

# MAPREDUCE & HADOOP

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

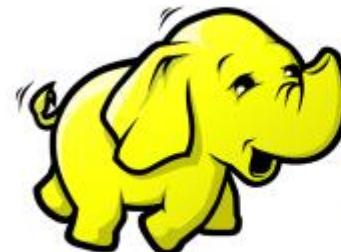
## MAPREDUCE

Mô hình



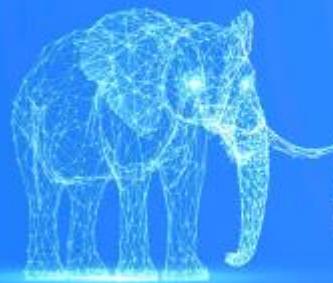
## APACHE HADOOP

Version 1.x – Framework



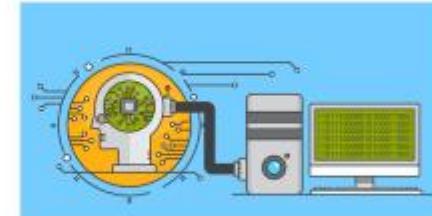
## APACHE HADOOP

Version 2.x – Framework



## THIẾT KẾ GIẢI THUẬT

Trên mô hình MapReduce



# MAPREDUCE

Mô hình



# NỘI DUNG

- Những hạn chế và vấn đề hiện tại
- Các thách thức
- Những công cụ hiện tại
- Ý tưởng từ lập trình hàm
- Mô hình MapReduce
- “Hello World”: Đếm từ
- Hiện thực mô hình

# Hạn chế

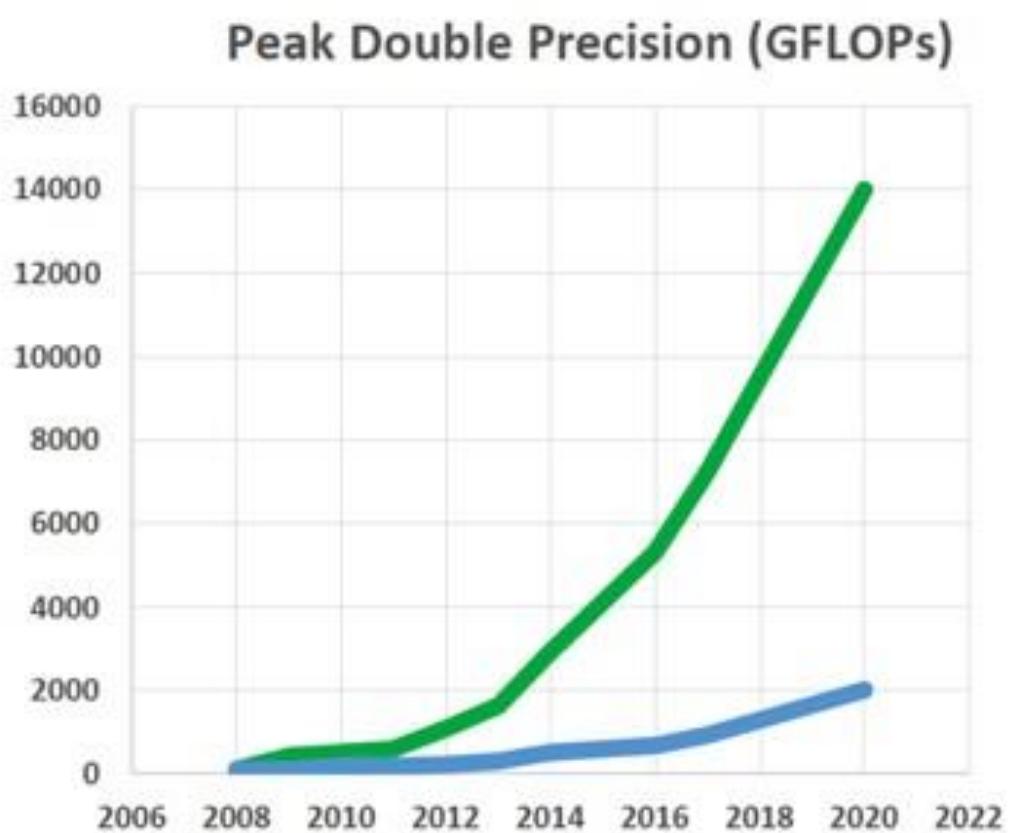
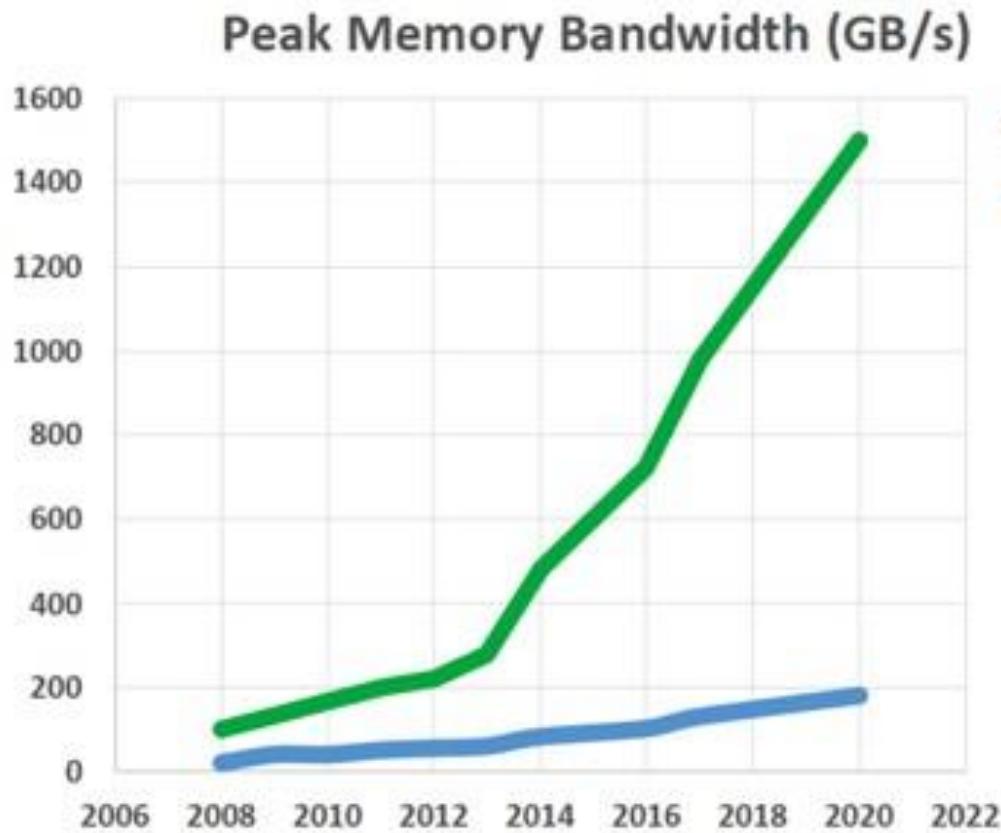
Trong sự phát triển của công nghệ



- Tốc độ đọc/ghi
  - HDD: UltraStar DC HC620 with SAS 12 GB/s interface = 255 MB/s
  - SSD: Samsung 970 Evo with PCIe 3 interface = 3500 MB/s (read), 2500 MB/s (write)

Lắp 1000 ổ cứng trong máy thì tốc độ là bao nhiêu?

Để đọc hết 100 PB dữ liệu thì mất bao lâu?



So sánh băng thông bộ nhớ cao nhất tính bằng GB/s và số phép toán có dấu chấm động tính được trên mỗi giây tính bằng GigaFlop của GPU Nvidia "Volta" V100 và CPU Intel "Cascade Lake" Xeon SP từ năm 2008 đến năm 2020.

**Tốc độ xử lý dữ liệu nhanh chóng, nhưng I/O rất chậm!**

# Hạn chế

Trong các mô hình phân tích dữ liệu hiện tại

- Chia sẻ dữ liệu vs. Không chia sẻ gì
  - Chia sẻ: quản lý trạng thái chung/toàn cục
  - Không chia sẻ gì: các thực thể độc lập, không có trạng thái chung
- Chia sẻ rất khó
  - Đồng bộ hóa, deadlock
  - Băng thông hữu hạn để truy cập dữ liệu từ SAN
  - Sự phụ thuộc theo thời gian rất phức tạp

- Chia phần dữ liệu để tính toán không theo tỷ lệ
- Không thể khai phá dữ liệu thô ban đầu (thường có độ chi tiết, đa dạng, trung thực cao)
- Mang đi cất trữ = dữ liệu chết sớm

# Hạn chế

## Trong các siêu máy tính

- Lỗi là điều xảy ra thường xuyên
  - Chủ yếu là do quy mô và môi trường chia sẻ
- Nguyên nhân gây lỗi
  - Phần cứng phần mềm
  - Điện, Làm mát, ...
  - Không có tài nguyên do quá tải
- Các loại lỗi
  - Dài hạn
  - Tạm thời

# Hạn chế

## Về băng thông kết nối

Kết nối mạng trở nên thắt nút cổ chai nếu lượng lớn dữ liệu cần được trao đổi giữa các nút tính toán và máy chủ

- Với băng thông mạng: 1Gbps
- Di chuyển 10 TB từ máy chủ này sang máy chủ khác mất 1 ngày
- Dữ liệu chỉ nên được di chuyển qua các nút khi điều này là bắt buộc
- Thông thường, các đoạn mã/chương trình có dung lượng nhỏ (vài MB)
- Di chuyển các đoạn mã/chương trình đến nơi chứa dữ liệu

→ **Cục bộ hóa dữ liệu**

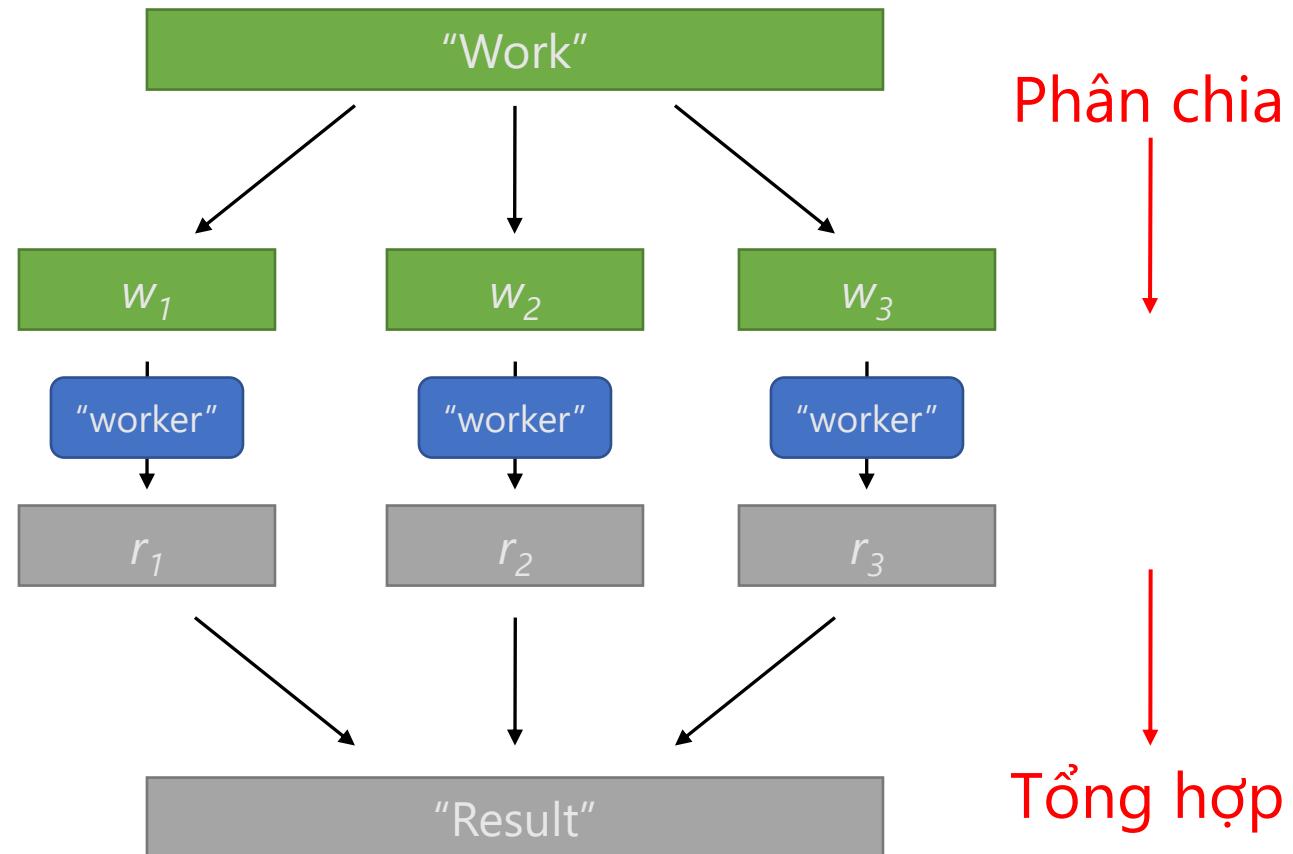
# Vấn đề điển hình

- Lần lượt duyệt qua số lượng lớn các dòng dữ liệu
- Trích xuất ra những thông tin cần quan tâm trong từng dòng
- Trộn và sắp xếp kết quả ngay lập tức
- Tổng hợp kết quả nhanh chóng
- Tạo ra tập tin kết quả ở đầu ra
- Vấn đề là:
  - Định dạng đầu vào đa dạng (dữ liệu đa dạng và không đồng nhất)
  - Quy mô lớn: Terabyte, Petabyte
  - Song song hóa

**Làm sao để tận dụng  
các máy tính giá rẻ?**



# “Chia để trị”



# **Thách thức của song song hóa**

- Làm thế nào để chia phần công việc cho 'worker'?
- Điều gì sẽ xảy ra nếu có nhiều phần công việc hơn số 'worker'?
- Điều gì sẽ xảy ra nếu 'worker' cần chia sẻ phần kết quả của chúng?
- Làm cách nào để tổng hợp các phần kết quả lại?
- Làm thế nào để biết tất cả các 'worker' đều đã hoàn thành?
- Nếu 'worker' không hoạt động thì sao?

**Điểm chung của những vấn đề trên là gì?**

## Điểm chung là

- Các vấn đề của song song hóa phát sinh từ:
  - Giao tiếp giữa các 'worker' (ví dụ: để trao đổi trạng thái)
  - Quyền truy cập vào các tài nguyên được chia sẻ (ví dụ: dữ liệu)
- Do đó, chúng ta cần một cơ chế đồng bộ hóa



Source: Ricardo Guimarães Herrmann

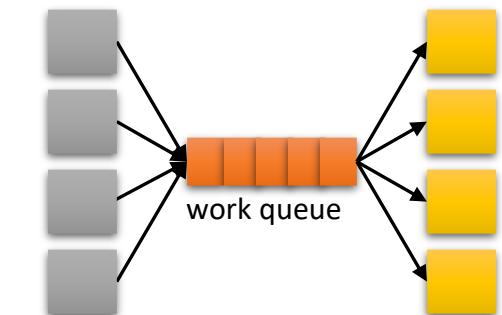
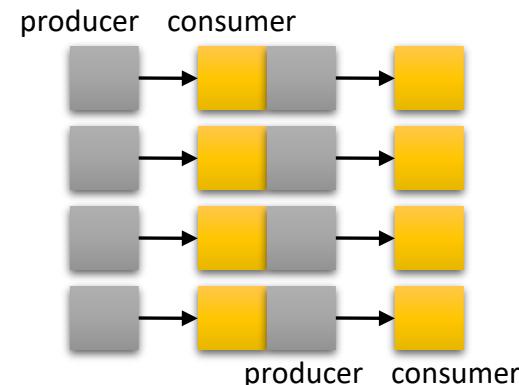
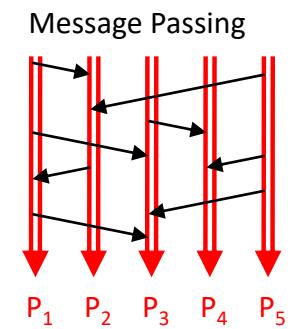
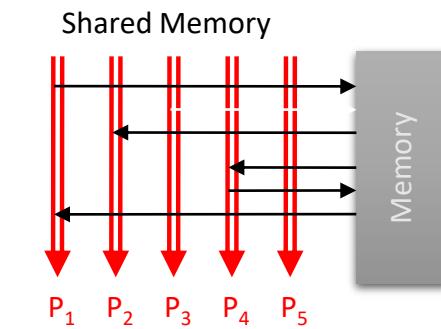
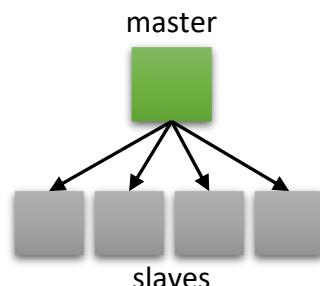
# Quản lý cùng lúc nhiều 'worker'

- Đây là một bài toán khó, vì
  - Không biết rõ thứ tự vận hành của các 'worker'
  - Không biết khi nào các 'worker' sẽ ngừng các 'worker' khác
  - Không biết thứ tự yêu cầu chia sẻ tài nguyên của các 'worker'
- Do đó, cần
  - Các cơ chế khóa (lock, unlock)
  - Đặt các biến điều kiện (wait, notify, broadcast)
  - Đưa ra các rào cản

- Vẫn còn rất nhiều vấn đề:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Hãy cẩn thận với thứ tự của việc thực thi!

