Lý thuyết Apache Spark

I. Introduction

Spark là 1 công cụ được Apache giới thiệu nhằm tăng tốc quá trình thực thi và tính toán
Spark không phải 1 phiên bản chính sửa của Hadoop và không phụ thuộc vào hadoop vì nó có cluster management riêng. Hadoop chỉ là 1 trong những cách triển khai Spark
Spark được sử dụng theo 2 cách:

- lưu trữ
- xử lý
  Spark chỉ sử dụng hadoop với mục đích lưu trữ

Apache Spark

Apache Spark là lightning-fast cluster computing technology (công nghệ điện toán cụm cực nhanh).
Nó dựa trên Hadoop MapReduce và mở rộng mô hình MapReduce để tính toán hiệu quả hơn, bao gồm:

- interactive queries (truy vấn tương tác)
- stream processing (xử lý luồng)
  Tính năng chính:
  In-memory cluster computing to increases the processing speed of an application.
  Điện toán cụm trong bộ nhớ giúp tăng tốc độ xử lý của ứng dụng.

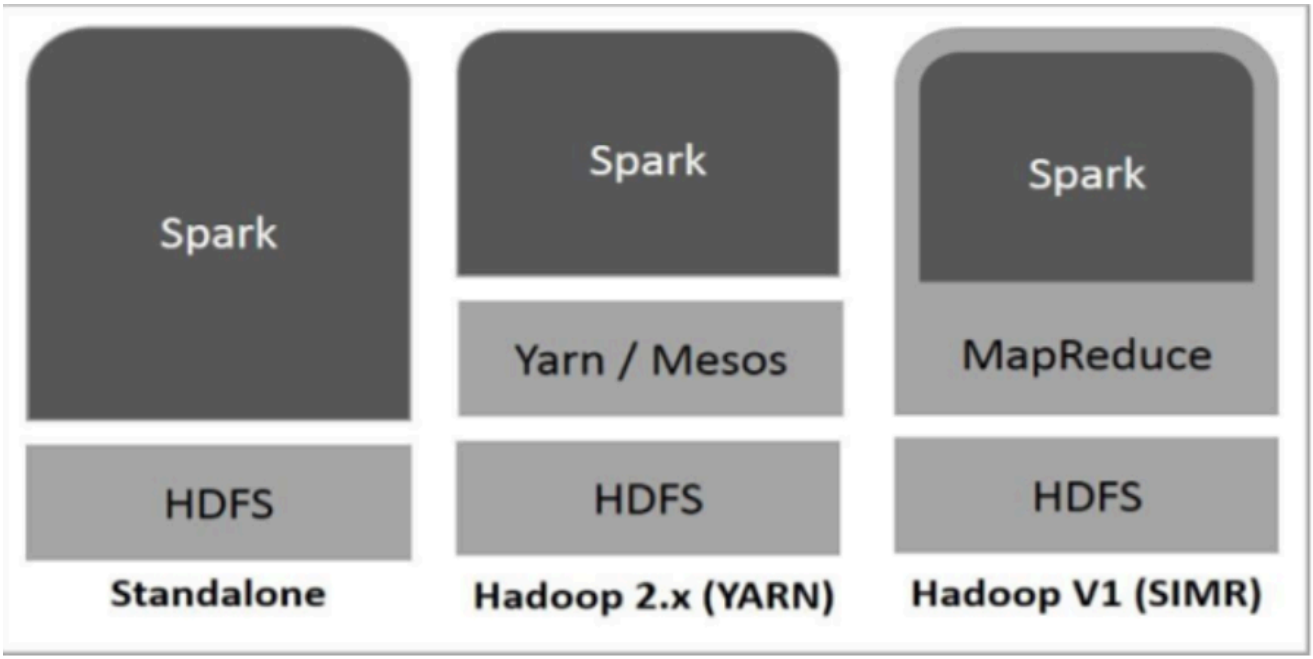
Spark được thiết kế để bao phủ nhiều khối lượng công việc như:

- batch applications (xử lý theo lô) [1]
- iterative algorithms (thuật toán lặp) [2]
- interactive queries (truy vấn tương tác) [3]
- streaming (xử lý dòng dữ liệu liên tục) [4]

Evolution of Apache Spark

- Spark là 1 dự án con của Hadoop, được phát triển vào 2009 tại AMPLab, UC Berkeley[5] bởi Matei Zaharia
- Nó được open source vào năm 2010 theo giấy phép BSD
- Nó được tặng cho Apache Software Foundation vào năm 2013
- Hiện tại Apache Spark đã trở thành dự án Apache cấp cao nhất từ tháng 2 năm 2014.

