# Báo cáo tháng 9

# anhnhv

[Báo cáo tháng 9 1](#_Toc525866694)

[1. Tìm hiểu về Spark 2](#_Toc525866696)

[2. Ưu điểm của Spark so với Hadoop 4](#_Toc525866697)

[3. RDD và Dataframe 6](#_Toc525866698)

[4. Lập trình với thư viện Pyspark trên Python 7](#_Toc525866699)

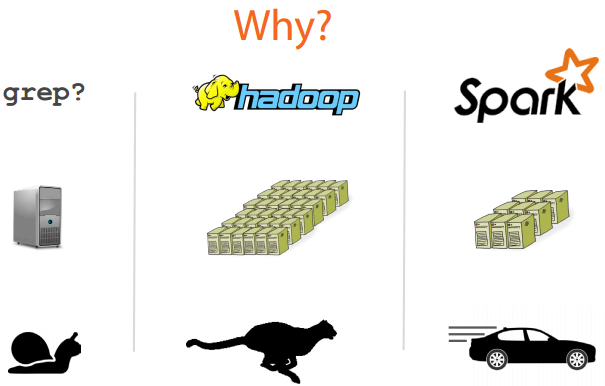
[5. Tài liệu tham khảo 11](#_Toc525866700)

## Tìm hiểu về Spark

Apache Spark là một *lightning-fast cluster computing technology*. Nó dựa trên Hadoop MapReduce và mở rộng mô hình MapReduce để sử dụng hiệu quả cho nhiều loại tính toán hơn, bao gồm các *interactive queries* và *stream processing*. Đặc điểm của Spark là **in-memory cluster computing** làm tăng tốc độ xử lý của một ứng dụng. [[1](#_Tài_liệu_tham)]

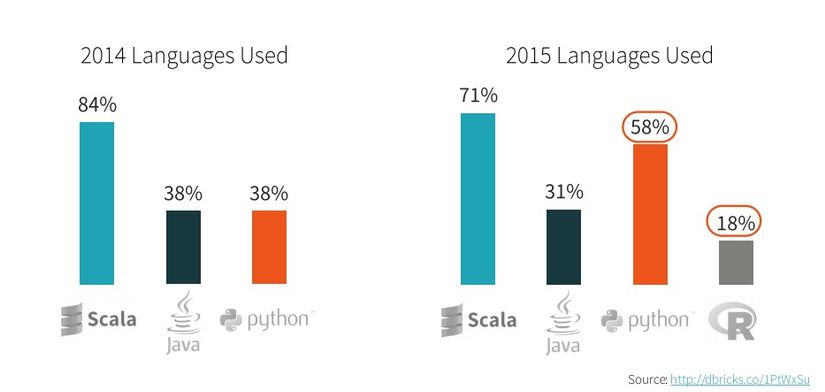
1. Đặc điểm của Spark

* Tốc độ: Spark giúp chạy một ứng dụng trong cụm Hadoop, nhanh hơn tới 100 lần trong bộ nhớ và nhanh hơn gấp 10 lần khi chạy trên đĩa. Điều này là có thể bằng cách giảm số lượng hoạt động đọc/ghi vào đĩa. Nó lưu trữ dữ liệu xử lý trung gian trong bộ nhớ.



Hình 1. So sánh tốc độ của Spark với Hadoop

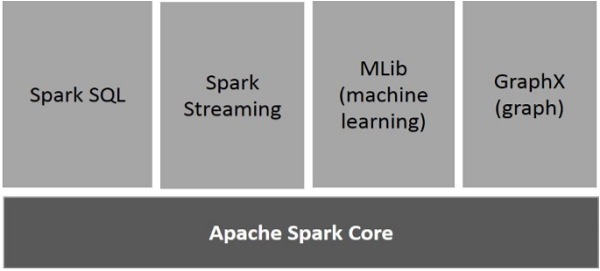
* Hỗ trợ nhiều ngôn ngữ: Spark cung cấp các API tích hợp trong Java, Scala hoặc Python, R. Do đó, bạn có thể viết các ứng dụng bằng các ngôn ngữ khác nhau. Spark có tới 80 toán tử cấp cao để truy vấn tương tác.



Hình 2. Tỷ lệ LTV sử dụng các ngôn ngữ được hỗ trợ bởi Spark

* Advanced Analytics: Spark không chỉ hỗ trợ ‘Map’ và ‘Reduce’ mà còn hỗ trợ SQL queries, Streaming data, Machine learning (ML), and Graph algorithms..

