CSC 735 – Data Analytics

Chapter 7

Aggregations – Window Functions

Working with Date-Time Functions

- Spark built-in date-time functions fall into the three categories:
  - converting from one format to another
  - performing date-time calculations
  - extracting parts from a date or timestamp

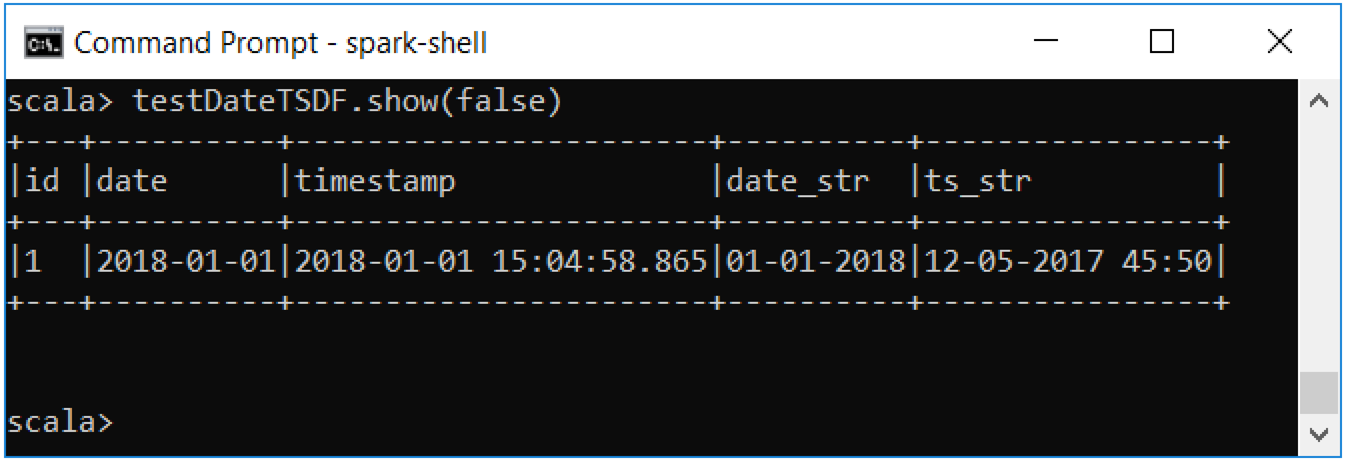
Date-Time Conversion Functions

- The default date format is yyyy-MM-dd HH:mm:ss
- If the date format of a date-timestamp column is different, you need to provide that pattern to the conversion function

Example

- Converting a date and timestamp in string type to Spark Date and Timestamp type

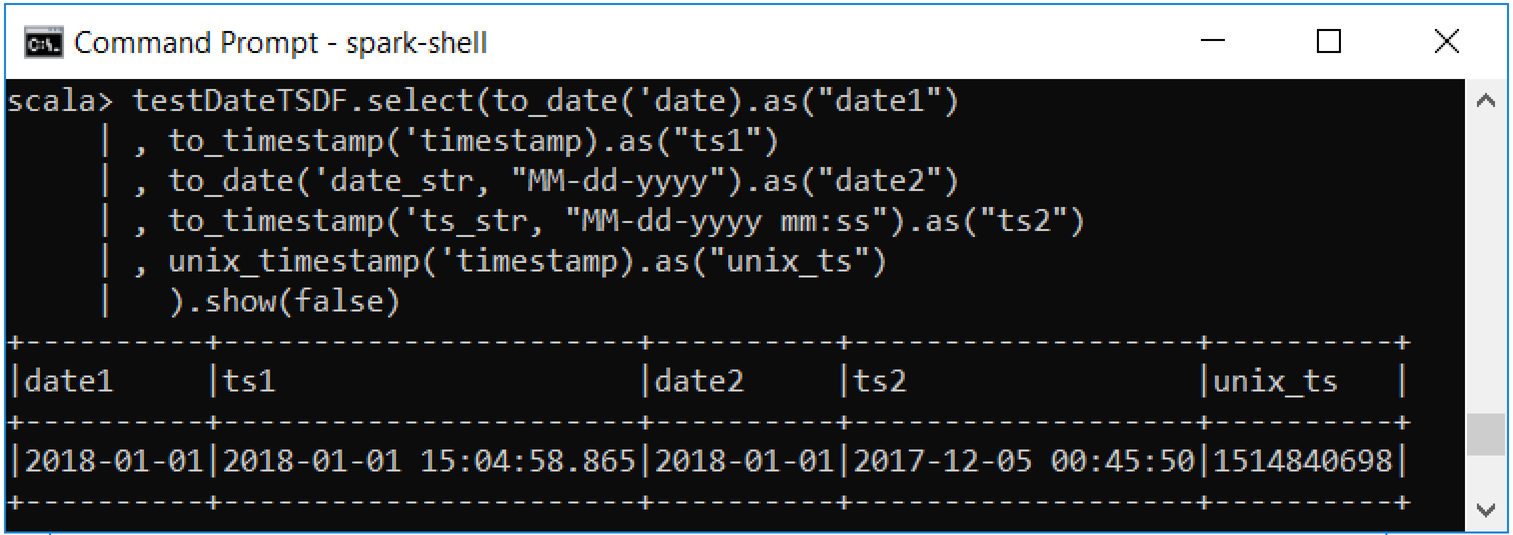
```
val testDateTSDF = Seq((1, "2018-01-01", "2018-01-01 15:04:58.865",
                                          "01-01-2018", "12-05-2017 45:50"))
                               .toDF("id", "date", "timestamp","date_str", "ts_str")

testDateTSDF.show(false)
```



