# PySpark – DataFrame Window Functions #

### Introduction to Window Functions #

Window functions in PySpark allow for performing calculations across a set of rows that are somehow related to the current row. Unlike aggregate functions, which return a single result for a group of rows, window functions return multiple rows while retaining individual row values. These functions are particularly useful for computing **running totals**, **rankings**, **moving averages**, and more.

Window functions in PySpark are performed using the `pyspark.sql.Window` class, which defines a window specification that controls how the rows are related.

### Key Window Functions in PySpark #

1. **Ranking Functions**: `rank()`, `dense_rank()`, `row_number()`
2. **Analytical Functions**: `lead()`, `lag()`, `ntile()`
3. **Aggregate Functions over Windows**: `sum()`, `avg()`, `min()`, `max()`

## Components of a Window Specification #

- **Partitioning**: Defines how to divide the data into partitions (similar to the `GROUP BY` clause in SQL).
- **Ordering**: Defines the order of the rows within each partition.
- **Frame Specification**: Defines the range of rows within the partition to be considered in the calculation.

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql.functions import col, row_number, rank, dense_rank, lead, lag, sum, avg

# Initialize a Spark session
spark = SparkSession.builder.appName("Window Functions Example").getOrCreate()

# Sample data
data = [
    ("Alice", "Sales", 5000),
    ("Bob", "Sales", 4800),
    ("John", "HR", 6000),
    ("Jane", "HR", 5500),
    ("Sam", "Sales", 4500),
    ("Nina", "HR", 5900),
    ("Tom", "IT", 6200),
    ("Rob", "IT", 5800)
]

# Create DataFrame
df = spark.createDataFrame(data, ["Name", "Department", "Salary"])
df.show()
```

### 1. Ranking Functions #

#### a) `row_number()` #

`row_number()` assigns a unique, sequential number to each row within a partition of a DataFrame, starting at 1.

```
# Define Window specification
window_spec = Window.partitionBy("Department").orderBy(col("Salary").desc())

# Apply row_number() window function
df.withColumn("row_number", row_number().over(window_spec)).show()
```

**Explanation**:

- `partitionBy("Department")`: Data is partitioned by the “Department” column.
- `orderBy(col("Salary").desc())`: Within each department, rows are ordered by “Salary” in descending order.

#### b) `rank()` #

`rank()` assigns ranks to rows within a partition, with gaps in rank values in case of ties.

```
df.withColumn("rank", rank().over(window_spec)).show()
```

#### c) `dense_rank()` #

`dense_rank()` is similar to `rank()`, but without gaps between rank values in case of ties.

```
df.withColumn("dense_rank", dense_rank().over(window_spec)).show()
```

### 2. Analytical Functions #

#### a) `lead()` #

`lead()` provides access to the next row’s data within the partition, useful for comparing values in consecutive rows.

```
df.withColumn("lead_salary", lead("Salary", 1).over(window_spec)).show()
```

**Explanation**:

- The `lead("Salary", 1)` retrieves the salary of the next row within the partition.

#### b) `lag()` #

`lag()` is similar to `lead()` but accesses the previous row’s data within the partition.

```
df.withColumn("lag_salary", lag("Salary", 1).over(window_spec)).show()
```

### 3. Aggregate Functions Over Windows #

#### a) `sum()` #

Calculating the cumulative or running total within a window:

```
df.withColumn("cumulative_sum", sum("Salary").over(window_spec)).show()
```

#### b) `avg()` #

Calculating a running average:

```
df.withColumn("running_avg", avg("Salary").over(window_spec)).show()
```

### 4. Frame Specification #

In addition to `partitionBy()` and `orderBy()`, you can specify a range or frame of rows that influence the calculation. You can define a **window frame** with the `rowsBetween()` or `rangeBetween()` method.

```
# Define a window specification with a frame of 2 previous rows
window_spec_frame = Window.partitionBy("Department").orderBy(col("Salary").desc()).rowsBetween(-2, 0)

# Apply a sum() function over the window with frame specification
df.withColumn("rolling_sum", sum("Salary").over(window_spec_frame)).show()
```

**Explanation**:

- `rowsBetween(-2, 0)`: This specifies that the window should include the current row and the 2 rows preceding it.

### Real-World Example: Employee Ranking by Salary #

Let’s put everything together into a more realistic example where we want to rank employees based on their salaries, calculate the cumulative sum, and compute the next and previous salaries for comparison.

```
# Define a more complex window spec with ordering and frame
window_spec = Window.partitionBy("Department").orderBy(col("Salary").desc())

# Apply multiple window functions
df.withColumn("row_number", row_number().over(window_spec)) \
  .withColumn("rank", rank().over(window_spec)) \
  .withColumn("dense_rank", dense_rank().over(window_spec)) \
  .withColumn("lead_salary", lead("Salary", 1).over(window_spec)) \
  .withColumn("lag_salary", lag("Salary", 1).over(window_spec)) \
  .withColumn("cumulative_sum", sum("Salary").over(window_spec)) \
  .withColumn("running_avg", avg("Salary").over(window_spec)) \
  .show()
```

### Key Window Functions in Action: #

- `row_number()`: Assigns a unique sequential number for each row in each department based on salary.
- `rank()`: Assigns rank to employees in each department with gaps for ties.
- `dense_rank()`: Ranks without gaps for ties.
- `lead()` and `lag()`: Access the next and previous salaries.
- `sum()`: Calculates a cumulative sum of salaries in each department.
- `avg()`: Calculates a running average salary within each department.

### Key Takeaways: #

- **Partitioning and Ordering**: Window functions work on partitions of data defined by `partitionBy()` and ordered by `orderBy()`.
- **Ranking and Analytics**: Functions like `rank()`, `lead()`, and `lag()` help in ranking and performing analytical operations across rows.
- **Frame Specification**: Define a custom window of rows with `rowsBetween()` or `rangeBetween()` to create more specific calculations.