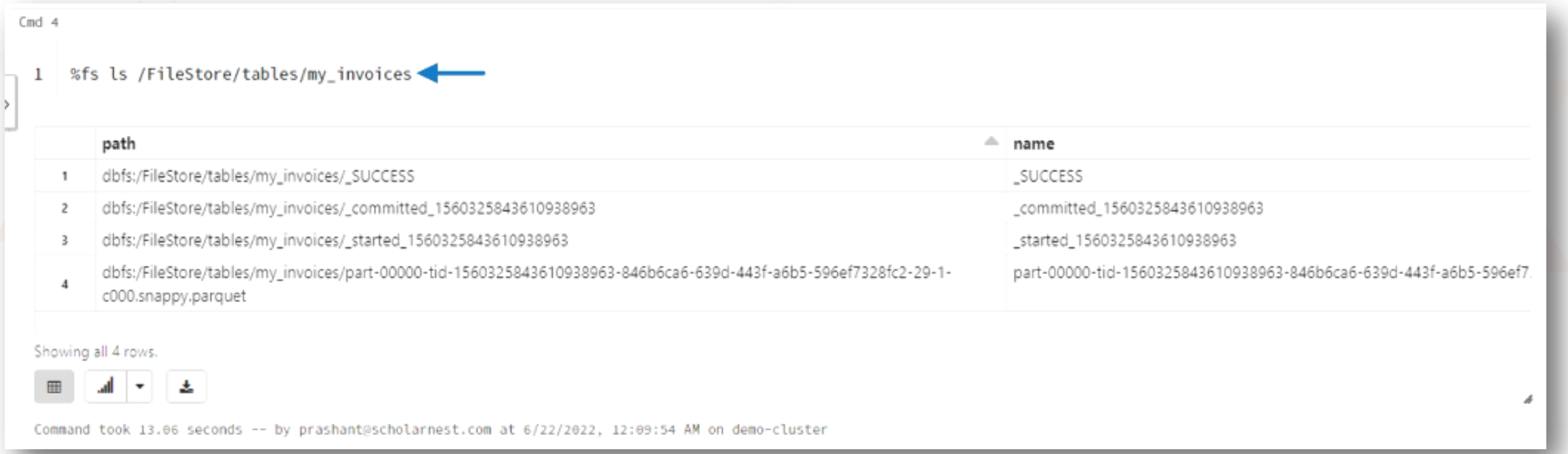
```
1  %fs ls /FileStore/tables/my_invoices
```

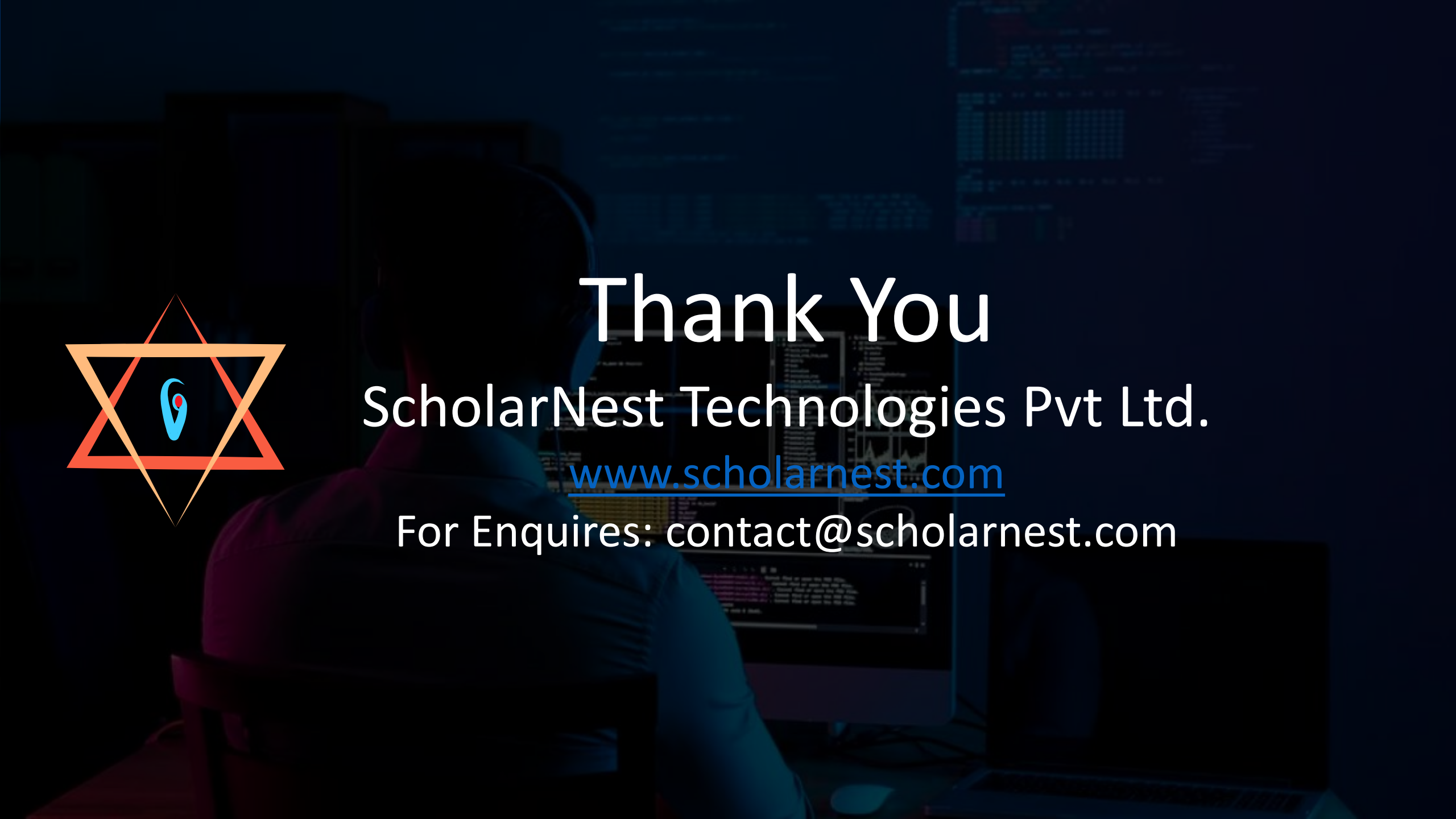
	path	name
1	dbfs:/FileStore/tables/my_invoices/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/my_invoices/_committed_1560325843610938963	_committed_1560325843610938963
3	dbfs:/FileStore/tables/my_invoices/_started_1560325843610938963	_started_1560325843610938963
4	dbfs:/FileStore/tables/my_invoices/part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7328fc2-29-1-c000.snappy.parquet	part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7

Showing all 4 rows.

Command took 13.06 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:09:54 AM on demo-cluster

You can create a Spark Dataframe using the directory location only.
And Spark Dataframe reader will read all data files in the directory.
But what about these _ SUCCESS, started and committed files?
Well! Spark Dataframe reader is smart enough to ignore those files.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

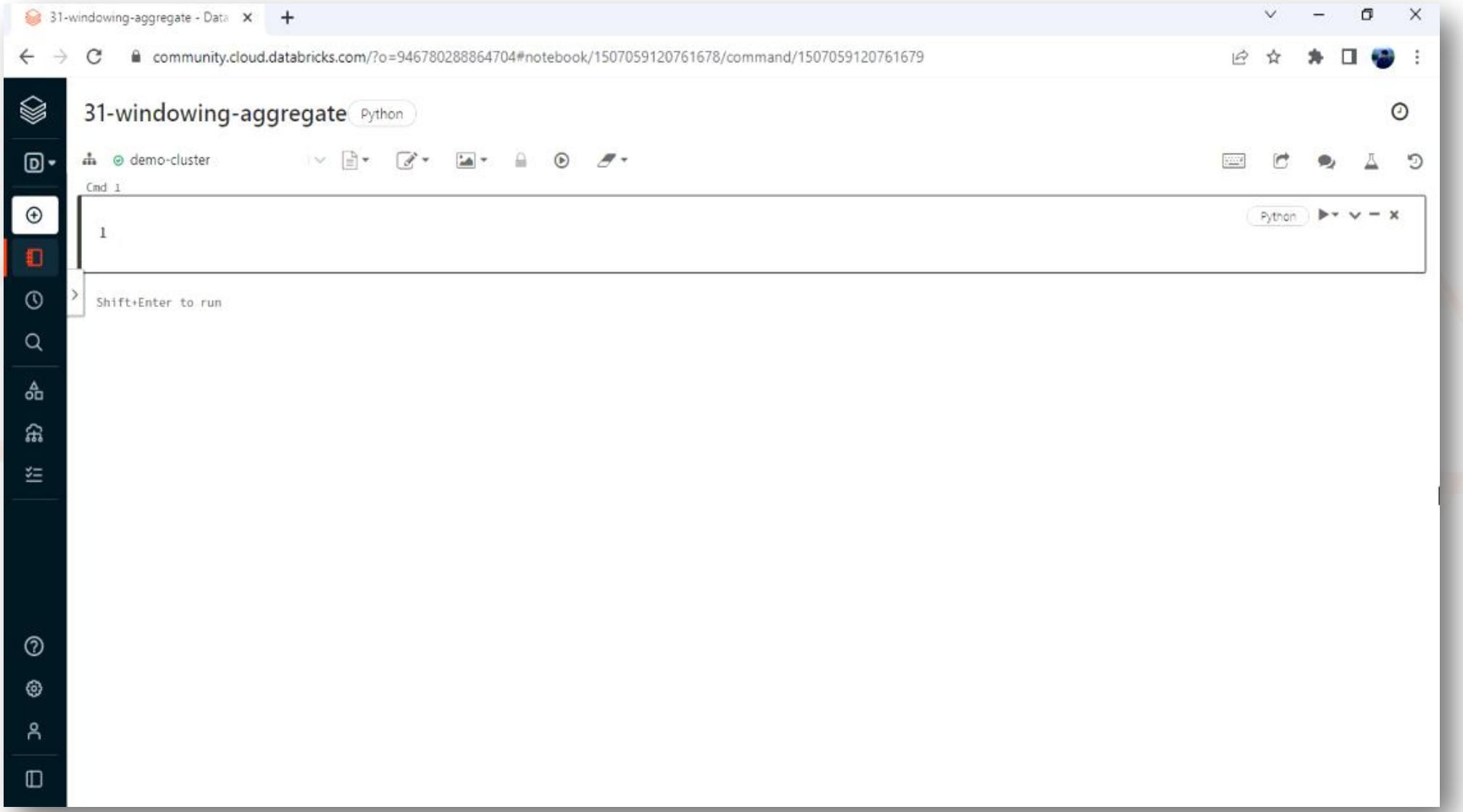
Databricks Spark Certification and Beyond

