## Parallel

* Với mô hình bộ nhớ chia sẻ, tất cả các xử lý đều truy cập một dữ liệu chung thông qua một vùng nhớ dùng chung.
* Với mô hình truyền thông điệp thì mỗi xử lý đều có riêng một bộ nhớ cục bộ của nó, các xử lý trao đổi dữ liệu với nhau thông qua hai phương thức gửi và nhận thông điệp.

## SQL:

Dùng cho các mô hình quản lý dữ liệu có cấu trúc, cần sự chuẩn mực và chính xác về kiểu dữ liệu và ràng buộc quan hệ giữa các thực thể. Ví dụ, dữ liệu ngân hàng, đặt vé tàu, vé phim có lượng giao dịch và truy cập đồng thời cao, SQL truyền thống giúp cho mô hình ứng dụng này được đảm bảo tính chính xác trong quá trình thực thi.

- Tính toàn vẹn: Các giá trị dữ liệu lưu trữ trong cơ sở dữ liệu phải thỏa mãn một số ràng buộc

- Tính nhất quán: Trong quá trình cập nhật dữ liệu, dữ liệu trong cơ sở dữ liệu phải được đảm bảo đúng đắn ngay cả khi có sự cố.

NoSQL: Not only SQL nghĩa là vẫn có SQL trong đó nhưng sẽ có thêm các tính năng giải quyết vấn đề của Big Data như lưu trữ tốc độ cao, dễ dàng nhân bản hay phân tán dữ liệu, schema dữ liệu không rõ ràng và thay đổi thường xuyên, mô hình hoá được cho các bài toán lưu trữ document, phân tích mạng xã hội, và hỗ trợ tìm kiếm thông tin nâng cao hơn SQL truyền thống.

Kết hợp: SQL thường dùng khi use case đã rõ ràng ít thay đổi trong quá trình triển khai. NoSQL thường dùng khi tính năng sản phẩm thay đổi nhiều (tính theo ngày, tuần).

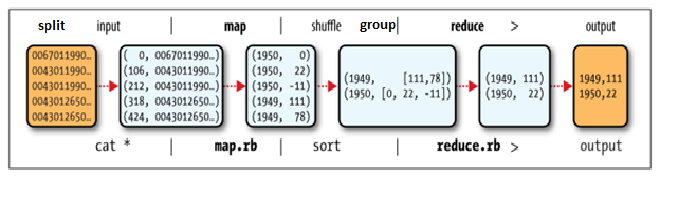
Hadoop frameword gồm hai layer chính:

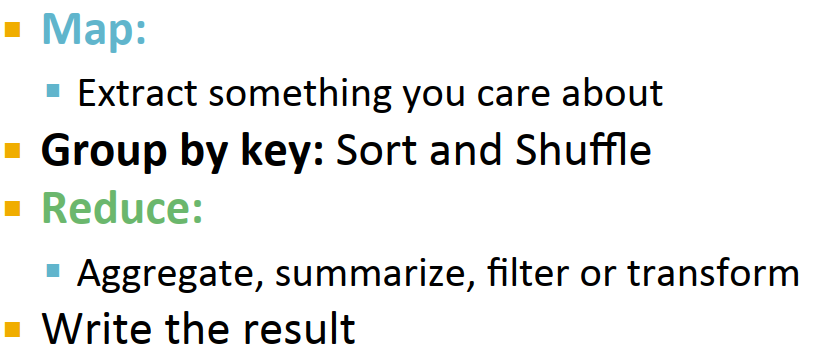
* Hadoop Distriduted file system (HDFS):
  + **HDFS** được thiết kế và tối ưu hóa để lưu trữ các tập tin rất lớn và với một mô hình truy cập trực tuyến. Kể từ khi nó được dự kiến sẽ chạy trên phần cứng thương mại, nó được thiết kế để tính đến và xử lý sự cố trên máy cá nhân. HDFS thông thường không lưu trữ dữ liệu chính. Thay vào đó, trong một việc điển hình, dữ liệu được sao chép sang HDFS cho mục đích thực hiện MapReduce, và kết quả sau đó sẽ được sao chép sang HDFS. Từ khi HDFS được tối ưu hóa cho các luồng truy cập thông tin lớn, truy cập ngẫu nhiên vào các thành phần của các tập tin thì tốn kém hơn truy cập tuần tự rõ ràng, và nó cũng không hỗ trợ cập nhật các tập tin, chỉ có thể gắn thêm. Các kịch bản điển hình của các ứng dụng sử dụng HDFS sau mô hình truy cập ghi một lần đọc nhiều lần
  + Các tập tin trong HDFS được chia thành một số khối lớn (thường là 64Mb) và được lưu trong Data Node. Một tập tin điển hình được phân phối trên một số DataNodes nhằm thúc đẩy băng thông cao và xử lý song song. Để nâng cao độ tin cậy, các khối dữ liệu trong HDFS (data blocks) được nhân bản và lưu trỡ trong 3 máy, với một trong những bản sao trong mỗi giá khác nhau để tăng sự sẵn có hơn nữa của dữ liệu. Việc duy trì các tập tin siêu dữ liệu được xử lý bởi một NameNode riêng biệt. Siêu dữ liệu này bao gồm ánh xạ từ tập tin vào khối và vị trí của khối (DataNode). NameNode theo định kì sẽ truyền đạt siêu dữ liệu của nó cho Secondary NameNode khi cần cấu hình lại để làm nhiệm của NameNode khi xảy ra trường hợp lỗi.
* Execution engine (MapReduce):
  + Trong Hadoop, JobTracker (công việc theo dõi) là một điển truy cập cho clients. Nhiệm vụ của JobTracker là đảm báo kế hoạch và lập kế hoạch công việc sắp tới của MR, và giao nhiệm vụ cho các TaskTrackers mà nó có trách nhiệm thực hiện. Một TaskTracker có thể chạy một số của nhiệm vụ phụ thuộc trong các nguồn lực có sẵn (một ví dụ 2 nhiệm vụ Map và 2 nhiệm vụ reduce) và sẽ phân bố cho các nhiệm vụ mới bằng JogTracker khi nó sẵn sàng. Các kích thước tương đối nhỏ của từng công việc so với số lượng lớn các nhiệm vụ trong tổng số giúp đảm bảo cân bằng tải giữa các máy. Cần lưu ý rằng trong khi số lượng công việc Map được thực hiện dựa trên kích thước dầu vào của dữ liệu (số lượng phân chia) thì số lượng công việc REDUCE là do người dùng chỉ định.
  + Trong một cụm lớn, việc các máy xảy ra sự cố là thường xuyên, và để xử lý vấn đề này, thông điệp này thường được gửi từ TaskTracker tới JobTracker định kì và từ công việc Map tới công việc REDUCE. Bằng cách này, lỗi có thể được phát hiện và JobTracker có thể sắp xếp lại những nhiệm vụ lỗi cho các TaskTracker khác. Hadoop theo một mô hình thực hiện suy đoán để xử lý sự cố. Thay vì cố định một nhiệm vụ thất bại hoặc chạy chậm, nó thực hiện một nhiệm vụ tương đương mới như sao lưu. Thất bại của JobTracker tự nó không thể được xử lý tự động, nhưng xác suất thất bại của một máy cụ thể là thấp để điều này không nên trình bày một vấn đề nói chung.