# Những công cụ hiện tại

- Những mô hình lập trình
  - Shared memory (pthreads)
  - Message passing (MPI)
- Những mẫu thiết kế (Design patterns)
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues



# Thách thức trong xử lý đồng thời

- Xử lý đồng thời rất khó để giải thích
- Thậm chí nó còn khó giải thích hơn khi
  - Ở quy mô datacenter (hay giữa các datacenter)
  - Khi gặp lỗi
  - Đề cập đến các dịch vụ tương tác
- Chưa kể đến việc gỡ lỗi...
- Thực tế là:
  - Rất nhiều giải pháp dùng một lần, mã nguồn tùy chỉnh
  - Viết thư viện dành riêng cho bạn, sau đó lập trình với nó
  - Lập trình viên sẽ chịu trách nhiệm lớn trong việc quản lý mọi thứ một cách rõ ràng

# Vấn đề ở đây là gì?

- Mức độ trừu tượng phù hợp
  - Kiến trúc von Neumann đã rất tốt đối với chúng ta, nhưng không còn phù hợp với môi trường đa lõi/cụm
- Giấu đi các chi tiết ở mức hệ thống đối với những nhà phát triển
  - Không còn race conditions, tranh chấp khóa, v.v...
- Tách biệt giữa “cái gì” và “như thế nào”
  - Người lập trình chỉ cần đưa ra những yêu cầu tính toán
  - Framework sẽ quyết định việc thực thi (“runtime”) thực tế như thế nào

# Ý tưởng chính

- “Mở rộng”, chứ không phải “nâng cấp”
  - Giới hạn của đa xử lý đối xứng (SMP) và máy tính lớn chia sẻ chung bộ nhớ
- Chuyển việc xử lý đến nơi chứa dữ liệu
  - Cụm tính toán có băng thông hạn chế
- Xử lý dữ liệu tuần tự, tránh truy cập ngẫu nhiên
  - Thao tác tìm kiếm gây tổn kém, quản lý thông lượng đĩa sao cho hợp lý
- Khả năng mở rộng liền mạch
  - Biến điều không thể thành có thể

# Vấn đề điển hình

- Lần lượt duyệt qua số lượng lớn các dòng dữ liệu

**Map** Trích xuất ra những thông tin cần quan tâm trong từng dòng

- Trộn và sắp xếp kết quả ngay lập tức
- Tổng hợp kết quả nhanh chóng **Reduce**
- Tạo ra tập tin kết quả ở đầu ra

**Ý tưởng chính: cung cấp một hàm trừu tượng  
(functional abstraction) cho hai hành động này**

# Ý tưởng từ dữ liệu

- Hệ thống phân tán truyền thống (ví dụ: HPC) di chuyển dữ liệu đến các nút máy tính (máy chủ)
  - Cách tiếp cận này không thể được sử dụng để xử lý dữ liệu đến mức TB
    - Băng thông mạng bị hạn chế
- Hadoop chuyển mã chương trình sang vị trí chứa dữ liệu
  - Mã chương trình (thường có dung lượng rất nhỏ) được sao chép và thực thi trên các máy chủ chứa các dữ liệu có liên quan
  - Cách tiếp cận này dựa trên yêu cầu **cục bộ hóa dữ liệu**

# Ý tưởng từ lập trình hàm

- MapReduce = lập trình hàm cộng với xử lý phân tán trên một mô hình lập đặc trưng
  - Không phải là một ý tưởng mới ... có từ những năm 50 (hoặc thậm chí 30)
- Lập trình hàm là gì?
  - Tính toán bằng cách áp dụng các hàm
  - Tính toán giá trị của các hàm toán học
  - Tránh các trạng thái và thay đổi dữ liệu
  - Nhấn mạnh việc áp dụng các hàm thay vì thay đổi trạng thái

- Ngôn ngữ lập trình hàm khác như thế nào?
  - Các khái niệm truyền thống về “dữ liệu” và “chỉ lệnh” không áp dụng được
  - Luồng dữ liệu bị ẩn trong chương trình
  - Các lệnh có thể được thực thi theo thứ tự khác nhau
  - Dựa trên nền tảng lý thuyết giải tích lambda
    - Phương pháp chính thức để định nghĩa một hàm
- Ví dụ ngôn ngữ LISP, Scheme

## Ngôn ngữ LISP

Các hàm được viết bằng ký hiệu tiền tố

(+ 1 2) → 3

(\* 3 4) → 12

(sqrt (+ (\* 3 3) (\* 4 4))) → 5

(define x 3) → x

(\* x 5) → 15

- Hàm = biểu thức lambda liên kết với các biến

Ví dụ biểu diễn bằng lambda: (+ 1 2) → 3     $\lambda x \lambda y . x + y$

```
(define foo  
  (lambda (x y)  
    (sqrt (+ (* x x) (* y y)))))
```

- Các dòng lệnh phía trên có thể được viết lại thành:

```
(define (foo x y)  
  (sqrt (+ (* x x) (* y y)))))
```

- Sau khi định nghĩa, có thể sử dụng hàm bằng cách gọi ra nó:

```
(foo 3 4) → 5
```

- Hai khái niệm quan trọng trong lập trình hàm
  - Map: thực hiện điều gì đó với mọi phần tử trong danh sách
  - Fold: kết hợp các kết quả của danh sách theo một cách nào đó

## Hàm Map

- Các hàm bậc cao hơn - chấp nhận các hàm khác làm đối số
  - Map
    - Xét một hàm **f** và giá trị đầu vào của nó, là một danh sách
    - Áp dụng cho tất cả các phần tử trong danh sách
    - Kết quả trả về là một danh sách
  - Danh sách là kiểu dữ liệu nguyên thủy
    - [1 2 3 4 5]
    - [[a 1] [b 2] [c 3]]

Ví dụ hàm map đơn giản:

```
(map (lambda (x) (* x x)) [1 2 3 4 5]) → [1 4 9 16 25]
```

## Hàm Reduce

- Fold
  - Xét hàm **g**, có 2 đối số: một giá trị ban đầu và một danh sách.
  - Áp dụng hàm **g** cho giá trị ban đầu và phần tử đầu tiên trong danh sách
  - Kết quả được lưu trữ trong biến trung gian
  - Tiếp tục áp dụng hàm **g** cho biến trung gian và phần tử tiếp theo trong danh sách
  - Fold trả về giá trị cuối cùng của biến trung gian

- Ví dụ hàm map đơn giản:

```
(map (lambda (x) (* x x)) [1 2 3 4 5]) → [1 4 9 16 25]
```

- Ví dụ fold:

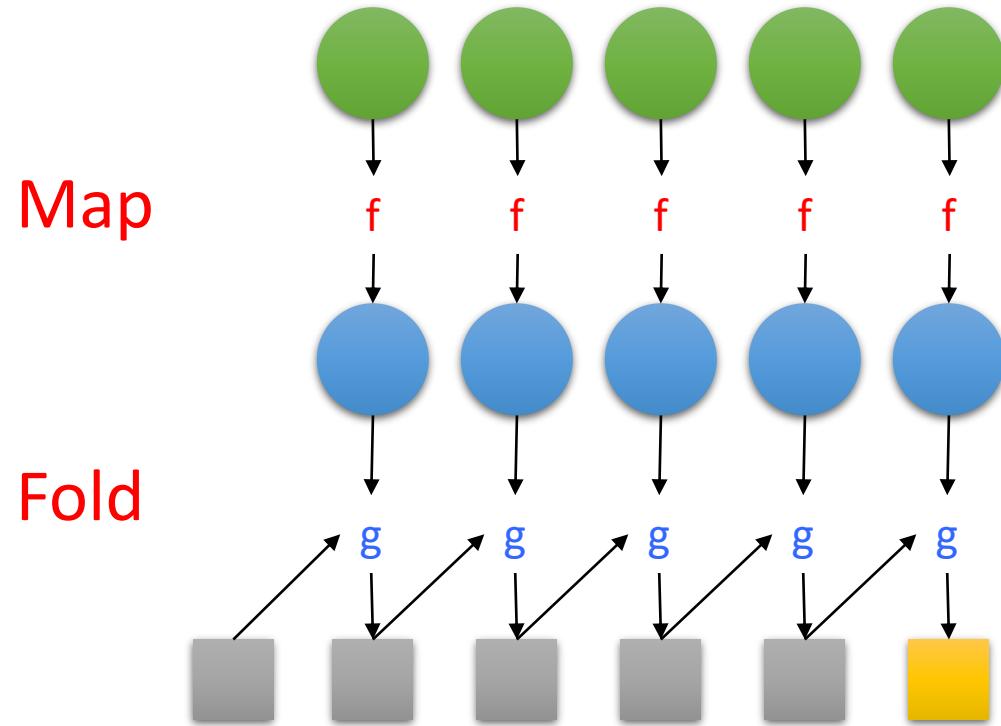
```
(fold + 0 [1 2 3 4 5]) → 15
```

```
(fold * 1 [1 2 3 4 5]) → 120
```

- Tổng các bình phương:

```
(define (sum-of-squares v) // voi v la mot danh sach  
      (fold + 0 (map (lambda (x) (* x x)) v)))
```

```
(sum-of-squares [1 2 3 4 5]) → 55
```



- Sử dụng kết hợp map/fold
- Map – biến đổi tập dữ liệu
- Fold- thực hiện việc tổng hợp
- Có thể song song hóa hàm map
- Fold – nhiều hạn chế hơn, các phần tử phải được kết hợp với nhau
  - Nhiều trường hợp không cần áp dụng hàm **g** lên tất cả các phần tử trong danh sách, lúc này có thể song song hóa hàm fold

# MapReduce

- Map trong MapReduce tương tự như lập trình hàm
- Reduce tương ứng với hàm fold
- Gồm 2 bước:
  - Tính toán do người dùng lập trình được áp dụng trên tất cả các phần tử đầu vào, có thể thực thi song song, trả về kết quả đầu ra ở biến trung gian
  - Kết quả được tổng hợp bằng một thao tác khác do người lập trình chỉ định

## Cấu trúc dữ liệu

- Cặp *khóa-giá trị* là cấu trúc dữ liệu cơ bản trong MapReduce
  - Các khóa và giá trị có thể là: số nguyên, số thực, chuỗi...
  - Chúng cũng có thể (gần như) là cấu trúc dữ liệu tùy ý do người thiết kế mong muốn
- Cả đầu vào và đầu ra của MapReduce đều là danh sách các cặp *khóa-giá trị*
  - Lưu ý rằng đầu vào cũng là danh sách các cặp *khóa-giá trị*

- Khi xây dựng thuật toán trên mô hình MapReduce, lập trình viên cần thiết kế cấu trúc *khóa-giá trị*
  - Áp đặt cấu trúc *khóa-giá trị* trên tập dữ liệu đầu vào và đầu ra
    - Ví dụ: đối với một tập hợp các trang Web, khóa đầu vào có thể là URL và giá trị có thể là nội dung HTML.

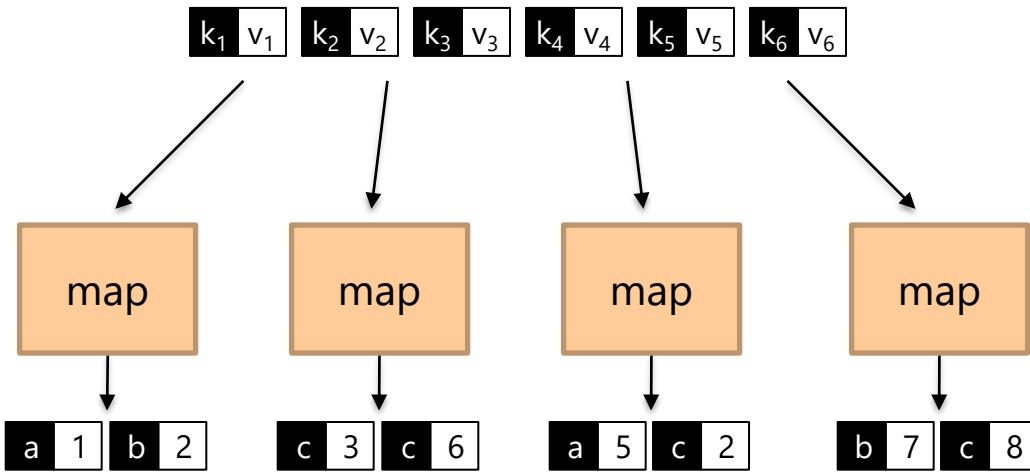
## Cấu trúc hàm

- Lập trình viên chỉ định hai hàm:

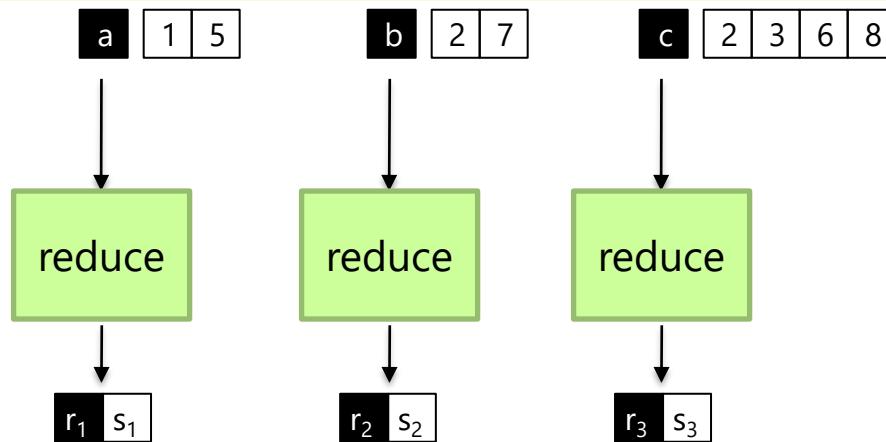
**map** ( $k, v$ )  $\rightarrow$   $[(k', v')]$

**reduce** ( $k', [v']$ )  $\rightarrow$   $[(k', v'')]$

- Tất cả các giá trị (value) nào có cùng khóa (key) sẽ được gửi đến cùng một reducer
- Framework thực thi sẽ xử lý tất cả những vấn đề khác...



**Shuffle và Sort:** tổng hợp các giá trị theo khóa



- Lập trình viên chỉ định hai hàm:

**map** ( $k, v$ )  $\rightarrow [(k', v')]$

**reduce** ( $k', [v']$ )  $\rightarrow [(k', v'')]$

- Tất cả các giá trị (value) nào có cùng khóa (key) sẽ được gửi đến cùng một reducer
- Framework thực thi sẽ xử lý tất cả những vấn đề khác...

**“Những vấn đề khác” ở đây là gì?**

## MapReduce “Runtime”

- Xử lý lập lịch
  - Gán các công việc map và reduce cho các worker.
- Xử lý “phân tán dữ liệu”
  - Chuyển việc xử lý đến nơi chứa dữ liệu
- Xử lý đồng bộ hóa
  - Thu thập, sắp xếp và xáo trộn dữ liệu trung gian
- Xử lý lỗi và hỏng hóc
  - Phát hiện lỗi của worker và khởi động lại
- Mọi thứ được thực hiện phía trên hệ thống tập tin phân tán

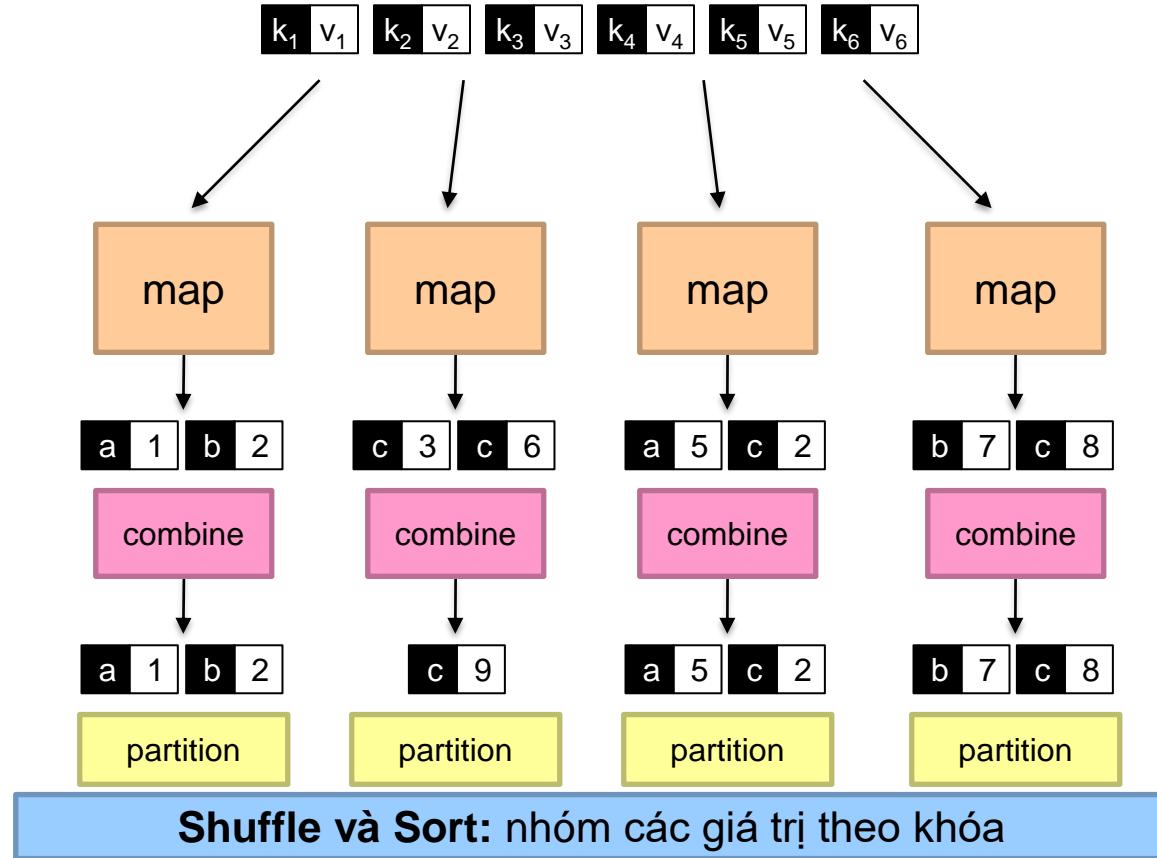
- Lập trình viên chỉ định hai hàm:
  - map** ( $k, v \rightarrow [(k', v')]$ )
  - reduce** ( $(k', [v']) \rightarrow [(k', v'')]$ )
    - Tất cả các giá trị (value) nào có cùng khóa (key) sẽ được gửi đến cùng một reducer
- Framework thực thi sẽ xử lý tất cả những vấn đề khác...
- Đôi khi, các lập trình viên cũng cần chỉ định các hàm khác như:

**partition** ( $k'$ , số lượng phân mảnh)  $\rightarrow$  phân mảnh khóa  $k'$

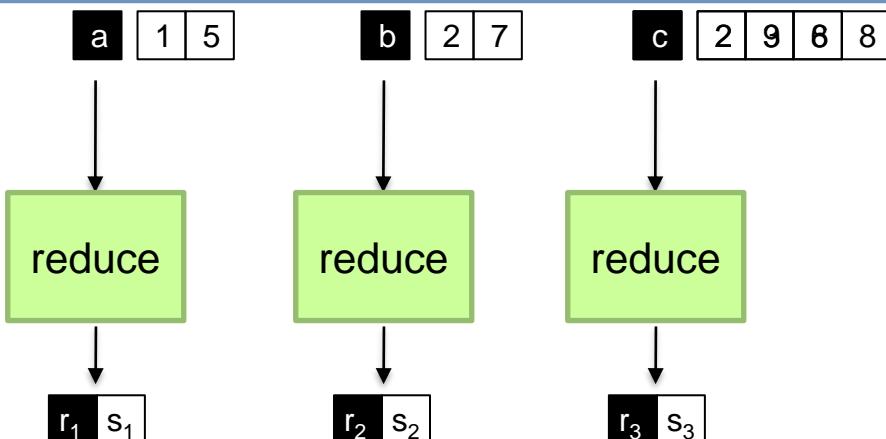
- Thường là một hàm băm khóa  $k'$  đơn giản, ví dụ:  $\text{hash}(k') \bmod n$
- Chia khoảng các khóa để song song hóa tác vụ reduce

**combine** ( $(k', [v']) \rightarrow [(k', v'')]$ )

- Một dạng hàm reduce nhỏ, chạy trong bộ nhớ sau pha map.
- Được sử dụng để tối ưu hóa nhờ việc giảm lưu lượng mạng.



**Shuffle và Sort:** nhóm các giá trị theo khóa



# **MapReduce có thể được hiểu là...**

- Một mô hình lập trình
- Một framework thực thi (hay còn gọi là “runtime”)
- Một công cụ được hiện thực chuyên biệt

**Tùy từng trường hợp!**

# **"Hello World": Đếm từ**

**Map(String docid, String text):**

```
    for each word w in text:  
        Emit(w, 1);
```

**Reduce(String term, Iterator<Int> values):**

```
    int sum = 0;  
    for each v in values:  
        sum += v;  
    Emit(term, sum);
```

# **Ưu điểm**

MapReduce được thiết kế cho các thao tác

- Xử lý hàng loạt, chủ yếu là quét toàn bộ dữ liệu
- Các ứng dụng sử dụng nhiều dữ liệu
  - Đọc và xử lý toàn bộ trang Web (ví dụ: thuật toán PageRank)
  - Đọc và xử lý toàn bộ Đồ thị mạng xã hội (ví dụ: thuật toán “gợi ý kết bạn” LinkPrediction)
  - Phân tích bản ghi nhật ký (ví dụ: dấu vết mạng, dữ liệu đồng hồ thông minh...)