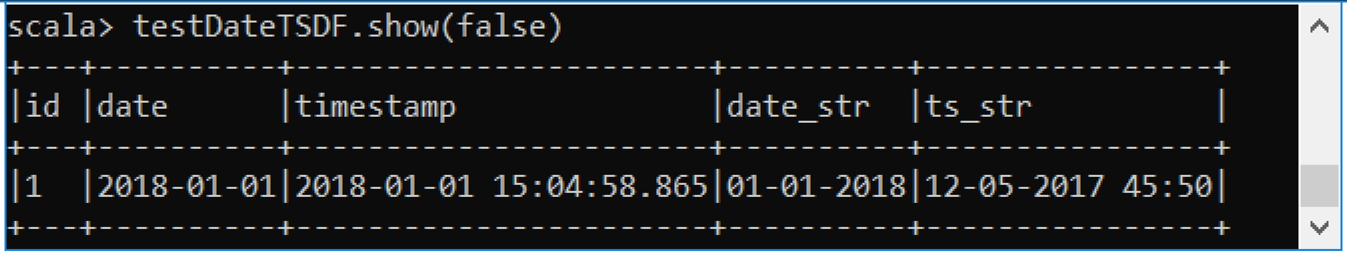
```
scala> testDateTSDF.show(false)
+---+----------+-----------------------+----------+----------------+
|id |date      |timestamp              |date_str  |ts_str          |
+---+----------+-----------------------+----------+----------------+
|1  |2018-01-01|2018-01-01 15:04:58.865|01-01-2018|12-05-2017 45:50|
+---+----------+-----------------------+----------+----------------+


scala>
```

Example (cont.)

```
testDateTSDF.select(to_date('date).as("date1")
              , to_timestamp('timestamp).as("ts1")
              , to_date('date_str, "MM-dd-yyyy").as("date2")
              , to_timestamp('ts_str, "MM-dd-yyyy mm:ss").as("ts2")
              , unix_timestamp('timestamp).as("unix_ts")
 ).show(false)
```



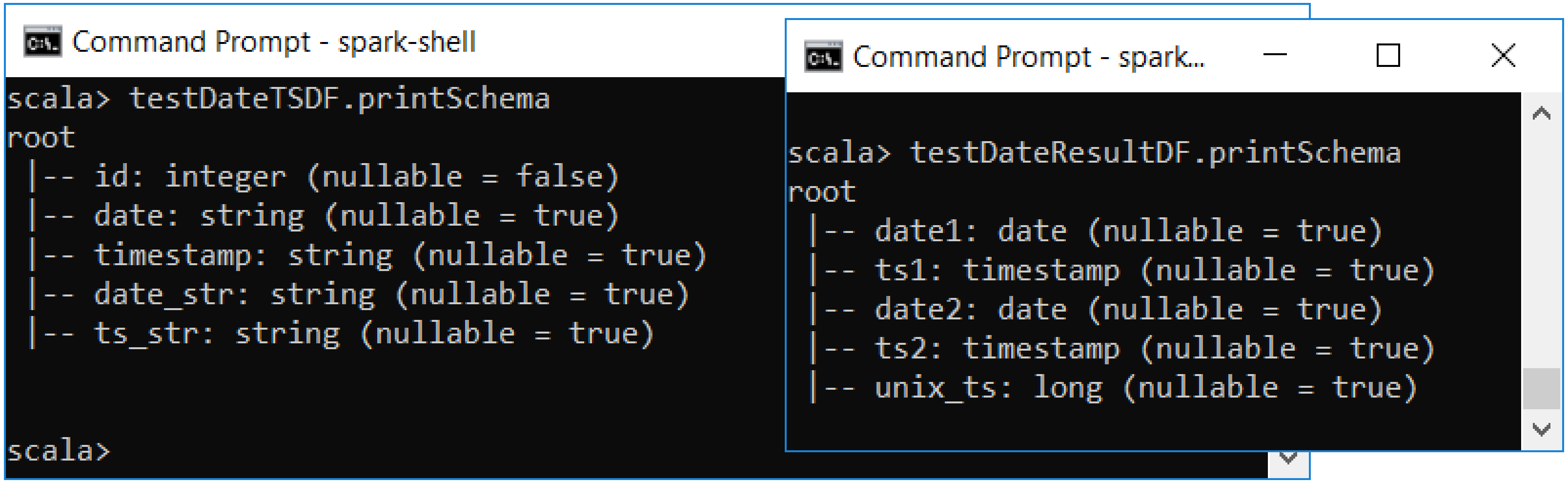
```
scala> testDateTSDF.select(to_date('date).as("date1")
     | , to_timestamp('timestamp).as("ts1")
     | , to_date('date_str, "MM-dd-yyyy").as("date2")
     | , to_timestamp('ts_str, "MM-dd-yyyy mm:ss").as("ts2")
     | , unix_timestamp('timestamp).as("unix_ts")
     |   ).show(false)
+----------+-----------------------+----------+-------------------+----------+
|date1     |ts1                    |date2     |ts2                |unix_ts   |
+----------+-----------------------+----------+-------------------+----------+
|2018-01-01|2018-01-01 15:04:58.865|2018-01-01|2017-12-05 00:45:50|1514840698|
+----------+-----------------------+----------+-------------------+----------+
```



```
scala> testDateTSDF.show(false)
+---+----------+-----------------------+----------+----------------+
|id |date      |timestamp              |date_str  |ts_str          |
+---+----------+-----------------------+----------+----------------+
|1  |2018-01-01|2018-01-01 15:04:58.865|01-01-2018|12-05-2017 45:50|
+---+----------+-----------------------+----------+----------------+
```

Example (cont.)

```
val testDateResultDF = testDateTSDF.select(to_date('date).as("date1")
              , to_timestamp('timestamp).as("ts1")
              , to_date('date_str, "MM-dd-yyyy").as("date2")
              , to_timestamp('ts_str, "MM-dd-yyyy mm:ss").as("ts2")
              , unix_timestamp('timestamp).as("unix_ts"))
testDateResultDF.printSchema
```



