

APACHE SPARK

Dữ liệu lớn

ThS. Nguyễn Hồ Duy Trí
trinhhd@uit.edu.vn

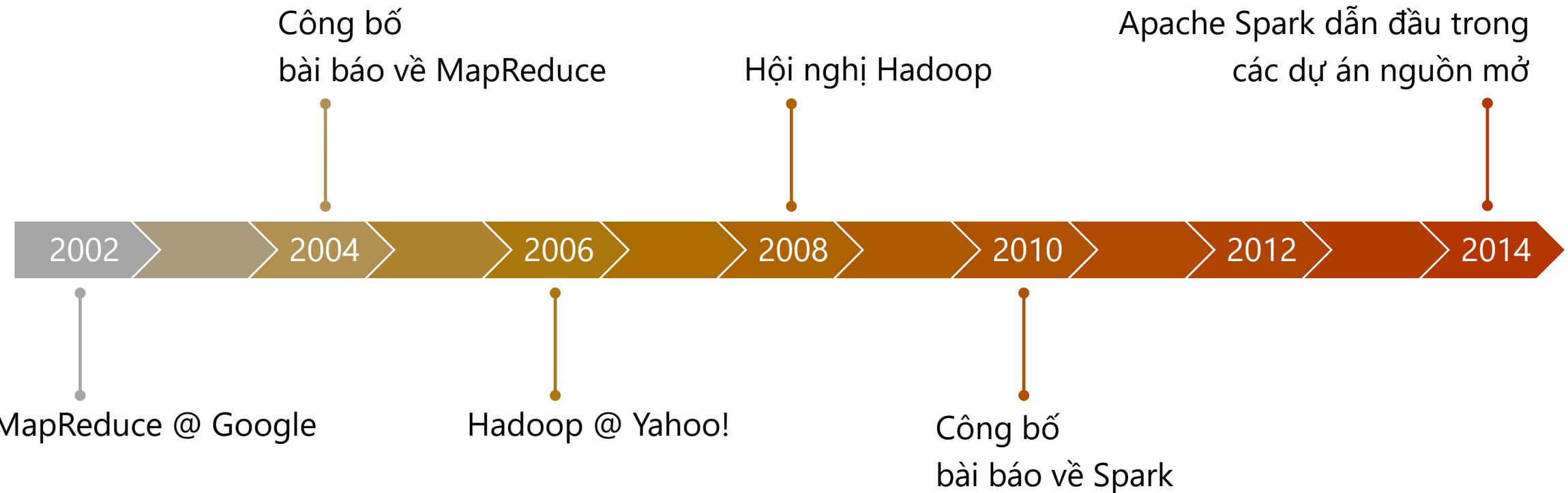


TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA HỆ THỐNG THÔNG TIN

2020



Lịch sử





- Khởi đầu ở AMPLab UC Berkeley
- Trưởng dự án: Dr. Matei Zaharia
- Dự án được mở mã nguồn năm 2010
- Bài báo đầu tiên về RDD đăng tải năm 2012
- Phiên bản 1.0 ra mắt vào tháng 05/2014. Phiên bản hiện tại 3.0.1



databricks

- Databricks là công ty được thành lập để hỗ trợ Spark và tất cả các công nghệ có liên quan. Matei hiện là CTO

Là dự án mã nguồn mở được quan tâm

nhiều nhất trên thế giới



Hơn **1000** người đóng góp thường xuyên

từ **250** công ty khác nhau



Tencent 腾讯



yahoo!

amazon

AUTODESK®

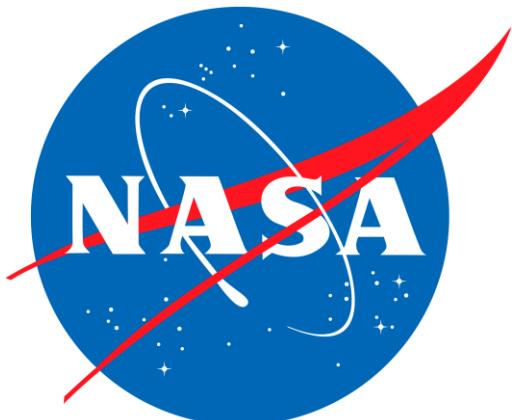
OpenTable®

SK telecom

GROUPON®

Alibaba Group

Baidu 百度



NOKIA

IBM

OOYALA®

Яндекс
eBay

shopify

NỘI DUNG



APACHE **Spark** TỔNG QUAN



Những vấn đề của MapReduce

MapReduce đơn giản hóa việc xử lý dữ liệu lớn, nhưng có hai vấn đề nan giải

- **Khả năng lập trình:** các hàm viết trên mô hình map/reduce khá phức tạp
 - Tất cả dữ liệu buộc phải xử lý trong hai quá trình duy nhất là Map và Reduce
 - Thiếu các chức năng thông dụng như: lọc, kết, hội, giao, nhóm dữ liệu theo yêu cầu...
- **Tốc độ:** MapReduce không hiệu quả đối với các ứng dụng chia sẻ dữ liệu qua nhiều bước
 - Dựa vào “Luồng dữ liệu tuần hoàn” từ đĩa sang đĩa (HDFS)
 - Đọc và ghi dữ liệu lên đĩa trước và sau quá trình Map/Reduce
 - Rất khó cài đặt các thuật toán lặp (máy học), truy vấn tương tác

- Nguyên bản chỉ hỗ trợ ngôn ngữ Java
 - Cần hỗ trợ các ngôn ngữ cần thiết khác
- Chỉ xử lý hàng loạt (batch processing)
 - Bổ sung khả năng xử lý tương tác và dữ liệu truyền theo thời gian thực

Ý tưởng

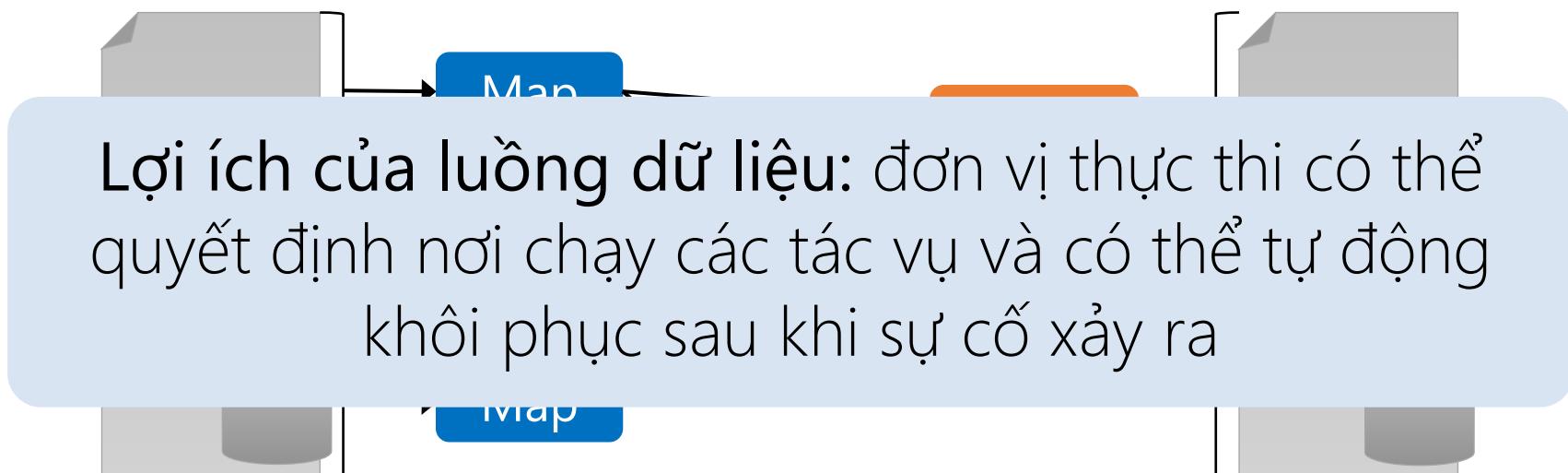
- Không giới hạn mô hình tính toán chỉ là MapReduce
- Sử dụng bộ nhớ hệ thống
 - Tránh lưu kết quả trung gian vào đĩa
 - Lưu trữ dữ liệu tạm thời trên bộ nhớ cho các truy vấn lặp lại
- Tương thích với Hadoop

Mục tiêu

- Mở rộng mô hình MapReduce để hỗ trợ tốt hơn hai loại ứng dụng phân tích phổ biến:
 - Các thuật toán lặp lại (máy học, xử lý đồ thị)
 - Khai thác dữ liệu tương tác
- Nâng cao khả năng lập trình:
 - Sử dụng ngôn ngữ lập trình Scala
 - Cho phép khả năng lập trình tương tác nhờ trình thông dịch Scala

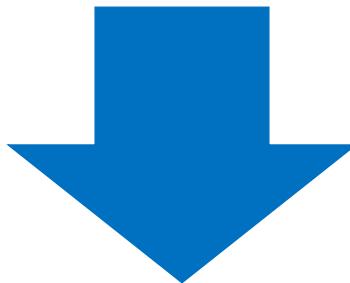
Động lực

Hầu hết các mô hình lập trình phân tán hiện tại đều sử dụng luồng dữ liệu tuần hoàn từ nguồn lưu trữ ổn định này đến nguồn lưu trữ ổn định khác



- Luồng dữ liệu tuần hoàn không hiệu quả đối với các ứng dụng sử dụng lại nhiều lần một bộ dữ liệu đang hoạt động
 - Các thuật toán lặp lại (máy học, xử lý đồ thị)
 - Các công cụ khai thác dữ liệu tương tác (R, Excel, Python)
- Với các framework hiện tại, ứng dụng sẽ phải tải lại dữ liệu từ nơi lưu trữ trong mỗi lần truy vấn

- Các thuật toán lặp phức tạp
- Truy vấn/lập trình tương tác
- Xử lý luồng dữ liệu thời gian thực



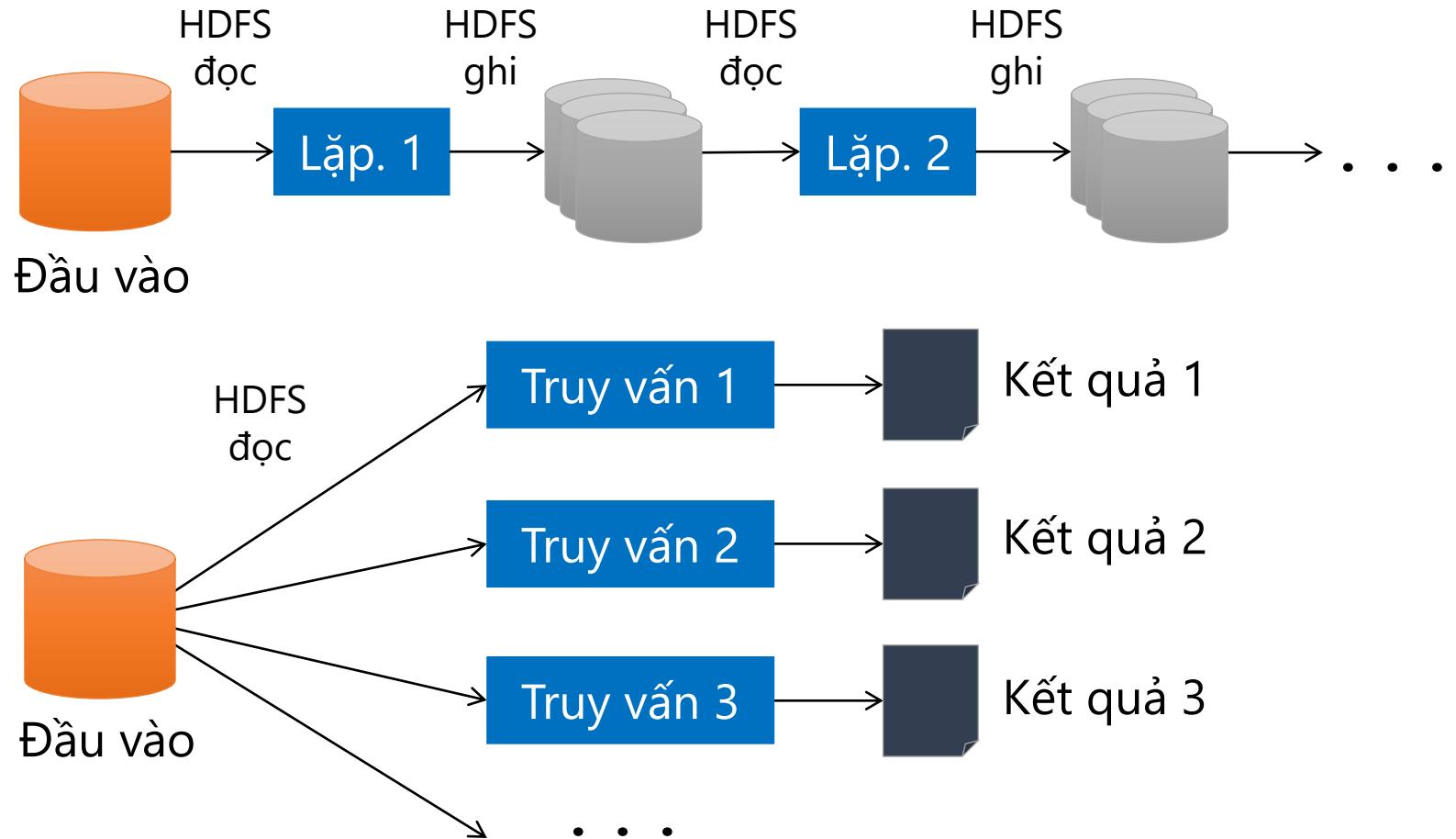
Tất cả các tác vụ trên đều cần chia sẻ và truyền dữ liệu hiệu quả

- Giá RAM ngày càng rẻ hơn
- Dung lượng RAM ở máy tính bình thường ngày càng lớn
- Nhờ đó, một cụm máy phân tán sẽ có nhiều RAM



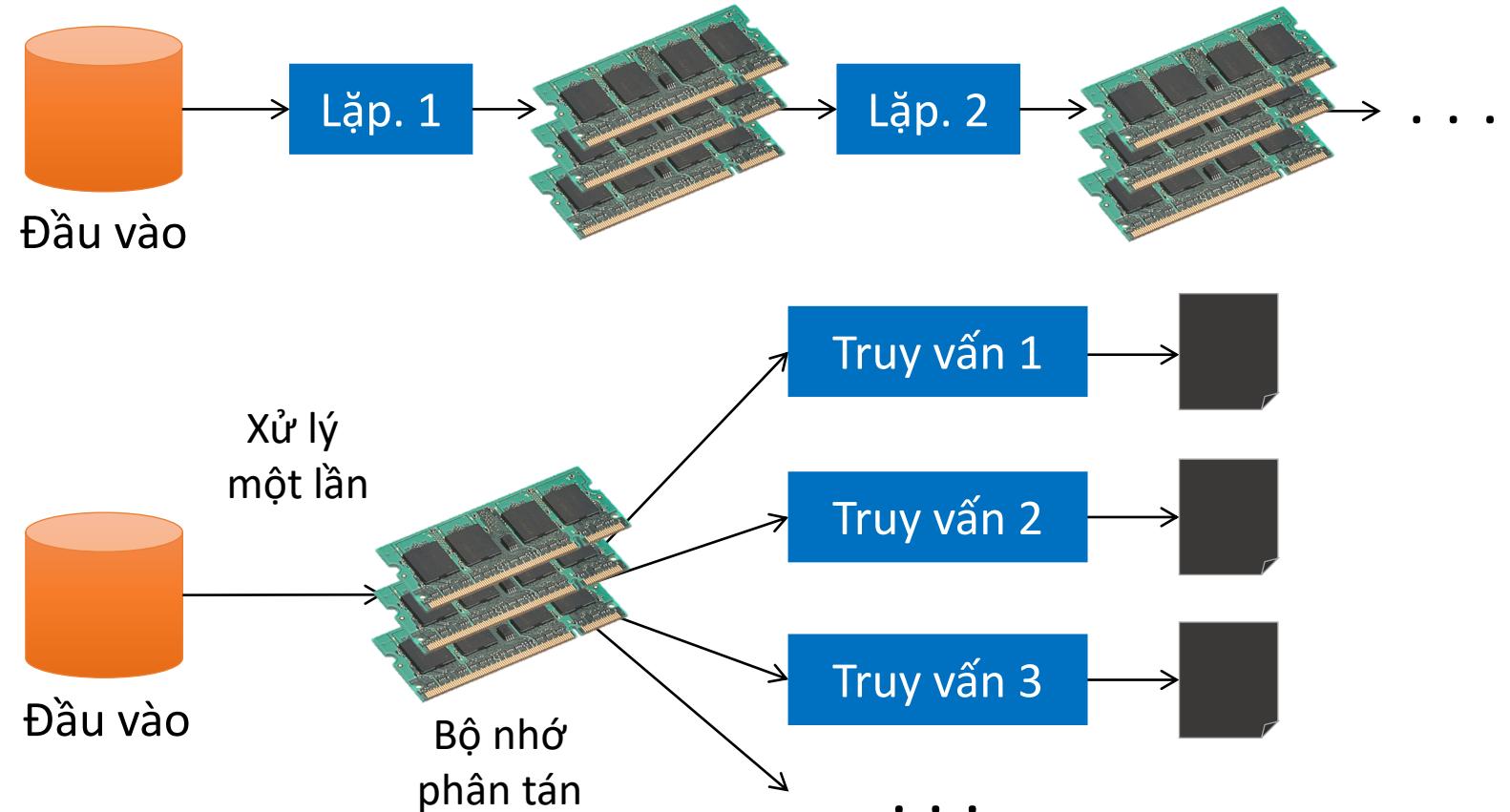
Nhiều tác vụ xử lý, lưu trữ và truyền dữ liệu
nên sử dụng RAM

