Spark Properties

Spark Properties

Spark properties 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 và truyền tới SparkContext. SparkConf cho phép cấu hình một số thuộc tính phổ biến (ví dụ: URL 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 có hai luồng như sau:

```
val conf = new SparkConf()
             .setMaster("local[2]")
             .setAppName("Spark Practice")
val sc = new SparkContext(conf)
```

Apache Spark RDD

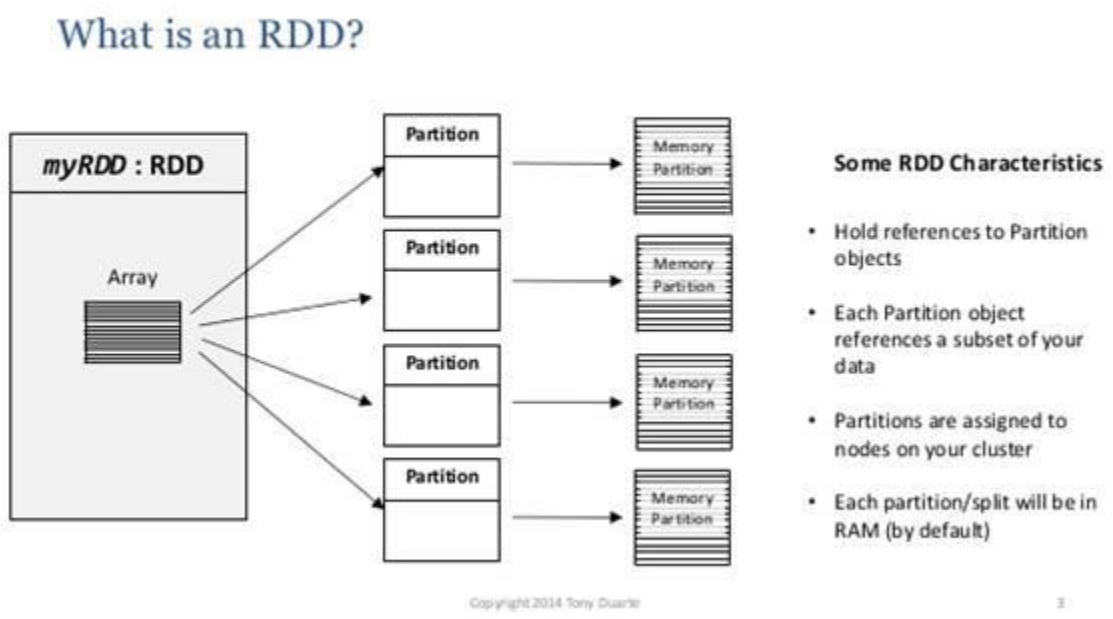
1. RDD là gì ?

RDD: *Resilient Distributed Dataset* là một cấu trúc cơ bản trong Apache Spark. RDD là đại diện cho tập dữ liệu phân tán, tức là RDD tập hợp các dữ liệu được phân tán đã xử lý qua các node. Ta hãy phân tích RDD:

- **Resilient:** là khả năng chịu lỗi bằng sự hỗ trợ từ các RDD lineage graph và có thể tính toán lại các RDD partitions khi node chứa RDD partitions đó bị lỗi.

- **Distributed:** Thể hiện việc các dữ liệu được phân tán trên các node.
- **Dataset:** Là tập dữ liệu được sử dụng.

Do đó, mỗi và mọi tập dữ liệu trong RDD được phân vùng một cách hợp lý trên nhiều máy chủ để chúng có thể được tính toán trên các node khác nhau của cụm. RDD có khả năng chịu lỗi, tức là nó có khả năng tự phục hồi trong trường hợp bị lỗi.



