

# APACHE SPARK

Dữ liệu lớn

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

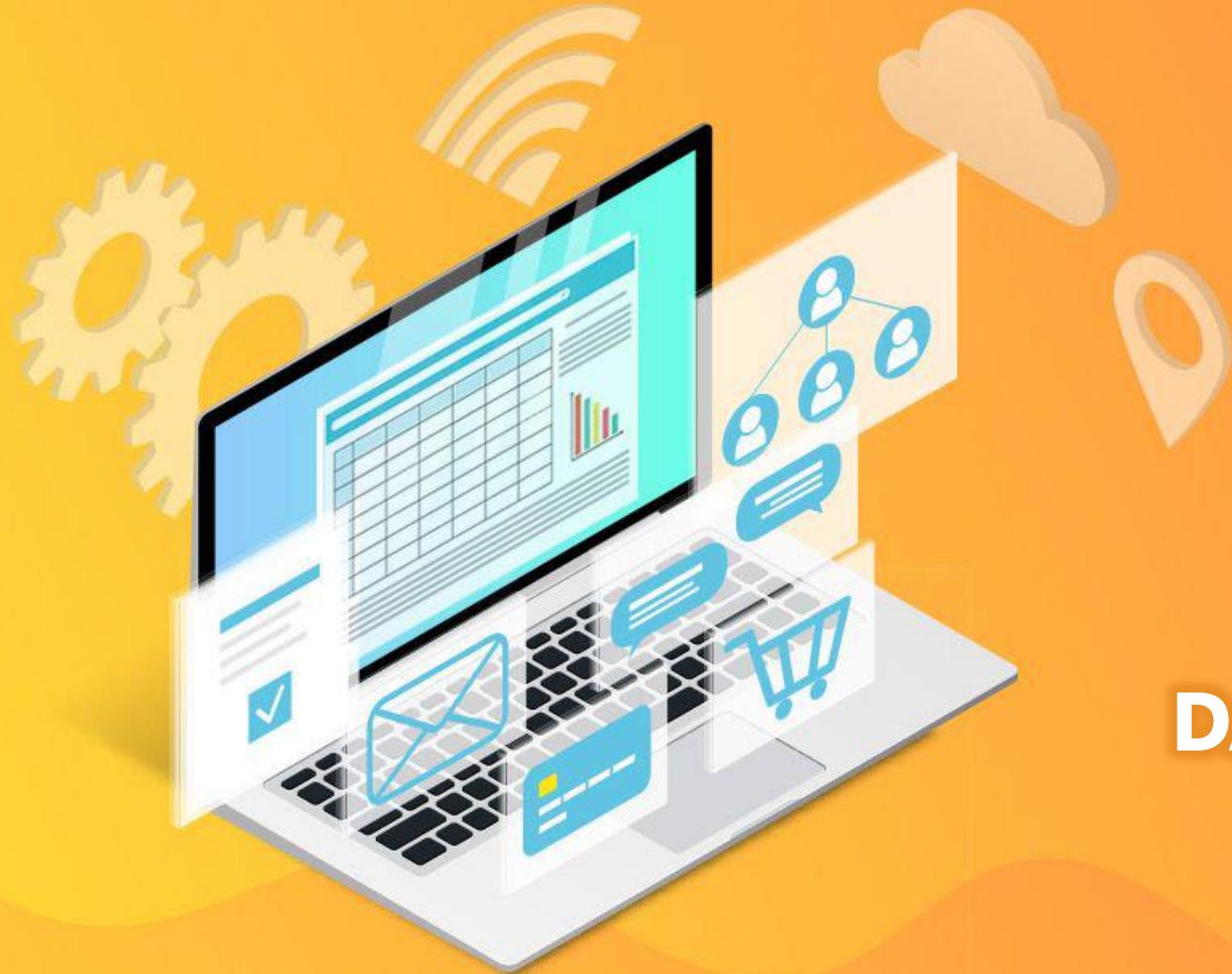


TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN  
KHOA HỆ THỐNG THÔNG TIN

2020

# NỘI DUNG





APACHE

# Spark

## DATASET & DATAFRAME

# Tập dữ liệu

## Datasets

- Là tập hợp dữ liệu bất biến được lưu trữ phân tán trong Spark
- Được thêm vào từ phiên bản Spark 1.6
- API chỉ có trong ngôn ngữ Scala và Java

- Kế thừa những ưu điểm của RDD
  - Quy định kiểu dữ liệu chặt chẽ (strong typing)
  - Khả năng áp dụng những hàm lambda mạnh mẽ
- Được tối ưu về khả năng khai thác thông tin dựa trên cấu trúc dữ liệu trong môi trường thực thi SQL.
  - Tính toán kế hoạch tốt nhất trước khi chạy chương trình

# Khung dữ liệu

## DataFrames

- Là một tập dữ liệu (bất biến) được tổ chức thành các cột được đặt tên.
- Về mặt khái niệm, nó tương đương với
  - Một bảng trong cơ sở dữ liệu quan hệ
  - Kiểu dữ liệu **data.frame** trong R, **Pandas Dataframe** trong Python nhưng có các tối ưu hóa phong phú hơn.
- Không bị ràng buộc quá chặt chẽ như RDD, Dataset
- API có sẵn bằng ngôn ngữ Scala, Java, Python và R.
- Xuất hiện từ phiên bản Spark 1.3

- Khung dữ liệu có thể được xây dựng từ nhiều nguồn như:
  - Tập tin dữ liệu có cấu trúc (CSV, JSON...)
  - Bảng trong Hive
  - Cơ sở dữ liệu bên ngoài
  - RDD/**Pandas Dataframe/list** hoặc **data.frame** có sẵn
    - Áp dụng các thao tác biến đổi/hành động trên khung dữ liệu có sẵn
- Trong Scala và Java, khung dữ liệu là tập dữ liệu (Dataset) của các dòng (Rows)
  - API Scala: **DataFrame = Dataset[Row]**
  - API Java: **DataFrame = Dataset<Row>**

# Lịch sử

- RDD là API đầu tiên.
- Sau đó, DataFrame được tạo ra để thao tác dễ dàng hơn. Nó được xây dựng dựa trên RDD và được lấy ý tưởng từ SQL (có thể chiếu các cột, nhóm dữ liệu...).
- Dataset là một cải tiến của DataFrame với những ràng buộc chặt chẽ. Nó là một phần mở rộng của DataFrame API. Được thêm vào ở phiên bản Spark 1.6 dưới dạng một API thử nghiệm.

- Từ phiên bản Spark 2.0 trở đi, Dataset và DataFrame được hợp nhất. **DataFrame** là bí danh của **Dataset[Row]**.

Ngôn ngữ	Lớp trừu tượng chính
Scala	Dataset[T] & DataFrame (bí danh của Dataset[Row])
Java	Dataset<T>
Python	DataFrame
R	DataFrame

# RDD vs. Dataset vs. DataFrame

## Cấp độ của API

- RDD: là API ở mức thấp. Với RDD, người lập trình có thể tự kiểm soát những hành động của mình nhiều hơn.
- DataFrame/Dataset: là API cấp cao. Các thao tác được tối ưu một cách tự động.

## Định nghĩa

- **RDD**: là một tập hợp phân tán của các đối tượng Java hoặc Scala đại diện cho các phần tử dữ liệu trải rộng trên nhiều máy trong cụm.
- **DataFrame**: là một tập hợp dữ liệu phân tán được tổ chức thành các cột được đặt tên, giống như một bảng trong CSDL quan hệ.
- **Dataset**: là phần mở rộng của DataFrame API với những ràng buộc chặt chẽ về kiểu dữ liệu (type safe), hình thức lập trình hướng đối tượng của RDD API và các lợi ích về hiệu suất của trình tối ưu hóa truy vấn Catalyst và cơ chế lưu trữ off-heap của DataFrame API.

## Định dạng dữ liệu

- **RDD**: có thể xử lý dữ liệu có cấu trúc cũng như phi cấu trúc dễ dàng và hiệu quả. Nhưng giống như DataFrame và Dataset, RDD không suy ra lược đồ của dữ liệu mà yêu cầu người dùng chỉ định.
- **DataFrame**: chỉ hoạt động trên dữ liệu có cấu trúc và bán cấu trúc. Tổ chức dữ liệu trong các cột được đặt tên và cho phép Spark quản lý lược đồ của dữ liệu.
- **Dataset**: xử lý hiệu quả dữ liệu có cấu trúc và phi cấu trúc. Biểu diễn dữ liệu dưới dạng các đối tượng JVM chứa các dòng hoặc một tập hợp các đối tượng dòng. Có thể biểu diễn dưới dạng bảng thông qua bộ mã hóa (encoder).

## Nguồn dữ liệu

- **RDD/DataFrame/DataSet**: cả ba API này đều có thể đọc được các nguồn dữ liệu đa dạng như:
  - Tập tin: văn bản, CSV, JSON, Parquet, HDFS, AVRO...
  - Cơ sở dữ liệu: SQL(Hive, MySQL...), NoSQL (MongoDB, Cassandra...)
  - Các hệ thống khác: luồng trực tuyến (Kafka, Storm...)...
  - ...

## Kiểm soát kiểu dữ liệu tại thời điểm biên dịch

- **RDD**: cung cấp hình thức lập trình kiểu hướng đối tượng quen thuộc và hỗ trợ kiểm soát kiểu dữ liệu tại thời điểm biên dịch.
- **DataFrame**: nếu người dùng cố gắng truy cập vào một cột dữ liệu không tồn tại trong bảng, DataFrame API sẽ không báo lỗi khi biên dịch. Nó chỉ phát hiện lỗi thuộc tính trong lúc chạy chương trình.
- **Dataset**: hỗ trợ kiểm soát kiểu dữ liệu tại thời điểm biên dịch.

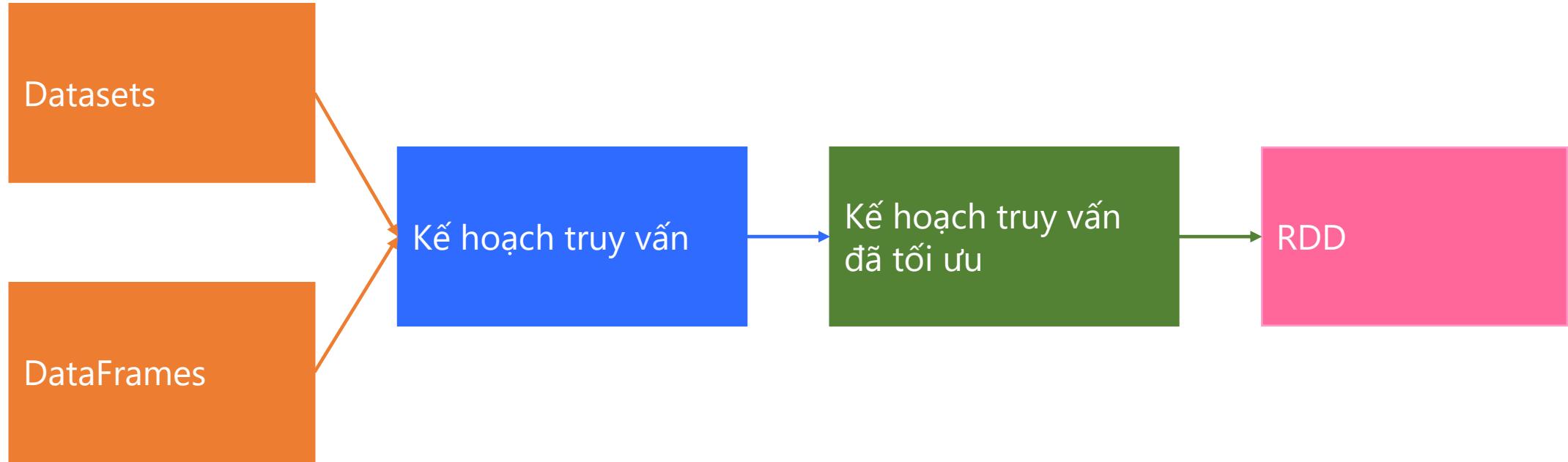
## Thời điểm phát hiện lỗi

	SQL	DataFrames	Datasets
Lỗi cú pháp	Lúc thực thi	Lúc biên dịch	Lúc biên dịch
Lỗi phân tích	Lúc thực thi	Lúc thực thi	Lúc biên dịch
Mức độ ràng buộc kiểu dữ liệu tăng dần →			

## Tối ưu hóa

- RDD: không có công cụ tối ưu hóa sẵn có. Khi làm việc với dữ liệu có cấu trúc, RDD không thể tận dụng lợi thế của các trình tối ưu hóa nâng cao của Spark. Ví dụ, trình tối ưu hóa Catalyst và công cụ thực thi Tungsten. Lập trình viên sẽ phải tối ưu hóa từng RDD trên cơ sở các thuộc tính của nó.

- **DataFrame/Dataset**: quá trình tối ưu hóa diễn ra bằng trình tối ưu hóa Catalyst. DataFrame/Dataset sử dụng framework biến đổi cây catalyst bao gồm 04 giai đoạn:
  - Phân tích kế hoạch luận lý để tìm ra các tham chiếu.
  - Tối ưu hóa kế hoạch luận lý.
  - Lập kế hoạch vật lý.
  - Biên dịch các phần của truy vấn sang Java bytecode.



## Tuần tự hóa (Serialization)

- **RDD**: Bất cứ khi nào Spark cần phân tán dữ liệu trong cụm hoặc ghi dữ liệu vào đĩa, nó sẽ sử dụng cơ chế tuần tự hóa của Java. Chi phí tuần tự hóa các đối tượng Java và Scala riêng lẻ rất低廉 và yêu cầu các nút trao đổi cả dữ liệu và cấu trúc cho nhau.

- **DataFrame**: có thể tuần tự hóa dữ liệu vào bộ lưu trữ off-heap (trong bộ nhớ) ở dạng nhị phân và sau đó thực hiện nhiều phép biến đổi trực tiếp trên bộ nhớ off-heap này. Vì Spark hiểu được lược đồ dữ liệu nên không cần sử dụng cơ chế tuần tự hóa của Java để mã hóa dữ liệu. Nó có sẵn một thành phần hỗ trợ thực thi vật lý Tungsten để quản lý bộ nhớ minh bạch và tự động tạo mã bytecode nhằm tính toán giá trị của các biểu thức.

- **DataSet**: trong việc tuần tự hóa dữ liệu, Dataset API trong Spark có một bộ mã hóa phụ trách chuyển đổi giữa các đối tượng JVM sang dạng bảng. Nó lưu trữ biểu diễn dạng bảng này bằng cách sử dụng định dạng nhị phân của Tungsten. Dataset cho phép thực hiện thao tác trên dữ liệu đã được tuần tự hóa và cải thiện việc sử dụng bộ nhớ. Nó cho phép truy cập theo yêu cầu vào các thuộc tính riêng lẻ mà không cần phải giải mã toàn bộ đối tượng.

## Rác bộ nhớ

- **RDD**: mất chi phí để thu thập rác do tạo và hủy các đối tượng riêng lẻ.
- **DataFrame**: tránh chi phí thu gom rác trong việc xây dựng các đối tượng riêng lẻ cho mỗi hàng trong tập dữ liệu.
- **DataSet**: không cần trình thu gom rác để hủy đối tượng vì quá trình tuần tự hóa diễn ra thông qua Tungsten. Lúc này, các đối tượng sẽ được lưu trữ trong off-heap.

## Hiệu quả sử dụng bộ nhớ

- **RDD**: hiệu quả bị giảm đi khi tuần tự hóa được thực hiện riêng lẻ trên từng đối tượng Java và Scala, điều này mất rất nhiều thời gian.
- **DataFrame**: sử dụng bộ nhớ off-heap để tuần tự hóa làm giảm chi phí. Tạo mã bytecode động để có thể thực hiện nhiều thao tác trên dữ liệu đó. Không cần giải mã dữ liệu cho các thao tác nhỏ.
- **DataSet**: cho phép thực hiện một hoạt động trên dữ liệu được tuần tự hóa và tối ưu việc sử dụng bộ nhớ. Vì vậy, API này cho phép thực hiện yêu cầu truy cập vào thuộc tính riêng lẻ mà không cần giải mã toàn bộ đối tượng.

## Trì hoãn tính toán

- **RDD/DataFrame/DataSet**: cả ba API này đều hoạt động theo phương thức trì hoãn tính toán (lazy evaluation). Các thao tác biến đổi sẽ không được thực thi ngay lập tức, mà thay vào đó, Spark sẽ ghi nhận lại. Chỉ khi có một thao tác hành động được gọi thì Spark mới tiến hành thực thi toàn bộ.

## Ngôn ngữ lập trình hỗ trợ

- **RDD/DataFrame**: là API có trên tất cả các ngôn ngữ mà Spark có thể làm việc, bao gồm Scala, Java, Python và R.
- **DataSet**: hiện tại mới chỉ hỗ trợ ngôn ngữ Scala và Java.

## Lược đồ dữ liệu

- **RDD**: sử dụng phép chiếu lược đồ được sử dụng một cách rõ ràng. Do đó, người dùng cần tự xác định lược đồ theo cách thủ công.
- **DataFrame/DataSet**: tự động phát hiện cấu trúc từ các tập tin mà không cần định nghĩa trước lược đồ của dữ liệu và hiển thị chúng dưới dạng bảng thông qua Hive Metastore. Và điều này cũng giúp API có thể kết nối với các nguồn dữ liệu SQL tiêu chuẩn.

## Thực thi các tác vụ tổng hợp

- **RDD**: API này chậm hơn trong việc thực hiện các thao tác gom nhóm và tổng hợp đơn giản.
- **DataFrame/DataSet**: rất dễ sử dụng và tốc độ thực thi nhanh hơn trong việc phân tích khám phá, tạo thống kê tổng hợp trên nhiều tập dữ liệu và các tập dữ liệu lớn.

## Sử dụng

- **RDD:** được sử dụng khi
  - Lập trình viên cần thực hiện những thao tác biến đổi/hành động và kiểm soát cấp thấp trên tập dữ liệu.
  - Xử lý những dữ liệu không có cấu trúc như luồng trực tuyến đa phương tiện/văn bản...
  - Muốn xử lý dữ liệu với các cấu trúc lập trình hàm hơn là sử dụng những biểu thức chuyên biệt.
  - Không cần áp đặt lược đồ lên dữ liệu, chẳng hạn như định dạng các cột trong khi xử lý hoặc truy cập các thuộc tính của dữ liệu theo tên/cột
  - Có thể bỏ qua một số lợi ích của việc tối ưu hóa và hiệu suất có sẵn với DataFrame và Dataset đối với dữ liệu có cấu trúc và bán cấu trúc.

- **DataFrame/DataSet**: được sử dụng khi
  - Người dùng muốn nội dung lập trình phong phú hơn (SQL, hàm đặc biệt, lập trình hàm...), trừu tượng cấp cao và API chuyên biệt.
  - Quá trình xử lý yêu cầu những biểu thức cao cấp như lọc dữ liệu, ánh xạ, tổng hợp, trung bình, tổng, truy vấn SQL, truy cập các cột và sử dụng các hàm lambda trên dữ liệu bán cấu trúc.
  - Muốn sử dụng trực tiếp các đối tượng JVM, tiết kiệm không gian lưu trữ và thời gian tuần tự hóa/giải mã dữ liệu.
  - Mong muốn tận dụng tối ưu hóa Catalyst và hưởng lợi từ việc tạo mã hiệu quả của Tungsten.

- Nếu muốn kiểm soát kiểu dữ liệu ở mức độ an toàn cao hơn tại thời điểm biên dịch
- Nếu muốn thống nhất và đơn giản hóa các API trên các thư viện Spark.
- Nếu lập trình viên là người dùng R, hãy sử dụng DataFrame.
- Nếu là người dùng Python, hãy sử dụng DataFrame và sử dụng thêm RDD khi cần kiểm soát nhiều hơn.

# Ví dụ: DataFrame

```
from pyspark.sql import Row  
  
#Tao cac dong du lieu  
dept1 = Row(id='123456', name='Computer Science')  
dept2 = Row(id='789012', name='Mechanical Engineering')  
dept3 = Row(id='345678', name='Theater and Drama')  
dept4 = Row(id='901234', name='Indoor Recreation')
```

## *#Tao DataFrame Departments*

```
dept = [dept1, dept2, dept3, dept4]
deptDF = spark.createDataFrame(dept)
deptDF.show()
```

```
+-----+-----+
|    id|      name|
+-----+-----+
|123456|Computer Science|
|789012|Mechanical Engine...|
|345678|Theater and Drama|
|901234|Indoor Recreation|
+-----+-----+
```

```
#Doc tap tin .csv goc
peopleDF = spark.read.format("csv").load("people.csv")
print("Doc tap tin .csv goc (people.csv):")
peopleDF.show()
```

Doc tap tin .csv goc (people.csv):

```
+-----+
|      _c0 |
+-----+
|  name;age;job|
| Jorge;30;Developer|
| Bob;32;Developer|
+-----+
```

```
#Doc tap tin .csv bang SQL
peopleSqlDF = spark.sql("SELECT * FROM csv.`people.csv`")
print("Doc tap tin .csv bang SQL:")
peopleSqlDF.show()
```

Doc tap tin .csv bang SQL:

```
+-----+
|      _c0 |
+-----+
| name;age;job|
| Jorge;30;Developer|
| Bob;32;Developer|
+-----+
```

```
#Su dung .option() de tai to chuc du Lieu trong tap tin .csv
peopleDFwOpt = spark.read.format("csv") \
    .option("sep", ";") \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .load("people.csv")
print("Su dung .option() de tai to chuc du lieu trong tap tin .csv:")
peopleDFwOpt.show()
```

Su dung .option() de tai to chuc du lieu trong tap tin .csv:

```
+----+---+-----+
| name|age|      job|
+----+---+-----+
|Jorge| 30|Developer|
|   Bob| 32|Developer|
+----+---+-----+
```



# Tại sao phải kết hợp với SQL

- SQL tương thích với các công cụ
  - Kết nối với các công cụ BI thông qua kết nối JDBC/ODBC
- SQL có số lượng lớn người dùng thành thạo
- Diễn đạt ngắn gọn hơn/dễ hiểu hơn so với ngôn ngữ lập trình bình thường khác/mô hình MapReduce.