Trong MapReduce



Tốc độ chậm do nhân bản dữ liệu và I/O của đĩa cứng

Thay đổi



Nhanh hơn mạng và đĩa cứng từ 10-100x

Luồng thực thi công việc

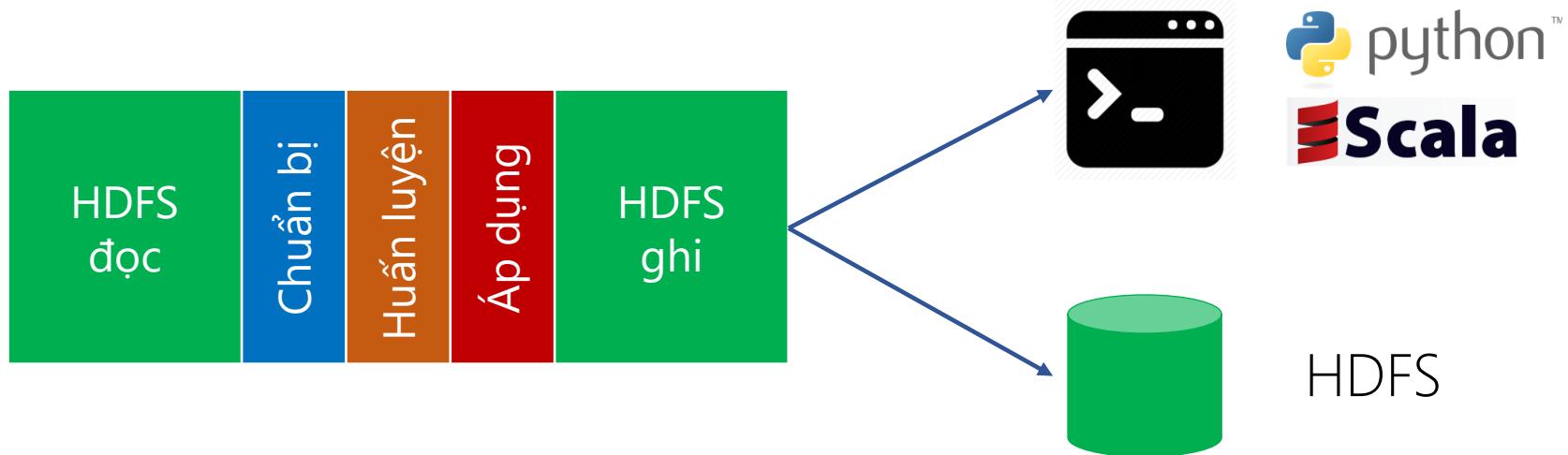
Trong MapReduce

Những quá trình độc lập



Luồng thực thi công việc

Trong Spark



Lập trình
tương tác

HDFS

Tóm tắt

- Hỗ trợ tốt hơn cho quá trình xử lý thời gian thực
- Tận dụng ưu thế của RAM nhiều nhất có thể
- Phân tán quy mô lớn hơn
- “Không cần phát minh lại bánh xe”

APACHE Spark

RDD



Ý tưởng

- Tập hợp các đối tượng (bản ghi) trong một đơn vị
- Đơn vị lưu trữ này tự động phân bổ trên các cụm máy
- Các thao tác xử lý song song được xây dựng dựa trên đơn vị này
- Có khả năng chịu lỗi mà không cần nhân bản (dựa vào xuất xứ hình thành đối tượng)

Mô hình lập trình

- Ngôn ngữ lập trình cấp cao để xây dựng luồng công việc (Scala)
- Hệ thống biên dịch thành các hoạt động song song phân tán
- Hai khái niệm trừu tượng
 - RDDs: Tập dữ liệu phân tán có khả năng tự phục hồi
 - Hoạt động song song

Tập dữ liệu phân tán linh hoạt

Resilient Distributed Dataset (RDD)

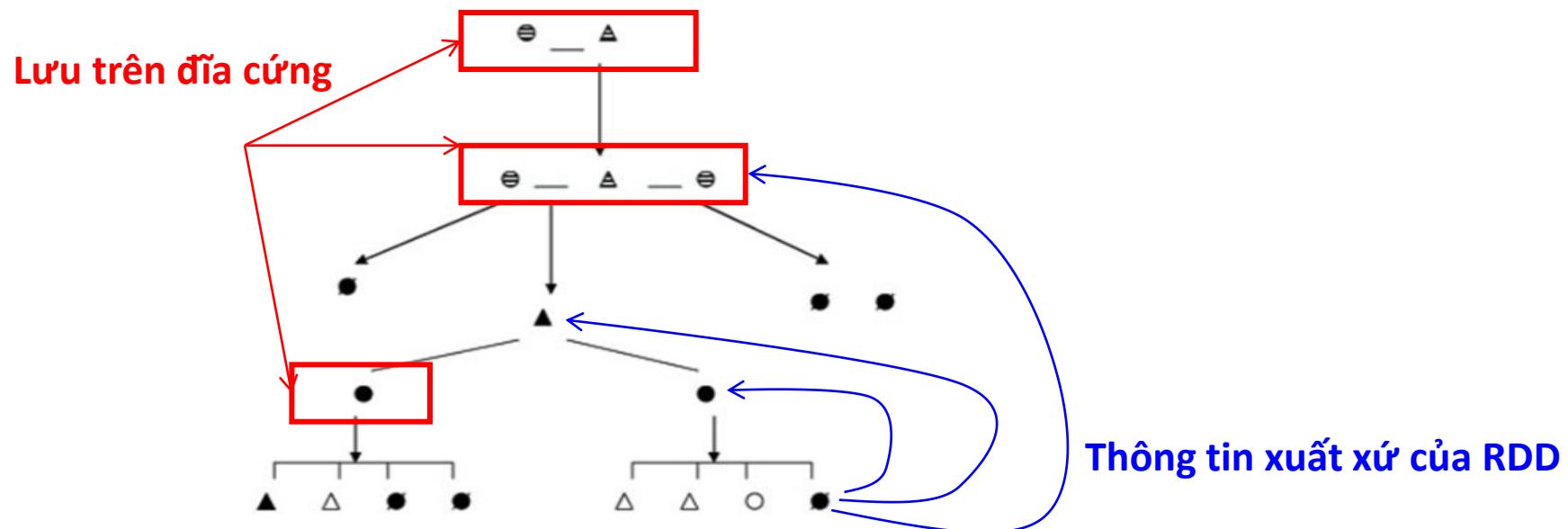
- Cho phép ứng dụng giữ dữ liệu đang vận hành trong bộ nhớ để sử dụng lại một cách hiệu quả
- Giữ lại các ưu điểm của MapReduce
 - Khả năng chịu lỗi, định vị dữ liệu, khả năng mở rộng
- Hỗ trợ một loạt các ứng dụng

- RDD là đối tượng chứa dữ liệu
- Tất cả các thành phần xử lý khác nhau trong Spark đều có chung một lớp trừu tượng gọi là RDD
- RDD được tạo bằng cách song song một bộ sưu tập (mảng, danh sách, từ điển...) hoặc đọc một tập tin.
- Có thể kết hợp các loại biến đổi khác nhau để tạo ra các RDD mới

- RDD là đối tượng chỉ đọc
- Dữ liệu được lưu trữ tạm thời trên bộ nhớ
 - Có nhiều cấp độ lưu trữ khác nhau
 - Dữ liệu sẽ được lưu dự phòng lên đĩa cứng nếu cần thiết
- Có khả năng chịu lỗi tốt

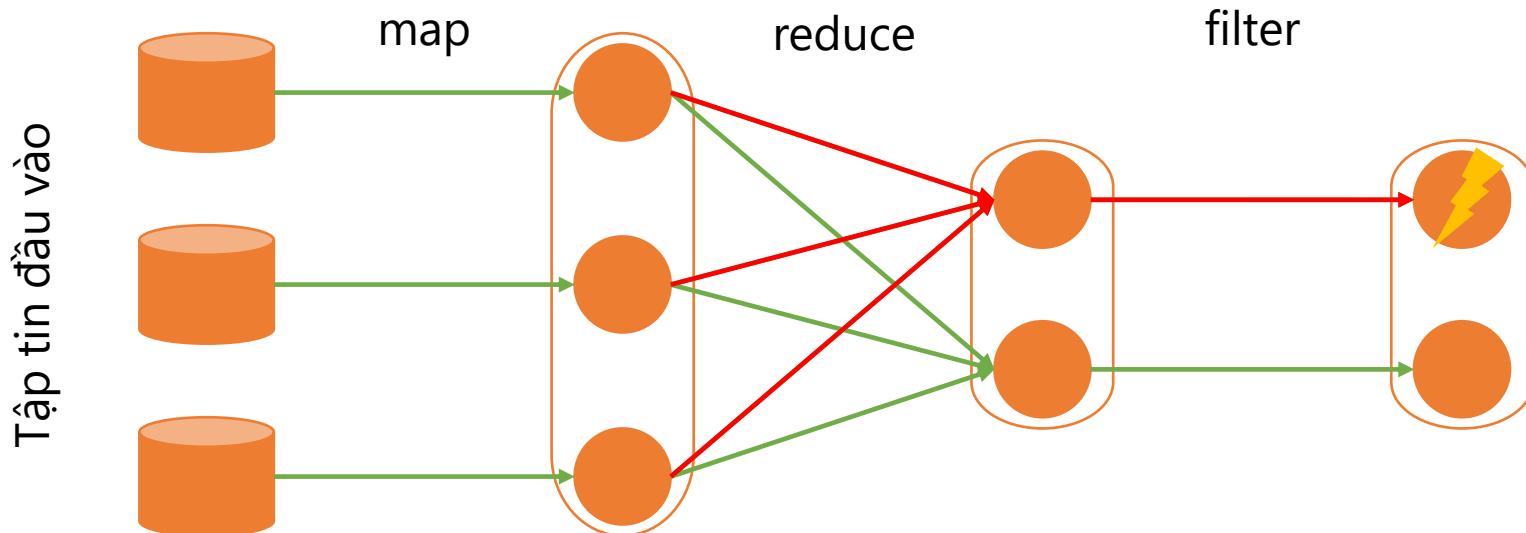
Khả năng chịu lỗi

Không cần phải nhân bản dữ liệu. Nhưng duy trì thông tin xuất xứ về cách tạo lại chúng bắt đầu từ dữ liệu đáng tin cậy



Ví dụ

```
file.map(lambda rec: (rec.type, 1))  
    .reduceByKey(lambda x, y: x + y)  
    .filter(lambda (type, count): count > 10)
```



Tùy chỉnh

- Chiến lược lưu trữ và phân vùng
- Cho biết những RDD nào sẽ được tái sử dụng và chọn chiến lược lưu trữ chúng (ví dụ: lưu trong bộ nhớ/đĩa cứng hay cả hai...)
- Chỉ định các mảnh của RDD trên mỗi máy, điều này rất hữu ích cho việc tối ưu hóa vị trí.

Ưu điểm

- MapReduce tận dụng sức mạnh tính toán của cụm máy tính, nhưng không sử dụng bộ nhớ phân tán
 - Tốn thời gian và chậm
 - RDD cho phép lưu trữ trong bộ nhớ và truyền dữ liệu

So sánh

RDD và Bộ nhớ chung truyền thống

	RDD	Bộ nhớ dùng chung
Đọc	Dữ liệu thô/chi tiết	Dữ liệu chi tiết
Ghi	Dữ liệu thô	Dữ liệu chi tiết
Tính nhất quán	Bất biến	Tùy thuộc vào chương trình/đơn vị thực thi
Khắc phục sự cố	Chi tiết và chi phí thấp nhờ sử dụng thông tin xuất xứ	Phải có điểm đánh dấu và chương trình phục hồi
Giảm thiểu rủi ro	Có thể sao lưu các tác vụ	Khó
Vị trí thao tác	Tự động dựa vào vị trí dữ liệu	Tùy thuộc vào chương trình
Hành vi nếu không đủ RAM	Tương tự như những hệ thống luồng dữ liệu đã có	Hiệu suất kém (cần swap)

Tạo ra RDD

- Đọc lên từ nguồn dữ liệu bên ngoài (tập tin...)
- Tạo ra từ RDD khác (biến đổi)
- Song song hóa một bộ sưu tập tập trung

Đọc lên từ nguồn dữ liệu bên ngoài

- Là phương pháp thông dụng nhất để tạo ra RDD
- Dữ liệu có thể được chứa trong các hệ thống lưu trữ như HDFS, HBase, Cassandra...
- Ví dụ

```
lines = spark.sparkContext.textFile("hdfs://...")
```



Hỗ trợ HDFS, HBase, Amazon S3...

#số phân mảnh = #số lượng block HDFS

Tạo ra từ RDD khác

- Có thể tạo ra RDD mới từ một RDD có sẵn
- RDD gốc vẫn còn nguyên vẹn và không bị sửa đổi
- RDD gốc có thể được sử dụng cho các thao tác sau đó
- Ví dụ

```
error = lines.filter(lambda ln: ln.startswith("ERROR"))
```



RDD mới

Song song hóa một bộ sưu tập tập trung

```
data = [1, 2, 3, 4, 5, 100, 8, 7, ...]
```

```
distData = sc.parallelize(data)
```

```
data = [1, 2, 3, 4, 5, 100, 8, 7, ...]
```

```
distData = sc.parallelize(data, 10)
```

Tạo ra
10 mảng



Cài đặt RDD

- Mục tiêu: hỗ trợ nhiều thao tác/toán tử và cho phép người dùng lập trình chúng tùy ý
- Không để người dùng phải sửa đổi bộ lập lịch cho từng RDD
- *Làm thế nào để ghi nhận xuất xứ của RDD một cách chung chung?*

RDD được tạo thành từ...

- Tập hợp các *phân mảnh dữ liệu* (*partitions*)
- Danh sách các RDD cha mà nó phụ thuộc. (*dependencies*)
- Các hàm để *tính toán* ra các phân mảnh từ RDD cha (*compute*)
- *Vùng lưu trữ ưu tiên* (nếu có) (*preferredLocations*)
- *Thông tin phân mảnh* (nếu có) (*partitioner*)

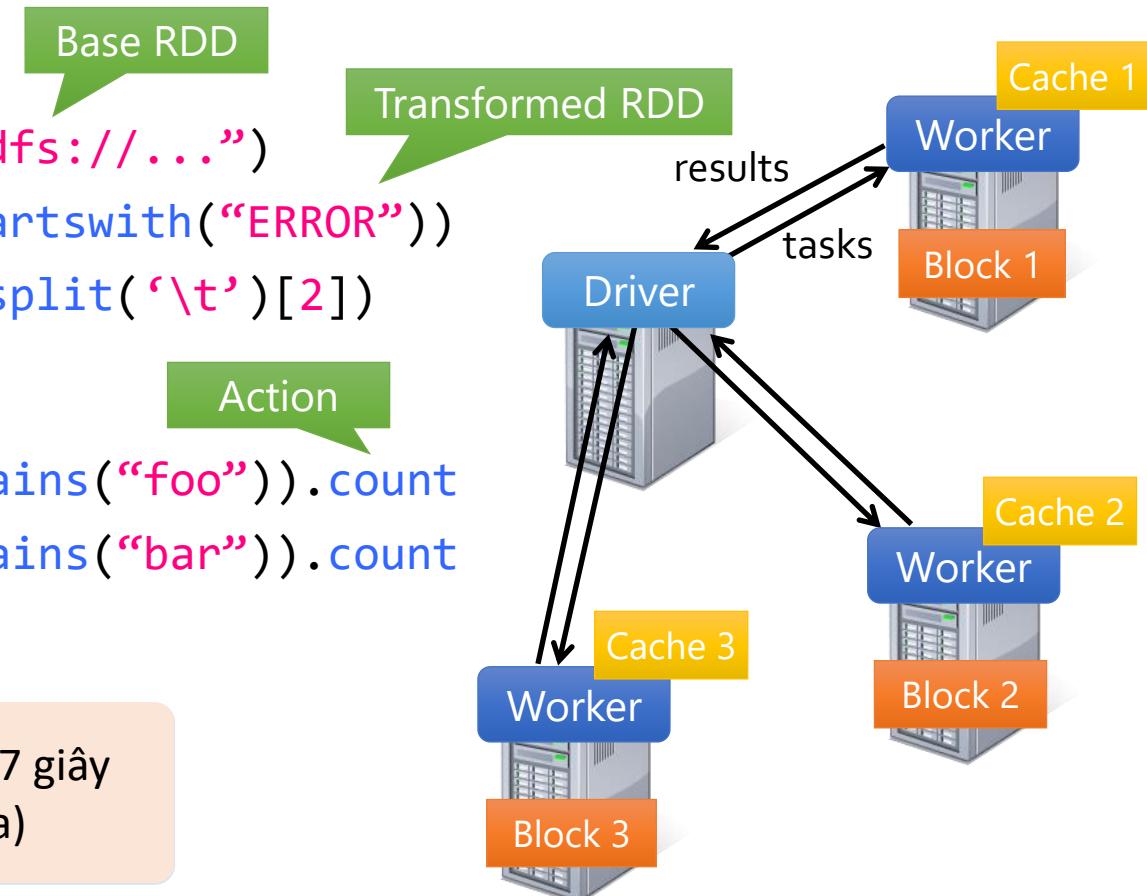
Đóng gói tất cả các thao tác của Spark!