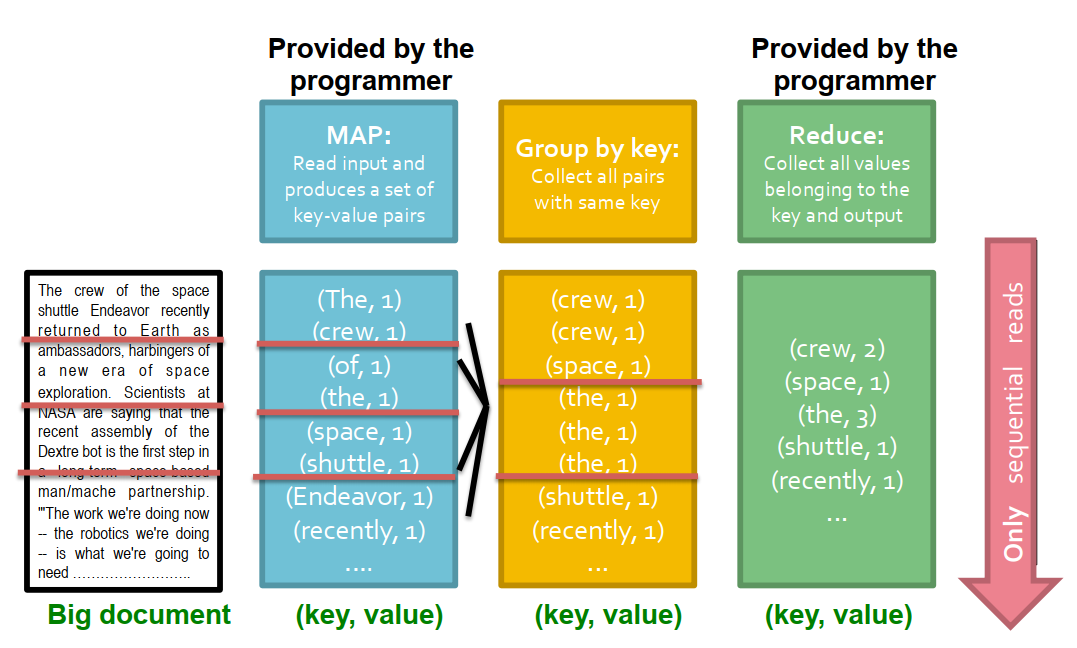
Nguyên tắc thiết kế của Hadoop:

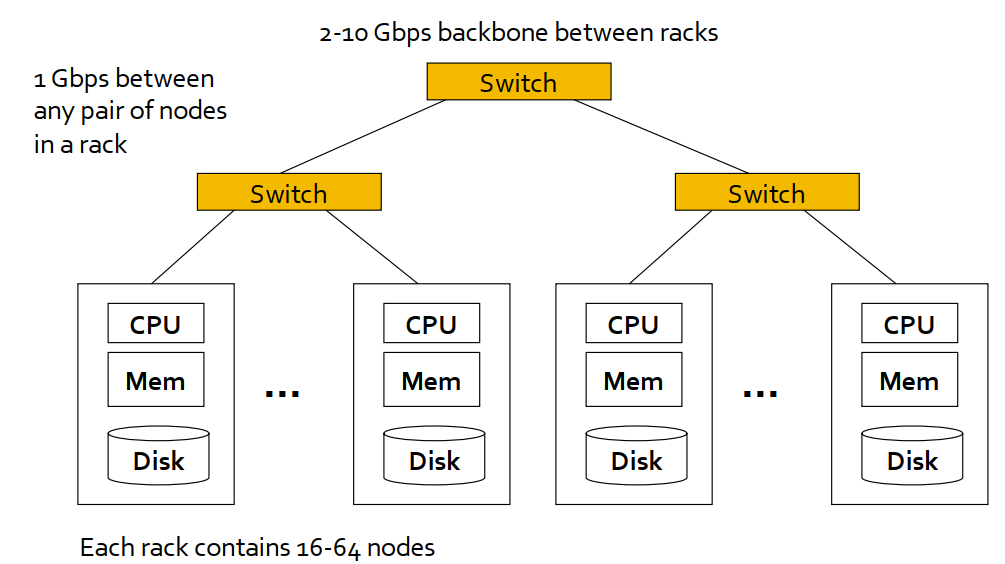
* Cần xử lý dữ liệu lớn
* Cần tính toán song song trên hàng ngàn node
* Chịu lỗi và phục hồi tự động