# Trước đây...

- Pig, Hive, Dremel, Shark
  - Tận dụng các khai báo truy vấn để cung cấp phương pháp tối ưu hóa tự động phong phú hơn.
  - Phương pháp tiếp cận theo mô hình quan hệ không đủ cho các ứng dụng dữ liệu lớn
    - ETL từ/đến các nguồn dữ liệu bán/phi cấu trúc (ví dụ: JSON) phải viết riêng mã
    - Thực hiện các phân tích nâng cao (máy học và xử lý đồ thị) là thách thức trong hệ thống quan hệ.



Quá trình thực thi trong Hive và Pig

- Vấn đề là
  - Các công cụ có quy trình giống nhau, tuy nhiên mỗi công cụ lại có những Trình tối ưu hóa/thực thi khác nhau
  - Không có một cấu trúc dữ liệu dùng chung cho Hive/Pig/các DSL khác

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.contains("ERROR"))  
print(errors.count())
```

- Mỗi RDD (**lines**, **errors**) đại diện cho một “kế hoạch luận lý” tính toán một tập dữ liệu, nhưng Spark đợi cho đến khi một hành động được gọi, **count**, để khởi chạy phép tính.
- Spark sẽ thực hiện tuần tự đọc các dòng, áp dụng bộ lọc và đếm.
- Các thành phần trung gian là không cần thiết.
- RDD hữu ích nhưng còn hạn chế.

# Shark

- Dự án đầu tiên xây dựng một phương pháp giao tiếp dựa trên mô hình quan hệ với Spark.
- Sửa đổi hệ thống Apache Hive với các tối ưu hóa RDBMS truyền thống.
- Cho thấy hiệu suất tốt và cơ hội tích hợp với các chương trình Spark.
- Khó khăn
  - Chỉ truy vấn các nguồn dữ liệu bên ngoài trong danh mục của Hive và do đó không áp dụng cho các truy vấn dữ liệu bên trong chương trình Spark (truy vấn trên RDD)
  - Bất tiện và không thể tránh khỏi lỗi khi làm việc.
  - Trình tối ưu hóa Hive được thiết kế riêng cho MapReduce và khó mở rộng.

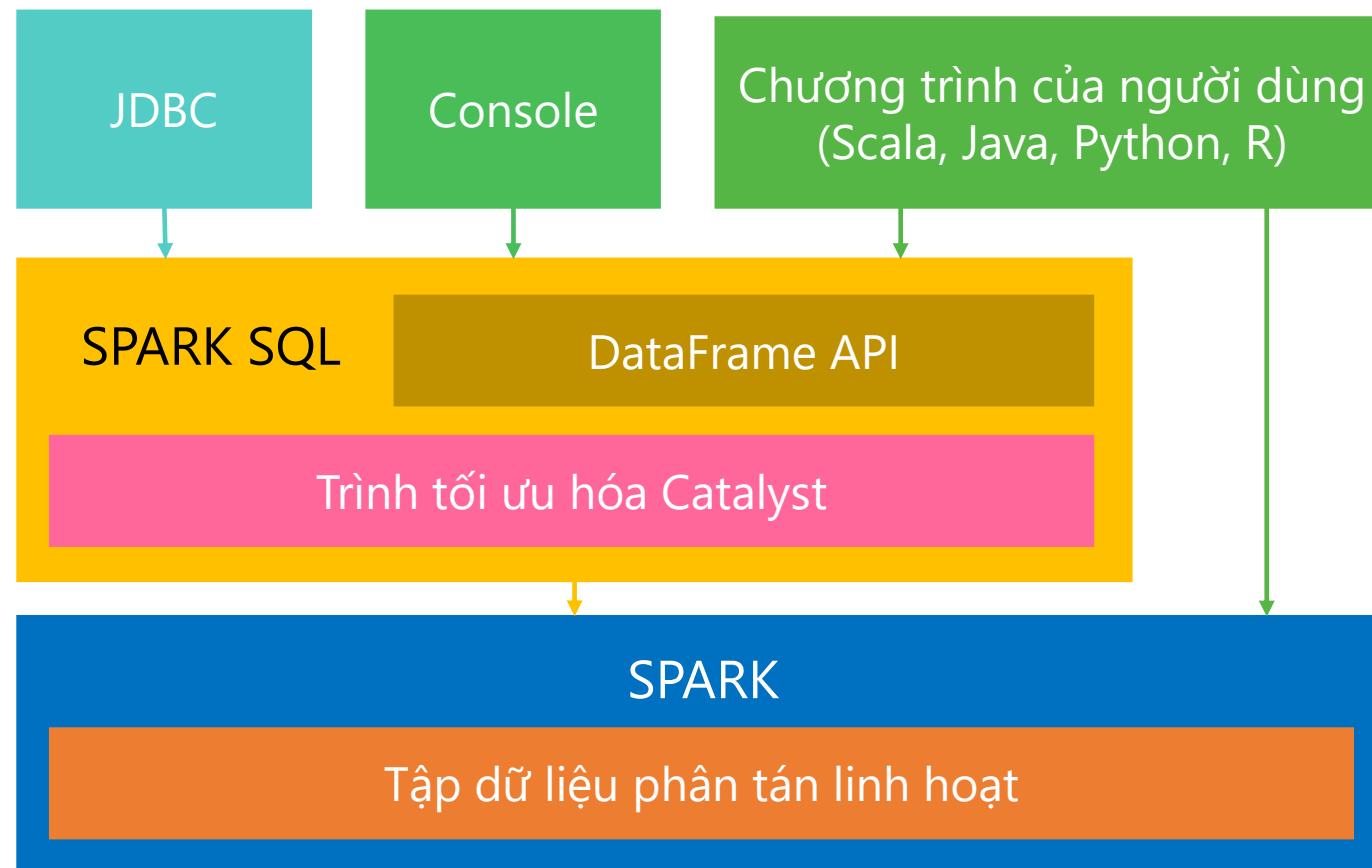
# Spark SQL

- Một mô-đun trong Apache Spark tích hợp xử lý theo mô hình quan hệ với mô hình API lập trình hàm của Spark.
- Tích hợp chặt chẽ hơn nhiều giữa quan hệ và thủ tục trong quá trình xử lý, thông qua API đặc tả DataFrame.
- Bao gồm trình tối ưu hóa có khả năng mở rộng cao, Catalyst, giúp dễ dàng thêm nguồn dữ liệu, quy tắc tối ưu hóa và kiểu dữ liệu.

# Mục đích của Spark SQL

- Hỗ trợ xử lý theo mô hình quan hệ cả trong chương trình Spark và các nguồn dữ liệu bên ngoài bằng cách sử dụng API thân thiện với lập trình viên.
- Nâng cao hiệu suất bằng cách sử dụng các kỹ thuật DBMS đã thiết lập sẵn.
- Dễ dàng hỗ trợ các nguồn dữ liệu mới, bao gồm dữ liệu bán cấu trúc và cơ sở dữ liệu bên ngoài có thể liên kết truy vấn.
- Dễ dàng mở rộng với các thuật toán phân tích nâng cao như xử lý đồ thị và máy học.

## Tương tác giữa Spark SQL và các thành phần khác



# DataFrame API

```
users = spark.hiveContext.table("users")
young = users.where(users("age") < 21)
print(young.count())
```

- Tương đương với bảng trong cơ sở dữ liệu quan hệ
- Có thể thao tác như một RDD bình thường.

# Ví dụ

Sử dụng RDD

```
data = spark.sparkContext.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 1])) \
    .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
    .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
    .collect()
```

Sử dụng SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

Sử dụng DataFrame

```
sqlContext.table("people") \
    .groupBy("name") \
    .agg("name", avg("age")) \
    .collect()
```

# Mô hình dữ liệu

- Sử dụng mô hình dữ liệu lồng nhau dựa trên Hive để xây dựng các bảng và DataFrame
  - Hỗ trợ tất cả các kiểu dữ liệu SQL chính
  - Hỗ trợ các kiểu dữ liệu do người dùng định nghĩa
  - Có thể lập mô hình dữ liệu từ nhiều nguồn và định dạng khác nhau (ví dụ: Hive, RDB, JSON và các đối tượng gốc trong Java/Scala/Python)

# Các thao tác của DataFrame

## Employees

```
.join(dept, employees("deptId") === dept("id"))
.where(employees("gender") === "female")
.groupBy(dept("id"), dept("name"))
.agg(count("name"))
```

```
users.where(users("age") < 21)
    .registerTempTable("young")
sqlContext.sql("SELECT count(*), avg(age)
    FROM young")
```

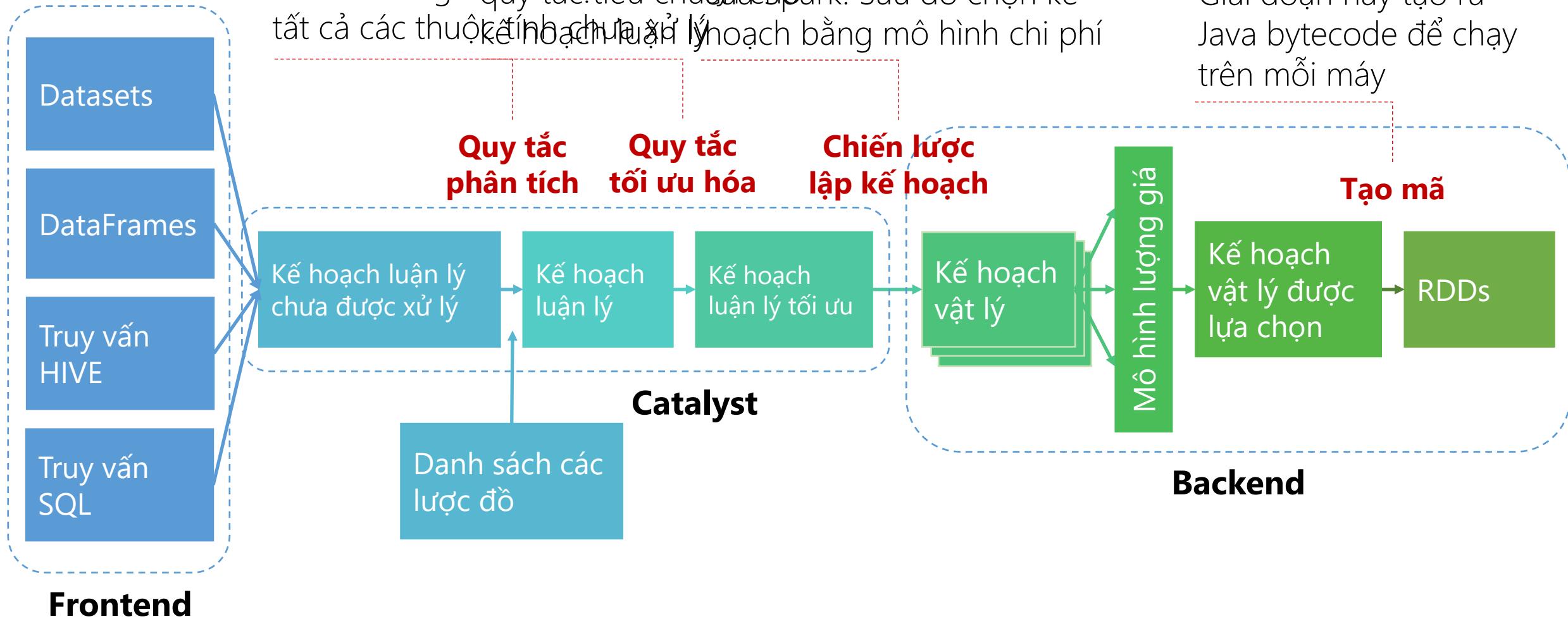
- Tất cả các toán tử trên sẽ xây dựng một cây cú pháp trừu tượng (abstract syntax tree - AST) của biểu thức, sau đó được chuyển đến Catalyst để tối ưu hóa.
- Các DataFrame được đăng ký trong danh mục vẫn có thể bị hủy đăng ký, do đó, tối ưu hóa có thể áp dụng trên SQL và các DataFrame ban đầu.
- Tích hợp trong tất cả các ngôn ngữ lập trình Spark hỗ trợ (DataFrame có thể được chuyển đổi giữa các ngôn ngữ khác nhau nhưng vẫn được hưởng lợi từ việc tối ưu hóa trên toàn bộ chương trình).

# Trình tối ưu hóa Catalyst

- Dựa trên cấu trúc lập trình hàm trong Scala.
- Dễ dàng thêm các kỹ thuật và tính năng tối ưu hóa mới,
  - Đặc biệt là để giải quyết các vấn đề khác nhau khi xử lý "dữ liệu lớn" (ví dụ: dữ liệu bán cấu trúc và phân tích nâng cao)
- Cho phép các nhà phát triển bên ngoài mở rộng trình tối ưu hóa.
  - Các quy tắc cụ thể của nguồn dữ liệu có thể đẩy kết quả lọc hoặc kết quả tổng hợp vào hệ thống lưu trữ bên ngoài
  - Hỗ trợ kiểu dữ liệu mới
- Hỗ trợ tối ưu hóa dựa trên quy tắc và dựa trên chi phí

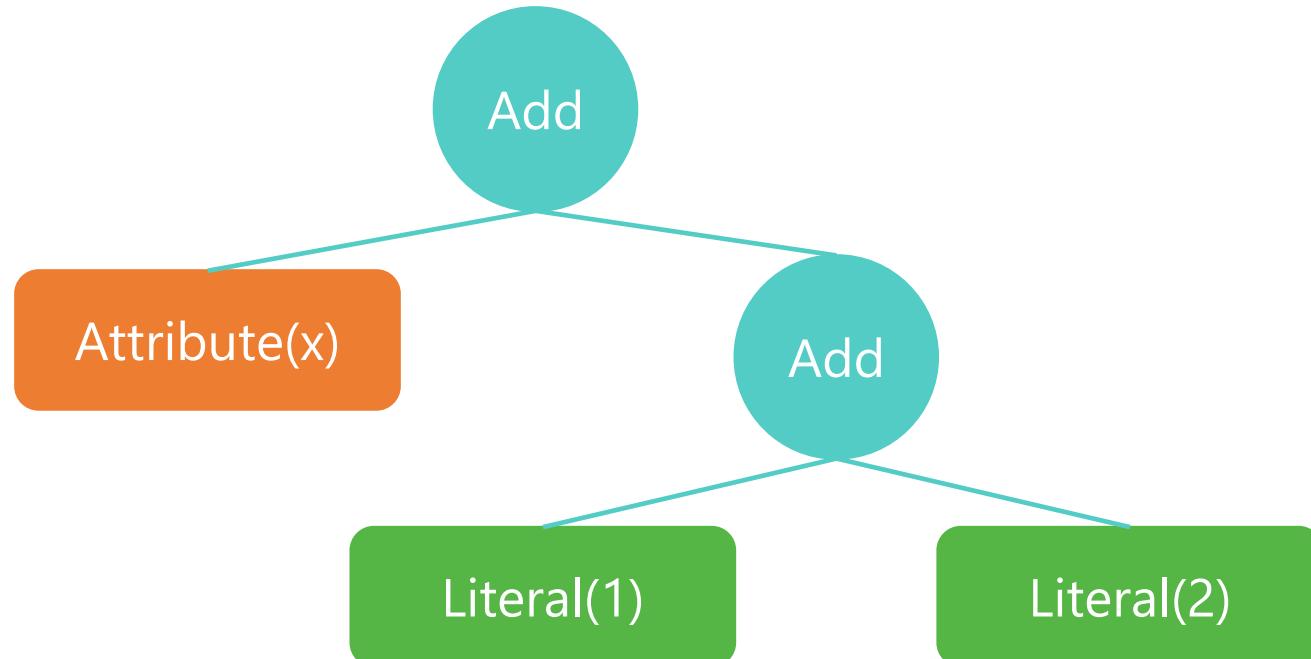
Spark SQL sử dụng các quy tắc tạo ra một hay nhiều kế hoạch Catalyst và danh sách các lược đồ. catalyst sử dụng các toán tử vật lý tương để kiểm tra khả năng trong tệp và hợp với bộ phận thực thi tất cả các nguồn dữ liệu để xác định kế hoạch spark. Sau đó chọn kế hoạch bằng mô hình chi phí tất cả các thuật toán bao gồm

Giai đoạn này tạo ra Java bytecode để chạy trên mỗi máy



# Cây Catalyst

Scala Code: `Add(Attribute(x), Add(Literal(1), Literal(2)))`



Cây catalyst biểu diễn phép tính  $x + (1 + 2)$

# Quy tắc tối ưu hóa

- Cây được biến đổi bằng các quy tắc, là các hàm tạo ra cây mới từ cây cũ.
  - Sử dụng một tập hợp các hàm so khớp mẫu để tìm và thay thế các cây con bằng một cấu trúc cụ thể.

```
tree.transform{  
    case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)  
}  
tree.transform {  
    case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)  
    case Add(left , Literal(0)) => left  
    case Add(Literal(0), right) => right  
}
```

- Catalyst nhóm các quy tắc thành từng bộ và thực hiện lần lượt cho đến khi đạt đến một điểm cố định.

## Phân tích câu truy vấn

- Nhận đầu vào từ trình phân tích cú pháp SQL hoặc đối tượng DataFrame
- Một đối tượng “chưa xử lý”: chưa khớp đối tượng với danh sách các bảng đầu vào hoặc không biết kiểu dữ liệu
- Danh sách các lược đồ dùng để theo dõi các bảng trong tất cả các nguồn dữ liệu.
- Có khoảng 1000 quy tắc

## Tối ưu hóa luận lý

- Áp dụng các tối ưu hóa dựa trên quy tắc tiêu chuẩn cho kế hoạch luận lý
  - Xử lý hằng số (constant folding)
  - Đẩy các vị từ lọc xuống mức thấp nhất (predicate pushdown)
  - Làm gọn phép chiếu (projection pruning)
  - Xử lý các biểu thức chứa NULL (null propagation)
  - Đơn giản hóa biểu thức Boolean (boolean expression simplification)
  - ...
- Rất dễ để thêm các quy tắc cho một tình huống cụ thể
- Có khoảng 800 quy tắc

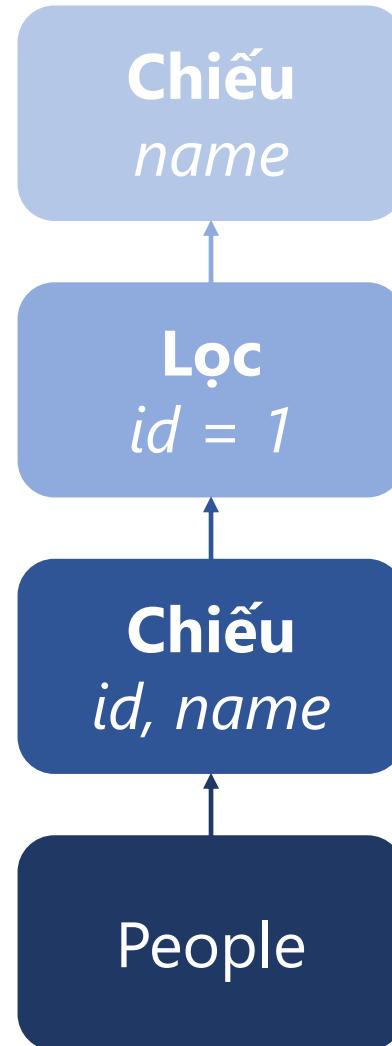
# Kế hoạch vật lý

- Từ kế hoạch luận lý có thể tạo ra một hay nhiều kế hoạch vật lý.
- Dựa trên chi phí
  - Chọn một kế hoạch sử dụng mô hình chi phí (hiện chỉ được sử dụng để chọn thuật toán kết hợp).
- Dựa trên quy tắc:
  - Kết hợp các phép chiếu hoặc bộ lọc thành một thao tác ánh xạ trong Spark
  - Đưa các thao tác đến gần nguồn dữ liệu nhất áp dụng kỹ thuật đẩy các vị từ lọc/phép chiếu xuống mức thấp nhất có thể.
- Có khoảng 500 quy tắc.

# Ví dụ

Cho câu truy vấn sau

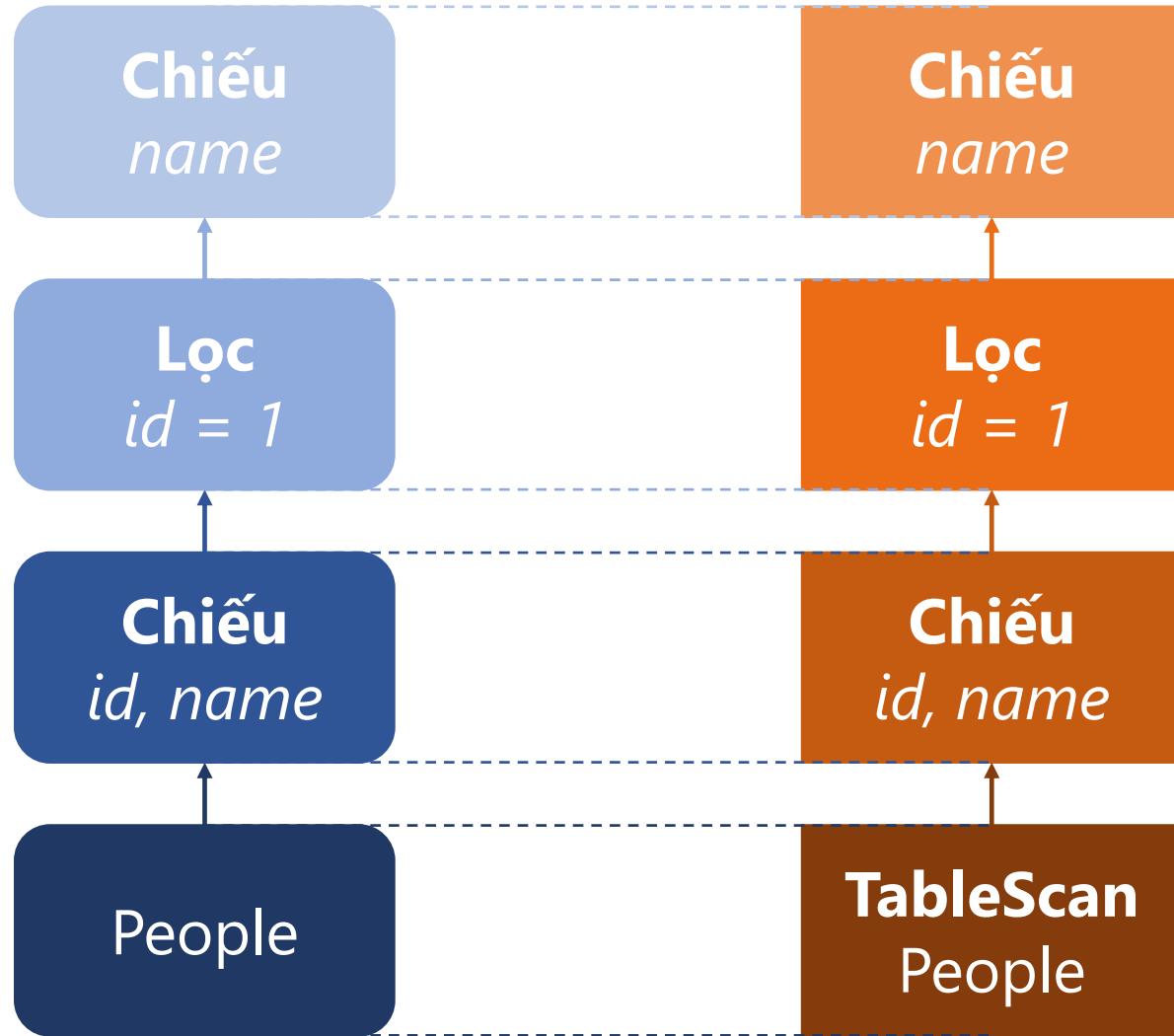
```
SELECT name  
FROM (  
    SELECT id, name  
    FROM People  
) p  
WHERE p.id = 1
```



Kế hoạch luận lý

Cho câu truy vấn sau

