**Spark properties**

Thuộc tính Spark kiểm soát hầu hết các cài đặt ứng dụng và được cấu hình riêng cho từng ứng dụng. Các thuộc tính này có thể được đặt trực tiếp trên SparkConf được chuyển tới SparkContext. SparkConf cho phép định cấu hình một số thuộc tính phổ biến (ví dụ: URL chính và tên ứng dụng), cũng như các cặp khóa-giá trị tùy ý thông qua phương thức set(). Ví dụ: chúng ta có thể khởi tạo một ứng dụng với hai luồng như sau:

**val** conf **=** **new** **SparkConf**()

.setMaster("local[2]")

.setAppName("CountingSheep")

**val** sc **=** **new** **SparkContext**(conf)

Lưu ý rằng có thể có nhiều hơn 1 luồng ở chế độ cục bộ và trong các trường hợp như Spark Streaming, có thể yêu cầu nhiều hơn 1 luồng để ngăn chặn bất kỳ loại vấn đề starvation.

Các thuộc tính chỉ định một số khoảng thời gian nên được cấu hình với một đơn vị thời gian. Định dạng sau được chấp nhận:

25ms (milliseconds)

5s (seconds)

10m or 10min (minutes)

3h (hours)

5d (days)

1y (years)

Các thuộc tính chỉ định kích thước byte nên được cấu hình với đơn vị kích thước. Định dạng sau được chấp nhận:

1b (bytes)

1k or 1kb (kibibytes = 1024 bytes)

1m or 1mb (mebibytes = 1024 kibibytes)

1g or 1gb (gibibytes = 1024 mebibytes)

1t or 1tb (tebibytes = 1024 gibibytes)

1p or 1pb (pebibytes = 1024 tebibytes)

Trong khi các số không có đơn vị thường được hiểu là byte, một số ít được hiểu là KiB hoặc MiB. Xem tài liệu về các thuộc tính cấu hình riêng lẻ. Việc chỉ định đơn vị là mong muốn nếu có thể.

## Dynamically Loading Spark Properties

Trong một số trường hợp, bạn có thể muốn tránh mã hóa cứng các cấu hình nhất định trong SparkConf. Ví dụ: nếu bạn muốn chạy cùng một ứng dụng với các bản chính khác nhau hoặc số lượng bộ nhớ khác nhau. Spark cho phép bạn chỉ cần tạo một conf trống:

**val** sc **=** **new** **SparkContext**(**new** **SparkConf**())

Sau đó, bạn có thể cung cấp các giá trị cấu hình trong thời gian chạy:

./bin/spark-submit **--name** "My app" **--master** local[4] **--conf** spark.eventLog.enabled=false

**--conf** "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myApp.jar

Spark shell và công cụ spark-submit hỗ trợ hai cách để tải cấu hình động. Đầu tiên là các tùy chọn dòng lệnh, chẳng hạn như --master, như hình trên. spark-submit có thể chấp nhận bất kỳ thuộc tính Spark nào sử dụng --conf / -c flag , nhưng sử dụng flag đặc biệt cho các thuộc tính đóng một vai trò trong việc khởi chạy ứng dụng Spark. Chạy ./bin/spark-submit --help sẽ hiển thị toàn bộ danh sách các tùy chọn này.

bin / spark-submit cũng sẽ đọc các tùy chọn cấu hình từ conf / spark-defaults.conf, trong đó mỗi dòng bao gồm một khóa và một giá trị được phân tách bằng khoảng trắng. Ví dụ:

spark.master spark://5.6.7.8:7077

spark.executor.memory 4g

spark.eventLog.enabled true

spark.serializer org.apache.spark.serializer.KryoSerializer

Mọi giá trị được chỉ định dưới dạng flag hoặc trong tệp thuộc tính sẽ được chuyển đến ứng dụng và được hợp nhất với những giá trị được chỉ định thông qua SparkConf. Các thuộc tính được đặt trực tiếp trên SparkConf được ưu tiên cao nhất, sau đó các flag được chuyển đến spark-submit hoặc spark-shell, sau đó là các tùy chọn trong tệp spark-defaults.conf. Một vài khóa cấu hình đã được đổi tên kể từ các phiên bản Spark trước đó; trong những trường hợp như vậy, các tên khóa cũ hơn vẫn được chấp nhận, nhưng được ưu tiên thấp hơn bất kỳ trường hợp nào của khóa mới hơn.

