Spark Chapter 7
Nhóm 3

Mục lục:

I. GIỚI THIỆU NHÓM 3 VÀ CHỦ ĐỀ

a) Giới thiệu nhóm 3

Phạm Ngọc Hải _ 21002139 _ Nhóm trưởng _ Mức độ hoàn thành 100/100

Lương Đức Anh _ 21002117 _ Thành viên _ Mức độ hoàn thành 100/100

Hoàng Đình Duy _ 21002125 _ Thành viên _ Mức độ hoàn thành 95/100

Nguyễn Văn Thắng _ 21002175 _ Thành viên _ Mức độ hoàn thành 100/100

b) Giới thiệu chủ đề

Tổng hợp (Aggregations) là hành động thu thập một cái gì đó lại với nhau và là nền tảng của phân tích dữ liệu lớn. Trong một tập hợp, bạn sẽ chỉ định 1 khóa hoặc nhóm khóa và hàm tổng hợp giúp chỉ định cách bạn nên chuyển đổi một hoặc nhiều cột. Hàm này phải tạo ra một kết quả cho mỗi nhóm, được cung cấp nhiều giá trị đầu vào. Khả năng tổng hợp của Spark rất phức tạp và kỹ càng, với nhiều trường hợp sử dụng và khả năng khác nhau. Nói chung, bạn sử dụng Tổng hợp (Aggregations) để tóm tắt dữ liệu số thường bằng một số nhóm. Ngoài ra, với Spark, bạn có thể tổng hợp bất kỳ loại giá trị nào vào một mảng, list hoặc map.

Ngoài việc làm việc với bất kỳ loại giá trị nào, Spark còn cho phép chúng ta tạo các loại nhóm sau:

- Nhóm đơn giản nhất là chỉ tóm tắt một DataFrame hoàn chỉnh bằng cách thực hiện tổng hợp trong một câu lệnh chọn.
- Một “group by” cho phép bạn chỉ định một hoặc nhiều khóa cũng như một hoặc nhiều hàm tổng hợp để chuyển đổi các cột giá trị.
- Một “window” cung cấp cho bạn khả năng chỉ định một hoặc nhiều khóa cũng như một hoặc nhiều hàm tổng hợp để chuyển đổi các cột giá trị. Tuy nhiên, các hàng được nhập vào hàm có liên quan nào đó đến hàng hiện tại.
- Một "grouping set" có thể sử dụng để tổng hợp ở nhiều cấp độ khác nhau. Grouping sets có sẵn dưới dạng nguyên thủy trong SQL và thông qua các rollup và các cube trong DataFrames.
- Một "roll up" giúp bạn có thể chỉ định một hoặc nhiều khóa cũng như một hoặc nhiều hàm tổng hợp để chuyển đổi các cột giá trị, các cột này sẽ được tóm tắt theo thứ bậc.
- Một "cube" cho phép bạn chỉ định một hoặc nhiều khóa cũng như một hoặc nhiều hàm tổng hợp để biến đổi các cột giá trị, các cột này sẽ được tóm tắt trên tất cả các tổ hợp cột.

Mỗi nhóm trả về `RelationalGroupedDataset`

II. CÁC HÀM TỔNG HỢP

Để bắt đầu làm việc với PySpark, trước hết ta phải khởi tạo một phiên làm việc của PySpark:

```
from pyspark.sql import SparkSession
spark = SparkSession\
    .builder\
    .appName("PythonLR")\
    .getOrCreate()
spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY")
```

Tiếp đến chúng ta sẽ đọc dữ liệu mua hàng, dữ liệu sẽ gồm các thuộc tính sau:

#	**Thuộc tính**	**Mô tả**
1	InvoiceNo	Số hoá đơn
2	StockCode	Mã chứng khoán
3	Description	Mô tả hàng hoá
4	Quantity	Số lượng hàng hoá
5	InvoiceDate	Ngày lập hoá đơn
6	UnitPrice	Đơn giá
7	CustomerID	Mã khách hàng
8	Country	Quốc gia

```
df = spark.read.format("csv")\
  .option("header", "true")\
  .option("inferSchema", "true")\
  .load("online-retail-dataset.csv")\
  .coalesce(5)
df.cache()
df.createOrReplaceTempView("dfTable")
```

Đây là một phần dữ liệu để bạn có thể hiểu đầu ra của một số hàm sắp tới:

```
+---------+---------+--------------------+--------+--------------+---------+----------+--------------+
|InvoiceNo|StockCode|         Description|Quantity|   InvoiceDate|UnitPrice|CustomerID|       Country|
+---------+---------+--------------------+--------+--------------+---------+----------+--------------+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/2010 8:26|     2.55|     17850|United Kingdom|
|   536365|    71053| WHITE METAL LANTERN|       6|12/1/2010 8:26|     3.39|     17850|United Kingdom|
|   536365|   84406B|CREAM CUPID HEART...|       8|12/1/2010 8:26|     2.75|     17850|United Kingdom|
|   536365|   84029G|KNITTED UNION FLA...|       6|12/1/2010 8:26|     3.39|     17850|United Kingdom|
|   536365|   84029E|RED WOOLLY HOTTIE...|       6|12/1/2010 8:26|     3.39|     17850|United Kingdom|
+---------+---------+--------------------+--------+--------------+---------+----------+--------------+
```

Như đã đề cập về các tập hợp cơ bản có thể áp dụng cho toàn bộ DataFrame, đơn giản nhất là phương thức `count`:

```
df.count()
Out: 541909
```

Phép đếm thực sự là một hành động trái ngược với một phép biến đổi, vì vậy kết quả được trả về ngay lập tức. Bạn có thể sử dụng `count` để biết kích thước của tập dữ liệu nhưng một mẫu phổ biến khác là sử dụng nó để lưu trữ toàn bộ DataFrame trong bộ nhớ như chúng ta vừa làm trong ví dụ trên.