```
SELECT name  
FROM (  
    SELECT id, name  
    FROM People  
) p  
WHERE p.id = 1
```

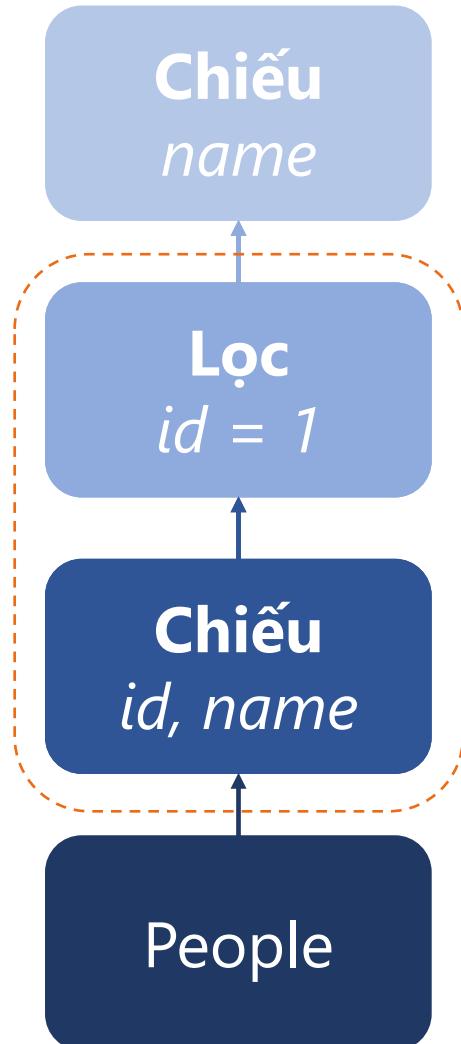


Kế hoạch luận lý

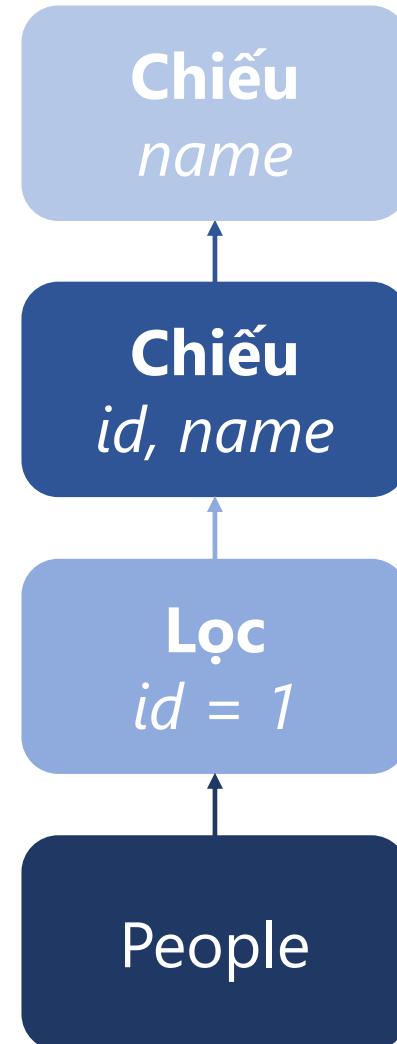
Kế hoạch vật lý

## Áp dụng các Luật tối ưu hóa

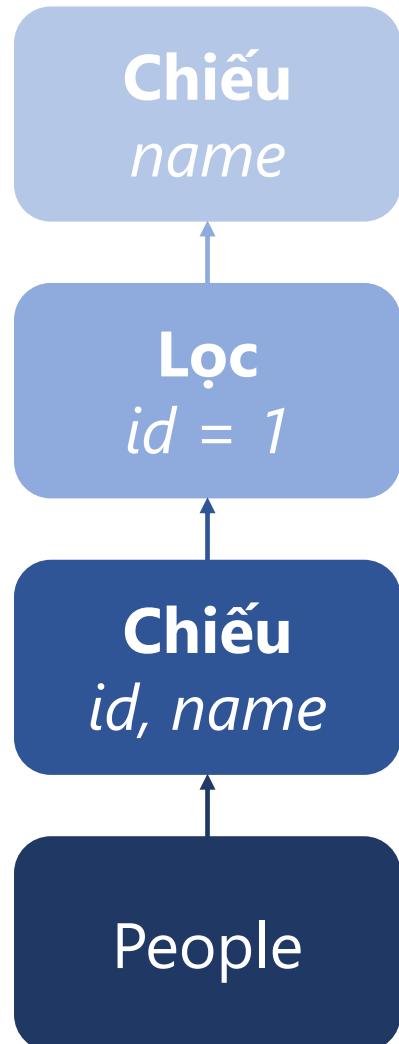
1. Tìm thao tác lọc ở trên phép chiếu
2. Kiểm tra phép lọc có thể được thực hiện mà không cần kết quả từ phép chiếu hay không
3. Nếu có, có thể đổi vị trí của hai thao tác này



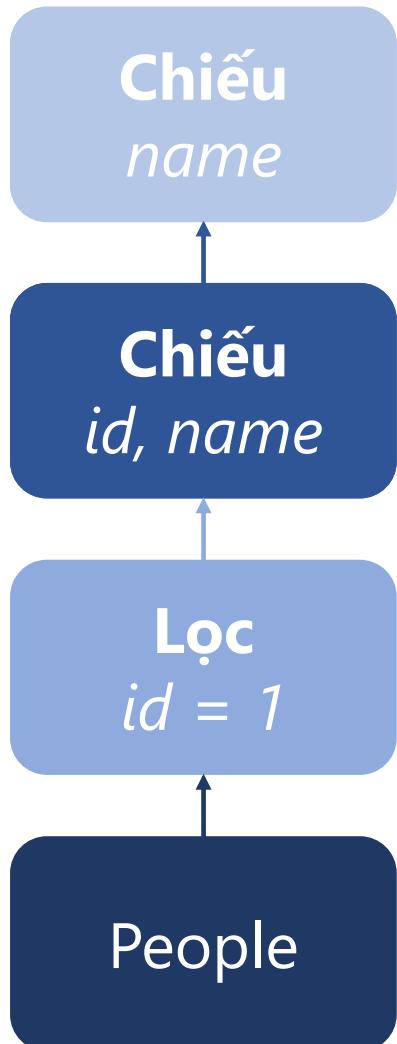
Kế hoạch ban đầu



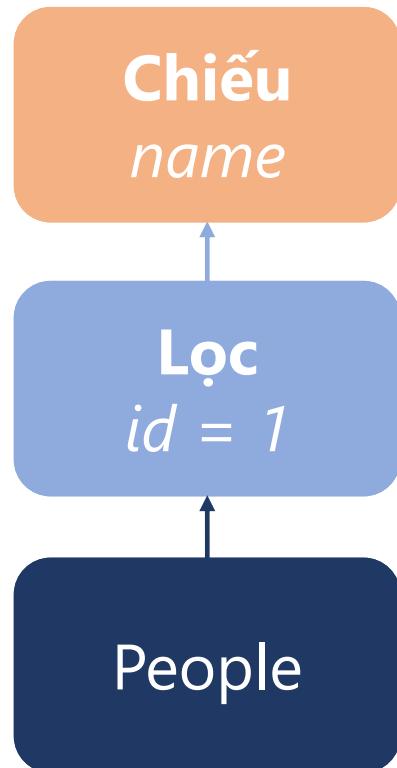
Đẩy thao tác lọc xuống dưới



Kế hoạch ban đầu



Đẩy thao tác  
lọc xuống dưới



Làm gọn  
các phép chiếu

**IndexLookup**  
 $id = 1$   
*return: name*

Kế hoạch  
vật lý

# Tạo mã

- Tạo mã Java bytecode để chạy trên từng máy.
- Dựa trên quasiquotes của Scala để biến đổi mã thành cây
- Chuyển đổi một cây đại diện cho một biểu thức SQL thành cây cú pháp trừu tượng trong Scala để thực thi biểu thức đó.
- Biên dịch (được tối ưu hóa trong Scala một lần nữa) và chạy mã đã tạo.
- Có khoảng 700 quy tắc

```
def compile(node: Node): AST = node match {  
    case Literal(value) => q"$value"  
    case Attribute(name) => q"row.get($name)"  
    case Add(left, right) => q"${compile(left)} + ${compile(right)}"  
}
```

# APACHE Spark

## MLLIB



# Giới thiệu

- Bắt đầu tại AMPLab, UC Berkeley
  - Được đưa vào Spark ở phiên bản 0.8
- Mục tiêu của MLlib là làm cho việc học máy thực tế trở nên dễ dàng và có khả năng mở rộng.
  - Có khả năng khai thác, học hỏi từ các bộ dữ liệu quy mô lớn
  - Dễ dàng xây dựng các ứng dụng máy học

## **spark.mllib vs. spark.ml**

- **spark.mllib** là API dựa trên RDD
  - Kể từ phiên bản 2.0, được đưa vào chế độ bảo trì
  - Vẫn còn được hỗ trợ sửa lỗi, không thêm chức năng mới
  - Không khuyến khích sử dụng
- **spark.ml** là API dựa trên DataFrame
  - Trở thành API máy học chính trong Spark từ phiên bản 2.0
  - Các chức năng đang được bổ sung để dần thay thế **spark.mllib**
  - Được khuyến khích sử dụng

## Tại sao spark.ml trở nên phổ biến hơn?

- API dựa trên DataFrame thân thiện với người dùng hơn RDD.
- DataFrame có nhiều lợi ích như: nguồn dữ liệu đa dạng, truy vấn SQL/DataFrame, tối ưu hóa Tungsten và Catalyst.
- MLlib API dựa trên DataFrame giúp thống nhất trên các thuật toán máy học, giữa các thành phần của Spark và trên nhiều ngôn ngữ.
- DataFrame tạo điều kiện thuận lợi cho các Quy trình học máy (ML Pipeline) thực tế, đặc biệt là thao tác biến đổi các thuộc tính.

## MLlib hỗ trợ các thuật toán

- 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
- 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...

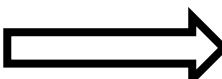
Ví dụ

## Phân lớp văn bản

Mục tiêu: Cho một văn bản, dự đoán chủ đề của nó.

### Các đặc trưng

Subject: Re: Lexan Polish?  
Suggest McQuires #1 plastic  
polish. It will help somewhat  
but nothing will remove deep  
scratches without making it  
worse than it already is.  
McQuires will do something...



văn bản, hình ảnh, vector, ...

### Nhãn

1: khoa học  
0: không nói về khoa học

\  
tần số nhấn chuột, lượng mưa, ...

## Huấn luyện

Cho dữ liệu đã gán nhãn:

DataFrame chứa (đặc trưng, nhãn)

Subject: Re: Lexan Polish?  
Suggest McQuires #1 plastic  
polish. It will help...

Nhãn 0

Subject: RIPEM FAQ  
RIPEM is a program which  
performs Privacy Enhanced...

Nhãn 1

...

Tạo ra một mô hình

## Kiểm thử/Thành phẩm

Cho dữ liệu mới chưa được gán nhãn:

DataFrame chứa các đặc trưng

Subject: Apollo Training  
The Apollo astronauts also  
trained at (in) Meteor... → Nhãn 1

Subject: A demo of Nonsense  
How can you lie about  
something that no one... → Nhãn 0

Sử dụng mô hình để dự đoán

Ví dụ

# Các bước xử lý trong máy học

Huấn luyện

Tải dữ liệu

↓  
*nhãn + văn bản thô*

Trích xuất đặc trưng

↓  
*nhãn + các vector đặc trưng*

Huấn luyện mô hình

↓  
*nhãn + nhãn dự đoán*

Đánh giá

Nhược điểm

Lập trình thành một kịch bản

- Không theo kiểu mô-đun
- Khó tái sử dụng các thao tác

## Huấn luyện

Tải dữ liệu

↓  
*nhãn + văn bản thô*

Trích xuất đặc trưng

↓  
*nhãn + các vector đặc trưng*

Huấn luyện mô hình

↓  
*nhãn + nhãn dự đoán*

Đánh giá

## Kiểm thử/Thành phẩm

Tải dữ liệu **mới**

↓  
*văn bản thô*

Trích xuất đặc trưng

↓  
*các vector đặc trưng*

Dự đoán bằng mô hình

↓  
*nhãn dự đoán*

Hành động theo dự đoán

*Các bước  
gần như  
giống hệt  
nhau*

# Huấn luyện



## Nhược điểm

Điều chỉnh tham số

- Là thành phần quan trọng trong máy học
- Liên quan đến huấn luyện nhiều mô hình
  - Cho các phần dữ liệu khác nhau
  - Cho những tập các tham số khác nhau

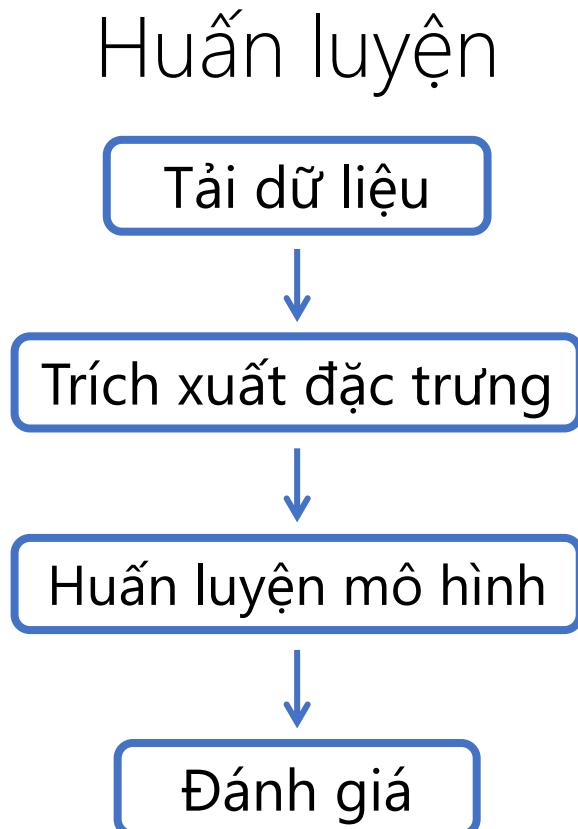
# Nhược điểm

Lập trình các bước xử lý như một kịch bản  
Khó khăn trong việc tối ưu hóa các tham số

*Pipelines!*

xuất hiện ở Spark 1.2 & 1.3

# Quy trình học máy



# Biến đổi

## Transformer

### Huấn luyện

Trích xuất đặc trưng

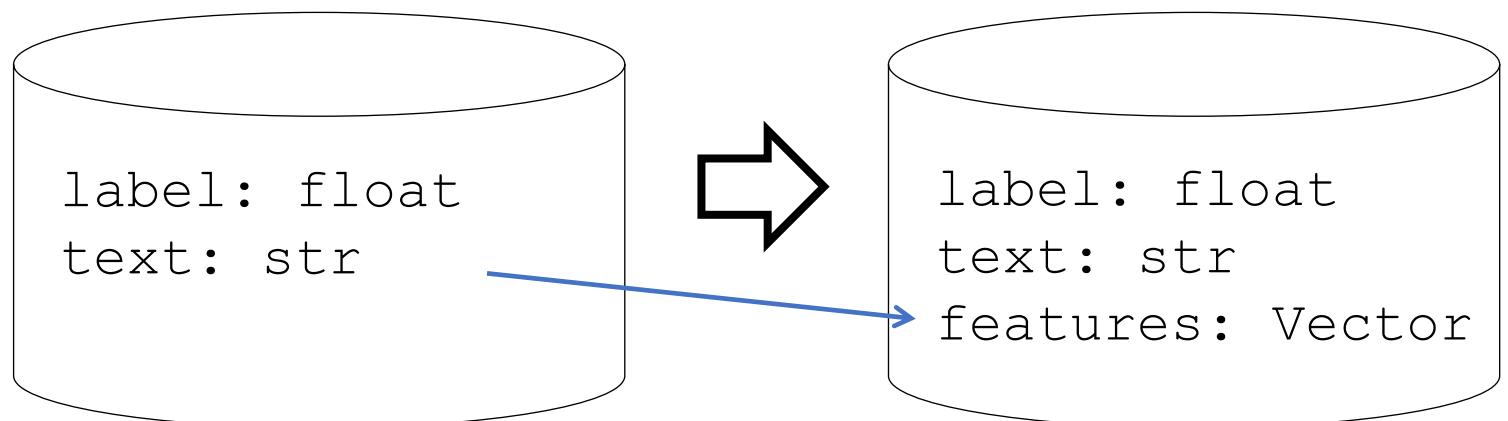


Huấn luyện mô hình



Đánh giá

```
def transform(self, dataset, params=None)  
    dataset: `pyspark.sql.DataFrame`  
    returns: transformed dataset
```



# Ước lượng

Estimator

Huấn luyện

Trích xuất đặc trưng

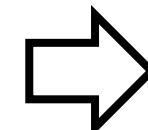
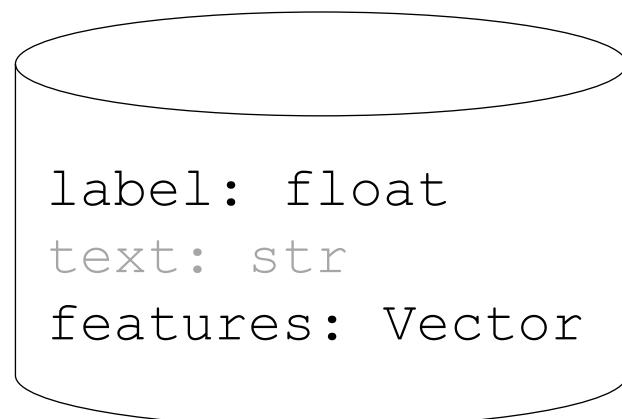


**Huấn luyện mô hình**



Đánh giá

```
def fit(self, dataset, params=None)  
    dataset: `pyspark.sql.DataFrame`  
    returns: fitted model(s)
```



LogisticRegression  
Model

# Đánh giá

Evaluator

Huấn luyện

Trích xuất đặc trưng

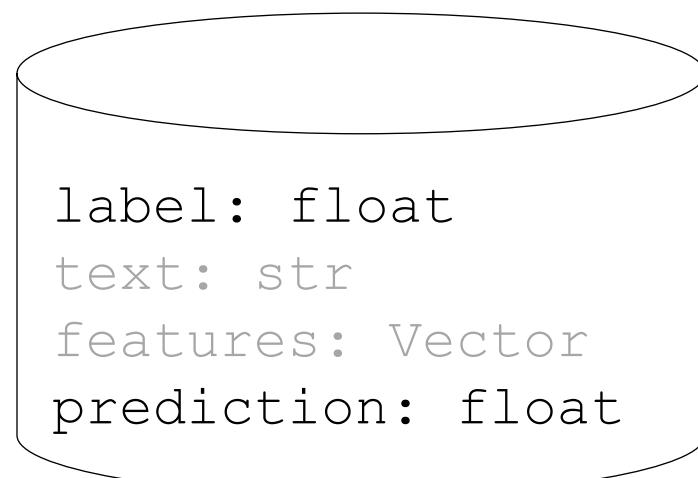


Huấn luyện mô hình

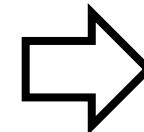


**Đánh giá**