Module:
Aggregation

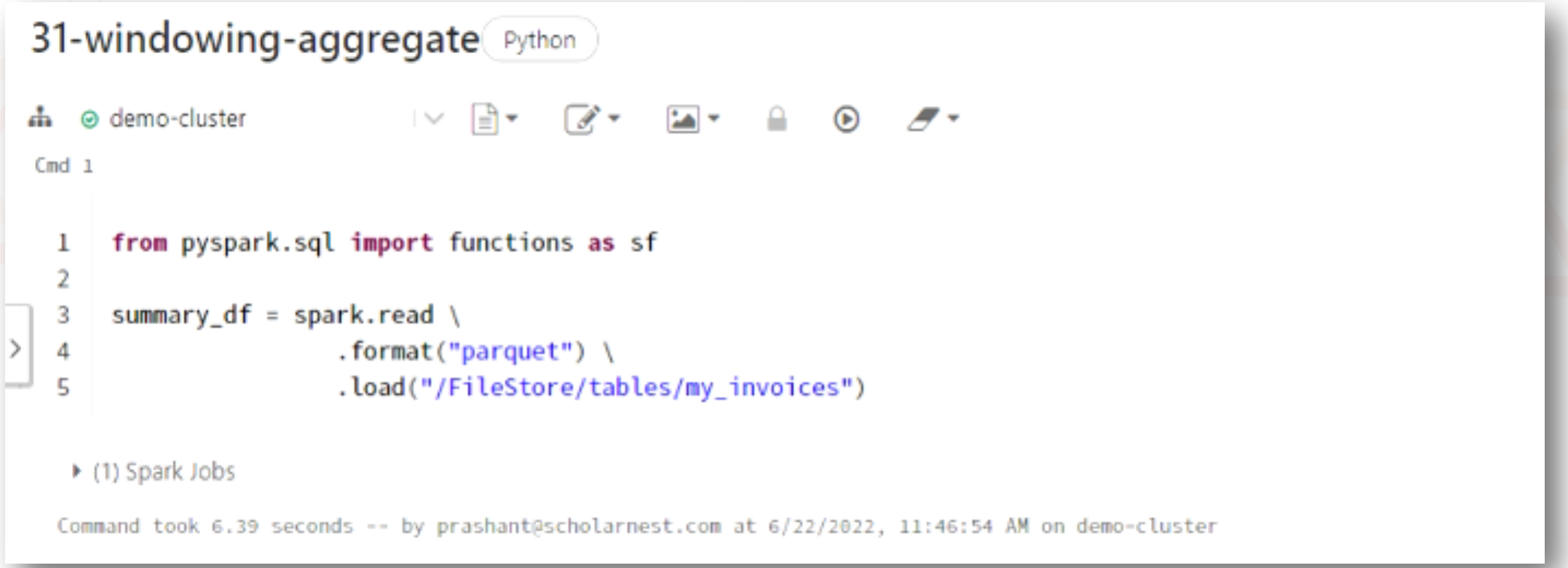
Lecture:
Windowing Aggregation

Windowing Aggregations

Go to your Databricks workspace and create a new notebook. **(Reference: 31-windowing-aggregate)**



Now, we want to read the parquet dataset from a directory where we created it in the previous lecture. And we have the code for that given below.
I am importing spark functions because I want to use some of the functions from this package. Then I am reading data into the Dataframe. The format is parquet, and the file location is the directory name.



You can see that the directory contains additional files such as _SUCCESS, _started, and _commited. But our focus is on the snappy compressed parquet file.

Cmd 2

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

	path	name
1	dbfs:/FileStore/tables/my_invoices/_SUCCESS	_SUCCESS
2	dbfs:/FileStore/tables/my_invoices/_committed_1560325843610938963	_committed_1560325843610938963
3	dbfs:/FileStore/tables/my_invoices/_started_1560325843610938963	_started_1560325843610938963
4	dbfs:/FileStore/tables/my_invoices/part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7328fc2-29-1-c000.snappy.parquet	part-00000-tid-1560325843610938963-846b6ca6-639d-443f-a6b5-596ef7

Showing all 4 rows.

Command took 14.73 seconds -- by prashant@scholarnest.com at 6/22/2022, 11:47:30 AM on demo-cluster

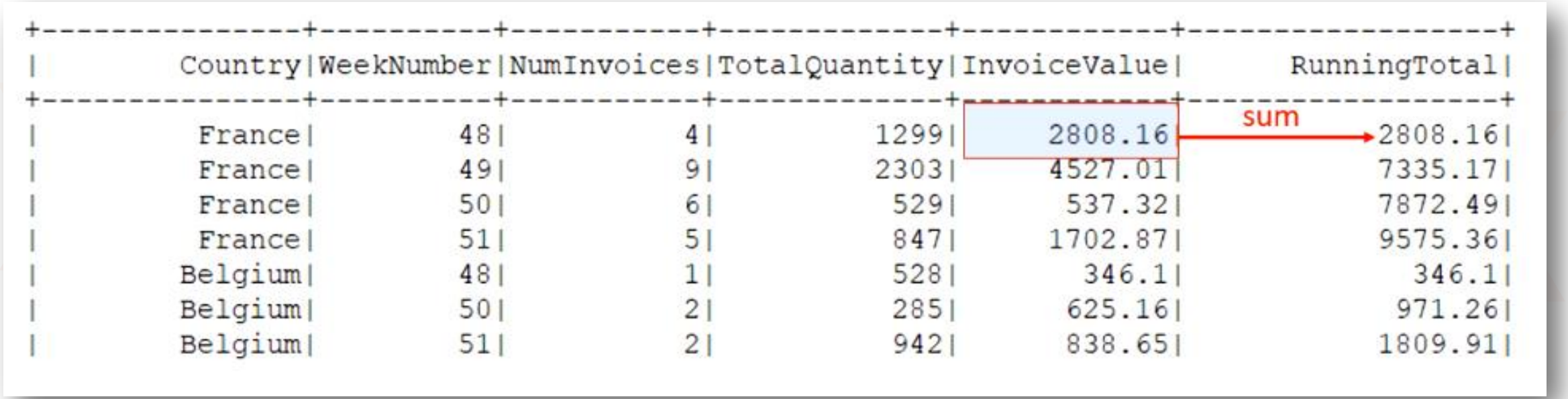
You can see that I am using only the directory name here in the load() method, and we are getting the desired output.

So you might think about how Spark will know which files to load and which ones to ignore?
Spark Dataframe reader will read all the data files in the given directory.
You can give a specific file name, and the Dataframe reader will read only one file.
But you can also give a directory name, and the Dataframe reader will read all the files in the given directory except the hidden files. And those hidden file names start with an underscore.
When you write a Dataframe, you can give a directory name.
You cannot give a file name. Spark Dataframe writer doesn't accept output file names.
It only takes a directory location and creates one or more files in the given directory.
However, the Dataframe writer will also create control files such as _SUCCESS, _started, and _commited. These are not data files. These are hidden control files. And the name starts with an underscore.
When you read the same directory using the Dataframe reader, they ignore the control files where the name starts with an underscore.
We write Dataframe data in a directory using the Dataframe writer and read it again using the Dataframe reader from the same directory.
The Dataframe write and the reader APIs are designed to create some control files and ignore those control files.

Now I have the data file and I want to compute each country's week-by-week running total.
The outcome should look like the one shown below.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17
France	50	6	529	537.32	7872.49
France	51	5	847	1702.87	9575.36
Belgium	48	1	528	346.1	346.1
Belgium	50	2	285	625.16	971.26
Belgium	51	2	942	838.65	1809.91

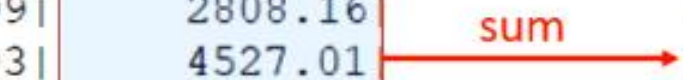
Assume we are starting with week 48.
So the running total of week 48 is the same as the invoice value.



Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17
France	50	6	529	537.32	7872.49
France	51	5	847	1702.87	9575.36
Belgium	48	1	528	346.1	346.1
Belgium	50	2	285	625.16	971.26
Belgium	51	2	942	838.65	1809.91

But for week 49, it is the sum of the previous two Invoice Values.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17
France	50	6	529	537.32	7872.49
France	51	5	847	1702.87	9575.36
Belgium	48	1	528	346.1	346.1
Belgium	50	2	285	625.16	971.26
Belgium	51	2	942	838.65	1809.91



Similarly, for week 50, it will be the sum of the previous three Invoice Values.
And it goes on in the same way for each country.
But remember, this logic restarts when you get to a new country.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17
France	50	6	529	537.32	7872.49
France	51	5	847	1702.87	9575.36
Belgium	48	1	528	346.1	346.1
Belgium	50	2	285	625.16	971.26
Belgium	51	2	942	838.65	1809.91

sum

Look at the data from Belgium.
It again starts with week 48, and the running total is the same as the invoice value of week 48.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17
France	50	6	529	537.32	7872.49
France	51	5	847	1702.87	9575.36
Belgium	48	1	528	346.1	346.1
Belgium	50	2	285	625.16	971.26
Belgium	51	2	942	838.65	1809.91

sum

We do not have week 49 data for Belgium.
But we have week 50 data. So the running total of wee 50 is the sum of the invoice value of all previous weeks.
Similarly, the running total of week 51 is the sum of the invoice value of all the previous weeks.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17
France	50	6	529	537.32	7872.49
France	51	5	847	1702.87	9575.36
Belgium	48	1	528	346.1	346.1
Belgium	50	2	285	625.16	971.26
Belgium	51	2	942	838.65	1809.91

sum

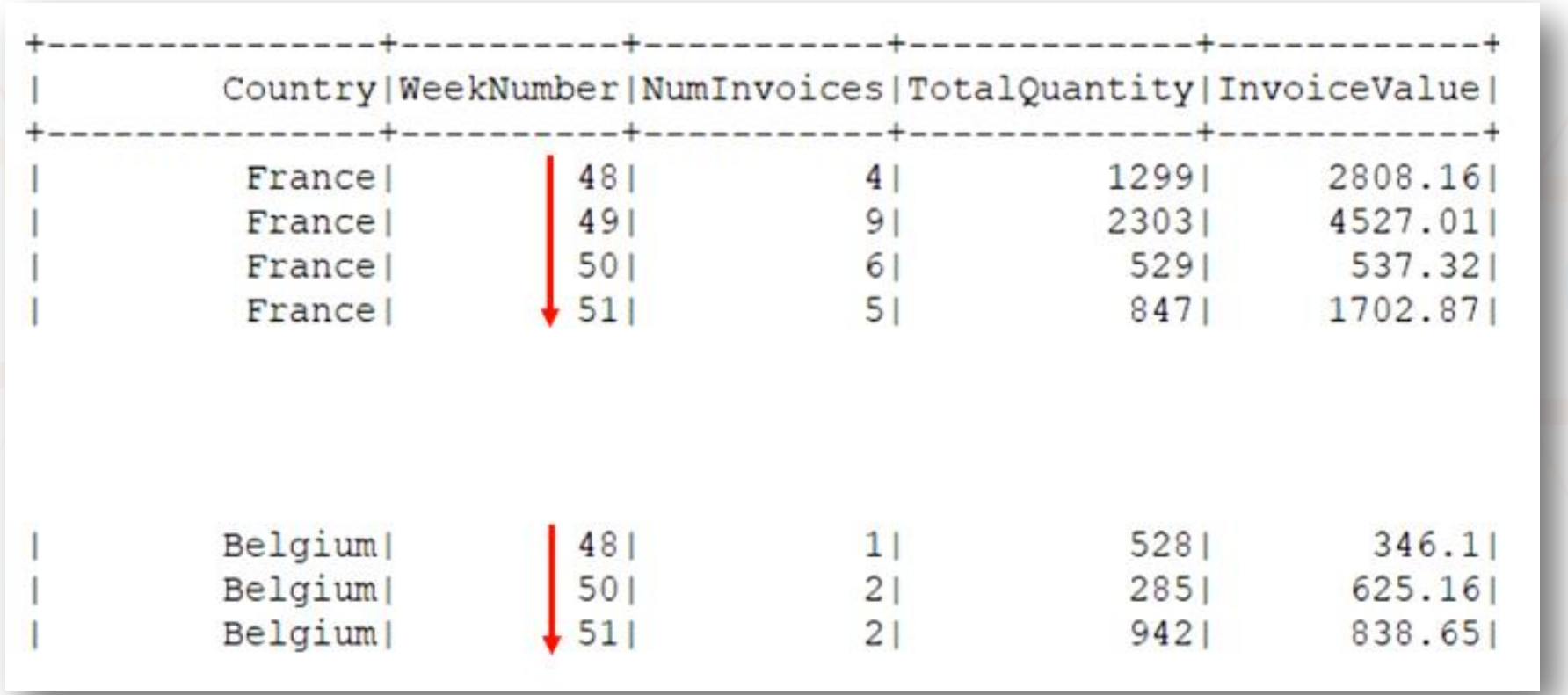
And that's what running total by country means.
How do you do it? We cannot achieve this requirement using simple or grouping aggregation.
These kinds of aggregates are called windowing aggregates.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
France	48	4	1299	2808.16
France	49	9	2303	4527.01
France	50	6	529	537.32
France	51	5	847	1702.87
Belgium	48	1	528	346.1
Belgium	50	2	285	625.16
Belgium	51	2	942	838.65

We want to compute running totals for each country. The running total should restart for each country. So the first thing is to make sure that we break this Dataframe by the country. It is like partitioning your data frame by country.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
France	48	4	1299	2808.16
France	49	9	2303	4527.01
France	50	6	529	537.32
France	51	5	847	1702.87
Belgium	48	1	528	346.1
Belgium	50	2	285	625.16
Belgium	51	2	942	838.65

Now the next critical thing is to make sure that we order each partition by the week number.
Why? Because we want to compute week-by-week totals.
So the running total for week 50 should be the sum of the week 48+49+50 in the same sequence.



Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
France	48	4	1299	2808.16
France	49	9	2303	4527.01
France	50	6	529	537.32
France	51	5	847	1702.87
Belgium	48	1	528	346.1
Belgium	50	2	285	625.16
Belgium	51	2	942	838.65

Now the last thing is to see how to compute the running total.
And we can calculate the totals using a window frame of records.
So, the running-total window frame starts with one record.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	
France	50	6	529	537.32	
France	51	5	847	1702.87	
Belgium	48	1	528	346.1	
Belgium	50	2	285	625.16	
Belgium	51	2	942	838.65	

We take the total and extend the window frame to two records.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	
France	50	6	529	537.32	
France	51	5	847	1702.87	
Belgium	48	1	528	346.1	
Belgium	50	2	285	625.16	
Belgium	51	2	942	838.65	

Take the sum once again. And then extend the window frame to three records. And this goes on for the entire partition.

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17
France	50	6	529	537.32	7872.49
France	51	5	847	1702.87	
Belgium	48	1	528	346.1	
Belgium	50	2	285	625.16	
Belgium	51	2	942	838.65	

The same logic repeats for the other partitions.

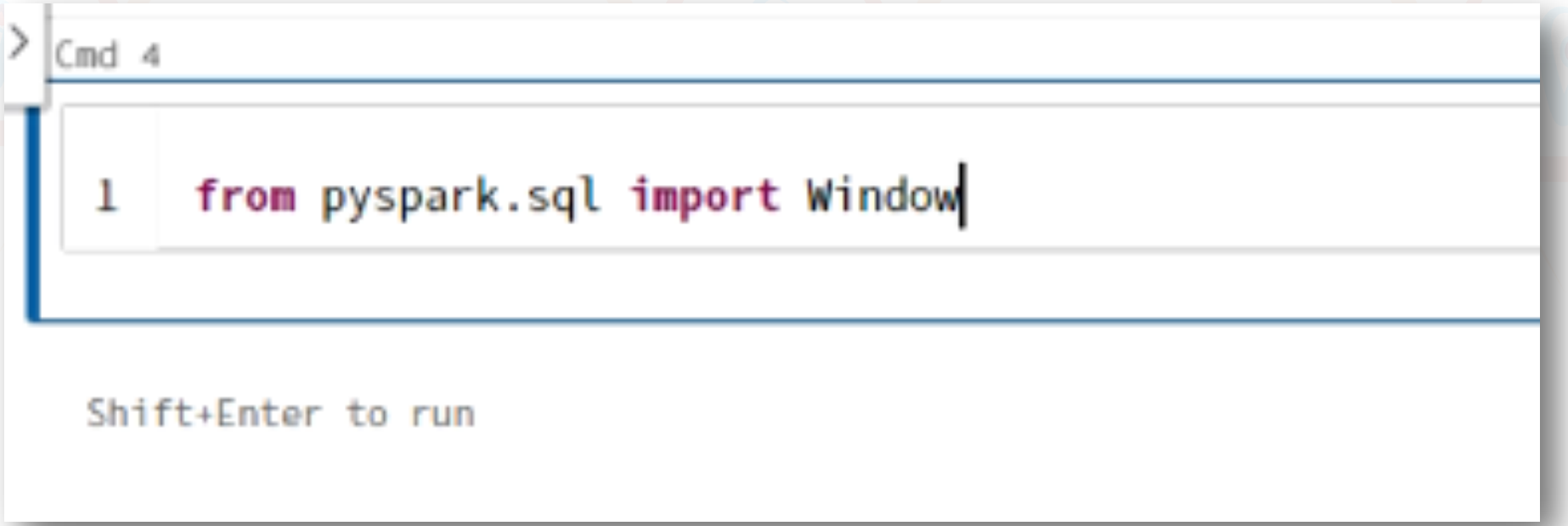
Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17
France	50	6	529	537.32	7872.49
France	51	5	847	1702.87	9575.36
Belgium	48	1	528	346.1	346.1
Belgium	50	2	285	625.16	971.26
Belgium	51	2	942	838.65	1809.91

This windowing can help you compute a whole new class of aggregations.
And using them is a three-step process.

1. Identify your partitioning columns. In our example, it was the country.
2. Identify your ordering requirement. And we wanted it to order by week number.
3. Finally, define your window frame. In our case, The window frame starts with the first record and includes everything till the current record.

For example, when computing the running total for week 50, the window frame begins at the first record, week 48.
And it ends at the current record, which will be week 50.

Now let us implement what we discussed till now.
We already loaded the parquet data to a data frame.
And now, we are ready to aggregate the data frame and compute the running total. However, in this example, we need windowing aggregate. So before we do any aggregation, let's define our window.
Spark gives you a window object which I have imported as shown below.



For any windowing aggregate, we have three essential things:

1. What is your Partition column?
2. What is your Ordering column?
3. What is your window frame?

The window object allows you to define these three things.

We define a variable for windowing and start with the window object.
I wanted to partition my Dataframe using the country column, so I have done the same at first. The next thing is to define the ordering. We wanted to order by the week number.

Cmd 4

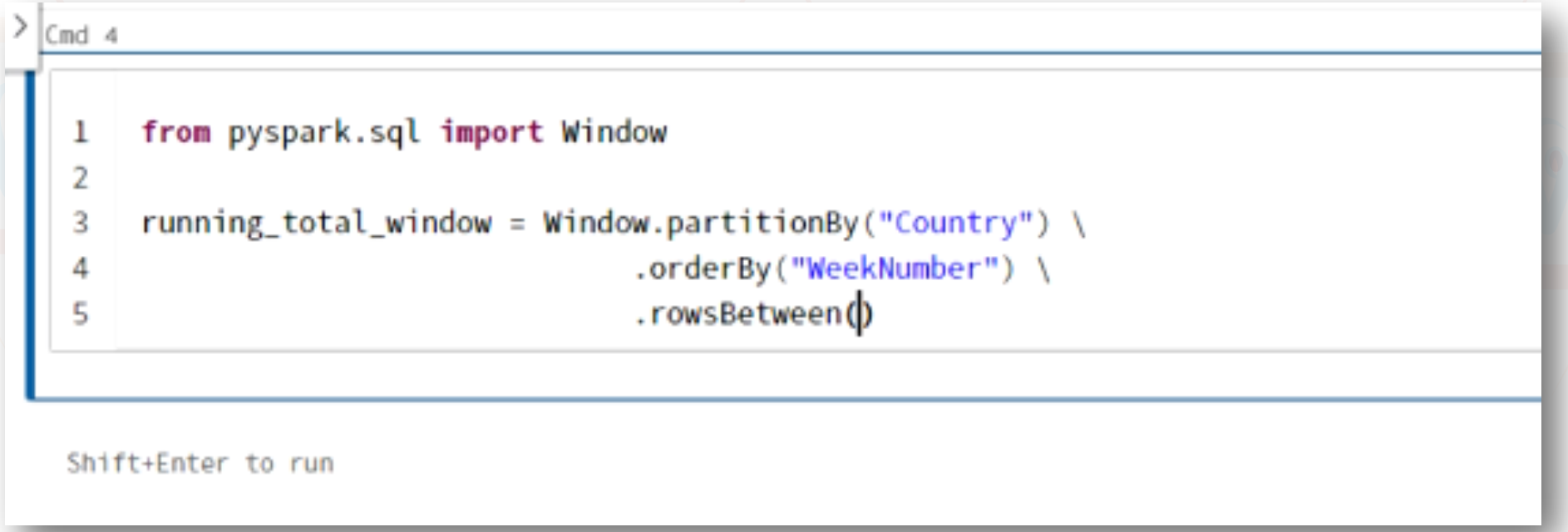
```
1  from pyspark.sql import Window
2
3  running_total_window = Window.partitionBy("Country") \
4                               .orderBy("WeekNumber")
```

Shift+Enter to run

Finally, we want to define our window frame. And that one is done using the rowsBetween()

The rowsBetween() takes two arguments:

1. starting position of the window frame
2. end position of your window frame



Start from the beginning of the partition. So, we have a constant to say that. The *Window.unboundedPreceding* says start from the beginning.
What is the end of our window frame? We want to take the total up to the current row. We have a constant for that also, *Window.currentRow*.
And that is all, we have the window definition.

Now computing aggregate over this window is straightforward.
We start with the Dataframe and use the withColumn() to add a new column for running total. The running total is a sum(InvoiceValue). But it is not a simple sum. It is a sum() over a window. And we use the over() function to say that.

Cmd 4

```
1  from pyspark.sql import Window
2
3  running_total_window = Window.partitionBy("Country") \
4                               .orderBy("WeekNumber") \
5                               .rowsBetween(Window.unboundedPreceding, Window.currentRow)
6
7  summary_df.withColumn("RunningTotal", sf.sum("InvoiceValue").over(running_total_window)) \
8          .show(100)
```


You can see the results below.
So this unboundedPreceding means to take all the rows from the beginning.

Cmd 4

```
1  from pyspark.sql import Window
2
3  running_total_window = Window.partitionBy("Country") \
4                               .orderBy("WeekNumber") \
5                               .rowsBetween(Window.unboundedPreceding, Window.currentRow)
6
7  summary_df.withColumn("RunningTotal", sf.sum("InvoiceValue").over(running_total_window)) \
8          .show(100)
```

▸ (2) Spark Jobs

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
Australia	48	1	107	358.25	358.25
Australia	49	1	214	258.9	617.15
Australia	50	2	133	387.95	1005.0999999999999
Austria	50	2	3	257.04	257.04
Bahrain	51	1	54	205.74	205.74
Belgium	48	1	528	346.1	346.1
Belgium	50	2	285	625.16	971.26
Belgium	51	2	942	838.65	1809.9099999999999
Channel Islands	49	1	80	363.53	363.53
Cyprus	50	1	917	1590.82	1590.82
Denmark	49	1	454	1281.5	1281.5
EIRE	48	7	2822	3147.23	3147.23
EIRE	49	5	1280	3284.1	6431.33
EIRE	50	5	1184	2321.78	8753.11

However, you can give any numeric value here. For example, If I can give -2 as the start of the window.