Ví dụ: Khai thác dữ liệu nhật ký

Đưa thông báo lỗi từ nhật ký vào bộ nhớ, sau đó lần lượt tìm kiếm các mẫu khác nhau

```
Base RDD  
lines = spark.sparkContext.textFile("hdfs://...")  
errors = lines.filter(lambda ln: ln.startswith("ERROR"))  
messages = errors.map(lambda err: err.split('\t')[2])  
cachedMsgs = messages.cache()  
  
Action  
cachedMsgs.filter(lambda msg: msg.contains("foo")).count  
cachedMsgs.filter(lambda msg: msg.contains("bar")).count  
. . .
```

Kết quả: xử lý đến 1 TB dữ liệu trong 5-7 giây
(so với 170 giây cho dữ liệu trên đĩa)



Đồ thị RDD

Khung nhìn từ tập dữ liệu

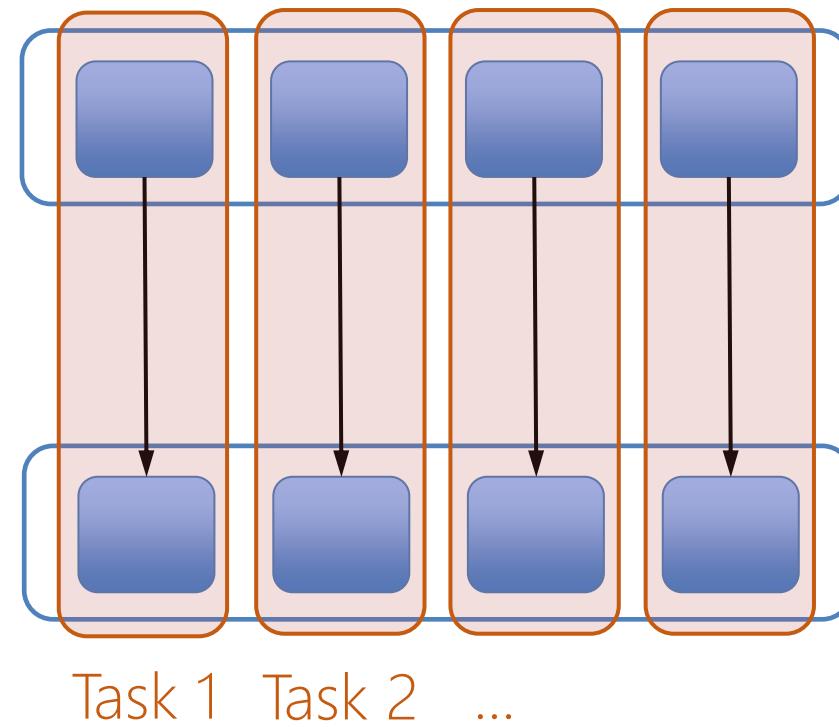
log:

```
HadoopRDD  
path = hdfs://...
```

errors:

```
FilteredRDD  
func = _.startswith(...)  
shouldCache = true
```

Khung nhìn từ mảng dữ liệu



Ví dụ: HadoopRDD

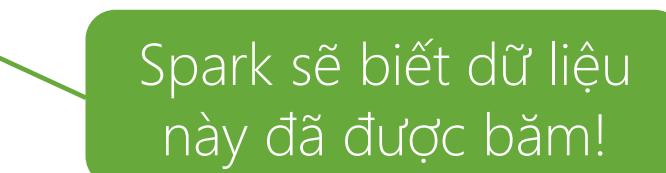
- `partitions` = một mảng/khối HDFS
- `dependencies` = không
- `compute(partition)` = đọc khối tương ứng
- `preferredLocations(part)` = vị trí khối HDFS
- `partitioner` = không

Ví dụ: FilteredRDD

- `partitions` = cùng vị trí với RDD cha
- `dependencies` = “một – một” với RDD cha
- `compute(partition)` = tính ra RDD cha và lọc lại nó
- `preferredLocations(part)` = không (có thể giống RDD cha)
- `partitioner` = không

Ví dụ: JoinedRDD

- `partitions` = một mảng/nhiệm vụ reduce
- `dependencies` = “xáo trộn” mỗi RDD cha
- `compute(partition)` = đọc và kết dữ liệu đã xáo trộn
- `preferredLocations(part)` = không
- `partitioner` = `HashPartitioner(numTasks)`



Spark sẽ biết dữ liệu
này đã được băm!

Xác định vị trí dữ liệu

- Lần chạy đầu tiên: dữ liệu chưa có trên bộ nhớ tạm, sử dụng thuộc tính *preferredLocations* của HadoopRDD
- Lần chạy thứ hai: FilteredRDD đã được đưa lên bộ nhớ tạm, sử dụng vị trí của nó
- Nếu có sự cố với bộ nhớ tạm, quay lại HDFS để khôi phục

Chính sách thay thế RDD

- RDD sử dụng thuật toán **LRU** để thay thế
- Khi một RDD mới được tạo ra
 - Nếu bộ nhớ còn trống → Đưa nó lên bộ nhớ
 - Nếu không còn → loại bỏ một hay nhiều RDD theo cơ chế LRU
- Sử dụng *độ ưu tiên lưu trữ* để ngăn việc loại bỏ các RDD quan trọng

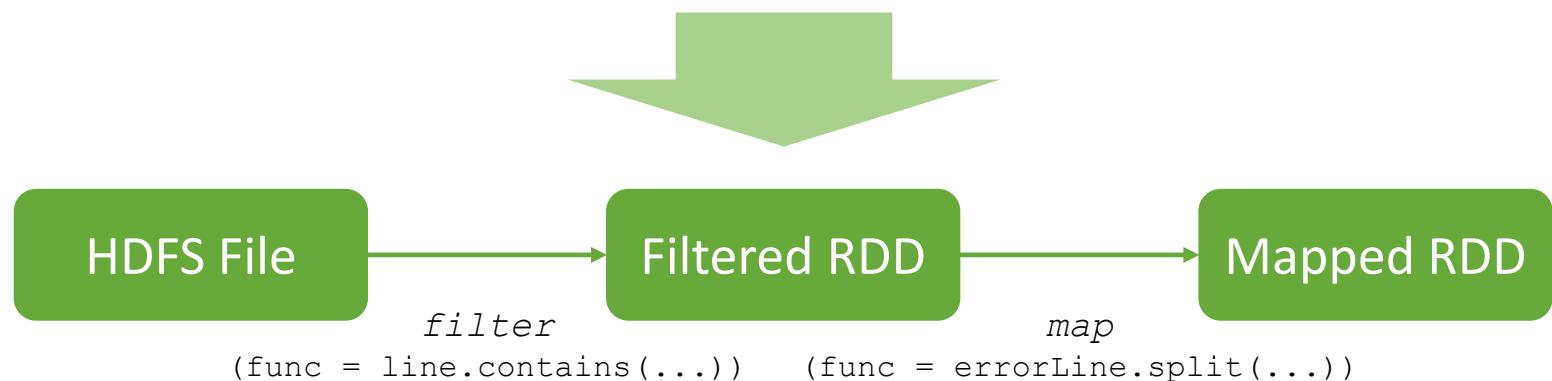
Khôi phục RDD

- Có thể tốn thời gian phục hồi đối với các RDD có chuỗi xuất xứ dài
- Sử dụng *Cơ chế kiểm tra* để kiểm soát tình trạng RDD
 - Người dùng thực hiện, HOẶC
 - Hệ thống kiểm soát, HOẶC
 - Các cơ chế thông minh hơn, ví dụ: xem xét khối lượng công việc

RDD lưu trữ thông tin xuất xứ để có thể sử dụng trong việc khôi phục các phân mảnh dữ liệu bị mất

Ví dụ:

```
messages = spark....textFile(...).filter(lambda ln: ln.startswith("ERROR"))
    .map(lambda err: err.split('\t')[2])
```



Lưu trữ RDD

- Lưu trữ lại một RDD

`RDD.persist()`

- Cấp độ lưu trữ:

MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, DISK_ONLY,.....

- Xóa lưu trữ RDD

`RDD.unpersist()`

APACHE **Spark**

MÔ HÌNH LẬP TRÌNH



Mô hình lập trình

Tập dữ liệu phân tán linh hoạt (RDD)

- Tập hợp các đối tượng được phân tán và bất biến
- Được tạo thông qua các phép *biến đổi (transformation)* song song (`map`, `filter`, `groupBy`, `join...`) trên dữ liệu.
- Có thể lưu vào bộ nhớ đệm để tái sử dụng

Hành động trên RDD

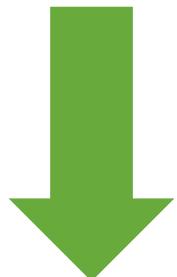
- `count`, `reduce`, `collect`, `save...`

Thao tác trên RDD

Tạo mới RDD

Ghi nhận việc thực thi nhưng
chưa thực hiện thao tác này

BIẾN ĐỔI



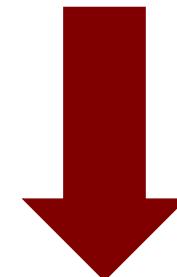
Tương tự như giai đoạn map của Hadoop

Trả về giá trị cho đối tượng gọi

Việc thực thi sẽ kích hoạt các
thao tác này

&

HÀNH ĐỘNG



Tương tự như giai đoạn reduce của Hadoop

Thao tác

Biến đổi

- Vận hành trên một RDD và tạo ra một RDD mới
- Trì hoãn tính toán (lazy evaluation)
- RDD đầu vào được giữ nguyên
- Ví dụ: *map*, *filter*, *join*

Ví dụ 1

```
lines = spark.sparkContext.textFile("hdfs://...")
```

```
error = lines.filter(lambda line: line.startswith("ERROR"))
```

- RDD ban đầu vẫn nguyên vẹn và có thể sử dụng trong những phép biến đổi sau này
- Không có *hành động* nào được gọi nên chỉ có siêu dữ liệu của RDD *error* được tạo ra

Ví dụ 2

```
lines = spark.sparkContext.textFile("data.txt")
lineLengths = lines.map(lambda line: len(line))
```



Spark sẽ lựa chọn lưu RDD trong bộ nhớ hoặc tính toán lại khi cần

```
lineLengths.persist()
```



Yêu cầu Spark lưu lại RDD này trong bộ nhớ

Thao tác

Hành động

- Thực hiện tính toán trên các RDD hiện có để tạo ra kết quả
- Là bước cuối cùng trong luồng thực thi công việc
- Kết quả có thể được
 - Trả về trình điều khiển
 - Lưu trữ trong hệ thống tập tin (như HDFS)
- Ví dụ
 - `count()`
 - `collect()`
 - `reduce()`
 - `save()`

Ví dụ

```
lines = spark.sparkContext.textFile("data.txt")
lineLengths = lines.map(lambda line: len(line))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

Trì hoãn tính toán

- Các thao tác *biến đổi* trên RDD tuân theo quy tắc trì hoãn tính toán
- Kết quả không được tính toán vật lý ngay lập tức
- Siêu dữ liệu về các phép *biến đổi* được ghi lại
- Thao tác *biến đổi* chỉ được thực hiện khi gọi một *hành động*

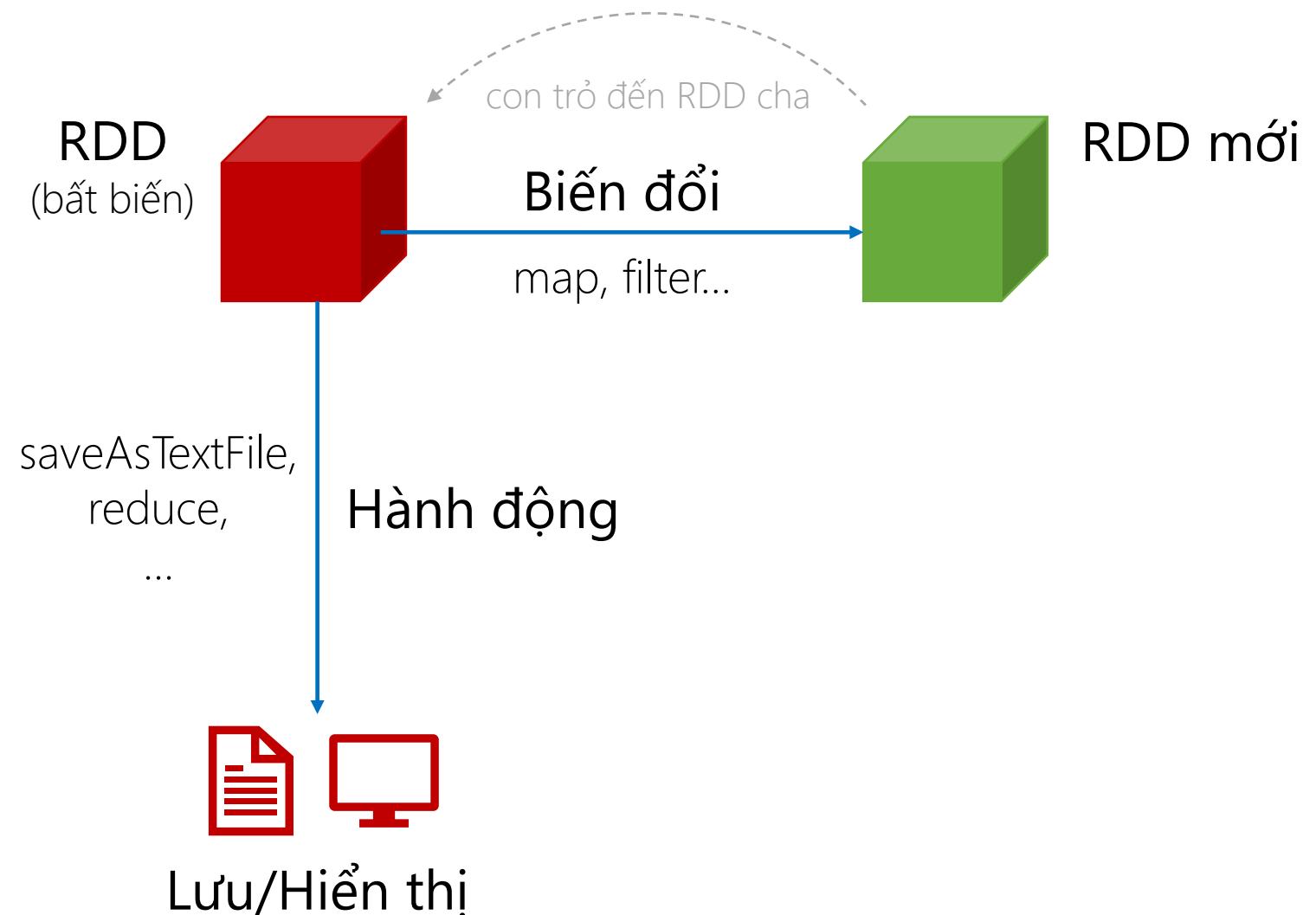
Ví dụ

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda ln: ln.startswith("ERROR"))  
errors.count()
```



Việc thực thi sẽ được gọi ở đây

Biến đổi vs. Hành động



Các loại phụ thuộc

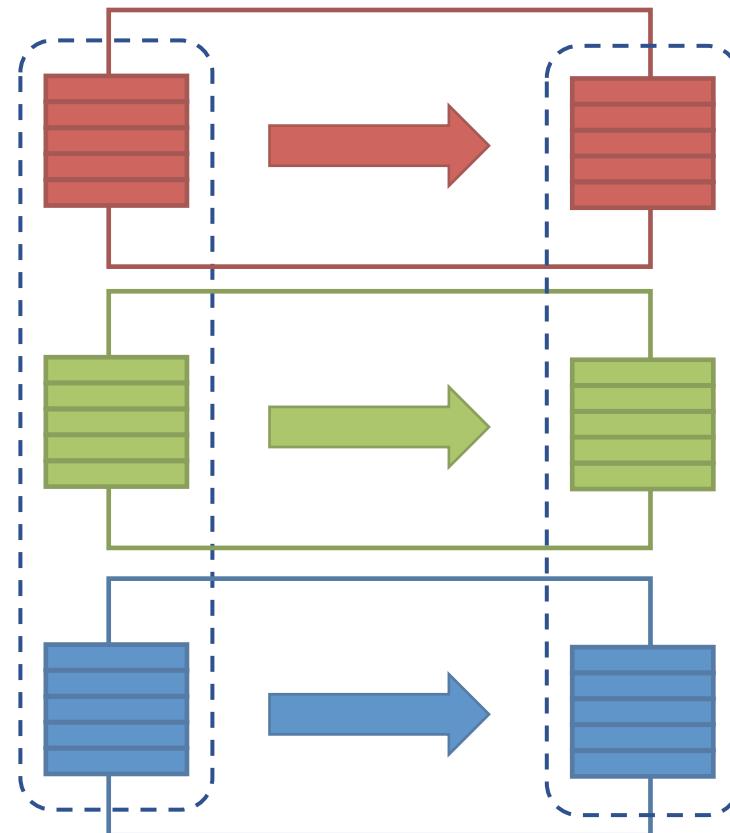
của RDD

Dựa vào thao tác biến đổi, có hai loại phụ thuộc giữa RDD cha và con

- Phụ thuộc hẹp (narrow)
- Phụ thuộc rộng (wide)

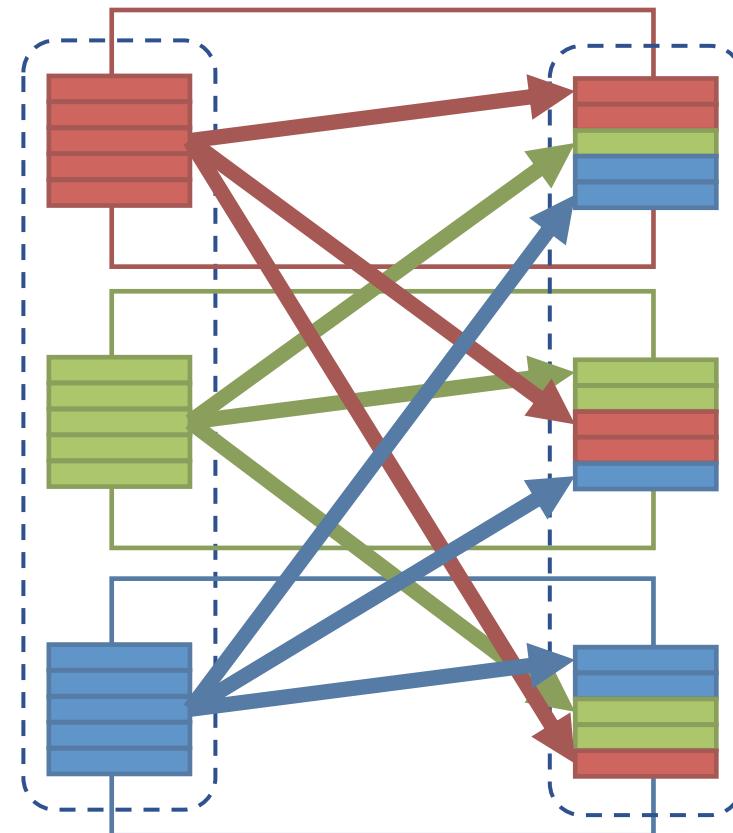
Phụ thuộc hép

- Dựa trên thao tác *Biến đổi hép*
- Đầu vào và đầu ra nằm trên cùng một phân mảnh
- Không cần di chuyển dữ liệu
- Mỗi quan hệ 1-1 giữa các phân mảnh cha – con
- Thao tác ví dụ: *filter* & *map*
- Quy trình tốn tương đối ít tài nguyên

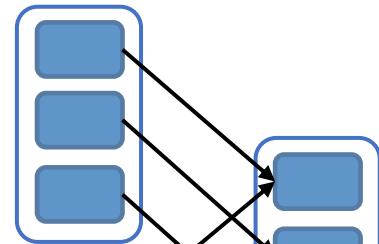
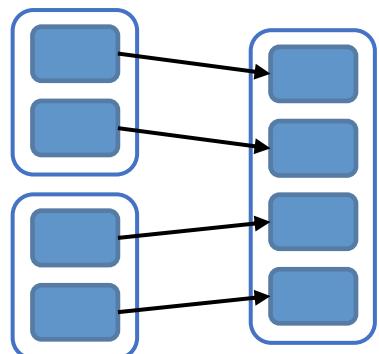
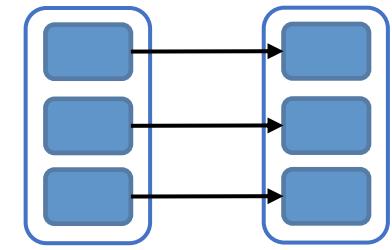