```
def evaluate(self, dataset, params=None)  
    dataset: `pyspark.sql.DataFrame`  
    returns: metric
```



label: float  
text: str  
features: Vector  
prediction: float

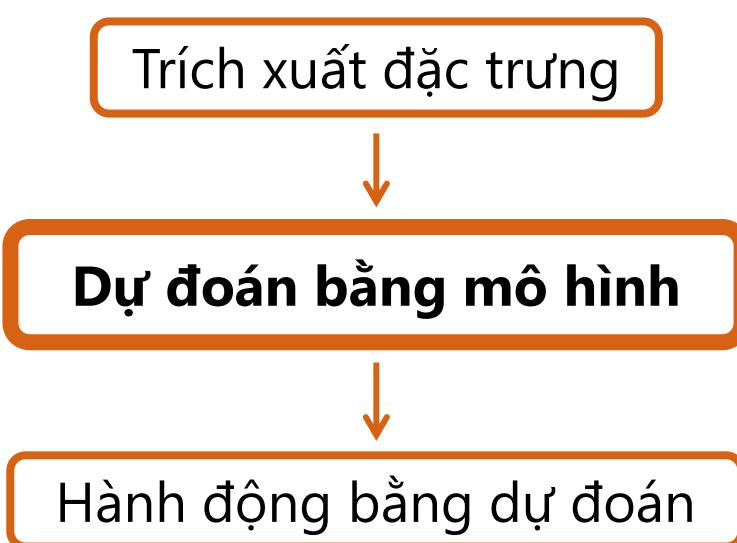


Đại lượng đánh giá:  
accuracy  
AUC  
MSE  
...

# Mô hình

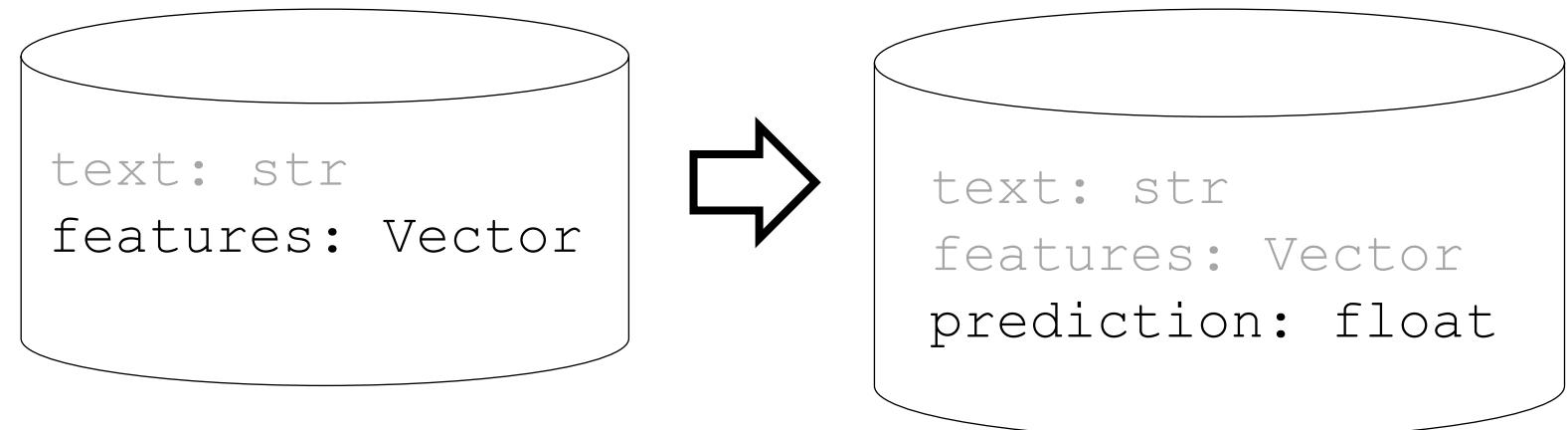
Model

Kiểm thử/Thành phẩm



Mô hình là một loại Biến đổi

```
def transform(self, dataset, params=None)
```



# Ước lượng (nhắc lại)

## Huấn luyện

Tải dữ liệu



Trích xuất đặc trưng

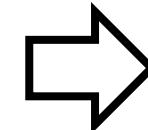
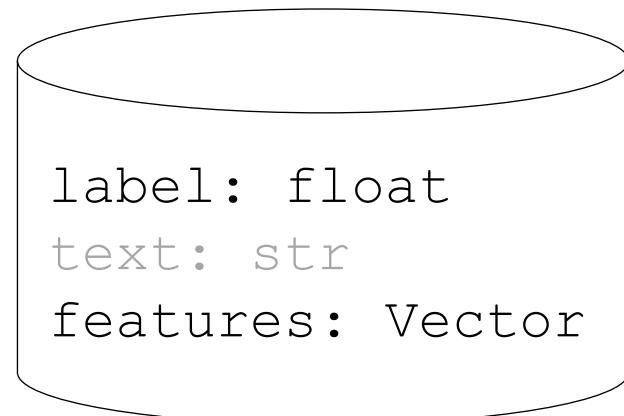


**Huấn luyện mô hình**



Đánh giá

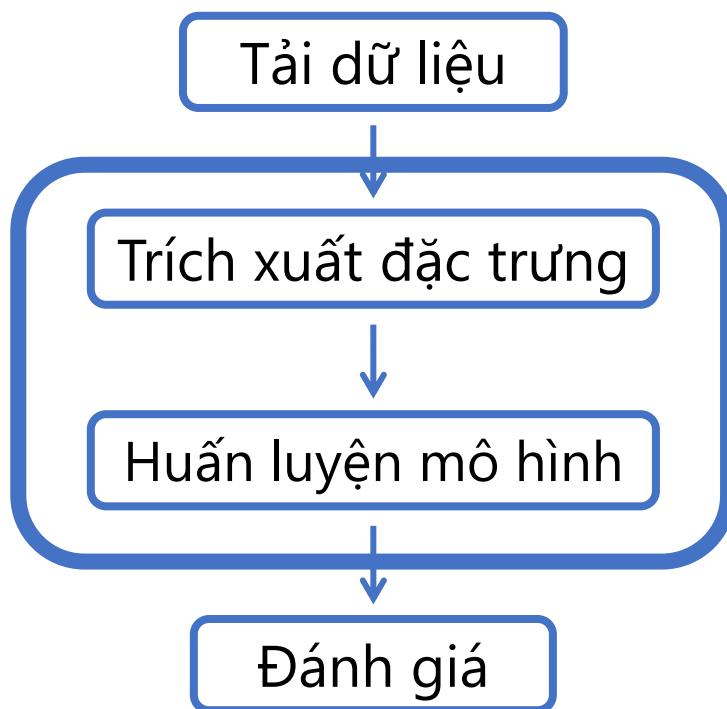
```
def fit(self, dataset, params=None)  
    dataset: `pyspark.sql.DataFrame`  
    returns: fitted model(s)
```



LogisticRegression  
Model

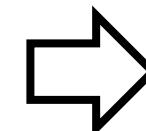
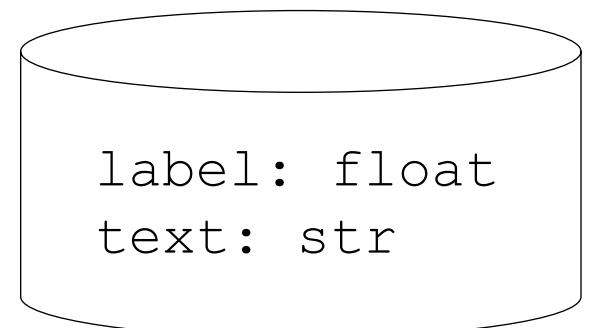
# Quy trình Pipeline

Huấn luyện



Quy trình là một loại Ước lượng

```
def fit(self, dataset, params=None)
```

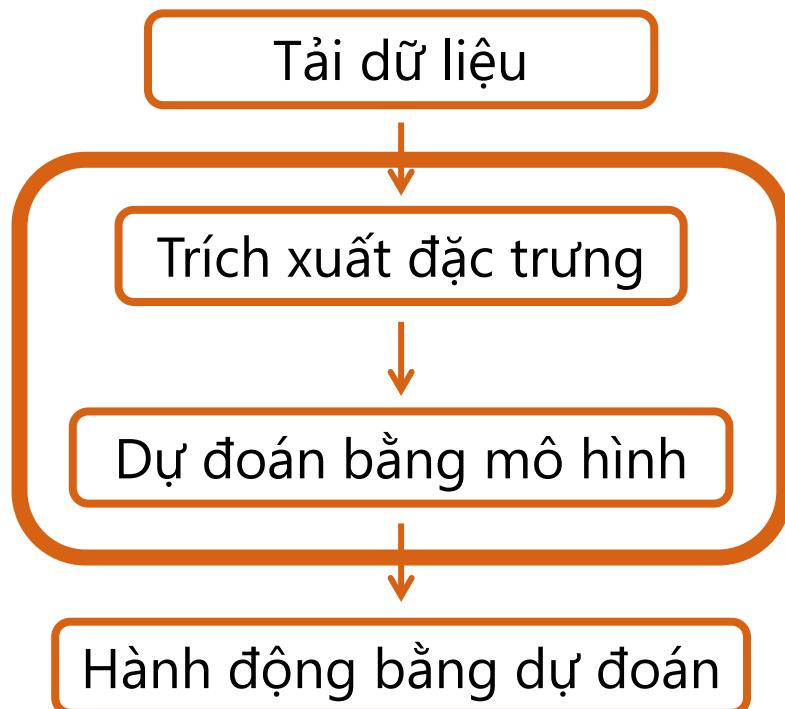


PipelineModel

# Mô hình quy trình

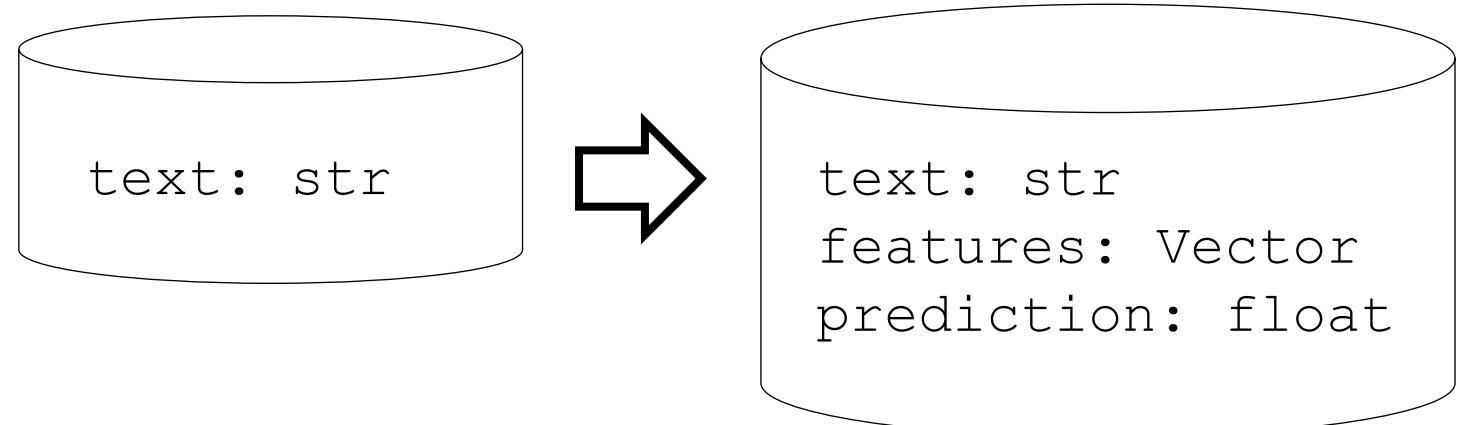
PipelineModel

Kiểm thử/Thành phẩm

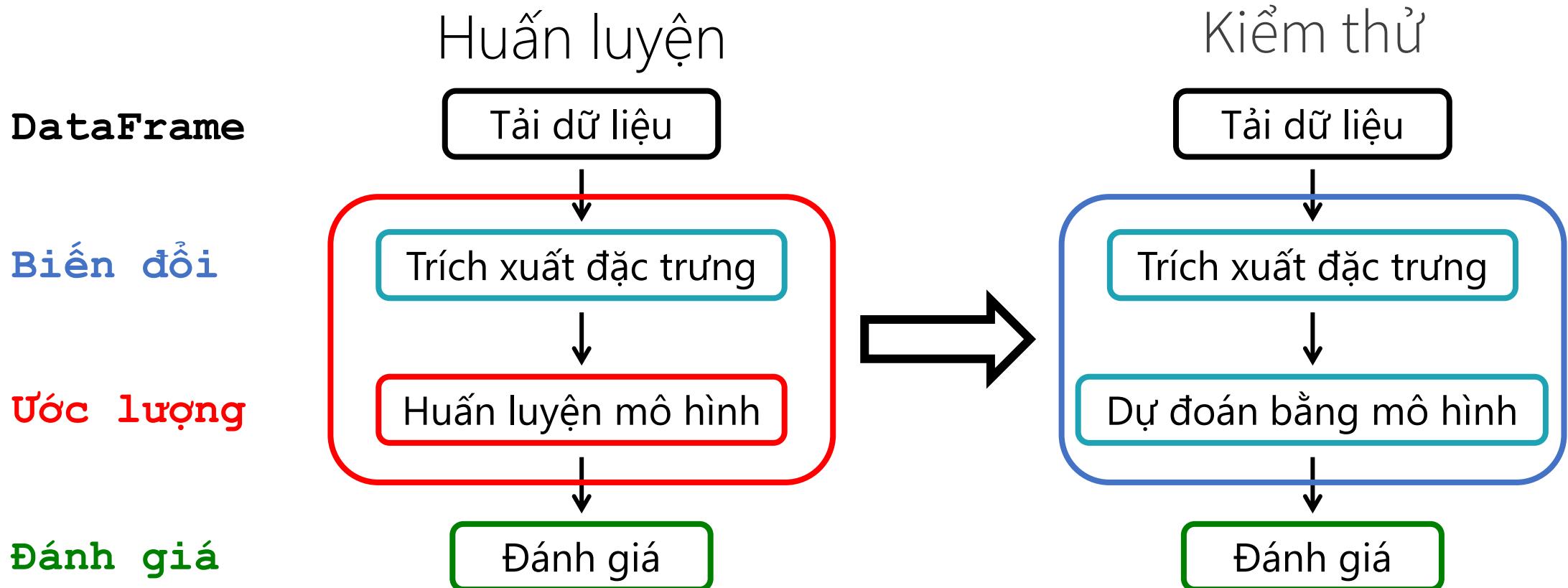


Mô hình quy trình là một loại Biến đổi

```
def transform(self, dataset, params=None)
```



# Tóm tắt



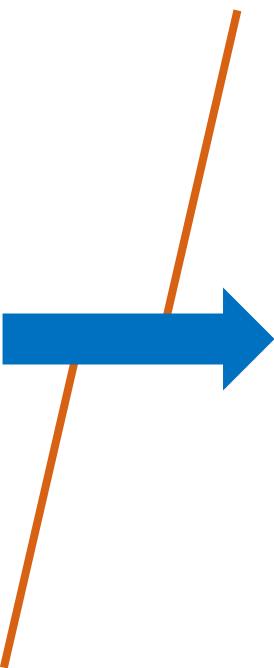
# Tại sao cần lưu trữ mô hình máy học?

Khoa học dữ liệu

Tạo nguyên mẫu (Python/R)  
Xây dựng mô hình

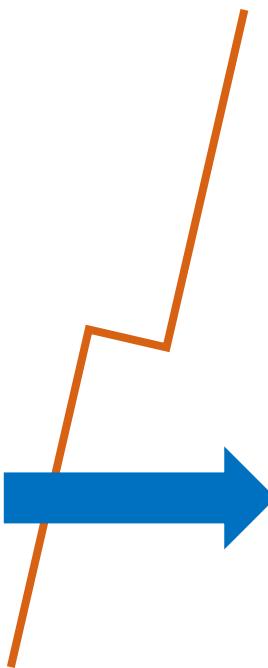
Kỹ thuật phần mềm

Xây dựng lại mô hình cho  
các sản phẩm (Java)  
Triển khai mô hình



## Khoa học dữ liệu

- Tạo nguyên mẫu (Python/R)
- Xây dựng quy trình
  - Trích xuất các đặc trưng thô
  - Biến đổi các đặc trưng
  - Chọn ra những đặc trưng quan trọng
  - Huấn luyện nhiều mô hình
  - Tổng hợp các kết quả để đưa ra dự đoán.



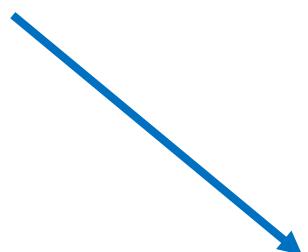
## Kỹ thuật phần mềm

- Xây dựng lại mô hình cho các sản phẩm (Java)
- Triển khai quy trình
  - Có thêm những việc phát sinh khi xây dựng sản phẩm
  - Quy trình lập trình khác nhau
  - Chi phí đồng bộ giữa hai bên
  - Cập nhật mô hình chậm

## Khoa học dữ liệu

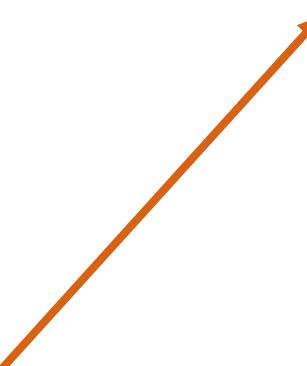
Tạo nguyên mẫu (Python/R)

Xây dựng quy trình



Lưu trữ mô hình hoặc quy trình

`model.save("path_to_save_model")`



Tải quy trình (Scala/Java)

`Model.load("path_to_model")`

Triển khai trong sản phẩm

- Siêu dữ liệu lưu trữ bằng JSON
- Dữ liệu mô hình lưu bằng Parquet

## Kỹ thuật phần mềm

# Ví dụ: k-Means

```
# Đọc dữ liệu
dataset = spark.read.format("libsvm").\
    load("data/mllib/sample_kmeans_data.txt")

# Huấn luyện mô hình k-means
kmeans = KMeans().setK(2).setSeed(1)
model = kmeans.fit(dataset)

# Dự đoán
predictions = model.transform(dataset)

# Đánh giá gom cụm bằng độ đo Silhouette
evaluator = ClusteringEvaluator()
silhouette = evaluator.evaluate(predictions)
```

# Ví dụ: quy trình

```
# Dữ liệu huấn luyện: list của (id, text, label) tuples
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Tạo quy trình gồm 3 bước: tokenizer, hashingTF, và lr
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), \
                      outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

```
# Huấn luyện quy trình
model = pipeline.fit(training)

# Lưu lại quy trình
model.write.save("pipeline-model")

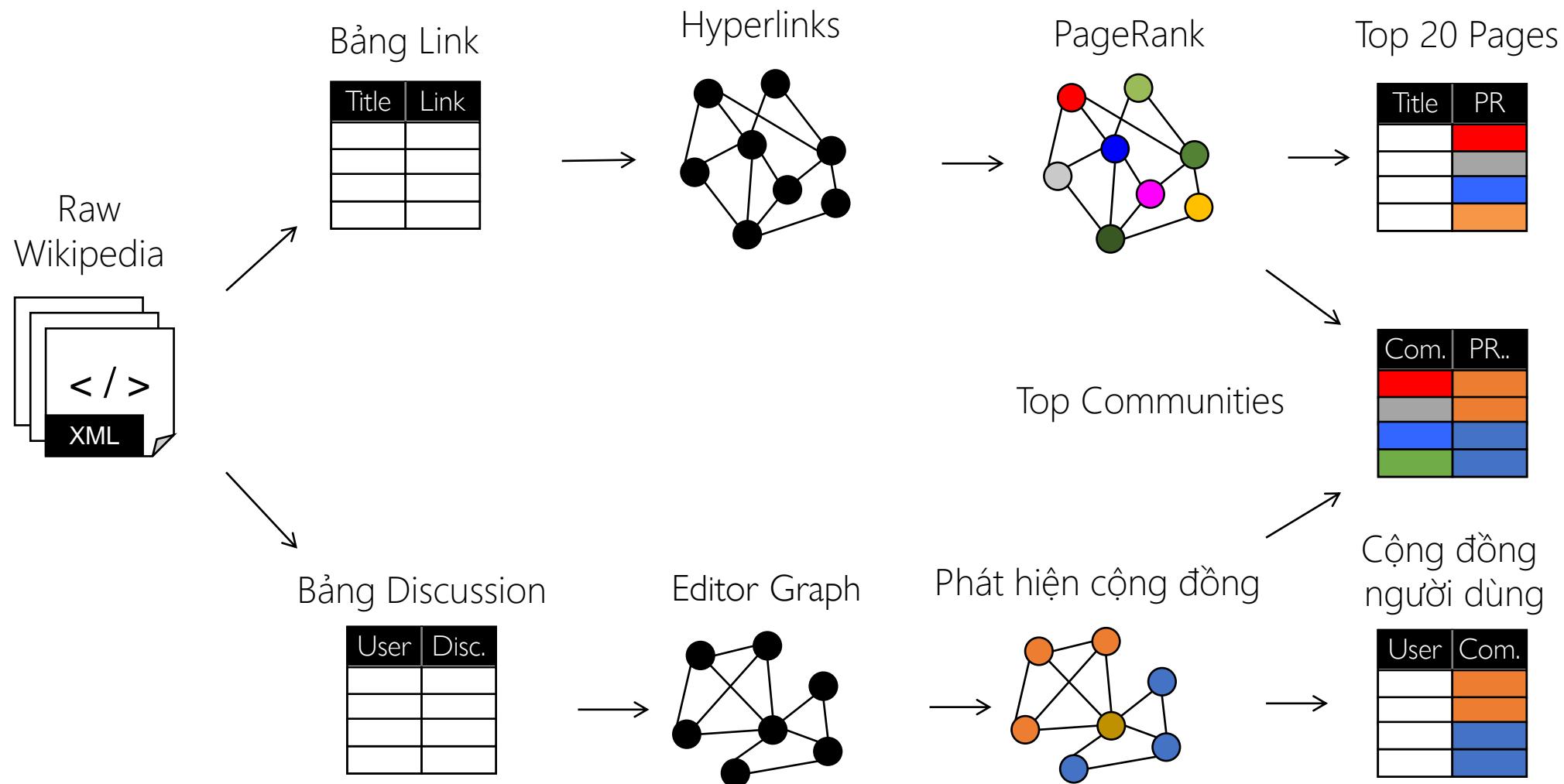
# Dữ liệu kiểm thử: (id, text) tuples chưa được gán nhãn
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])
```

```
# Dự đoán trên dữ liệu kiểm thử và in ra các thông tin mong muốn
prediction = model.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s, prediction=%f" % \
          (rid, text, str(prob), prediction))
```