Cmd 4

```
1  from pyspark.sql import Window
2
3  running_total_window = Window.partitionBy("Country") \
4                              .orderBy("WeekNumber") \
5                              .rowsBetween(-2, Window.currentRow)
6
7  summary_df.withColumn("RunningTotal", sf.sum("InvoiceValue").over(running_total_window)) \
8          .show(100)
```

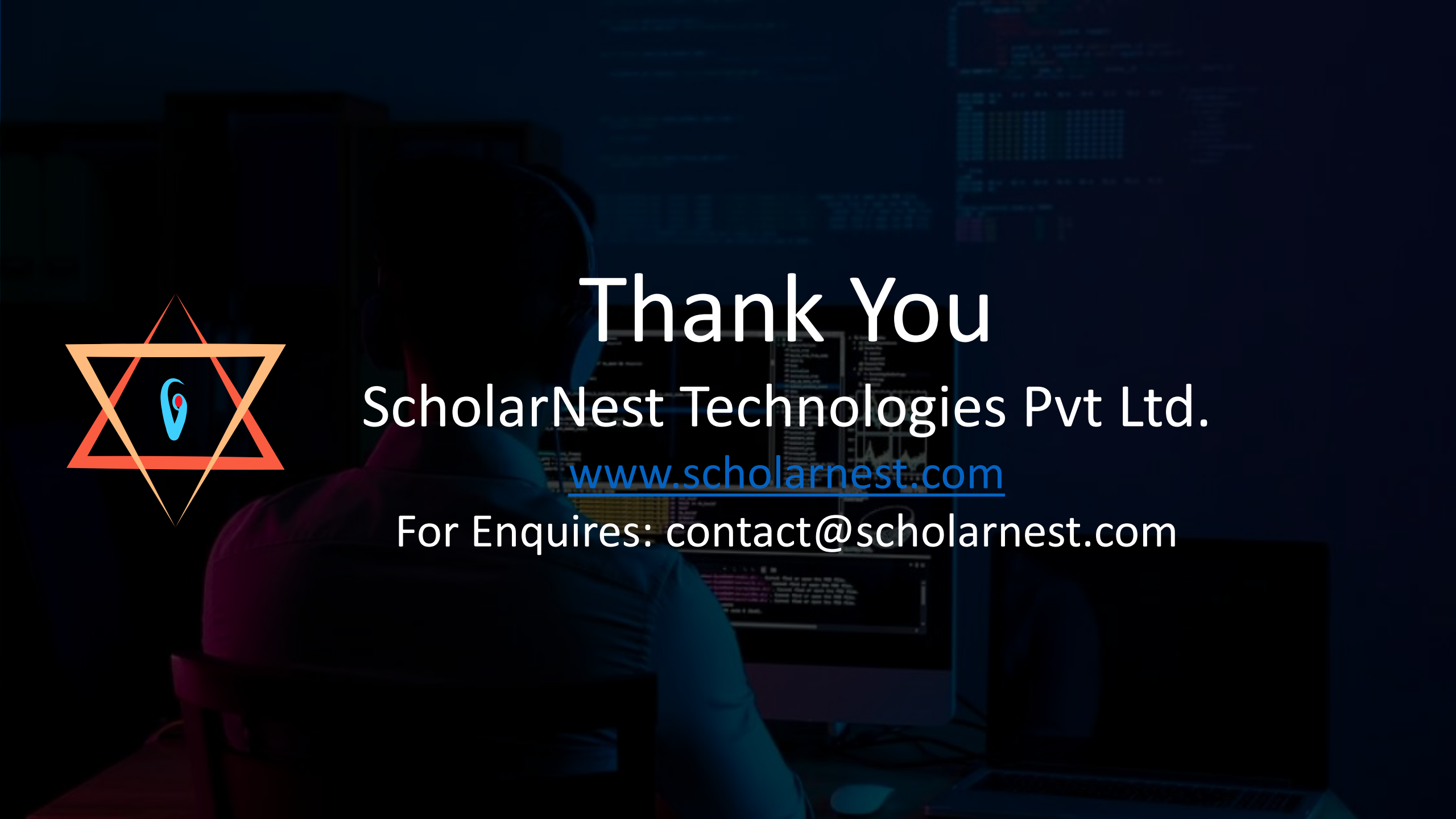
▸ (2) Spark Jobs

So, the window will now start from the two rows before the current row. The end of the window is the current row. So, you will be calculating the running total for a three-week window.

▸ (2) Spark Jobs

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
Australia	48	1	107	358.25	358.25
Australia	49	1	214	258.9	617.15
Australia	50	2	133	387.95	1005.0999999999999
Austria	50	2	3	257.04	257.04
Bahrain	51	1	54	205.74	205.74
Belgium	48	1	528	346.1	346.1
Belgium	50	2	285	625.16	971.26
Belgium	51	2	942	838.65	1809.9099999999999
Channel Islands	49	1	80	363.53	363.53
Cyprus	50	1	917	1590.82	1590.82
Denmark	49	1	454	1281.5	1281.5
EIRE	48	7	2822	3147.23	3147.23
EIRE	49	5	1280	3284.1	6431.33
EIRE	50	5	1184	2321.78	8753.11
EIRE	51	5	95	276.84	5882.72
Finland	50	1	1254	892.8	892.8
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17

Command took 1.72 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:09:06 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

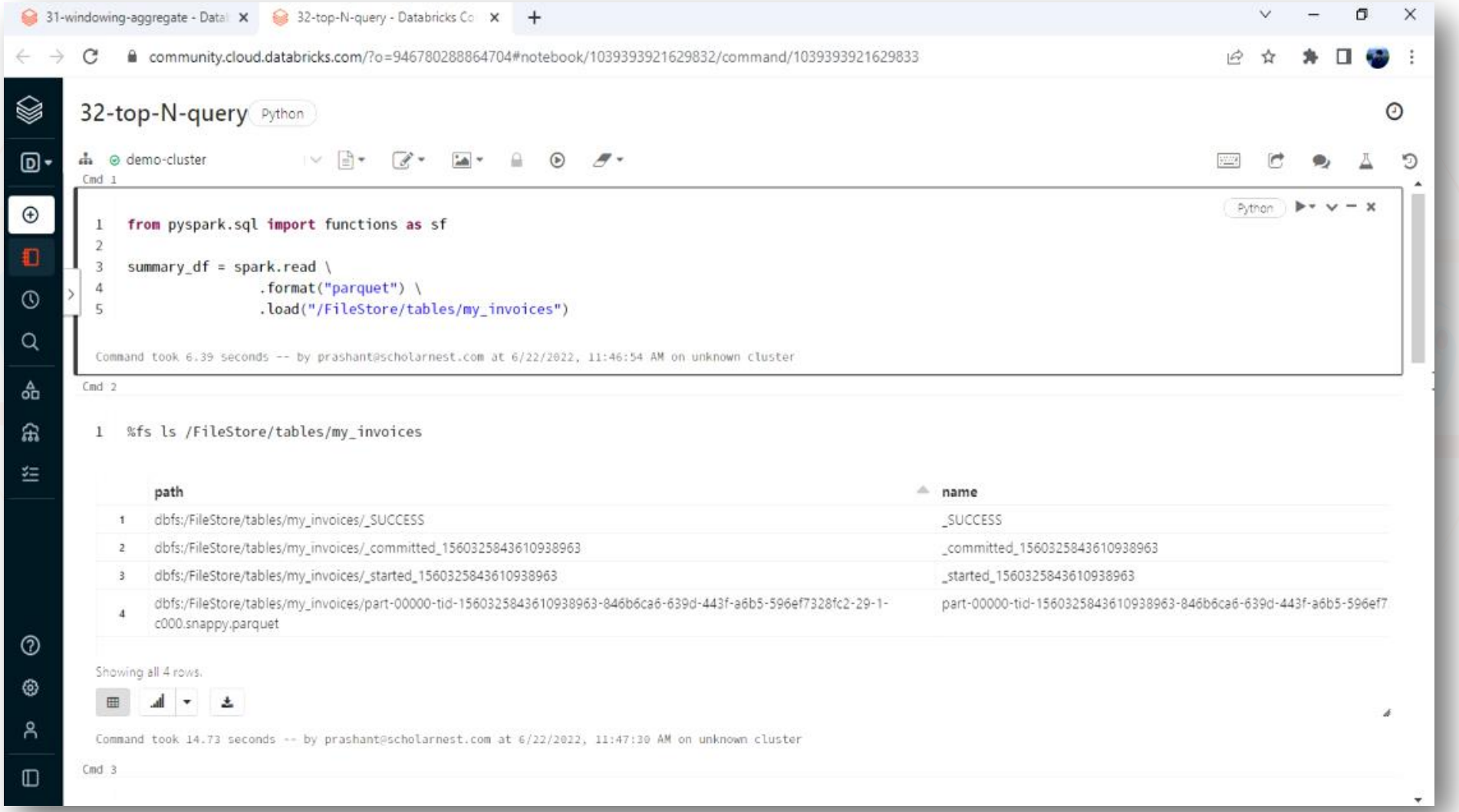
Module:
Aggregation

Lecture:
Top-N
Aggregation

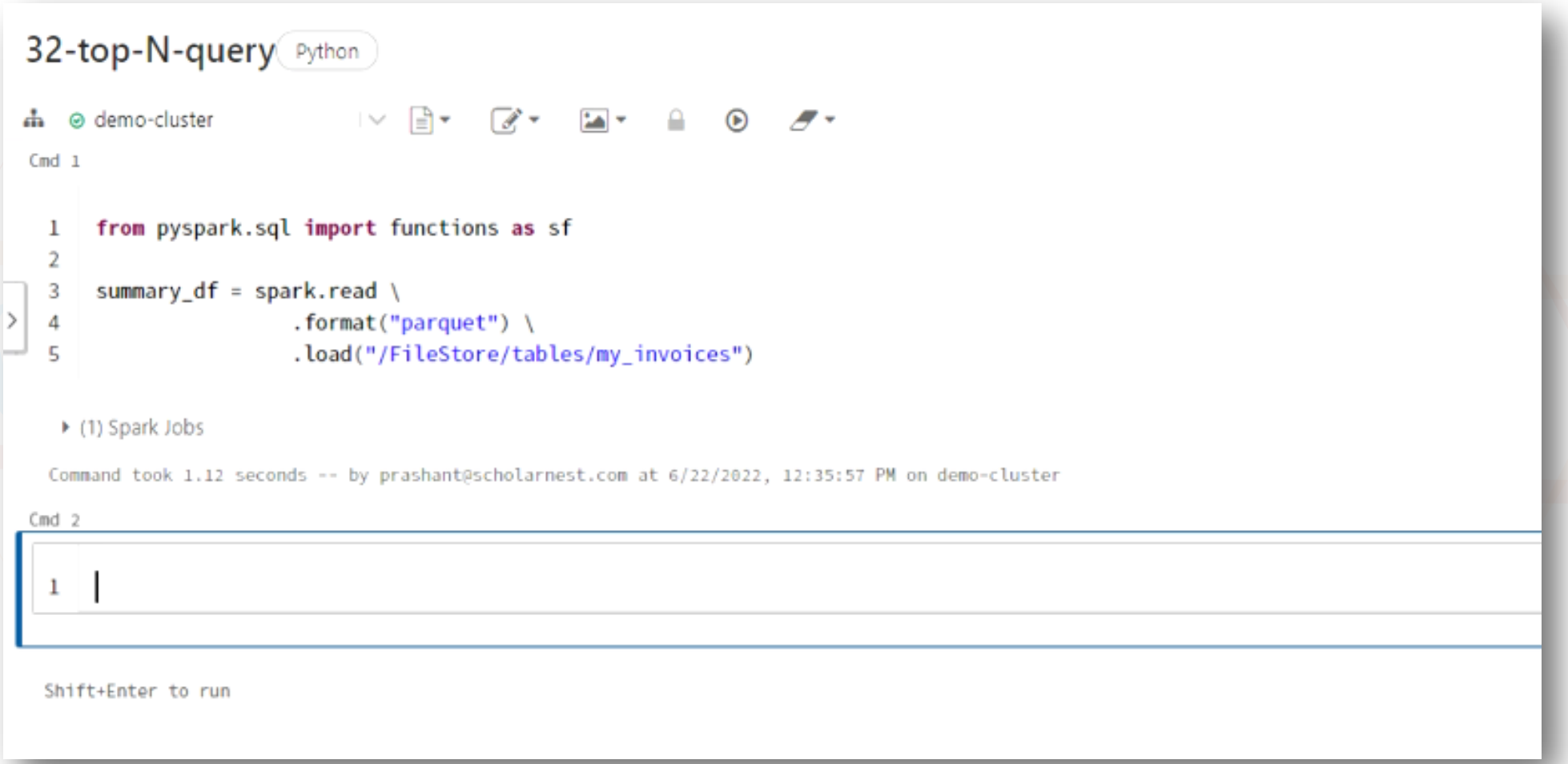
Top-N Aggregations

Go to your Databricks workspace and clone the notebook we used in the previous lecture(31-windowing-aggregation).

(Reference: 32-top-N-query)



I will keep only the first cell and remove everything else. Run it once, so that we have a Dataframe.



You can see the Dataframe in the screen shot below.
I sorted this Dataframe by country and the week number.

Cmd 2

```
1  summary_df.sort("Country", "WeekNumber").show()
```

▸ (1) Spark Jobs

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
Australia	48	1	107	358.25
Australia	49	1	214	258.9
Australia	50	2	133	387.95
Austria	50	2	3	257.04
Bahrain	51	1	54	205.74
Belgium	48	1	528	346.1
Belgium	50	2	285	625.16
Belgium	51	2	942	838.65
Channel Islands	49	1	80	363.53
Cyprus	50	1	917	1590.82
Denmark	49	1	454	1281.5
EIRE	48	7	2822	3147.23
EIRE	49	5	1280	3284.1
EIRE	50	5	1184	2321.78
EIRE	51	5	95	276.84
Finland	50	1	1254	892.8
France	48	4	1299	2808.16
France	49	9	2303	4527.01

Command took 1.18 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:36:23 PM on demo-cluster

Now let's see the first three records. These records are for Australia from week 48 to week 50. The total invoice value for each week is also shown in the output. Which week had the maximum sale? Week 50.