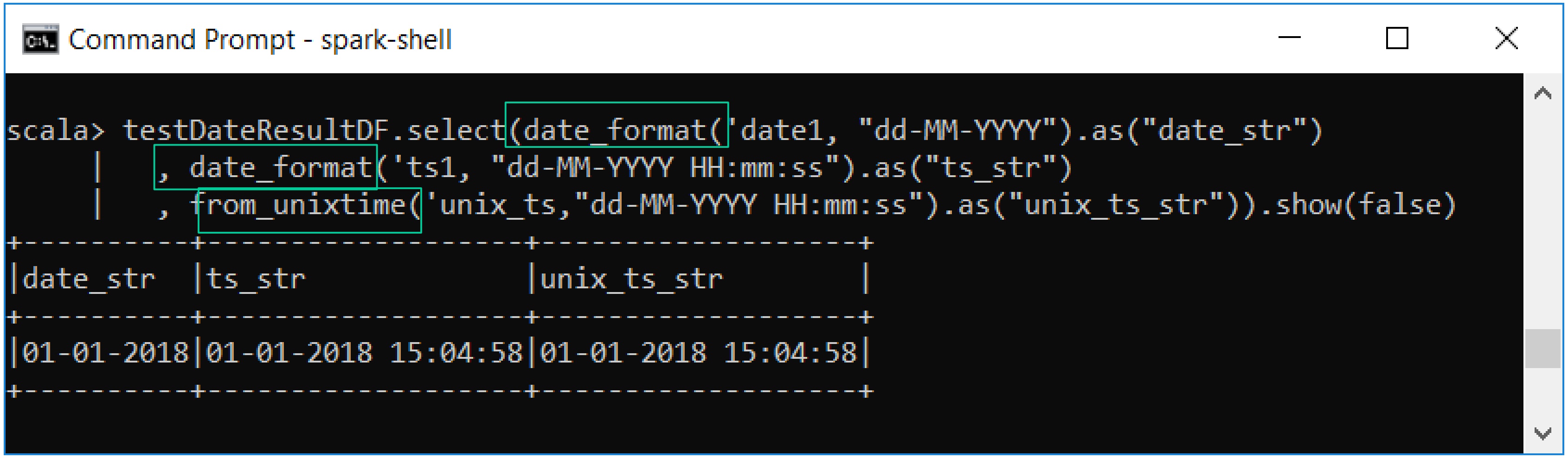
```
scala> testDateTSDF.printSchema
root
 |-- id: integer (nullable = false)
 |-- date: string (nullable = true)
 |-- timestamp: string (nullable = true)
 |-- date_str: string (nullable = true)
 |-- ts_str: string (nullable = true)


scala>
```

```
scala> testDateResultDF.printSchema
root
 |-- date1: date (nullable = true)
 |-- ts1: timestamp (nullable = true)
 |-- date2: date (nullable = true)
 |-- ts2: timestamp (nullable = true)
 |-- unix_ts: long (nullable = true)
```

Converting Date-Time to String

- Use the **date_format** function convert a date-timestamp to a string by using with a custom date format
- Use the **from_unixtime** function to convert a Unix timestamp from seconds to a string

```
 testDateResultDF.select(date_format('date1, "dd-MM-YYYY").as("date_str")
  , date_format('ts1, "dd-MM-YYYY HH:mm:ss").as("ts_str")
  , from_unixtime('unix_ts,"dd-MM-YYYY
HH:mm:ss").as("unix_ts_str")).show
```



```
scala> testDateResultDF.select(date_format('date1, "dd-MM-YYYY").as("date_str")
     |     , date_format('ts1, "dd-MM-YYYY HH:mm:ss").as("ts_str")
     |     , from_unixtime('unix_ts,"dd-MM-YYYY HH:mm:ss").as("unix_ts_str")).show(false)
+----------+-------------------+-------------------+
|date_str  |ts_str             |unix_ts_str        |
+----------+-------------------+-------------------+
|01-01-2018|01-01-2018 15:04:58|01-01-2018 15:04:58|
+----------+-------------------+-------------------+
```

Window Functions

- Read https://www.postgresql.org/docs/current/static/tutorial-window.html

Window Functions

- Read https://www.postgresql.org/docs/current/static/tutorial-window.html
- Window functions are used to compute some aggregation on a specific "window" of data

Window Functions

- Read https://www.postgresql.org/docs/current/static/tutorial-window.html
- Window functions are used to compute some aggregation on a specific "window" of data
- Use cases include calculating a moving average, a cumulative sum, and the rank of each row

Window Functions

- Read https://www.postgresql.org/docs/current/static/tutorial-window.html
- Window functions are used to compute some aggregation on a specific "window" of data
- Use cases include calculating a moving average, a cumulative sum, and the rank of each row
- The window specification determines which rows are used by the window function

Window Functions

- There are two main steps for working with window functions
- The first one is to **define a window** specification that defines a logical grouping of rows
- The second step is to **apply** a window function as needed

Window Specification

- import org.apache.spark.sql.expressions.Window
- The window specification involves three parts:

1. **partition by**: where you specify one or more columns to group the rows by

Window Specification

- import org.apache.spark.sql.expressions.Window
- The window specification involves three parts:

1. **partition by**: where you specify one or more columns to group the rows by
2. **order by:** defines how the rows should be ordered based on one or more columns

Window Specification

- import org.apache.spark.sql.expressions.Window
- The window specification involves three parts:

1. **partition by**: where you specify one or more columns to group the rows by
2. **order by:** defines how the rows should be ordered based on one or more columns
3. **frame**: it defines the boundary of the window with respect to the current row
   - The frame restricts which rows to be included when calculating a value for the current row

"Window" Frames

- A range of rows to include in a window frame can be specified using the row index or the actual value of the order by expression
- The **rowsBetween** and **rangeBetweeen** functions are used to define the range by row index and actual value, respectively

Window Functions

- Spark supports three kinds of window functions:
  - ranking functions
  - analytic functions
  - aggregate functions