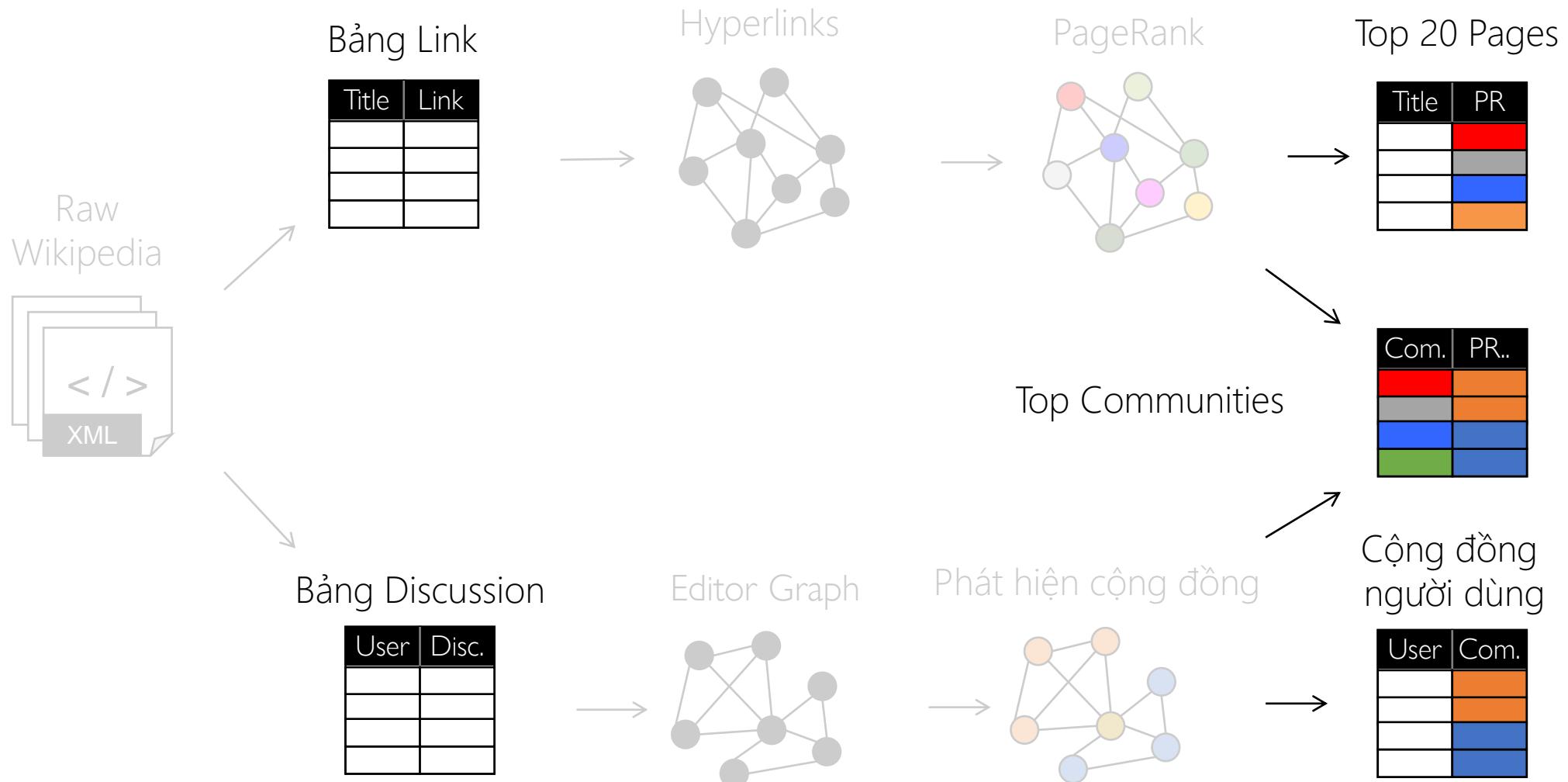
# APACHE **Spark** GRAPHX



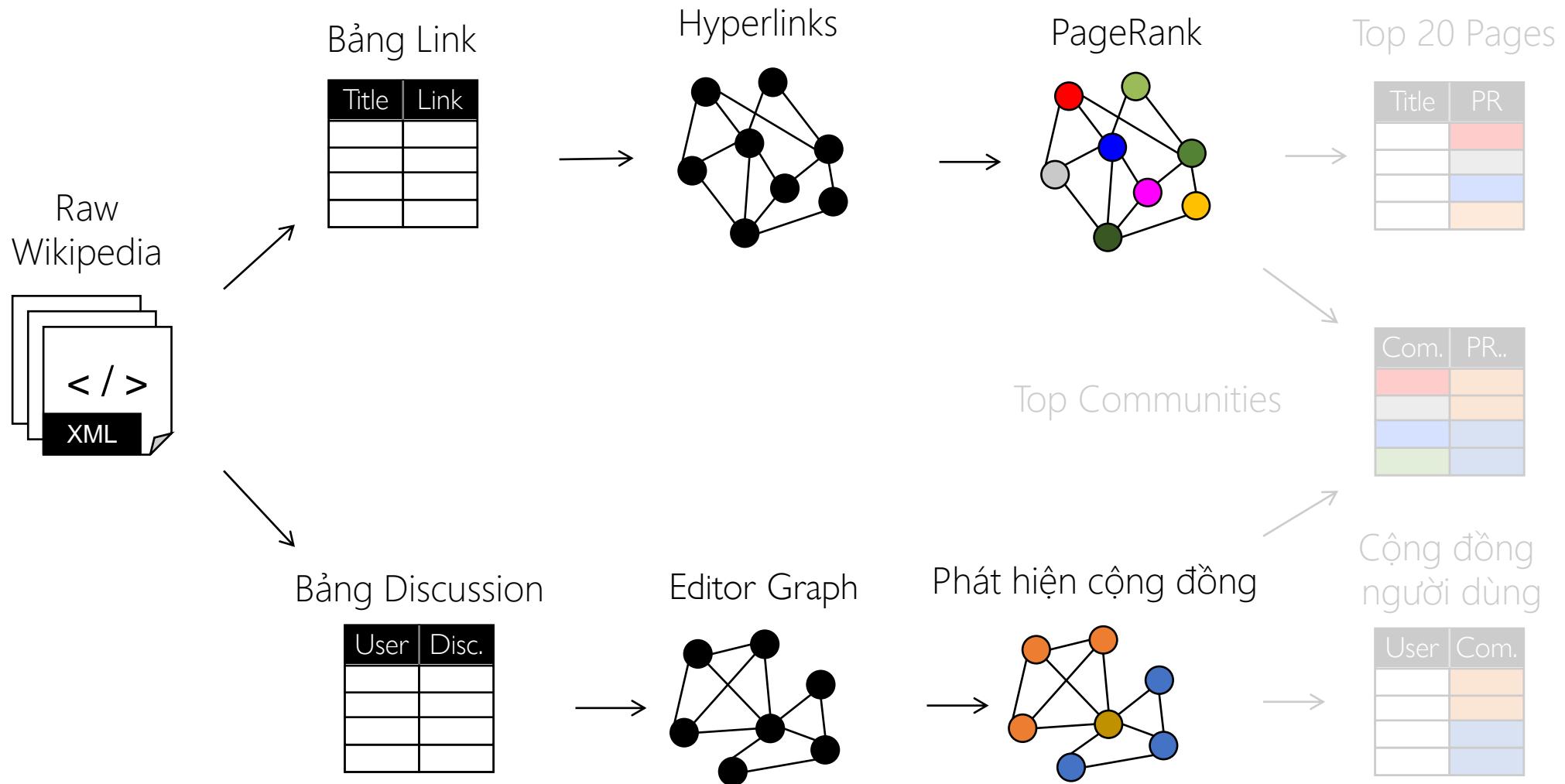
# Bài toán phân tích hiện đại



# Bảng



# Đồ thị

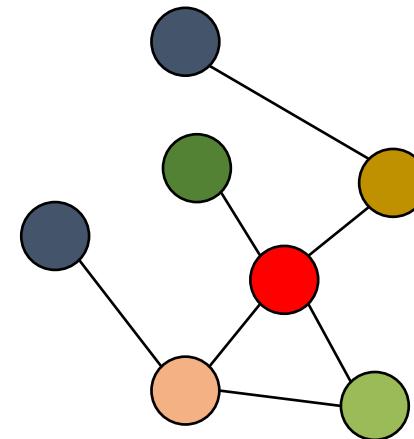


# Những hệ thống riêng biệt

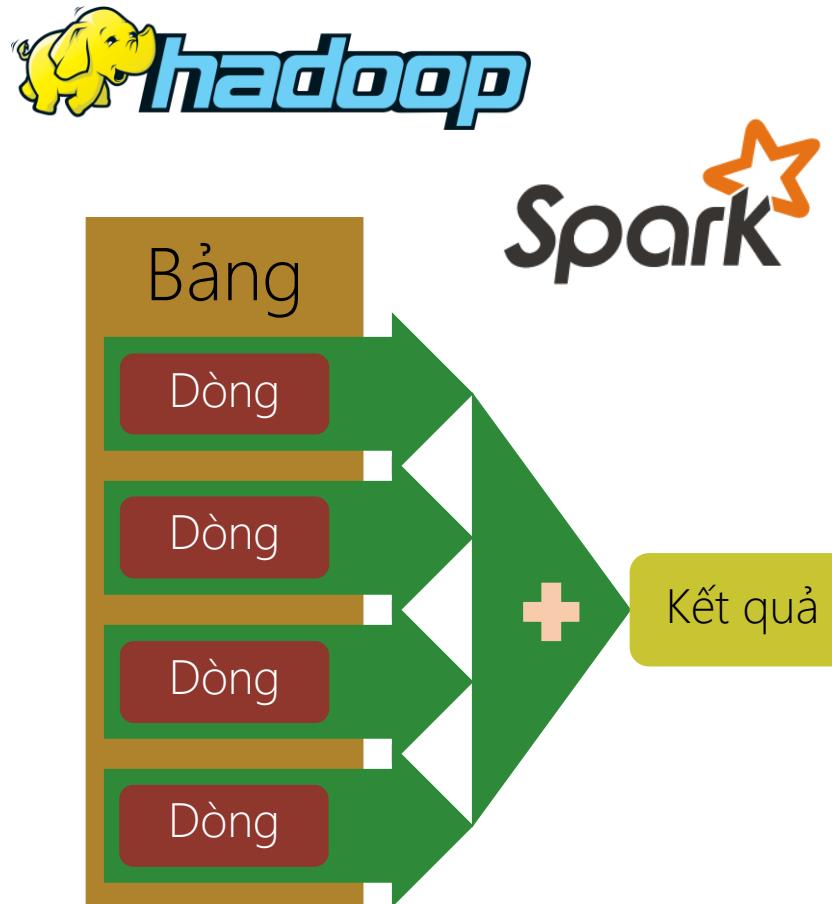
Bảng

		1
1	2	3

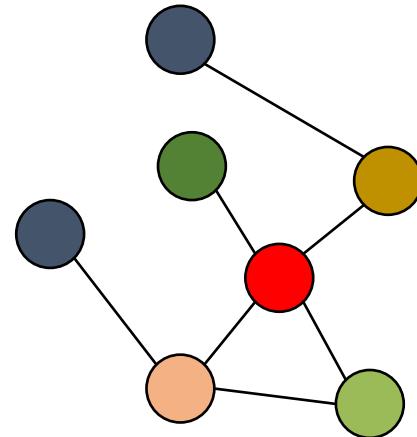
Đồ thị



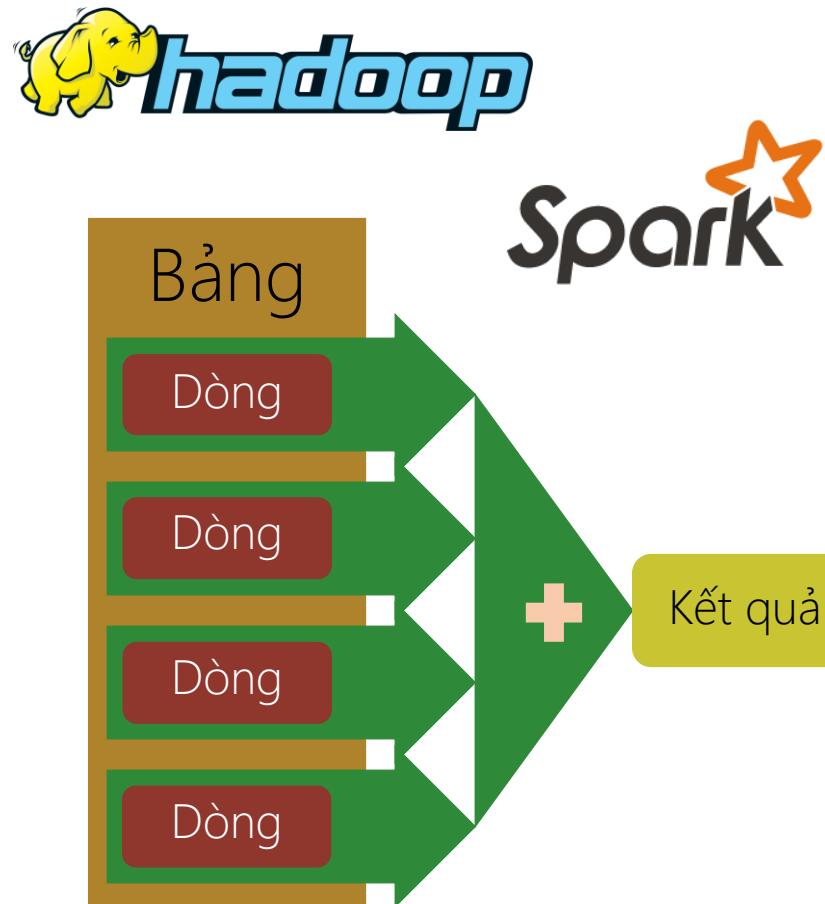
# Hệ thống theo mô hình luồng dữ liệu



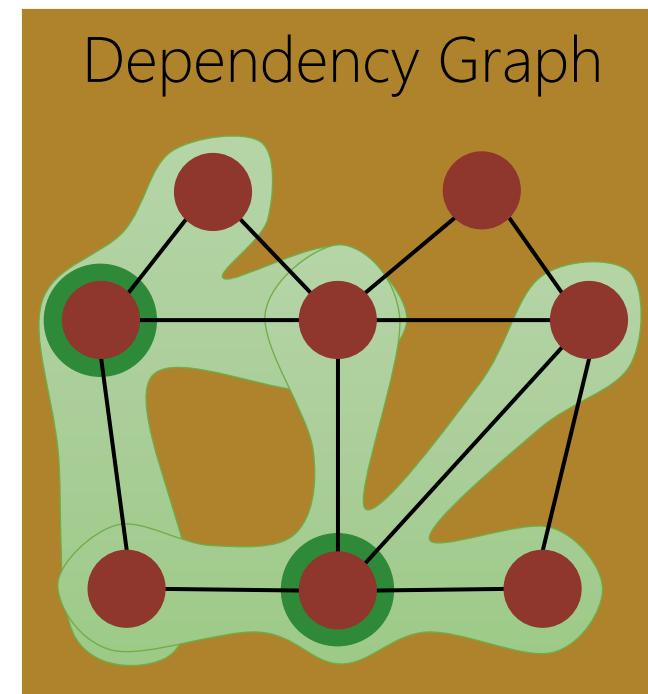
Đồ thị



## Hệ thống theo mô hình luồng dữ liệu

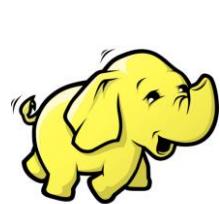


## Hệ thống theo mô hình đồ thị



# Khó sử dụng

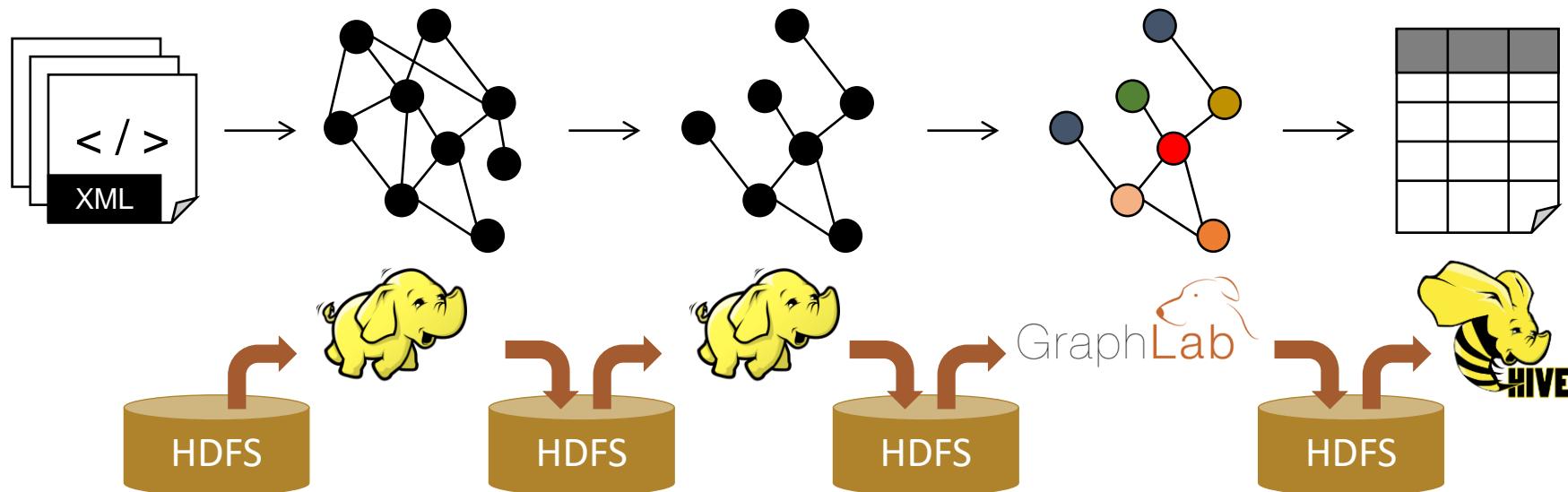
Người dùng phải **Tìm hiểu, Triển khai và Quản lý**  
nhiều hệ thống cùng lúc



Giao tiếp giữa các hệ thống thường yếu và phức tạp

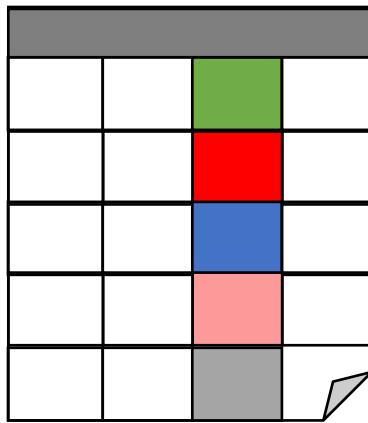
# Không hiệu quả

Di chuyển và sao chép dữ liệu diện rộng trên toàn bộ mạng và hệ thống tập tin

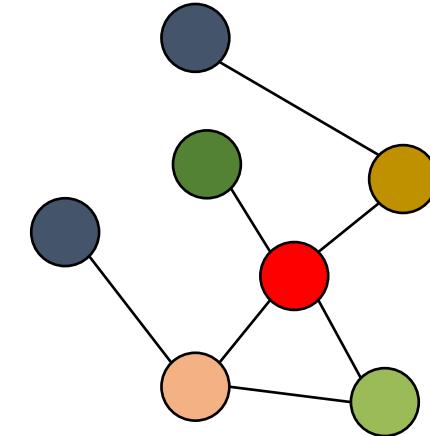
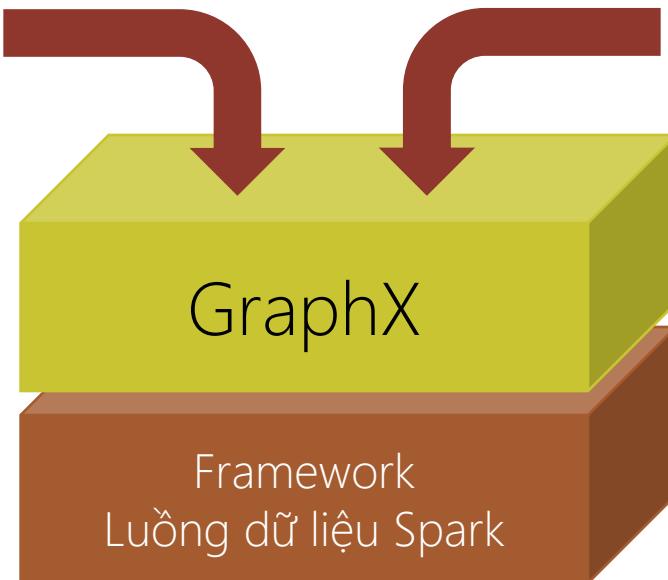


