

Java-Success.com

Prepare to fast-track, choose & go places with 800+ Java & Big Data Q&As with lots of code & diagrams.

[Home](#) [Why? ▾](#) [300+ Java FAQs ▾](#) [300+ Big Data FAQs ▾](#) [Courses ▾](#)

[👤 Membership ▾](#) [Your Career ▾](#)

[Home](#) › [bigdata-success.com](#) › [Tutorials - Big Data](#) › [TUT - Spark Scala on Zeppelin](#) ›

07: Spark on Zeppelin – window functions in Scala

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.

```
1 $ docker pull apache/zeppelin:0.7.3
2
```

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

Step 2: Run the container with the above image.

```
1 $ docker run --rm -it -p 8080:8080 apache/zeppelin
2
```

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

800+ Java Interview Q&As

300+ Core Java Q&As



300+ Enterprise Java Q&As



150+ Java Frameworks Q&As



120+ Companion Tech Q&As



Tutorials - Enterprise Java



```
1 %spark
2
3
4 case class Employee (id: Integer, name: String, location: String, salary: Double)
5
6 val employees = Seq(
7   Employee(1, "John", "USA", 50000.0),
8   Employee(2, "Peter", "AU", 60000.0),
9   Employee(3, "Sam", "AU", 60000.0),
10  Employee(4, "Susan", "USA", 50000.0),
11  Employee(5, "David", "USA", 70000.0),
12  Employee(6, "Elliot", "AU", 50000.0)
13 )
14
15 val employee_df = sc.parallelize(employees).toDF()
16 employee_df.show()
17
```

Output:

```
1 +---+-----+-----+-----+
2 | id|  name|location| salary|
3 +---+-----+-----+-----+
4 |  1|  John|     USA|50000.0|
5 |  2| Peter|     AU|60000.0|
6 |  3|  Sam|     AU|60000.0|
7 |  4| Susan|     USA|50000.0|
8 |  5| David|     USA|70000.0|
9 |  6| Elliot|     AU|50000.0|
10 +---+-----+-----+-----+
11
```

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

```
1 %spark
2
3 import org.apache.spark.sql.expressions.Window
4
5 case class Employee (id: Integer, name: String, location: String, salary: Double)
6
7 val employees = Seq(
8   Employee(1, "John", "USA", 50000.0),
9   Employee(2, "Peter", "AU", 60000.0),
10  Employee(3, "Sam", "AU", 60000.0),
11  Employee(4, "Susan", "USA", 50000.0),
12  Employee(5, "David", "USA", 70000.0),
13  Employee(6, "Elliot", "AU", 50000.0)
14 )
15
```

```

16 val employee_df = sc.parallelize(employees).toDF()
17 val windowSpec = Window.partitionBy("location").orderBy("salary")
18 val rankBySalaryForALocation = rank().over(windowSpec)
19 val ranked_employees_df = employee_df.withColumn("rank", rankBySalaryForALocation)
20
21 ranked_employees_df.show()
22

```

Output:

```

1 +---+-----+-----+-----+-----+
2 | id|  name|location| salary|rank|
3 +---+-----+-----+-----+-----+
4 |  2| Peter|      AU|60000.0|  1|
5 |  3|  Sam|      AU|60000.0|  1|
6 |  6|Elliot|      AU|50000.0|  3|
7 |  5| David|     USA|70000.0|  1|
8 |  1|  John|     USA|50000.0|  2|
9 |  4| Susan|     USA|50000.0|  2|
10 +---+-----+-----+-----+-----+
11

```

As you can see 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**.

```

1 val rankBySalaryForALocation = dense_rank().over(windowSpec)

```

```

1 +---+-----+-----+-----+-----+
2 | id|  name|location| salary|rank|
3 +---+-----+-----+-----+-----+
4 |  2| Peter|      AU|60000.0|  1|
5 |  3|  Sam|      AU|60000.0|  1|
6 |  6|Elliot|      AU|50000.0|  2|
7 |  5| David|     USA|70000.0|  1|
8 |  1|  John|     USA|50000.0|  2|
9 |  4| Susan|     USA|50000.0|  2|
10 +---+-----+-----+-----+-----+
11

```

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**

```

1  +---+-----+-----+-----+-----+
2  | id|  name|location| salary|rank|
3  +---+-----+-----+-----+-----+
4  |  2| Peter|      AU|60000.0|  1|
5  |  3|  Sam|      AU|60000.0|  2|
6  |  6|Elliot|      AU|50000.0|  3|
7  |  5| David|     USA|70000.0|  1|
8  |  1|  John|     USA|50000.0|  2|
9  |  4| Susan|     USA|50000.0|  3|
10 +---+-----+-----+-----+-----+
11

```

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

```

1  %spark
2
3  import org.apache.spark.sql.expressions.Window
4
5  case class Employee (id: Integer, name: String, location: String, salary: Double)
6
7  val employees = Seq(
8    Employee(1, "John", "USA", 50000.0),
9    Employee(2, "Peter", "AU", 60000.0),
10   Employee(3, "Sam", "AU", 60000.0),
11   Employee(4, "Susan", "USA", 50000.0),
12   Employee(5, "David", "USA", 70000.0),
13   Employee(6, "Elliot", "AU", 50000.0)
14 )
15
16 val employee_df = sc.parallelize(employees).toDF()
17
18 val windowSpec = Window.partitionBy("location")
19
20 val max_employees_df = employee_df.withColumn("max_salary", max(salary) over windowSpec)
21
22 max_employees_df.show()
23