Ranking Functions

Name	Description
rank	Returns the rank or order of rows within a frame based on some sorting order.
dense_rank	Similar to rank, but leaves no gaps in the ranks when there are ties.
percent_rank	Returns the relative rank of rows within a frame.
ntile(n)	Returns the ntile group ID in an ordered window partition. For example, if n is 4, the first quarter of the rows will get a value of 1, the second quarter of rows will get a value of 2, and so on.
row_number	Returns a sequential number starting with 1 within a frame

Analytic Functions

Name	Description
cume_dist	Returns the cumulative distribution of values with a frame. In other words, the fraction of rows that are below the current row
lag(col, offset)	Returns the value of the column that is offset rows before the current row.
lead(col, offset)	Returns the value of the column that is offset rows after the current row

Grouping Functions

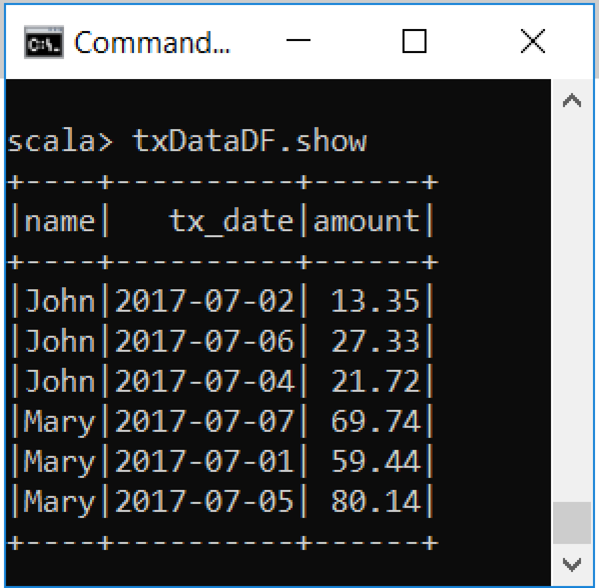
- We can use any aggregation functions as a window function

The Dataset

- The data frame contains shopping transaction data of two fictitious users, John and Mary

```
import org.apache.spark.sql.expressions.Window
val txDataDF = Seq(
  ("John", "2017-07-02", 13.35), ("John", "2017-07-06", 27.33),
  ("John", "2017-07-04", 21.72), ("Mary",  "2017-07-07", 69.74),
  ("Mary",  "2017-07-01", 59.44), ("Mary",  "2017-07-05", 80.14)
 ).toDF("name", "tx_date", "amount")
```

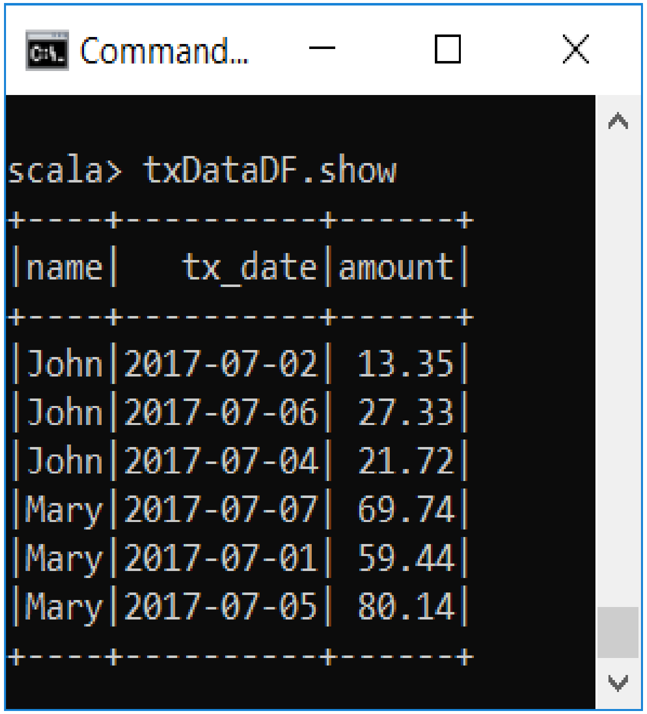
txData ≡ transaction Data



```
scala> txDataDF.show
+----+----------+------+
|name|   tx_date|amount|
+----+----------+------+
|John|2017-07-02| 13.35|
|John|2017-07-06| 27.33|
|John|2017-07-04| 21.72|
|Mary|2017-07-07| 69.74|
|Mary|2017-07-01| 59.44|
|Mary|2017-07-05| 80.14|
+----+----------+------+
```

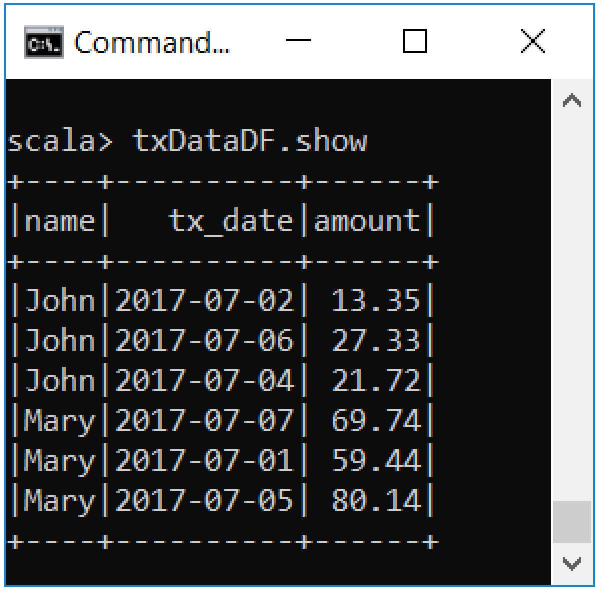
Data Analysis Tasks

1. For each user, what are the two highest transaction amounts?
2. What is the difference between the transaction amount of each user and their highest transaction amount?
3. What is the moving average transaction amount of each user?
4. What is the cumulative sum of the transaction amount of each user?



```
scala> txDataDF.show
+----+----------+------+
|name|   tx_date|amount|
+----+----------+------+
|John|2017-07-02| 13.35|
|John|2017-07-06| 27.33|
|John|2017-07-04| 21.72|
|Mary|2017-07-07| 69.74|
|Mary|2017-07-01| 59.44|
|Mary|2017-07-05| 80.14|
+----+----------+------+
```