Ít sử dụng lại các cấu trúc dữ liệu nội bộ qua các bước

# Thống nhất tính toán



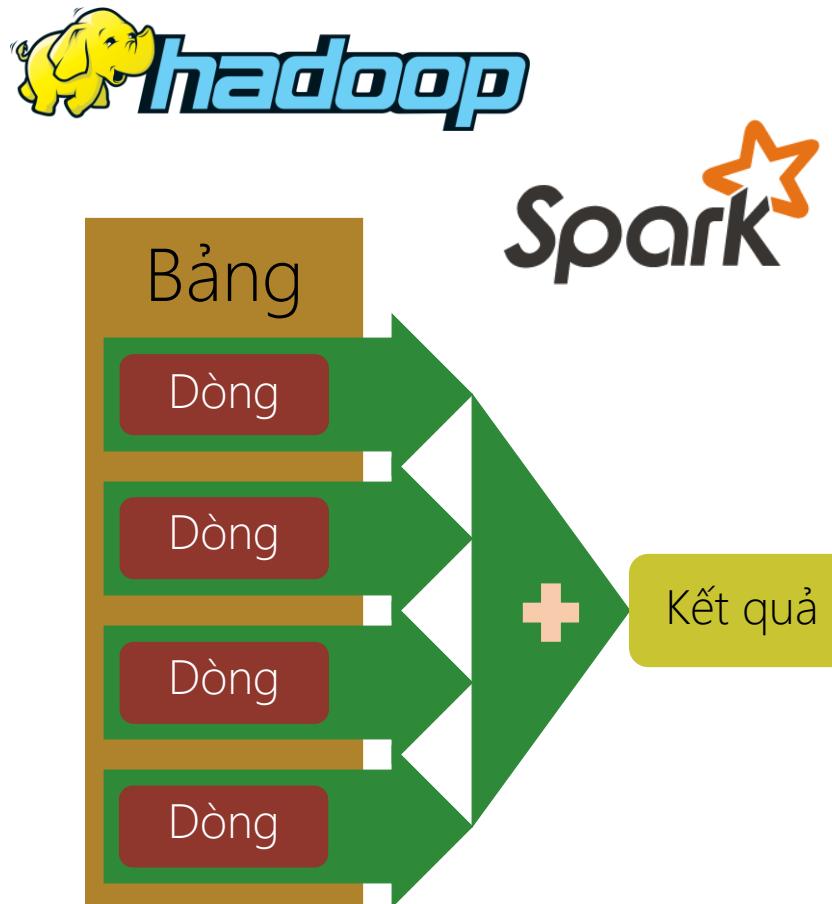
Thể hiện  
dạng bảng



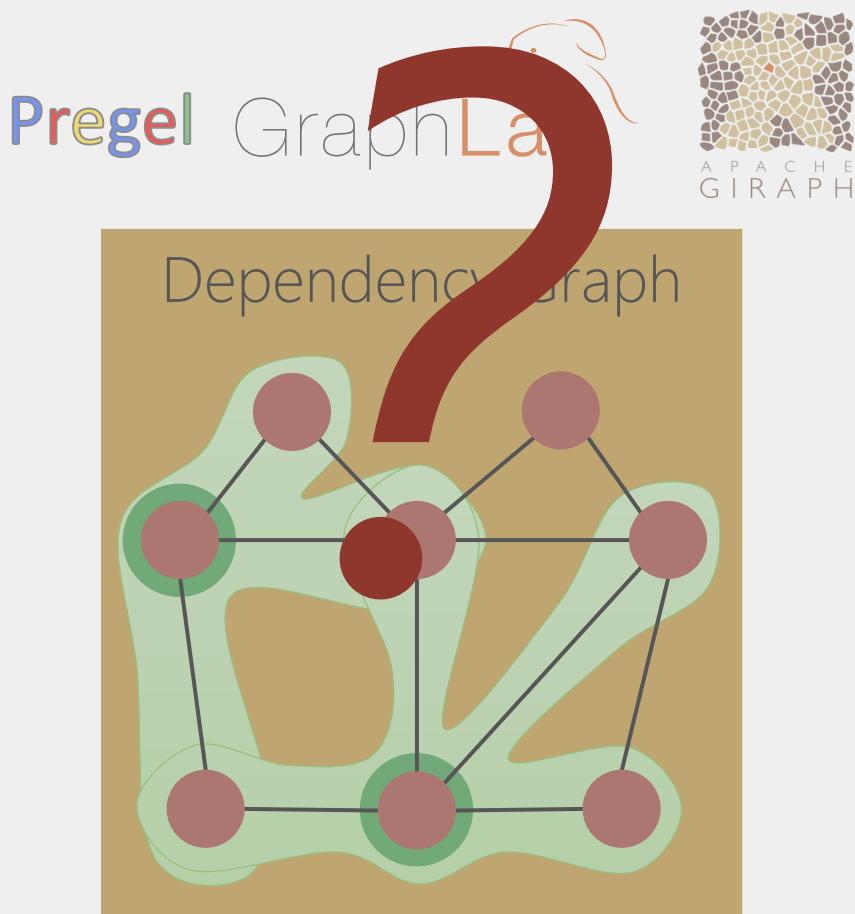
Thể hiện dạng  
đồ thị

Sử dụng một hệ thống duy nhất để xử lý toàn bộ quy trình  
một cách **dễ dàng** và **hiệu quả**

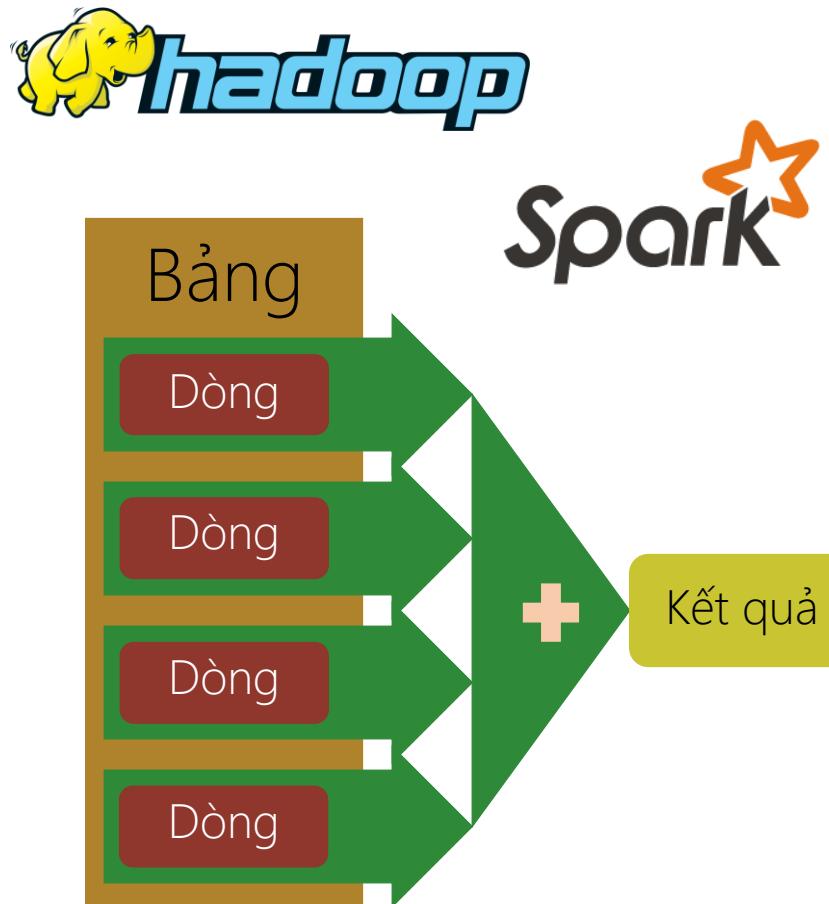
## Hệ thống theo mô hình luồng dữ liệu



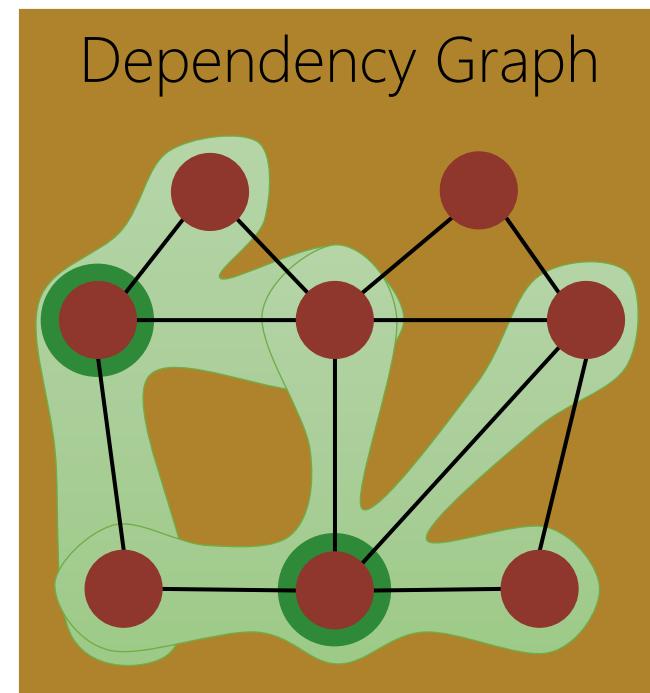
## Hệ thống theo mô hình đồ thị



## Hệ thống theo mô hình luồng dữ liệu



## Hệ thống theo mô hình đồ thị



## Vấn đề...

Làm cách nào để **biểu diễn một cách tự nhiên** và **xử lý hiệu quả** tính toán đồ thị trong một framework chung theo mô hình luồng dữ liệu?

Chắt lọc những bài học kinh nghiệm từ các hệ thống xử lý đồ thị chuyên biệt

## Vấn đề...

Làm cách nào để **biểu diễn** một cách tự nhiên và **xử lý hiệu quả** tính toán đồ thị trong một framework chung theo mô hình luồng dữ liệu?

**BIỂU DIỄN**

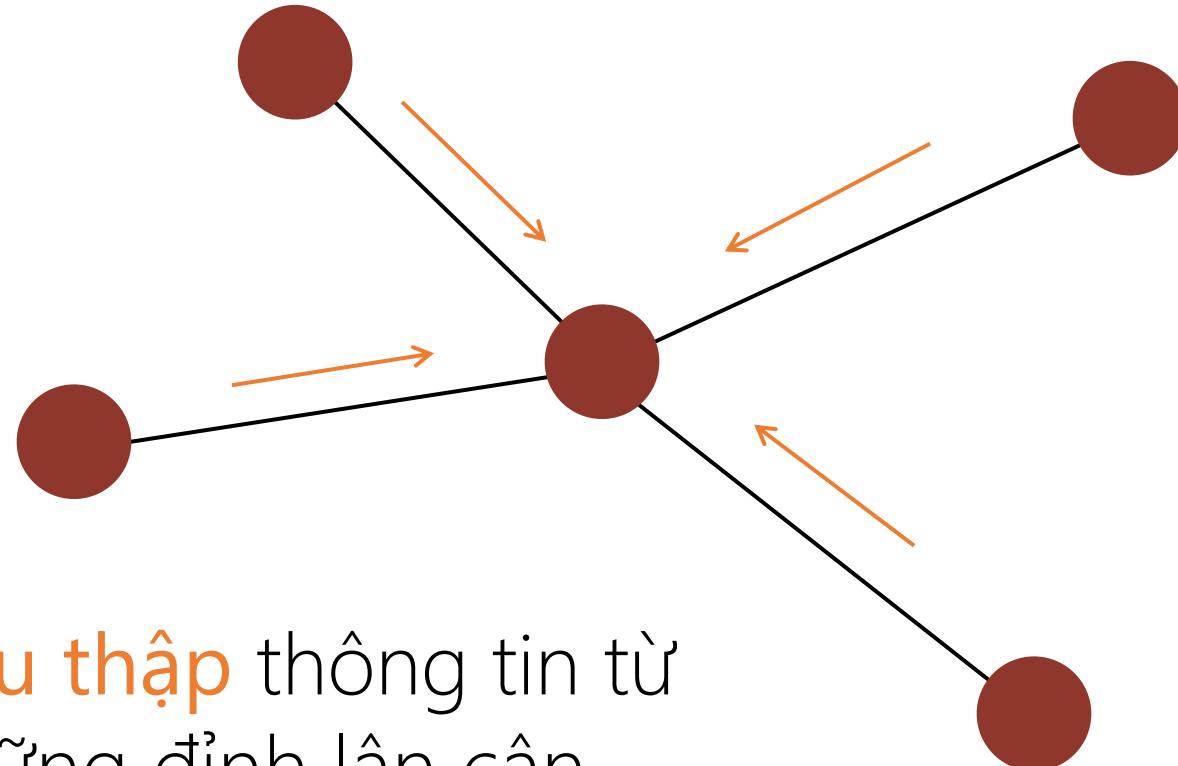
**TỐI ƯU HÓA**

*"Think like a Vertex."*

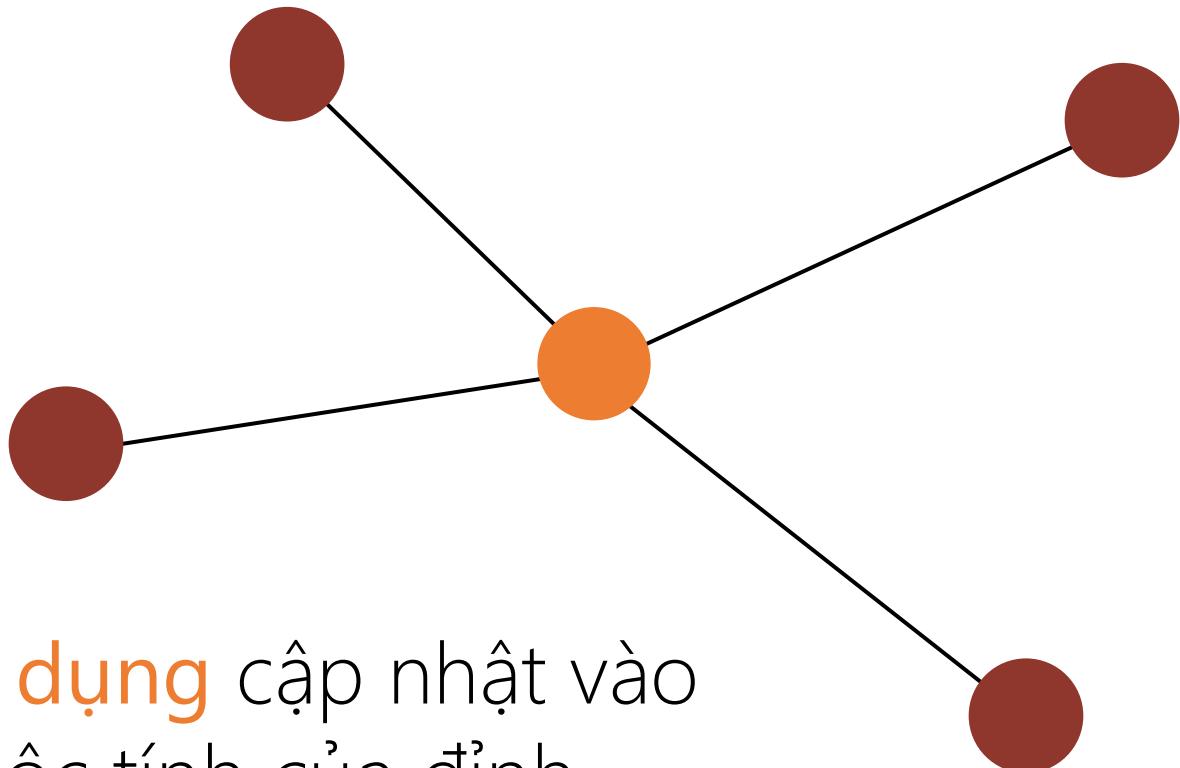
- Malewicz et al., SIGMOD'10

# Mô hình đồ thị song song

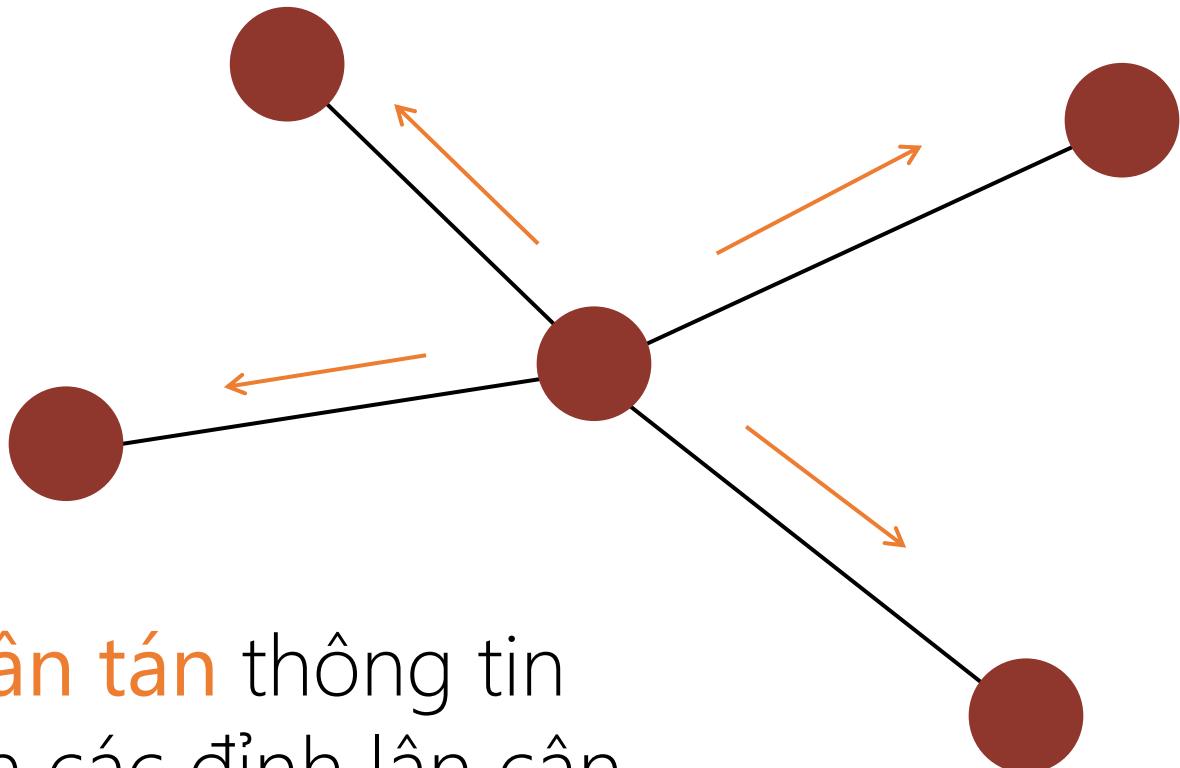
Gonzalez et al. [OSDI'12]



Thu thập thông tin từ  
những đỉnh lân cận



Áp dụng cập nhật vào  
thuộc tính của đỉnh



Phân tán thông tin  
đến các đỉnh lân cận

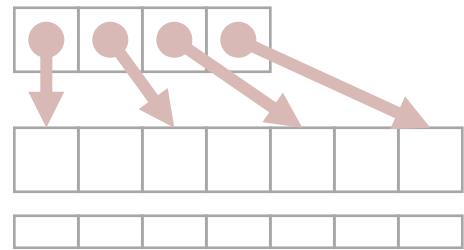
Mô hình  
tính toán  
chuyên biệt



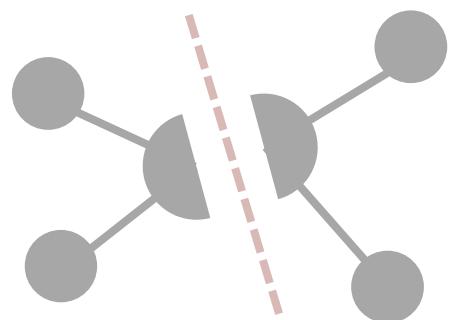
Tối ưu hóa  
đồ thị  
chuyên biệt

# Tối ưu hóa hệ thống đồ thị

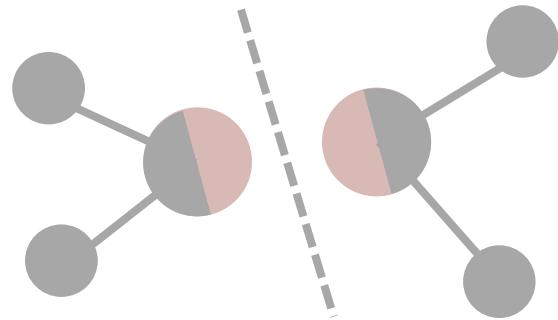
Cấu trúc dữ  
liệu chuyên biệt



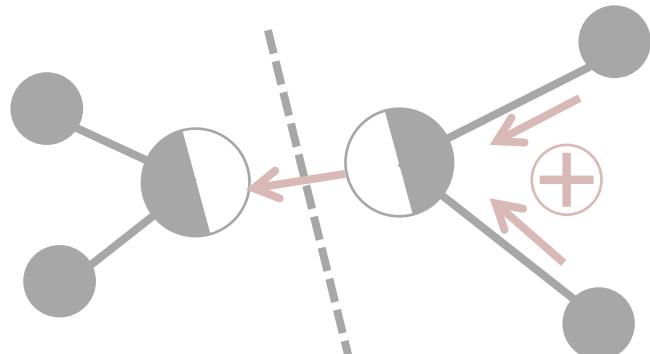
Phân chia theo  
thao tác cắt đỉnh



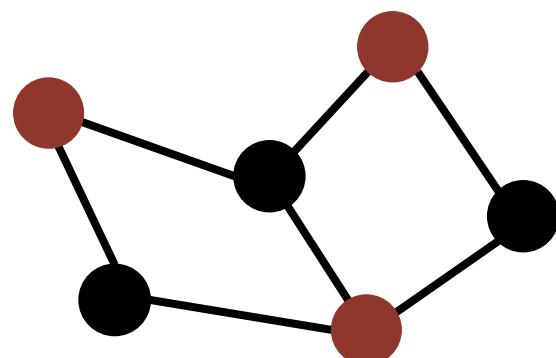
Bộ nhớ đệm/Phản chiếu  
từ xa



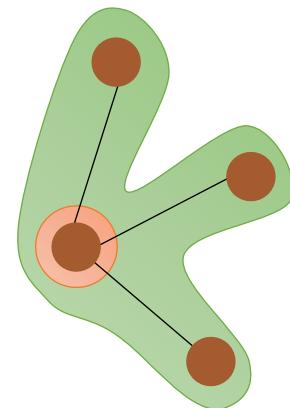
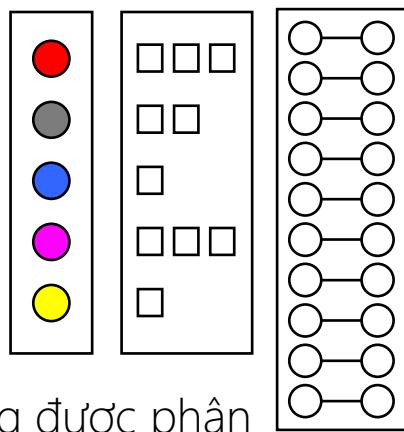
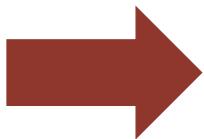
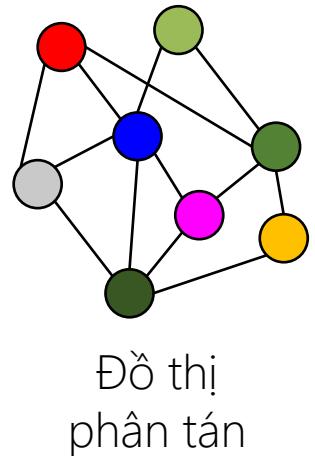
Tổng hợp thông điệp