Phụ thuộc rộng

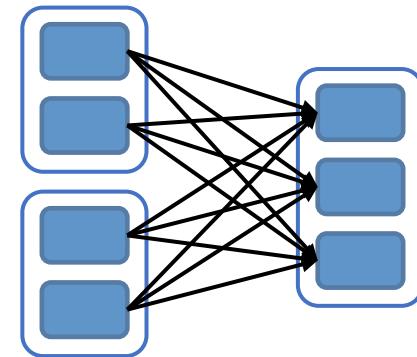
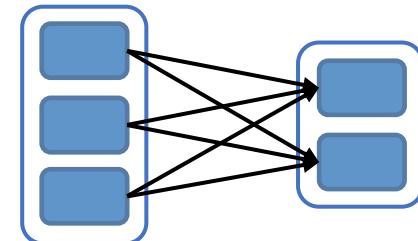
- Dựa trên thao tác *Biến đổi rộng*
- Yêu cầu đầu vào nằm ở những phân mảnh khác
- Cần xáo trộn dữ liệu trước khi xử lý
- Mỗi quan hệ $n-1/n-n$ giữa các phân mảnh cha – con
- Thao tác ví dụ: *join* & *grouping*
- Quy trình tốn nhiều tài nguyên

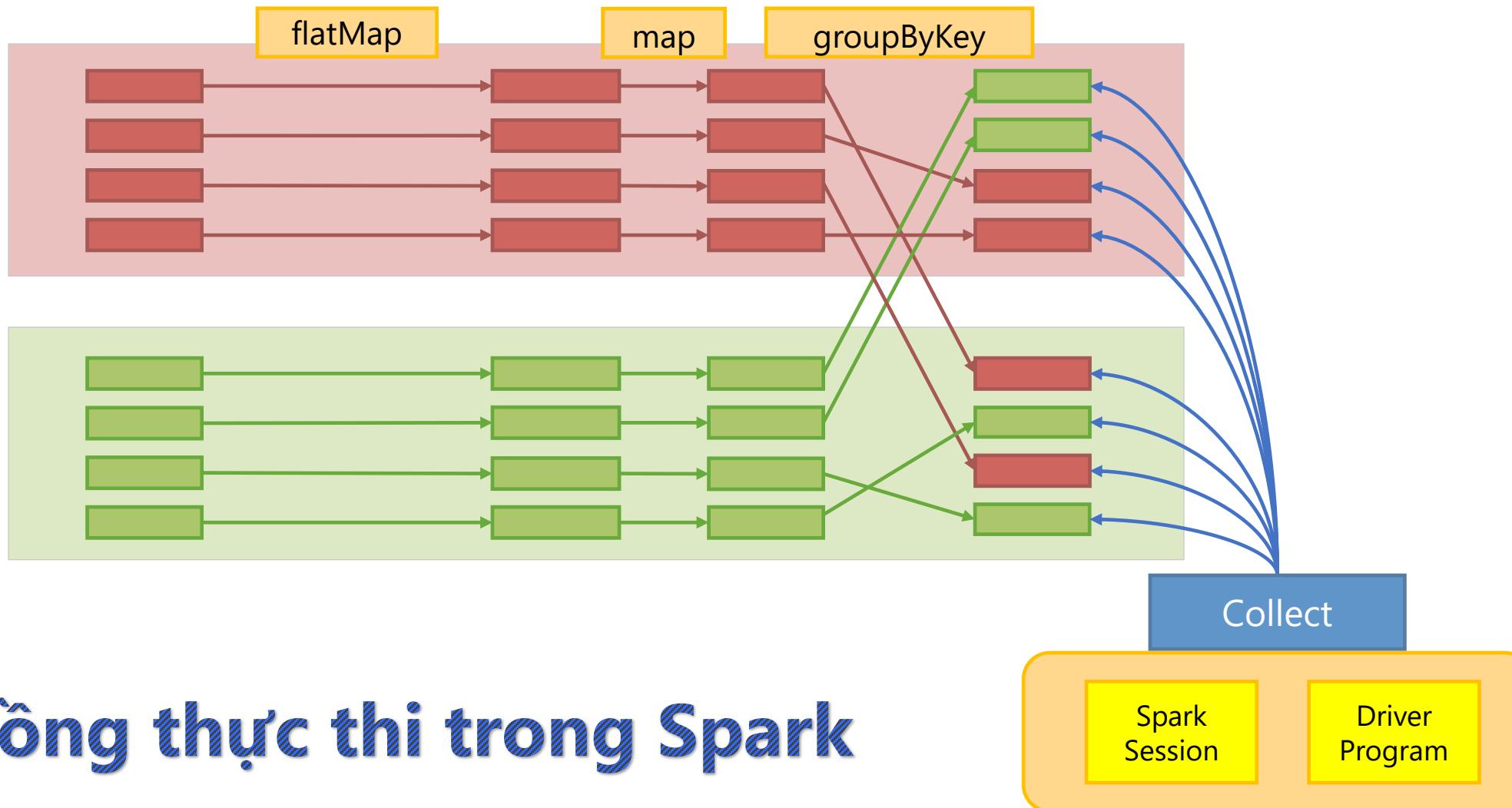


Phụ thuộc hẹp



Phụ thuộc rộng





Luồng thực thi trong Spark

Spark
Session

Driver
Program

Một số hàm thông dụng

Biến đổi (tạo RDD mới)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Hành động (trả về kết quả)		collect reduce count save lookupKey



APACHE **Spark**

MỘT VÀI KHÁI NIỆM

Một số khái niệm

Tên tiếng Anh	Tên tiếng Việt	Định nghĩa
Application	Ứng dụng	Ứng dụng được người dùng xây dựng trên Spark. Bao gồm một trình điều khiển và các quá trình thực thi trên cụm máy.
Driver program	Trình điều khiển	Chương trình chạy hàm <code>main()</code> của ứng dụng và tạo <code>SparkSession</code>
Cluster Manager	Trình quản lý cụm	Một chương trình bên ngoài để quản lý tài nguyên trên cụm (ví dụ: trình quản lý độc lập, Mesos, YARN)
Executor	Đơn vị thực thi	Một quy trình được khởi tạo cho <i>một</i> ứng dụng trên <i>một</i> nút tính toán, chạy các tác vụ và giữ dữ liệu trong bộ nhớ hoặc ổ lưu trữ tương ứng tại nút đó. Mỗi ứng dụng có các đơn vị thực thi riêng.
Task	Tác vụ	Một đơn vị (phần) công việc sẽ được gửi cho <i>một</i> đơn vị thực thi
Job	Công việc	Một chuỗi tính toán song song bao gồm <i>nhiều</i> tác vụ được tạo ra để đáp ứng một hành động trong Spark (ví dụ: <code>save</code> , <code>collect</code>)
Stage	Nhóm tác vụ	Mỗi công việc được chia thành các nhóm tác vụ nhỏ hơn phụ thuộc vào nhau (tương tự như giai đoạn Map và Reduce trong MapReduce)

Cụm Cluster

Là một nhóm các máy ảo Java - JVM (nút) được kết nối với nhau, mỗi máy ảo đều chạy Spark, với vai trò điều khiển (driver) hoặc tính toán (worker).

Trình điều khiển

Driver

- Là một trong những nút thuộc cụm.
- Không chạy tính toán (`filter`, `map`, `reduce...`)
- Đóng vai trò quản lý trong cụm máy Spark.
- Khi gọi hàm `collect()` trên RDD hoặc Dataset, toàn bộ dữ liệu sẽ được gửi đến trình điều khiển. Do đó nên cẩn thận khi gọi hàm này

Công việc *Job*

Là một chuỗi các nhóm tác vụ, được kích hoạt bởi một hành động như `count()`, `foreachRdd()`, `collect()`, `read()`, `write()`...

Xáo trộn

shuffle

- Là một thao tác phân vùng lại dữ liệu trong các cụm.
- **join** và bất kỳ thao tác nào kết thúc bằng **ByKey** sẽ kích hoạt xáo trộn.
- Đây là một thao tác tốn kém vì rất nhiều dữ liệu sẽ được gửi qua mạng.

Nhóm tác vụ

Stage

Là một chuỗi các tác vụ có thể chạy liên tiếp nhau mà không có thao tác xáo trộn (shuffle).

- Ví dụ: sử dụng `read()` để đọc một tập tin từ đĩa, sau đó chạy `map()` và `filter()` đều có thể được thực hiện mà không cần xáo trộn, vì vậy có thể sắp xếp chúng trong một nhóm.

Tác vụ

Task

- Là một thao tác đơn lẻ (`map` hoặc `filter`) được áp dụng cho mỗi phân vùng dữ liệu.
- Mỗi tác vụ được hiện thực thành một luồng xử lý duy nhất trong đơn vị thực thi
- Nếu tập dữ liệu được chia thành 2 phân vùng, thì một thao tác chẳng hạn như `filter()` sẽ kích hoạt 2 tác vụ, mỗi phân vùng có một tác vụ đảm nhận.

Phân vùng dữ liệu

Partition

- Là tập các phân mảnh logic của RDD/Dataset.
- Dữ liệu được chia thành các phân mảnh để mỗi đơn vị thực thi có thể hoạt động trên một phần duy nhất, từ đó hiện thực song song hóa.
- Có thể được xử lý bởi một lõi thực thi (executor core) duy nhất.
- Ví dụ: Nếu có 4 phân vùng dữ liệu và có 4 lõi thực thi, thì có thể xử lý song song từng nhóm tác vụ trên toàn bộ RDD/Dataset trong một lần xử lý.

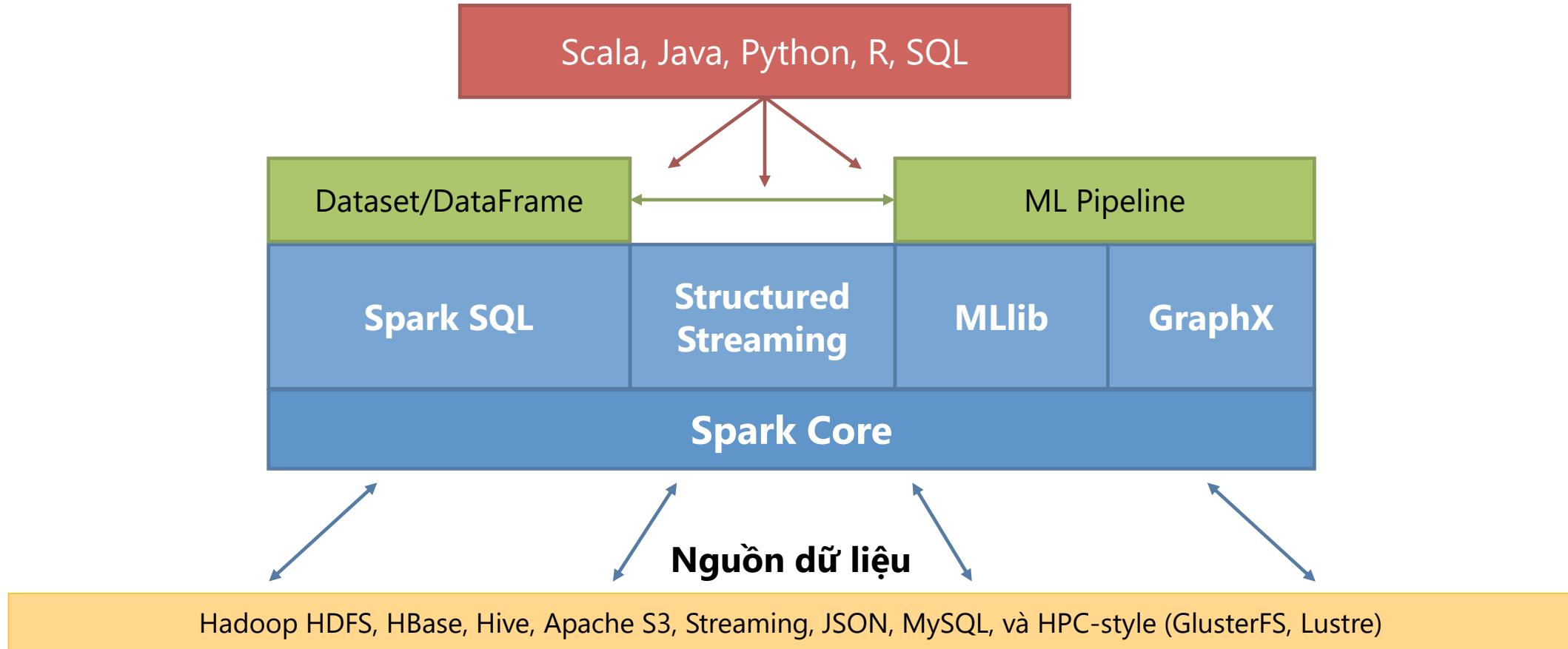


KIẾN TRÚC



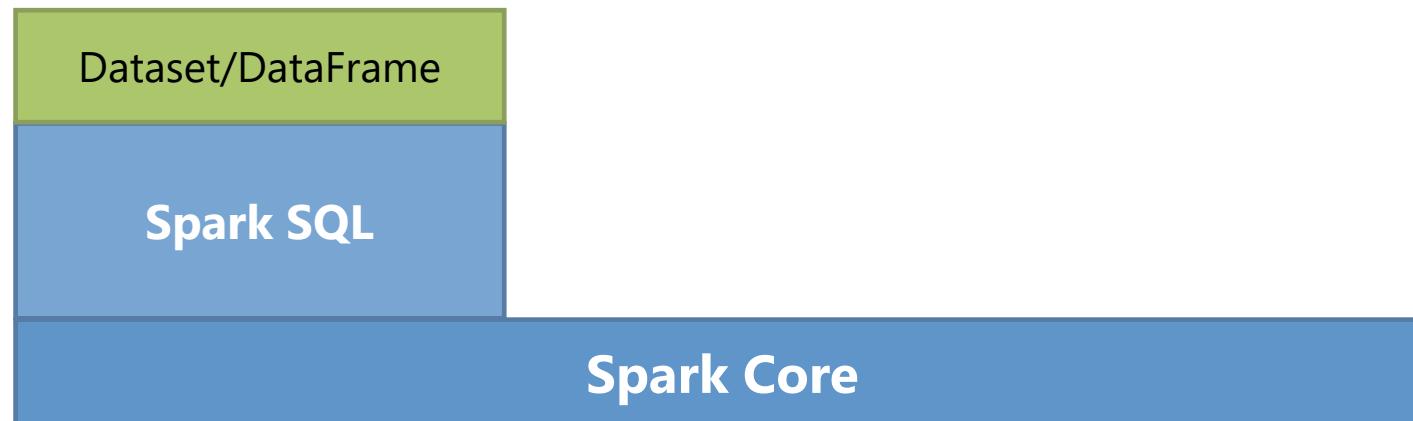
Những thành phần chính

Trong hệ thống Apache Spark



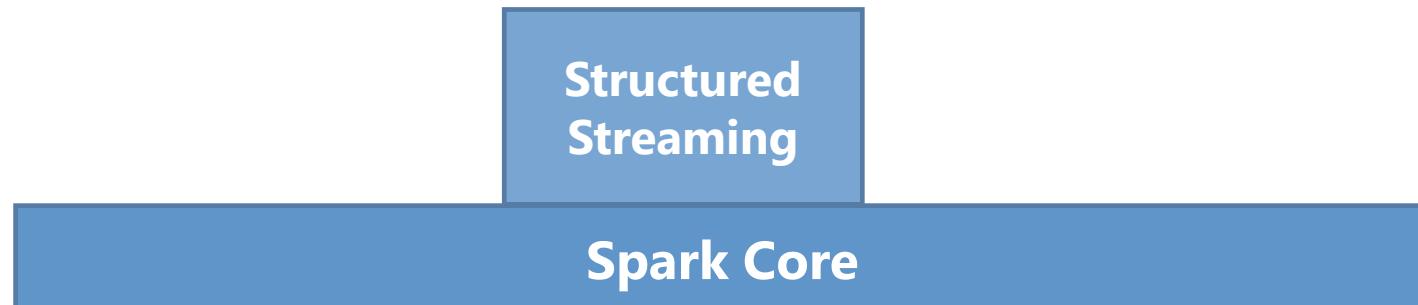
Spark SQL

- Kết hợp tự nhiên các câu truy vấn SQL trong chương trình Spark
- Kết nối với bất kỳ nguồn dữ liệu nào theo cùng một cách.
- Chạy các truy vấn SQL hoặc HiveQL trên các kho dữ liệu hiện có.
- Cung cấp kết nối tiêu chuẩn với các công cụ bên ngoài thông qua ODBC hoặc JDBC



Spark Structured Streaming

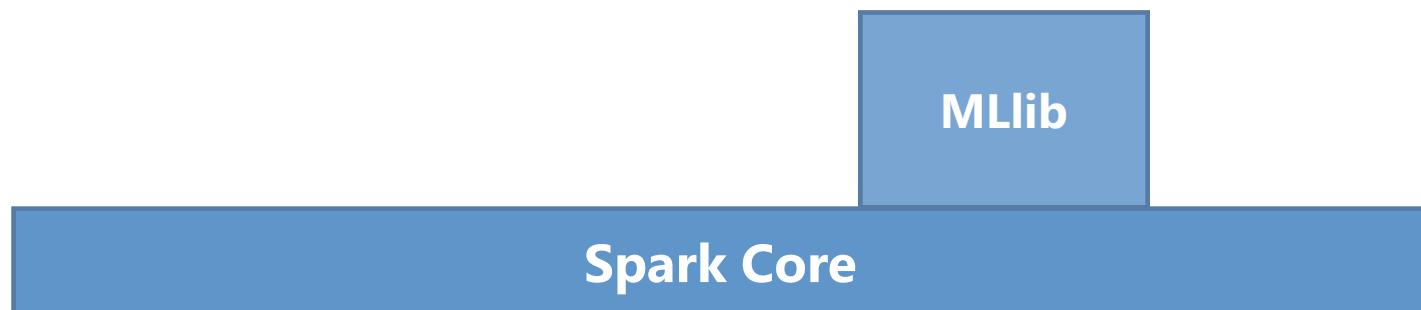
- Là API xử lý dữ liệu luồng trực tuyến được thêm vào từ phiên bản 2.0
- Được xây dựng dựa trên Spark SQL. Tận dụng được sức mạnh của Dataset/DataFrame API và khả năng tối ưu hóa của Spark SQL.
- Kể từ Spark 2.3, bổ sung chế độ xử lý có độ trễ thấp mới được gọi là Xử lý liên tục (Continuous Processing)
- Đảm bảo tốc độ, khả năng mở rộng và chịu lỗi.