## Hạn chế

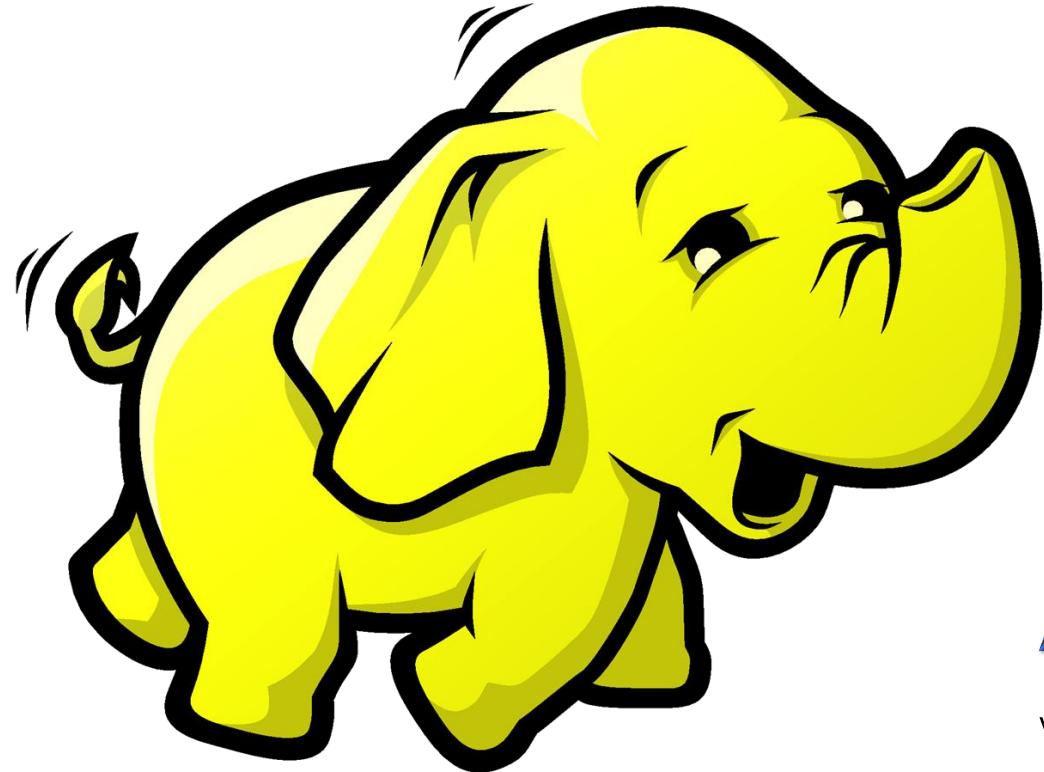
MapReduce không có khả năng giải quyết tất cả các vấn đề về dữ liệu lớn

MapReduce không hoạt động tốt với

- Những vấn đề lặp đi lặp lại
- Bài toán đệ quy
- Xử lý dữ liệu luồng (stream)

# Hiện thực mô hình MapReduce

- Google có một ứng dụng độc quyền hiện thực mô hình được viết trên C++
  - Có thể kết nối với Java, Python
- Hadoop là một hiện thực mô hình, mã nguồn mở, viết trên Java
  - Phát triển bởi Yahoo, để sử dụng trong sản phẩm của họ
  - Hiện trở thành một dự án của tổ chức Quỹ Phần mềm Apache (ASF)
  - Hệ sinh thái phần mềm của Hadoop đang được mở rộng nhanh chóng
- Rất nhiều các hiện thực khác đã được tùy chỉnh lại
  - Dành cho GPU, bộ xử lý di động, v.v...



# APACHE HADOOP

Version 1.x – Framework

# NỘI DUNG

- Lịch sử phát triển
- Giới thiệu Apache Hadoop
- Ví dụ: Map-Reduce-Driver
- Tối ưu hóa hàm Combiner
- Hệ thống tập tin phân tán HDFS
- Các thành phần của Hadoop

# Lịch sử phát triển Hadoop

- 12/2004 – Bài báo về Google GFS được công bố
- 07/2005 – Nutch sử dụng mô hình MapReduce
- 02/2006 – Trở thành dự án con của Lucene
- 04/2007 – Yahoo! Triển khai cụm tính toán 1000 nút
- 01/2008 – Trở thành dự án hàng đầu của Apache
- 07/2008 – Thủ nghiệm cụm tính toán 4000 nút

- 09/2008 – Hive trở thành một dự án của Hadoop
- 02/2009 – The Yahoo! Search Webmap là một ứng dụng Hadoop chạy trên cụm máy Linux có hơn 10000 nhân và xử lý dữ liệu cho tất cả các truy vấn trên dịch vụ tìm kiếm của Yahoo!.
- 06/2009 – Ngày 10/06/2009, Yahoo! công bố mã nguồn của phiên bản Hadoop mà họ đang sử dụng trong các dịch vụ của mình.
- 2010 – Facebook tuyên bố họ đang sử dụng cụm tính toán Hadoop lớn nhất thế giới với 21 PB dữ liệu lưu trữ. Ngày 27/07/2011 Facebook thông báo khối lượng dữ liệu đã lên đến 30 PB.

# Apache Hadoop

- Hệ thống phân tán, có khả năng chịu lỗi và mở rộng, cho Dữ liệu lớn:
  - Lưu trữ dữ liệu
  - Xử lý dữ liệu
  - Một máy ảo Dữ liệu lớn
  - Mượn ý tưởng từ Google; Mã nguồn mở dưới giấy phép Apache
- Nhân Hadoop gồm có hai hệ thống chính:
  - **Hadoop/MapReduce**: cung cấp hạ tầng xử lý dữ liệu lớn phân tán (mô hình, khả năng chịu lỗi, lập lịch, thực thi)
  - **HDFS (Hadoop Distributed File System)**: hệ thống lưu trữ phân tán, có khả năng chịu lỗi, băng thông rộng, tính sẵn sàng cao

# Ai đang sử dụng Hadoop?

- Amazon/A9
- Facebook
- Google
- IBM
- Joost
- Last.fm
- New York Times
- PowerSet
- Veoh
- Yahoo!
- ...



# Ví dụ: Đếm từ Hàm map()

```
public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}
```

## Hàm reduce()

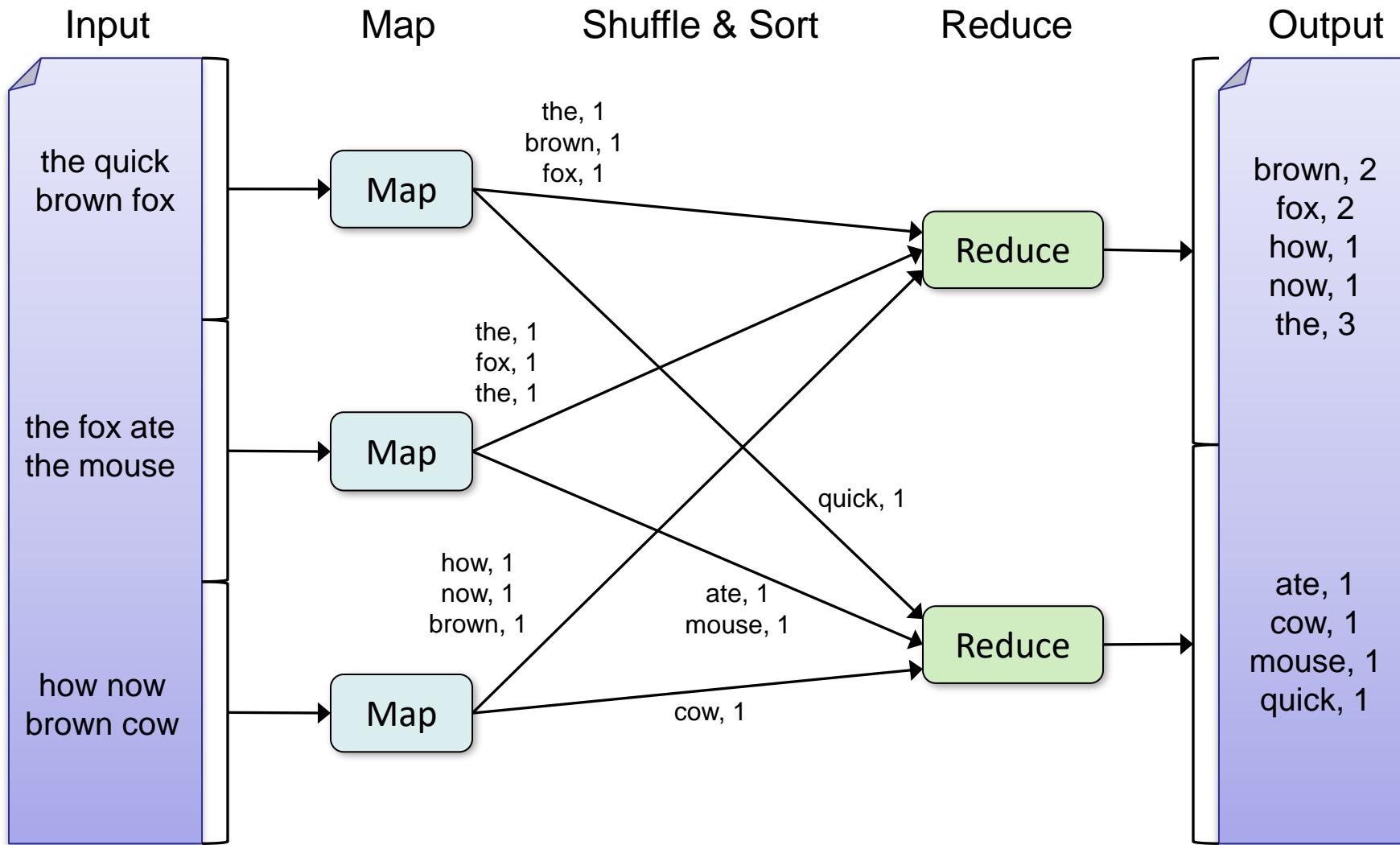
```
public static class IntSumReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                      Context context
                      ) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

## Hàm driver()

```
public static void main(String[] args) throws Exception {  
    Configuration conf = new Configuration();  
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();  
    if (otherArgs.length != 2) {  
        System.err.println("Usage: wordcount <in> <out>");  
        System.exit(2);  
    }  
    Job job = new Job(conf, "word count");  
    job.setJarByClass(WordCount.class);  
    job.setMapperClass(TokenizerMapper.class);  
    job.setCombinerClass(IntSumReducer.class);  
    job.setReducerClass(IntSumReducer.class);  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(IntWritable.class);  
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));  
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));  
    System.exit(job.waitForCompletion(true) ? 0 : 1);  
}
```

# Thực thi ứng dụng Đếm từ

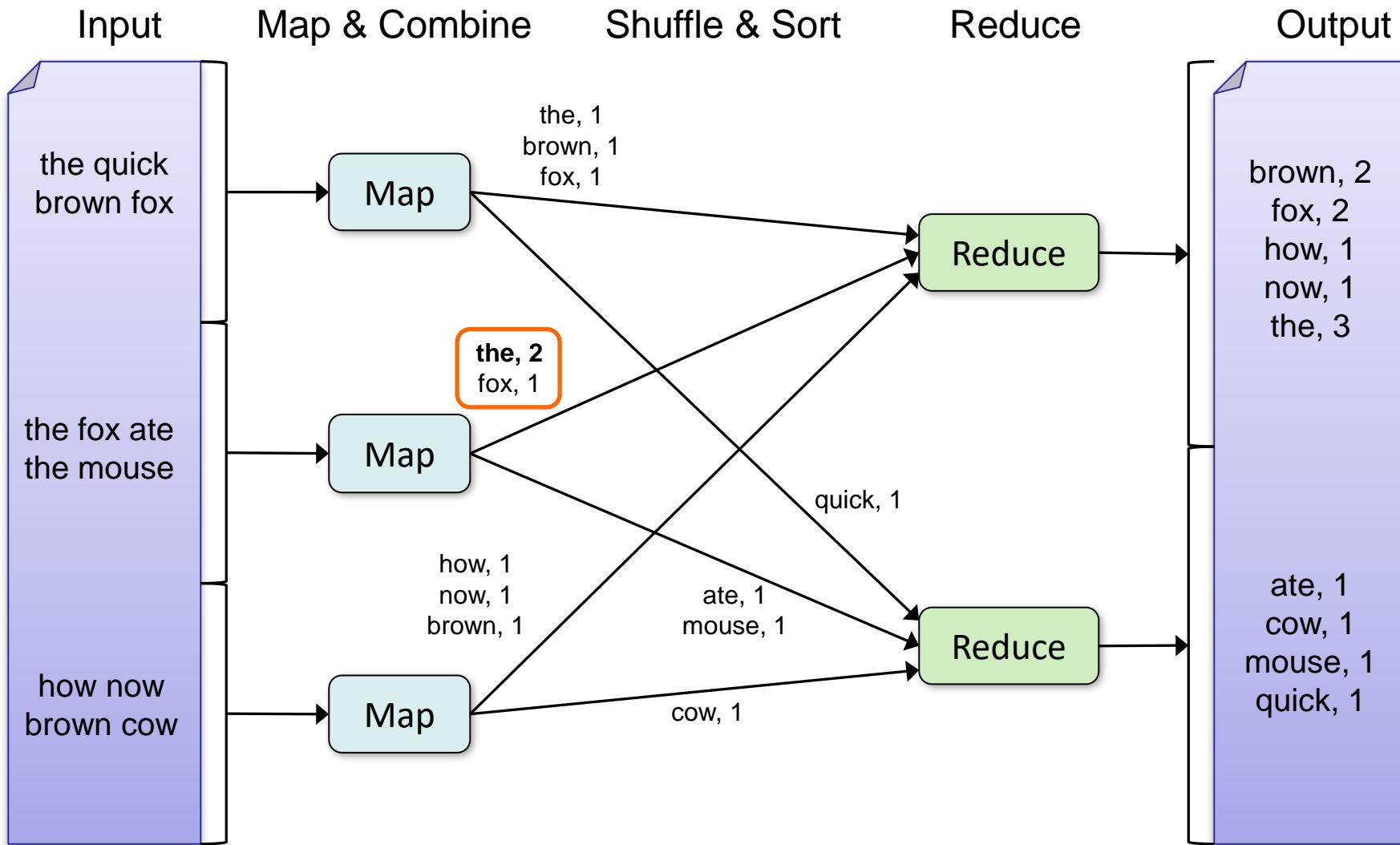


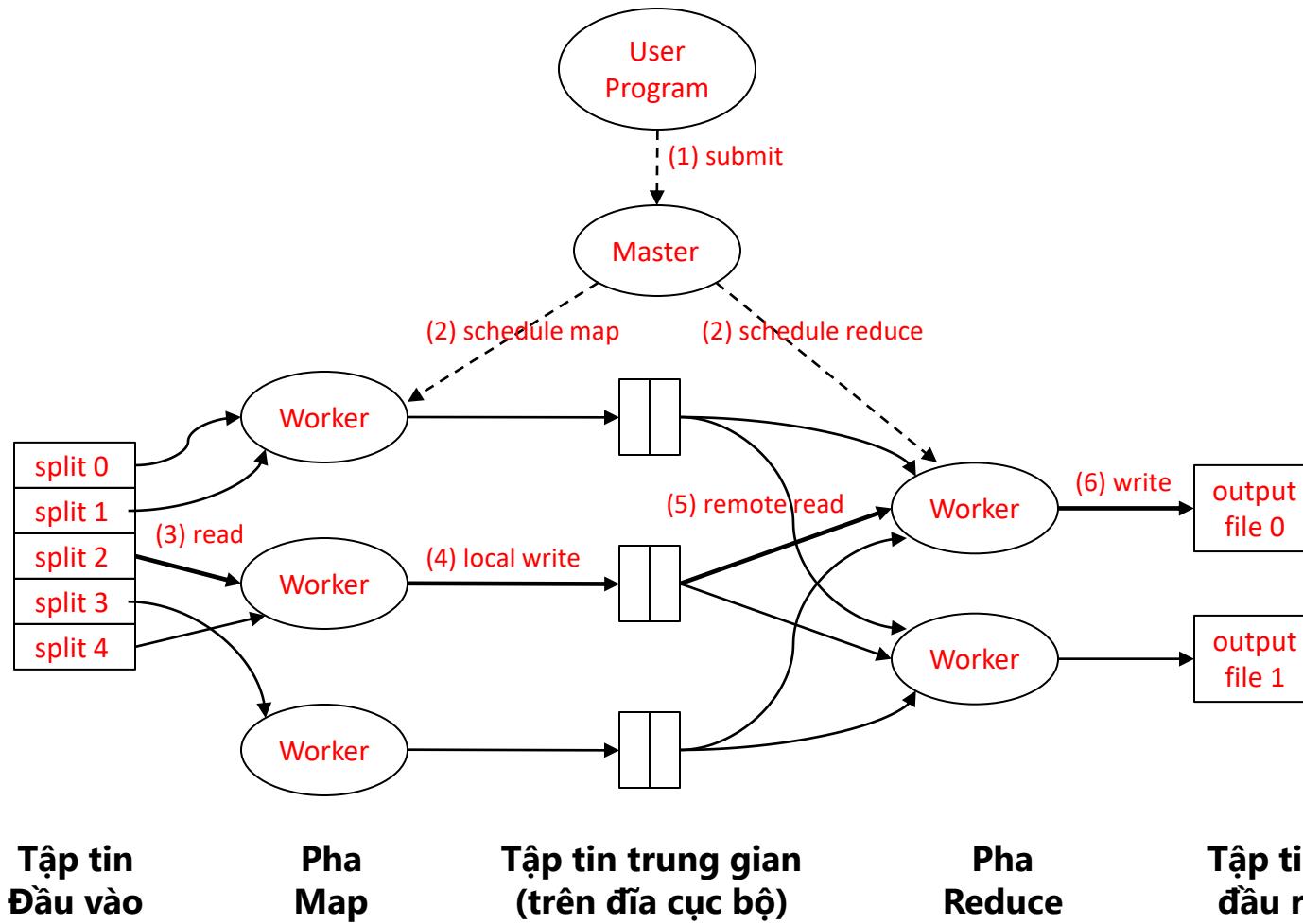
# Tối ưu hóa: hàm Combiner

- Combiner là hàm tổng hợp cục bộ cho các khóa lặp lại được tạo bởi cùng một hàm map.
- Với những thao tác liên kết như sum, count, max
- Giảm kích thước của dữ liệu trung gian
- Ví dụ: hàm đếm cục bộ cho ứng dụng Đếm từ:

```
def combiner(key, values):  
    output(key, sum(values))
```

# Đếm từ có sử dụng hàm Combiner





# Làm sao đưa dữ liệu cho worker?



Sẽ có những vấn đề gì xảy ra?

# Hệ thống tập tin phân tán

- Đừng chuyển dữ liệu đến worker... hãy chuyển worker đến dữ liệu!
  - Dữ liệu được lưu tại đĩa cục bộ trong các nút của cụm máy
  - Chạy worker tại nút có dữ liệu cục bộ.
- Tại sao?
  - Không đủ RAM để giữ tất cả dữ liệu trên bộ nhớ
  - Thao tác đọc ổ cứng chậm, nhưng thông lượng đã lý tưởng
- Hệ thống tập tin phân tán là câu trả lời
  - GFS (Google File System) cho Google's MapReduce
  - HDFS (Hadoop Distributed File System) cho Hadoop

# GFS: Giả định

- Phần cứng bình thường hơn phần cứng “xịn”
  - “Mở rộng”, không phải “nâng cấp”
- Tỷ lệ hỏng hóc linh kiện cao
  - Các thành phần linh kiện không đắt tiền luôn hỏng hóc
- “Đông đảo” số lượng tập tin lớn
  - Các tập tin nhiều gigabyte cần xử lý sẽ khá phổ biến
- Các tập tin được ghi một lần, sau đó chủ yếu được nối vào
  - Thao tác nối nhiều lúc được các ‘worker’ thực hiện đồng thời
- Luồng trực tuyến lớn đọc dữ liệu thông qua truy cập ngẫu nhiên
  - Thông lượng duy trì cao với độ trễ thấp

# GFS: Thiết kế

- Tập tin được lưu trữ dưới dạng khối
  - Kích thước cố định (64MB)
- Độ tin cậy thông qua nhân rộng
  - Mỗi khối được sao chép trên 3+ server
- Một máy chính để điều phối quyền truy cập, giữ siêu dữ liệu
  - Quản lý tập trung đơn giản
- Không có bộ nhớ đệm dữ liệu
  - Lợi ích nhỏ do bộ dữ liệu lớn, đọc thành luồng.
- Đơn giản hóa API
  - Đầu một số vấn đề lên máy khách (ví dụ: bố cục dữ liệu)

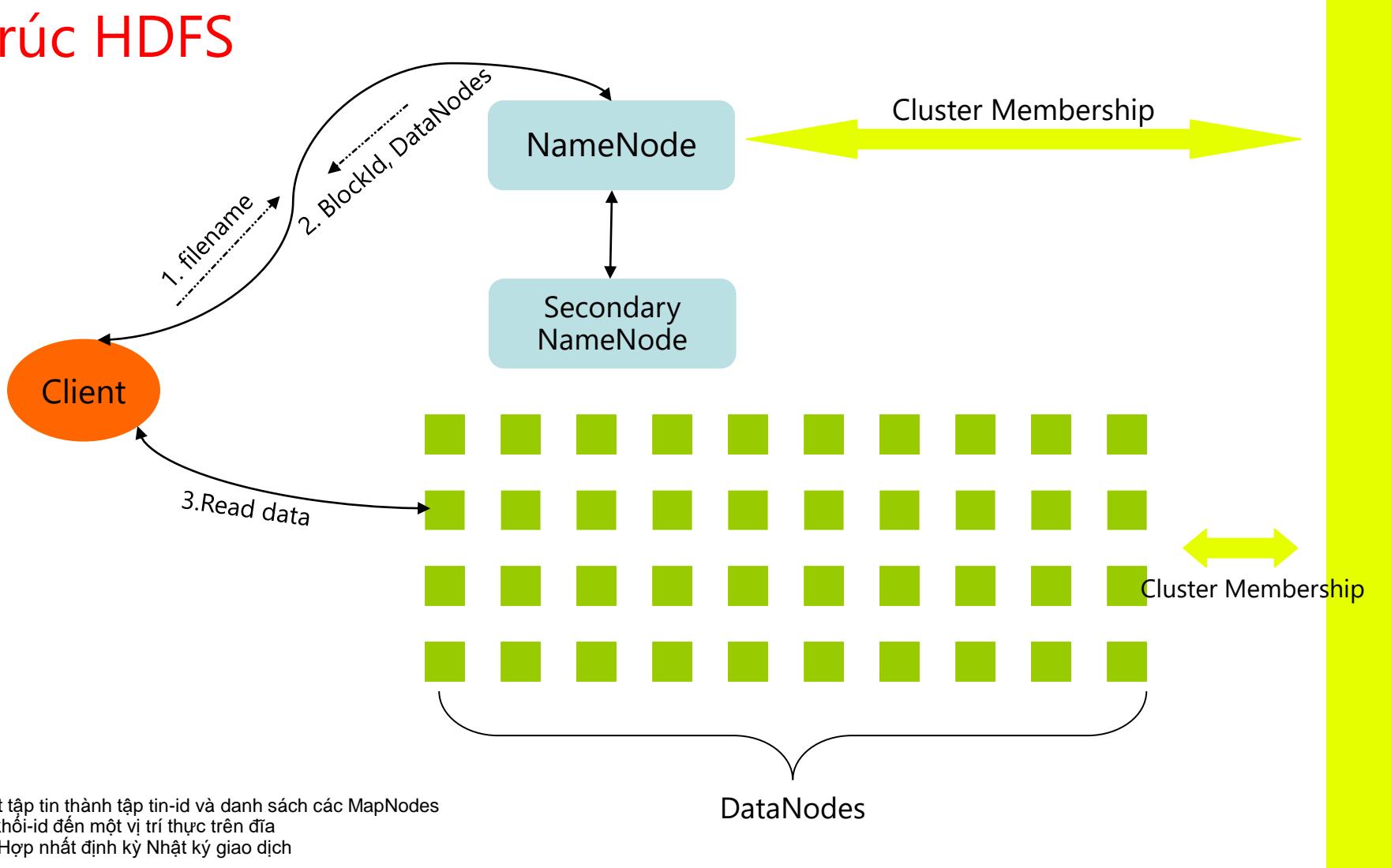
**HDFS = bản sao GFS (giống nhau ý tưởng cơ bản)**

# **GFS vs. HDFS**

- Tên gọi khác nhau:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Chức năng khác biệt:
  - Hiệu năng của HDFS (có vẻ) chậm hơn

**Phần lớn, sẽ sử dụng thuật ngữ Hadoop...**

# Kiến trúc HDFS



NameNode: Ánh xạ một tập tin thành tập tin-id và danh sách các MapNodes

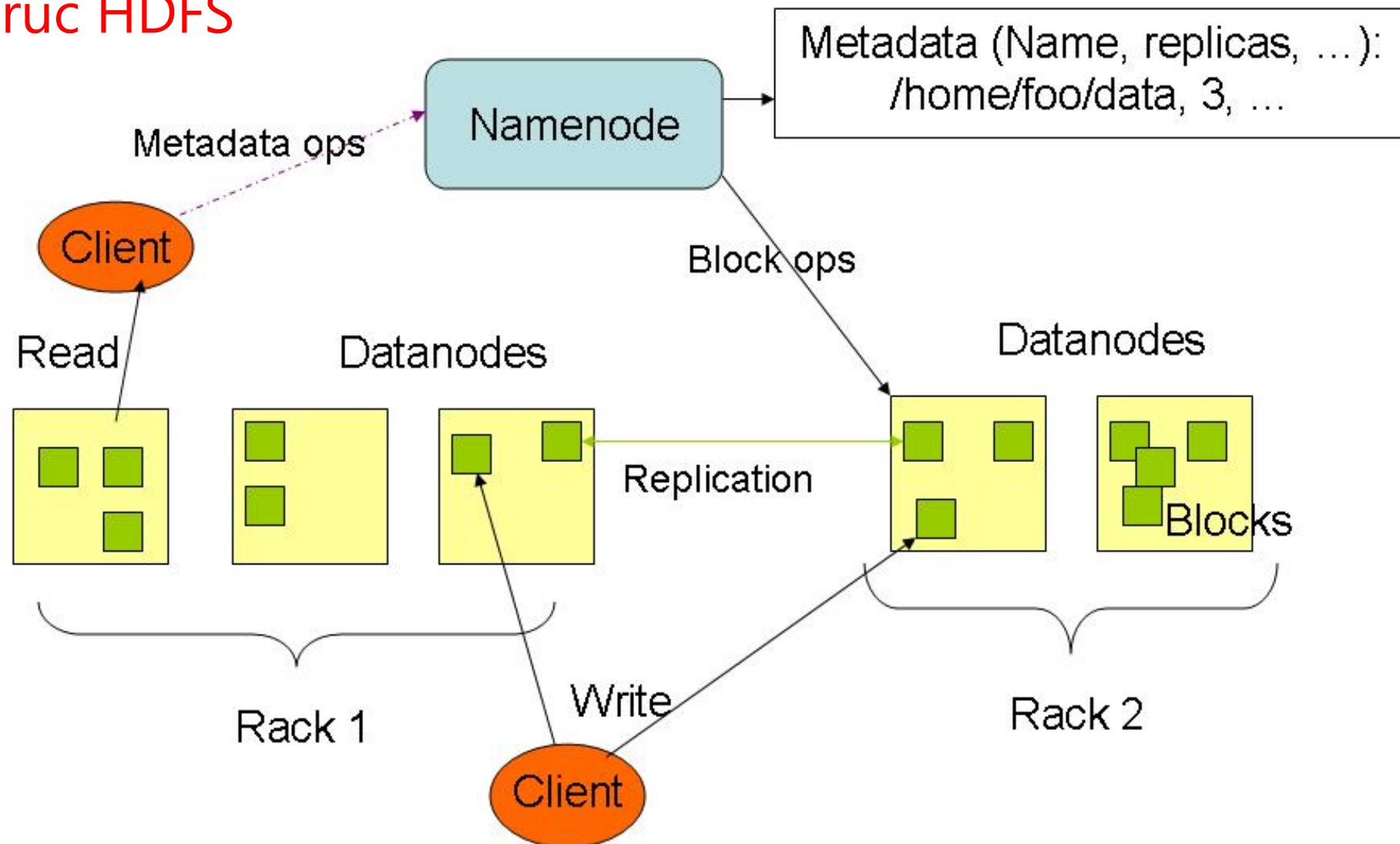
DataNode: Ánh xạ mã khối-id đến một vị trí thực trên đĩa

SecondaryNameNode: Hợp nhất định kỳ Nhật ký giao dịch

# Hệ thống tập tin phân tán

- Single Namespace cho toàn bộ cụm máy
- Dữ liệu kết hợp
  - Mô hình truy xuất ghi-một\_lần-đọc-nhiều\_lần
  - Ứng dụng chỉ có thể thêm vào các tập tin hiện có
- Các tập tin được chia thành các khối
  - Kích thước khối thường là 128 MB
  - Mỗi khối được sao chép trên nhiều DataNodes
- Ứng dụng thông minh
  - Ứng dụng có thể tìm thấy vị trí của các khối
  - Ứng dụng có thể truy cập dữ liệu trực tiếp từ DataNode

## Kiến trúc HDFS



# **Siêu dữ liệu NameNode**

- Siêu dữ liệu trong bộ nhớ
  - Toàn bộ siêu dữ liệu nằm trong bộ nhớ chính
  - Siêu dữ liệu không cần phân trang theo yêu cầu
- Các loại siêu dữ liệu
  - Danh sách các tập tin
  - Danh sách các khối của mỗi tập tin
  - Danh sách các DataNodes cho mỗi khối
  - Thuộc tính tập tin, ví dụ: thời gian tạo, yếu tố nhân bản
- Nhật ký giao dịch (EditLog)
  - Ghi lại các thao tác tạo/xóa tập tin.

# Trách nhiệm của NameNode

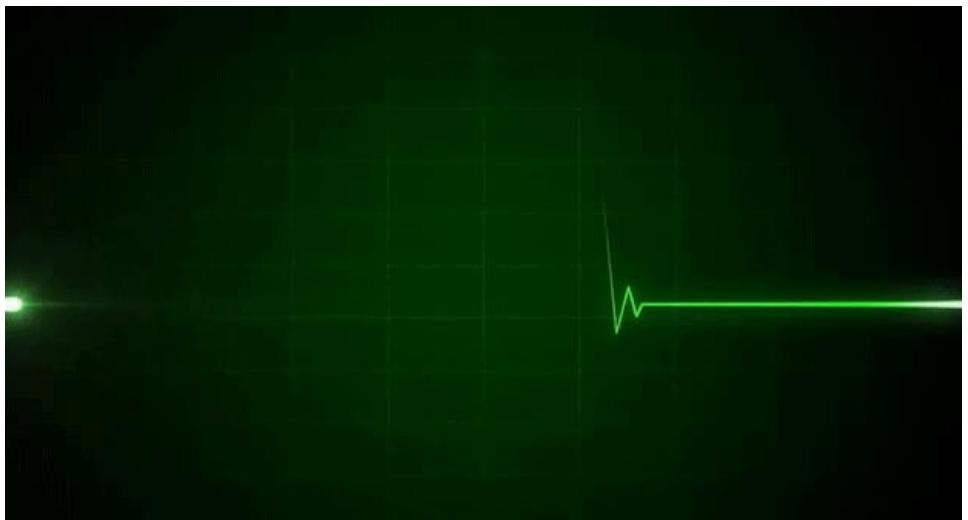
- Quản lý tập tin không gian tên (namespace) của hệ thống:
  - Giữ cấu trúc tập tin/thư mục, siêu dữ liệu, chỉ định tập tin tới các khối, chỉ định vị trí đặt khối ở các DataNode, quyền truy cập, v.v.
- Điều phối các hoạt động tập tin:
  - Hướng dẫn ứng dụng đến các nút dữ liệu để đọc và ghi
  - Không có dữ liệu nào được di chuyển qua NameNode
- Duy trì sức khỏe tổng thể của hệ thống:
  - Giao tiếp định kỳ với các nút dữ liệu
  - Tái nhân bản khối và tái cân bằng
  - Thu gom rác hệ thống

# DataNode

- **Máy chủ khối (block server)**
  - Lưu trữ dữ liệu trong hệ thống tập tin cục bộ (ví dụ: ext3)
  - Lưu trữ siêu dữ liệu của một khối (ví dụ: CRC)
  - Cung cấp dữ liệu và siêu dữ liệu cho ứng dụng
- **Báo cáo khối**
  - Định kỳ gửi một báo cáo về tất cả các khối hiện có tới NameNode
- **Tạo điều kiện thuận lợi cho việc phân chia dữ liệu**
  - Chuyển tiếp dữ liệu đến các DataNode được chỉ định khác

# “...Nhịp...tim...”

- Các DataNode sẽ gửi nhịp tim đến NameNode
  - Cứ 3 giây một lần
- NameNode sử dụng nhịp tim để phát hiện DataNode nào gặp lỗi
  - Nếu trong vòng 10 phút NameNode không “nghe thấy” nhịp tim từ DataNode, nó sẽ bắt đầu tái nhân bản các khối của DataNode đó.

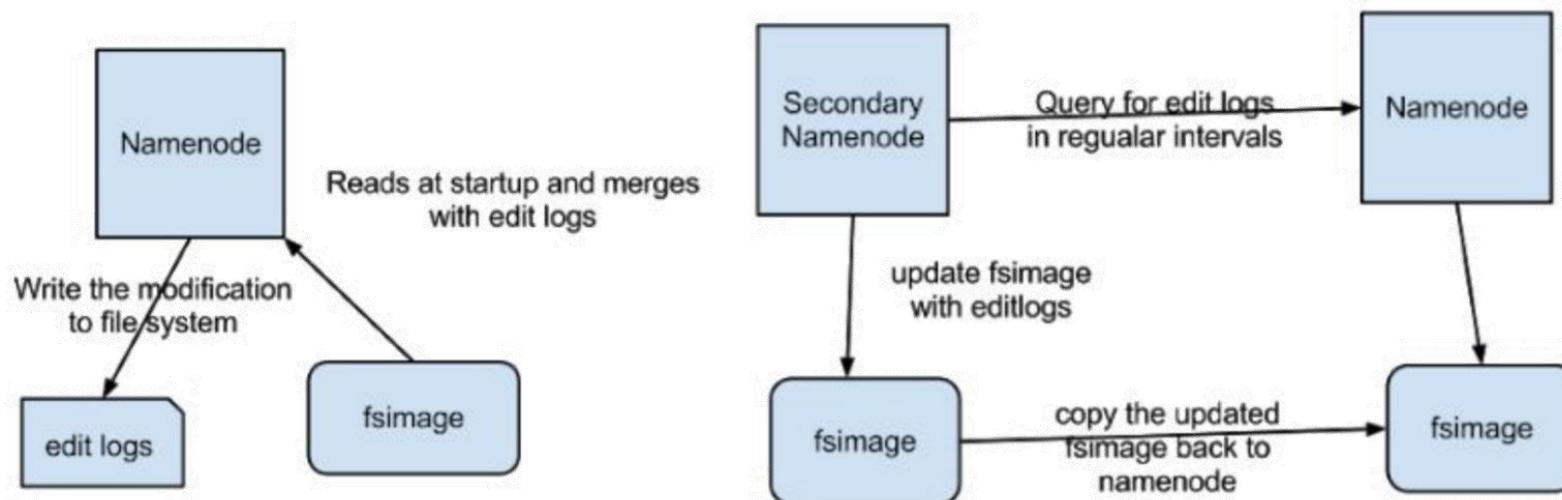


# **Độ chính xác của dữ liệu**

- Sử dụng Checksums để kiểm tra dữ liệu
  - Sử dụng CRC32
- Tạo tập tin
  - Ứng dụng tính toán checksum mỗi 512 byte
  - DataNode lưu trữ checksum