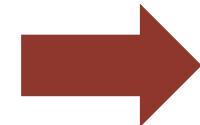
Theo dõi thành  
phần hoạt động



# Biểu diễn



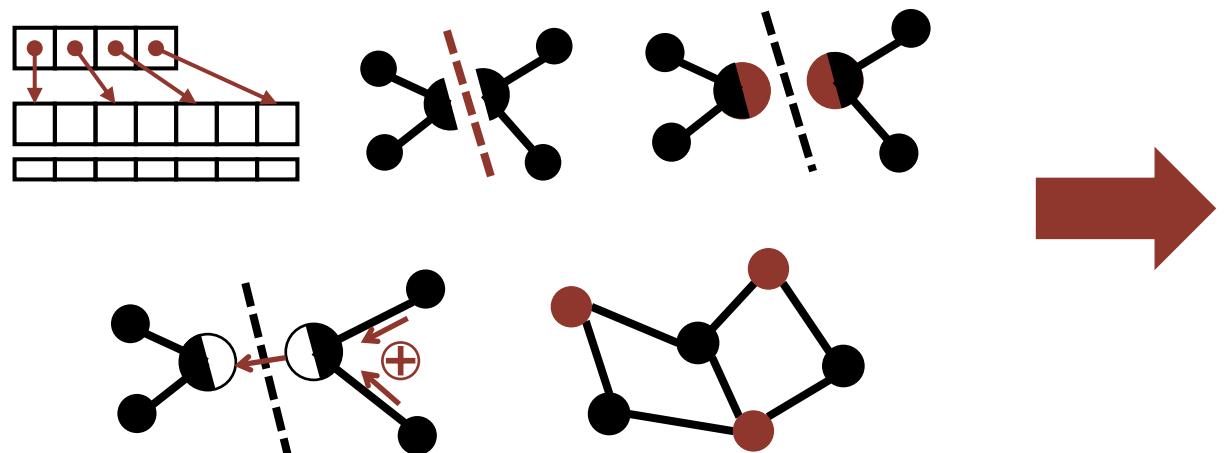
Chương trình chạy trên các đỉnh của đồ thị



Kết  
Các toán tử trên luồng dữ liệu

# Tối ưu hóa

Hệ thống xử lý đồ thị nâng cao

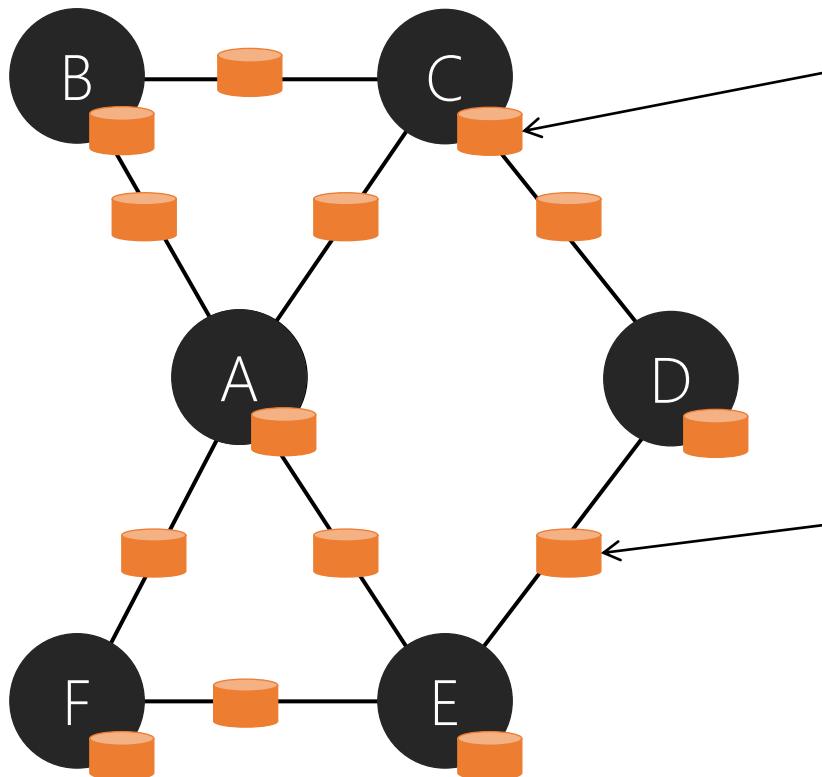


Tối ưu hóa phép kết phân tán  
Bảo toàn biểu diễn đồ thị cụ thể

# Đồ thị thuộc tính

## Property Graph

Đồ thị thuộc tính



Thuộc tính của đỉnh:

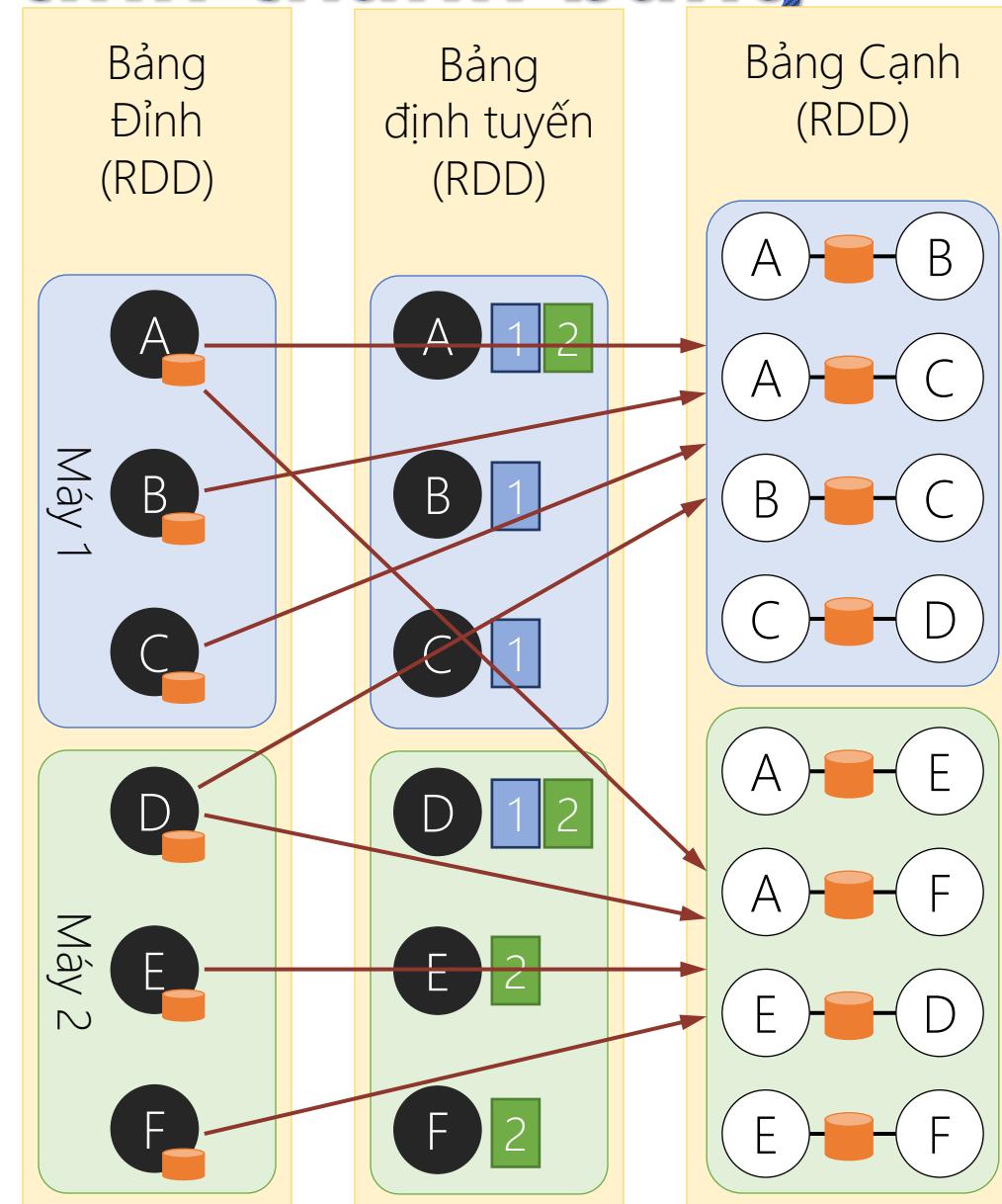
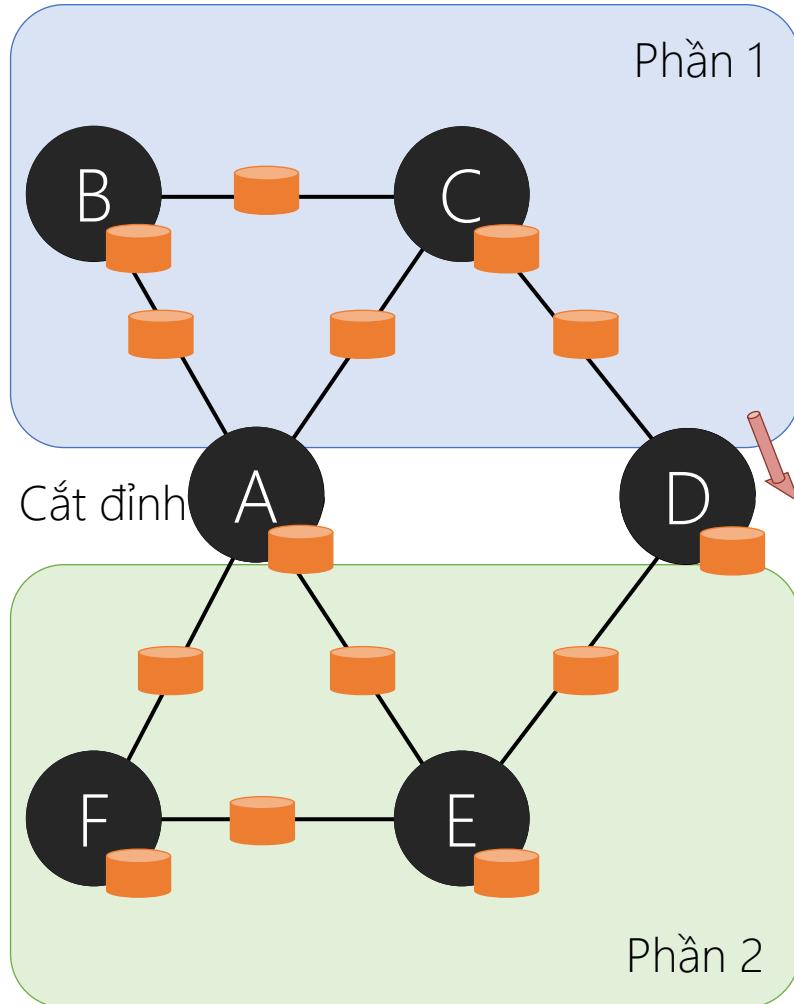
- Thông tin của đỉnh
- Giá trị PageRank hiện tại

Thuộc tính của cạnh

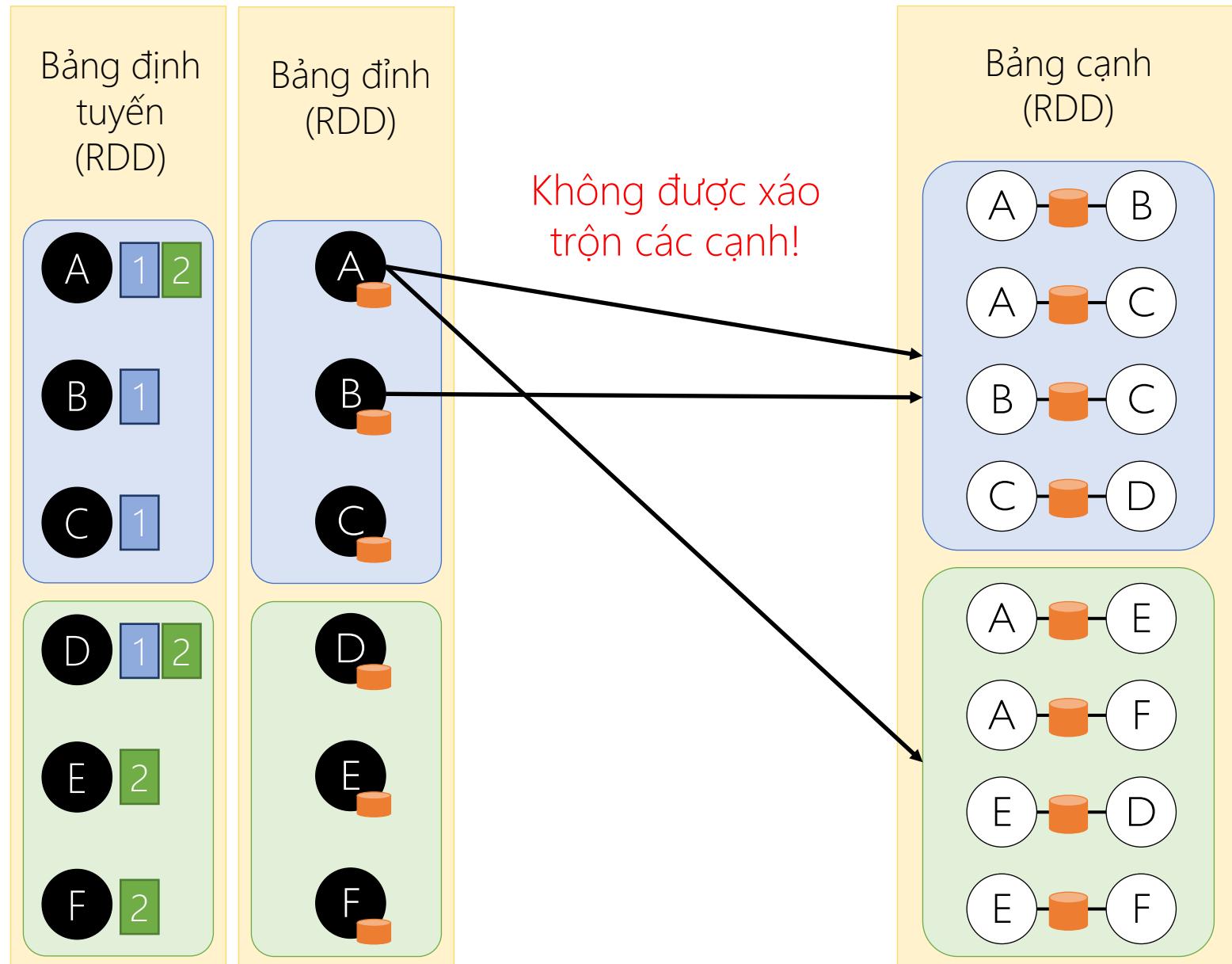
- Trọng số cạnh
- Nhãn thời gian

# Mã hóa đồ thị thuộc tính thành bảng

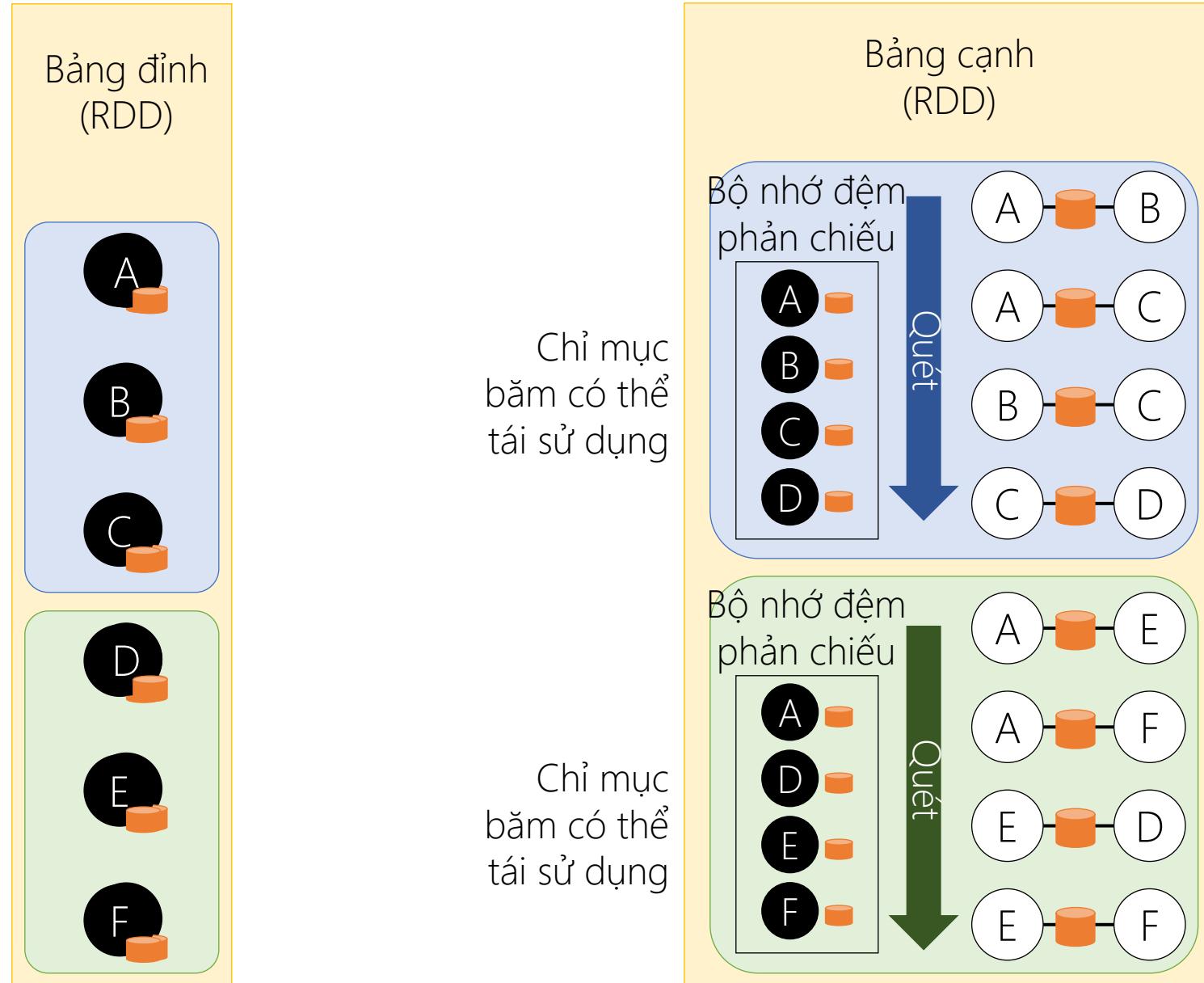
Đồ thị thuộc tính



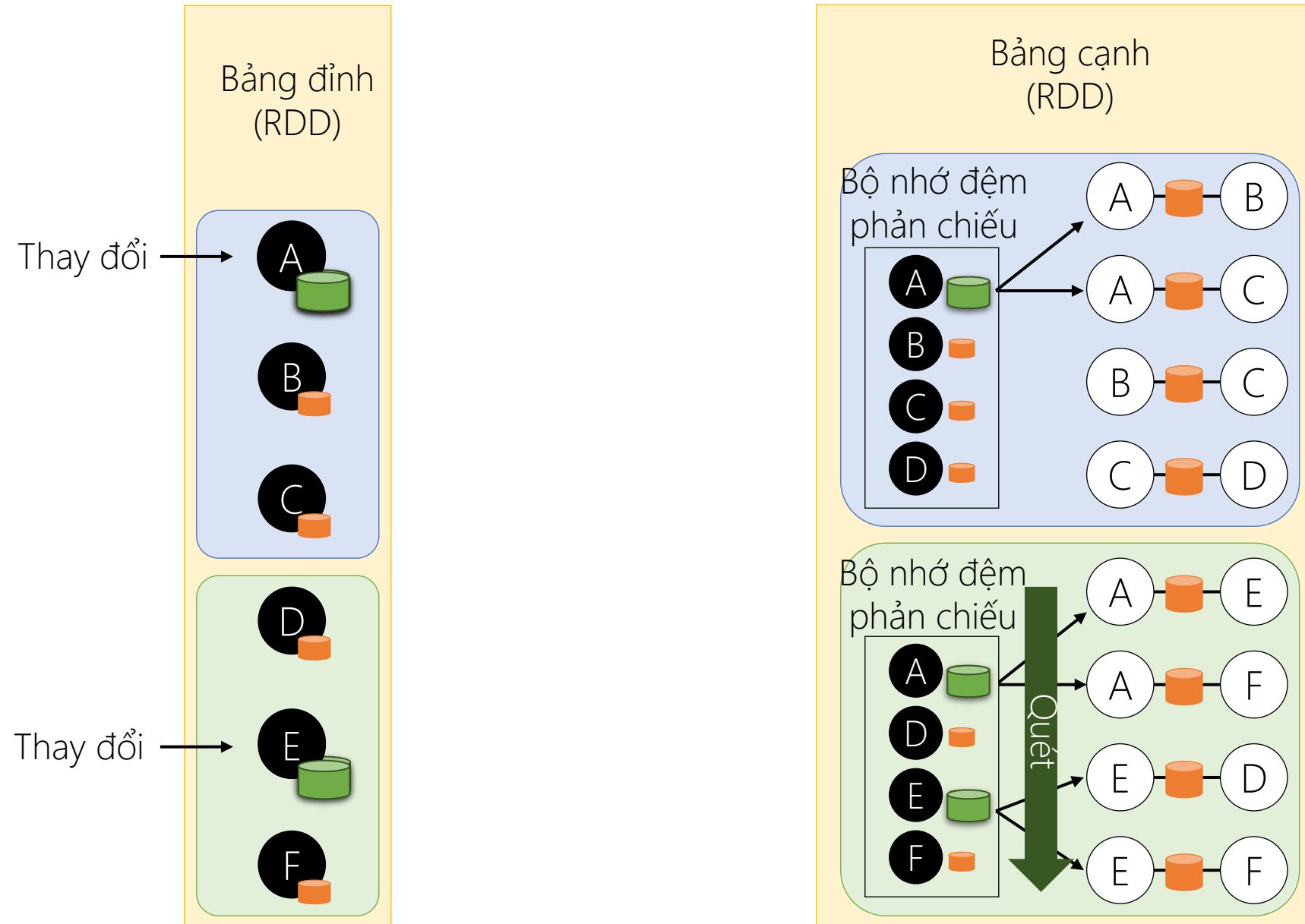
Sử dụng  
bảng định  
tuyến để  
xác định  
vị trí