MLlib

Là thư viện máy học của Spark. Mục tiêu của nó là làm cho việc áp dụng máy học trong thực tế có thể phổ biến và dễ dàng hơn. MLlib bao gồm các công cụ:

- Các thuật toán phổ biến như phân lớp, hồi quy, gom cụm và lọc cộng tác
- Trích xuất thuộc tính, biến đổi, giảm kích thước và lựa chọn
- Công cụ để xây dựng, đánh giá và điều chỉnh Quy trình học máy (ML Pipeline)
- Lưu trữ/tái lập các thuật toán, mô hình và quy trình
- Các tiện ích: đại số tuyến tính, thống kê, thao tác với dữ liệu...



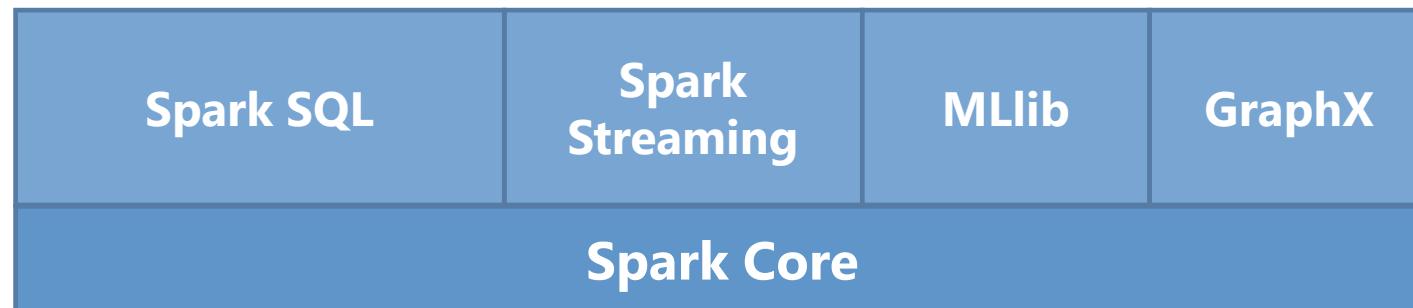
GraphX

- Phân tích khám phá và xử lý đồ thị lặp trong một hệ thống duy nhất.
- Có thể xem cùng một dữ liệu dưới dạng biểu đồ và bộ sưu tập. Biến đổi, kết hợp các biểu đồ với RDD một cách hiệu quả.
- Tùy chỉnh các thuật toán bằng cách sử dụng API Pregel.



Lợi ích của một nền tảng hợp nhất

- Không cần sao chép hoặc ETL dữ liệu giữa các hệ thống
- Có thể kết hợp các kiểu xử lý trong một chương trình
- Dễ dàng tái sử dụng mã nguồn
- Chỉ cần tìm hiểu một hệ thống duy nhất
- Chỉ cần quản lý, bảo trì một hệ thống duy nhất

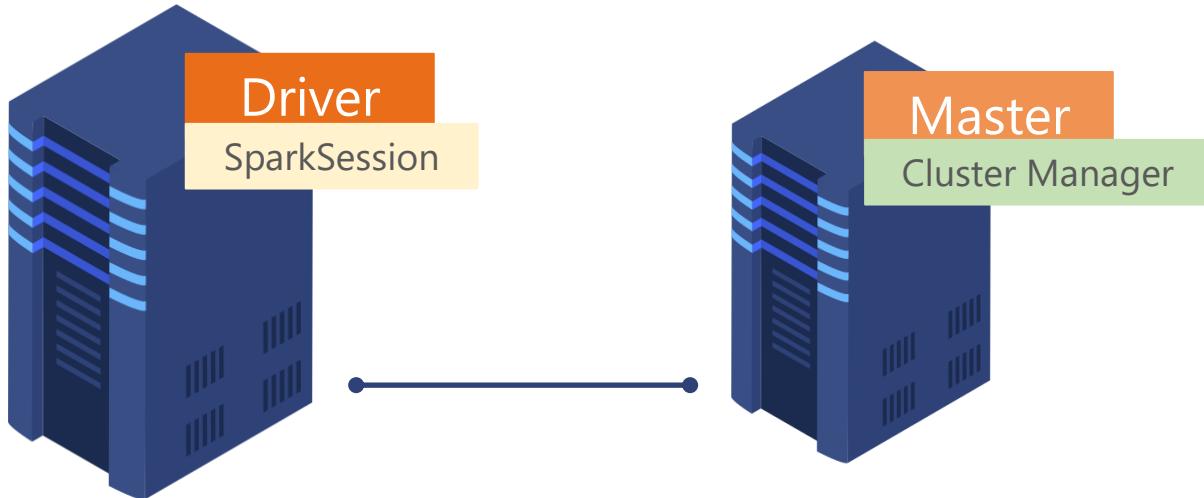


Cụm máy Spark



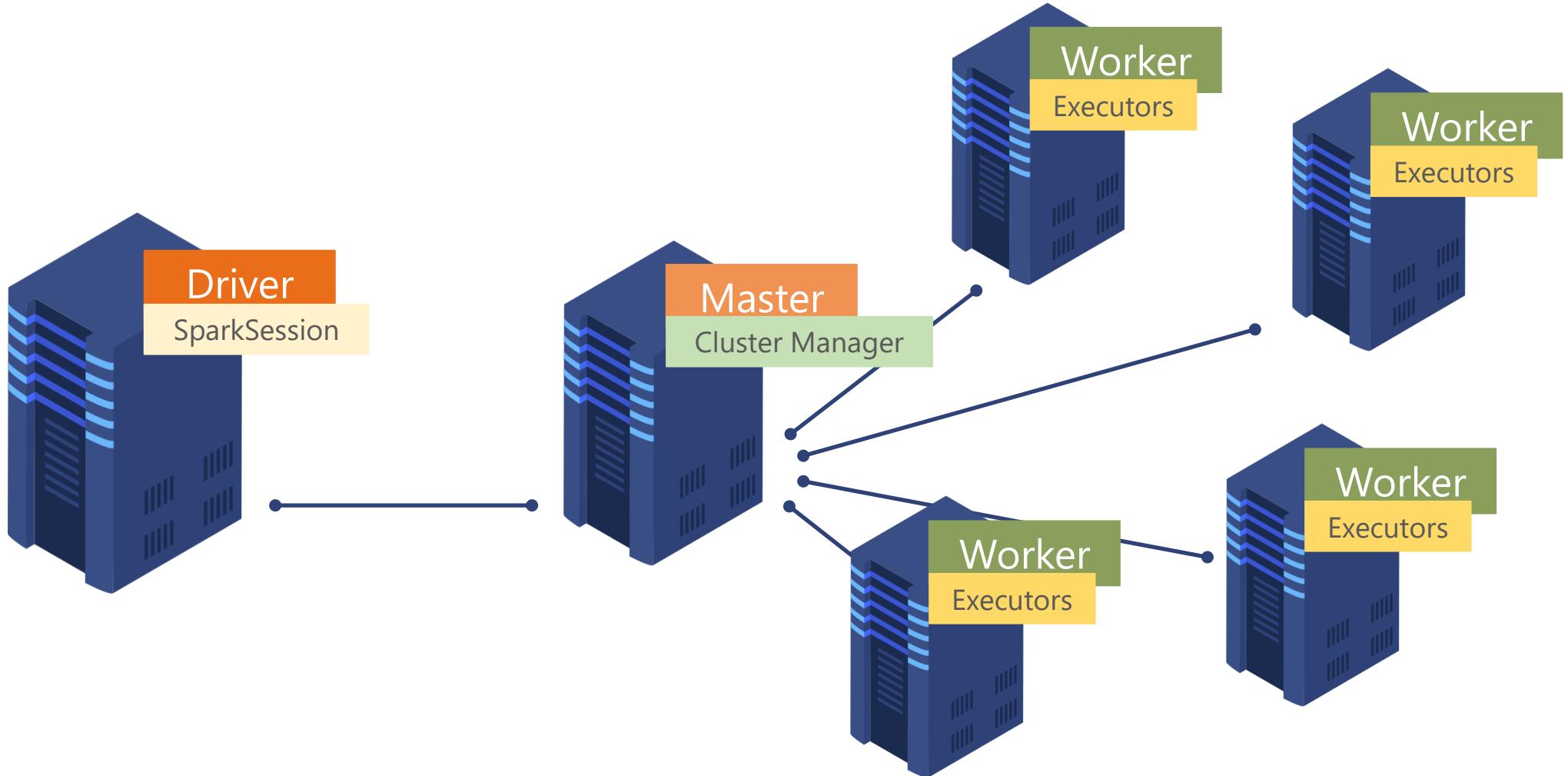
Driver

- Điểm đầu vào của ứng dụng Spark
- Ứng dụng Spark chính được chạy tại đây
- Kết quả của chương trình được tổng hợp tại đây



Master điều khiển phân tán các máy trạm trong Spark, bao gồm:

- Kiểm tra tình trạng của máy trạm
- Giao lại các tác vụ không thành công
- Điểm đầu vào của công việc và cụm máy



Worker sinh ra những đơn vị thực thi (executor) để thực hiện các tác vụ trên những phân vùng dữ liệu

Chi tiết các thành phần

Chương trình

```
spark = SparkSession.builder  
f = spark....textFile("...")  
f.filter(...)  
.count()  
...
```

Spark Master

Đồ thị RDD
Bộ lập lịch
Trình giám sát khối
Trình giám sát xáo trộn

Spark Worker

Trình quản lý cụm
Các luồng xử lý tác vụ
Trình quản lý khối



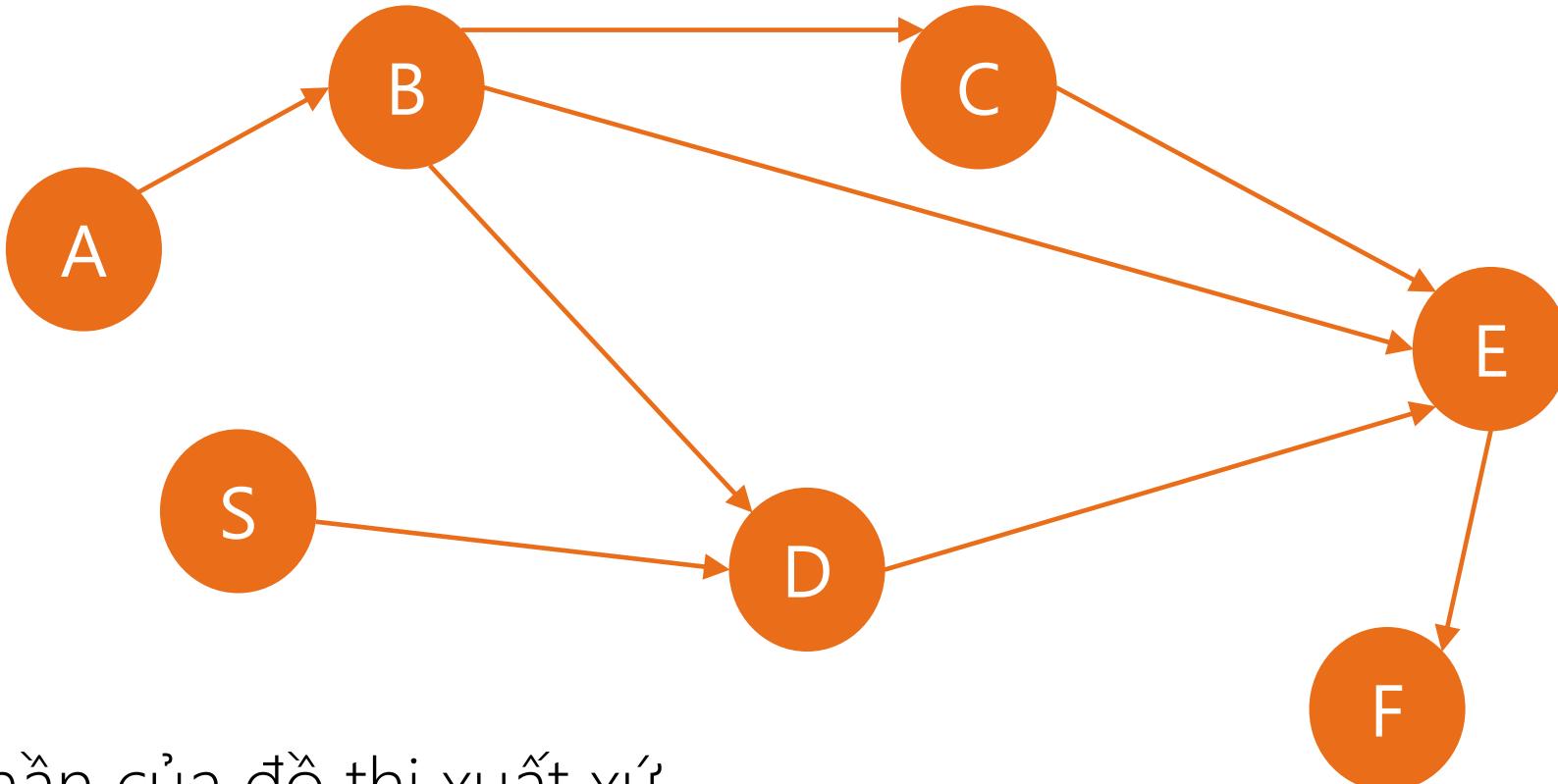
Spark chạy như một thư viện trong ứng dụng của người dùng (một thực thể cho mỗi ứng dụng)

HDFS, HBase, ...

Đồ thị xuất xứ

Lineage graph

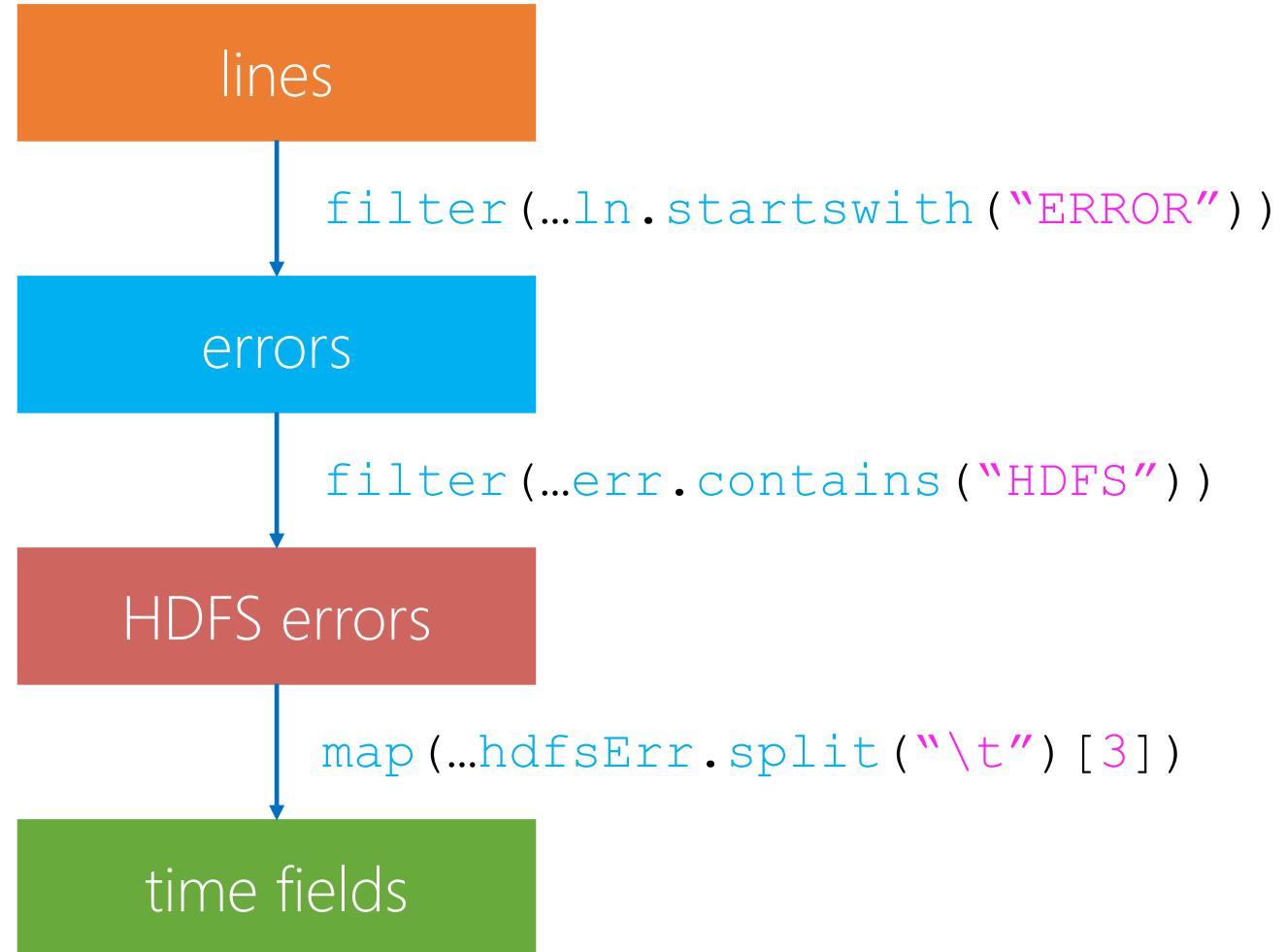
- RDD trong bộ nhớ không được nhân bản
- RAM vẫn bị giới hạn về kích thước (đôi lúc bộ nhớ sẽ trở nên khan hiếm)
- **Đồ thị xuất xứ** là một đồ thị định hướng không tuần hoàn (Directed Acyclic Graph – DAG)
 - Ghi nhận sự phụ thuộc giữa các RDD
 - Giúp khôi phục lại RDD khi sự cố xảy ra