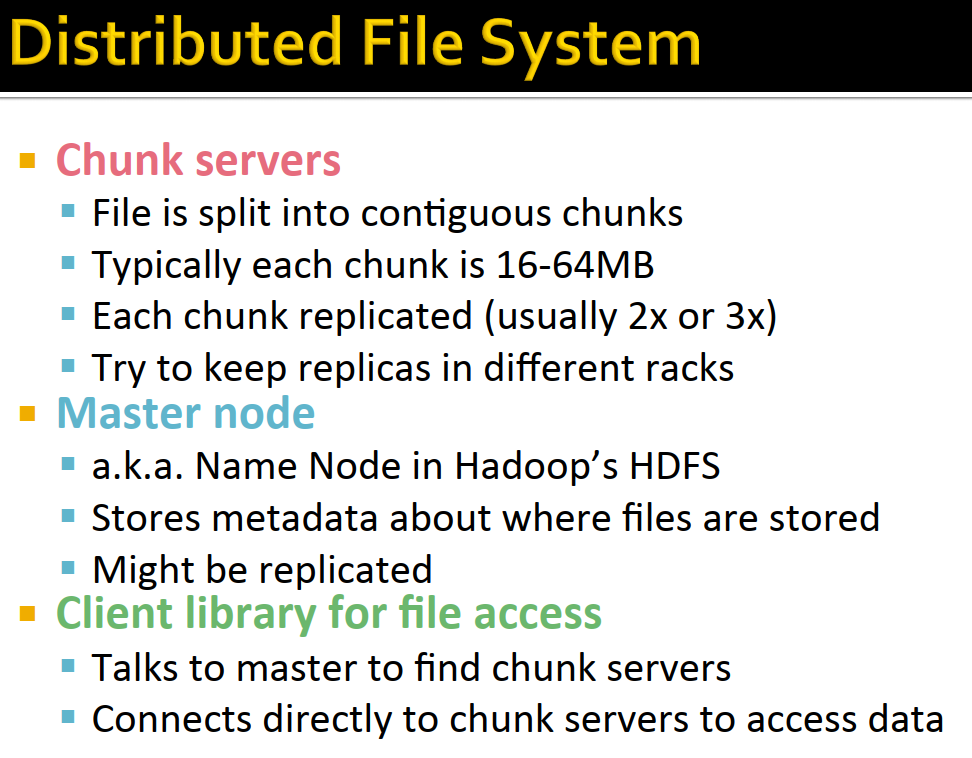
1. HBase: Thành phần đầu tiên cung cấp truy cập trực tuyến là HBase, một kho lưu trữ key-value sử dụng HDFS cho lưu trữ cơ bản của nó. HBase cung cấp cả truy cập đọc / ghi trực tuyến các hàng và các hoạt động hàng loạt chia đều để đọc và ghi dữ liệu hàng loạt, làm cho nó trở thành một giải pháp tốt để xây dựng các ứng dụng trên.
2. YARN: YARN (Yet Another Resource Negotiator) trong Hadoop 2. YARN là một hệ thống quản lý tài nguyên cụm, cho phép bất kỳ chương trình phân tán nào (không chỉ MapReduce) chạy trên dữ liệu trong cụm Hadoop.