Cmd 2

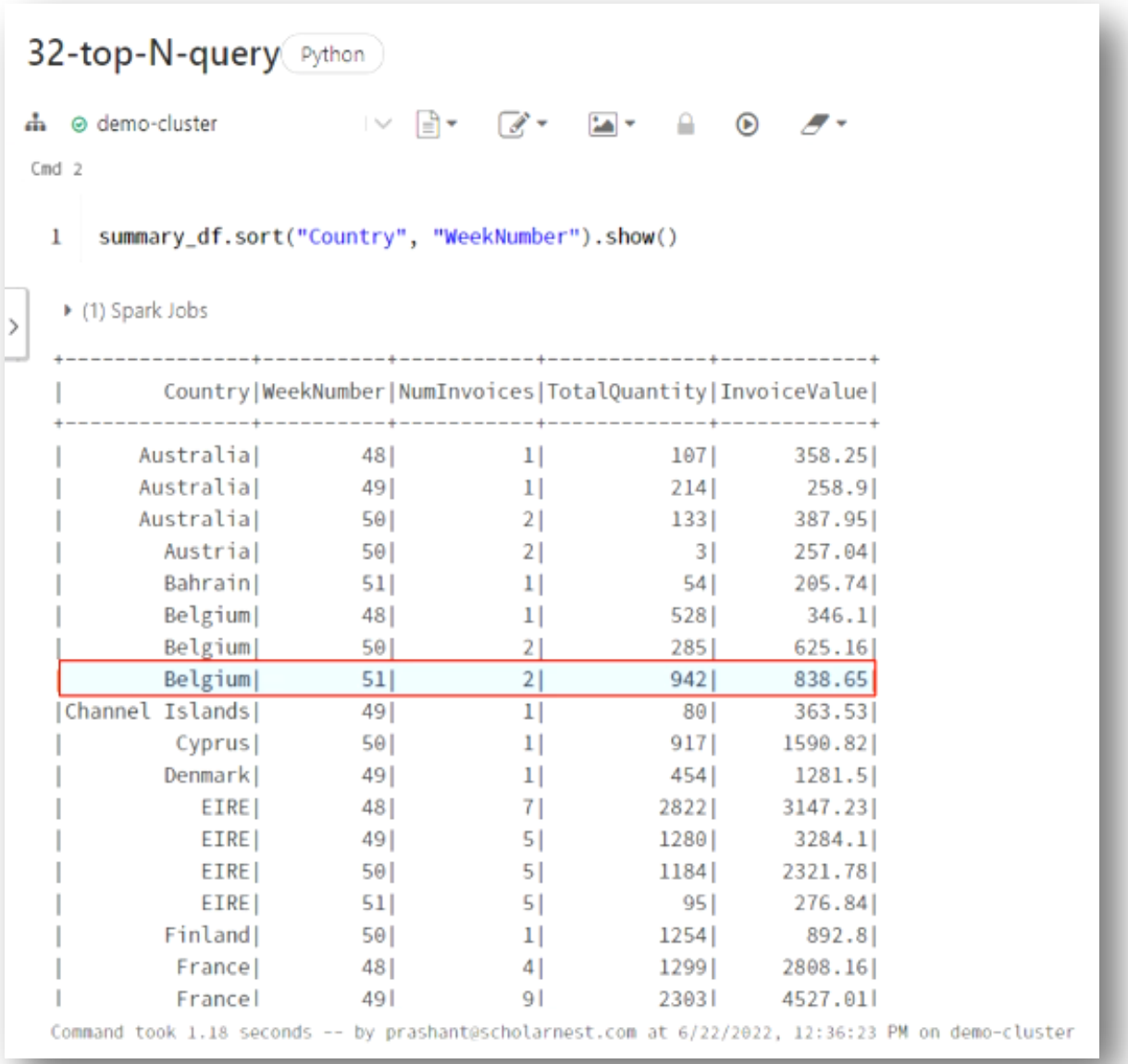
```
1  summary_df.sort("Country", "WeekNumber").show()
```

▸ (1) Spark Jobs

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
Australia	48	1	107	358.25
Australia	49	1	214	258.9
Australia	50	2	133	387.95
Austria	50	2	3	257.04
Bahrain	51	1	54	205.74
Belgium	48	1	528	346.1
Belgium	50	2	285	625.16
Belgium	51	2	942	838.65
Channel Islands	49	1	80	363.53
Cyprus	50	1	917	1590.82
Denmark	49	1	454	1281.5
EIRE	48	7	2822	3147.23
EIRE	49	5	1280	3284.1
EIRE	50	5	1184	2321.78
EIRE	51	5	95	276.84
Finland	50	1	1254	892.8
France	48	4	1299	2808.16
France	49	9	2303	4527.01

Command took 1.18 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:36:23 PM on demo-cluster

Similarly, which week for Belgium had the maximum sale? The correct answer is week 51.



So we have weekly sales numbers for different countries. We want to find the top-selling week for each country. So for Australia, we want to see only week 50, and for Belgium, it should be week 51. And that's a top one query.

Cmd 2

```
1  summary_df.sort("Country", "WeekNumber").show()
```

▸ (1) Spark Jobs

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
Australia	48	1	107	358.25
Australia	49	1	214	258.9
Australia	50	2	133	387.95
Austria	50	2	3	257.04
Bahrain	51	1	54	205.74
Belgium	48	1	528	346.1
Belgium	50	2	285	625.16
Belgium	51	2	942	838.65
Channel Islands	49	1	80	363.53
Cyprus	50	1	917	1590.82
Denmark	49	1	454	1281.5
EIRE	48	7	2822	3147.23
EIRE	49	5	1280	3284.1
EIRE	50	5	1184	2321.78
EIRE	51	5	95	276.84
Finland	50	1	1254	892.8
France	48	4	1299	2808.16
France	49	9	2303	4527.01

Command took 1.18 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:36:23 PM on demo-cluster

We want to find the top-selling weeks for each country.
How will you do it?
I thought of a three step procedure:

1. Break the Dataframe into country-wise partitions. Because we want to find top selling week by each country.
2. Sort each partition in descending order of the total sales. So we know which one comes at the top.
3. Then we take the first record from each sorted partition.

Finally, you will get the record for the top-selling week for each country.

So we need to partition the data by country and sort each partition by the sales. And that's what windowing allows us to do. So whenever you see a requirement to partition data and sort each partition, you are likely to use the windowing aggregate.
You will get a window of records that is partitioned and sorted in descending order. Then you can rank each record with its position within the window so you can take the top ranking records.

The first thing is to import the window class. Then I will define a window and call it rank_window. Then, we want to partition this window by country.

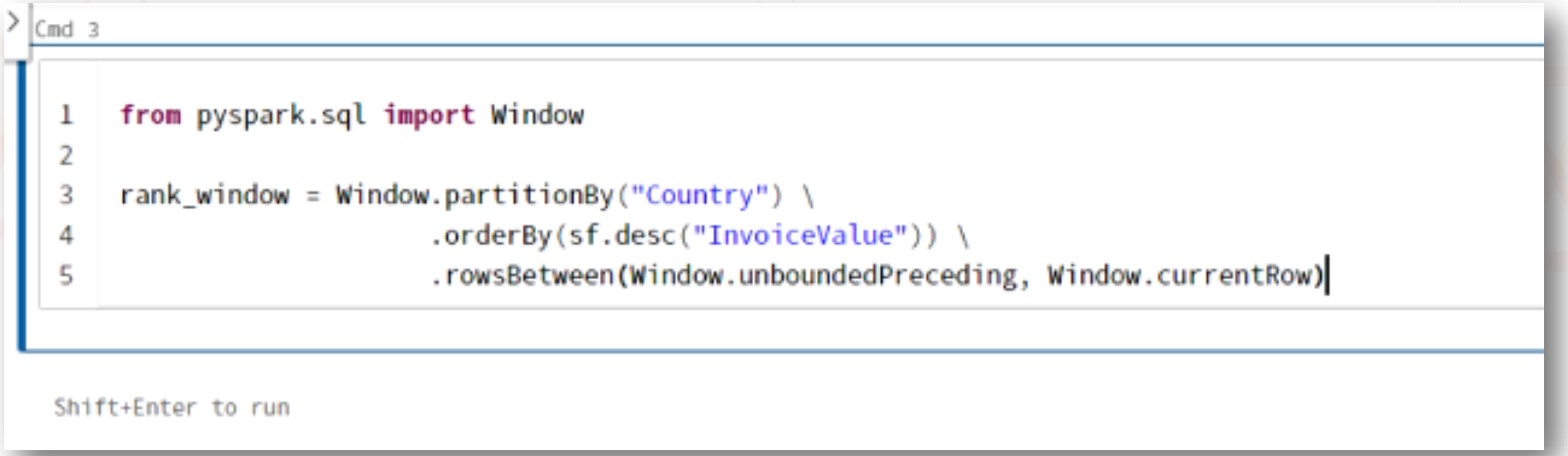
Then we should order each partition by the InvoiceValue. But we wanted to sort it in descending order. The highest value must come first. So, we should apply the desc() function alongwith orderBy().

Cmd 3

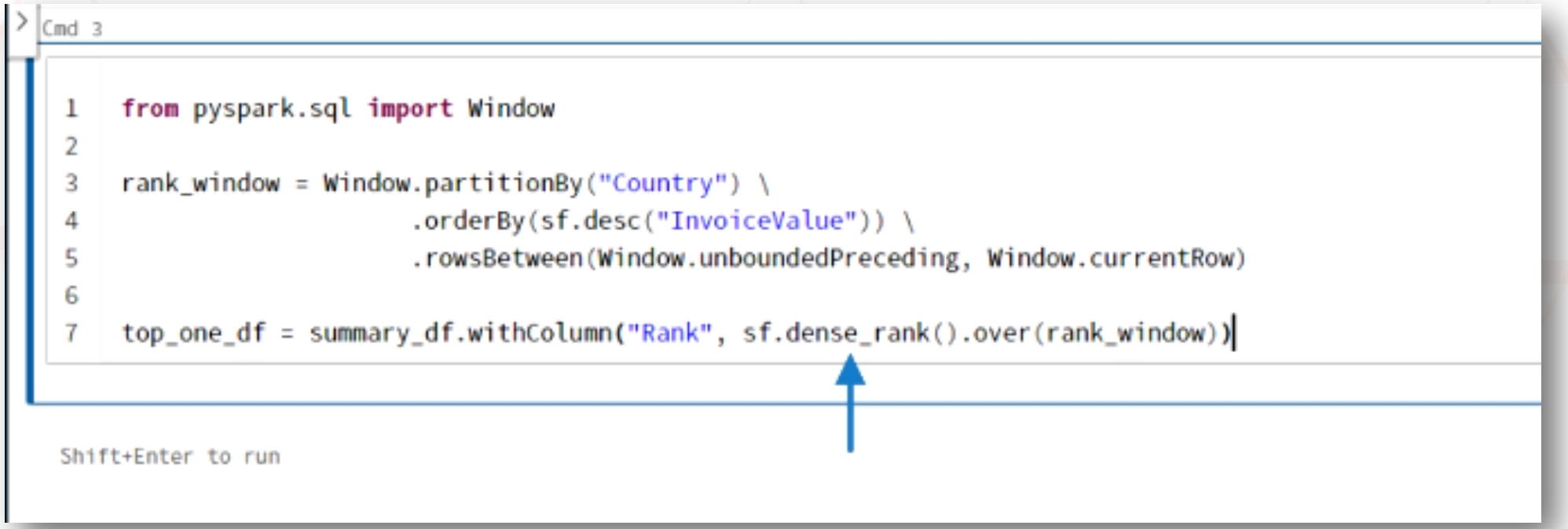
```
1  from pyspark.sql import Window
2
3  rank_window = Window.partitionBy("Country") \
4                      .orderBy(sf.desc("InvoiceValue"))
```

Shift+Enter to run

Every window must have a window start and a window end. So you should apply the rowsBetween() method. But what are the start and end of the window frame?
We need all the rows. So we start from the unboundedPreceding and take everything up to the currentRow.
Now, we are done setting up a window.