1. Thành phần của Spark



Hình 3. Các thành phần của Spark

* Apache Spark Core:

Spark Core là công cụ thực thi chung cơ bản cho nền tảng Spark mà tất cả các chức năng khác được xây dựng dựa trên. Nó cung cấp tính toán In-Memory và tham chiếu dữ liệu trong các hệ thống lưu trữ ngoài.

* Spark SQL:

Spark SQL là một thành phần phía trên của Spark Core, giới thiệu một dữ liệu trừu tượng mới được gọi là SchemaRDD, hỗ trợ cho dữ liệu có cấu trúc và bán cấu trúc.

* Spark Streaming

Spark Streaming tận dụng khả năng *scheduling* nhanh của Spark Core để thực hiện phân tích trực tuyến. Nó nhập dữ liệu theo các *mini-batches* và thực hiện các *RDD* *transformations* trên các *mini-batches* đó.

* MLlib (Thư viện học máy):

MLlib là một framework máy học máy phân tán trên Spark vì kiến ​​trúc Spark dựa trên bộ nhớ phân tán. Đó là, theo tiêu chuẩn, được thực hiện bởi các nhà phát triển MLlib dựa vào việc triển khai Alternating Least Squares (ALS). Spark MLlib nhanh gấp 9 lần phiên bản của Apache Mahoop trên đĩa (trước khi Mahout nhận được Spark interface).

* GraphX:

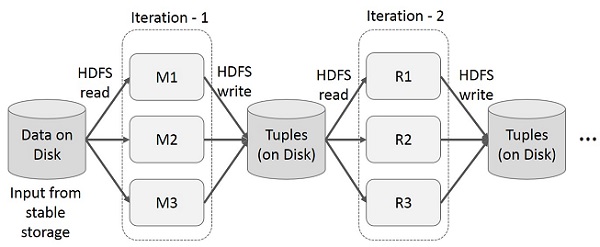
GraphX ​​là một framework xử lý đồ thị phân tán phía trên Spark. Nó cung cấp một API để biểu diễn đồ thị tính toán, mô hình hóa các đồ thị do người dùng định nghĩa bằng cách sử dụng API trừu tượng Pregel. Nó cũng cung cấp một thời gian chạy tối ưu hóa cho sự trừu tượng này.

## Ưu điểm của Spark so với Hadoop

Tốc độ xử lý của Spark nhanh hơn rất nhiều so với Hadoop, thể hiện qua các Iterative Operations và Interactive Operations [[2](#_Tài_liệu_tham)], đều là do dữ liệu trung gian được lưu trữ trên RAM thay vì Disk.

1. Iterative Operations on Hadoop

Kết quả của các Iterative Operation sẽ được lưu trữ trên Disk. Khi một Iteration khác cần kết quả đó làm input thì phải đọc từ disk vào, gây ra chi phí đáng kể do sao chép dữ liệu, disk I/O và tuần tự hóa, làm cho hệ thống chậm.



Hình 4. Iterative Operations on Hadoop

1. Iterative Operations on Spark

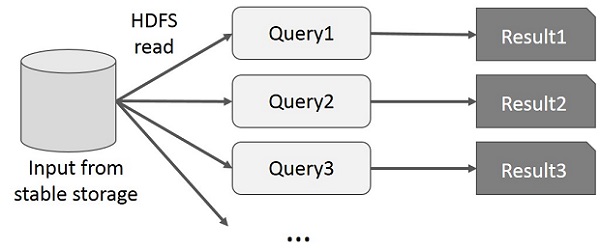
Kết quả trung gian được lưu trữ trong Distriuted Memmory (RAM) thay cho Stable Storage (Disk) và làm cho hệ thống nhanh hơn. Nếu RAM không đủ thì nó sẽ lưu trữ kết quả trung gian đó trên Disk.



Hình 5. Iterative Operations on Spark

1. Interactive Operations on Hadoop

Các truy vấn trên cùng một tập dữ liệu khiến hoạt động Disk I/O lặp lại nhiều lần, chiếm phần lớn thời gian thực thi của ứng dụng.



Hình 6. Interactive Operations on Hadoop

1. Interactive Operaions on Spark

Nếu các truy vấn khác nhau được chạy trên cùng một tập dữ liệu nhiều lần, dữ liệu cụ thể này có thể được lưu trong RAM để có thời gian thực hiện tốt hơn.