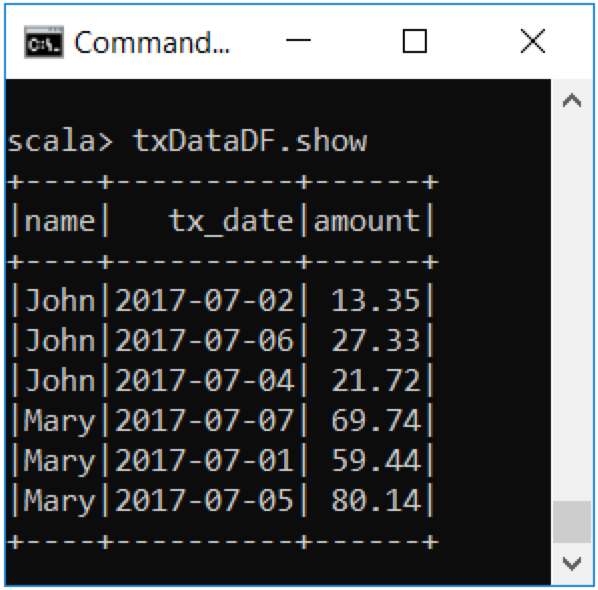
1. For each user, what are the two highest transaction amounts?



```
scala> txDataDF.show
+----+----------+------+
|name|   tx_date|amount|
+----+----------+------+
|John|2017-07-02| 13.35|
|John|2017-07-06| 27.33|
|John|2017-07-04| 21.72|
|Mary|2017-07-07| 69.74|
|Mary|2017-07-01| 59.44|
|Mary|2017-07-05| 80.14|
+----+----------+------+
```

1. For each user, what are the two highest transaction amounts?

- Idea: Apply the rank window function over a window specification that partitions the data by user and sorts it by the amount in descending order
- The rank function assigns a rank to each row based on the sorting order of each row in each frame

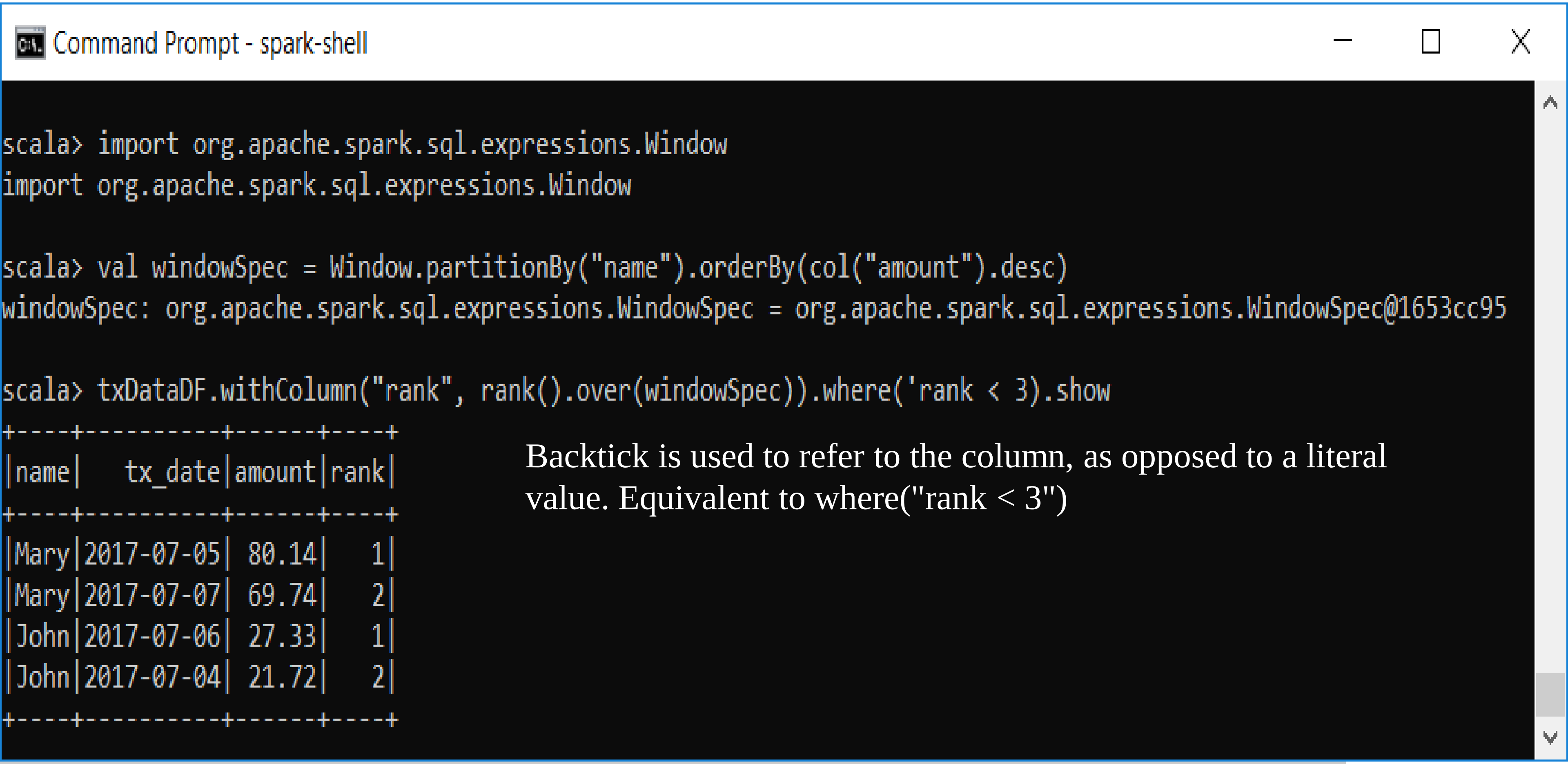


```
scala> txDataDF.show
+----+----------+------+
|name|   tx_date|amount|
+----+----------+------+
|John|2017-07-02| 13.35|
|John|2017-07-06| 27.33|
|John|2017-07-04| 21.72|
|Mary|2017-07-07| 69.74|
|Mary|2017-07-01| 59.44|
|Mary|2017-07-05| 80.14|
+----+----------+------+
```