Now we want to rank all the records within a window based on their position.
So I want to add a new column to the Dataframe.
The column name is rank, and the value should be the record rank within the country window.
The dense_rank() function gives me a record rank. But I do not want continuous rank. I want record ranks over the country window. And I have written the code for all this in the last line shown below.



Now you can display the Dataframe.
Look at Australia. The highest record is rank 1, the second-highest in rank two, and so on.

```
1  from pyspark.sql import Window
2
3  rank_window = Window.partitionBy("Country") \
4                      .orderBy(sf.desc("InvoiceValue")) \
5                      .rowsBetween(Window.unboundedPreceding, Window.currentRow)
6
7  top_one_df = summary_df.withColumn("Rank", sf.dense_rank().over(rank_window))
8
9  display(top_one_df.orderBy("Country", "Rank"))
```

▸ (2) Spark Jobs

	Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	Rank
1	Australia	50	2	133	387.95	1
2	Australia	48	1	107	358.25	2
3	Australia	49	1	214	258.9	3
4	Austria	50	2	3	257.04	1
5	Bahrain	51	1	54	205.74	1
6	Belgium	51	2	942	838.65	1
7	Belgium	50	2	285	625.16	2

Showing all 51 rows.

Command took 1.61 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:42:36 PM on demo-cluster

If you scroll down and look for the Eire. You can see that we ranked all the records from highest to lowest.

```
1  from pyspark.sql import Window
2
3  rank_window = Window.partitionBy("Country") \
4                      .orderBy(sf.desc("InvoiceValue")) \
5                      .rowsBetween(Window.unboundedPreceding, Window.currentRow)
6
7  top_one_df = summary_df.withColumn("Rank", sf.dense_rank().over(rank_window))
8
9  display(top_one_df.orderBy("Country", "Rank"))
```

▸ (2) Spark Jobs

	Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	Rank
11	Denmark	49	1	454	1281.5	1
12	EIRE	49	5	1280	3284.1	1
13	EIRE	48	7	2822	3147.23	2
14	EIRE	50	5	1184	2321.78	3
15	EIRE	51	5	95	276.84	4
16	Finland	50	1	1254	892.8	1

Showing all 51 rows.

Command took 1.61 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:42:36 PM on demo-cluster

Now we can filter it as per our requirement.
I wanted to see only the top one, so I could filter the Dataframe for rank==1, and you can see that we got the desired result of top most week for each country.

```
1  top_one_df.where("Rank == 1").show()
```

▸ (2) Spark Jobs

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	Rank
Australia	50	2	133	387.95	1
Austria	50	2	3	257.04	1
Bahrain	51	1	54	205.74	1
Belgium	51	2	942	838.65	1
Channel Islands	49	1	80	363.53	1
Cyprus	50	1	917	1590.82	1
Denmark	49	1	454	1281.5	1
EIRE	49	5	1280	3284.1	1
Finland	50	1	1254	892.8	1
France	49	9	2303	4527.01	1
Germany	50	15	1973	5065.79	1
Iceland	49	1	319	711.79	1
Israel	50	1	-56	-227.44	1
Italy	48	1	164	427.8	1
Japan	49	2	3897	7384.99	1
Lithuania	48	3	622	1598.06	1
Netherlands	51	2	6714	8591.88	1
Norway	48	1	1852	1919.14	1

Command took 1.79 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:43:58 PM on demo-cluster

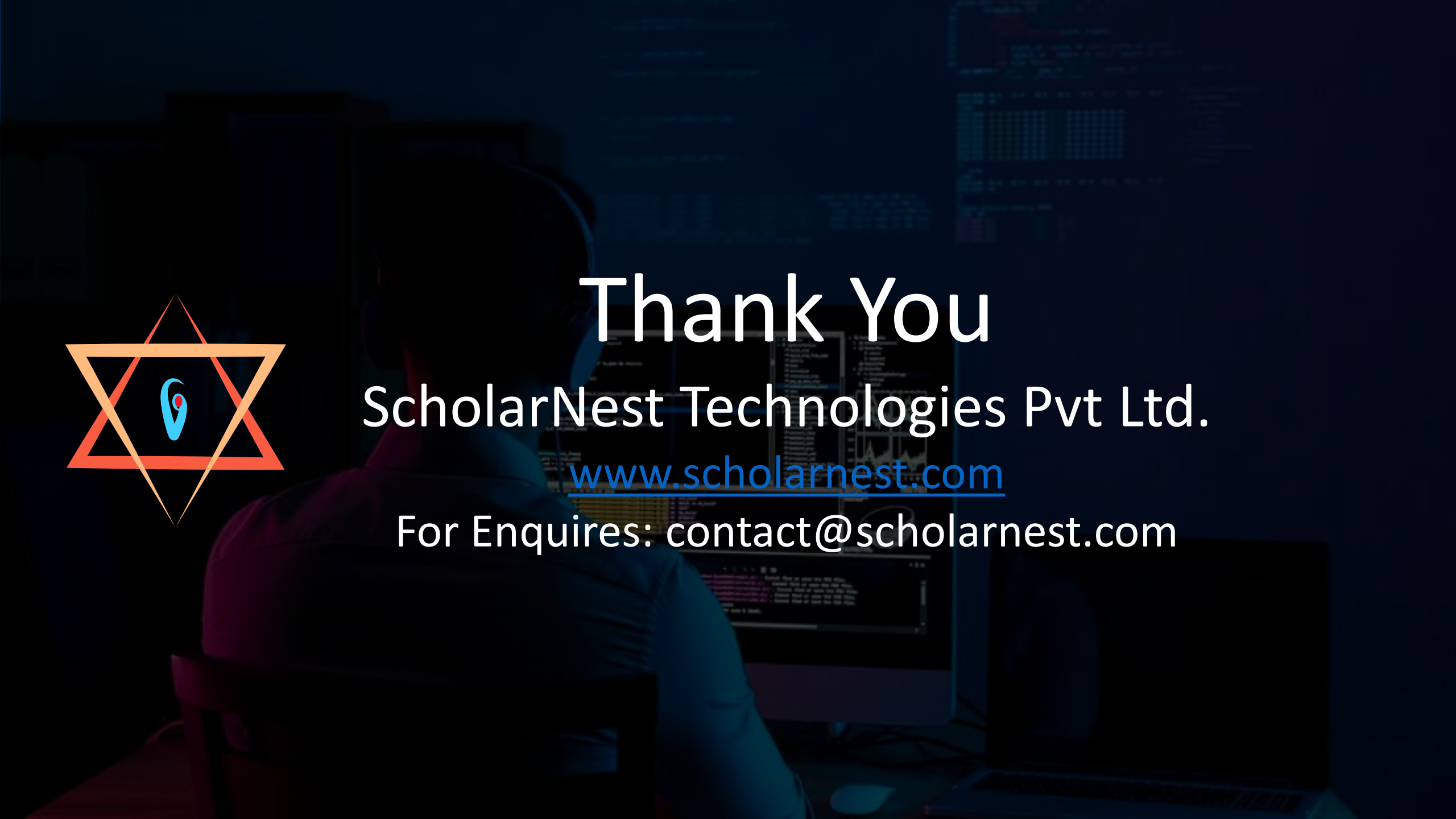
You can filter it where rank <= 3, and it gives you the top three weeks for each country.

```
1  top_one_df.where("Rank <= 3").show()
```

▸ (2) Spark Jobs

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	Rank
Australia	50	2	133	387.95	1
Australia	48	1	107	358.25	2
Australia	49	1	214	258.9	3
Austria	50	2	3	257.04	1
Bahrain	51	1	54	205.74	1
Belgium	51	2	942	838.65	1
Belgium	50	2	285	625.16	2
Belgium	48	1	528	346.1	3
Channel Islands	49	1	80	363.53	1
Cyprus	50	1	917	1590.82	1
Denmark	49	1	454	1281.5	1
EIRE	49	5	1280	3284.1	1
EIRE	48	7	2822	3147.23	2
EIRE	50	5	1184	2321.78	3
Finland	50	1	1254	892.8	1
France	49	9	2303	4527.01	1
France	48	4	1299	2808.16	2
France	51	5	847	1702.87	3

Command took 1.16 seconds -- by prashant@scholarnest.com at 6/22/2022, 12:44:54 PM on demo-cluster



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

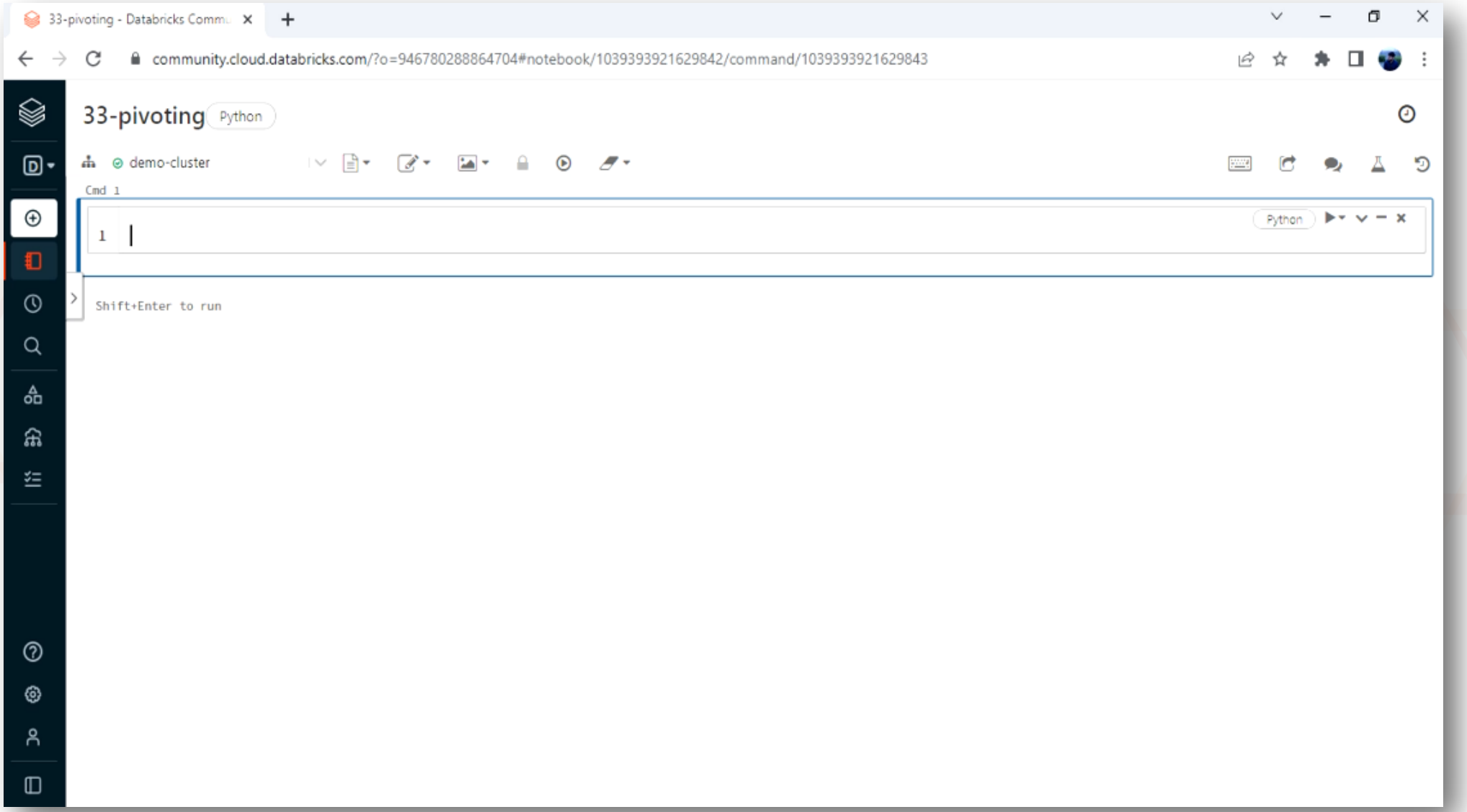
Databricks Spark Certification and Beyond