Hình 7. Interactive Operations on Spark

## RDD và Dataframe

1. RDD [[2](#_Tài_liệu_tham)]

Resilient distributed dataset là tập dữ liệu phân tán phục hồi mà các dữ liệu này được phân tán vào các node của cluster để thực hiện tính toán song song.

RDD hỗ trợ 2 loại phương thức: transformations và actions.

* Transformations: Qua 1 phương thức transformations thì sẽ cho phép tạo mới 1 RDD từ 1 RDD đã tồn tại.

Tất cả các transformation đều là **lazy**, có nghĩa là các transformation này sẽ không thực hiện tính toán ngay mà chúng sẽ được lưu lại thành dữ liệu và chỉ thực hiện tính toán khi 1 action được gọi để yêu cầu trả về kết quả cho driver program. Lazy ở đây tốt ở chỗ là chúng ta không phải tạo ra 1 tập dữ liệu và thực hiện lưu trữ chúng trong file rồi sau đó các actions sẽ phải tìm cách truy cập đến các file lưu trữ này mà thay vào đó, chúng ta sẽ tiết kiệm được thời gian (thời gian truy cập và thời gian lưu trữ) và không gian lưu trữ kết quả (RDD) qua transformation.

Một số hàm transformation như: map, flatMap, filter, groupByKey, reduceByKey, sortByKey…

* Actions: Qua 1 phương thức actions thì sẽ cho phép trả về 1 giá trị cho driver program sau khi chạy tính toán trên tập dữ liệu, như: collect, take, first, reduce, count, saveAsTextFile, foreach…

1. Dataframe [[2](#_Tài_liệu_tham)]

Giống như RDD, DataFrame là một tập dữ liệu phân tán không thay đổi. Không giống như RDD, dữ liệu được tổ chức thành các cột được đặt tên, như một bảng trong cơ sở dữ liệu quan hệ. Được thiết kế để làm cho các bộ dữ liệu lớn xử lý dễ dàng hơn, DataFrame cho phép các nhà phát triển áp đặt một cấu trúc vào tập dữ liệu phân tán, cho phép trừu tượng mức cao hơn; nó cung cấp một *domain specific language API* để thao tác dữ liệu phân tán, làm cho Spark có thể tiếp cận với đối tượng rộng hơn, ngoài các kỹ sư dữ liệu chuyên ngành.

Ưu điểm của Dataframe so với RDD:

* DataFrame được xây dựng trên Spark SQL engine, nó sử dụng Catalyst optimizer để tạo ra một kế hoạch truy vấn logic và vật lý được tối ưu hóa, cung cấp hiệu quả không gian và hiệu suất tốc độ.
* Spark sử dụng Tungsten Encoders có thể serialize/deserialize các đối tượng JVM một cách hiệu quả cũng như tạo ra bytecode nhỏ gọn có thể thực thi ở tốc độ cao hơn.

## Lập trình với thư viện Pyspark trên Python

1. So sánh các phương pháp tìm max

|  |  |
| --- | --- |
| Code | Time (s) |
| df.groupBy().max('VolData','NoSMS','DurationVoice') | 33.30 |
| for f in features:  max\_values.append(df.groupBy().max(f)) | 98.17 |
| df.select([max(f) for f in features]) | 33.34 |
| df.groupBy().agg(max('VolData'),max('NoSMS'),  max('DurationVoice')) | 32.67 |
| spark.sql(  'SELECT MAX(VolData),MAX(NoSMS),MAX(DurationVoice))  FROM user') | 34.23 |
| max\_values = rdd.map(lambda x: [x[f] for f in features])  .reduce(lambda a, b:  [max(a[i], b[i]) for i in range(n\_features)]) | 137.69 |

1. Hàm persist()

Thường thì mỗi transformation sẽ thực hiện chạy lại mỗi khi bạn chạy actions. Tuy nhiên, có thể persist 1 RDD trong bộ nhớ sử dụng phương thức *persist* (hoặc *caching*). Điều này rất tiện lợi cho việc tính toán ngay tại bộ nhớ trong và tái sử dụng chúng cho actions khác trên tập dữ liệu, nó làm cho việc tính toán thực hiện nhanh hơn và khả năng chịu lỗi tốt. Cụ thể, lần đầu tiên tính toán với 1 actions, nó sẽ lưu trữ tại bộ nhớ trong trên các node. Nếu 1 phần nào đó của RDD bị mất, nó sẽ tự động tính toán lại sử dụng transformation được tạo ban đầu.[[3](#_Tài_liệu_tham)]