- Truy xuất tập tin
  - Ứng dụng truy xuất dữ liệu và checksum từ DataNode
  - Nếu xác thực không thành công, ứng dụng sẽ thử các bản sao khác

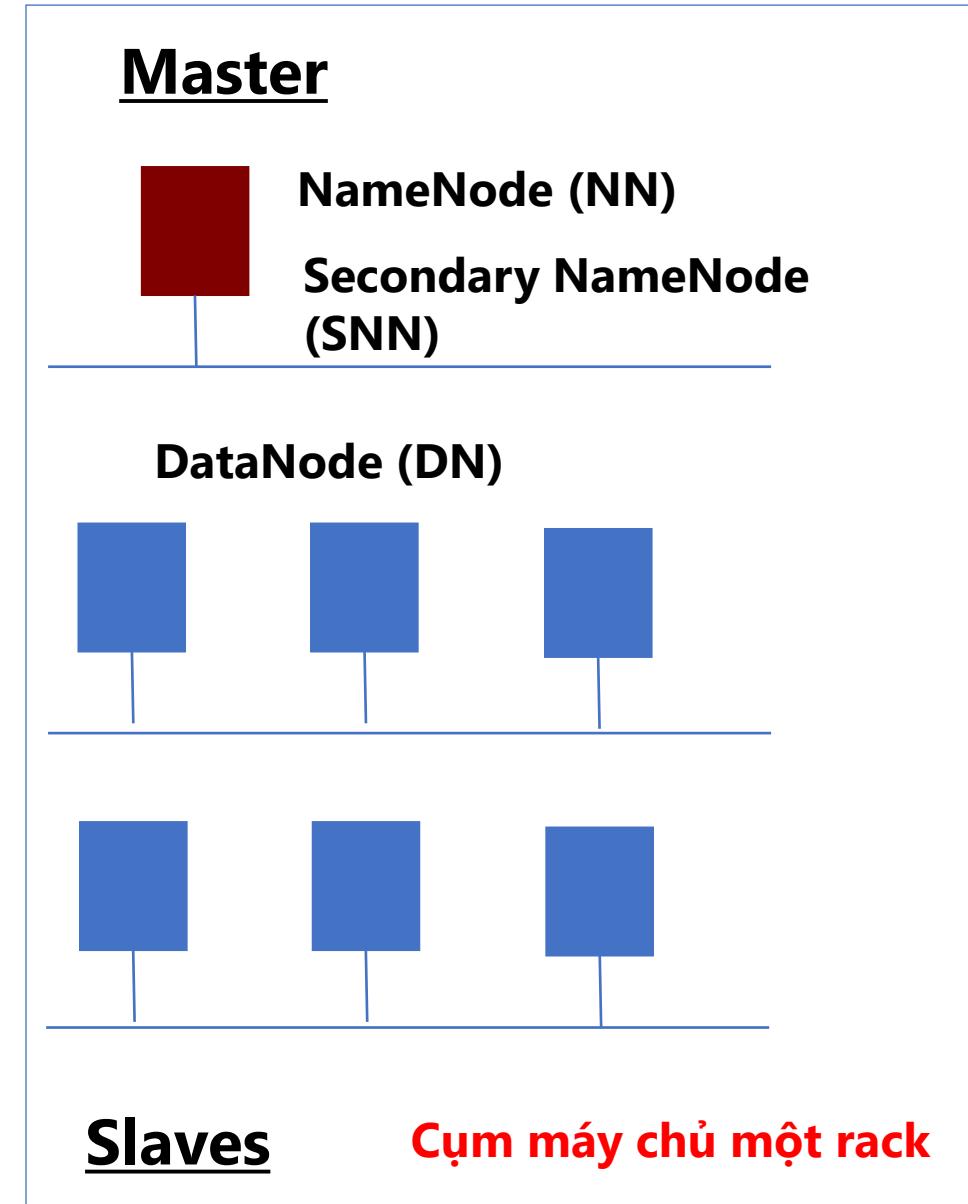
# Secondary NameNode

- Sao chép FsImage và EditLog từ NameNode vào một thư mục tạm thời
- Hợp nhất FSImage và EditLog vào một tập tin FSImage mới trong thư mục tạm thời
- Khôi phục lại FSImage ở NameNode
  - Nếu EditLog trên NameNode bị xóa

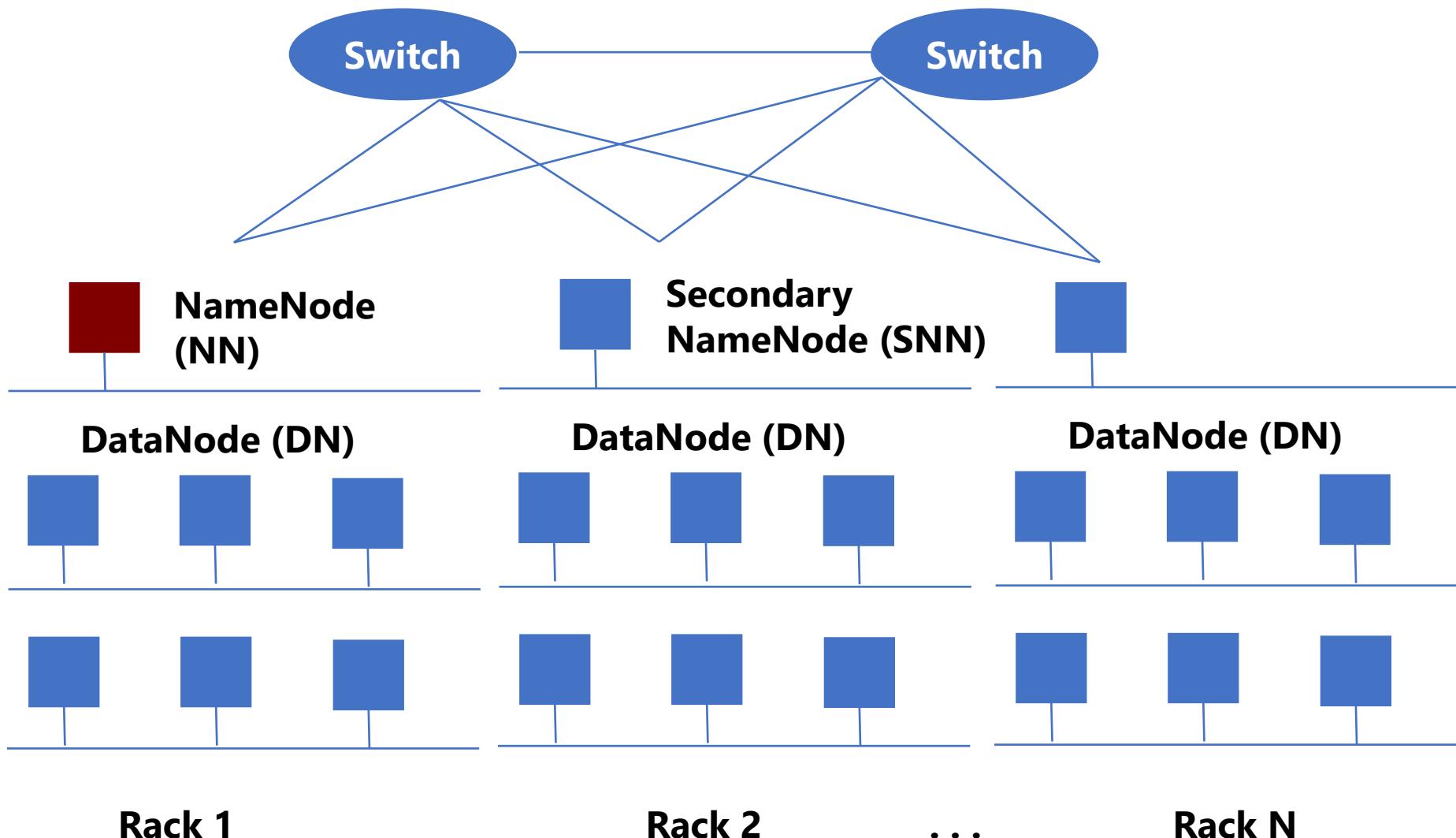


- Không phải là NameNode phụ/thay thế
- Định kỳ kết nối với NameNode, đọc tập tin EditLog và cập nhật các thay đổi vào tập tin FSImage
- Quản lý, sao lưu siêu dữ liệu NameNode
- Siêu dữ liệu đã lưu dùng để khôi phục NameNode gặp lỗi

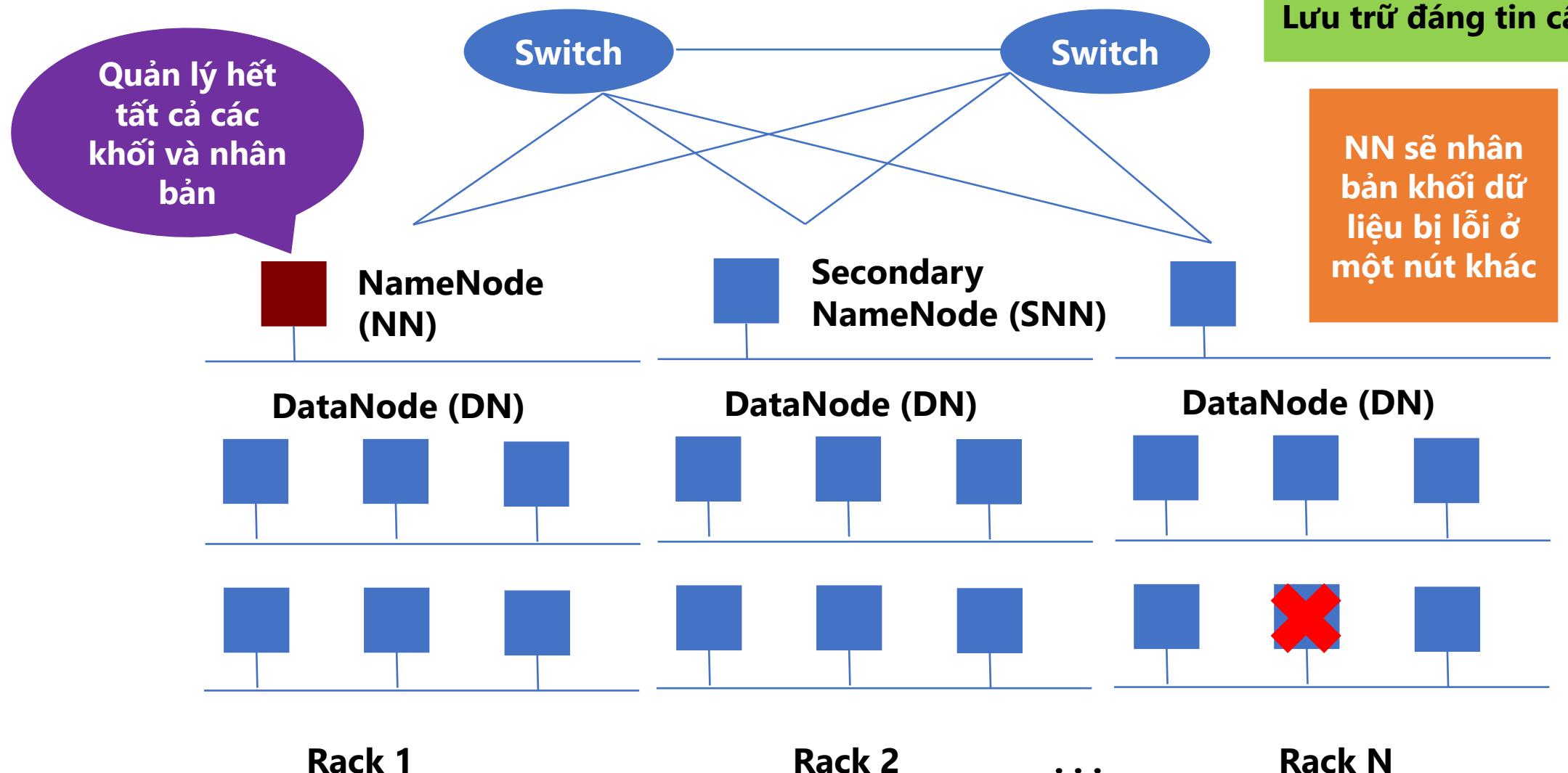
# Đặt các thành phần vào HDFS...



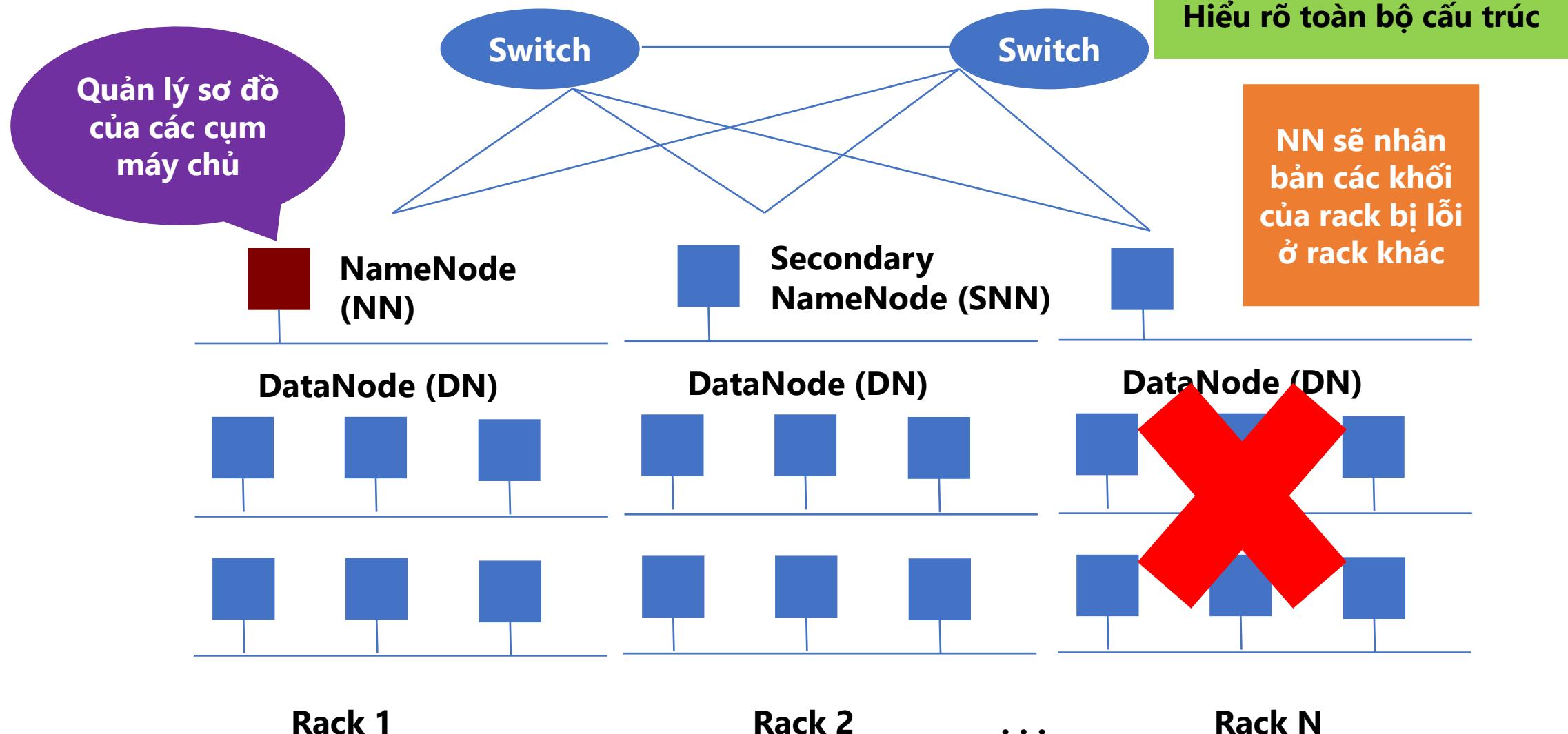
## Cụm máy chủ nhiều rack



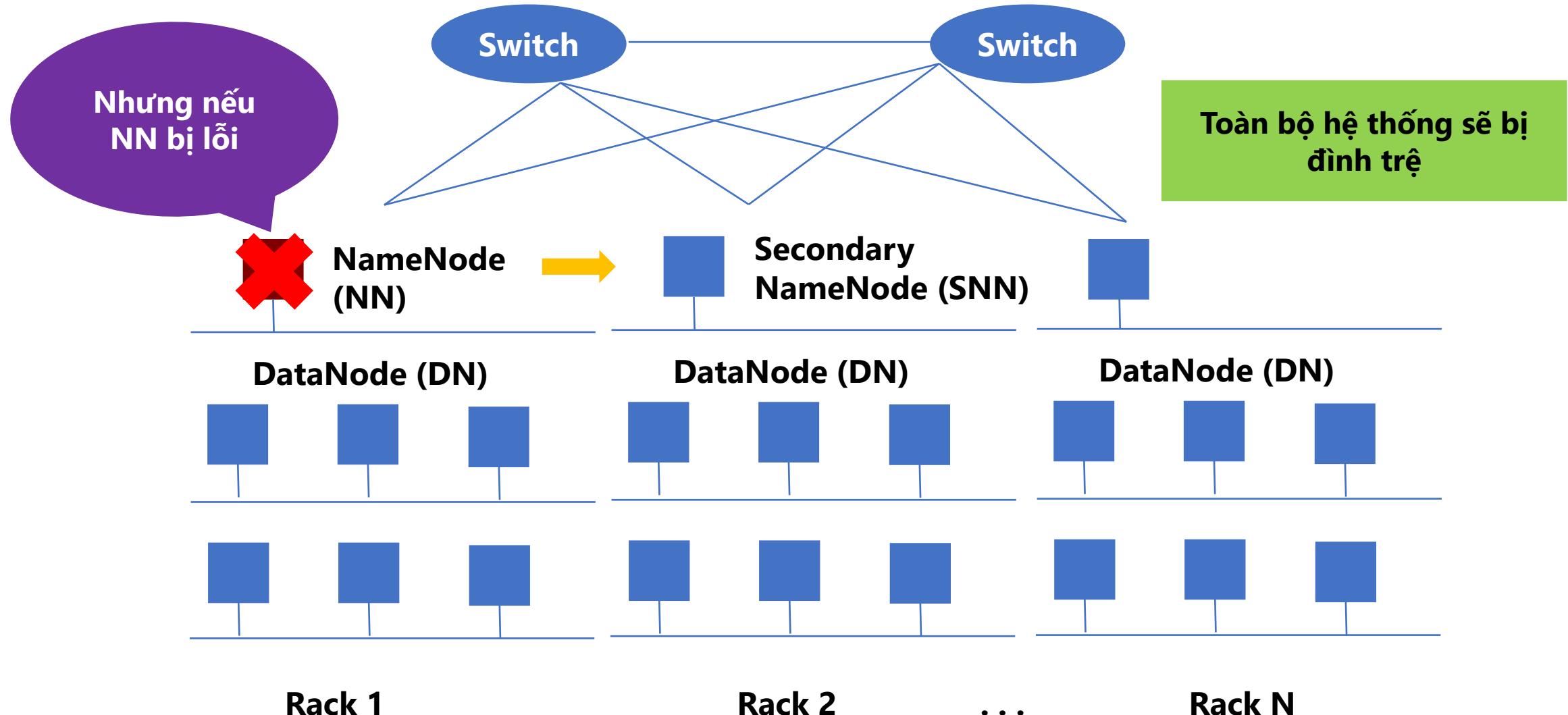
## Cụm máy chủ nhiều rack



## Cụm máy chủ nhiều rack



## Cụm máy chủ nhiều rack



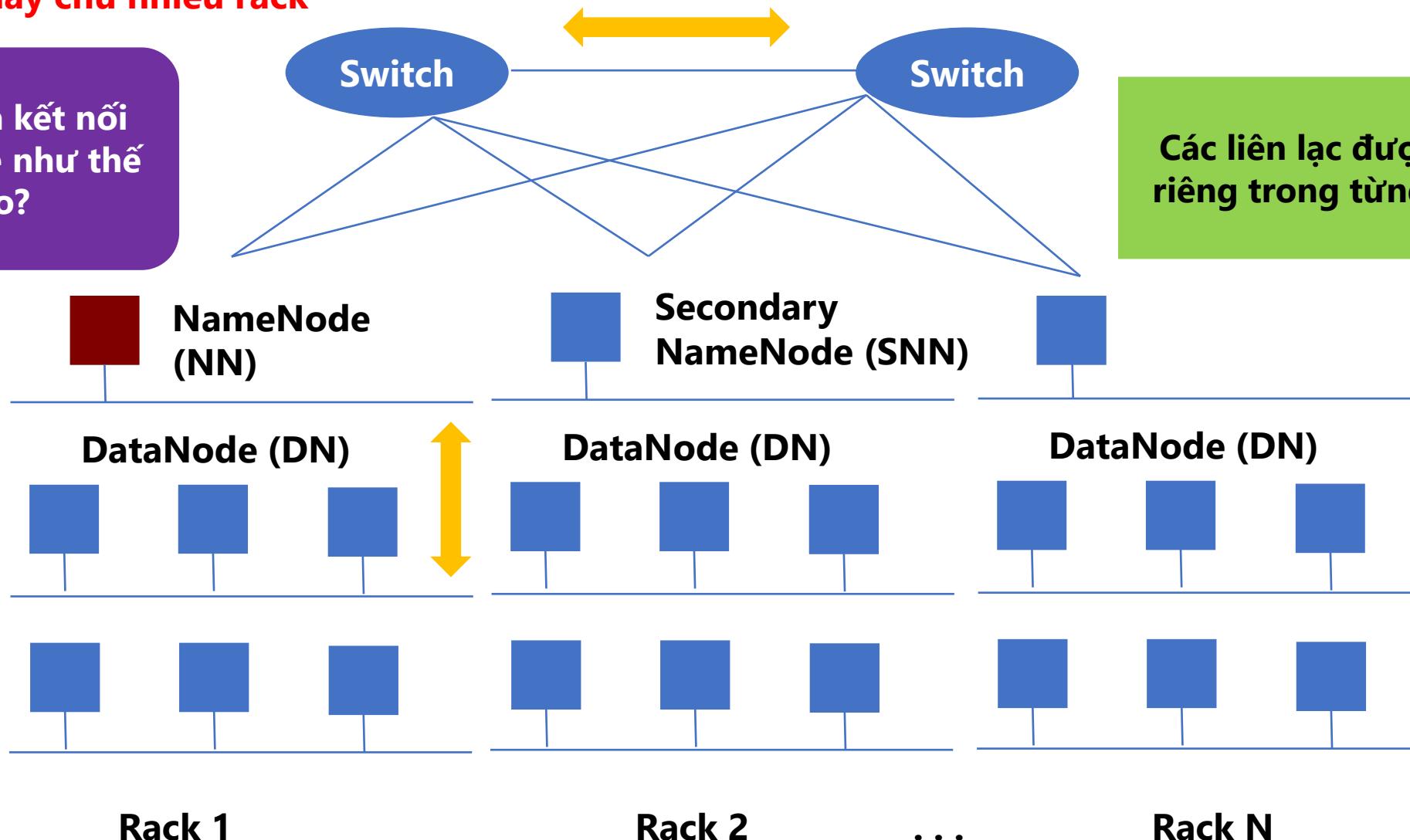
# Lỗi NameNode

- Là thành phần duy nhất nên khi gặp lỗi sẽ làm hệ thống ngưng trệ
- Nhật ký giao dịch (EditLog) được lưu trữ trong nhiều thư mục
  - Một thư mục trên hệ thống tập tin cục bộ
  - Một thư mục trên hệ thống tập tin từ xa (NFS/CIFS)
- Cần phát triển một giải pháp có tính khả dụng cao thực sự

## Cụm máy chủ nhiều rack

Hiệu quả kết nối  
lúc này sẽ như thế  
nào?

Các liên lạc được giữ  
riêng trong từng rack



# Khối dữ liệu

Tại sao lại cần khái niệm “Khối” bên cạnh khái niệm “Tập tin”?

Lý do vì:

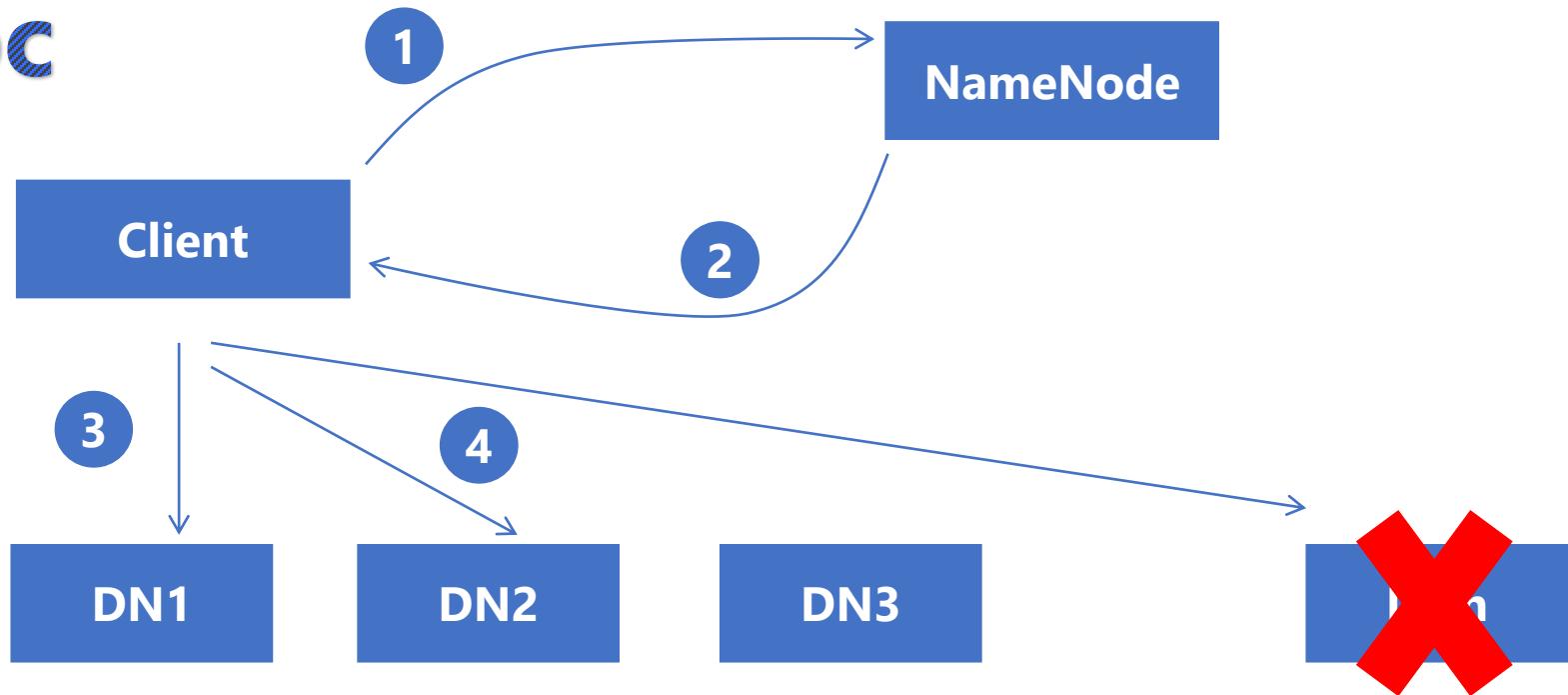
- Tập tin có thể lớn hơn kích thước của một đĩa cứng
- Block có kích thước cố định, dễ quản lý và thao tác
- Dễ dàng sao chép và cân bằng tải chi tiết hơn

Kích thước khối mặc định trong HDFS là **64 MB**, tại sao giá trị này lại lớn hơn rất nhiều lần so với kích thước khối của tập tin hệ thống?

Câu trả lời là:

- Giảm thiểu chi phí: thời gian tìm kiếm trên đĩa gần như không đổi
- Ví dụ: thời gian tìm kiếm: 10 ms, tốc độ truyền tập tin: 100MB/s, chi phí (thời gian tìm kiếm/thời gian truyền khối) là 1%, *kích thước khối tối ưu là bao nhiêu?*
- 100 MB (HDFS  $\Rightarrow$  128 MB)

# Thao tác đọc



1. Ứng dụng kết nối với NN để đọc dữ liệu
2. NN cho ứng dụng biết nơi tìm các khối dữ liệu
3. Ứng dụng đọc các khối trực tiếp từ các nút dữ liệu (mà không cần thông qua NN)
4. Trong trường hợp nút bị lỗi, ứng dụng kết nối với nút khác đáp ứng khối bị thiếu

Tại sao HDFS lại chọn một thiết kế như vậy để đọc? Tại sao không yêu cầu ứng dụng đọc khối thông qua NN?

Lý do là vì:

- Ngăn việc NN trở thành điểm nghẽn cổ chai
- Cho phép HDFS mở rộng số lượng lớn các ứng dụng truy xuất đồng thời
- Phân bổ lưu lượng dữ liệu trên toàn bộ cụm máy

Một khối có nhiều nhân bản, làm thế nào NN quyết định bản sao ứng dụng nên đọc?

Giải pháp của HDFS là:

- Dựa vào cấu trúc của mạng để quyết định

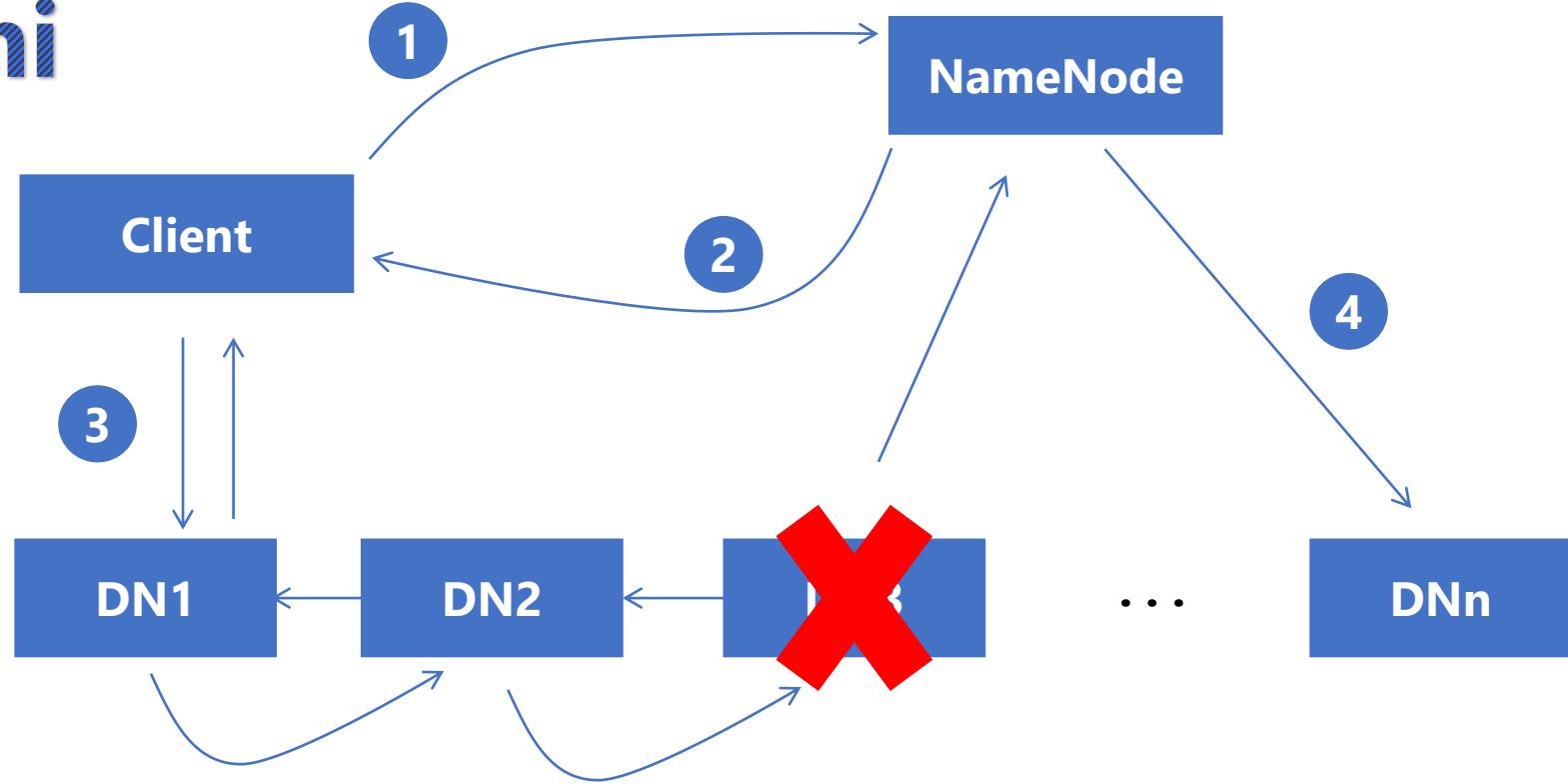
# Cấu trúc mạng

- Tài nguyên quan trọng trong HDFS là **băng thông**
- Việc đo băng thông giữa bất kỳ cặp nút nào quá phức tạp và khó chính xác
- **Thứ tự ưu tiên xử lý:**
  - Xử lý trên cùng một nút
  - Các nút khác nhau trên cùng một rack
  - Các nút trên các rack khác nhau trong cùng một data center (cụm)
  - Các nút trong các data center khác nhau



**Tốn kém  
băng thông  
nhiều hơn**

# Thao tác ghi



1. Ứng dụng kết nối với NN để ghi dữ liệu
2. NN hướng dẫn ứng dụng địa chỉ các nút để ghi dữ liệu
3. Ứng dụng ghi các khối trực tiếp vào các nút với hệ số nhân bản mong muốn
4. Trong trường hợp nút bị lỗi, NN sẽ tìm và sao chép các khối bị thiếu

HDFS nên đặt ba bản sao của một khối ở đâu? Chúng ta cần cân nhắc những đánh đổi nào?

Những điều phải cân nhắc là:

- Độ tin cậy
- Băng thông đọc dữ liệu
- Băng thông ghi dữ liệu

## Nhân bản vs. Những điều phải đánh đổi/cân nhắc

	<b>Độ tin cậy</b>	<b>Băng thông ghi</b>	<b>Băng thông đọc</b>
Đặt tất cả các bản sao trên cùng một nút			
Đặt tất cả các bản sao trên những rack khác nhau			

# Nhân bản khối dữ liệu

- Chiến lược hiện tại
  - Một bản sao trên nút cục bộ
  - Bản sao thứ hai được đặt ở một rack khác
  - Bản sao thứ ba được đặt ở cùng rack với bản sao thứ hai
  - Các bản sao bổ sung được đặt ngẫu nhiên
- Ứng dụng đọc từ bản sao gần nhất

## Nhân bản vs. Những điều phải đánh đổi/cân nhắc

	<b>Độ tin cậy</b>	<b>Băng thông ghi</b>	<b>Băng thông đọc</b>
Đặt tất cả các bản sao trên cùng một nút			
Đặt tất cả các bản sao trên những rack khác nhau			
HDFS			