Module:
Aggregation

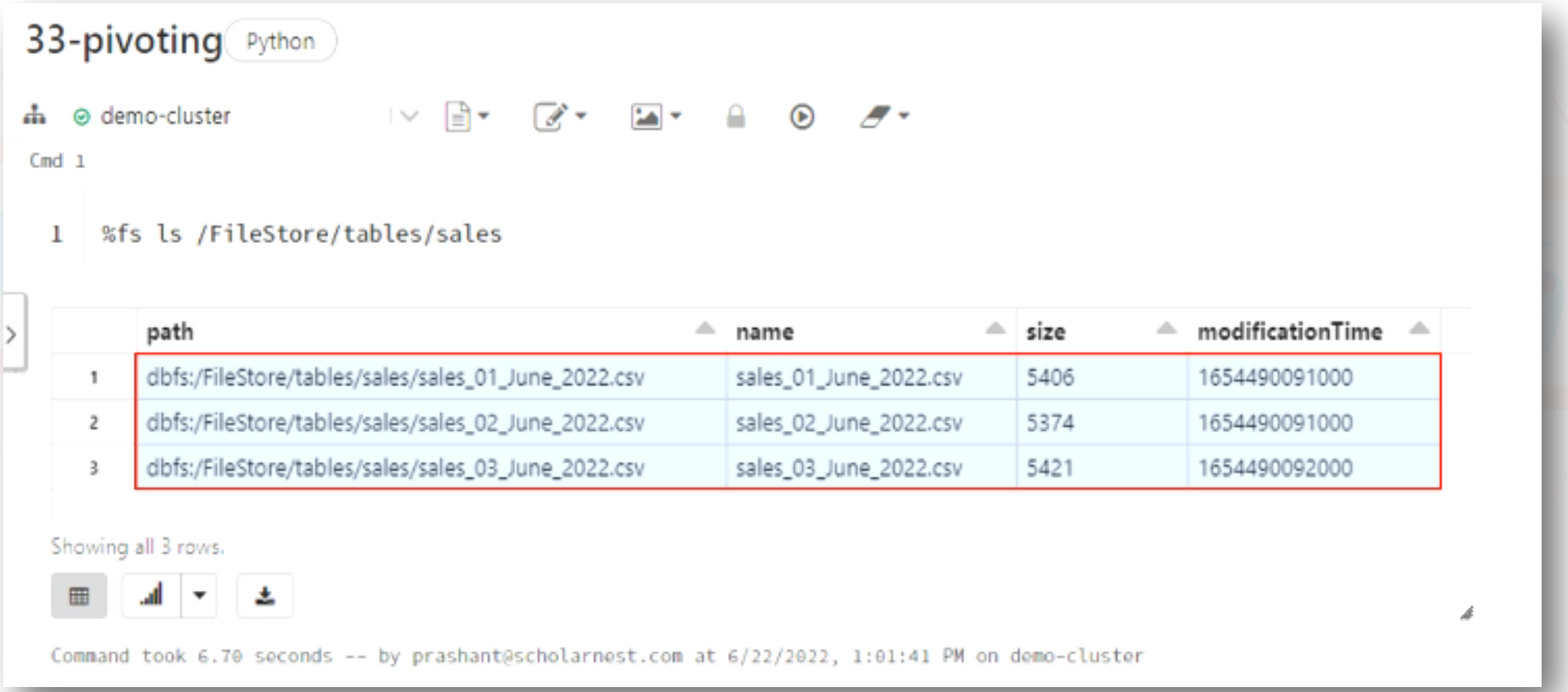
Lecture:
Pivoting
&
Aggregation

Pivoting and Aggregations

Go to your Databricks workspace and create a new notebook. **(Reference: 33-pivoting)**



We need some data to work with an example.
So I have some sales data files. **(Reference: /data/sales)**
So I have these three days of sales data. It is a CSV data file, and all of them are sitting in a sales directory.



Here are some records from one of the sales file.
So we have four fields in this file. The first one is the customer_name. Then we have item_name, purchase_quantity, and unit_price.

Cmd 2

```
1  %fs head /FileStore/tables/sales/sales_01_June_2022.csv
```

```
Bob,apples,4,14
Alice,butter,2,23
Mike,apples,4,14
Mike,milk,3,18
Bob,bread,4,6
Mike,butter,3,23
Alice,butter,3,23
Mike,oranges,2,8
Mike,oranges,4,8
Alice,milk,1,18
Mike,bread,1,6
Alice,oranges,1,8
Bob,bread,2,6
Mike,oranges,1,8
Bob,milk,4,18
Bob,oranges,1,8
Bob,oranges,2,8
Mike,milk,4,18
Bob,apples,3,14
Alice,butter,1,23
Mike,apples,1,14
```

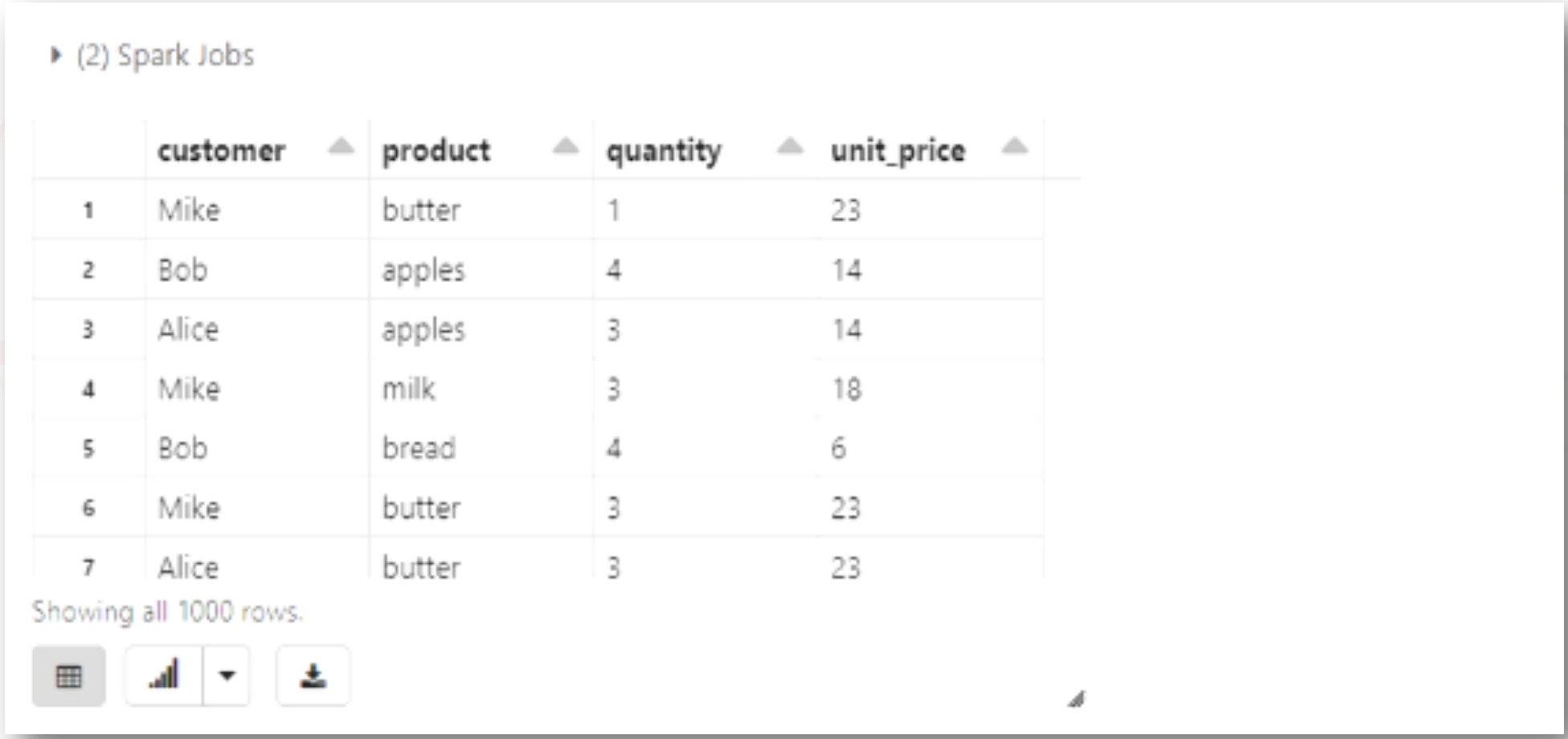
Command took 0.62 seconds -- by prashant@scholarnest.com at 6/22/2022, 1:04:10 PM on demo-cluster

Now, I will load all the files from this directory and create one single sales Dataframe. The code for the same is shown below. I am importing the pyspark functions package because I want to use some aggregation functions. Then I defined the file schema and loaded the Dataframe. The format is CSV, and I am not using the header option because the data file doesn't have a header row. The input to the load() is the directory location. So the Dataframe reader will load all the data files in the given directory.

Cmd 3

```
1  import pyspark.sql.functions as sf
2
3  schema = "customer string, product string, quantity int, unit_price double"
4
5  sales_df = spark.read \
6                  .format("csv") \
7                  .schema(schema) \
8                  .load("/FileStore/tables/sales")
9
10 display(sales_df)
```


Now let's look at the Dataframe.
We have customer_name, item_name, purchase_quantity, and unit_price. And we want to create a pivot table on this data for analysis.
Pivoting is a grouping and cross tab aggregation technique to summarize the data.



	customer	product	quantity	unit_price
1	Mike	butter	1	23
2	Bob	apples	4	14
3	Alice	apples	3	14
4	Mike	milk	3	18
5	Bob	bread	4	6
6	Mike	butter	3	23
7	Alice	butter	3	23

So this one is a cross tab or the pivot table of customer_name and products. We arranged the customers in the rows. The products are arranged in columns.
So it is a matrix of customer and product. Each cell represents the total sales. It means, Bob purchased apples worth 2366. Similarly, Alice purchased milk worth 3042.

customer_name	apples	oranges	milk	bread
Bob	2366.0	1240.0	3366.0	1098.0
Alice	2436.0	1360.0	3042.0	960.0
Mike	2408.0	1328.0	3438.0	014.0

I could have transposed this pivot table differently.
This time, each column represents a customer, and the row represents a product. But the cells are the total sales. So the first and the second Pivot gives you the same information, but the presentation is different.

item_name	Alice	Bob	Mike
oranges	1360.0	1240.0	1328.0
bread	960.0	1098.0	1014.0
apples	2436.0	2366.0	2408.0
milk	3042.0	3366.0	3438.0
butter	3841.0	4186.0	3657.0

We have the sales_df Dataframe. And we want to create a pivot table using customer and product. We already learned the Dataframe crosstab() transformation, and we have the code for the same given below.