Có ba cách để tạo RDD trong Spark, chẳng hạn như - Dữ liệu bộ lưu trữ ổn định (*Data in stable storage)*, các RDD khác và song song hóa collection đã tồn tại trong chương trình trình điều khiển. Người ta cũng có thể vận hành Spark RDD song song với một API cấp thấp cung cấp các chuyển đổi (transformings) và hành động (actions)

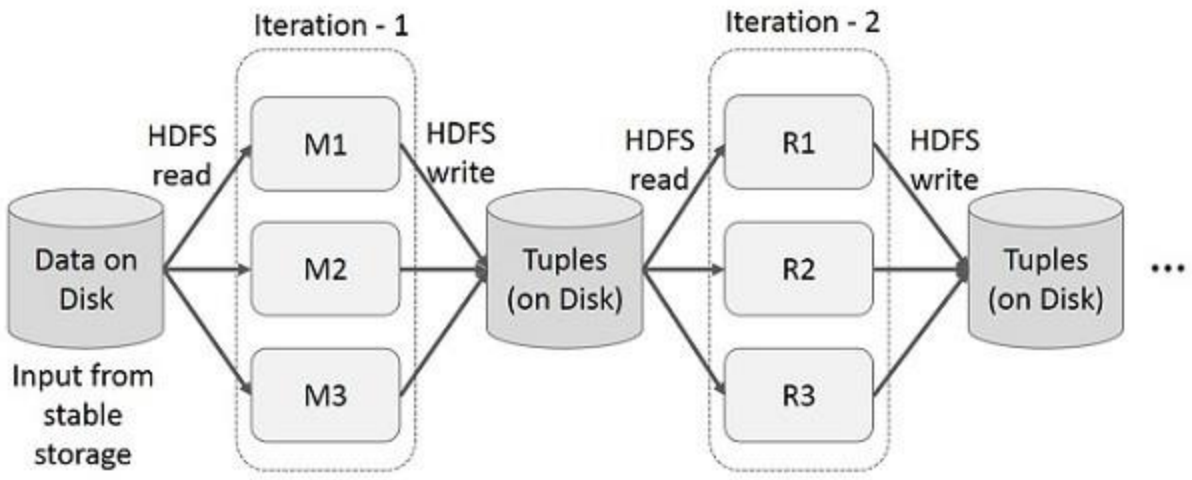


Figure 1Data in stable storage

Đặc điểm quan trọng của 1 RDD là số **partitions**. Một RDD bao gồm nhiều partition nhỏ, mỗi partition này đại diện cho 1 phần dữ liệu phân tán. Khái niệm partition là logical, tức là 1 node xử lý có thể chứa nhiều hơn 1 RDD partition. Theo mặc định, dữ liệu các partitions sẽ lưu trên memory. Thử tưởng tượng ta cần xử lý 1TB dữ liệu, nếu lưu hết trên mem tính ra thì cung khá tốn kém. Tất nhiên nếu ta có 1TB ram để xử lý thì tốt quá nhưng điều đó không cần thiết. Với việc chia nhỏ dữ liệu thành các partition và cơ chế lazy evaluation của Spark ta có thể chỉ cần vài chục GB ram và 1 chương trình được thiết kế tốt để xử lý 1TB dữ liệu, chỉ là sẽ chậm hơn có nhiều RAM thôi

2. Tại sao chúng ta cần phải sử dụng RDD ?

Các nhu cầu chính khi sử dụng RDD đó là :

- Các thuật toán bị lặp lại (Iterative algorithms)
- Công cụ khai tác dữ liệu tương tác (Interactive data mining tools)
- DSM (Distributed Shared Memory) có khả năng chịu lỗi trên một cụm và triển khải các công việc kém.
- Trong hệ điện toán phân tán, dữ liệu sẽ được lưu trữ trong hệ thống ổn định như HDFS hay Amazon S3. Việc này khiến cho công việc tính toán chậm hơn vì liên quan nhiều đến các khả năng I/O, sao chép và tuần tự hoá trong các tiến trình.

Trong hai trường hợp đầu tiên thì RDD lưu dữ liệu trong bộ nhớ và việc này giúp cải thiệu hiệu suất theo cấp độ magnitude.

Thách thức trong việc thiết kế RDD đó là việc tạo ra một API cung cấp khả năng chịu lỗi hiệu quả. Để làm được điều này, RDDs đã cung cấp một dạng bộ nhớ chung [shared memory] hạn chế dựa vào **coarse-grained transformation** thay vì **fine-grained** updates cho các trạng thái chung.