Mỗi RDD được lưu trữ với các mức khác nhau. Ví dụ, có thể *persist* dữ liệu trên ổ cứng, bộ nhớ trong, hoặc sao lặp qua các node. Các mức lưu trữ tùy thuộc truyền đối tượng org.apache.spark.storage.StorageLevel vào hàm *persist()*. Phương thức *cache()* là cách viết tắt cho phương thức *persist* với mức lưu trữ mặc định, cụ thể là *StorageLevel.MEMORY\_ONLY*.

Sau đây là các mức lưu trữ :

|  |  |
| --- | --- |
| Storage Level | Ý nghĩa |
| MEMORY\_ONLY | Lưu trữ RDD như đối tượng deserialized Java trên JVM. Nếu RDD vượt quá bộ nhớ trong thì 1 số vùng sẽ không được cached và nó sẽ tính toán lại mỗi khi nó cần. Đây là mức lưu trữ mặc định. |
| DISK\_ONLY | Lưu trữ các vùng RDD chỉ trên ổ cứng. |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, ... | Giống như các mức trên nhưng sao lặp mỗi vùng RDD trên 2 cluster nodes. |

Để thay đổi mức lưu trữ, sử dụng phương thức *apply()* của đối tượng StorageLevel.

Dưới đây là đoạn source code và bảng so sánh thời gian thực thi bài toán: “Tìm top 10 movies tương đồng với các movie cho trước” của bộ dữ liệu MovieLens **ml-100k**:

# (movie1, movie2) = > (rating1, rating2), (rating1, rating2) ...

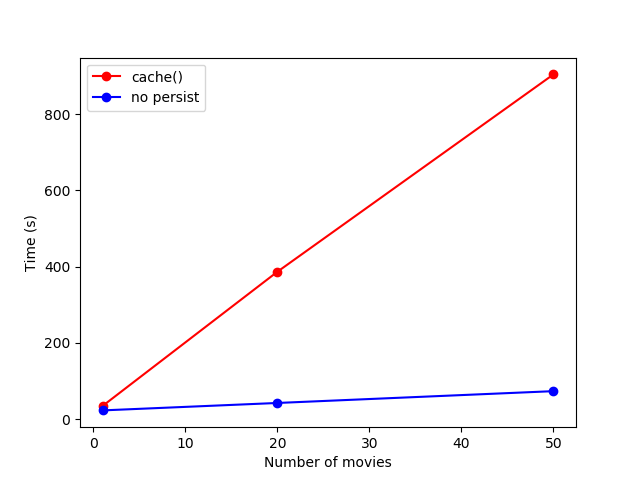
moviePairRatings = moviePairs.groupByKey()  
  
moviePairSimilarities = moviePairRatings.mapValues(computeCosineSimilarity).cache()

# Extract similarities for the movie we care about that are "good".  
scoreThreshold = 0.97  
coOccurrenceThreshold = 50  
# Filter for movies with this sim that are "good" as defined by  
# our quality thresholds above  
s = time.time()  
for i in range(1, 21):  
 movieID = i  
 filteredResults = moviePairSimilarities.filter(  
 lambda pairSim: (pairSim[0][0] == movieID or pairSim[0][1] == movieID)  
 and pairSim[1][0] > scoreThreshold  
 and pairSim[1][1] > coOccurrenceThreshold)  
  
 # Sort by quality score.  
 results = filteredResults.map(lambda pairSim: (pairSim[1], pairSim[0]))

.sortByKey(ascending=False).take(10)

Kết quả:

|  |  |  |  |
| --- | --- | --- | --- |
| Number of movies | 1 | 20 | 50 |
| persist(MEMORY\_ONLY) = cache() | 23 | 42.5 | 73.5 |
| no persist | 34.6 | 386.8 | 904.6 |



