07: Spark on Zeppelin – window functions in Scala

Posted on August 26, 2018

Pre-requisite: Docker is installed on your machine for Mac OS X (E.g. $ brew cask install docker) or Windows 10. Docker interview Q&As. This extends setting up Apache Zeppelin Notebook.

Q. What are the different types of functions in Spark SQL?
A. There are 4 types of functions:

1) Built-in functions: from org.apache.spark.sql.**functions** like to_date(Column e), to_utc_timestamp(Column e), etc. Take values

300+ Java Interview FAQs

300+ Java FAQs 🔥
16+ Java Key Areas Q&As
150+ Java Architect FAQs
80+ Java Code Quality Q&As
150+ Java Coding Q&As

300+ Big Data Interview FAQs

300+ Big Data FAQs 🔥
Tutorials - Big Data
TUT - ⏯ Starting Big Data
TUT - Starting Spark & Scala

from a single row as input and, and return single value for each input row.

2) UDFs (User Defined Functions): as the name implies defined by the users. Take values from a single row as input and, and return single value for each input row.

3) Aggregate functions: like SUM, MAX, AVG, MIN, etc, which operate on a group of rows and calculate a single return value for every group.

4) Window functions: are useful if you want to operate on a group of rows, but return a single value for every input row. For example, ranking a group of rows, calculating the cumulative total, etc.

Step 1: Pull this from the docker hub, and build the image with the following command.

```
$ docker pull apache/zeppelin:0.7.3
```

You can verify the image with the “docker images” command.

Step 2: Run the container with the above image.

```
$ docker run --rm -it -p 8080:8080 apache/zeppelin
```

Step 3: Open Zeppelin notebook via a web browser “http:localhost:8080”. Create a note book with “spark” as a default interpreter.

TUT - Starting with Python
TUT - Kafka
TUT - Pig
TUT - Apache Storm
TUT - Spark Scala on Zeppelin
TUT - Cloudera
TUT - Cloudera on Docker
TUT - File Formats
TUT - Spark on Docker
TUT - Flume
TUT - Hadoop (HDFS)
TUT - HBase (NoSQL)
TUT - Hive (SQL)
TUT - Hadoop & Spark
TUT - MapReduce
TUT - Spark and Scala
TUT - Spark & Java
TUT - PySpark on Databricks
TUT - Zookeeper

```
%spark


case class Employee (id: Integer, name: String, lo

val employees = Seq(
    Employee(1, "John", "USA", 50000.0),
    Employee(2, "Peter", "AU",60000.0),
    Employee(3, "Sam", "AU", 60000.0),
    Employee(4, "Susan", "USA", 50000.0),
    Employee(5, "David", "USA", 70000.0),
    Employee(6, "Elliot", "AU", 50000.0)
)

val employee_df = sc.parallelize(employees).toDF()
employee_df.show()

```

Output:

```
+---+------+--------+-------+
| id|  name|location| salary|
+---+------+--------+-------+
|  1|  John|     USA|50000.0|
|  2| Peter|      AU|60000.0|
|  3|   Sam|      AU|60000.0|
|  4| Susan|     USA|50000.0|
|  5| David|     USA|70000.0|
|  6|Elliot|      AU|50000.0|
+---+------+--------+-------+

```

Step 4: Let's now rank() them by **locations**.

```
%spark

import org.apache.spark.sql.expressions.Window

case class Employee (id: Integer, name: String, lo

val employees = Seq(
    Employee(1, "John", "USA", 50000.0),
    Employee(2, "Peter", "AU",60000.0),
    Employee(3, "Sam", "AU", 60000.0),
    Employee(4, "Susan", "USA", 50000.0),
    Employee(5, "David", "USA", 70000.0),
    Employee(6, "Elliot", "AU", 50000.0)
)

```

```
val employee_df = sc.parallelize(employees).toDF()
val windowspec = Window.partitionBy("location").or
val rankBySalaryForaLocation = rank().over(windows
val ranked_employees_df =  employee_df.withColumn(

ranked_employees_df.show()
```

Output:

```
+---+------+--------+-------+----+
| id|  name|location| salary|rank|
+---+------+--------+-------+----+
|  2| Peter|      AU|60000.0|   1|
|  3|   Sam|      AU|60000.0|   1|
|  6|Elliot|      AU|50000.0|   3|
|  5| David|     USA|70000.0|   1|
|  1|  John|     USA|50000.0|   2|
|  4| Susan|     USA|50000.0|   2|
+---+------+--------+-------+----+
```

As you can when there are two rank 1s, the next rank skips 2 and goes to 3. If you want sequentially, you can use a dense_rank.

```
val rankBySalaryForaLocation = dense_rank().over(w
```

```
+---+------+--------+-------+----+
| id|  name|location| salary|rank|
+---+------+--------+-------+----+
|  2| Peter|      AU|60000.0|   1|
|  3|   Sam|      AU|60000.0|   1|
|  6|Elliot|      AU|50000.0|   2|
|  5| David|     USA|70000.0|   1|
|  1|  John|     USA|50000.0|   2|
|  4| Susan|     USA|50000.0|   2|
+---+------+--------+-------+----+
```