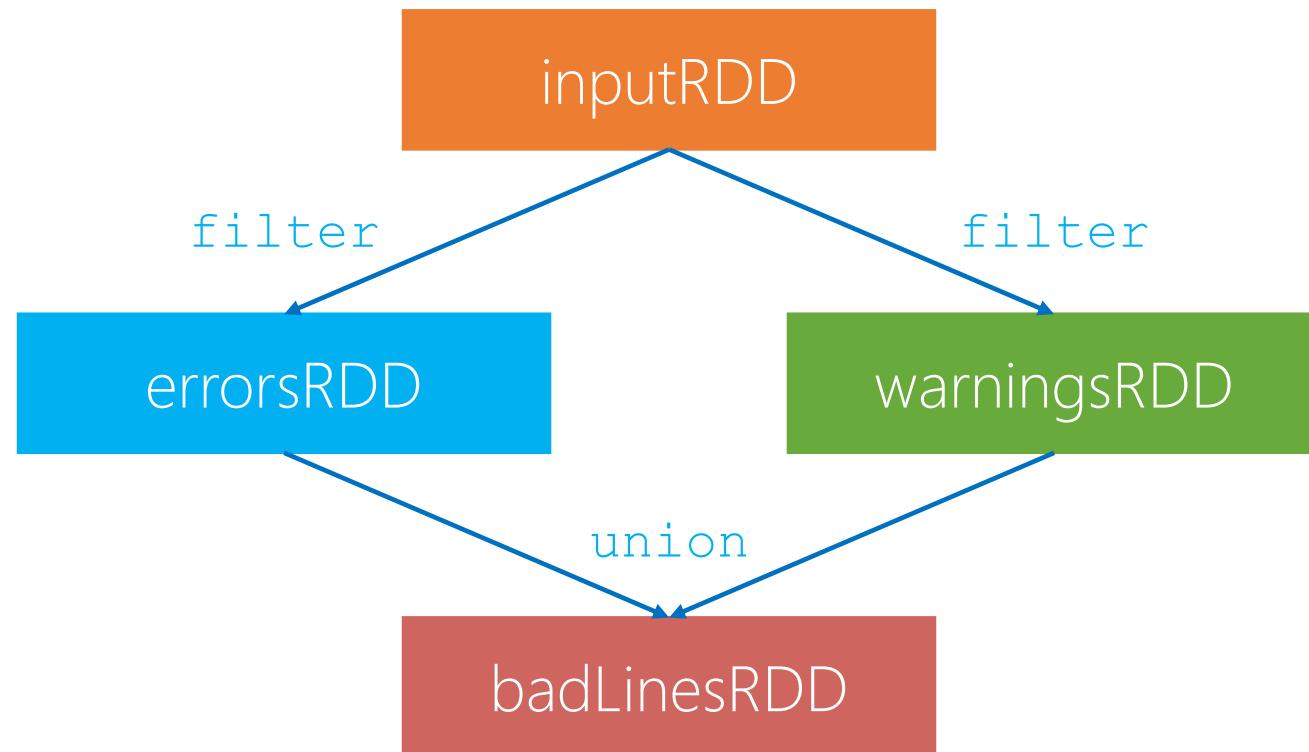
Thành phần của đồ thị xuất xứ

- Các nút là RDD
- Cạnh là các thao tác biến đổi

- DAG không lưu trữ dữ liệu mà thay vào đó là cách các RDD được tạo ra (các bước xử lý)

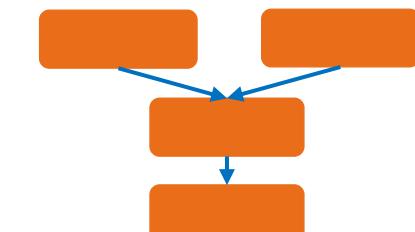


Ví dụ Đồ thị xuất xứ tạo ra trong quá trình phân tích nhật ký



Lập lịch công việc

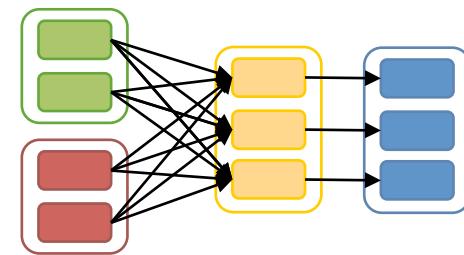
Đối tượng RDD



rdd1.join(rdd2)
.groupBy(...)
.filter(...)

xây dựng DAG

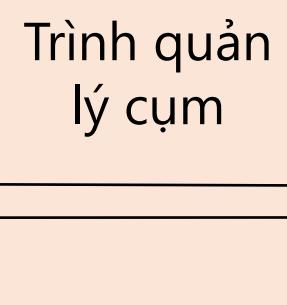
Bộ lập lịch DAG



cắt đồ thị thành
từng *nhóm tác vụ*
trả về kết quả ngay
khi hoàn thành

DAG

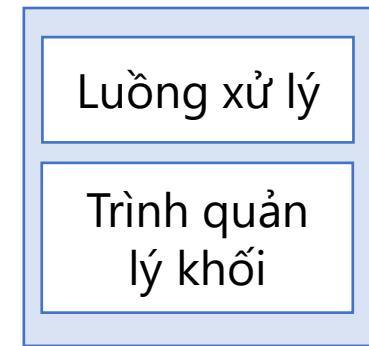
Bộ lập lịch tác vụ



chạy tác vụ thông
qua trình quản lý cụm
thử lại các tác vụ
hỏng hoặc chậm

Tập tác vụ

Máy trạm



thực thi các tác vụ
lưu trữ và quản lý
truy cập các khối

Tác vụ

Bộ lập lịch DAG

DAGScheduler

- Lập lịch hướng nhóm tác vụ (stage-oriented)
- Chuyển đổi kế hoạch thực thi logic (ví dụ: quá trình xây dựng RDD thông qua các phép biến đổi phụ thuộc vào nhau) thành kế hoạch thực thi vật lý (sử dụng các nhóm tác vụ).
- Khi gọi một hành động, SparkContext chuyển một kế hoạch thực thi logic cho Bộ lập lịch DAG để nó chuyển thành một tập hợp các nhóm tác vụ được gửi dưới dạng các Tập tác vụ (TaskSets) để thực thi.

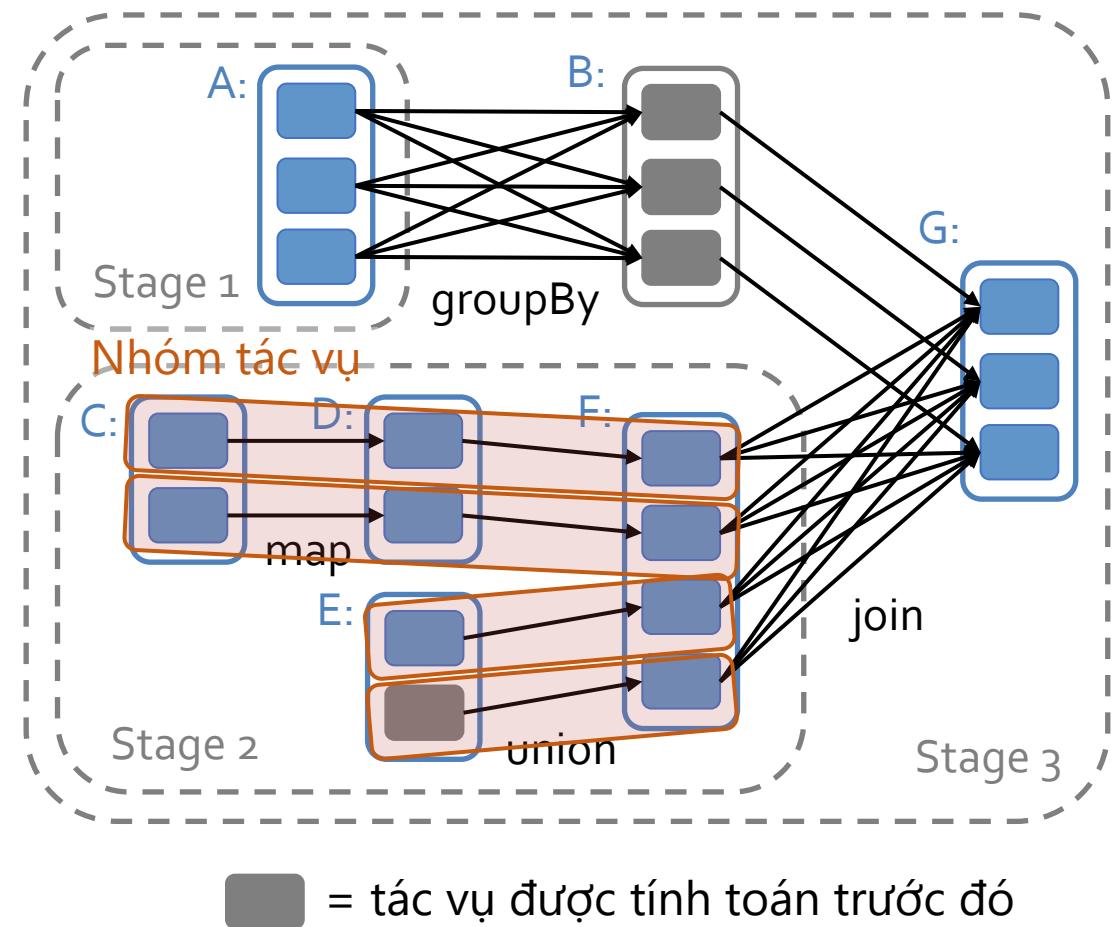
- Bộ lập lịch DAG chỉ hoạt động trên trình điều khiển (Driver)
- Được tạo ra trong quá trình khởi tạo SparkContext (ngay sau khi Bộ lập lịch tác vụ (TaskScheduler) và Bộ hỗ trợ lập lịch (SchedulerBackend) sẵn sàng).

Nhiệm vụ

- Tính toán các DAG thực thi (DAG của các nhóm tác vụ, cho một công việc cụ thể)
- Xác định các vị trí ưu tiên để chạy từng tác vụ.
- Xử lý lỗi (gửi lại các nhóm tác vụ) nếu đầu ra gặp sự cố.

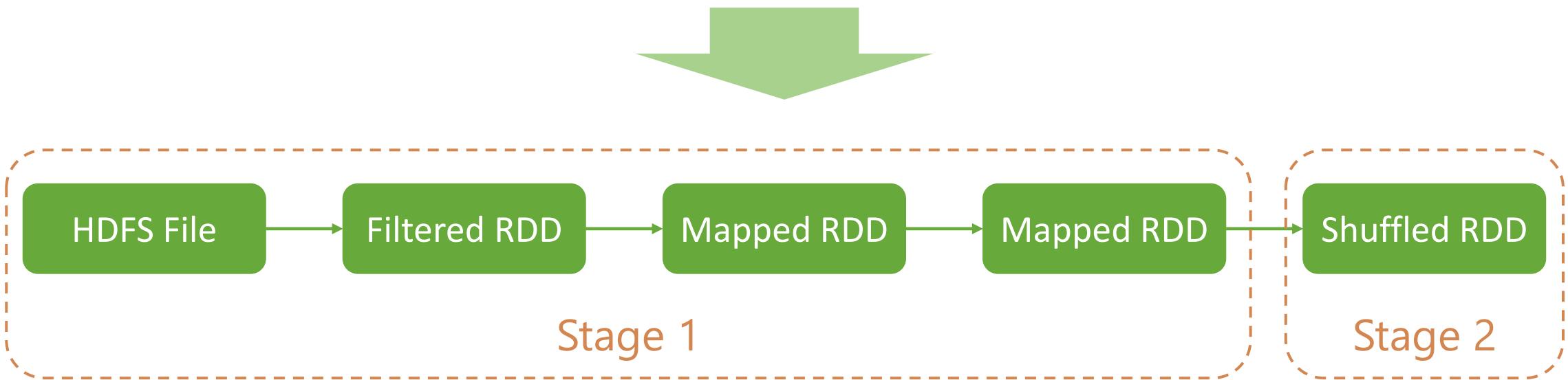
Tính toán các DAG thực thi

- Bộ lập lịch DAG chia đồ thị xuất xứ thành nhiều nhóm tác vụ
- Các nhóm được tạo dựa trên các phép biến đổi (transformation).
- Các phép biến đổi hép sẽ được nhóm lại với nhau thành một chuỗi tác vụ duy nhất.



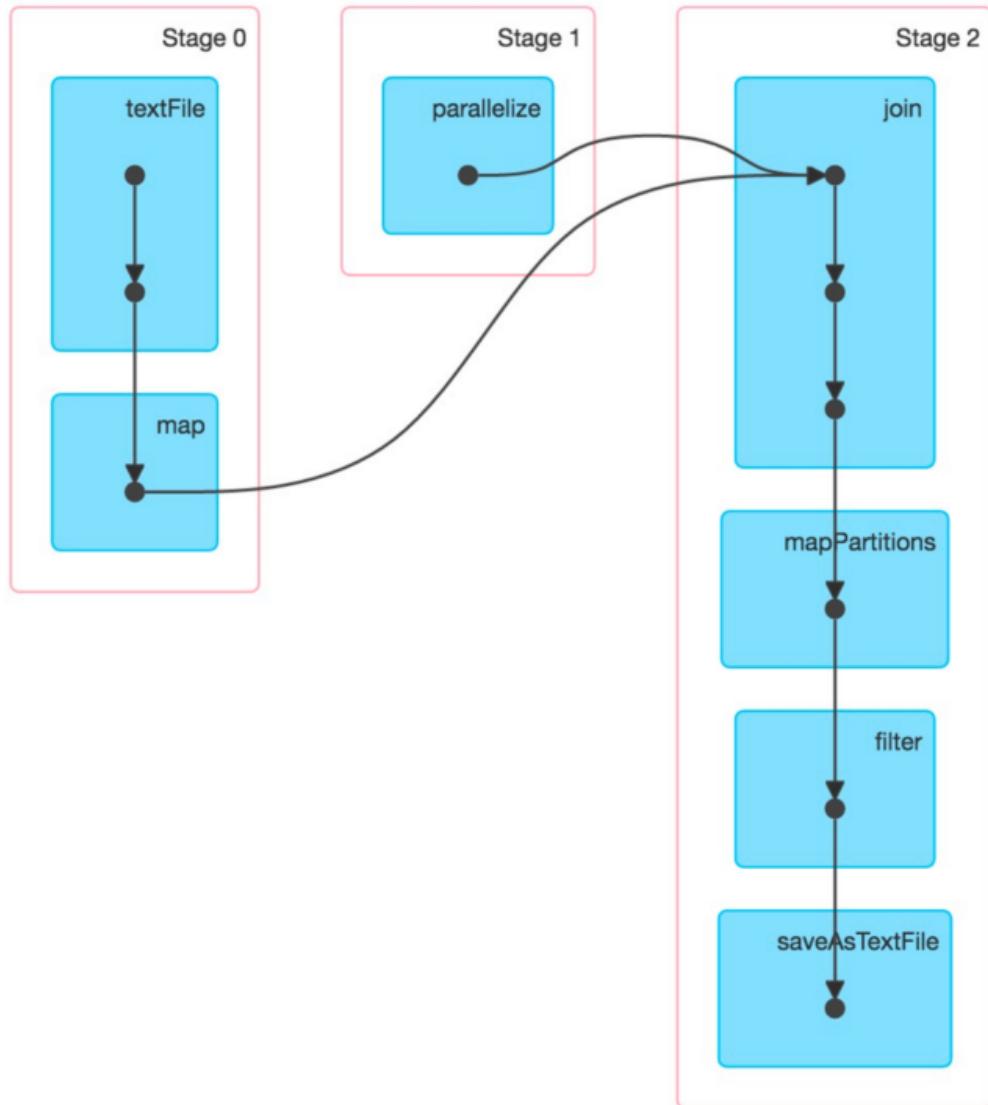
Ví dụ

```
messages = spark...textFile(...).filter(lambda ln: ln.startswith("ERROR"))
    .map(lambda err: err.split('\t')[2])
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
```



- Sau đó, bộ lập lịch DAG sẽ gửi các nhóm tác vụ vào Bộ lập lịch tác vụ.
- Số lượng tác vụ có được phụ thuộc vào số lượng phân đoạn trong [textFile](#).
- Ví dụ: giả sử có 100 phân đoạn trong ví dụ trên, sau đó sẽ có 100 nhóm tác vụ được tạo, gửi đi và thực hiện song song với điều kiện có đủ máy/nhân xử lý.

- Các nhóm tác vụ không phụ thuộc lẫn nhau có thể được gửi cho cụm để thực hiện song song: điều này tối đa hóa khả năng song hành hóa trên các cụm máy phân tán.
- Do đó, nếu các hoạt động trong luồng xử lý có thể diễn ra đồng thời, sẽ nhiều nhóm tác vụ được khởi chạy.



- Nhóm 0 và nhóm 1 có thể thực hiện song song
- Nhóm 2 phụ thuộc vào nhóm 0 và 1 nên phải đợi cả hai hoàn thành mới thực thi được.
- Các tác vụ trong nhóm 2 phải thực hiện tuần tự, vì chúng phụ thuộc lẫn nhau

Xác định vị trí ưu tiên

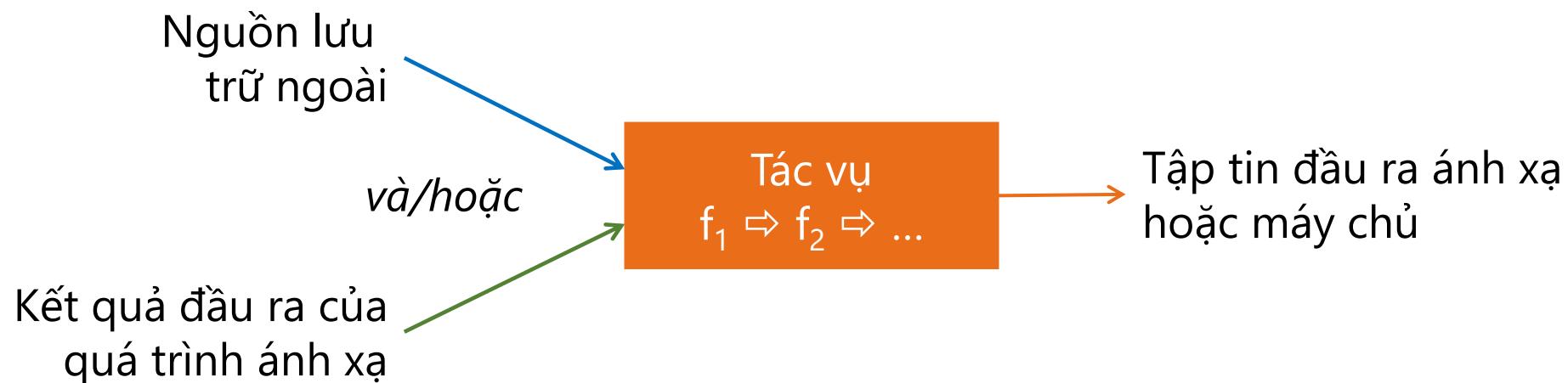
- Bộ lập lịch DAG tính toán vị trí chạy từng tác vụ trong một nhóm dựa trên
 - Vị trí ưu tiên của các RDD cơ bản của nó
 - Vị trí của bộ nhớ tạm thời (cache) hoặc dữ liệu xáo trộn.