Các thuộc tính của Spark chủ yếu có thể được chia thành hai loại: một là liên quan đến triển khai, như “spark.driver.memory”, “spark.executor.instances”, loại thuộc tính này có thể không bị ảnh hưởng khi thiết lập lập trình thông qua SparkConf trong thời gian chạy, hoặc hành vi tùy thuộc vào trình quản lý cụm và chế độ triển khai bạn chọn, vì vậy bạn nên đặt thông qua tệp cấu hình hoặc tùy chọn dòng lệnh spark-submit; một loại khác chủ yếu liên quan đến kiểm soát thời gian chạy Spark, như “spark.task.maxFailures”, loại thuộc tính này có thể được đặt theo một trong hai cách.

## Viewing Spark Properties

Giao diện người dùng web ứng dụng tại http: // <driver>: 4040 liệt kê các thuộc tính Spark trong tab "Môi trường". Đây là một nơi hữu ích để kiểm tra để đảm bảo rằng các thuộc tính của bạn đã được đặt chính xác. Lưu ý rằng chỉ các giá trị được chỉ định rõ ràng thông qua spark-defaults.conf, SparkConf hoặc dòng lệnh mới xuất hiện. Đối với tất cả các thuộc tính cấu hình khác, bạn có thể giả sử giá trị mặc định được sử dụng.

SPARK RDD (Resilient Distributed Datasets)

Resilient Distributed Datasets (RDD) là một cấu trúc dữ liệu cơ bản của Spark. Nó là một tập hợp bất biến phân tán của một đối tượng. Mỗi dataset trong RDD được chia ra thành nhiều phần vùng logical. Có thể được tính toán trên các node khác nhau của một cụm máy chủ (cluster).

RDDs có thể chứa bất kỳ kiểu dữ liệu nào của Python, Java, hoặc đối tượng Scala, bao gồm các kiểu dữ liệu do người dùng định nghĩa. Thông thường, RDD chỉ cho phép đọc, phân mục tập hợp của các bản ghi. RDDs có thể được tạo ra qua điều khiển xác định trên dữ liệu trong bộ nhớ hoặc RDDs, RDD là một tập hợp có khả năng chịu lỗi mỗi thành phần có thể được tính toán song song.

Có hai cách để tạo RDDs:

* Tạo từ một tập hợp dữ liệu có sẵn trong ngôn ngữ sử dụng như Java, Python, Scala.
* Lấy từ dataset hệ thống lưu trữ bên ngoài như HDFS, Hbase hoặc các cơ sở dữ liệu quan hệ.