Phương thức này hơi khác một chút vì nó tồn tại dưới dạng một phương thức (trong trường hợp này) trái ngược với một hàm và được đánh giá ngay khi gặp phải. Trong phần sau, ta sẽ sử dụng `count` như một lazy function.

Tất cả các phép tổng hợp đều tồn tại dưới dạng hàm, ngoài ra các trường hợp đặc biệt có thể xuất hiện trong các DataFrame.

1. count

Hàm đầu tiên là `count`, khác với ví dụ trên thì nó sẽ thực hiện như một phép biến đổi thay vì một hành động. Trong trường hợp này, chúng ta có thể thực hiện một trong hai việc: đếm một cột cụ thể hoặc tất cả các cột bằng cách sử dụng `count(*)` hoặc `count(1)` để biểu thị rằng chúng ta muốn đếm mọi hàng, như minh hoạ trong ví dụ dưới đây:

```
from pyspark.sql.functions import count
df.select(count("StockCode")).show()
Out:
+----------------+
|count(StockCode)|
+----------------+
|          541909|
+----------------+
```

2. countDistinct

Đôi khi, tổng số không thích hợp; đúng hơn, đó là số nhóm duy nhất mà bạn muốn. Để lấy được con số này, bạn có thể dùng hàm `countDistinct`. Điều này phù hợp hơn với các cột riêng lẻ:

```
from pyspark.sql.functions import countDistinct
df.select(countDistinct("StockCode")).show()
```

```
Out:
+-------------------------+
|count(DISTINCT StockCode)|
+-------------------------+
|                     4070|
+-------------------------+
```

3. approx_count_distinct

Thông thường, khi làm việc với các dữ liệu lớn và số lượng khác biệt chính xác là không thích hợp. Đôi khi, một phép tính gần đúng ở mức chính xác nhất định sẽ hoạt động tốt, để làm điều đó, ta sử dụng hàm `approx_count_distinct`

```
from pyspark.sql.functions import approx_count_distinct
df.select(approx_count_distinct("StockCode", 0.1)).show()
```

```
Out:
+--------------------------------+
|approx_count_distinct(StockCode)|
+--------------------------------+
|                            3364|
+--------------------------------+
```

Bạn sẽ để ý rằng `approx_count_distinct` lấy một đối truyền vào khác mà cho phép bạn chỉ định sai số ước tính tối đa cho phép. Trong trường hợp này, với mức ý nghĩa 0.1, ta nhận được câu trả lời khá xa so với `countDistinct` nhưng tốc độ nhanh hơn. Điều này sẽ giúp tăng hiệu suất với những bộ dữ liệu lớn hơn.

4. first và last

Bạn có thể lấy các giá trị đầu hoặc cuối của một DataFrame bằng cách sử dụng hai hàm cùng tên này:

```
from pyspark.sql.functions import first, last
df.select(first("StockCode"), last("StockCode")).show()
```

```
Out:
+---------------+--------------+
|first(StockCode)|last(StockCode)|
+---------------+--------------+
|         85123A|         22138|
+---------------+--------------+
```

5. min và max

Để lấy ra giá trị nhỏ nhất và lớn nhất từ DataFrame, ta sử dụng hàm `min`, `max`:

```
from pyspark.sql.functions import min, max
df.select(min("Quantity"), max("Quantity")).show()
```

```
Out:
+-------------+-------------+
|min(Quantity)|max(Quantity)|
+-------------+-------------+
|       -80995|        80995|
+-------------+-------------+
```

6. sum

Một công việc đơn giản khác là cộng các giá trị trong một hàng bằng hàm `sum`:

```
from pyspark.sql.functions import sum
df.select(sum("Quantity")).show()
```

```
Out:
+-------------+
|sum(Quantity)|
+-------------+
|      5176450|
+-------------+
```

7. sumDistinct

Bạn có thể cộng một tập các giá trị khác nhau bằng cách sử dụng hàm `sumDistinct`:

```
from pyspark.sql.functions import sumDistinct
df.select(sumDistinct("Quantity")).show()
```

```
Out:
+----------------------+
|sum(DISTINCT Quantity)|
+----------------------+
|                 29310|
+----------------------+
```

8. avg

Một trong những số liệu quan trọng trong phân tích dữ liệu chính là giá trị trung bình. Trong Spark, hàm hỗ trợ tính toán giá trị trung bình là `avg`.

```
from pyspark.sql.functions import sum, count, avg, expr
df.select(
    count("Quantity").alias("total_transactions"),
    sum("Quantity").alias("total_purchases"),
    avg("Quantity").alias("avg_purchases"),
    expr("mean(Quantity)").alias("mean_purchases"))\
  .selectExpr(
    "total_purchases/total_transactions",
    "avg_purchases",
    "mean_purchases").show()
```

```
Out:
+------------------------------------+---------------+----------------+
|(total_purchases / total_transactions)|   avg_purchases|  mean_purchases|
+------------------------------------+---------------+----------------+
|                    9.55224954743324|9.55224954743324|9.55224954743324|
+------------------------------------+---------------+----------------+
```