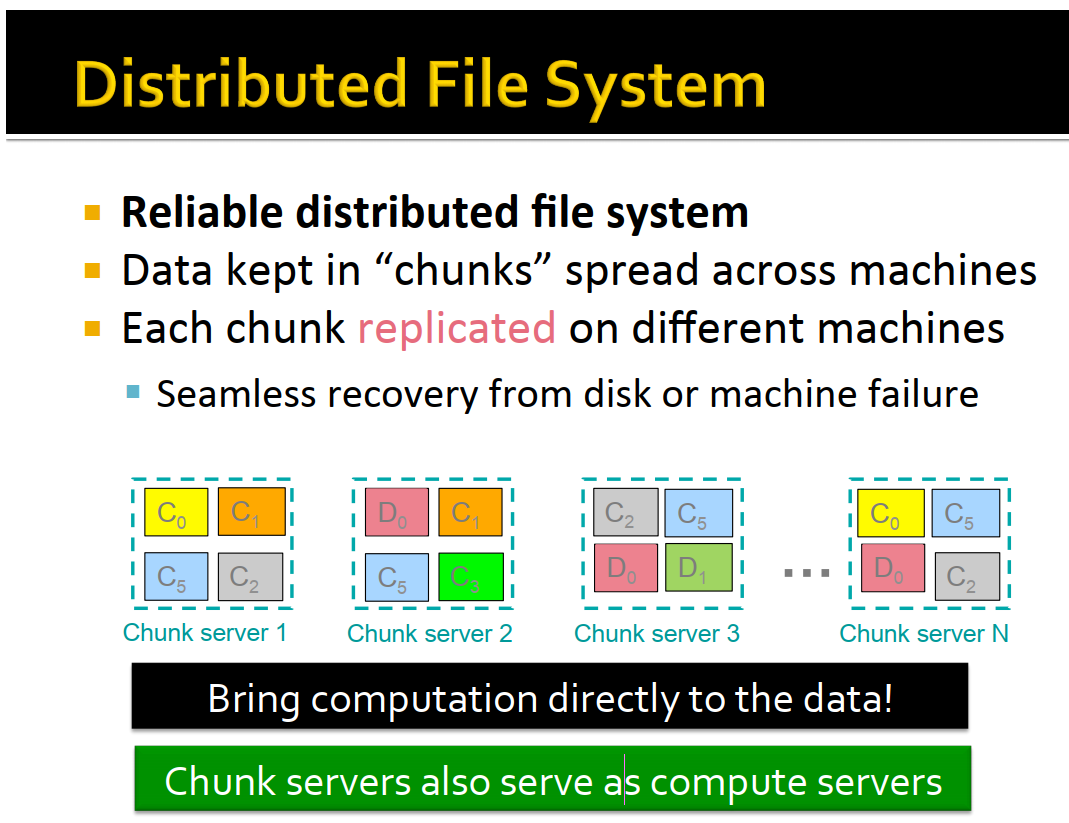


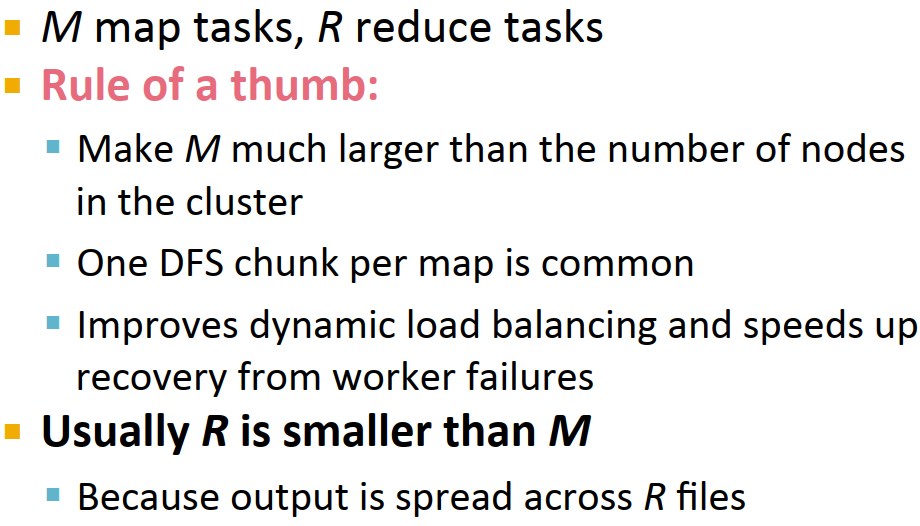




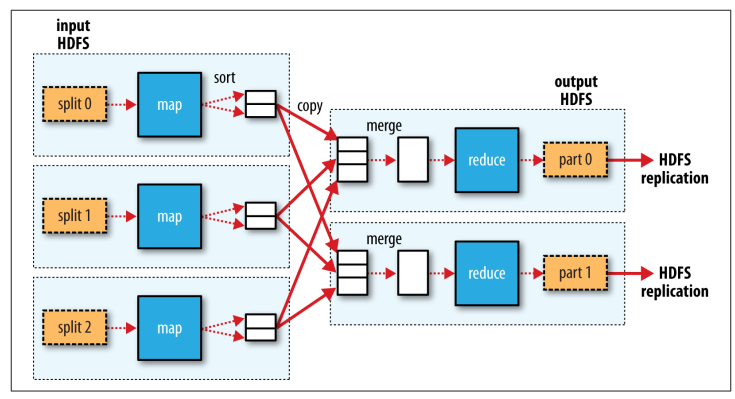






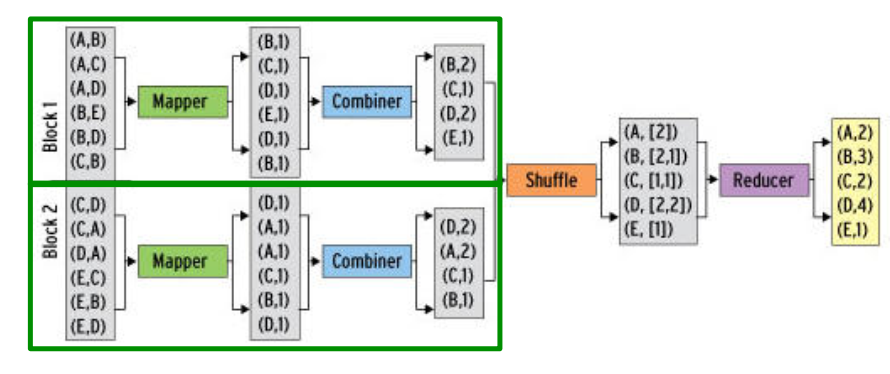


1. Map tasks ghi đầu ra của chúng vào đĩa cục bộ, không ghi vào HDFS. Do đầu ra của map là đầu ra trung gian: nó được xử lý bằng reduce tasks để tạo ra kết quả cuối cùng và khi công việc hoàn tất, đầu ra của map có thể bị vứt bỏ. Vì vậy, lưu trữ nó trong HDFS với nhân rộng sẽ là quá mức cần thiết. Nếu nút chạy tác vụ map không thành công trước khi đầu ra map được tiêu thụ bởi tác vụ reduce, thì Hadoop sẽ tự động chạy lại tác vụ map trên một nút khác để tạo lại map đầu ra



1. Combiner Function

Nhiều công việc MapReduce bị giới hạn bởi băng thông có sẵn trên cụm, do đó, nó trả tiền để giảm thiểu dữ liệu được truyền giữa các tác vụ bản đồ và giảm. Hadoop cho phép người dùng chỉ định một *Combiner function* để chạy trên đầu ra bản đồ và đầu ra của *Combiner function* tạo thành đầu vào cho hàm reduce. Bởi vì *Combiner function* là một tối ưu hóa, Hadoop không đảm bảo bao nhiêu lần nó sẽ gọi nó cho một bản ghi đầu ra bản đồ cụ thể, nếu có. Nói cách khác, gọi hàm tổ hợp số không, một hoặc nhiều lần sẽ tạo ra cùng một đầu ra từ reducer.



Chức năng kết hợp không thay thế chức năng giảm (ví dụ như tính trung bình values). Chức năng giảm vẫn cần thiết để xử lý các bản ghi có cùng khóa từ các bản đồ khác nhau. Nhưng nó có thể giúp giảm số lượng dữ liệu xáo trộn giữa những người lập bản đồ và bộ giảm tốc, và vì lý do này, bạn nên cân nhắc xem liệu bạn có thể sử dụng chức năng kết hợp trong công việc MapReduce của mình hay không.



1. TaskTracker quản lý việc thực hiện các nhiệm vụ riêng (map task, reduce task) trên mỗi node. JobTracker quản lý các TaskTracker, quyết định kế hoạch thực hiện bằng cách xác định những tập tin nào sẽ xử lý, các nút được giao các nhiệm vụ khác nhau, và theo dõi tất cả các nhiệm vụ khi chúng đang chạy.

Một trong những trách nhiệm của các TaskTracker là liên tục liên lạc với JobTracker. Nếu JobTracker không nhận được nhịp đập từ một TaskTracker trong vòng một lượng thời gian đã quy định, nó sẽ cho rằng TaskTracker đã bị treo (cashed) và sẽ gửi lại nhiệm vụ tương ứng cho các nút khác trong cluster.

1. Hadoop sử dụng một kiển trúc master/slave.

Khi ghi một file HDFS, file đó được chia nhỏ thành các khối lưu trữ trong các node do DataNode quản lý.