##### Thực thi trên Spark RDD

**- Iterative Operation trên Spark RDD:**



**- Interactive Operations trên Spark RDD:**



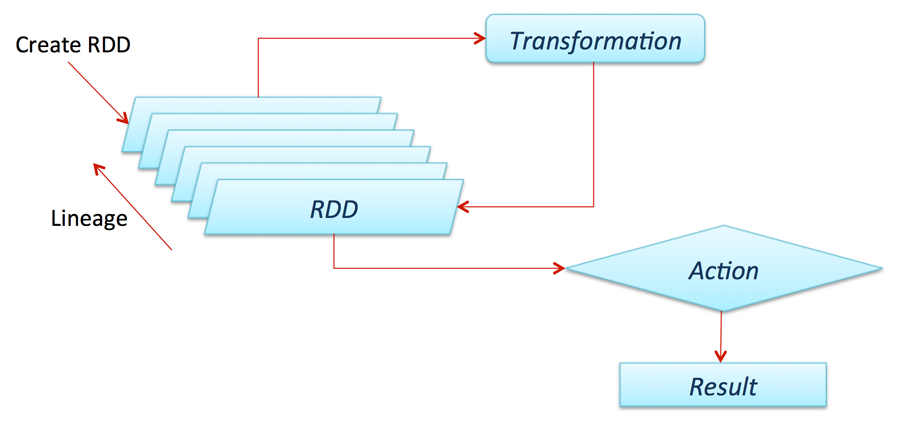
##### Các loại RDD



* Các RDD biểu diễn một tập hợp cố định, đã được phân vùng các record để có thể xử lý song song.
* Các record trong RDD có thể là đối tượng Java, Scale hay Python tùy lập trình viên chọn. Không giống như DataFrame, mỗi record của DataFrame phải là một dòng có cấu trúc chứa các field đã được định nghĩa sẵn.
* RDD đã từng là API chính được sử dụng trong series Spark 1.x và vẫn có thể sử dụng trong version 2.X nhưng không còn được dùng thường xuyên nữa.
* RDD API có thể được sử dụng trong Python, Scala hay Java:
  + Scala và Java: Perfomance tương đương trên hầu hết mọi phần. (Chi phí lớn nhất là khi xử lý các raw object)
  + Python: Mất một lượng performance, chủ yếu là cho việc serialization giữa tiến trình Python và JVM

##### Các transformation và action với RDD

RDD cung cấp các transformation và action hoạt động giống như DataFrame lẫn DataSets. Transformation xử lý các thao tác lazily và Action xử lý thao tác cần xử lý tức thời.



**Một số transformation:**

Nhiều phiên bản transformation của RDD có thể hoạt động trên các Structured API, transformation xử lý lazily, tức là chỉ giúp dựng execution plans, dữ liệu chỉ được truy xuất thực sự khi thực hiện action

* **distinct**: loại bỏ trùng lắp trong RDD
* **filter**: tương đương với việc sử dụng where trong SQL – tìm các record trong RDD xem những phần tử nào thỏa điều kiện. Có thể cung cấp một hàm phức tạp sử dụng để filter các record cần thiết – Như trong Python, ta có thể sử dụng hàm lambda để truyền vào filter
* **map**: thực hiện một công việc nào đó trên toàn bộ RDD. Trong Python sử dụng lambda với từng phần tử để truyền vào map
* **flatMap**: cung cấp một hàm đơn giản hơn hàm map. Yêu cầu output của map phải là một structure có thể lặp và mở rộng được.
* **sortBy**: mô tả một hàm để trích xuất dữ liệu từ các object của RDD và thực hiện sort được từ đó.
* **randomSplit**: nhận một mảng trọng số và tạo một random seed, tách các RDD thành một mảng các RDD có số lượng chia theo trọng số.

**- Một số action:**

Action thực thi ngay các transformation đã được thiết lập để thu thập dữ liệu về driver để xử lý hoặc ghi dữ liệu xuống các công cụ lưu trữ.

* **reduce**: thực hiện hàm reduce trên RDD để thu về 1 giá trị duy nhất
* **count**: đếm số dòng trong RDD
* **countApprox:** phiên bản đếm xấp xỉ của count, nhưng phải cung cấp timeout vì có thể không nhận được kết quả.
* **countByValue**: đếm số giá trị của RDD  
  chỉ sử dụng nếu map kết quả nhỏ vì tất cả dữ liệu sẽ được load lên memory của driver để tính toán  
  chỉ nên sử dụng trong tình huống số dòng nhỏ và số lượng item khác nhau cũng nhỏ.
* **countApproxDistinct**: đếm xấp xỉ các giá trị khác nhau
* **countByValueApprox**: đếm xấp xỉ các giá trị
* **first**: lấy giá trị đầu tiên của dataset
* **max và min:** lần lượt lấy giá trị lớn nhất và nhỏ nhất của dataset
* **take và các method tương tự**: lấy một lượng giá trị từ trong RDD. take sẽ trước hết scan qua một partition và sử dụng kết quả để dự đoán số lượng partition cần phải lấy thêm để thỏa mãn số lượng lấy.
* **top và takeOrdered**: top sẽ hiệu quả hơn takeOrdered vì top lấy các giá trị đầu tiên được sắp xếp ngầm trong RDD.
* **takeSamples**: lấy một lượng giá trị ngẫu nhiên trong RDD

Code minh họa

**Lấy các tệp dữ liệu:**

import urllib

f = urllib.urlretrieve ("http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data\_10\_percent.gz", "kddcup.data\_10\_percent.gz")