Xử lý lỗi

- Một nhóm tác vụ có thể được thực hiện lại nhiều lần để khắc phục lỗi.
- Nếu Bộ lập lịch tác vụ báo cáo rằng một tác vụ không thành công do đầu ra từ nhóm tác vụ trước bị mất, Bộ lập lịch DAG sẽ gửi lại nhóm bị mất.
- Bộ lập lịch DAG sẽ đợi một khoảng thời gian nhỏ để xem liệu các nút hoặc tác vụ khác có bị lỗi hay không, sau đó gửi lại các tập tác vụ cho bất kỳ nhóm nào bị lỗi để tính toán lại các tác vụ thiếu.

Tác vụ

- Ranh giới của nhóm tác vụ chỉ có thể là RDD đầu vào hoặc những thao tác “xáo trộn”
- Do đó, mỗi tác vụ sẽ có dạng như sau:



- Ghi chú: đầu ra xáo trộn sẽ được ghi lên RAM/đĩa để dễ dàng thử lại

- Mỗi tác vụ là một đối tượng khép kín
 - Chứa tất cả mã nguồn của thao tác biến đổi cho đến ranh giới đầu vào (ví dụ: HadoopRDD \Rightarrow filter \Rightarrow map)
 - Cho phép tác vụ thực thi trên dữ liệu trong bộ nhớ tạm ngay cả khi những dữ liệu này lớn đến mức tràn khỏi đó.

Mục tiêu thiết kế: tác vụ có thể chạy trên bất kỳ nút tính toán nào

Cách duy nhất làm hỏng tác vụ là mất các tập tin đầu ra của ánh xạ

Bộ hỗ trợ lập lịch

Task SchedulerBackend

- Là một thành phần quan trọng trong SparkContext
- Hỗ trợ lập lịch tác vụ trong chương trình điều khiển (driver)
- Với mỗi phương pháp triển khai cụm máy Spark sẽ có bộ hỗ trợ lập lịch tương ứng.

Nhiệm vụ

- Theo dõi các đơn vị thực thi (executor) đã đăng ký và tài nguyên sẵn có của nó.
- Thiết lập các kết nối RPC với điểm đầu là trình quản lý tài nguyên (application master) và điểm cuối là các đơn vị thực thi.
- Cung cấp các API cho bộ lập lịch tác vụ.

Bộ lập lịch tác vụ

TaskScheduler

- Là một thành phần quan trọng trong trình điều khiển, cùng với bộ hỗ trợ lập lịch, quản lý tổng thể việc lập lịch tác vụ trong quá trình thực thi ứng dụng Spark.
- Với mỗi nhóm tác vụ sẽ được thực thi trên một cụm, một tập tác vụ (TaskSet) sẽ được gửi đến bộ lập lịch tác vụ để thực thi.
- Bộ hỗ trợ lập lịch sẽ cung cấp các đơn vị thực thi mới hoặc các đơn vị thực thi còn tài nguyên trống.

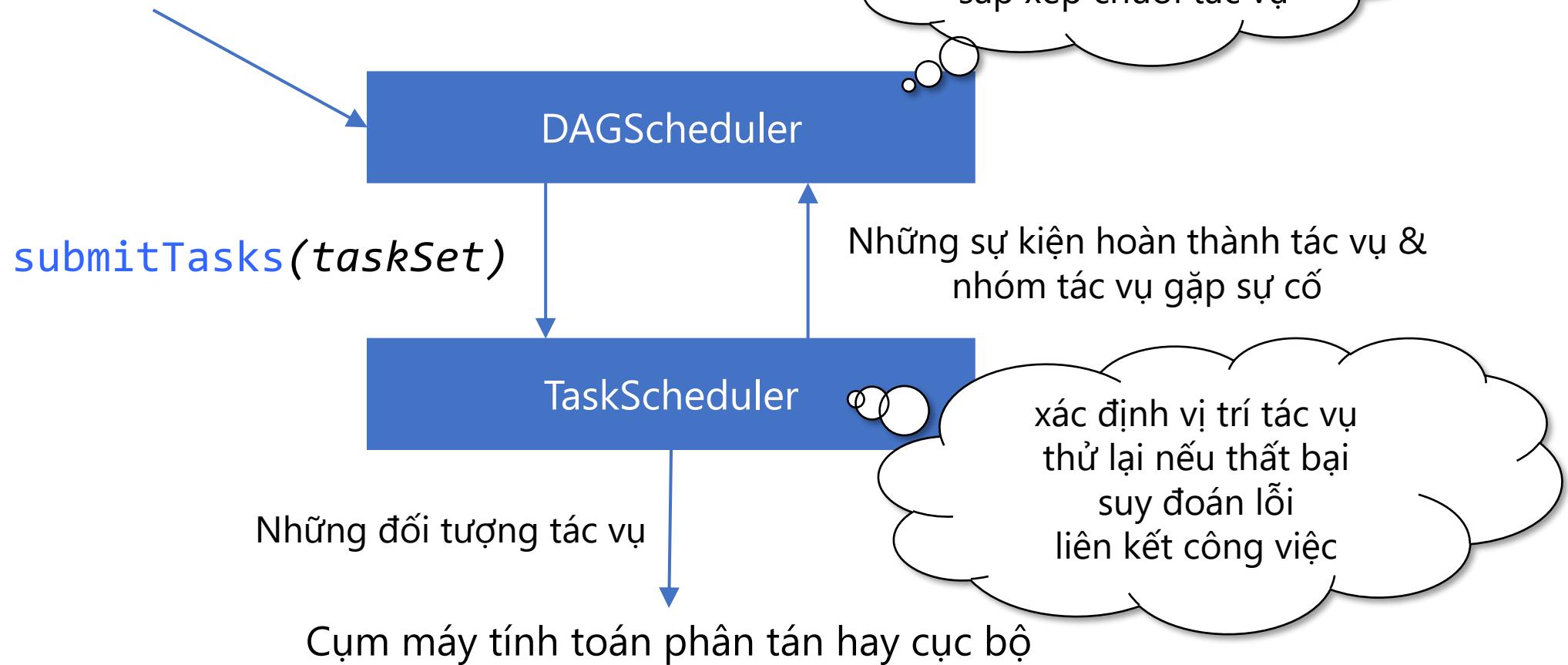
- Thực hiện lập lịch ở hai cấp
 - Đầu tiên ở cấp Tập tác vụ: bộ lập lịch tác vụ chọn một tập tác vụ có mức ưu tiên cao nhất trong số nhiều tập được gửi đến để thực thi.
 - Thứ tự ưu tiên lập lịch được sắp xếp theo thuật toán FIFO hoặc FAIR.
 - Thứ hai ở cấp Tác vụ: một hoặc nhiều tác vụ được chọn để thực thi trong số nhiều tác vụ đang chờ xử lý trong một tập duy nhất đã được chọn ở cấp trên.
 - Dựa trên các ràng buộc cục bộ của từng tác vụ và khả năng tài nguyên tổng thể sẵn có.
- Nếu bộ hỗ trợ lập lịch chưa thông báo hết tài nguyên, tập tác vụ tiếp theo sẽ được lựa chọn dựa trên thứ tự ưu tiên.

- Khi tất cả các tác vụ (trong số một hoặc nhiều tập tác vụ) được xác định thực thi tùy thuộc vào các ràng buộc về khả năng cung cấp tài nguyên (nhờ chức năng của bộ hỗ trợ lập lịch), bộ hỗ trợ lập lịch sẽ gửi các tác vụ này đến các đơn vị thực thi tương ứng để thực hiện.
- Bộ lập lịch tác vụ cũng xử lý những sự kiện hoàn thành do đơn vị thực thi gửi đến thông qua bộ hỗ trợ lập lịch.

- Trong một số tình huống, nếu quá trình thực thi bị lỗi, bộ lập lịch tác vụ sẽ lên kế hoạch thử lại.
- Nếu một tác vụ nào đó được thực hiện quá chậm, bộ lập lịch cũng sẽ có cơ chế suy đoán để thực thi bổ sung tác vụ thay thế.

Luồng sự kiện

`runJob(targetRDD, partitions, func, listener)`



Máy trạm

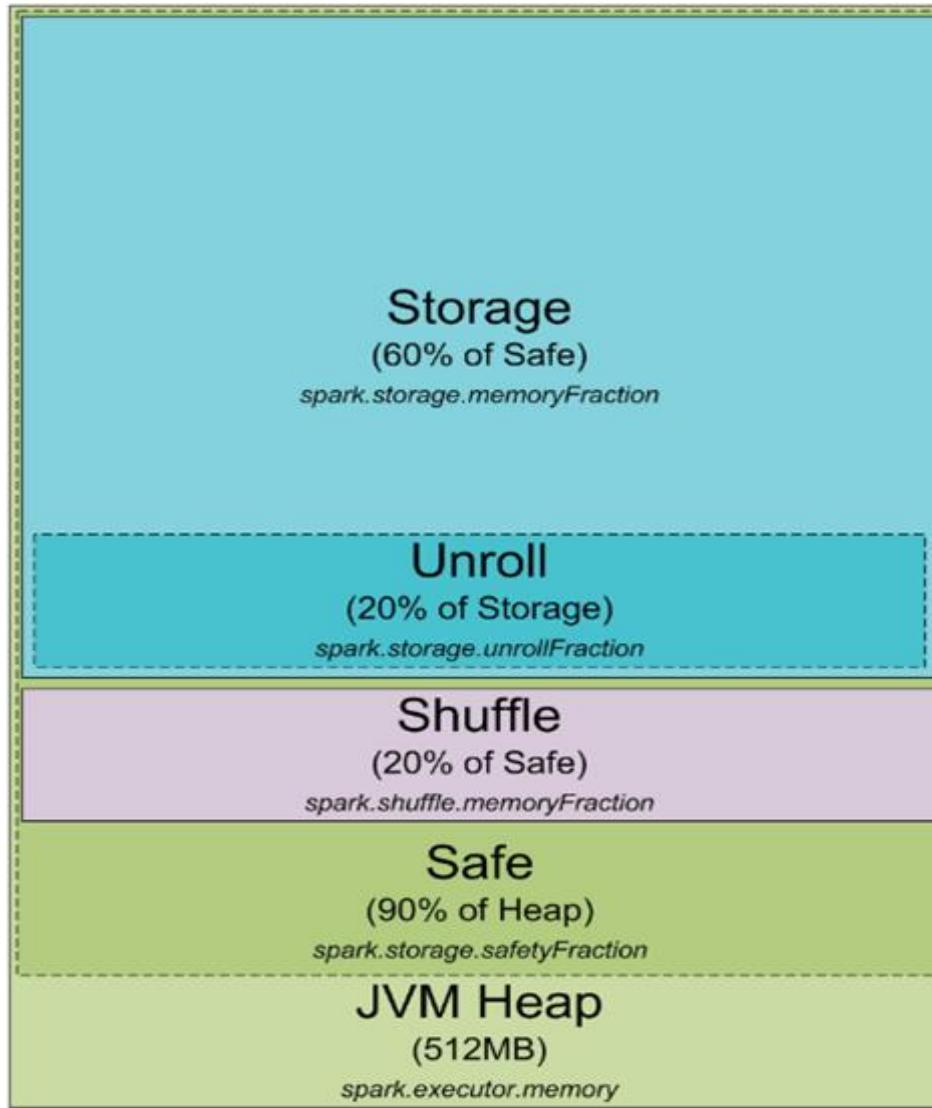
Worker

- Là nút tính toán trong hệ thống Spark
- Mỗi máy chạy một phiên bản Spark, chứa các đơn vị thực thi để thực hiện các tác vụ.
- Nhận vào các tác vụ đã được tuần tự hóa (lập lịch) và chạy chúng trong một chuỗi các luồng xử lý (thread pool)
- Mỗi máy trạm chứa một trình quản lý khối cục bộ của riêng nó để trao đổi các khối và giao tiếp với nhau.

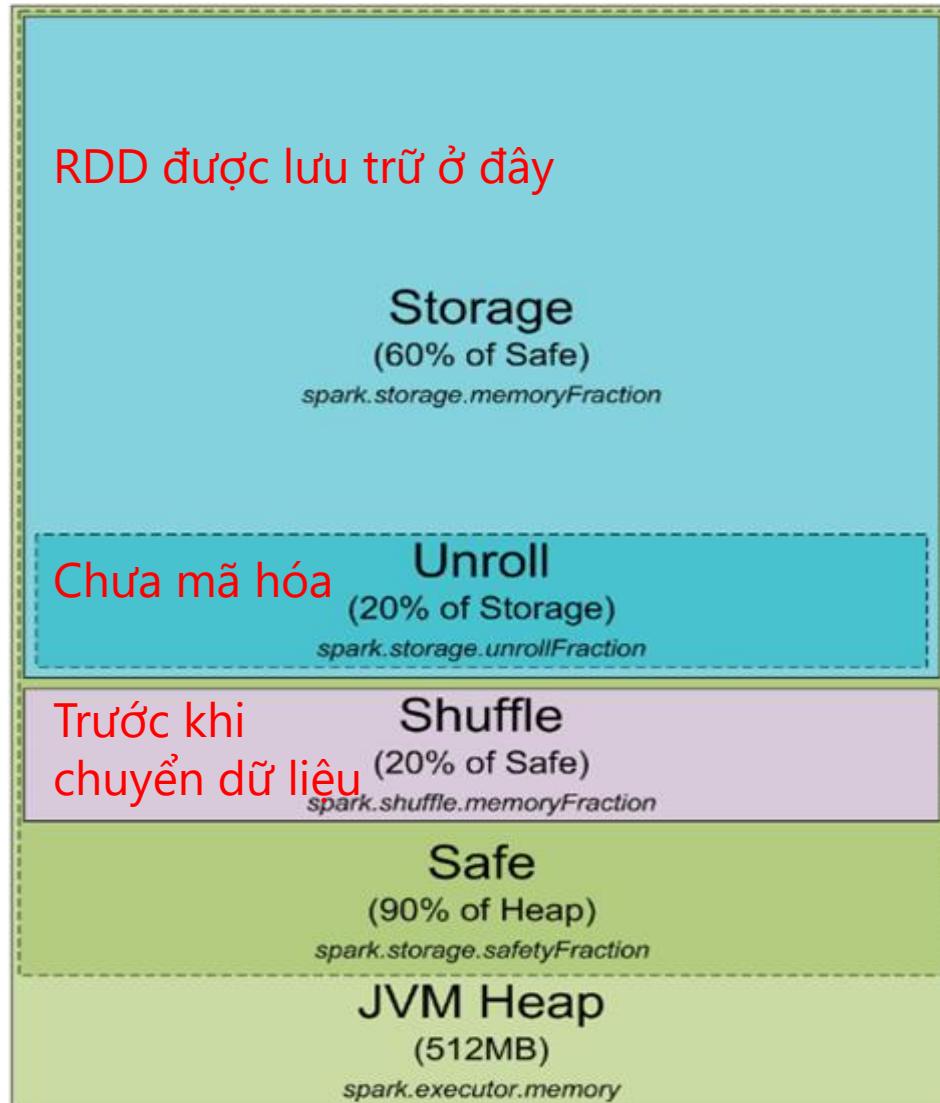
Quản lý bộ nhớ

- Trong Spark, sử dụng bộ nhớ đệm là rất cần thiết
- Có ba lựa chọn lưu trữ RDD:
 - Lưu trữ trong bộ nhớ dưới dạng các đối tượng Java chưa mã hóa (nhanh nhất)
 - Lưu trữ trong bộ nhớ dưới dạng dữ liệu đã mã hóa (hiệu quả về lưu trữ nhưng chậm hơn)
 - Lưu trữ trên đĩa (RDD quá lớn để chứa trong bộ nhớ, chi phí cao nhất)

- Một tiến trình trong Spark
là tiến trình JVM



- Bộ nhớ mặc định là 512MB
- Có các tham số để kiểm soát việc sử dụng các phân đoạn bộ nhớ



Các thành phần khác

Trình quản lý khối (BlockManager)

- Kho lưu trữ cặp khóa-giá trị “chỉ ghi một lần” trên mỗi nút tính toán
- Cung cấp dữ liệu xáo trộn giống các RDD được lưu trong bộ nhớ tạm
- Theo dõi mức lưu trữ (StorageLevel) của mỗi khối (ví dụ: trên đĩa hay RAM)
- Có thể chuyển dữ liệu vào đĩa nếu sắp hết RAM
- Có thể sao chép dữ liệu giữa các nút

Trình quản lý giao tiếp (CommunicationManager)

- Thư viện trong mạng dựa trên IO không đồng bộ
- Cho phép tìm nạp các khối từ các Trình quản lý khối
- Cho phép ưu tiên/phân nhóm trên các kết nối
- Cố gắng tối ưu hóa kích thước khối

Trình quản lý đầu ra của thao tác ánh xạ (MapOutputTracker)

- Theo dõi vị trí từng tác vụ ánh xạ trong một lần xáo trộn
- Cho tác vụ reduce biết vị trí các tác vụ ánh xạ
- Đánh dấu vị trí bộ nhớ tạm của mỗi nút tính toán để tránh tìm lại
- Tạo ra một *định danh đặc biệt* chuyển tới mỗi tác vụ cho phép vô hiệu hóa bộ nhớ tạm khi đầu ra của thao tác ánh xạ gấp sự cố.

APACHE **spark** THỰC HÀNH



Spark Python

- Cách đơn giản nhất: trình thông dịch Spark (**spark-shell** hoặc **pyspark**)
 - Trình thông dịch hỗ trợ ngôn ngữ Scala và Python
- Mặc định chạy 1 luồng ở chế độ cục bộ, có thể tùy chỉnh trong biến môi trường **MASTER**

```
MASTER=local      ./spark-shell          # cục bộ, 1 luồng
MASTER=local[2]   ./spark-shell          # cục bộ, 2 luồng
MASTER=spark://host:port ./spark-shell  # cụm máy Spark độc lập
```

SparkContext

- Xuất hiện từ phiên bản 1.x
- Được định nghĩa trong gói **org.apache.spark**
- Là điểm xuất phát của tất cả các chức năng trong Spark
- Sử dụng để tạo ra RDD, biến lan truyền và biến tích lũy
- Trong môi trường dòng lệnh, được tạo sẵn trong biến **sc**
- Có thể tạo tự động, sử dụng lớp **SparkContext**

Chức năng

- Lấy trạng thái hiện tại của ứng dụng Spark
- Thiết lập tùy chỉnh
- Kết nối đến các dịch vụ khác nhau
- Hủy một công việc
- Hủy một nhóm tác vụ
- Dọn dẹp bộ nhớ khi kết thúc
- Đăng ký các Spark-Listener
- Điều chỉnh việc cấp phát động các luồng xử lý
- Truy xuất đến các RDD được lưu trữ

Nhược điểm

- Cần phải tạo riêng từng loại ngữ cảnh cho những loại thao tác (API) khác nhau (mỗi ngữ cảnh phải có một **SparkConf**)
- DataSet và DataFrame trở thành những API độc lập nhưng chưa có điểm khởi đầu.
- Trước Spark 2.0, không thể tạo ra nhiều môi trường cùng chia sẻ một ngữ cảnh Spark. Chỉ có thể tạo ra nhiều ngữ cảnh, mỗi ngữ cảnh nằm trên một máy ảo JVM riêng biệt.