What if you have ties, and still want to have sequential numbering so that you pick a single ranked 1. This is where row_number

```
+---+------+--------+-------+----+
| id|  name|location| salary|rank|
+---+------+--------+-------+----+
|  2| Peter|      AU|60000.0|   1|
|  3|   Sam|      AU|60000.0|   2|
|  6|Elliot|      AU|50000.0|   3|
|  5| David|     USA|70000.0|   1|
|  1|  John|     USA|50000.0|   2|
|  4| Susan|     USA|50000.0|   3|
+---+------+--------+-------+----+
```

What if you want print the max salary next to each salary?

```
%spark

import org.apache.spark.sql.expressions.Window

case class Employee (id: Integer, name: String, l

val employees = Seq(
    Employee(1, "John", "USA", 50000.0),
    Employee(2, "Peter", "AU",60000.0),
    Employee(3, "Sam", "AU",60000.0),
    Employee(4, "Susan", "USA", 50000.0),
    Employee(5, "David", "USA", 70000.0),
    Employee(6, "Elliot", "AU", 50000.0)
)

val employee_df = sc.parallelize(employees).toDF()

val windowspec = Window.partitionBy("location")

val max_employees_df =  employee_df.withColumn("m

max_employees_df.show()
```

Output:

```
+---+------+--------+-------+-------+
| id|  name|location| salary|    max|
+---+------+--------+-------+-------+
|  2| Peter|      AU|60000.0|60000.0|
|  3|   Sam|      AU|60000.0|60000.0|
|  6|Elliot|      AU|50000.0|60000.0|
|  1|  John|     USA|50000.0|70000.0|
```

```
|  4| Susan|     USA|50000.0|70000.0|
|  5| David|     USA|70000.0|70000.0|
+---+------+--------+-------+-------+
```

How about printing the difference in salaries by location?

```
%spark

import org.apache.spark.sql.expressions.Window

case class Employee (id: Integer, name: String, lo

val employees = Seq(
    Employee(1, "John", "USA", 50000.0),
    Employee(2, "Peter", "AU",60000.0),
    Employee(3, "Sam", "AU",60000.0),
    Employee(4, "Susan", "USA", 50000.0),
    Employee(5, "David", "USA", 70000.0),
    Employee(6, "Elliot", "AU", 50000.0)
)

val employee_df = sc.parallelize(employees).toDF()

val windowspec = Window.partitionBy("location").or

val salary_diff = max("salary").over(windowspec) -

val max_employees_df =  employee_df.withColumn("sa

max_employees_df.show()
```

Output:

```
+---+------+--------+-------+-----------+
| id|  name|location| salary|salary_diff|
+---+------+--------+-------+-----------+
|  2| Peter|      AU|60000.0|        0.0|
|  3|   Sam|      AU|60000.0|        0.0|
|  6|Elliot|      AU|50000.0|    10000.0|
|  5| David|     USA|70000.0|        0.0|
|  1|  John|     USA|50000.0|    20000.0|
|  4| Susan|     USA|50000.0|    20000.0|
+---+------+--------+-------+-----------+
```

Getting a **running total of salaries by location**:

```
%spark

import org.apache.spark.sql.expressions.Window

case class Employee (id: Integer, name: String, l

val employees = Seq(
    Employee(1, "John", "USA", 50000.0),
    Employee(2, "Peter", "AU",60000.0),
    Employee(3, "Sam", "AU",60000.0),
    Employee(4, "Susan", "USA", 50000.0),
    Employee(5, "David", "USA", 70000.0),
    Employee(6, "Elliot", "AU", 50000.0)
)

val employee_df = sc.parallelize(employees).toDF()

val windowspec = Window.partitionBy("location").o

val running_total = sum("salary").over(windowspec

val running_total_employees_df =  employee_df.wit

running_total_employees_df.show()

```

Output:

```
+---+------+--------+-------+-------------+
| id|  name|location| salary|running_total|
+---+------+--------+-------+-------------+
|  2| Peter|      AU|60000.0|      60000.0|
|  3|   Sam|      AU|60000.0|     120000.0|
|  6|Elliot|      AU|50000.0|     170000.0|
|  1|  John|     USA|50000.0|      50000.0|
|  4| Susan|     USA|50000.0|     100000.0|
|  5| David|     USA|70000.0|     170000.0|
+---+------+--------+-------+-------------+

```

Finally, getting the running total for all the employees just **remove** the “partitionBy”

```
val windowspec = Window.orderBy($"id")
```

```
2
```

Output:

```
+---+------+--------+-------+-------------+
| id|  name|location| salary|running_total|
+---+------+--------+-------+-------------+
|  1|  John|     USA|50000.0|      50000.0|
|  2| Peter|      AU|60000.0|     110000.0|
|  3|   Sam|      AU|60000.0|     170000.0|
|  4| Susan|     USA|50000.0|     220000.0|
|  5| David|     USA|70000.0|     290000.0|
|  6|Elliot|      AU|50000.0|     340000.0|
+---+------+--------+-------+-------------+

```

‹ 06: Spark on Zeppelin – RDD operation zipWithIndex

08: Spark on Zeppelin – convert DataFrames to RDD[Row] and RDD[Row] to DataFrame ›