**Tạo RDD từ một tệp hoặc sử dụng song song:**

#Creating a RDD from a file

data\_file = "./kddcup.data\_10\_percent.gz"

raw\_data = sc.textFile(data\_file)

#Creating RDD using parallelize from an already existing list.

a = range(100)

data = sc.parallelize(a)

Thao tác cơ bản trong RRD : Map , Filter và Collect

|  |
| --- |
| # The filter transformation |
|  | normal\_raw\_data = raw\_data.filter(lambda x: 'normal.' in x) |
|  | normal\_count = normal\_raw\_data.count() |
|  |  |
|  | # The map transformation |
|  | csv\_data = raw\_data.map(lambda x: x.split(",")) |
|  | head\_rows = csv\_data.take(5) |
|  |  |
|  | # Using map and predefined functions |
|  | def parse\_interaction(line): |
|  | elems = line.split(",") |
|  | tag = elems[41] |
|  | return (tag, elems) |
|  |  |
|  | key\_csv\_data = raw\_data.map(parse\_interaction) |
|  | head\_rows = key\_csv\_data.take(5) |
|  |  |
|  | # The collect action |
|  | all\_raw\_data = raw\_data.collect() |

**Spark DataFrame**

là phiên bản Spark 1.3. Nó là một tập hợp phân phối dữ liệu được sắp xếp vào các cột được đặt tên. Khái niệm khôn ngoan, nó bằng với bảng trong cơ sở dữ liệu quan hệ hoặc khung dữ liệu trong [R](http://data-flair.training/blogs/r-programming-tutorial/) / Python. Chúng tôi có thể tạo DataFrame bằng cách sử dụng:

* Tệp dữ liệu có cấu trúc
* Bàn trong tổ ong
* Cơ sở dữ liệu bên ngoài
* Sử dụng RDD hiện có
* Giống như viết SQL, đầy đủ chức năng như select, where ... đặc biệt là join với các DataFrame khác.
* Sử dụng các method như filter, select để trích xuất dữ liệu theo cột, hàng.
* Xử gọn các loại data như Log ... với groupBy → agg
* Thêm 1 cột dễ dàng với UDF(User Defined Function)
* Giống như SQL, Spark DataFrame đã hỗ trợ [Pivot](http://oracletechtalk.blogspot.jp/2015/09/pivot-va-unpivot-phan-1.html) (Spark 1.6 trở lên) rất hữu ích cho việc lập bảng biểu, báo cáo.

Cách tạo DataFrame :

**Cách 1: Tạo từ RDD**

Nếu bạn đã có RDD với tên column và type tương ứng (TimestampType, IntegerType, StringType)thì bạn có thể dễ dàng tạo DataFrame bằng

sqlContext.createDataFrame(my\_rdd, my\_schema)

Với printSchema(), dtypes sẽ in thông tin của schema  
và count() trả về số record

Và nếu chỉ muốn in n record đầu tiên thì sử dụng show(n)

fields = [StructField("access\_time", TimestampType(), True), StructField("userID", IntegerType(), True), StructField("campaignID", StringType(), True)]

schema = StructType(fields)

whole\_log\_df = sqlContext.createDataFrame(whole\_log, schema)

print whole\_log\_df.count()

print whole\_log\_df.printSchema()

print whole\_log\_df.dtypes

print whole\_log\_df.show(5)

#327430

#root

# |-- access\_time: timestamp (nullable = true)

# |-- userID: integer (nullable = true)

# |-- campaignID: string (nullable = true)#

#[('access\_time', 'timestamp'), ('userID', 'int'), ('campaignID', 'string')]#

#+--------------------+------+-----------+

#| access\_time|userID| campaignID|

#+--------------------+------+-----------+

#|2015-04-27 20:40:...|144012|Campaign077|

#|2015-04-27 00:27:...| 24485|Campaign063|

#|2015-04-27 00:28:...| 24485|Campaign063|

#|2015-04-27 00:33:...| 24485|Campaign038|

#|2015-04-27 01:00:...| 24485|Campaign063|

**Cách 2: Tạo trực tiếp từ file CSV**

from pyspark.shell import spark

from pyspark.sql.types import \*

# Định nghĩa Schema

struct = StructType([

StructField('a', StringType(), False),

StructField('b', StringType(), False),

StructField('c', StringType(), False)])

# Tạo DataFrame từ file CSV

df\_data = spark.read.csv('click\_data\_sample', struct)

Ngoài cách trên còn rất nhiều cách khác, có thể kể đến là gọi cổ 1 thằng thuộc họ nhà [Spark Package](https://spark-packages.org/) tên là [spark-csv](https://github.com/databricks/spark-csv) ra xài, hỗ trợ nhiều method hữu ích, dễ chơi, dễ trúng thưởng hơn.