Spark làm rõ RDD thông qua API tích hợp ngôn ngữ. Khi đó, mỗi tập dữ liệu sẽ được biểu diễn dưới dạng một object và quá trình chuyển đổi RDD có liên quan đến việc sử dụng các phương thức của object này.

Apache Spark đánh giá RDD một cách lười biếng (lazy). RDD chỉ được gọi khi cần thiết, việc này giúp tiết kiệm thời gian và nâng cao hiệu quả.

3. Các tính năng của Spark RDD

RDD có các tính năng như là:

Figure 2 Các tính năng của RDD

3.1. Tính toán trên RAM (In-Memory Computation)

RDDs lưu trữ các kết quả trung gian trên bộ nhớ phân tán (RAM) thay bộ nhớ ổn định [stable storage] (Disk).

3.2. Đánh giá lười biếng (Lazy Evaluations)

Các phép biến đổi (transformations) trong Spark đều lười ở chỗ chúng không tính ngay kết quả mà thay vào đó, chúng chỉ nhớ các phép biến đổi được áp dụng trên các tập dữ liệu. Spark chỉ các phép biến đổi đó khi một hành động(actions) được yêu cầu.

3.3. Khả năng bất biến (Immutablility)

Dữ liệu an toàn để chia sẻ trên các tiến trình. Nó cũng có thể được tạo hoặc truy xuất bất cứ lúc nào giúp dễ dàng lưu vào bộ nhớ đệm, chia sẻ và nhân rộng. Do đó, nó là một cách để đạt được sự nhất quán trong tính toán.

3.4. Khả năng chịu lỗi (Fault Tolerance)

Spark RDD có khả năng chịu lỗi nhờ vào sự theo dõi thông tin dòng dữ liệu để tự động xây dựng lại dữ liệu bị mất khi bị lỗi. Để làm được điều này, mỗi RDD nhớ cách nó được tạo từ các tập dữ liệu khác (bằng các phép biến đổi như map, join hoặc groupBy) để tạo lại chính nó.

3.5. Sự bền bỉ (Persistence)

Người dùng có thể cho biết họ sẽ sử dụng lại những RDD nào và tự thiết lập khả năng lưu trữ cho họ (ví dụ: lưu trữ trong bộ nhớ hoặc trên Đĩa)

3.6. Phân vùng (Partitioning)

Phân vùng là đơn vị cơ bản của tính song song trong Spark RDD. Mỗi phân vùng là một phân chia dữ liệu hợp lý có thể thay đổi được. Người ta có thể tạo một phân vùng thông qua một số biến đổi trên các phân vùng hiện có.

3.7. Location-Stickness

RDD có khả năng xác định ưu tiên vị trí để tính toán các partition. Tùy chọn vị trí đề cập đến thông tin về vị trí của RDD. DAGScheduler đặt các phân vùng theo cách sao cho tác vụ gần với dữ liệu nhất có thể.

3.8. Coarse-gained Operation

Coarse-gained Operation áp dụng cho tất cả các phần tử trong bộ dữ liệu thông qua map hoặc filter hoặc nhóm theo các phép toán.

4. Các phép toán trong Spark RDD

RDD hỗ trợ 2 phép toán là :