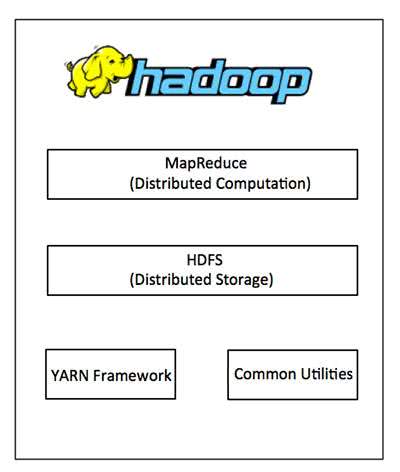
NameNode là master của HDFS để chỉ đạo các trình nền DataNode slave để thực hiện các nhiệm vụ I/O mức thấp. NameNode theo dõi HDFS, cách các tập tin của bạn được phân chia thành các block, những node nào lưu các khối đó, và “kiểm tra sức khỏe” tổng thể của hệ thống tệp phân tán

**Kiến trúc Hadoop**

Hadoop framework gồm 4 module:

* **Hadoop Common**: Đây là các thư viện và tiện ích cần thiết của Java để các module khác sử dụng. Những thư viện này cung cấp hệ thống file và lớp OS trừu tượng, đồng thời chứa các mã lệnh Java để khởi động Hadoop.
* **Hadoop YARN**: Đây là framework để quản lý tiến trình và tài nguyên của các cluster.
* **Hadoop Distributed File System (HDFS)**: Đây là hệ thống file phân tán cung cấp truy cập thông lượng cao cho ứng dụng khai thác dữ liệu.
* **Hadoop MapReduce**: Đây là hệ thống dựa trên YARN dùng để xử lý song song các tập dữ liệu lớn.

Có thể sử dụng sơ đồ sau để mô tả bốn thành phần có trong Hadoop framework.



Kể từ 2012, thuật ngữ “Hadoop” không chỉ đề cập đến các module cơ sở nêu trên mà còn đề cập đến các gói phần mềm mở rộng (additional software packages) có thể được cài đặt bên cạnh Hadoop, chẳng hạn như Apache Pig, Apache Hive, Apache HBase, Apache Spark.

**MapReduce**

Hadoop MapReduce là một framework dùng để viết các ứng dụng xử lý song song một lượng lớn dữ liệu có khả năng chịu lỗi cao xuyên suốt hàng ngàn cụm máy tính.

Thuật ngữ MapReduce liên quan đến hai tác vụ mà chương trình Hadoop thực hiện:

* **Map**: đây là tác vụ đầu tiên, trong đó dữ liệu đầu vào được chuyển đổi thành tập dữ liệu theo cặp key/value.
* **Reduce**: tác vụ này nhận kết quả đầu ra từ tác vụ Map, kết hợp dữ liệu lại với nhau thành tập dữ liệu nhỏ hơn.

Thông thường, kết quả input và output được lưu trong hệ thống file. Framework này sẽ tự động quản lý, theo dõi và tái thực thi các tác vụ bị lỗi.

MapReduce framework gồm một single master (máy chủ) JobTracker và các slave (máy trạm) TaskTracker trên mỗi cluster-node. Master có nhiệm vụ quản lý tài nguyên, theo dõi quá trình tiêu thụ tài nguyên và lập lịch quản lý các tác vụ trên các máy trạm, theo dõi chúng và thực thi lại các tác vụ bị lỗi. Những máy slave TaskTracker thực thi các tác vụ được master chỉ định và cung cấp thông tin trạng thái tác vụ (task-status) để master theo dõi.

JobTracker là một điểm yếu của Hadoop Mapreduce. Nếu JobTracker bị lỗi thì mọi công việc liên quan sẽ bị ngắt quãng.

**Hadoop Distributed File System**

Hadoop có thể làm việc trực tiếp với bất kì hệ thống dữ liệu phân tán như Local FS, HFTP FS, S3 FS, và các hệ thống khác. Nhưng hệ thống file thường được dùng bởi Hadoop là Hadoop Distributed File System (HDFS).

Hadoop Distributed File System (HDFS) dựa trên Google File System (GFS), cung cấp một hệ thống dữ liệu phân tán, được thiết kế để chạy trên các cụm máy tính lớn (gồm hàng ngàn máy tính) có khả năng chịu lỗi cao.

HDFS sử dụng kiến trúc master/slave, trong đó master gồm một NameNode để quản lý hệ thống file metadata và một hay nhiều slave DataNodes để lưu trữ dữ liệu thực tại.

Một tập tin với định dạng HDFS được chia thành nhiều block và những block này được lưu trữ trong một tập các DataNodes. NameNode định nghĩa ánh xạ từ các block đến các DataNode. Các DataNode điều hành các tác vụ đọc và ghi dữ liệu lên hệ thống file. Chúng cũng quản lý việc tạo, huỷ, và nhân rộng các block thông qua các chỉ thị từ NameNode.

HDFS cũng hỗ trợ các câu lệnh shell để tương tác với tập tin như các hệ thống file khác.

**Hadoop hoạt động như thế nào?**

**Giai đoạn 1**

Một user hay một ứng dụng có thể submit một job lên Hadoop (hadoop job client) với yêu cầu xử lý cùng các thông tin cơ bản:

1. Nơi lưu (location) dữ liệu input, output trên hệ thống dữ liệu phân tán.
2. Các java class ở định dạng jar chứa các dòng lệnh thực thi các hàm map và reduce.
3. Các thiết lập cụ thể liên quan đến job thông qua các thông số truyền vào.

**Giai đoạn 2**

Hadoop job client submit job (file jar, file thực thi) và các thiết lập cho JobTracker. Sau đó, master sẽ phân phối tác vụ đến các máy slave để theo dõi và quản lý tiến trình các máy này, đồng thời cung cấp thông tin về tình trạng và chẩn đoán liên quan đến job-client.

**Giai đoạn 3**

TaskTrackers trên các node khác nhau thực thi tác vụ MapReduce và trả về kết quả output được lưu trong hệ thống file.

Khi “chạy Hadoop” có nghĩa là chạy một tập các trình nền - daemon, hoặc các chương trình thường trú, trên các máy chủ khác nhau trên mạng của bạn. Những trình nền có vai trò cụ thể, một số chỉ tồn tại trên một máy chủ, một số có thể tồn tại trên nhiều máy chủ.