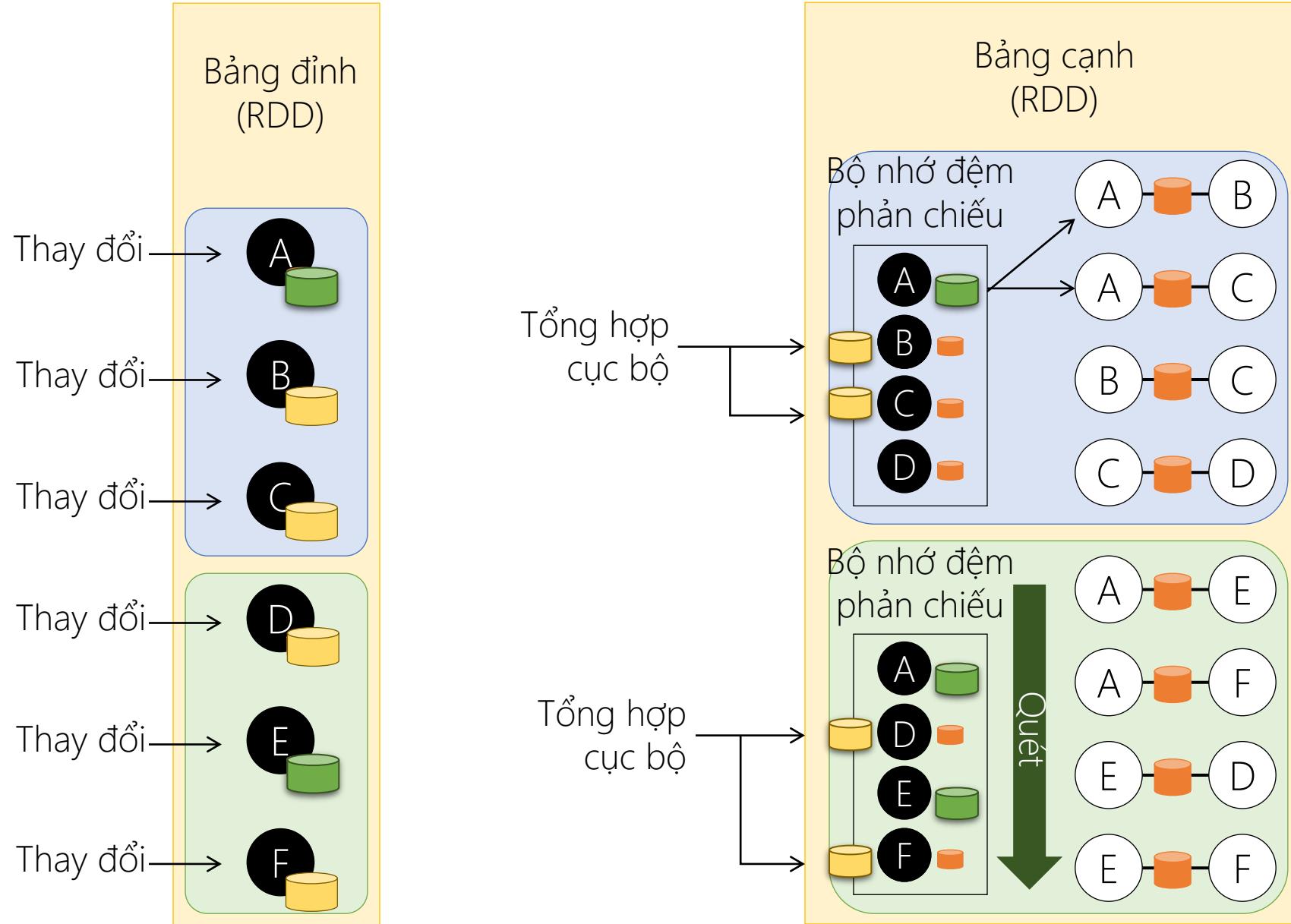
# Sử dụng bộ nhớ đệm cho các thao tác lặp lại trên bộ ba



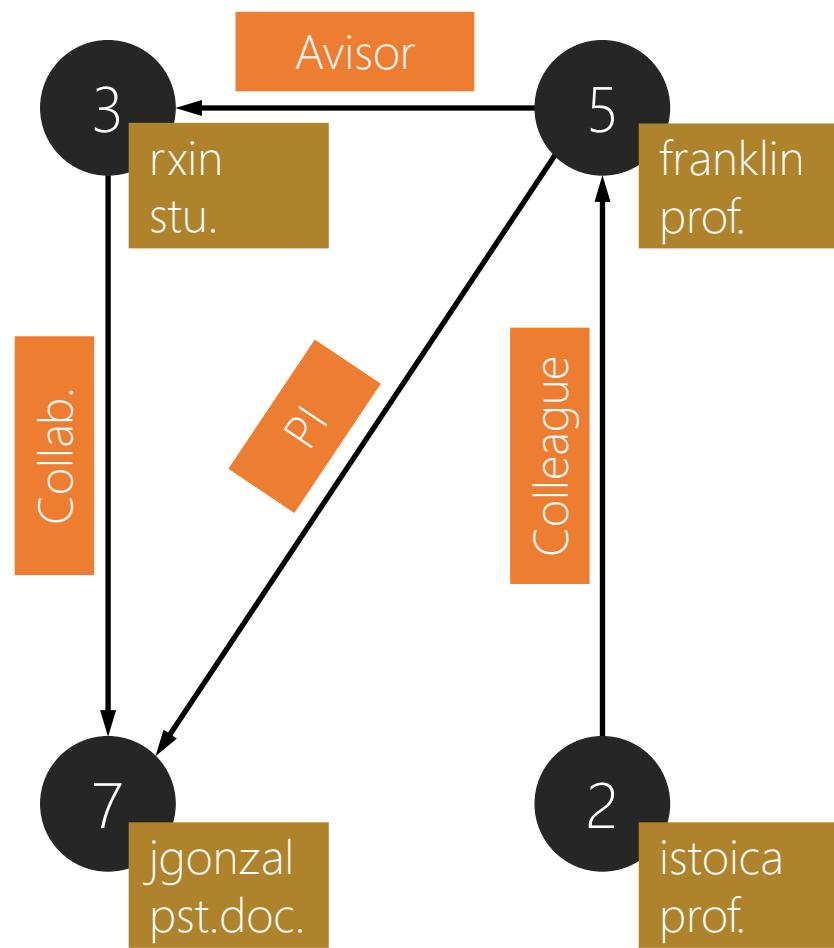
# Thực hiện cập nhật



Nếu có  
thao  
tác  
tổng  
hợp  
được  
thực  
hiện  
thì...



## Đồ thị thuộc tính



## Bảng định

<b>id</b>	<b>Thuộc tính (V)</b>
<b>3</b>	(rxin, student)
<b>7</b>	(jgonzal, postdoc)
<b>5</b>	(franklin, professor)
<b>2</b>	(istoica, professor)

## Bảng cạnh

<b>SrcId</b>	<b>DstId</b>	<b>Thuộc tính (E)</b>
<b>3</b>	<b>7</b>	Collaborator
<b>5</b>	<b>3</b>	Advisor
<b>2</b>	<b>5</b>	Colleague
<b>5</b>	<b>7</b>	PI

```
// Tạo một RDD chứa các đỉnh
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Seq((3L, ("rxin", "student")),
    (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")),
    (2L, ("istoica", "prof"))))

// Tạo một RDD cho các cạnh
val relationships: RDD[Edge[String]] =
  sc.parallelize(Seq(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),
    Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// Khai báo user mặc định trong trường hợp có cạnh nối với user bị thiếu
val defaultUser = ("John Doe", "Missing")

// Tạo đồ thị ban đầu
val graph = Graph(users, relationships, defaultUser)
```

# Các toán tử trên bảng

Các toán tử trên bảng được kế thừa từ Spark

map	reduce	sample
filter	count	take
groupBy	fold	first
sort	reduceByKey	partitionBy
union	groupByKey	mapWith
join	cogroup	pipe
leftOuterJoin	cross	save
rightOuterJoin	zip	...

# Các tôán tử trên đồ thị

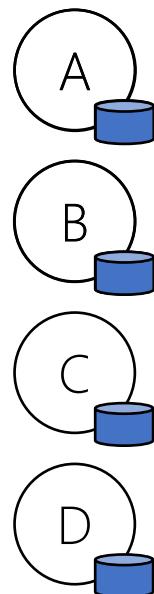
```
class Graph[VD, ED] {  
    // Thông tin về đồ thị  
    val numEdges: Long  
    val numVertices: Long  
    val inDegrees: VertexRDD[Int]  
    val outDegrees: VertexRDD[Int]  
    val degrees: VertexRDD[Int]  
    // Các thành phần biểu diễn đồ thị  
    val vertices: VertexRDD[VD]  
    val edges: EdgeRDD[ED]  
    val triplets: RDD[EdgeTriplet[VD, ED]]  
}
```

# Bộ ba

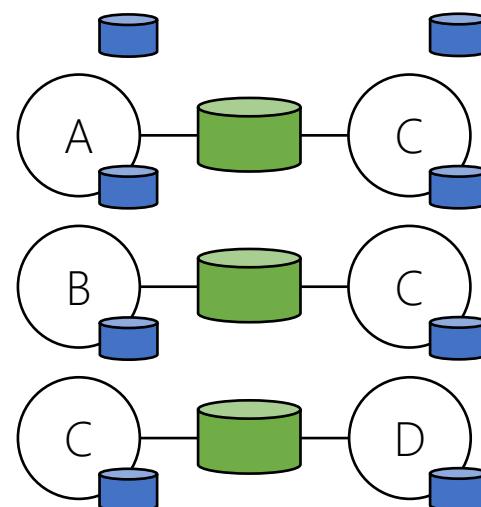
## Triplet

Là một cách biểu diễn dữ liệu được tạo ra bằng toán tử nối các đỉnh và cạnh

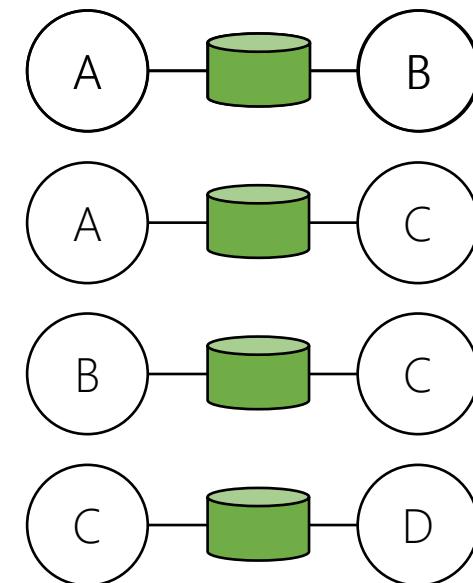
Đỉnh



Bộ ba



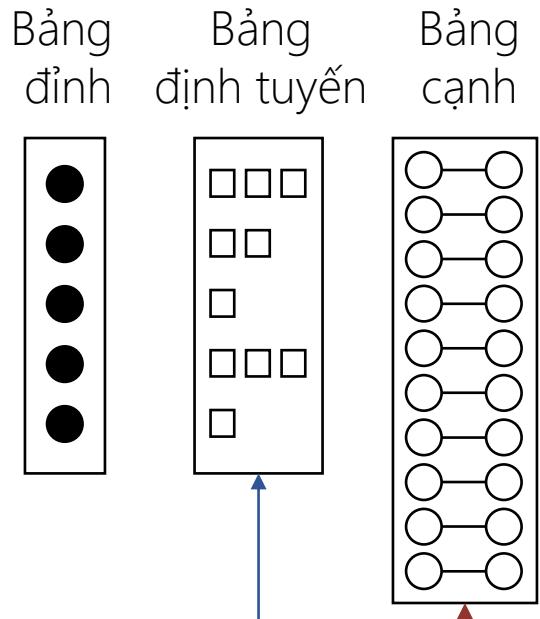
Cạnh



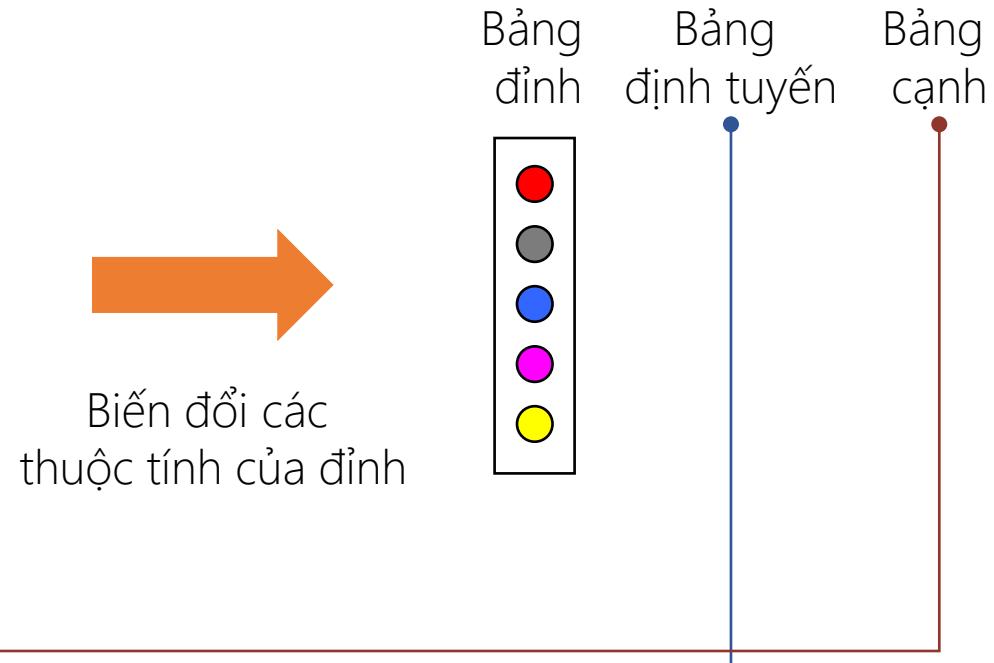
# Tách biệt giữa thuộc tính và cấu trúc

Có thể *tái sử dụng* thông tin cấu trúc trên nhiều biểu đồ

Đồ thị đầu vào



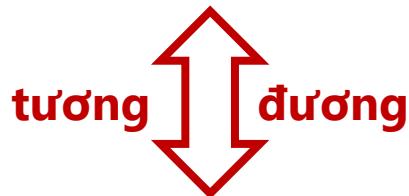
Đồ thị đã biến đổi



# Biến đổi thuộc tính

```
class Graph[VD, ED] {  
    // Biến đổi các thuộc tính của đỉnh và cạnh  
    def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]  
    def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]  
    def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]))  
        => Iterator[ED2]): Graph[VD, ED2]  
    def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]  
    def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]))  
        => Iterator[ED2]): Graph[VD, ED2]  
}
```

```
val newVertices = graph.vertices.map { case (id, attr) => (id, mapUdf(id, attr)) }  
val newGraph = Graph(newVertices, graph.edges)
```



```
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

**Khác biệt giữa hai đoạn code trên là gì?**  
Đoạn code đầu tiên không bảo toàn các cấu trúc và sẽ không được hưởng lợi từ việc tối ưu hóa hệ thống của GraphX

# Sửa đổi cấu trúc

```
class Graph[VD, ED] {  
    // Sửa đổi cấu trúc của đồ thị  
    def reverse: Graph[VD, ED]  
    def subgraph(  
        epred: EdgeTriplet[VD,ED] => Boolean = (x => true),  
        vpred: (VertexId, VD) => Boolean = ((v, d) => true)  
    ): Graph[VD, ED]  
    def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]  
    def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]  
}
```

# Kết với RDD

```
class Graph[VD, ED] {  
    // Kết RDD với đồ thị  
    def joinVertices[U](table: RDD[(VertexId, U)])(  
        mapFunc: (VertexId, VD, U) => VD): Graph[VD, ED]  
    def outerJoinVertices[U, VD2](other: RDD[(VertexId, U)])(  
        mapFunc: (VertexId, VD, Option[U]) => VD2) : Graph[VD2, ED]  
}
```

# Tổng hợp thông tin

```
class Graph[VD, ED] {  
    // Tổng hợp thông tin từ các bộ ba gần kề  
    def collectNeighborIds(edgeDirection: EdgeDirection):  
        VertexRDD[Array[VertexId]]  
    def collectNeighbors(edgeDirection: EdgeDirection):  
        VertexRDD[Array[(VertexId, VD)]]  
    def aggregateMessages[Msg: ClassTag](  
        sendMsg: EdgeContext[VD, ED, Msg] => Unit,  
        mergeMsg: (Msg, Msg) => Msg,  
        tripletFields: TripletFields = TripletFields.All  
    ): VertexRDD[A]  
}
```

# Tính toán trên đồ thị song song

```
class Graph[VD, ED] {  
    // Hàm Lặp trên đồ thị song song  
    def pregel[A](initialMsg: A, maxIterations: Int,  
                  activeDirection: EdgeDirection)(  
        vprog: (VertexId, VD, A) => VD,  
        sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],  
        mergeMsg: (A, A) => A  
    ) : Graph[VD, ED]  
}
```

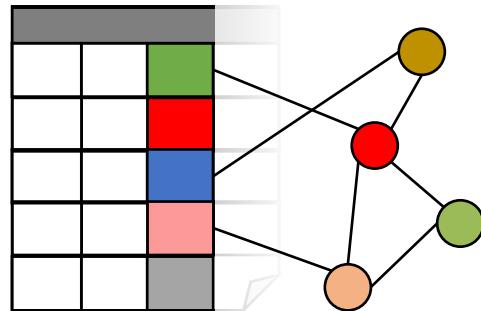
# Thuật toán cơ bản trên đồ thị

```
class Graph[VD, ED] {  
    // Những thuật toán đồ thị cơ bản  
    def pageRank(tol: Double,  
                 resetProb: Double = 0.15): Graph[Double, Double]  
    def connectedComponents(): Graph[VertexId, ED]  
    def triangleCount(): Graph[Int, ED]  
    def stronglyConnectedComponents(numIter: Int): Graph[VertexId, ED]  
}
```

# Hợp nhất bảng và đồ thị

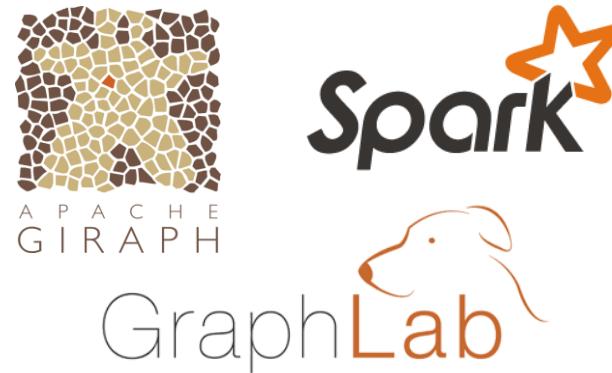
## API mới

Xóa nhòa sự khác biệt  
giữa Bảng và Đồ thị



## Hệ thống mới

Hợp nhất các hệ thống dữ liệu  
song song và đồ thị song song



Cho phép người dùng xây dựng toàn bộ quy trình phân tích một cách **dễ dàng** và **hiệu quả**

# TÀI LIỆU THAM KHẢO

1. Tom White. 2015. **Hadoop: The Definitive Guide (4<sup>th</sup> ed.)**. O'Reilly Media, Inc.
2. Donald Miner and Adam Shook. 2012. **MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems (1<sup>st</sup> ed.)**. O'Reilly Media, Inc.
3. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. **Learning Spark: Lightning-Fast Big Data Analytics (1<sup>st</sup> ed.)**. O'Reilly Media, Inc.
4. Sandy Ryza, Uri Laserson, Sean Owen and Josh Wills. 2017. **Advanced Analytics with Spark: Patterns for Learning from Data at Scale (2<sup>nd</sup> ed.)**. O'Reilly Media, Inc.

Q & A