- Tranformations
- Actions

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

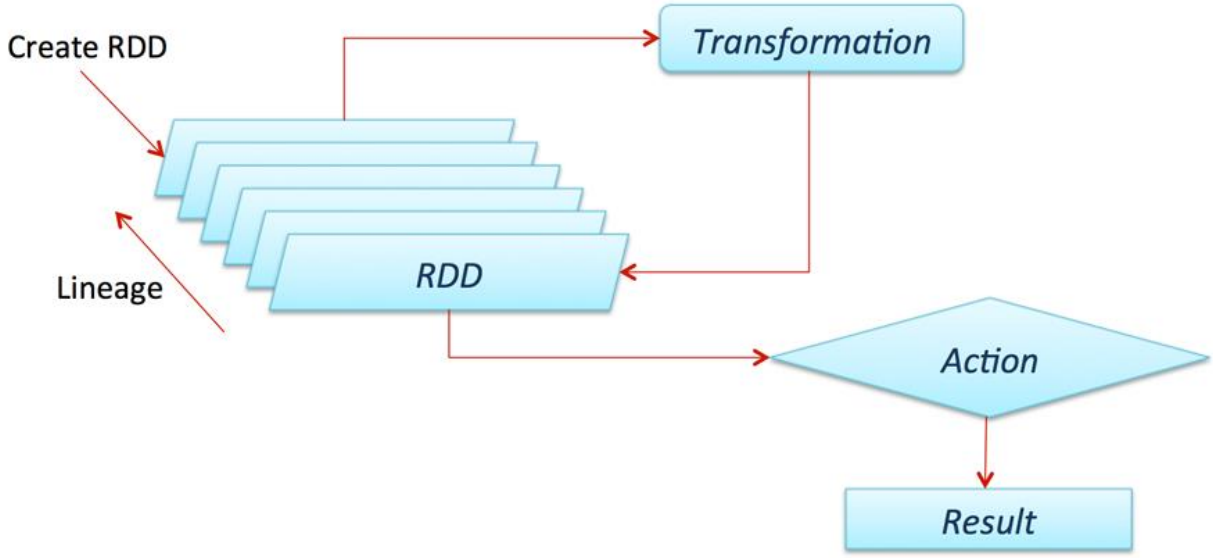


Figure 3 Cách thức RDD chuyển hoá data

4.1. Transformations

Spark RDD Tranformations là một hàm nhận vào một RDD và trả về một hay nhiều RDD khác. RDD ban đầu sẽ không bị thay đổi vì RDD mang tính bất biến (Immutability) mà nó sẽ sinh ra các RDD mới bằng các áp dụng các phương thức như Map(), filter(), reduceByKey(), ...

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

Transformations gồm các phương thức như:

- **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.

4.2. Actions

Actions trả về kết quả cuối cùng qua các tính toán RDD. Nó kích hoạt thực thi bằng cách sử dụng đồ thị tuyến tính (lineage graph) để load dữ liệu vào RDD gốc, thực hiện tất cả các phép biến đổi trung gian và trả về kết quả cuối cùng cho Driver để xử lý hoặc ghi dữ liệu xuống các công cụ lưu trữ

Actions gồm các phương thức như:

- **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. Phương thức chỉ sử dụng nếu map kết quả nhỏ vì tất cả dữ liệu sẽ được load lên memory của driver để tính toán và ta chỉ nên sử dụng trong tình huống số dòng nhỏ và lượng item khác nhau cũng nhỏ.
- **first:** lấy giá trị đầu tiên của dataset
- **max và min:** lấy lần lượcgiá trị lớn nhấy và nhỏ nhất của dataset
- **take và các method tương tự:** lấy một lượng giá trị từ trong RDD. take sẽ scan qua một partition trước và sử dụng kết quả để dự đoán số lượng partition cần phải lấy thể để thoả số lượng.