Các daemon bao gồm:

* NameNode
* DataNode
* SecondaryNameNode
* JobTracker
* TaskTracker

**NameNode**

Là một trình nền quan trọng nhất của Hadoop - các NameNode. Hadoop sử dụng một kiển trúc master/slave cho cả lưu trữ phân tán và xử lý phân tán. Hệ thống lưu trữ phân tán được gọi là Hadoop File System hay HDFS. NameNode là master của HDFS để chỉ đạo các trình nền DataNode slave để thực hiện các nhiệm vụ I/O mức thấp. NameNode theo dõi HDFS, cách các tập tin của bạn được phân chia thành các block, những node nào lưu các khối đó, và “kiểm tra sức khỏe” tổng thể của hệ thống tệp phân tán. Chức năng của NameNode là nhớ (memory) và I/O chuyên sâu. Như vậy, máy chủ lưu trữ NameNode thường không lưu trữ bất cứ dữ liệu người dùng hoặc thực hiện bất cứ một tính toán nào cho một ứng dụng MapReduce để giảm khổi lượng công việc trên máy. Điều này có nghĩa là máy chủ NameNode không gấp đôi (double) như là DataNode hay một TaskTracker.

Có điều đáng tiếc là có một khía cạnh tiêu cực đến tầm quan trọng của NameNode nó có một điểm của thất bại của một cụm Hadoop của bạn. Đối với bất cứ một trình nền khác, nếu các nút máy của chúng bị hỏng vì lý do phần mềm hay phần cứng, các Hadoop cluster có thể tiếp tục hoạt động thông suốt hoặc bạn có thể khởi động nó một cách nhanh chóng. Nhưng không thể áp dụng cho các NameNode.

**DataNode**

Mỗi máy slave trong cluster của bạn sẽ lưu trữ (host) một trình nền DataNode để thực hiện các công việc nào đó của hệ thống file phân tán - đọc và ghi các khối HDFS tới các file thực tế trên hệ thống file cục bộ (local filesytem). Khi bạn muốn đọc hay ghi một file HDFS, file đó được chia nhỏ thành các khối và NameNode sẽ nói cho các client của bạn nơi các khối trình nền DataNode sẽ nằm trong đó. Client của bạn liên lạc trực tiếp với các trình nền DataNode để xử lý các file cục bộ tương ứng với các block. Hơn nữa, một DataNode có thể giao tiếp với các DataNode khác để nhân bản các khối dữ liệu của nó để dự phòng.

Các DataNode thường xuyên báo cáo với các NameNode. Sau khi khởi tạo, mỗi DataNode thông báo với NameNode của các khối mà nó hiện đang lưu trữ. Sau khi Mapping hoàn thành, các DataNode tiếp tục thăm dò ý kiến NameNode để cung cấp thông tin về thay đổi cục bộ cũng như nhận được hướng dẫn để tạo, di chuyển hoặc xóa các blocks từ đĩa địa phương (local).

**Secondary NameNode**

Các Secondary NameNode (SNN) là một trình nền hỗ trợ giám sát trạng thái của các cụm HDFS. Giống như NameNode, mỗi cụm có một SNN, và nó thường trú trên một máy của mình. Không có các trình nền DataNode hay TaskTracker chạy trên cùng một server. SNN khác với NameNode trong quá trình xử lý của nó không nhận hoặc ghi lại bất cứ thay đổi thời gian thực tới HDFS. Thay vào đó, nó giao tiếp với các NameNode bằng cách chụp những bức ảnh của siêu dữ liệu HDFS (HDFS metadata) tại những khoảng xác định bởi cấu hình của các cluster.

Như đã đề cập trước đó, NameNode là một điểm truy cập duy nhất của lỗi (failure) cho một cụm Hadoop, và các bức ảnh chụp SNN giúp giảm thiểu thời gian ngừng (downtime) và mất dữ liệu. Tuy nhiên, một NameNode không đòi hỏi sự can thiệp của con người để cấu hình lại các cluster sẻ dùng SSN như là NameNode chính.

**jobTracker**

Trình nền JobTracker là một liên lạc giữa ứng dụng của bạn và Hadoop. Một khi bạn gửi mã nguồn của bạn tới các cụm (cluster), JobTracker sẽ quyết định kế hoạch thực hiện bằng cách xác định những tập tin nào sẽ xử lý, các nút được giao các nhiệm vụ khác nhau, và theo dõi tất cả các nhiệm vụ khi chúng đang chạy. Nếu một nhiệm vụ (task) thất bại (fail), JobTracker sẽ tự động chạy lại nhiệm vụ đó, có thể trên một node khác, cho đến một giới hạn nào đó được định sẵn của việc thử lại này.

Chỉ có một JobTracker trên một cụm Hadoop. Nó thường chạy trên một máy chủ như là một nút master của cluster.

**TaskTraker**

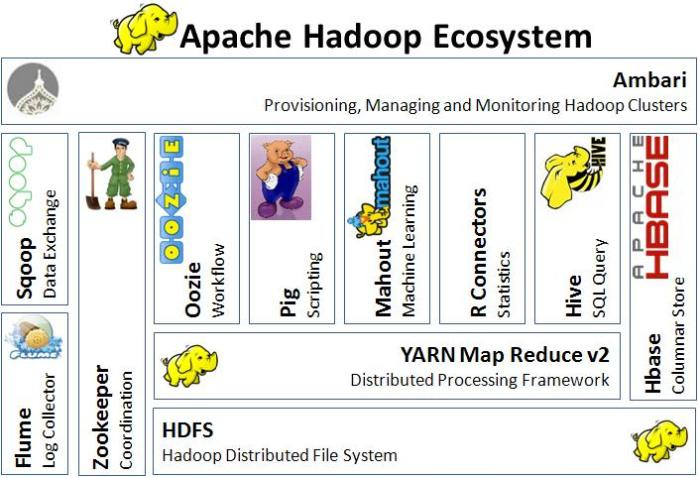
Như với các trình nền lưu trữ, các trình nền tính toán cũng phải tuân theo kiến trúc master/slave: JobTracker là giám sát tổng việc thực hiện chung của một công việc MapRecude và các taskTracker quản lý việc thực hiện các nhiệm vụ riêng trên mỗi node slave. Mỗi TaskTracker chịu trách nhiệm thực hiện các task riêng mà các JobTracker giao cho. Mặc dù có một TaskTracker duy nhất cho một node slave, mỗi TaskTracker có thể sinh ra nhiều JVM để xử lý các nhiệm vụ Map hoặc Reduce song song.