# JobTracker

Là thành phần chính trong Hadoop chịu trách nhiệm quản lý tài nguyên, lập lịch công việc và quản lý. JobTracker có những nhiệm vụ sau:

- Ứng dụng giao tiếp với JobTracker để gửi hoặc hủy công việc và thăm dò tiến độ công việc.
- JobTracker giao tiếp với NameNode để xác định vị trí của dữ liệu.
- JobTracker tìm các nút TaskTracker tốt nhất để thực thi các tác vụ dựa trên vị trí dữ liệu (gần dữ liệu nhất) và các vị trí có sẵn để thực thi một tác vụ trên một nút nhất định.

- Giám sát các nút TaskTracker và việc sử dụng tài nguyên của chúng (có bao nhiêu tác vụ hiện đang chạy, số lượng các vị trí đang trống, quyết định xem nút TaskTracker có cần được đánh dấu vào danh sách đen hay không, v.v...)
- Giám sát tiến trình của công việc và nếu một công việc không thành công, nó sẽ tự động khởi động lại công việc trên một nút TaskTracker khác
- Lưu giữ lịch sử của các công việc được thực thi trên cụm máy.
- Khi JobTracker không hoạt động, HDFS sẽ vẫn hoạt động nhưng không thể bắt đầu thực thi MapReduce và các công việc MapReduce hiện có sẽ bị tạm dừng.

# TaskTracker

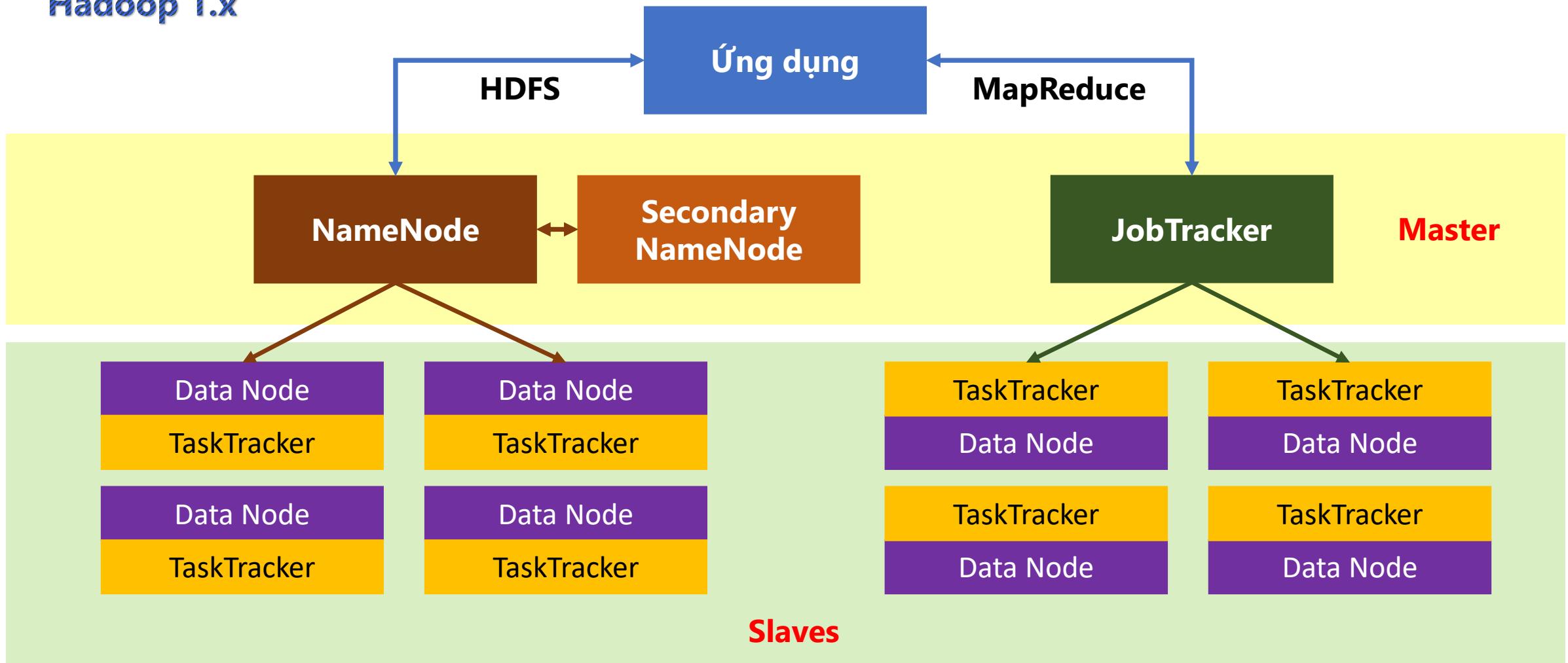
Là một module trên mỗi nút chịu trách nhiệm thực hiện các tác vụ mapreduce. Một nút TaskTracker được định cấu hình để chấp nhận một số (tối đa) tác vụ mapreduce từ JobTracker. Nó có nhiệm vụ như sau:

- Khởi tạo một tiến trình mới trong máy ảo Java để thực hiện mapreduce. Chạy tác vụ trên một máy ảo riêng để đảm bảo nếu lỗi xảy ra không gây hại đến TaskTracker.
- Giám sát các máy ảo Java và cập nhật tình hình định kỳ cho JobTracker.

- Gửi tín hiệu nhịp tim và chỉ số sử dụng tài nguyên hiện tại của nó (các vị trí trống) đến JobTracker định kỳ vài phút.
- Khi một TaskTracker không phản hồi, JobTracker sẽ chỉ định nhiệm vụ được thực thi bởi TaskTracker cho một nút khác

# Kiến trúc

## Hadoop 1.x



# Câu hỏi

Có thể lưu trữ 1 tỷ tập tin trong cụm máy Hadoop 1.x được không?

- A. Có
- B. Không



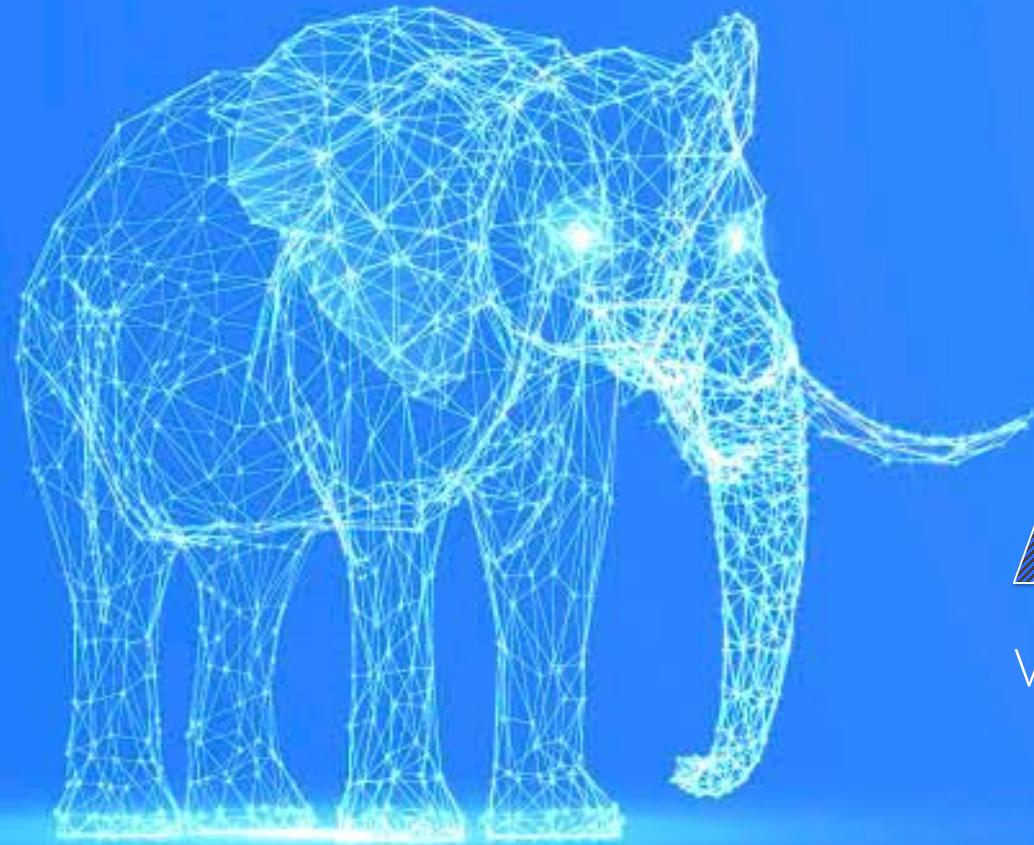
Điều nào sau đây là một nhược điểm đáng kể trong Hadoop 1.x?

- 'Cái chết' của NameNode
- JobTracker quá tải

Có thể sử dụng hàng trăm DataNode cho bất kỳ xử lý nào khác ngoài MapReduce trong Hadoop 1.x không?

- A. Có
- B. Không





# APACHE HADOOP

Version 2.x – Framework

# NỘI DUNG

- Hạn chế của Hadoop 1.x
- Giới thiệu Hadoop 2.x
- HDFS Liên kết
- Kiến trúc của Hadoop 2.x
- YARN
- Các thành phần của YARN

# Hạn chế của Hadoop 1.x

- NameNode không có khả năng mở rộng theo chiều ngang
- Không hỗ trợ NameNode hiệu năng cao
- JobTracker quá tải
- Không thể chạy các ứng dụng dữ liệu lớn Non-MapReduce trên HDFS
- Không hỗ trợ kết nối với những nền tảng xử lý khác

Vấn đề	Mô tả
NameNode – Không thể mở rộng theo chiều ngang	Chỉ hỗ trợ duy nhất một NameNode và một không gian tên giới hạn bởi bộ nhớ RAM.
NameNode – Hiệu suất không cao	NameNode hỏng sẽ gây đình trệ hệ thống, để khôi phục lại cần những thao tác thủ công sử dụng Secondary NameNode.
Job tracker – Quá tải	Mất đáng kể thời gian và công sức để quản lý vòng đời của các ứng dụng.
MRv1 – chỉ hỗ trợ tác vụ Map và Reduce	Lượng dữ liệu khổng lồ được lưu trữ trong HDFS không thể được sử dụng cho các bài toán khác như xử lý đồ thị, v.v..

## Khi kích thước cụm tăng lên và đạt đến 4000 nút

- Hỗn hối nối tiếp
  - Các lỗi DataNode dẫn đến sự suy giảm nghiêm trọng về hiệu suất tổng thể của cụm. Hadoop sẽ nỗ lực sao chép dữ liệu và làm quá tải các nút trực tiếp, do nghẽn mạng.
- Không kết nối được với nhiều nền tảng khác
  - Hadoop 1.x dành các nút của nó cho tác vụ mapreduce. Không thể sử dụng vào các mô hình/nền tảng khác.

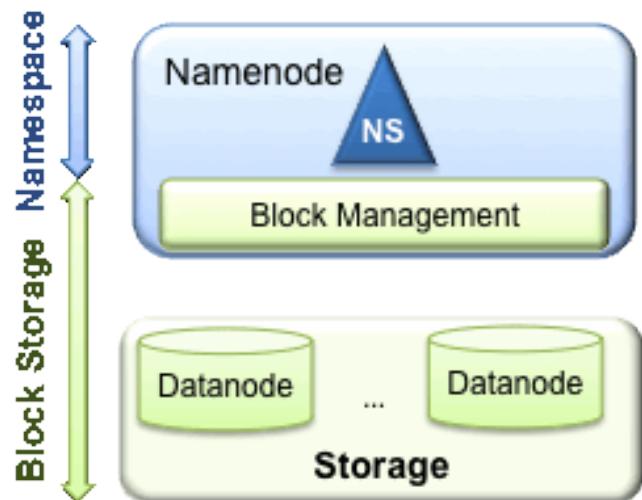
# Hadoop 2.x

Chức năng	Hadoop 1.x	Hadoop 2.x
HDFS Liên kết	Một Namenode và Không gian tên	Nhiều Namenode và Không gian tên
Hiệu quả cao	Không	Tốt
YARN – Điều khiển xử lý và kết nối đa nền tảng	Job Tracker, Task Tracker	Resource Manager, Node Manager, App Master, Capacity Scheduler

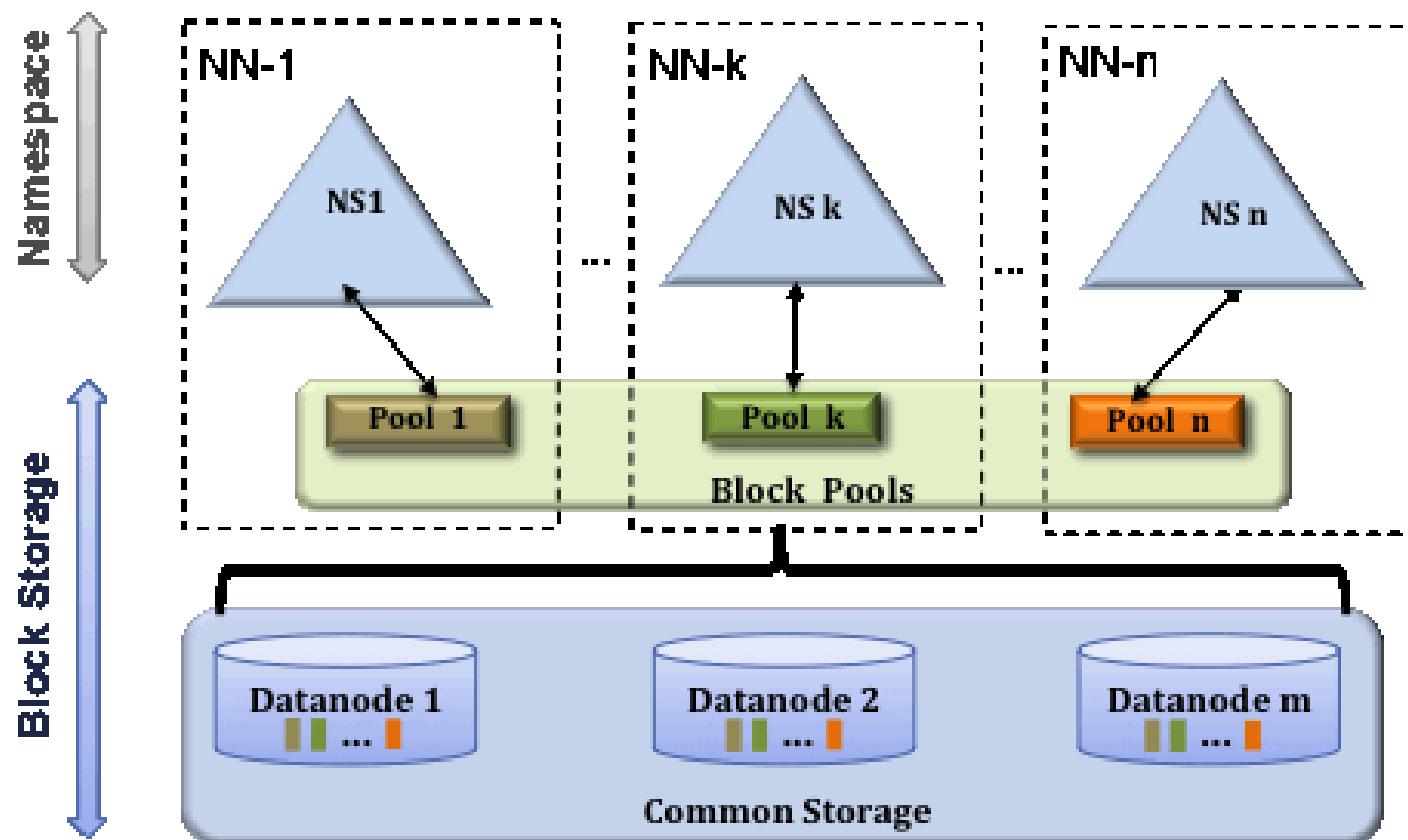
## Những chức năng quan trọng khác của Hadoop 2.x

- Hỗ trợ ảnh sao lưu (snapshot) HDFS
- Hỗ trợ NFSv3 trong HDFS
- Hỗ trợ chạy Hadoop trên MS Windows
- Khả năng tương thích nhị phân cho các ứng dụng MapReduce được xây dựng trên Hadoop 1.0
- Có khả năng kết nối với nhiều dự án (chẳng hạn như PIG, HIVE) trong hệ sinh thái Hadoop

# HDFS Liên kết



**Hadoop 1.x**

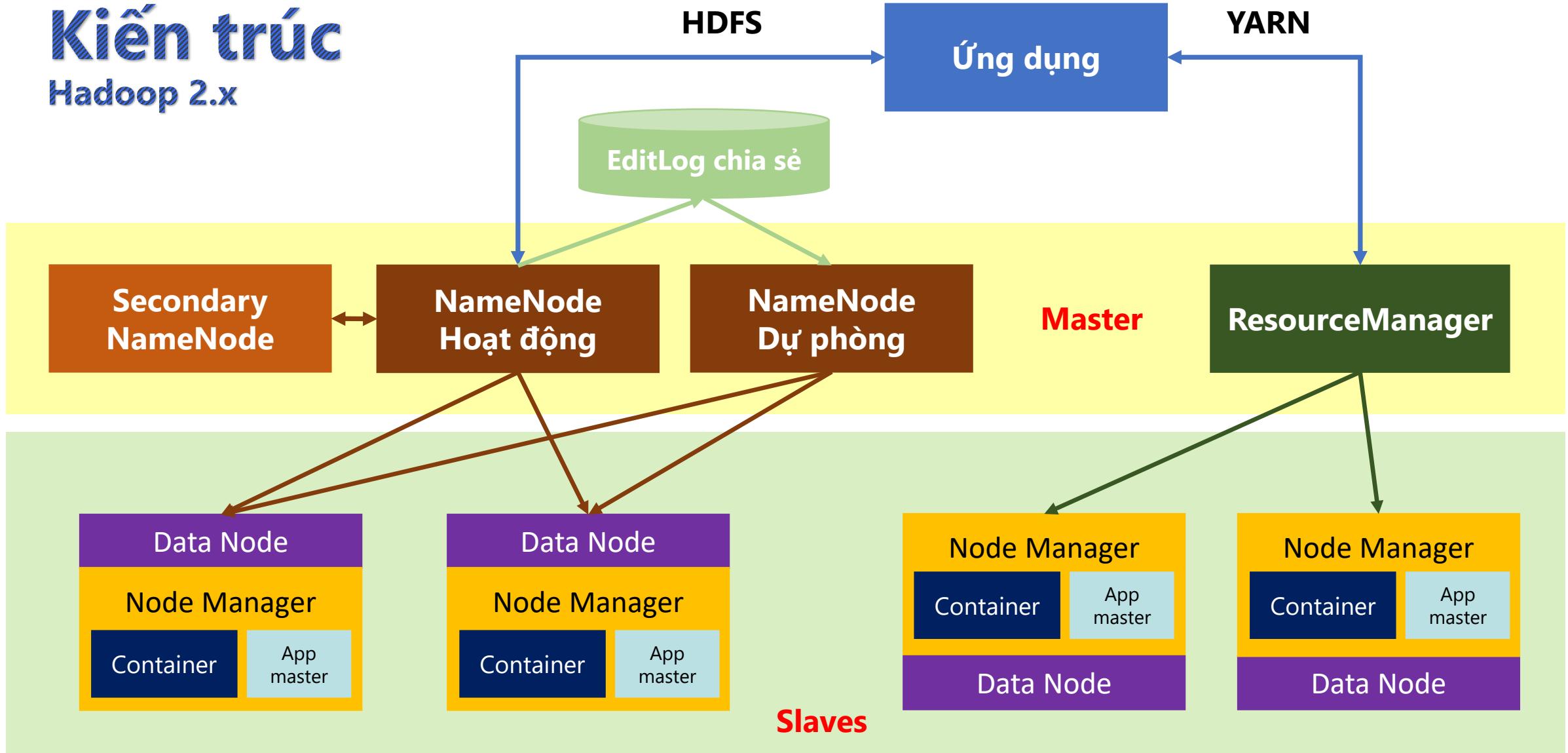


**Hadoop 2.x**

## Để mở rộng thêm HDFS theo chiều ngang

- Liên kết sử dụng nhiều NameNode/Không gian tên độc lập.