5. Giới hạn của Spark RDD

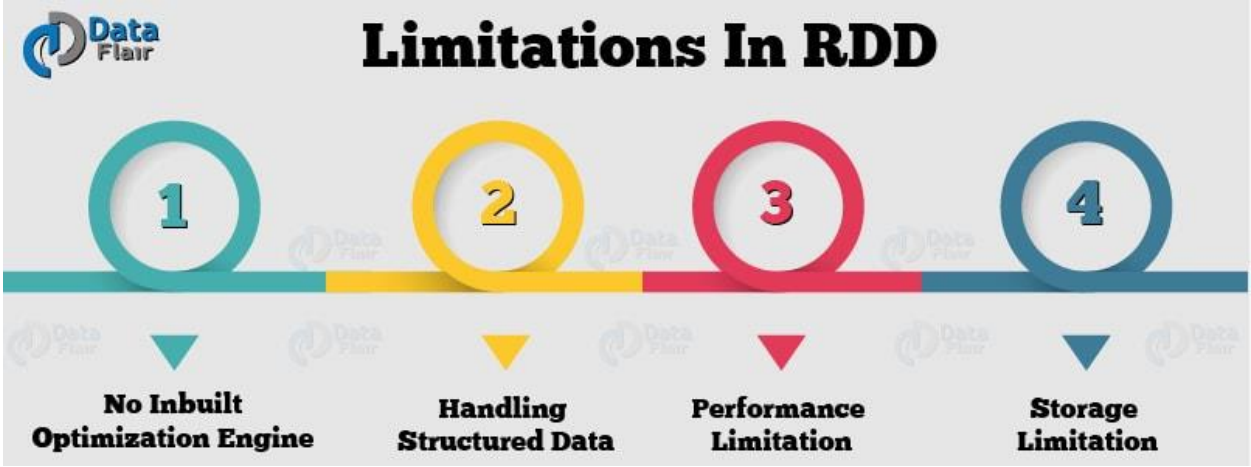


Figure 4 Các hạn chế của RDD

5.1. Không có công cụ tối ưu sẵn

Khi làm việc với cấu trúc dữ liệu, RDD không thể phát huy tối đa lợi thế từ bộ tối ưu của Spark như **catalyst optimizer** and **Tungsten execution engine**

5.2. Việc xử lý cấu trúc dữ liệu

Không giống như Dataframe và datasets, RDD không suy ra lược đồ của dữ liệu đã nhập và yêu cầu người dùng phải chỉ định nó.

5.3. Giới hạn hiệu suất

Là các đối tượng JVM trong bộ nhớ, RDD liên quan đến chi phí Thu gom rác và Tuần tự hóa Java, những thứ này rất tốn kém khi dữ liệu phát triển.

5.4. Giới hạn lưu trữ

RDDs suy giảm khi không có đủ bộ nhớ để lưu trữ chúng. Người ta cũng có thể lưu trữ partition của RDD đó trên đĩa do không đủ với RAM. Do đó, nó sẽ cung cấp hiệu suất tương tự như các hệ thống song song dữ liệu.

Apache Spark SQL - DataFrame

---

1. DataFrame là gì

DataFrame là một kiểu dữ liệu collection phân tán, được tổ chức thành các cột được đặt tên. Về mặt khái niệm, nó tương đương với các bảng quan hệ (relational tables) đi kèm với các kỹ thuật tối ưu tính toán.

DataFrame có thể được xây dựng từ nhiều nguồn dữ liệu khác nhau như Hive table, các file dữ liệu có cấu trúc hay bán cấu trúc (csv, json), các hệ cơ sở dữ liệu phổ biến (MySQL, MongoDB, Cassandra), hoặc RDDs hiện hành. API này được thiết kế cho các ứng dụng Big Data và Data Science hiện đại. Kiểu dữ liệu này được lấy cảm hứng từ DataFrame trong Lập trình R và Pandas trong Python hứa hẹn mang lại hiệu suất tính toán cao hơn.