Lưu ý là nếu không định nghĩa schema thì tất cả các column sẽ có kiểu string. Nhưng nếu dùng thêm thằng em inferSchema thì mọi chuyện sẽ êm xuôi. ^^ Không tin đơn giản như vậy ư, bạn thử như code dưới xem ^^

whole\_log\_df\_2 = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").load("/user/hadoop/click\_data\_sample.csv")

print whole\_log\_df\_2.printSchema()

print whole\_log\_df\_2.show(5)

#root

# |-- click.at: string (nullable = true)

# |-- user.id: string (nullable = true)

# |-- campaign.id: string (nullable = true)#

#+-------------------+-------+-----------+

#| click.at|user.id|campaign.id|

#+-------------------+-------+-----------+

#|2015-04-27 20:40:40| 144012|Campaign077|

#|2015-04-27 00:27:55| 24485|Campaign063|

#|2015-04-27 00:28:13| 24485|Campaign063|

#|2015-04-27 00:33:42| 24485|Campaign038|

#|2015-04-27 01:00:04| 24485|Campaign063|

whole\_log\_df\_3 = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema", "true").load("click\_data\_sample.csv")

print whole\_log\_df\_3.printSchema()

#root

# |-- click.at: timestamp (nullable = true)

# |-- user.id: integer (nullable = true)

# |-- campaign.id: string (nullable = true)

**Cách 3: Giao lưu trực tiếp từ file json**

Bằng cách sử dụng sqlContext.read.json. Mỗi dòng của file json sẽ được coi là 1 object. Trong trường hợp object thiếu data thì sẽ null tại đó.

# test\_json.json gồm 3 dòng như dưới, dòng cuối không có "campaignID"#

#{"access\_time": "2015-04-27 20:40:40", "userID": "24485", "campaignID": "Campaign063"}

#{"access\_time": "2015-04-27 00:27:55", "userID": "24485", "campaignID": "Campaign038"}

#{"access\_time": "2015-04-27 00:27:55", "userID": "24485"}

df\_json = sqlContext.read.json("test\_json.json")

df\_json.printSchema()

df\_json.show(5)

#root

# |-- access\_time: string (nullable = true)

# |-- campaignID: string (nullable = true)

# |-- userID: string (nullable = true)

#

#+-------------------+-----------+------+

#| access\_time| campaignID|userID|

#+-------------------+-----------+------+

#|2015-04-27 20:40:40|Campaign063| 24485|

#|2015-04-27 00:27:55|Campaign038| 24485|

#|2015-04-27 00:27:55| null| 24485|

#+-------------------+-----------+------+

**Cách 4: Giao thông trực tiếp từ parquet**

Nếu bạn chưa biết parquet là gì thì tham khảo tại [đây](https://parquet.apache.org/) nhé. Với rất nhiều ưu điểm, mình sẽ giới thiệu vào 1 bài nào đó ở lúc nào đó nhé. ^^  
Và cách đọc, đơn giản như đan rổ là sử dụng sqlContext.read.parquet. Như đoạn code bên dưới, cả folder chứa file parquet sẽ được đọc. Quá nhanh gọn lẹ phải không.

sqlContext.read.parquet("/user/hadoop/parquet\_folder/")

### **Thao tác với DataFrame**

#### **1 Thử query bằng SQL**

Bằng cách sử dụng registerTempTable, bạn sẽ có một table được tham chiếu đến Dataframe đó, bạn có thể sử dụng tên table này để viết query SQL. Nếu bạn sử dụng sqlContext.sql('query SQL') thì giá trị trả về cũng là Dataframe.