Cmd 4

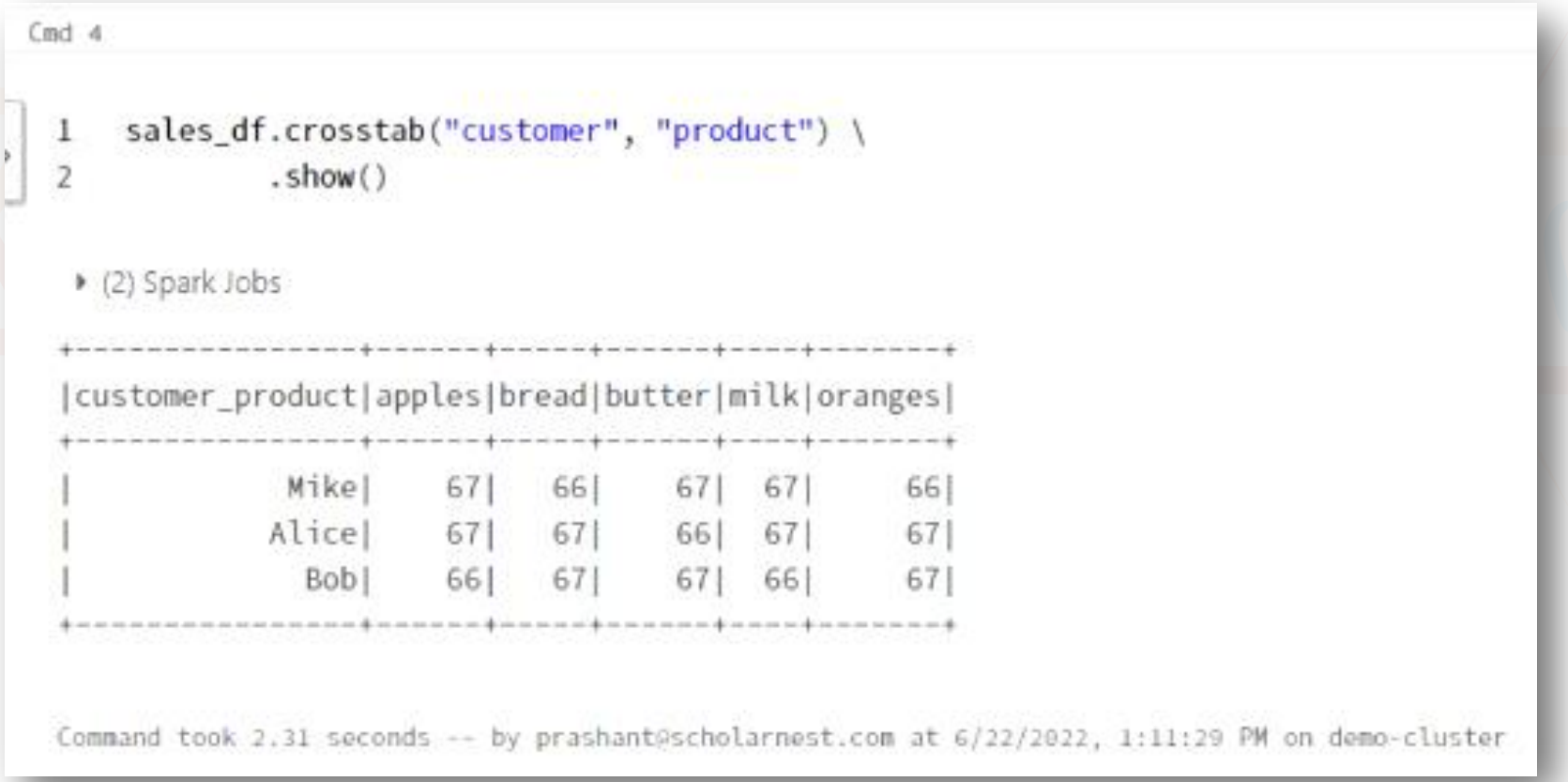
```
1   sales_df.crosstab("customer", "product") \
2           .show()
```

▸ (2) Spark Jobs

```
+----------------+------+-----+------+----+-------+
|customer_product|apples|bread|butter|milk|oranges|
+----------------+------+-----+------+----+-------+
|            Mike|    67|   66|    67|  67|     66|
|           Alice|    67|   67|    66|  67|     67|
|             Bob|    66|   67|    67|  66|     67|
+----------------+------+-----+------+----+-------+
```

Command took 2.31 seconds -- by prashant@scholarnest.com at 6/22/2022, 1:11:29 PM on demo-cluster

We got customer names listed here and the product names listed at the top.
So the crosstab() successfully creators the Pivot of the two-column values.
But what are the cells? These cells are the record count. You can read it like this. Mike purchased apples 67 times. Alice purchased bread 67 times. But we are not looking for the count. We want to see the dollar values. And that's the crosstab() limitation. Spark crosstab() gives you count. You cannot change the behaviour.



Spark also gives a Pivot() function. We want to pivot on customer and product.
Pivot is a grouping operation. So you must apply a group by an operation before you can apply the Pivot. Once grouped, you can apply the Pivot.
So I will groupBy() the customer and Pivot by the product. These two operations will create a row-column structure. Then I wanted to calculate sales value. So I will apply the formula in the agg() method.

Cmd 5

```
1  sales_df.groupBy("customer") \
2          .pivot("product") \
3          .agg(sf.expr("sum(quantity * unit_price)")) \
4          .show()
```

▸ (7) Spark Jobs

```
+--------+------+------+------+------+-------+
|customer|apples| bread|butter|  milk|oranges|
+--------+------+------+------+------+-------+
|     Bob|2366.0|1098.0|4186.0|3366.0| 1240.0|
|   Alice|2436.0| 960.0|3841.0|3042.0| 1360.0|
|    Mike|2408.0|1014.0|3657.0|3438.0| 1328.0|
+--------+------+------+------+------+-------+
```

Command took 2.49 seconds -- by prashant@scholarnest.com at 6/22/2022, 1:15:07 PM on demo-cluster

Now you can look at the results.
So your groupBy() column values are arranged vertically. The Pivot column values are arranged horizontally. The cells are the aggregate values.

Cmd 5

```
1  sales_df.groupBy("customer") \
2          .pivot("product") \
3          .agg(sf.expr("sum(quantity * unit_price)")) \
4          .show()
```

▸ (7) Spark Jobs

```
+--------+------+------+------+------+-------+
|customer|apples| bread|butter|  milk|oranges|
+--------+------+------+------+------+-------+
|     Bob|2366.0|1098.0|4186.0|3366.0| 1240.0|
|   Alice|2436.0| 960.0|3841.0|3042.0| 1360.0|
|    Mike|2408.0|1014.0|3657.0|3438.0| 1328.0|
+--------+------+------+------+------+-------+
```

Command took 2.49 seconds -- by prashant@scholarnest.com at 6/22/2022, 1:15:07 PM on demo-cluster

You can swap the customer and product values by swapping the groupBy() and Pivot() fields.
Now the products are arranged vertically, and customers are on the horizontal axis.
The cells are still showing the sum(quantity*unit_price).

Cmd 6

```
1  sales_df.groupBy("product") \
2          .pivot("customer") \
3          .agg(sf.expr("sum(quantity * unit_price)")) \
4          .show()
```

▸ (7) Spark Jobs

```
+-------+------+------+------+
|product| Alice|   Bob|  Mike|
+-------+------+------+------+
|oranges|1360.0|1240.0|1328.0|
|  bread| 960.0|1098.0|1014.0|
| apples|2436.0|2366.0|2408.0|
|   milk|3042.0|3366.0|3438.0|
| butter|3841.0|4186.0|3657.0|
+-------+------+------+------+
```

Command took 2.13 seconds -- by prashant@scholarnest.com at 6/22/2022, 1:16:09 PM on demo-cluster

You can limit the previous list. So I gave a list of item values in the pivot method.
Now the Pivot will restrict the values to only these three items, and you can see the same below.

Cmd 7

```
1  sales_df.groupBy("customer") \
2          .pivot("product", ["milk", "bread", "butter"]) \
3          .agg(sf.expr("sum(quantity * unit_price)")) \
4          .show()
```

▸ (3) Spark Jobs

```
+--------+------+------+------+
|customer|  milk| bread|butter|
+--------+------+------+------+
|     Bob|3366.0|1098.0|4186.0|
|   Alice|3042.0| 960.0|3841.0|
|    Mike|3438.0|1014.0|3657.0|
+--------+------+------+------+
```

Command took 0.94 seconds -- by prashant@scholarnest.com at 6/22/2022, 1:18:52 PM on demo-cluster

If you want to limit the customers, you can use the where() method.
And you can see that we have only Alice and Mike in the output below.

Cmd 7

```
1  sales_df.where("customer in ('Alice', 'Mike')") \
2          .groupBy("customer") \
3          .pivot("product", ["milk", "bread", "butter"]) \
4          .agg(sf.expr("sum(quantity * unit_price)")) \
5          .show()
```

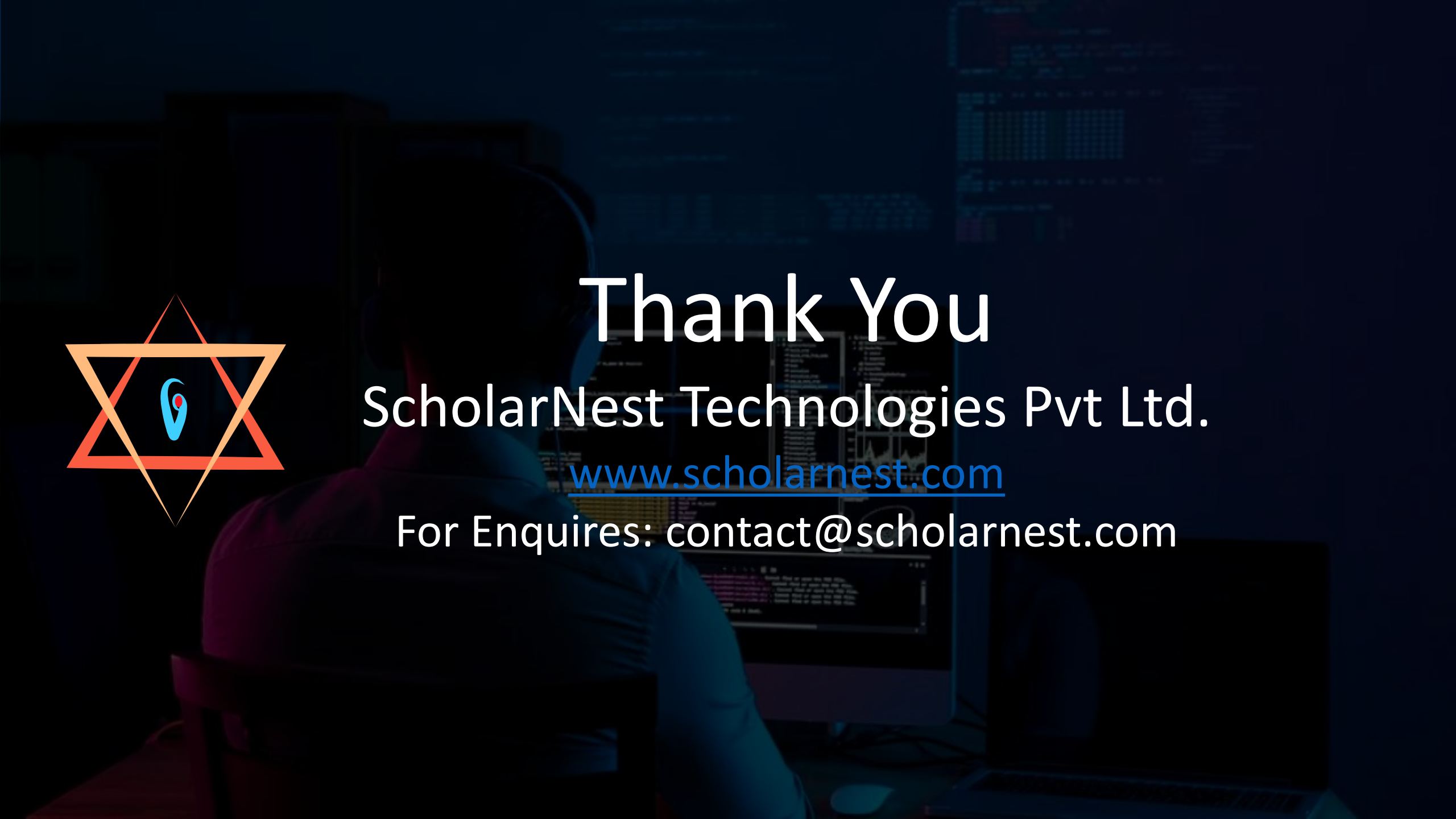
▸ (3) Spark Jobs

```
+--------+------+------+------+
|customer|  milk| bread|butter|
+--------+------+------+------+
|   Alice|3042.0| 960.0|3841.0|
|    Mike|3438.0|1014.0|3657.0|
+--------+------+------+------+
```

Command took 1.45 seconds -- by prashant@scholarnest.com at 6/22/2022, 1:20:17 PM on demo-cluster

Creating Pivot table is a groupBy() and Pivot() operation.
You can groupBy() on one column and Pivot on the other to create a pivot table.
Then you can calculate the aggregates like sum(), avg(), min(), max(), or whatever you want.

You can limit the pivot values by supplying a list of values.
And you can also apply the where condition before the groupBy() to restrict the values.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com