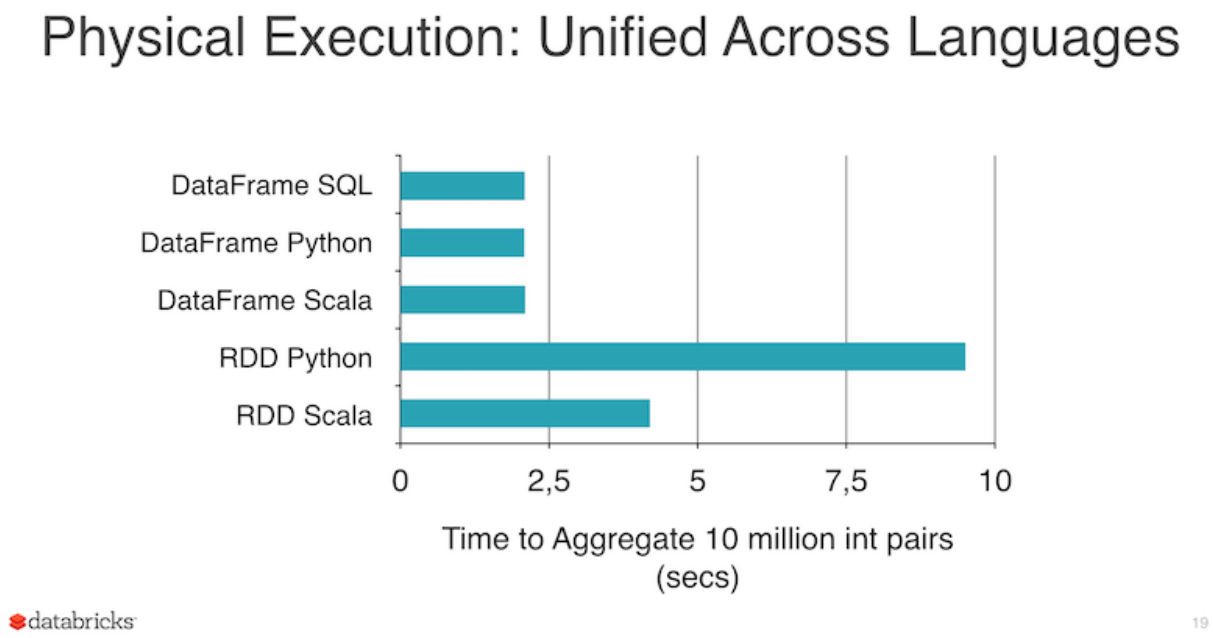


Figure 5 Tốc độ thực thi của các công cụ khác nhau

2. Tính năng của DataFrame

Một số tính năng đặc trưng của DataFrame như:

- Tối ưu hóa đầu vào: DataFrames sử dụng các công cụ tối ưu hóa đầu vào như **Catalyst Optimizer** cho phép xử lý dữ liệu hiệu quả. Ta có thể sử dụng cùng một công cụ cho tất cả các API Python, Java, Scala và R DataFrame.

- Xử lý lớn: DataFrames có thể tích hợp với nhiều công cụ BigData khác và cho phép xử lý megabyte đến petabyte dữ liệu cùng một lúc.
- Tính linh hoạt: DataFrames, giống như RDD, có thể hỗ trợ nhiều định dạng dữ liệu khác nhau, chẳng hạn như CSV, Cassandra, v.v.
- Quản lý bộ nhớ tùy chỉnh: Trong RDD, dữ liệu được lưu trữ trong bộ nhớ RAM, trong khi DataFrames lưu trữ dữ liệu off-heap (bên ngoài không gian chính của Java Heap, nhưng vẫn bên trong RAM), do đó làm giảm các collection quá tải dư thừa.
- Xử lý dữ liệu có cấu trúc: DataFrames cung cấp một cái nhìn sơ lược về dữ liệu. Ở đây, dữ liệu có một số ý nghĩa đối với nó khi nó được lưu trữ

3. SQL Context

SQLContext là một lớp và được sử dụng để khởi tạo các chức năng của Spark SQL. Đối tượng SparkContext là bắt buộc để có thể khởi tạo đối tượng SQLContext. Lệnh sau được sử dụng để khởi tạo SparkContext.

```
import sys
from pyspark import SparkContext, SparkConf
from pyspark.sql import SQLContext

# Config Spark context
conf = SparkConf().setMaster("local").setAppName("example app")
sc = SparkContext.getOrCreate(conf=conf)

# Config SQL context
sqlContext = SQLContext(sc)
```

4. Tương tác với Spark DataFrame

Đọc file csv

Để đọc file, pyspark.shell cung cấp phương thức read.csv() cho phép đọc file csv vào dataframe.

```
from pyspark.shell import spark
from pyspark.sql.types import *

# Tạo DataFrame từ file CSV
df_data = spark.read.csv('drive/My Drive/Colab Notebooks/click_data_sample
.csv')
print(df_data.head(5))
```