Một trong những trách nhiệm của các TaskTracker là liên tục liên lạc với JobTracker. Nếu JobTracker không nhận được nhịp đập từ một TaskTracker trong vòng một lượng thời gian đã quy định, nó sẽ cho rằng TaskTracker đã bị treo (cashed) và sẽ gửi lại nhiệm vụ tương ứng cho các nút khác trong cluster.

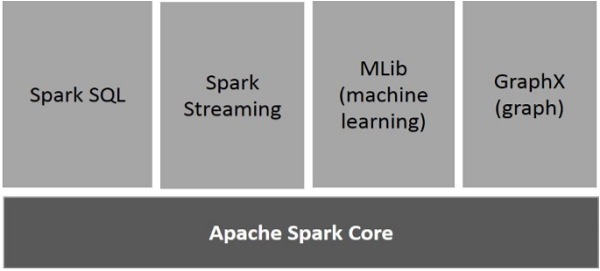
Cấu trúc liên kết này có một node Master là trình nền NameNode và JobTracker và một node đơn với SNN trong trường hợp node Master bị lỗi. Đối với các cụm nhở, thì SNN có thể thường chú trong một node slave. Mặt khác, đối với các cụm lớn, phân tách NameNode và JobTracker thành hai máy riêng. Các máy slave, mỗi máy chỉ lưu trữ một DataNode và Tasktracker, để chạy các nhiệm vụ trên cùng một node nơi lưu dữ liệu của chúng

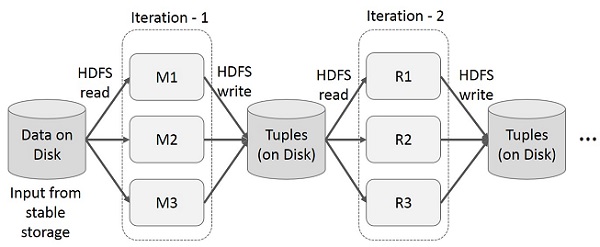
**Ưu điểm của Hadoop**

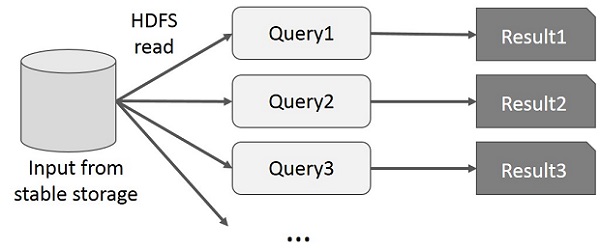
* Hadoop framework cho phép người dùng nhanh chóng viết và kiểm tra các hệ thống phân tán. Đây là cách hiệu quả cho phép phân phối dữ liệu và công việc xuyên suốt các máy trạm nhờ vào cơ chế xử lý song song của các lõi CPU.
* Hadoop không dựa vào cơ chế chịu lỗi của phần cứng fault-tolerance and high availability (FTHA), thay vì vậy bản thân Hadoop có các thư viện được thiết kế để phát hiện và xử lý các lỗi ở lớp ứng dụng.
* Các server có thể được thêm vào hoặc gỡ bỏ từ cluster một cách linh hoạt và vẫn hoạt động mà không bị ngắt quãng.
* Một lợi thế lớn của Hadoop ngoài mã nguồn mở đó là khả năng tương thích trên tất cả các nền tảng do được phát triển trên Java.



## Hadoop – Spark









## Spark

* RDD (resilient distributed dataset) là tập dữ liệu phân tán mà các dữ liệu này được phân tán vào các node của cluster để thực hiện tính toán song song.

1. Có 2 loại RDD trong Spark là: parallelized collections và Hadoop datasets.

* Parallelized collections

Parallelized collections được tạo bằng cách gọi phương thức parallelize với 1 collection có sẵn của Scala( ví dụ: Array, List,…).

scala> val data = Array(1, 2, 3, 4, 5)

data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)

distData: spark.RDD[Int] = spark.ParallelCollection@10d13e3e

* Hadoop Datasets

Spark cũng có thể tạo tập dữ liệu phân tán từ file bất kỳ được lưu trữ trong Hadoop( HDFS) hoặc hệ thống lưu trữ hỗ trợ bởi Hadoop. Khi đó, sẽ dùng hàm textFile để thực hiện kết nối đến file được lưu trữ ở trên.

scala> val distFile = sc.textFile("data.txt")

distFile: spark.RDD[String] = spark.HadoopRDD@1d4cee08

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

* Transformations:

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

Tất cả các transformation đều là lazy, có nghĩa là các transformation này sẽ không thực hiện tính toán trong phương thức ngay mà chúng sẽ được lưu lại thành dữ liệu cơ bản (ví dụ như file) và chúng chỉ thực hiện tính toán khi 1 action được gọi để yêu cầu trả về kết quả cho driver program. Nhờ thiết kế này mà Spark chạy hiệu quả hơn.

Ví dụ:

Chúng ta có thể nhận thấy kết quả là 1 bộ dữ liệu được tạo qua hàm map và chỉ trả về kết quả cho driver program qua hàm reduce. Lazy ở đây tốt ở chỗ là chúng ta không phải tạo ra 1 tập dữ liệu và thực hiện lưu trữ chúng trong file rồi sau đó các actions sẽ phải tìm cách truy cập đến các file lưu trữ này mà thay vào đó, chúng ta sẽ tiết kiệm được thời gian(thời gian truy cập và thời gian lưu trữ) và không gian lưu trữ kết quả (RDD) qua transformation.