Có 1 lưu ý là: Bạn cũng có thể viết subquery nhưng subquery cần được gán Alias, nếu không toạch lô (Syntax error) đấy.

#SQL query

whole\_log\_df.registerTempTable("whole\_log\_table")

print sqlContext.sql(" SELECT \* FROM whole\_log\_table where campaignID == 'Campaign047' ").count()

#18081

print sqlContext.sql(" SELECT \* FROM whole\_log\_table where campaignID == 'Campaign047' ").show(5)

#+--------------------+------+-----------+

#| access\_time|userID| campaignID|

#+--------------------+------+-----------+

#|2015-04-27 05:26:...| 14151|Campaign047|

#|2015-04-27 05:26:...| 14151|Campaign047|

#|2015-04-27 05:26:...| 14151|Campaign047|

#|2015-04-27 05:27:...| 14151|Campaign047|

#|2015-04-27 05:28:...| 14151|Campaign047|

#+--------------------+------+-----------+

#Trường hợp thêm biến số vào trong câu SQL

for count in range(1, 3):

print "Campaign00" + str(count)

print sqlContext.sql("SELECT count(\*) as access\_num FROM whole\_log\_table where campaignID == 'Campaign00" + str(count) + "'").show()

#Campaign001

#+----------+

#|access\_num|

#+----------+

#| 2407|

#+----------+#

#Campaign002

#+----------+

#|access\_num|

#+----------+

#| 1674|

#+----------+

#Trường hợp Sub Query：

print sqlContext.sql("SELECT count(\*) as first\_count FROM (SELECT userID, min(access\_time) as first\_access\_date FROM whole\_log\_table GROUP BY userID) subquery\_alias WHERE first\_access\_date < '2015-04-28'").show(5)

#+------------+

#|first\_count |

#+------------+

#| 20480|

#+------------+

#### **Tìm kiếm sử dụng filter, select**

Đối với DataFrame , tìm kiếm kèm điều kiện rất đơn giản. Giống với câu query ở trên nhưng filter, select dễ dàng hơn rất nhiều. Vậy filter và select khác nhau thế nào ?

Cùng là để tìm kiếm nhưng filter trả về những row thoả mãn điều kiện, trong đó select lấy dữ liệu theo column.

#Ví dụ filter

print whole\_log\_df.filter(whole\_log\_df["access\_time"] < "2015-04-28").count()

#41434

print whole\_log\_df.filter(whole\_log\_df["access\_time"] > "2015-05-01").show(3)

#+--------------------+------+-----------+

#| access\_time|userID| campaignID|

#+--------------------+------+-----------+

#|2015-05-01 22:11:...|114157|Campaign002|

#|2015-05-01 23:36:...| 93708|Campaign055|

#|2015-05-01 22:51:...| 57798|Campaign046|

#+--------------------+------+-----------+

#Ví dụ select

print whole\_log\_df.select("access\_time", "userID").show(3)

#+--------------------+------+

#| access\_time|userID|

#+--------------------+------+

#|2015-04-27 20:40:...|144012|

#|2015-04-27 00:27:...| 24485|

#|2015-04-27 00:28:...| 24485|

#+--------------------+------+

#### **3. Sử dụng groupBy**

groupBy có chức năng giống với reduceByKey của RDD, nhưng nó còn cung cấp [1 rổ method](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.GroupedData). Ở đây mình sẽ run thử code count, agg.

**groupBy→count**

Ví dụ sau sẽ lấy key là campaignID và tiến hành groupBy. Sau đó dùng count() để lấy số record ứng với mỗi key.

print whole\_log\_df.groupBy("campaignID").count().sort("count", ascending=False).show(5)

#+-----------+-----+

#| campaignID|count|

#+-----------+-----+

#|Campaign116|22193|

#|Campaign027|19206|

#|Campaign047|18081|

#|Campaign107|13295|

#|Campaign131| 9068|

#+-----------+-----+

print whole\_log\_df.groupBy("campaignID", "userID").count().sort("count", ascending=False).show(5)