1. For each user, what are the two highest transaction amounts?

```
import org.apache.spark.sql.expressions.Window
val windowSpec =  Window.partitionBy("name").orderBy(col("amount").desc)
txDataDF.withColumn("rank", rank().over(windowSpec)).where('rank < 3).show
```

1. For each user, what are the two highest transaction amounts?



```
scala> import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.expressions.Window

scala> val windowSpec = Window.partitionBy("name").orderBy(col("amount").desc)
windowSpec: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@1653cc95

scala> txDataDF.withColumn("rank", rank().over(windowSpec)).where('rank < 3).show
+----+----------+------+----+
|name|   tx_date|amount|rank|
+----+----------+------+----+
|Mary|2017-07-05| 80.14|   1|
|Mary|2017-07-07| 69.74|   2|
|John|2017-07-06| 27.33|   1|
|John|2017-07-04| 21.72|   2|
+----+----------+------+----+
```

Backtick is used to refer to the column, as opposed to a literal value. Equivalent to where("rank < 3")

```
import org.apache.spark.sql.expressions.Window
val windowSpec =  Window.partitionBy("name").orderBy(col("amount").desc)
txDataDF.withColumn("rank", rank().over(windowSpec)).where('rank < 3).show
```

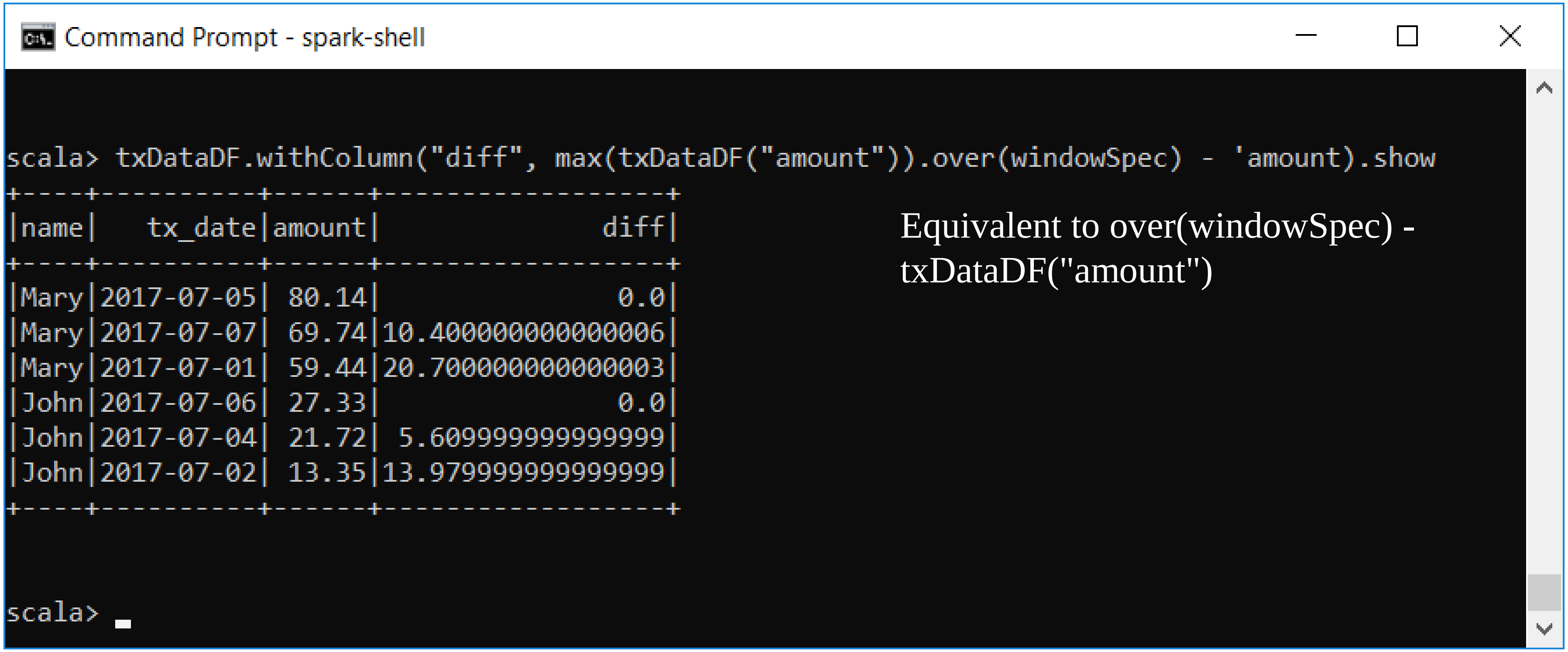
2. Find is the difference between the transaction amount of each user and their highest transaction amount?