```

Output:

```

1  +---+-----+-----+-----+-----+
2  | id|  name|location| salary|max|
3  +---+-----+-----+-----+-----+
4  |  2| Peter|      AU|60000.0|60000.0|
5  |  3|  Sam|      AU|60000.0|60000.0|
6  |  6|Elliot|      AU|50000.0|60000.0|
7  |  1|  John|     USA|50000.0|70000.0|

```

8		4		Susan		USA		50000.0		70000.0	
9		5		David		USA		70000.0		70000.0	
10	+---+---+---+---+---+---+---+---+---+---+---+---										
11											

How about printing the difference in salaries by location?

```

1 %spark
2
3 import org.apache.spark.sql.expressions.Window
4
5 case class Employee (id: Integer, name: String, location: String, salary: Double)
6
7 val employees = Seq(
8   Employee(1, "John", "USA", 50000.0),
9   Employee(2, "Peter", "AU", 60000.0),
10  Employee(3, "Sam", "AU", 60000.0),
11  Employee(4, "Susan", "USA", 50000.0),
12  Employee(5, "David", "USA", 70000.0),
13  Employee(6, "Elliot", "AU", 50000.0)
14 )
15
16 val employee_df = sc.parallelize(employees).toDF()
17
18 val windowSpec = Window.partitionBy("location").orderBy("salary")
19
20 val salary_diff = max("salary").over(windowSpec)
21
22 val max_employees_df = employee_df.withColumn("salary_diff", salary_diff)
23
24 max_employees_df.show()
25

```

Output:

1	+---+---+---+---+---+---+---+---+---+---+---+---										
2		id		name		location		salary		salary_diff	
3	+---+---+---+---+---+---+---+---+---+---+---+---										
4		2		Peter		AU		60000.0		0.0	
5		3		Sam		AU		60000.0		0.0	
6		6		Elliot		AU		50000.0		10000.0	
7		5		David		USA		70000.0		0.0	
8		1		John		USA		50000.0		20000.0	
9		4		Susan		USA		50000.0		20000.0	
10	+---+---+---+---+---+---+---+---+---+---+---+---										
11											

Getting a running total of salaries by location:

```
1 %spark
2
3 import org.apache.spark.sql.expressions.Window
4
5 case class Employee (id: Integer, name: String, location: String, salary: Double)
6
7 val employees = Seq(
8     Employee(1, "John", "USA", 50000.0),
9     Employee(2, "Peter", "AU", 60000.0),
10    Employee(3, "Sam", "AU", 60000.0),
11    Employee(4, "Susan", "USA", 50000.0),
12    Employee(5, "David", "USA", 70000.0),
13    Employee(6, "Elliot", "AU", 50000.0)
14 )
15
16 val employee_df = sc.parallelize(employees).toDF()
17
18 val windowSpec = Window.partitionBy("location").orderBy($"id")
19
20 val running_total = sum("salary").over(windowSpec)
21
22 val running_total_employees_df = employee_df.withColumn("running_total", running_total)
23
24 running_total_employees_df.show()
25
```

Output:

```
1 +---+-----+-----+-----+-----+
2 | id|  name|location| salary|running_total|
3 +---+-----+-----+-----+-----+
4 |  2| Peter|      AU|60000.0|    60000.0|
5 |  3|   Sam|      AU|60000.0|   120000.0|
6 |  6|Elliot|      AU|50000.0|   170000.0|
7 |  1|  John|     USA|50000.0|    50000.0|
8 |  4| Susan|     USA|50000.0|   100000.0|
9 |  5| David|     USA|70000.0|   170000.0|
10 +---+-----+-----+-----+-----+
11
```

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

```
1 val windowSpec = Window.orderBy($"id")
```

Output:

```

1  +---+---+---+---+---+---+
2  | id|  name|location| salary|running_total|
3  +---+---+---+---+---+---+
4  |  1|  John|    USA|50000.0|    50000.0|
5  |  2| Peter|    AU|60000.0|   110000.0|
6  |  3|   Sam|    AU|60000.0|   170000.0|
7  |  4| Susan|    USA|50000.0|   220000.0|
8  |  5| David|    USA|70000.0|   290000.0|
9  |  6| Elliot|    AU|50000.0|   340000.0|
10 +---+---+---+---+---+---+
11

```

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

Disclaimer

The contents in this Java-Success are copyrighted and from EmpoweringTech Pty Ltd. The EmpoweringTech Pty Ltd has the right to correct or enhance the current content without any prior notice. These are general advice only, and one needs to take his/her own circumstances into consideration. The EmpoweringTech Pty Ltd will not be held liable for any damages caused or alleged to be caused either directly or indirectly by these materials and resources. Any trademarked names or labels used in this blog remain the property of their respective trademark owners. Links to external sites do not imply endorsement of the linked-to sites. [Privacy Policy](#)