#+-----------+------+-----+

#| campaignID|userID|count|

#+-----------+------+-----+

#|Campaign047| 30292| 633|

#|Campaign086|107624| 623|

#|Campaign047|121150| 517|

#|Campaign086| 22975| 491|

#|Campaign122| 90714| 431|

#+-----------+------+-----+

**groupBy→pivot**

Pivot thì từ Spark v1.6 trở lên được đưa vào, có chức năng giống với pivot trong SQL. Thử áp dụng xem sao nhỉ :

Trước khi pivot (agged\_df)

* Số hàng = số UserID(=75,545) \* campainID(=133)
* Số cột = 3

Sau khi pivot(pivot\_df)

* Số hàng = số UserID(=75,545)
* Số cột = UserID + số CampainID = 1 + 133 = 134

Tất nhiên là bạn phải groupBy(cột giữ nguyên).pivot(cột muốn chuyển sang ngang).sum()

agged\_df = whole\_log\_df.groupBy("userID", "campaignID").count()

print agged\_df.show(3)

#+------+-----------+-----+

#|userID| campaignID|count|

#+------+-----------+-----+

#|155812|Campaign107| 4|

#|103339|Campaign027| 1|

#|169114|Campaign112| 1|

#+------+-----------+-----+

#Những cell không có giá trị -> null

pivot\_df = agged\_df.groupBy("userID").pivot("campaignID").sum("count")

print pivot\_df.printSchema()

#root

# |-- userID: integer (nullable = true)

# |-- Campaign001: long (nullable = true)

# |-- Campaign002: long (nullable = true)# ..

# |-- Campaign133: long (nullable = true)

#TH muốn add 0 vào cell NULL

pivot\_df2 = agged\_df.groupBy("userID").pivot("campaignID").sum("count").fillna(0)

#### **4. Thêm cột sử dụng UDF**

Trong Spark DataFrame có thể sử dụng UDF, với ứng dụng chính là thêm cột. Như đã nói ở trên, bản chất DataFrame là immutable nên khi thêm cột thì 1 DataFrame mới sẽ được tạo.

from pyspark.sql.functions import UserDefinedFunction

from pyspark.sql.types import DoubleType

def add\_day\_column(access\_time):

return int(access\_time.strftime("%Y%m%d"))

my\_udf = UserDefinedFunction(add\_day\_column, IntegerType())

print whole\_log\_df.withColumn("access\_day", my\_udf("access\_time")).show(5)

#+--------------------+------+-----------+----------+

#| access\_time|userID| campaignID|access\_day|

#+--------------------+------+-----------+----------+

#|2015-04-27 20:40:...|144012|Campaign077| 20150427|

#|2015-04-27 00:27:...| 24485|Campaign063| 20150427|

#|2015-04-27 00:28:...| 24485|Campaign063| 20150427|

#|2015-04-27 00:33:...| 24485|Campaign038| 20150427|

#|2015-04-27 01:00:...| 24485|Campaign063| 20150427|

#+--------------------+------+-----------+----------+

Cũng có thể sử dụng lambda

my\_udf2 = UserDefinedFunction(lambda x: x + 5, IntegerType())

print whole\_log\_df.withColumn("userID\_2", my\_udf2("userID")).show(5)

#+--------------------+------+-----------+--------+

#| access\_time|userID| campaignID|userID\_2|

#+--------------------+------+-----------+--------+

#|2015-04-27 20:40:...|144012|Campaign077| 144017|

#|2015-04-27 00:27:...| 24485|Campaign063| 24490|

#|2015-04-27 00:28:...| 24485|Campaign063| 24490|

#|2015-04-27 00:33:...| 24485|Campaign038| 24490|

#|2015-04-27 01:00:...| 24485|Campaign063| 24490|

#+--------------------+------+-----------+--------+

Ngược lại, muốn xóa cột thì sử dụng df.drop()

print whole\_log\_df.drop("userID").show(3)

#+--------------------+-----------+

#| access\_time| campaignID|

#+--------------------+-----------+

#|2015-04-27 20:40:...|Campaign077|

#|2015-04-27 00:27:...|Campaign063|

#|2015-04-27 00:28:...|Campaign063|

#+--------------------+-----------+

#### **5. Join 2 DataFrame**

Tính năng rất quan trọng khi xử lý dữ liệu chính là join.  
Mình sẽ sử dụng data đầu vào ban đầu để tạo ra 1 DataFrame mới chứa những userID có count >100

heavy\_user\_df1 = whole\_log\_df.groupBy("userID").count()

heavy\_user\_df2 = heavy\_user\_df1.filter(heavy\_user\_df1 ["count"] >= 100)

print heavy\_user\_df2 .printSchema()

print heavy\_user\_df2 .show(3)

print heavy\_user\_df2 .count()