9. Phương sai và độ lệch chuẩn

Đi kèm với giá trị trung bình là phương sai và độ lệch chuẩn. Đây là cả hai phép đo sự lan truyền của dữ liệu xung quanh giá trị trung bình. Phương sai là giá trị trung bình của bình phương sự khác biệt so với giá trị trung bình và độ lệch chuẩn là căn bậc hai của phương sai. Bạn có thể tính toán những thứ này trong Spark bằng cách sử dụng các chức năng tương ứng của chúng. Tuy nhiên, một điều cần lưu ý là Spark có cả công thức cho độ lệch chuẩn mẫu cũng như công thức cho độ lệch chuẩn tổng thể. Đây là những công thức thống kê khác nhau về cơ bản và chúng ta cần phân biệt giữa chúng. Theo mặc định, Spark thực hiện công thức cho độ lệch chuẩn hoặc phương sai mẫu nếu bạn sử dụng hàm `variance` hoặc hàm `stddev`.

Bạn cũng có thể chỉ định những điều này một cách rõ ràng hoặc tham khảo độ lệch chuẩn hoặc phương sai tổng thể:

```
from pyspark.sql.functions import var_pop, stddev_pop
from pyspark.sql.functions import var_samp, stddev_samp
df.select(var_pop("Quantity"), var_samp("Quantity"),
  stddev_pop("Quantity"), stddev_samp("Quantity")).show()
Out:
+-----------------+------------------+--------------------+---------------------+
|var_pop(Quantity)|var_samp(Quantity)|stddev_pop(Quantity)|stddev_samp(Quantity)|
+-----------------+------------------+--------------------+---------------------+
| 47559.3036466091|  47559.3914092988|  218.08095663447807|   218.08115785023426|
+-----------------+------------------+--------------------+---------------------+
```

10. Độ xiên (skewness) và độ nhọn (kurtosis)

Độ xiên và độ nhọn đều là phép đo các điểm cực trị trong dữ liệu của bạn. Độ xiên đo lường sự bất đối xứng của các giá trị trong dữ liệu của bạn xung quanh giá trị trung bình, trong khi độ nhọn là thước đo phần đuôi của dữ liệu. Cả hai điều này đều có liên quan cụ thể khi lập mô hình dữ liệu của bạn dưới dạng phân phối xác suất của một biến ngẫu nhiên. Bạn có thể tính toán những điều này bằng cách sử dụng các chức năng:

```
from pyspark.sql.functions import skewness, kurtosis
df.select(skewness("Quantity"), kurtosis("Quantity")).show()
Out:
+-------------------+------------------+
|  skewness(Quantity)|kurtosis(Quantity)|
+-------------------+------------------+
|-0.26407557610529564|119768.05495534712|
+-------------------+------------------+
```

11. Hiệp phương sai (covariance) và tương quan (correlation)

Chúng ta đã thảo luận về tập hợp cột đơn, nhưng một số hàm so sánh tương tác của các giá trị trong hai cột khác nhau với nhau. Hai trong số các hàm này lần lượt là `cov` và `corr`, tương ứng với hiệp phương sai và tương quan. Tương quan đo lường hệ số tương quan Pearson, được chia tỷ lệ giữa –1 và +1. Hiệp phương sai được chia tỷ lệ theo các đầu vào trong dữ liệu.

Giống như hàm `var`, hiệp phương sai có thể được tính dưới dạng hiệp phương sai mẫu hoặc hiệp phương sai tổng thể. Do đó, điều quan trọng là phải chỉ định công thức bạn muốn sử dụng. Tương quan không có khái niệm về điều này và do đó không có tính toán cho dân số hoặc mẫu. Đây là cách chúng hoạt động:

```
from pyspark.sql.functions import corr, covar_pop, covar_samp
df.select(corr("InvoiceNo", "Quantity"), covar_samp("InvoiceNo",
"Quantity"),
    covar_pop("InvoiceNo", "Quantity")).show()
```

```
Out:
+-------------------------+-----------------------------+-----------------------------+
|corr(InvoiceNo, Quantity)|covar_samp(InvoiceNo, Quantity)|covar_pop(InvoiceNo, Quantity)|
+-------------------------+-----------------------------+-----------------------------+
|     4.912186085642775E-4|            1052.7280543915654|            1052.7260778754612|
+-------------------------+-----------------------------+-----------------------------+
```

12. Tổng hợp thành các loại phức tạp

Trong Spark, bạn có thể thực hiện tổng hợp không chỉ các giá trị số bằng công thức, bạn cũng có thể thực hiện chúng trên các loại phức tạp. Ví dụ: chúng ta có thể thu thập danh sách các giá trị có trong một cột nhất định hoặc chỉ các giá trị duy nhất bằng cách thu thập thành một tập hợp.