Output:

```
[Row(_c0='click.at', _c1='user.id', _c2='campaign.id'),
 Row(_c0='2015-04-27 20:40:40', _c1='144012', _c2='Campaign077'),
 Row(_c0='2015-04-27 00:27:55', _c1='24485', _c2='Campaign063'),
 Row(_c0='2015-04-27 00:28:13', _c1='24485', _c2='Campaign063'),
 Row(_c0='2015-04-27 00:33:42', _c1='24485', _c2='Campaign038')]
```

Đổi tên cột

Ta có thể dễ dàng thay đổi tên cột bằng `withColumnRenamed.` Tuy nhiên, về cơ bản thì DataFrame là bất biến (immutable) nên khi thay đổi thì 1 DataFrame mới sẽ được tạo ra.

```
new_df = df_data.withColumnRenamed("_c0", "access_time")\
                .withColumnRenamed("_c1", "userID")\
                .withColumnRenamed("_c2", "campaignID")
print(new_df.printSchema())
```

Output:

```
root
 |-- access_time: string (nullable = true)
 |-- userID: string (nullable = true)
 |-- campaignID: string (nullable = true)

None
```

Query bằng SQL

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

Có 1 lưu ý là: Ta cũng có thể viết subquery nhưng subquery cần được gán Alias, nếu không sẽ bị (Syntax error).

Ta thử tìm các dòng có cột `campaignID` có giá trị là `Campaign047`

```
#SQL query

new_df.registerTempTable("whole_log_table")

# Query
print (sqlContext.sql(" SELECT * FROM whole_log_table where campaignID == 'Campaign047' ").count())
```

Output:

```
18081
```

Ta in thử 5 dòng đầu trong đó

```
print(sqlContext.sql(" SELECT * FROM whole_log_table where campaignID == 'Campaign047' ").show(5))
```

Output:

```
+-------------------+------+-----------+
|        access_time|userID| campaignID|
+-------------------+------+-----------+
|2015-04-27 05:26:14| 14151|Campaign047|
|2015-04-27 05:26:32| 14151|Campaign047|
|2015-04-27 05:26:34| 14151|Campaign047|
|2015-04-27 05:27:47| 14151|Campaign047|
|2015-04-27 05:28:16| 14151|Campaign047|
+-------------------+------+-----------+
only showing top 5 rows
```

Ta cũng có thể query linh động hơn

```
#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())
```

Output:

```
Campaign001
+----------+
|access_num|
+----------+
|      2407|
+----------+
```

```
None
Campaign002
+----------+
|access_num|
+----------+
|      1674|
+----------+

none
```

Đối với trường hợp subquery

```
#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 userI
D) subquery_alias WHERE first_access_date < '2015-04-28'").show(5))
```

Output:

```
+-----------+
|first_count|
+-----------+
|      20480|
+-----------+

None
```

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 Filer

```
#Ví dụ filter
print(new_df.filter(new_df["access_time"] > "2015-05-01").show(3))
```

Output:

```
+-------------------+-------+-----------+
|        access_time| userID| campaignID|
+-------------------+-------+-----------+
|           click.at|user.id|campaign.id|
|2015-05-01 22:11:57| 114157|Campaign002|
|2015-05-01 23:36:25|  93708|Campaign055|
+-------------------+-------+-----------+
```

```
only showing top 3 rows

None
```

Ví dụ với select

```
#Ví dụ select
print(new_df.select("access_time", "userID").show(3))
```

Output:

```
+-------------------+-------+
|        access_time| userID|
+-------------------+-------+
|           click.at|user.id|
|2015-04-27 20:40:40| 144012|
|2015-04-27 00:27:55|  24485|
+-------------------+-------+
only showing top 3 rows

None
```

Tài liệu tham khảo

Spark Properties:

[1] Spark Configuration

Spark RDD:

[2] Apache Spark Fundamentals - Phần 2: Spark Core và RDD

[3] Spark RDD – Introduction, Features & Operations of RDD

[4] Apache Spark RDD

Spark DataFrame:

[5] Xử lý dữ liệu với Spark DataFrame

[6] Spark SQL - DataFrame