#root

# |-- userID: integer (nullable = true)

# |-- count: long (nullable = false)#

#+------+-----+

#|userID|count|

#+------+-----+

#| 84231| 134|

#| 13431| 128|

#|144432| 113|

#+------+-----+#

#177

Được heavy\_user\_df2 rồi tiến hành join (mặc định là inner join).

Các kiểu join bao gồm : inner, outer, left\_outer, rignt\_outer

joinded\_df = whole\_log\_df.join(heavy\_user\_df2, whole\_log\_df["userID"] == heavy\_user\_df2["userID"], "inner").drop(heavy\_user\_df2["userID"]).drop("count")

print joinded\_df.printSchema()

print joinded\_df.show(3)

print joinded\_df.count()

#root

# |-- access\_time: timestamp (nullable = true)

# |-- campaignID: string (nullable = true)

# |-- userID: integer (nullable = true)

#None

#+--------------------+-----------+------+

#| access\_time| campaignID|userID|

#+--------------------+-----------+------+

#|2015-04-27 02:07:...|Campaign086| 13431|

#|2015-04-28 00:07:...|Campaign086| 13431|

#|2015-04-29 06:01:...|Campaign047| 13431|

#+--------------------+-----------+------+#

#38729

#### **6. Lấy data theo cột trong DataFrame**

* Lấy lable của cột (df.columns) -> Trả về list tên cột (not DataFrame )
* Lấy riêng 1 cột (df.select("userID").map(lambda x: x[0]).collect()) -> Trả về list userID (not RDD/Dataframe)
* Lấy cột distinct, chỉ cần thêm .distinct()

print whole\_log\_df.columns

#['access\_time', 'userID', 'campaignID']

print whole\_log\_df.select("userID").map(lambda x: x[0]).collect()[:5]

#[144012, 24485, 24485, 24485, 24485]

print whole\_log\_df.select("userID").distinct().map(lambda x:x[0]).collect()[:5]

#[4831, 48631, 143031, 39631, 80831]

#### **7. Từ DataFrame , tạo RDD**

Có 2 cách chính là:

* Sử dụng map() : mỗi hàng của DataFrame được chuyển sang RDD theo dạng list
* Sử dụng .rdd: mỗi hàng của DataFrame được chuyển sang RDD [Row Object](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Row) (Tức là mỗi hàng sẽ là 1 Object) Tiếp theo sử dụng .asDict() với Row Object để chuyển về RDD Key-Value.

#convert to rdd by ".map"

print whole\_log\_df.groupBy("campaignID").count().map(lambda x: [x[0], x[1]]).take(5)

#[[u'Campaign033', 786], [u'Campaign034', 3867], [u'Campaign035', 963], [u'Campaign036', 1267], [u'Campaign037', 1010]]

# rdd -> normal list can be done with "collect".

print whole\_log\_df.groupBy("campaignID").count().map(lambda x: [x[0], x[1]]).collect()[:5]

#[[u'Campaign033', 786], [u'Campaign034', 3867], [u'Campaign035', 963], [u'Campaign036', 1267], [u'Campaign037', 1010]]

#convert to rdd by ".rdd" will return "Row" object

print whole\_log\_df.groupBy("campaignID").rdd.take(3)

#[Row(campaignID=u'Campaign033', count=786), Row(campaignID=u'Campaign034', count=3867), Row(campaignID=u'Campaign035', count=963)]

#`.asDict()` will convert to Key-Value RDD from Row object

print whole\_log\_df.groupBy("campaignID").rdd.map(lambda x:x.asDict()).take(3)

#[{'count': 786, 'campaignID': u'Campaign033'}, {'count': 3867, 'campaignID': u'Campaign034'}, {'count': 963, 'campaignID': u'Campaign035'}]