Bạn có thể sử dụng điều này để thực hiện một số quyền truy cập có lập trình hơn sau này trong quá trình giải quyết hoặc chuyển toàn bộ bộ sưu tập vào một hàm do người dùng xác định (UDF):

```
from pyspark.sql.functions import collect_set, collect_list
df.agg(collect_set("Country"), collect_list("Country")).show()
```

```
Out:
+--------------------+---------------------+
|collect_set(Country)|collect_list(Country)|
+--------------------+---------------------+
|[Portugal, Italy,...| [United Kingdom, ...|
+--------------------+---------------------+
```

III. GOM NHÓM

Một nhiệm vụ phổ biến hơn là thực hiện các phép tính dựa trên các nhóm trong dữ liệu. Điều này thường được thực hiện trên dữ liệu phân loại mà chúng tôi nhóm dữ liệu của mình trên một cột và thực hiện một số phép tính trên các cột khác nằm trong nhóm đó.

Cách tốt nhất để giải thích điều này là bắt đầu thực hiện một số nhóm. Đầu tiên sẽ là đếm, giống như chúng ta đã làm trước đây. Chúng tôi sẽ nhóm theo từng số hóa đơn duy nhất và nhận số lượng mặt hàng trên hóa đơn đó. Lưu ý rằng điều này trả về một DataFrame khác và là lazily performed.

Chúng tôi thực hiện việc phân nhóm này theo hai giai đoạn. Trước tiên, chúng tôi chỉ định (các) cột mà chúng tôi muốn nhóm trên đó, sau đó chúng tôi chỉ định (các) tập hợp. Bước đầu tiên trả về một RelationalGroupedDataset và bước thứ hai trả về một DataFrame.

Như đã đề cập, chúng tôi có thể chỉ định bất kỳ số lượng cột nào mà chúng tôi muốn nhóm:

```
df.groupBy("InvoiceNo", "CustomerId").count().show()
```

```
Out:
+---------+----------+-----+
|InvoiceNo|CustomerId|count|
+---------+----------+-----+
|   536846|     14573|   76|
|   537026|     12395|   12|
|   537883|     14437|    5|
|   538068|     17978|   12|
|   538279|     14952|    7|
|   538800|     16458|   10|
|   538942|     17346|   12|
|  C539947|     13854|    1|
|   540096|     13253|   16|
|   540530|     14755|   27|
|   541225|     14099|   19|
|   541978|     13551|    4|
|   542093|     17677|   16|
|   543188|     12567|   63|
|   543590|     17377|   19|
|  C543757|     13115|    1|
|  C544318|     12989|    1|
|   544578|     12365|    1|
|   536596|      null|    6|
|   537252|      null|    1|
+---------+----------+-----+
only showing top 20 rows
```

1. Nhóm với biểu thức

Thay vì chuyển hàm dưới dạng một biểu thức vào câu lệnh `select`, ta có thể chỉ định chúng trong `agg`.

Điều này giúp bạn có thể chuyển vào các biểu thức tùy ý mà chỉ cần có một số tập hợp được chỉ định. Bạn thậm chí có thể thực hiện những việc như đặt tên (`alias`) cho một cột sau khi đã chuyển đổi để sử dụng sau này:

```
from pyspark.sql.functions import count
df.groupBy("InvoiceNo").agg(
    count("Quantity").alias("quan"),
    expr("count(Quantity)")).show()
```

```
Out:
+---------+----+---------------+
|InvoiceNo|quan|count(Quantity)|
+---------+----+---------------+
|   536596|   6|              6|
|   536938|  14|             14|
|   537252|   1|              1|
|   537691|  20|             20|
|   538041|   1|              1|
+---------+----+---------------+
only showing top 5 rows
```

2. Nhóm với Maps

Đôi khi, việc chỉ định các phép biến đổi của bạn dưới dạng một chuỗi Maps có khóa là cột và giá trị là hàm tổng hợp (dưới dạng chuỗi) mà bạn muốn thực hiện. Bạn cũng có thể sử dụng lại nhiều tên cột nếu bạn chỉ định chúng nội tuyến:

```
df.groupBy("InvoiceNo").agg(expr("avg(Quantity)"),expr("stddev_pop(Quantity)
"))\
  .show()
```

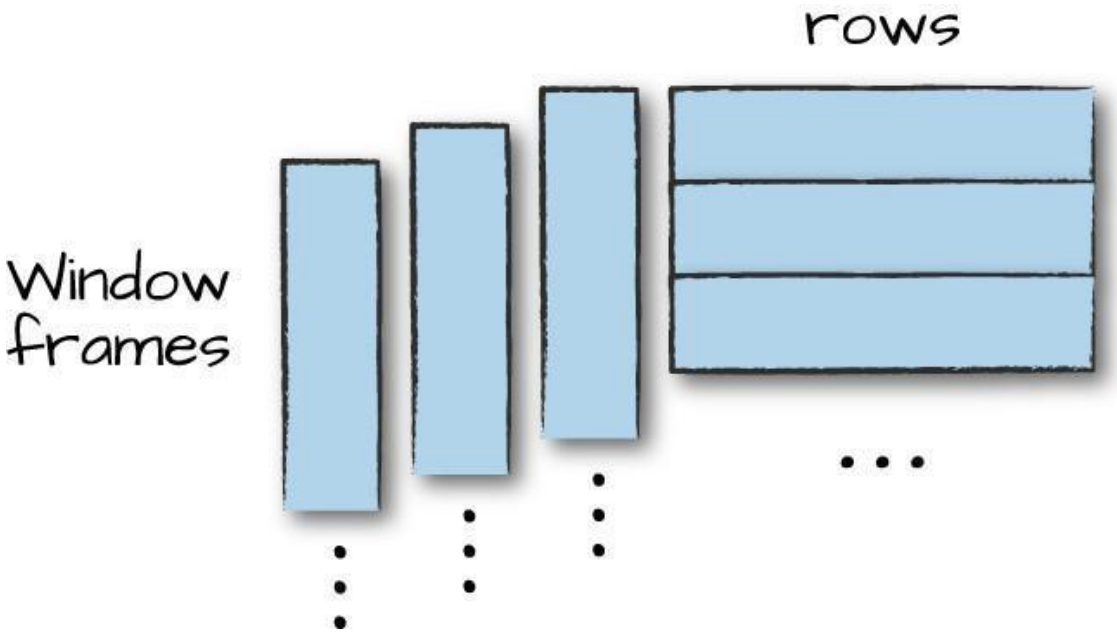
```
Out:
+---------+------------------+--------------------+
|InvoiceNo|     avg(Quantity)|stddev_pop(Quantity)|
+---------+------------------+--------------------+
|   536596|               1.5|  1.1180339887498947|
|   536938|33.142857142857146|  20.698023172885524|
|   537252|              31.0|                 0.0|
|   537691|              8.15|   5.597097462078001|
|   538041|              30.0|                 0.0|
+---------+------------------+--------------------+
only showing top 5 rows
```