|  |  |
| --- | --- |
| **Transformation** | **Ý nghĩa** |
| **map**(*func*) | Trả về 1 RDD mới bằng cách truyền mỗi phần tử đầu vào(nguồn)qua hàm *func*. |
| **filter**(*func*) | Trả về 1 RDD mới bằng cách chọn những phần tử đầu vào(nguồn)mà hàm *func* trả về kết quả *true.* |
| **flatMap**(*func*) | Tương tự *map*nhưng khác *map* ở chỗ, mỗi phần tử đầu vào qua*flatMap*sẽ trả về 0 hoặc nhiều phần tử đầu ra(có thể hiểu qua *map*sẽ là 1-1). |
| **mapPartitions**(*func*) | Tương tự như *map* nhưng chạy riêng biệt trên mỗi vùng RDD,Hàm*func* phải có dạng Iterator[T] => Iterator[U] khi chạy RDD kiểuT. |
| **mapPartitionsWithSplit**(*func*) | Tương tự *mapPartitions*, nhưng hàm *func*được cung cấp thêm 1 sốnguyên để chỉ số split. |
| **union**(*otherDataset*) | Trả về 1 RDD mới là hợp của tập dữ liệu phần tử đầu vào(nguồn) vàcác phần tử của đối(*otherDataset*). |
| **distinct**([*numTasks*])) | Trả về 1 RDD mới chứa mỗi phần tử là duy nhất của tập dữ liệunguồn(đầu vào). |
| **groupByKey**([*numTasks*]) | Khi gọi đến 1 tập dữ liệu (K,V) sẽ trả về 1 tập là cặp (K,Seq(V))( Tức lànhóm tập các phần tử cùng *Key*). **Chú ý:**mặc định chỉ có 8 task song song khi grouping. Có thể thay đổi số task song song này bằng việc truyềnvào tham số đầu vào. |
| **reduceByKey**(*func*, [*numTasks*]) | Khi gọi tập dữ liệu (K,V), trả về 1 tập (K,V) mà giá trị của key được tổng hợp sử dụng hàm reduce *func.* |
| **sortByKey**([*ascending*], [*numTasks*]) | Khi gọi tập dữ liệu (K,V) với K có thể thực hiện sắp thứ tự được.Khi đó, nó sẽ trả về tập dữ liệu (K,V) được sắp sếp tăng dần hoặc giảm dần theo key. **Chú ý:***ascending*là kiểu Boolean. |
| **join**(*otherDataset*, [*numTasks*]) | Khi gọi tập dữ liệu có kiểu (K,V) và (K,W), nó sẽ trả về 1 cặp mới(K,(V,W))( nối 2 phần tử có cùng key). |
| **cogroup**(*otherDataset*, [*numTasks*]) | Khi gọi tập dữ liệu có kiểu (K,V) và (K,W), nó sẽ trả về 1 tập dữ liệu(K,seq(V),seq(W)). |
| **cartesian**(*otherDataset*) | Khi gọi 1 tập dữ liệu kiểu T và U, nó sẽ trả về tập dữ liệu mới (T,U). |

* + Actions

Qua 1 phương thức actions thì sẽ cho phép trả về 1 giá trị cho driver program sau khi chạy tính toán trên tập dữ liệu.

|  |  |
| --- | --- |
| **Action** | **Ý nghĩa** |
| **reduce**(*func*) | Tổng hợp các phần tử của tập dữ liệu sử dụng hàm *func*(có 2 đối và trả về 1 kết quả). |
| **collect**() | Trả về tất cả các phần tử của tập dữ liệu như 1 mảng ở driverProgram. Hàm này hữu ích sau khi lọc hoặc thao tác khác mà trả về tập dữ liệu con đủ nhỏ.  Cẩn thận khi sử dụng vì tập hợp dữ liệu có thể tràn bộ nhớ, nên sử dụng take() |
| **count**() | Trả về số phần tử của tập dữ liệu |
| **first**() | Trả về phần tử đầu tiên của tập dữ liệu( tương tự take(1)). |
| **take**(*n*) | Trả về mảng gồm n phần tử đầu tiên của tập dữ liệu. |
| **saveAsTextFile**(*path*) | Ghi các phần tử của tập dữ liệu như 1 file text( hoặc tập file text)lên 1 thư mục trong hệ thống local, HDFS hoặc hệ thống hỗ trợ Hadoop bất kỳ. |
| **countByKey**() | Chỉ cho RDD có kiểu (K,V). Trả về 1 Map (K,Int). Int là chỉ số key. |
| **foreach**(*func*) | Chạy hàm *func*cho mỗi phần tử của tập dữ liệu. Điều này có tác dụngkhi thực hiện cập nhật 1 biến accumulator hoặc tương tác vớihệ thống lưu trữ ngoài. |

* + RDD Persistence

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

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

Sau đây là các mức lưu trữ :

|  |  |
| --- | --- |
| **Storage Level** | **Ý nghĩa** |
| MEMORY\_ONLY | Lưu trữ RDD như đối tượng deserialized Java trên JVM.Nếu RDD vượt quá bộ nhớ trong thì 1 số vùng sẽ khôngđược cached và nó sẽ tính toán lại mỗi khi nó cần. Đây  là mức lưu trữ mặc định. |
| DISK\_ONLY | Lưu trữ các vùng RDD chỉ trên ổ cứng. |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, etc. | Giống như các mức trên nhưng sao lặp mỗi vùng RDD trên2 cluster nodes. |

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

* Shared variables thực hiện chia sẻ biến giữa các task hoặc giữa các task với driver program.

Spark cung cấp 2 kiểu chia sẻ biến: broadcast variable và accumulators.

* + Broadcast Variables

Là biến mà chỉ được phép đọc giá trị của biến trên mỗi máy, không cho phép sửa đối giá trị nhằm mục đích đảm bảo cùng giá trị của biến broadcast trên tất cả các node. Có thể sử dụng giá trị của biến qua phương thức value.

scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))

broadcastVar: spark.Broadcast[Array[Int]] = spark.Broadcast(b5c40191-a864-4c7d-b9bf-d87e1a4e787c)

scala> broadcastVar.value

res0: Array[Int] = Array(1,2,3)

* + Accumulators

Là biến mà chỉ thực hiện chỉ “added” qua phương thức kết hợp (như sum hoặc count). Ví dụ:

scala> val accum = sc.accumulator(0)

accum: spark.Accumulator[Int] = 0

scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

...10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

scala> accum.value

res2:Int=10

Khác với biến broadcast, biến accumulator cho phép thay đối giá trị, cho phép đọc ghi dữ liệu lên biến này. Điều này rất tiện lợi cho việc tính toán song song trên các cụm và thực hiện cho cùng mục đích là cùng nguồn tài nguyên. Sau khi tính toán song nó sẽ trả về driver program.