- Có thể tạo ra nhiều ngữ cảnh trên một máy ảo bằng cách điều chỉnh biến môi trường **spark.driver.allowMultipleContexts**
- Tuy nhiên đây không phải là cách hay
 - Không đảm bảo các ngữ cảnh sẽ hoạt động chính xác
 - Driver sẽ gặp áp lực, khó quản lý các ngữ cảnh
 - Nếu một ngữ cảnh bị lỗi sẽ dẫn đến JVM bị lỗi, tất cả các ngữ cảnh trên cùng JVM cũng sẽ bị hỏng theo
- Cách này đã được gỡ khỏi phiên bản Spark hiện tại

SparkSession

- Được giới thiệu từ phiên bản 2.x
- Định nghĩa trong gói **org.apache.spark.sql**
- Có thể tạo ra RDD, DataFrame, DataSet tự động
- Trong môi trường dòng lệnh, được tạo sẵn trong biến **spark**
- Có thể tạo tự động, sử dụng hàm **SparkSession.builder()**
- Tạo session mới từ một session cũ **spark.newSession()**

- SparkSession là lớp hợp nhất các ngữ cảnh của Spark
 - Bên trong nó có chứa các ngữ cảnh:
 - SparkContext
 - SQLContext
 - StreamingContext
 - HiveContext
- ⇒ Do đó SparkSession có thể thực hiện tất cả các chức năng của những ngữ cảnh thành phần

- Có thể tạo ra một môi trường làm việc tách biệt bằng cách tạo ra SparkSession mới từ một SparkSession cũ.
 - Chia sẻ CÙNG ngữ cảnh với nhau
 - Mọi thao tác trên ngữ cảnh sẽ tác động cùng lúc trên các SparkSession
 - Tuy nhiên các thiết lập và môi trường HOÀN TOÀN TÁCH BIỆT nhau.
- Sử dụng câu lệnh **SparkSession.builder()** nhiều lần sẽ chỉ nhận được một SparkSession.

Tạo RDD

```
# Biến một bộ sưu tập cục bộ thành RDD  
spark.sparkContext.parallelize([1, 2, 3])
```

```
# Đọc tập tin văn bản từ FS cục bộ, HDFS, hoặc S3  
spark.sparkContext.textFile("file.txt")  
spark.sparkContext.textFile("directory/*.txt")  
spark.sparkContext.textFile("hdfs://namenode:9000/path/file")
```

```
# Sử dụng bất kỳ Hadoop InputFormat nào  
spark.sparkContext.hadoopFile(keyClass, valClass, inputFmt,  
conf)
```

Biến đổi cơ bản

```
nums = spark.sparkContext.parallelize([1, 2, 3])
```

```
# Ánh xạ 1-1 từng phần tử vào RDD mới thông qua một hàm  
squares = nums.map(lambda x: x*x) # => {1, 4, 9}
```

```
# Chỉ đưa những phần tử thỏa điều kiện lọc vào RDD mới  
even = squares.filter(lambda x: x % 2 == 0) # => {4}
```

```
# Ánh xạ mỗi phần tử thành 0 hay nhiều phần tử trong RDD mới  
nums.flatMap(lambda x: range(0, x)) # => {0, 0, 1, 0, 1, 1, 2}
```

Đối tượng range (chuỗi các số
0, 1, ..., x-1)

Hành động cơ bản

```
nums = spark.sparkContext.parallelize([1, 2, 3])  
  
# Thu thập giá trị RDD thành một bộ sưu tập cục bộ  
nums.collect() # => [1, 2, 3]  
  
# Trả về K phần tử đầu tiên  
nums.take(2) # => [1, 2]  
  
# Đếm số lượng phần tử  
nums.count() # => 3  
  
# Hợp nhất các phần tử bằng một hàm liên kết  
nums.reduce(lambda x, y: x + y) # => 6  
  
# Ghi các phần tử vào tập tin văn bản  
nums.saveAsTextFile("hdfs://file.txt")
```

Cặp khóa – giá trị

- Thao tác biến đổi reduce của Spark hoạt động dựa trên RDD của các cặp khóa – giá trị
- Python:

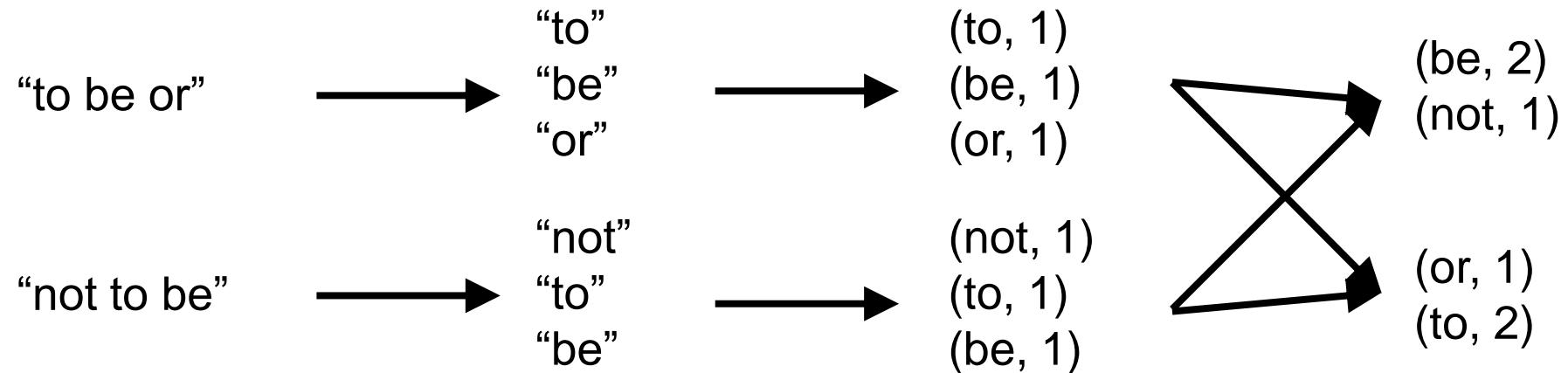
```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

```
pets = spark.sparkContext.parallelize([('cat', 1), ('dog', 1), ('cat', 2)])  
pets.reduceByKey(lambda x, y: x + y)  
# => {('cat', 3), ('dog', 1)}  
  
pets.groupByKey()  
# => {('cat', ResultIterable(1, 2)), ('dog', ResultIterable(1))}  
  
pets.sortByKey()  
# => {('cat', 1), ('cat', 2), ('dog', 1)}
```

reduceByKey cũng tự động xây dựng combiner ở giai đoạn map

Ví dụ: Đếm từ

```
lines = spark.sparkContext.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split(" "))
    .map(lambda word: (word, 1))
    .reduceByKey(lambda x, y: x + y)
```



Kết hợp các RDD

```
visits = spark.sparkContext.parallelize([('index.html', "1.2.3.4"),
                                         ('about.html', "3.4.5.6"),
                                         ('index.html', "1.3.3.1")])

pageNames = spark.sparkContext.parallelize([('index.html', "Home"),
                                             ('about.html', "About")])

visits.join(pageNames)
# ("index.html", ("1.2.3.4", "Home"))
# ("index.html", ("1.3.3.1", "Home"))
# ("about.html", ("3.4.5.6", "About"))

visits.cogroup(pageNames)
# ("index.html", tuple(ResultIterable("1.2.3.4", "1.3.3.1"), ResultIterable("Home")))
# ("about.html", tuple(ResultIterable("3.4.5.6"), ResultIterable("About")))
```

Kiểm soát mức độ song song hóa

- Tất cả các hoạt động RDD của cặp khóa – giá trị đều có tham số thứ hai để tùy chỉnh số tác vụ

```
words.reduceByKey(lambda x, y: x + y, 5)
```

```
words.groupByKey(5)
```

```
visits.join(pageViews, 5)
```

Sử dụng biến cục bộ

- Những biến bên ngoài được sử dụng trong các nút tính toán sẽ tự động được đóng gói gửi đến các cụm (broadcast):

```
query = raw_input("Enter a query:")
pages.filter(lambda x: x.startswith(query)).count()
```

- Một số lưu ý:
 - Mỗi tác vụ sẽ nhận được một bản sao (việc cập nhật giá trị sẽ không được gửi ngược lại)
 - Biến phải là Serializable (Java/Scala) hoặc Pickle-able (Python)
 - Không sử dụng các thuộc tính của một đối tượng bên ngoài (vì Spark sẽ đóng gói và gửi toàn bộ đối tượng đó!)

Chia sẻ giá trị giữa các nút tính toán

Biến lan truyền

- Các biến lan truyền (broadcast) cho phép người dùng giữ một biến *chỉ đọc* được lưu vào bộ nhớ đệm trên mỗi nút, thay vì gửi một bản sao của biến đó trong mỗi tác vụ

```
broadcastV1 = spark.sparkContext.broadcast([1, 2, 3, 4, 5, 6])
broadcastV1.value
# [1, 2, 3, 4, 5, 6]
```

Chia sẻ giá trị giữa các nút tính toán

Biến tích lũy

- Biến tích lũy (accumulator) là các biến chỉ được “thêm vào”, hỗ trợ thu thập giá trị trong các tính toán song song một cách hiệu quả

```
accum = sc.accumulator(0)
```

```
accum.add(x)
```

```
accum.value
```

Ví dụ: ước lượng số π

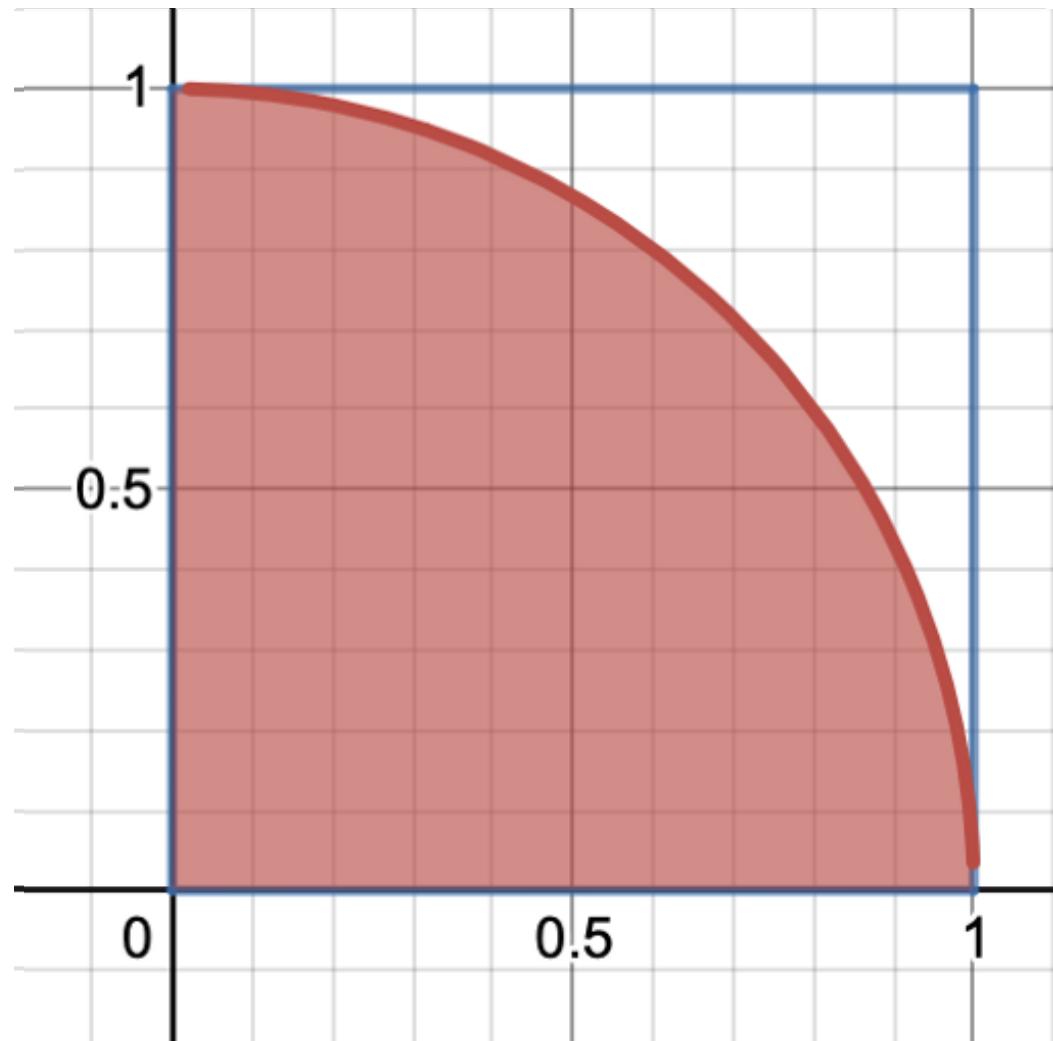
Bằng phương pháp Monte Carlo

- Cho hình vuông 1×1
- Và $\frac{1}{4}$ hình tròn ở gốc tọa độ
- Công thức tính diện tích hình tròn

$$S = \pi r^2$$

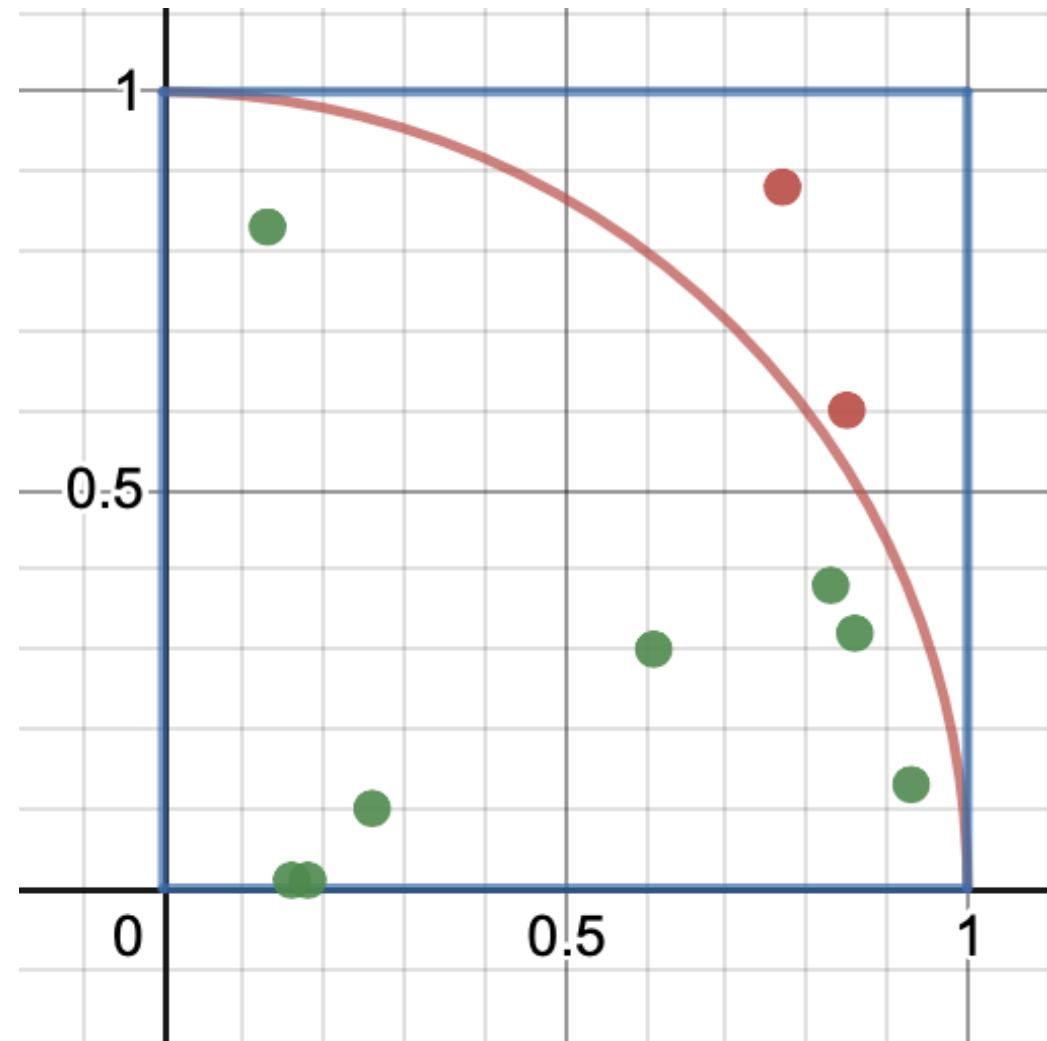
- Công thức tính diện tích $\frac{1}{4}$ hình tròn

$$S = \frac{\pi r^2}{4}$$



- Gieo ngẫu nhiên n (n đủ lớn) điểm vào hình vuông.
- Một điểm (x, y) nằm trong phần hình tròn khi

$$(x^2 + y^2) \leq 1$$



- Tỷ số diện tích giữa phần hình tròn và vuông là

$$k = \frac{\pi}{4} = \frac{\text{số điểm nằm trong } \frac{1}{4} \text{ hình tròn}}{n}$$

$$\Rightarrow \pi = 4 \times \frac{\text{số điểm nằm trong } \frac{1}{4} \text{ hình tròn}}{n}$$

```
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y <= 1

count = spark.sparkContext.parallelize( \
    range(0, NUM_SAMPLES)) \
    .filter(inside).count()

print("Pi xap xi %f" % (4.0 * count / NUM_SAMPLES))
```

Bài tập

- Cho đồ thị có hướng được ghi nhận trong tập tin như sau
 - 3 → 1, 2, 5
 - 1 → 2, 3
 - 4 → 3, 5
 - 5 → 1, 4
- Thiết kế hàm trong Spark đảo ngược chiều các cạnh của đồ thị
Kết quả mong đợi

1 → 3, 5
2 → 1, 3
3 → 1, 4
4 → 5
5 → 3, 4

TÀI LIỆU THAM KHẢO

1. Jean-Georges Perrin. 2020. **Spark in Action (2nd ed.)**. Manning Publications.
2. Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee. 2020. **Learning Spark (2nd ed.)**. O'Reilly Media, Inc.
3. Hien Luu. 2018. **Beginning Apache Spark 2**. Apress.
4. Bill Chambers, Matei Zaharia. 2018. **Spark: The Definitive Guide**. O'Reilly Media, Inc.

Q & A