IV. HÀM WINDOW

Bạn cũng có thể sử dụng các *hàm window* để thực hiện một số tập hợp duy nhất bằng cách tính toán một số tập hợp trên một "cửa sổ" dữ liệu cụ thể mà bạn xác định bằng cách sử dụng tham chiếu đến dữ liệu hiện tại. Window này xác định những hàng nào sẽ được chuyển vào hàm này. Bây giờ điều này hơi trừu tượng và có thể tương tự như tiêu chuẩn theo nhóm, vì vậy hãy phân biệt chúng thêm một chút.

Một *group-by* lấy dữ liệu và mỗi hàng chỉ có thể đi vào một nhóm. Hàm window tính toán giá trị trả về cho mọi hàng đầu vào của bảng dựa trên một nhóm hàng, được gọi là khung. Mỗi hàng có thể rơi vào một hoặc nhiều khung. Một trường hợp sử dụng phổ biến là xem xét mức trung bình luân phiên của một số giá trị mà mỗi hàng đại diện cho một ngày. Nếu bạn làm điều này, mỗi hàng sẽ có bảy khung hình khác nhau. Chúng tôi sẽ đề cập đến việc xác định khung sau một chút, nhưng để bạn tham khảo, Spark hỗ trợ ba loại hàm window: hàm xếp hạng, hàm phân tích và hàm tổng hợp.

Hình minh họa cách một hàng nhất định có thể rơi vào nhiều khung:



Để biểu lộ, chúng ta sẽ thêm cột date sẽ chuyển invoice date vào một cột mà chỉ có thông tin về ngày:

```
from pyspark.sql.functions import col, to_date
dfWithDate = df.withColumn("date", to_date(col("InvoiceDate"), "MM/d/yyyy H:mm"))
dfWithDate.createOrReplaceTempView("dfWithDate")
```

Out:

```
+---------+---------+--------------------+--------+--------------+---------+----------+--------------+---------+
|InvoiceNo|StockCode|         Description|Quantity|   InvoiceDate|UnitPrice|CustomerID|       Country|     date|
+---------+---------+--------------------+--------+--------------+---------+----------+--------------+---------+
|   536365|   85123A|WHITE HANGING HEA...|       6|12/1/2010 8:26|     2.55|     17850|United Kingdom|2010-12-01|
|   536365|    71053| WHITE METAL LANTERN|       6|12/1/2010 8:26|     3.39|     17850|United Kingdom|2010-12-01|
|   536365|   84406B|CREAM CUPID HEART...|       8|12/1/2010 8:26|     2.75|     17850|United Kingdom|2010-12-01|
|   536365|   84029G|KNITTED UNION FLA...|       6|12/1/2010 8:26|     3.39|     17850|United Kingdom|2010-12-01|
|   536365|   84029E|RED WOOLLY HOTTIE...|       6|12/1/2010 8:26|     3.39|     17850|United Kingdom|2010-12-01|
+---------+---------+--------------------+--------+--------------+---------+----------+--------------+---------+
only showing top 5 rows
```

Bước đầu tiên đối với hàm window là tạo mô tả cho window. Lưu ý rằng `partition by` không liên quan đến khái niệm sơ đồ phân vùng mà chúng ta đã đề cập. Nó chỉ là một khái niệm tương tự mô tả cách chúng ta sẽ chia nhóm của mình. Thứ tự xác định thứ tự trong một phân vùng nhất định và cuối cùng, đặc tả khung (câu lệnh `rowsBetween`) cho biết những hàng nào sẽ được đưa vào khung dựa trên tham chiếu của nó tới hàng đầu vào hiện tại. Trong ví dụ sau, chúng ta xem xét tất cả các hàng trước đó cho đến hàng hiện tại:

```
from pyspark.sql.window import Window
from pyspark.sql.functions import desc
windowSpec = Window\
  .partitionBy("CustomerId", "date")\
  .orderBy(desc("Quantity"))\
  .rowsBetween(Window.unboundedPreceding, Window.currentRow)
```