Spark built on Hadoop



1. Standalone

- Spark chạy trực tiếp trên HDFS mà không cần các thành phần khác của Hadoop
- Spark có trình quản lý tài nguyên riêng, không phục thuộc vào YARN[6] hay Mesos[7]
- Cấu hình đơn giản, phù hợp khi chỉ chạy Spark mà không cần tích hợp sâu vào hệ sinh thái Hadoop

2. Hadoop YARN

- Spark được triển khai trên YARN
- Spark không cần cài đặt mà có thể chạy trực tiếp trên YARN
- Giúp tích hợp Spark vào hệ sinh thái Hadoop mà không cần quyền quản trị hệ thống

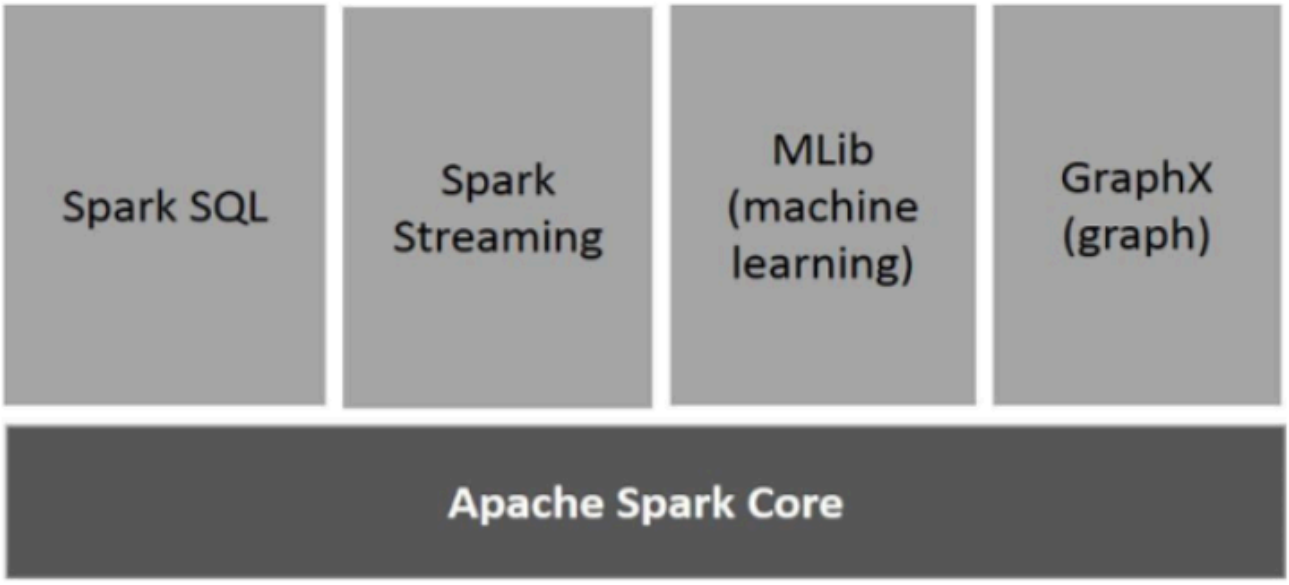
- Linh hoạt hơn vì có thể chia sẻ tài nguyên với các ứng dụng Hadoop khác

3. Spark in MapReduce (SIMR)

- Spark chạy trong môi trường MapReduce của hadoop V1
- không cài đặt Spark từ đầu, có thể khởi động Spark từ giao diện dòng lệnh.
- Không yêu cầu quyền admin, nhưng hiệu suất không cao

Triển khai	Quản lý tài nguyên	Tích hợp với Hadoop	Hiệu suất	Quyền admin
Standalone	trình quản lý tài nguyên riêng	Thấp	Cao	Có
YARN	Yarn của Hadoop	Cao	Trung bình cao	không
SIMR	MapReduce	Trung Bình	Thấp	Không

Components of Spark



1. Apache Spark Core

- Là **thành phần cốt lõi** của Spark, cung cấp khả năng **xử lý dữ liệu phân tán** và thực thi tác vụ.
- Hỗ trợ **tính toán trong bộ nhớ (In-Memory Computing)** để tăng tốc độ xử lý dữ liệu.