- Các NameNode được liên kết với nhau; các NameNode sẽ hoạt động độc lập và không yêu cầu phối hợp với nhau.
- Các DataNode được sử dụng làm nơi lưu trữ chung cho các khối của tất cả các NameNode. Mỗi DataNode đăng ký với tất cả các NameNode trong cụm.
- Các DataNode gửi nhịp tim định kỳ, báo cáo khối và xử lý các yêu cầu từ NameNode.

# Kiến trúc Hadoop 2.x



# **YARN là gì?**

- **YARN (Yet Another Resource Negotiator)** là hệ thống quản lý tài nguyên theo cụm cho Hadoop.
- YARN được giới thiệu trong Hadoop 2.x để cải thiện việc triển khai MapReduce và loại bỏ các nhược điểm quản lý ở Hadoop 1.x.
- YARN có khả năng kết nối các ứng dụng cấp cao (Spark, Hbase v.v...) với môi trường Hadoop cấp thấp.
- Với sự ra đời của YARN, Hadoop đã không còn là một framework MapReduce nữa mà trở thành thành phần cốt lõi xử lý dữ liệu lớn.

# Các thành phần của YARN

- **Resource Manager (RM)** (1 thành phần/cụm) chịu trách nhiệm theo dõi các tài nguyên trong một cụm và lập lịch cho các ứng dụng.
- Trên thực tế, RM phản hồi các yêu cầu tài nguyên từ ApplicationMaster (1 thành phần/ứng dụng Yarn), thông qua yêu cầu tới Node Manager.
- RM không theo dõi và thu thập bất kỳ lịch sử công việc nào.
- RM chỉ chịu trách nhiệm về lập lịch cho cụm.
- Vì mỗi cụm chỉ có duy nhất một RM, nên nếu thành phần này bị lỗi sẽ ảnh hưởng rất lớn đến việc thực hiện các tác vụ. Tuy nhiên, Hadoop 2.x có tính sẵn sàng cao, RM trong trường hợp lỗi sẽ được khôi phục dữ liệu.

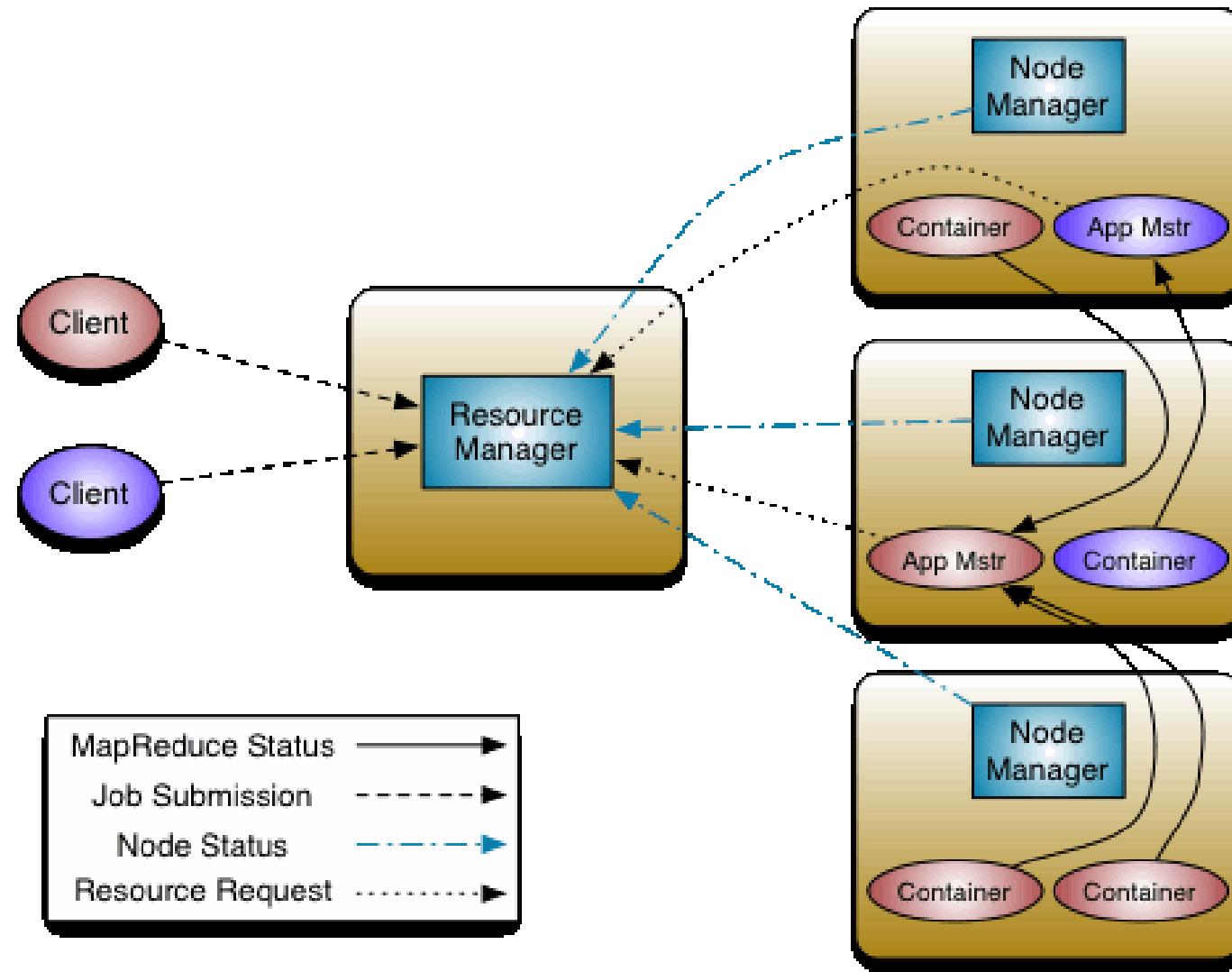
- **Node Manager** (1 thành phần/nút) chịu trách nhiệm giám sát các nút và vùng chứa (container) các tài nguyên như CPU, bộ nhớ, dung lượng đĩa, mạng v.v...
- Nó cũng thu thập dữ liệu nhật ký và báo cáo thông tin đó cho ResourceManager.

- **ApplicationMaster (AM)** chạy trong một tiến trình riêng biệt trên một nút khách (slave).
- Một ứng dụng sẽ có một phiên bản AM, khác với JobTracker, vốn là một thành phần duy nhất chạy trên một nút chủ (master) và theo dõi tiến trình của tất cả các ứng dụng, đó là một điểm yếu.

- ApplicationMaster (AM) chịu trách nhiệm gửi nhịp tim đến ResourceManager chứa thông tin trạng thái của nó và trạng thái nhu cầu tài nguyên của ứng dụng.
- Hadoop 2.x hỗ trợ các tác vụ chạy cùng (uber task) nhẹ có thể được chạy bởi AM trên cùng một nút mà không mất thời gian phân bổ.
- AM nên được khai báo cho từng loại ứng dụng YARN khác nhau, trong trường hợp MapReduce, nó được thiết kế để thực thi các tác vụ map và reduce.

## Các bước để chạy ứng dụng YARN

- Ứng dụng gửi yêu cầu cho ResourceManager để được thực thi.
- ResourceManager yêu cầu NodeManager phân bổ vùng chứa để tạo phiên bản ApplicationMaster trên nút có sẵn (có đủ tài nguyên).
- Khi thành phần ApplicationMaster đã chạy, bản thân nó sẽ gửi các yêu cầu (nhịp tim, tài nguyên ứng dụng cần v.v...) đến ResourceManager và quản lý ứng dụng.



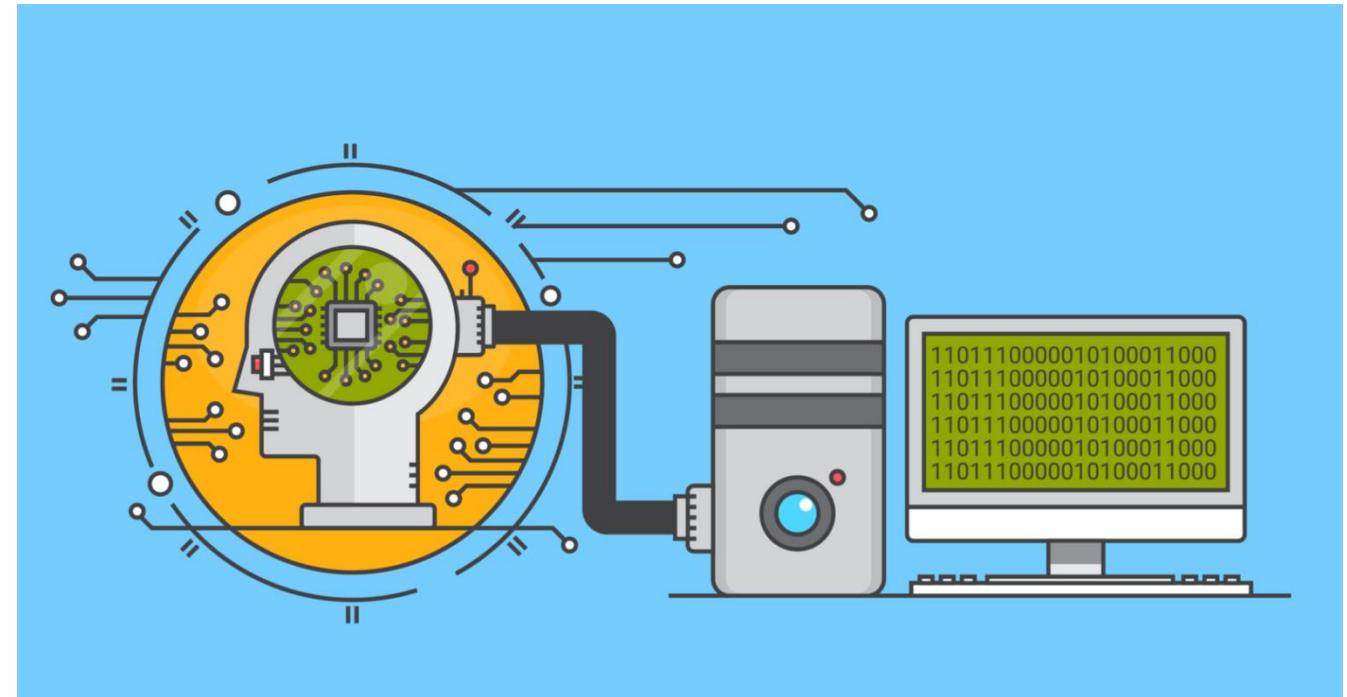
# Câu hỏi

Có thể sử dụng Hadoop 2.x để xử lý các tác vụ thời gian thực không?

- A. Có
- B. Không

# THIẾT KẾ GIẢI THUẬT

Trên mô hình MapReduce



# NỘI DUNG

- Các bước thiết kế giải thuật
- Những vấn đề cơ bản
- Tổng hợp cục bộ
- Cắt và sọc

# Thiết kế giải thuật

Trên mô hình MapReduce

Các bước thiết kế giải thuật bao gồm:

- Chuẩn bị dữ liệu đầu vào
- Thiết kế lớp *mapper* và *reducer*
- Thiết kế lớp *combiner* và *partitioner*, trong trường hợp cần thiết.

## Làm cách nào để viết lại các thuật toán hiện có áp dụng mô hình MapReduce?

- Cách diễn đạt thuật toán không phải lúc nào cũng rõ ràng
- Cấu trúc dữ liệu đóng một vai trò quan trọng
- Khó khăn trong việc tối ưu hóa  
⇒ Người thiết kế cần "uốn cong" framework

## Học thông qua những ví dụ

- Mẫu thiết kế
- Đồng bộ có lẽ là khía cạnh phức tạp nhất

## Những vấn đề không nằm trong tầm kiểm soát của người thiết kế

- *Nơi* một mapper hoặc reducer sẽ chạy
- *Khi nào* mapper hoặc reducer bắt đầu hoặc kết thúc
- Những cặp khóa-giá trị đầu vào *nào* được xử lý bởi một mapper cụ thể
- Những cặp khóa-giá trị trung gian *nào* được xử lý bởi một reducer cụ thể

## Những vấn đề kiểm soát được

- Xây dựng cấu trúc dữ liệu dưới dạng khóa-giá trị
- Có thể chạy những đoạn mã lúc khởi tạo và kết thúc trong quá trình map và reduce.
- Duy trì trạng thái trên nhiều khóa đầu vào và khóa trung gian trong mapper và reducer
- Kiểm soát thứ tự sắp xếp của các khóa trung gian và do đó kiểm soát được thứ tự mà reducer sẽ gặp các khóa cụ thể
- Kiểm soát sự phân vùng của không gian khóa và do đó kiểm soát được tập hợp các khóa sẽ được xử lý bởi một reducer cụ thể

## Ví dụ: Đếm từ cơ bản