Bây giờ chúng ta sử dụng chức năng tổng hợp để tìm hiểu thêm về từng khách hàng cụ thể. Một ví dụ là thiết lập số lượng mua tối đa mọi lúc. Để trả lời câu hỏi này, chúng ta sử dụng các hàm tổng hợp giống như chúng ta đã thấy trước đó bằng cách chuyển tên cột hoặc biểu thức. Ngoài ra, chúng tôi chỉ ra đặc tả window xác định khung dữ liệu nào mà chức năng này sẽ áp dụng:

```
from pyspark.sql.functions import max
maxPurchaseQuantity = max(col("Quantity")).over(windowSpec)
```

Bạn sẽ nhận thấy rằng điều này trả về một cột (hoặc biểu thức). Bây giờ chúng ta có thể sử dụng điều này trong câu lệnh chọn DataFrame. Tuy nhiên, trước khi làm như vậy, ta sẽ tạo thứ hạng số lượng mua. Để làm điều đó, ta sử dụng hàm `density_rank` để xác định ngày nào có số lượng mua tối đa cho mọi khách hàng. Có thể sử dụng `thick_rank` thay vì `rank` để tránh các khoảng trống trong chuỗi xếp hạng khi có các giá trị bị ràng buộc (hoặc trong trường hợp này là các hàng trùng lặp):

```
from pyspark.sql.functions import dense_rank, rank
purchaseDenseRank = dense_rank().over(windowSpec)
purchaseRank = rank().over(windowSpec)
```

Điều này cũng trả về một cột mà chúng ta có thể sử dụng trong các câu lệnh chọn. Bây giờ chúng ta có thể thực hiện chọn để xem các giá trị của window được tính toán:

```
dfWithDate.where("CustomerId IS NOT NULL").orderBy("CustomerId")\
  .select(
    col("CustomerId"),
    col("date"),
    col("Quantity"),
    purchaseRank.alias("quantityRank"),
    purchaseDenseRank.alias("quantityDenseRank"),
    maxPurchaseQuantity.alias("maxPurchaseQuantity")).show()
```

Out:

```
+----------+----------+--------+------------+----------------+--------------------+
|CustomerId|      date|Quantity|quantityRank|quantityDenseRank|maxPurchaseQuantity|
+----------+----------+--------+------------+----------------+--------------------+
|     12346|2011-01-18|   74215|           1|               1|               74215|
|     12346|2011-01-18|  -74215|           2|               2|               74215|
|     12347|2010-12-07|      36|           1|               1|                  36|
|     12347|2010-12-07|      30|           2|               2|                  36|
|     12347|2010-12-07|      24|           3|               3|                  36|
+----------+----------+--------+------------+----------------+--------------------+
only showing top 5 rows
```

V. BỘ NHÓM

Cho đến giờ, chúng ta đã thấy các biểu thức theo nhóm đơn giản mà chúng ta có thể sử dụng để tổng hợp trên một tập hợp các cột có giá trị trong các cột đó. Tuy nhiên, đôi khi chúng ta muốn một cái gì đó hoàn chỉnh hơn một chút - một tập hợp trên nhiều nhóm. Ta có thể làm được điều này bằng cách sử dụng *grouping sets*. Grouping sets là một công cụ cấp thấp để kết hợp các tập hợp lại với nhau. Chúng cung cấp cho bạn khả năng tạo tập hợp tùy ý trong các câu lệnh theo nhóm của chúng.

Để dễ hiểu hơn, hãy làm một ví dụ sau. Ta sẽ lấy ra tổng số lượng của tất cả các mã chứng khoán và mã khách hàng:

```
dfNoNull = dfWithDate.drop()
dfNoNull.createOrReplaceTempView("dfNoNull")

sqlDF_1 = spark.sql("SELECT CustomerId, stockCode, sum(Quantity) " +
                    "FROM dfNoNull " +
                    "GROUP BY customerId, stockCode " +
                    "GROUPING SETS((customerId, stockCode)) " +
                    "ORDER BY CustomerId DESC, stockCode DESC")
sqlDF_1.show(10)

+----------+---------+-------------+
|customerId|stockCode|sum(Quantity)|
+----------+---------+-------------+
|     18287|    85173|           48|
|     18287|   85040A|           48|
|     18287|   85039B|          120|
|     18287|   85039A|           96|
|     18287|    84920|            4|
|     18287|    84584|            6|
|     18287|   84507C|            6|
|     18287|   72351B|           24|
|     18287|   72351A|           24|
|     18287|   72349B|           60|
+----------+---------+-------------+
only showing top 10 rows
```

Nhờ việc sử dụng `GROUPING SETS` ta đã tạo được 1 tập hợp trên 2 nhóm `customerId` và `stockCode`

Toán tử `GROUPING SETS` chỉ có sẵn trong SQL. Để thực hiện điều tương tự trong DataFrames, ta sử dụng toán tử `rollup` và `cube`.

1. Rollups

Cho đến bây giờ, chúng ta đã tìm hiểu các nhóm rõ ràng. Khi ta đặt các khóa nhóm của nhiều cột, Spark sẽ xem xét các khóa đó cũng như các kết hợp thực tế hiển thị trong tập dữ liệu. Rollup là tập hợp đa chiều thực hiện nhiều tính toán theo nhóm.