- Idea: max(col(amount)) finds the max transaction amount for each person. Subtract from current tx amount

2. Find is the difference between the transaction amount of each user and their highest transaction amount?

- Idea: max(col(amount)) finds the max transaction amount for each person. Subtract from current tx amount

```
txDataDF.withColumn("diff",max(txDataDF("amount")).over(windowSpec) - 'amount).show
```



```
scala> txDataDF.withColumn("diff", max(txDataDF("amount")).over(windowSpec) - 'amount).show
+----+----------+------+------------------+
|name|   tx_date|amount|              diff|
+----+----------+------+------------------+
|Mary|2017-07-05| 80.14|               0.0|
|Mary|2017-07-07| 69.74|10.400000000000006|
|Mary|2017-07-01| 59.44|20.700000000000003|
|John|2017-07-06| 27.33|               0.0|
|John|2017-07-04| 21.72| 5.609999999999999|
|John|2017-07-02| 13.35|13.979999999999999|
+----+----------+------+------------------+

scala>
```

Equivalent to over(windowSpec) - txDataDF("amount")

3. What is the moving average transaction amount of each user?

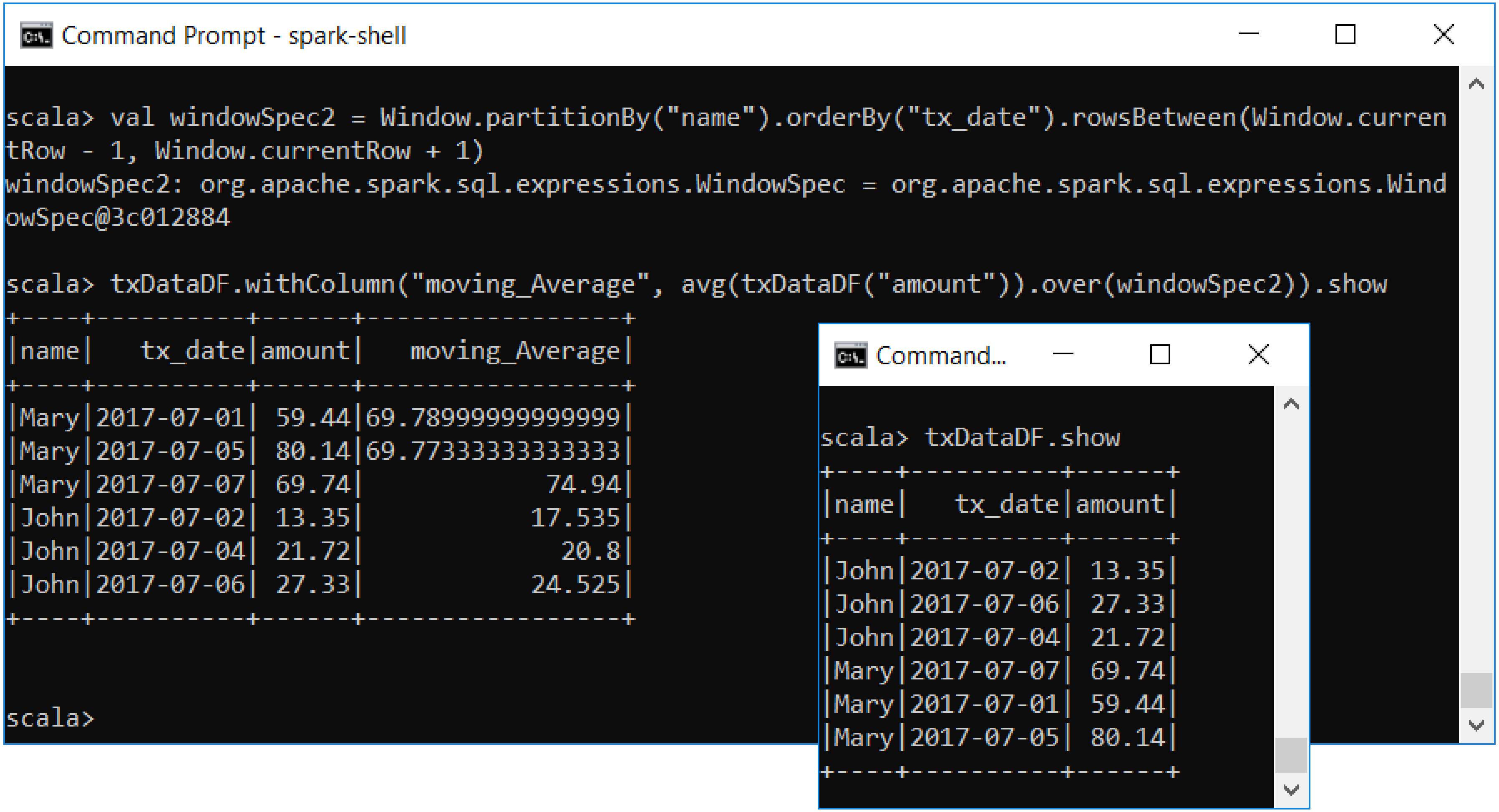
- Assumption: the moving average is over a window Frame of three rows, starting with the row before the current row and ending with the row after the current row. We order by tx_date as this is time series data.

3. What is the moving average transaction amount of each user?

- Assumption: the moving average is over a window Frame of three rows, starting with the row before the current row and ending with the row after the current row. We order by tx_date as this is time series data.

```
val windowSpec2 =
Window.partitionBy("name").orderBy("tx_date").rowsBetween(Window.currentRow - 1, Window.currentRow + 1)

txDataDF.withColumn("moving_Average",avg(txDataDF("amount"))
.over(windowSpec2)).show
```

3. What is the moving average transaction amount of each user?



```
scala> val windowSpec2 = Window.partitionBy("name").orderBy("tx_date").rowsBetween(Window.curren
tRow - 1, Window.currentRow + 1)
windowSpec2: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.Wind
owSpec@3c012884

scala> txDataDF.withColumn("moving_Average", avg(txDataDF("amount")).over(windowSpec2)).show
+----+----------+------+-----------------+
|name|   tx_date|amount|   moving_Average|
+----+----------+------+-----------------+
|Mary|2017-07-01| 59.44|69.78999999999999|
|Mary|2017-07-05| 80.14|69.77333333333333|
|Mary|2017-07-07| 69.74|            74.94|
|John|2017-07-02| 13.35|           17.535|
|John|2017-07-04| 21.72|             20.8|
|John|2017-07-06| 27.33|           24.525|
+----+----------+------+-----------------+


scala>
```

```
scala> txDataDF.show
+----+----------+------+
|name|   tx_date|amount|
+----+----------+------+
|John|2017-07-02| 13.35|
|John|2017-07-06| 27.33|
|John|2017-07-04| 21.72|
|Mary|2017-07-07| 69.74|
|Mary|2017-07-01| 59.44|
|Mary|2017-07-05| 80.14|
+----+----------+------+
```