```
class MAPPER
```

```
    method MAP(docid a, doc d)
        for all term t  $\in$  doc d do
            EMIT(term t, count 1)
```

```
class REDUCER
```

```
    method REDUCE(term t, counts[c1, c2, ...])
        sum  $\leftarrow$  0
        for all count c  $\in$  counts[c1, c2, ...] do
            sum  $\leftarrow$  sum + c
        EMIT(term t, count s)
```

## Các thuật toán MapReduce có thể phức tạp

- Nhiều thuật toán không thể được chuyển đổi dễ dàng thành một công việc MapReduce đơn (a single MR job).
- Phân tích các thuật toán phức tạp thành một chuỗi công việc
  - Yêu cầu thiết kế dữ liệu để đầu ra của một công việc trở thành đầu vào cho công việc tiếp theo
  - Các thuật toán lặp lại yêu cầu hàm điều khiển bên ngoài để kiểm tra độ hội tụ của kết quả.

## Mẫu thiết kế cơ bản

- Tổng hợp cục bộ (local aggregation)
- Cặp và sọc (pairs and stripes)
- Đảo ngược thứ tự (order inversion)

# Tổng hợp cục bộ

- Trong bối cảnh xử lý phân tán số lượng lớn dữ liệu, khía cạnh quan trọng nhất của đồng bộ hóa là **trao đổi các kết quả trung gian**
  - Điều này liên quan đến việc sao chép các kết quả trung gian từ các tiến trình tạo ra chúng sang các tiến trình sử dụng chúng
  - Trước hết, điều này liên quan đến việc truyền dữ liệu qua mạng
  - Trong Hadoop, tốc độ đọc/ghi của đĩa cũng góp phần ảnh hưởng, vì các kết quả trung gian được ghi vào đĩa

- Chi phí để giảm độ trễ mạng và đĩa rất đắt
  - Giảm lượng dữ liệu trung gian sẽ làm tăng hiệu quả thuật toán
- Combiner và duy trì trạng thái trên các đầu vào
  - Sẽ làm giảm số lượng và kích thước của các cặp khóa-giá trị đưa vào quá trình xáo trộn

## Combiner trong mapper

- Combiner trong mapper, giống như giai đoạn combiner
  - Hadoop không chắc chắn các combiner được thực thi
  - Combiner có thể tốn kém về CPU và I/O
- Sử dụng một mảng liên kết để tích lũy các kết quả trung gian
  - Mảng được sử dụng để kiểm đếm số lượng từ trong một "tài liệu" duy nhất
  - Phương thức Emit chỉ được gọi sau khi tất cả các InputRecords đã được xử lý

## Ví dụ: Đếm từ với Combiner trong Mapper

```
class MAPPER
    method MAP(docid a, doc d)
        H ← new ASSOCIATIVEARRAY
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            EMIT(term t, count H{t})
```

- Thực hiện ý tưởng thêm một bước nữa
  - Khai thác các đặc điểm chi tiết của Hadoop
  - Một đối tượng mapper sẽ được tạo ra ứng với mỗi tác vụ map
- Duy trì trạng thái trong và trên toàn bộ các lệnh gọi tới phương thức map
  - Phương thức Initialize, được sử dụng để tạo cấu trúc dữ liệu sử dụng xuyên suốt toàn bộ các map
  - Phương thức Close, được sử dụng để Emit các cặp khóa-giá trị trung gian chỉ khi tất cả tác vụ map trên một máy được hoàn thành.

## Ví dụ: Đếm từ với Combiner trong Mapper

```
class MAPPER
    method INITIALIZE
        H ← new ASSOCIATIVEARRAY
    method MAP(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
    method CLOSE
        for all term t ∈ H do
            EMIT(term t, count H{t})
```

- “Mẫu thiết kế” đầu tiên, combine trong bộ nhớ
  - Thực hiện việc tổng hợp cục bộ, có khả năng kiểm soát
  - Lập trình viên có thể xác định chính xác cách thực hiện việc tổng hợp
- Hiệu quả so với combiner
  - Không phát sinh thêm chi phí khi xử lý các cặp khóa-giá trị
    - Không tạo và phá hủy các đối tượng không cần thiết (dọn rác cũng cần thời gian)
    - Hạn chế ghi đối tượng xuống đĩa/khôi phục lại khi bộ nhớ bị giới hạn
  - Với combiner, mapper vẫn cần đưa ra tất cả các cặp khóa-giá trị; combiner “chỉ” làm giảm lượng dữ liệu trung gian cần xáo trộn.

- Chú ý
  - Combine trong bộ nhớ phá vỡ mô hình lập trình hàm do duy trì các trạng thái
  - Duy trì trạng thái trong nhiều trường hợp dẫn đến ảnh hưởng kết quả của thuật toán do sự can thiệp vào thứ tự dữ liệu
    - Hoạt động tốt với các thao tác có tính chất giao hoán/kết hợp
    - Mặc khác, những lỗi phụ thuộc vào thứ tự rất khó tìm thấy (với dữ liệu lớn)
- Dung lượng bộ nhớ có hạn
  - Việc combine trong bộ nhớ hoàn toàn phụ thuộc vào việc có đủ bộ nhớ để lưu trữ các kết quả trung gian
  - Một giải pháp khả thi: "block" và "flush"

**Ví dụ:** Tính giá trị trung bình cơ bản

Giả sử, cho một tập dữ liệu lớn gồm khóa là các chuỗi và giá trị là các số nguyên. Yêu cầu là tính trung bình các giá trị có cùng khóa.

```
class MAPPER
    method MAP(string t, integer r)
        EMIT(string t, integer r)

class REDUCER
    method REDUCE(string t, integers[r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers[r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        ravg ← sum/cnt
        EMIT(string t, integer ravg)
```

Bổ sung  
thêm quá  
trình  
combine

```
class COMBINER
    method COMBINE(string t, integers[r1, r2, ...])
        sum  $\leftarrow$  0
        cnt  $\leftarrow$  0
        for all integer r  $\in$  integers[r1, r2, ...] do
            sum  $\leftarrow$  sum + r
            cnt  $\leftarrow$  cnt + 1
        EMIT(string t, pair (sum, cnt))
```

```
class REDUCER
    method REDUCE(string t, pairs[(s1, c1), (s2, c2), ...])
        sum  $\leftarrow$  0
        cnt  $\leftarrow$  0
        for all pair (s, c)  $\in$  pairs[(s1, c1), (s2, c2), ...] do
            sum  $\leftarrow$  sum + s
            cnt  $\leftarrow$  cnt + c
        ravg  $\leftarrow$  sum/cnt
        EMIT(string t, integer ravg)
```



## Điều chỉnh class MAPPER, COMBINER

```
class MAPPER
    method MAP(string t, integer r)
        EMIT(string t, pair (r, 1))

class COMBINER
    method COMBINE(string t, pairs[(s1, c1), (s2, c2), ...])
        sum ← 0
        cnt ← 0
        for all pair (s, c) ∈ pairs[(s1, c1), (s2, c2), ...] do
            sum ← sum + s
            cnt ← cnt + c
        EMIT(string t, pair (sum, cnt))
```



# Cặp và sọc

- Một cách tiếp cận phổ biến trong MapReduce: xây dựng các khóa-giá trị phức tạp
  - Sử dụng khuôn khổ để nhóm dữ liệu lại với nhau
- Hai kỹ thuật cơ bản:
  - Cặp (Pairs): xây dựng cấu trúc dữ liệu phức tạp hơn để làm khóa
  - Sọc (Stripes): sử dụng cấu trúc dữ liệu bộ nhớ phức tạp trong mapper
- Tận dụng ưu điểm từ hai phương pháp này, để giải quyết vấn đề cụ thể trong ví dụ sau.

- Vấn đề: xây dựng ma trận từ đồng hiện cho kho ngũ liệu lớn
  - Ma trận từ đồng hiện của một kho ngũ liệu là ma trận vuông  $n \times n$ ,  $M$
  - $n$  là số lượng từ duy nhất (là số lượng từ vựng)
  - Một ô  $m_{ij}$  chứa số lần từ  $w_i$  đồng xuất hiện với từ  $w_j$  trong một ngũ cảnh cụ thể
    - Ngũ cảnh: một câu, một đoạn văn, một tài liệu hoặc một cửa sổ gồm  $t$  từ
  - LƯU Ý: ma trận có thể đối xứng trong một số trường hợp
- Động lực
  - Vấn đề này là một bước trong những thuật toán phức tạp hơn
  - Ước tính sự phân bố của các sự kiện rời rạc từ một số lượng lớn các quan sát
  - Vấn đề tương tự trong các lĩnh vực khác:
    - Khách hàng mua mặt hàng này cũng có xu hướng mua mặt hàng kia

- Yêu cầu về không gian
  - Yêu cầu về không gian là  $O(n^2)$ , trong đó  $n$  là kích thước của tập từ
  - Trong thực tế tiếng Việt, kho ngữ liệu có thể có hàng trăm nghìn từ, hoặc thậm chí hàng tỷ từ trong một số trường hợp cụ thể ( $n$  rất lớn)
- Vậy vấn đề là gì?
  - Nếu ma trận có thể nằm gọn trong bộ nhớ của một máy, thì chỉ cần sử dụng bất kỳ phương pháp đơn giản nào
  - Thay vào đó, nếu ma trận lớn hơn bộ nhớ khả dụng, thì bộ nhớ bị tràn, thao tác phân trang (paging) sẽ tự động bắt đầu và thuật toán đơn giản sẽ bị lỗi.
- Kỹ thuật nén
  - Những kỹ thuật như vậy có thể giúp giải quyết vấn đề trên một máy duy nhất
  - Tuy nhiên, nếu có những yêu cầu về khả năng mở rộng?

## Cặp

- Đầu vào là định danh (docid a) và câu/đoạn/văn bản (doc d)
- Mỗi mapper thao tác với một câu/đoạn/văn bản:
  - Tạo ra tất cả các cặp từ đồng hiện
  - Với tất cả các cặp, EMIT (a, b) → count
- Reducers tổng hợp tần số đồng hiện của từng cặp từ
- Khóa-giá trị cuối cùng ở đầu ra sẽ là cặp từ và tần suất đồng hiện
  - Tương ứng với một ô trong ma trận từ đồng hiện
- Sử dụng combiners!

```
class MAPPER
    method MAP(docid a, doc d)
        for all term w ∈ doc d do
            for all term u ∈ NEIGHBORS(w) do
                EMIT(pair (w, u), count 1)

class REDUCER
    method REDUCE(pair p, counts[c1, c2, ...])
        s ← 0
        for all count c ∈ counts[c1, c2, ...] do
            s ← s + c
        EMIT(pair p, count s)
```

## Sơc

- Ý tưởng: nhóm các cặp lại với nhau thành một mảng kết hợp

(a, b) → 1

(a, c) → 2

(a, d) → 5      a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

(a, e) → 3

(a, f) → 2

- Mỗi mapper thao tác với một câu/đoạn/văn bản:

- Tạo ra tất cả các cặp từ đồng hiện

- Với mỗi từ, EMIT a → { b: count<sub>b</sub>, c: count<sub>c</sub>, d: count<sub>d</sub> ... }

- Reducer thực hiện tính tổng theo phần tử của các mảng kết hợp

a → {b: 1,                  d: 5, e: 3}

$$\begin{array}{r} + \\ a \rightarrow \{b: 1, c: 2, d: 2, \quad \quad \quad f: 2\} \\ \hline a \rightarrow \{b: 2, c: 2, d: 7, e: 3, f: 2\} \end{array}$$

```

class MAPPER
    method MAP(docid a, doc d)
        for all term w ∈ doc d do
            H ← new ASSOCIATIVEARRAY
            for all term u ∈ NEIGHBORS(w) do
                H{u} ← H{u} + 1
            EMIT(term w, stripe H)

```

```

class REDUCER
    method REDUCE(term w, stripes [H1, H2, ...])
        Hf ← new ASSOCIATIVEARRAY
        for all stripe H ∈ stripes[H1, H2, ...] do
            SUM(Hf, H)
        EMIT(term w, stripe Hf)

```

## So sánh hai phương pháp

- Phương pháp cặp
  - Tạo ra số lượng lớn các cặp khóa-giá trị
    - Đặc biệt, những cặp khóa-giá trị trung gian được gửi qua mạng
  - Lợi ích từ các combiner bị hạn chế, vì khóa tạo ra từ quá trình mapper là một cặp từ ít khi lặp lại cục bộ.
  - Không bị các vấn đề tràn bộ nhớ

## So sánh hai phương pháp

- Phương pháp sọc
  - Nhỏ gọn hơn
  - Tạo ít khóa trung gian hơn và ngắn hơn
    - Ít phải xử lý sắp xếp hơn
  - Các giá trị phức tạp hơn và tốn chi phí ghi đổi tương xuống đĩa cứng/khôi phục trạng thái
  - Rất nhiều lợi ích từ các combiner, vì tập khóa chính là tập từ vựng
  - Gặp phải sự cố tràn bộ nhớ, nếu không được thiết kế đúng cách

# Bài tập

Thiết kế thuật toán  $k$ -Means trong MapReduce với những yêu cầu sau:

- Chỉ 1 lần lặp
- Các điểm trong không gian 2 chiều
- Mỗi cặp tọa độ nằm trên 1 dòng trong tập tin đầu vào
- Số cụm  $k = 2$
- Tâm cụm ban đầu đặt trong các biến ở cả Mapper và Reducer đều đọc được.

# Bài tập về nhà

Thiết kế thuật toán Apriori trong MapReduce.

# 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. J. Lin and C. Dyer. 2010. **Data-Intensive Text Processing With MapReduce**. San Rafael, CA: Morgan & Claypool Publishers. doi: 10.2200/S00274ED1V01Y201006HLT007

Q & A