Hãy tạo một rollup xem xét theo thời gian (với cột Date) và không gian (với cột Country) và tạo một DataFrame mới bao gồm tổng cộng của tất cả các ngày, tổng cộng cho mỗi ngày trong DataFrame và tổng phụ cho mỗi quốc gia vào mỗi ngày trong DataFrame:

```
rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))\
  .selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")\
  .orderBy("Date")
rolledUpDF.show()
```

```
Out:
+----------+--------------+--------------+
|      Date|       Country|total_quantity|
+----------+--------------+--------------+
|      null|          null|       5176450|
|2010-12-01|United Kingdom|         23949|
|2010-12-01|          null|         26814|
|2010-12-01|     Australia|           107|
|2010-12-01|        France|           449|
+----------+--------------+--------------+
only showing top 5 rows
```

Bây giờ nơi bạn nhìn thấy các giá trị `null` là nơi bạn sẽ tìm thấy tổng số lớn. Giá trị `null` trong cả hai cột rollup chỉ định tổng số trên cả hai cột đó:

```
rolledUpDF.where("Country IS NULL").show()
rolledUpDF.where("Date IS NULL").show()
```

```
Out:
+----------+-------+--------------+
|      Date|Country|total_quantity|
+----------+-------+--------------+
|      null|   null|       5176450|
|2010-12-01|   null|         26814|
|2010-12-02|   null|         21023|
|2010-12-03|   null|         14830|
|2010-12-05|   null|         16395|
+----------+-------+--------------+
only showing top 5 rows

+----+-------+--------------+
|Date|Country|total_quantity|
+----+-------+--------------+
|null|   null|       5176450|
+----+-------+--------------+
```

2. Cube

Cube đưa rollup lên một cấp độ sâu hơn. Thay vì xử lý các phần tử theo thứ bậc, cube thực hiện điều tương tự trên tất cả các chiều. Điều này có nghĩa là nó sẽ không chỉ chạy theo ngày trong toàn bộ khoảng thời gian mà còn cả quốc gia. Để đặt lại câu hỏi này, bạn có thể tạo một bảng bao gồm các nội dung sau không?

- Tổng số trên tất cả các ngày và quốc gia
- Tổng số cho mỗi ngày trên tất cả các quốc gia
- Tổng số cho mỗi quốc gia vào mỗi ngày
- Tổng số cho mỗi quốc gia trong tất cả các ngày

Cách gọi phương thức khá giống nhau, nhưng thay vì gọi `rollup`, chúng ta gọi `cube`:

```
dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))\
  .select("Date", "Country", "sum(Quantity)").orderBy("Date").show()
```

```
Out:
+----+--------------------+-------------+
|Date|             Country|sum(Quantity)|
+----+--------------------+-------------+
|null|United Arab Emirates|          982|
|null|               Italy|         7999|
|null|           Singapore|         5234|
|null|             Finland|        10666|
|null|              Greece|         1556|
+----+--------------------+-------------+
only showing top 5 rows
```

Đây là một bản tóm tắt nhanh chóng và dễ dàng truy cập gần như tất cả thông tin trong bảng và đó là một cách tuyệt vời để tạo một bảng tóm tắt nhanh mà những người khác có thể sử dụng sau này.

3. Grouping Metadata

Đôi khi, khi sử dụng các cube và rollup, bạn muốn có thể truy vấn các mức tổng hợp để bạn có thể dễ dàng lọc chúng một cách phù hợp. Ta có thể làm điều này bằng cách sử dụng `grouping_id`, nó cung cấp một cột chỉ định mức độ tổng hợp mà ta có trong bộ kết quả của mình. Truy vấn trong ví dụ sau trả về bốn ID nhóm riêng biệt:

Grouping ID	Mô tả
3	Đại diện cho mức tổng hợp cao nhất, cái mà cho chúng ta tổng số lượng bất kể `customId` và `stockCode`
2	Đại diện cho các tổng hợp các mã thị trường riêng lẻ. Nó sẽ cho ta tổng số lượng trên mỗi mã thị trường, bất kể khách hàng
1	Cho ta tổng số trên cơ sở từng khách hàng bất kể mặt hàng đã mua
0	Cho ta tổng số các tổ hợp của từng mã khách hàng và mã thị trường riêng lẻ

```
dfNoNull.cube("customerId", "stockCode").agg(grouping_id(),
sum("Quantity"))\
.orderBy(expr("grouping_id()").desc())\
.show()
```

Out:

```
+----------+---------+-------------+-------------+
|customerId|stockCode|grouping_id()|sum(Quantity)|
+----------+---------+-------------+-------------+
|      null|     null|            3|      5176450|
|      null|    22907|            2|         6535|
|      null|    85187|            2|          337|
|      null|    22832|            2|          155|
|      null|    22111|            2|         4537|
+----------+---------+-------------+-------------+
only showing top 5 rows
```

4. Pivot

Pivot giúp bạn có thể chuyển đổi một hàng thành một cột. Ví dụ: trong dữ liệu hiện tại có cột `Country`. Với một pivot, chúng tôi có thể tổng hợp theo một số chức năng cho từng quốc gia cụ thể đó và hiển thị chúng theo cách dễ truy vấn:

```
pivoted = dfWithDate.groupBy("date").pivot("Country").sum()
```

DataFrame này sẽ có một cột cho mọi tổ hợp của quốc gia, biến số và một cột chỉ định ngày. Ví dụ: với USA ta sẽ có các cột sau: `USA_sum(Quantity)`, `USA_sum(UnitPrice)`, `USA_sum(CustomerID`. Nó sẽ đại diện cho mỗi cột trong tập dữ liệu của ta.