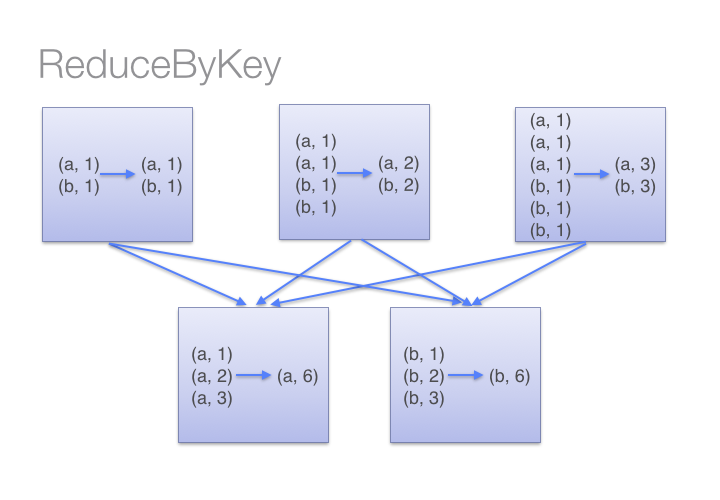
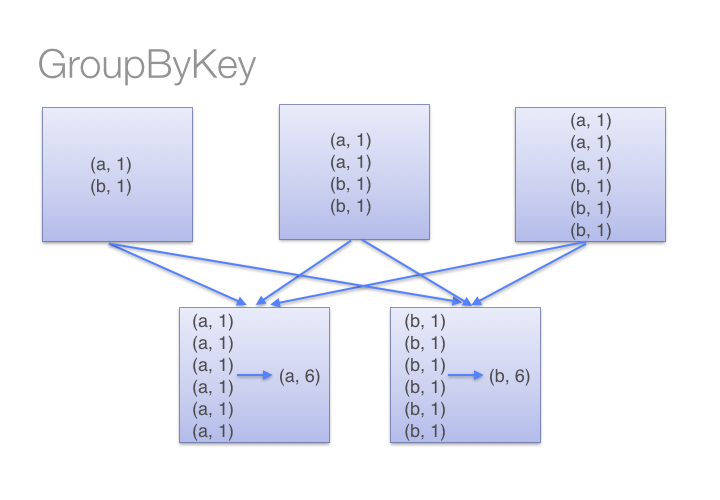
Hình 8. So sánh thời gian thực thi khi sử dụng cache()

1. So sánh groupByKey và reduceByKey [[4](#_Tài_liệu_tham)]

Khi gọi *groupByKey*, tất cả các cặp key-value được shuffle về cùng một partition khiến rất nhiều dữ liệu không cần thiết được chuyển qua mạng giữa các partition và có rất nhiều cặp key-value có thể không lưu trữ đủ trong RAM, gây lỗi. Phiên bản Spark sau này có thể xử lý bằng cách đưa một phần vào Disk, nhưng vẫn nên tránh vì I/O trên Disk làm hiệu suất bị ảnh hưởng nghiêm trọng.

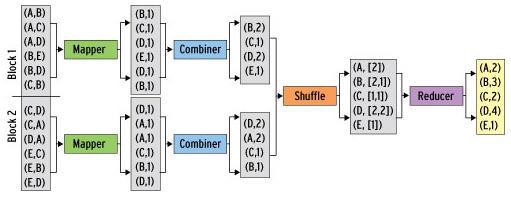
*reduceByKey* hoạt động tốt hơn nhiều trên tập dữ liệu lớn. Đó là bởi vì Spark sẽ kết hợp các value có cùng key trên mỗi partition trước khi shuffle dữ liệu.

Vì vậy nên tránh sử dụng *groupByKey*, thay vào đó có thể sử dụng các phương thức thích hợp hơn như: *reduceByKey*, *combineByKey*, *foldByKey*…



Hình 9. So sánh groupByKey và reduceByKey

Ưu điểm của *reduceByKey* so với *groupByKey* tương tự như việc sử dụng *combiner function* trong mô hình MapReduce Hadoop.



Hình 10. Combiner function trong MapReduce Hadoop

Bảng sau đây cho thấy ưu điểm của groupByKey khi chạy local là xử lý một tập value cùng key nhanh hơn là reduce từng cặp, nhưng lại không tốt với dữ liệu lớn.

|  |  |  |  |
| --- | --- | --- | --- |
| Data | Number of records | groupByKey | reduceByKey |
| All\_User0 | 6,513,721 | 71s | 259.83s |
| All\_User0-10 | 65,523,580 | Out of memory | 5476.9s |

## Tài liệu tham khảo

1. <https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm>
2. <https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html>
3. <https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm>
4. <https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html>