4. What is the cumulative sum of the transaction amount of each user?

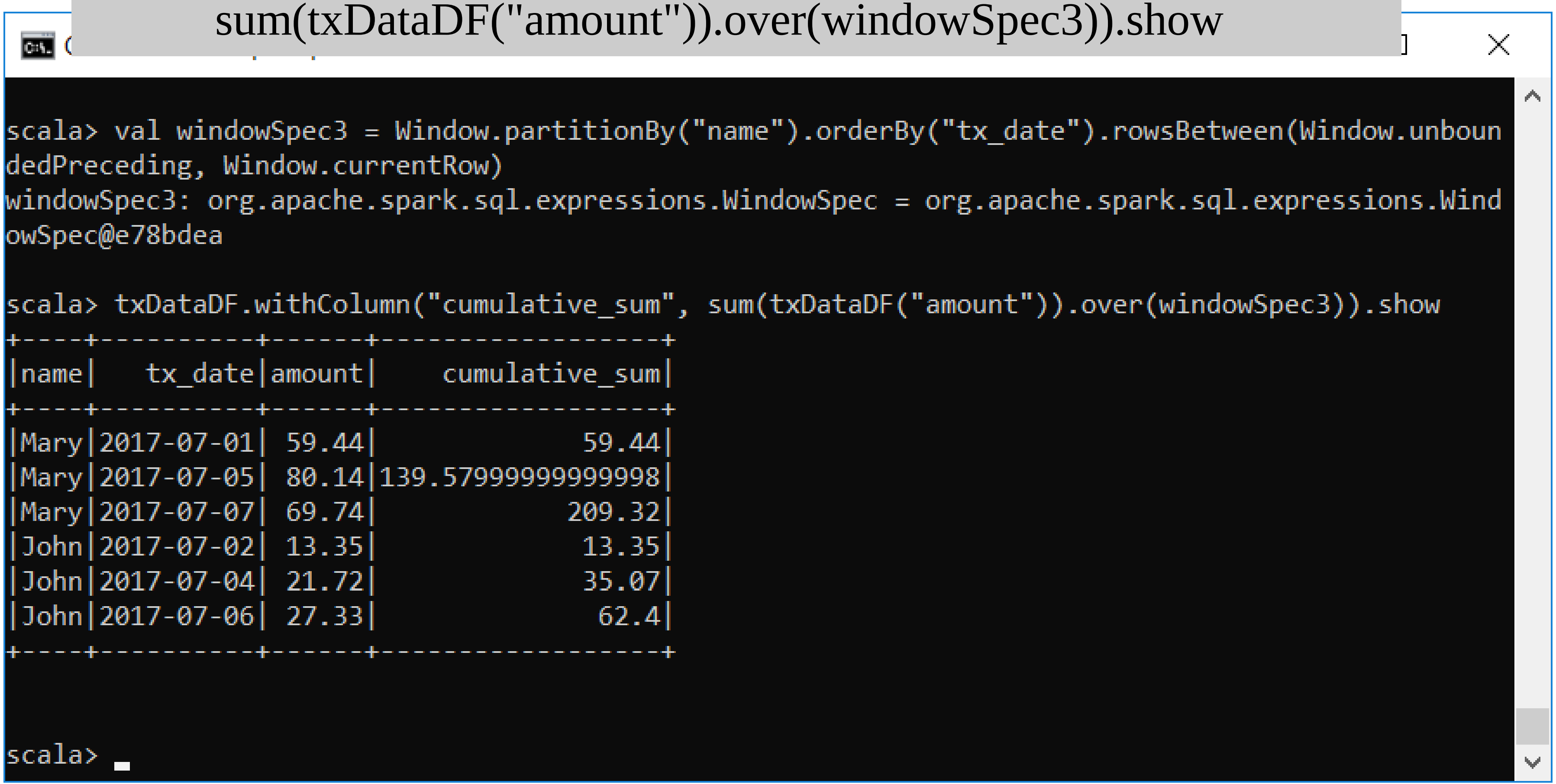
4. What is the cumulative sum of the transaction amount of each user?

- Idea: apply the sum function over a frame that consists of all the rows up to the current row

```
val windowSpec3 = Window.partitionBy("name").orderBy("tx_date")
  .rowsBetween(Window.unboundedPreceding, Window.currentRow)

txDataDF.withColumn("cumulative_sum",
          sum(txDataDF("amount")).over(windowSpec3)).show
```

4. What is the cumulative sum of the transaction amount of each user?

```
val windowSpec3 = Window.partitionBy("name").orderBy("tx_date")
  .rowsBetween(Window.unboundedPreceding, Window.currentRow)
txDataDF.withColumn("cumulative_sum",
          sum(txDataDF("amount")).over(windowSpec3)).show
```



```
scala> val windowSpec3 = Window.partitionBy("name").orderBy("tx_date").rowsBetween(Window.unboun
dedPreceding, Window.currentRow)
windowSpec3: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.Wind
owSpec@e78bdea

scala> txDataDF.withColumn("cumulative_sum", sum(txDataDF("amount")).over(windowSpec3)).show
+----+----------+------+------------------+
|name|   tx_date|amount|    cumulative_sum|
+----+----------+------+------------------+
|Mary|2017-07-01| 59.44|             59.44|
|Mary|2017-07-05| 80.14|139.57999999999998|
|Mary|2017-07-07| 69.74|            209.32|
|John|2017-07-02| 13.35|             13.35|
|John|2017-07-04| 21.72|             35.07|
|John|2017-07-06| 27.33|              62.4|
+----+----------+------+------------------+


scala>
```