Đây là một truy vấn mẫu:

```
pivoted.where("date > '2011-12-05'").select("date"
,"`USA_sum(Quantity)`").show()
```

```
Out:
+----------+------------------+
|      date|USA_sum(Quantity)|
+----------+------------------+
|2011-12-06|              null|
|2011-12-09|              null|
|2011-12-08|              -196|
|2011-12-07|              null|
+----------+------------------+
```

Giờ đây, tất cả các cột có thể được tính toán bằng các nhóm đơn lẻ, nhưng giá trị của trục phụ thuộc vào cách bạn muốn khám phá dữ liệu. Nó có thể hữu ích, nếu bạn có lực lượng đủ thấp trong một cột nhất định để chuyển đổi nó thành các cột để người dùng có thể xem lược đồ và biết ngay những gì cần truy vấn.

VI. HÀM TỔNG HỢP NGƯỜI DÙNG ĐỊNH NGHĨA

User-defined aggregation functions (UDAFs) là một cách để người dùng xác định các hàm tổng hợp của riêng họ dựa trên các công thức tùy chỉnh. Bạn có thể sử dụng UDAF để tính toán các phép tính tùy chỉnh trên các nhóm dữ liệu đầu vào (trái ngược với các hàng đơn lẻ). Spark duy trì một AggregationBuffer duy nhất để lưu trữ kết quả trung gian cho mọi nhóm dữ liệu đầu vào.

Để tạo UDAF, bạn phải kế thừa từ lớp cơ sở UserDefinedAggregateFunction và triển khai các phương thức sau:

inputSchema biểu thị các đối số đầu vào dưới dạng bộ đệm StructType

bufferSchema biểu thị các kết quả UDAF trung gian dưới dạng StructType

dataType biểu thị kết quả trả về DataType

deterministic là một giá trị Boolean chỉ định liệu UDAF này sẽ trả về kết quả tương tự cho input đã cho

initialize cho phép khởi tạo các giá trị của 1 aggregation buffer

update aggregation buffer mô tả cách bạn nên cập nhật aggregation buffer dựa trên việc hợp nhất hàng đã cho

merge mô tả cách hợp nhất 2 aggregation buffer

evaluate sẽ tạo ra kết quả cuối cùng của việc tổng hợp (tức aggregation buffer)

Ví dụ sau triển khai BoolAnd, sẽ cho chúng ta biết liệu tất cả các hàng (đối với một cột nhất định) là true hay false:

```
import org.apache.spark.sql.expressions.MutableAggregationBuffer
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
class BoolAnd extends UserDefinedAggregateFunction {
        def inputSchema: org.apache.spark.sql.types.StructType =
                StructType(StructField("value", BooleanType) :: Nil)
        def bufferSchema: StructType = StructType(
                StructField("result", BooleanType) :: Nil
                )
        def dataType: DataType = BooleanType
        def deterministic: Boolean = true
        def initialize(buffer: MutableAggregationBuffer): Unit = {
                buffer(0) = true
                }
        def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
                buffer(0) = buffer.getAs[Boolean](0) && input.getAs[Boolean](0)
                }
        def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
                buffer1(0) = buffer1.getAs[Boolean](0) && buffer2.getAs[Boolean](0)
                }
        def evaluate(buffer: Row): Any = {
                buffer(0)
                }
        }
```

Bây giờ, chúng ta chỉ cần khởi tạo đối tượng ba từ class BoolAnd vừa xây dựng và đăng ký BoolAnd như một hàm bằng cách sử dụng spark.udf.register("booland", ba):

```
val ba = new BoolAnd
spark.udf.register("booland", ba)
import org.apache.spark.sql.functions._
spark.range(1)
        .selectExpr("explode(array(TRUE, TRUE, TRUE)) as t")
        .selectExpr("explode(array(TRUE, FALSE, TRUE)) as f", "t")
        .select(ba(col("t")), expr("booland(f)"))
        .show()
```

Kết quả thu được như sau:

```
+----------------+----------------+
|     booland(t)|     booland(f)|
+----------------+----------------+
|            true|           false|
+----------------+----------------+
```

UDAF hiện tại chỉ xuất hiện trong Scala hoặc Java. Muốn xây dựng những hàm tương tự như vậy trong ngôn ngữ khác như Python thì không thể implement 1 User Defined Aggregate Function (UDAF). Thay vào đó có nhiều giải pháp thay thế khác.

- Với những phiên bản trước Spark 2.4.0: UDAF là các chức năng hoạt động trên dữ liệu được nhóm theo một khóa. Cụ thể, chúng cần xác định cách hợp nhất nhiều giá trị trong nhóm vào một phân vùng duy nhất và sau đó là cách hợp nhất các kết quả trên các phân vùng cho khóa. Hiện tại không có cách nào trong python để triển khai UDAF, chúng chỉ có thể được triển khai trong Scala. Tuy nhiên, bạn có thể giải quyết nó bằng Python. Bạn có thể sử dụng collect_set để thu thập các giá trị được nhóm của mình và sau đó sử dụng UDF thông thường để thực hiện những gì bạn muốn với chúng. Thông báo trước duy nhất là collect_set chỉ hoạt động trên các giá trị nguyên thủy.
- Với những phiên bản sau Spark 2.4.0: bạn có thể sử dụng PANDAS được xây dựng bổ sung trong thư viện. Muốn hiểu rõ hơn về PANDAS bạn có thể đọc documentation sau đã viết rất cụ thể: Pandas API on Spark — PySpark 3.2.0 documentation