- Cho phép tham chiếu các tập dữ liệu từ các hệ thống lưu trữ bên ngoài như **HDFS, Cassandra, S3, và HBase**.

2. Spark SQL

- Cung cấp **SchemaRDD**, một dạng RDD mở rộng hỗ trợ dữ liệu có cấu trúc (structured data) và bán cấu trúc (semi-structured data).
- Hỗ trợ truy vấn dữ liệu bằng **SQL** hoặc **DataFrame API** giống như các hệ quản trị cơ sở dữ liệu truyền thống.
- Có thể **tích hợp với các hệ thống dữ liệu lớn** như Hive, Avro, Parquet, ORC.

3. Spark Streaming

- Hỗ trợ **xử lý dữ liệu thời gian thực (real-time streaming analytics)**.
- Dữ liệu đầu vào được xử lý dưới dạng **mini-batches**, sau đó chuyển đổi thông qua RDD.
- Có thể **tích hợp với Kafka, Flume, HDFS, và Sockets** để thu thập và xử lý luồng dữ liệu liên tục.

4. MLlib (Machine Learning Library)

- Thư viện **học máy phân tán (distributed machine learning)** được tối ưu hóa cho kiến trúc bộ nhớ của Spark.
- Cung cấp nhiều thuật toán **học máy phổ biến** như hồi quy, phân cụm, cây quyết định, SVM, PCA.
- Theo benchmark, MLlib **nhanh hơn gấp 9 lần** so với Apache Mahout (trước khi Mahout tích hợp Spark).

5. GraphX

- Một **hệ thống xử lý đồ thị phân tán** trên nền tảng Spark.
- Cung cấp API giúp biểu diễn và tính toán đồ thị với **Pregel API**.
- Hỗ trợ các **thuật toán đồ thị phổ biến** như PageRank, Connected Components, Shortest Paths,...
- Có thể **chuyển đổi qua lại giữa dữ liệu dạng bảng (RDD, DataFrame) và đồ thị** để khai thác mối quan hệ trong dữ liệu.

RDD - Resilient Distributed Datasets

1. Định nghĩa

- RDD là tập các đối tượng bất biến (immutable) được phân tán trên nhiều nút trong cụm
- Mỗi tập dữ liệu tỏng RDD được chia thành các phân vùng logic (logical partitions), có thể được tính toán trên các nút khác nhau trong cụm
- RDD có thể chứa bất kỳ các đối tượng Python, Java hoặc Scala nào, bao gồm các lớp do người dùng định nghĩa (user-definded classes)
- Về cơ bản, RDD là bộ sưu tập các bản ghi phân vùng chỉ đọc (read-only partitioned)
- RDD có thể được tạo thông qua các hoạt động xác định trên trên dữ liệu ổn định (stable storage) hoặc các RDD khác.
- RDD là bộ sư tập các phần tử có khả năng chịu lỗi (fault-tolerant), có thể được vận hành song song.

2. Tính chất của RDD

- Tính bất biến (Immutable): sau khi được tạo ra, RDD không thể thay đổi. Mọi thao tác biến đổi trên RDD sẽ tạo ra RDD mới .
- Tính phân tán (Distributed): Dữ liệu trong RDD được phân tán trên nhiều nút, giúp tận dụng tài nguyên và tăng hiệu suất xử lý.

3. Tạo RDD

Có 2 cách để tạo ra RDD:

- Tham chiếu đến tập dữ liệu bên ngoài: Đọc dữ liệu từ cá hệ thống lưu trữ như HDFS, Cassandra, HBase hoặc từ tệp cục bộ.
- Áp dụng các phép biến đổi (transformations): Sử dụng các phép biển đổi như map, filter trên các RDD hện có thể tạo ra RDD mới.
- Spark sử dụng khái niệm RDD để chạy MapReduce nhanh hơn và hiệu quả hơn

4. Các phép biến đổi và hành động trên RDD

- Biến đổi (Transformations): là phép toán tạo ra RDD mới từ RDD hiện có(map, zfilter, flatMap). Các phép biến đổi lazy, chỉ thực thi khi hành động được gọi.
- Hành động (Actions): thực thi các phép toán và trả về kết quả cho driver program hoặc ghi ra hệ thống lưu trữ như collect, count, saveAsTextFile.

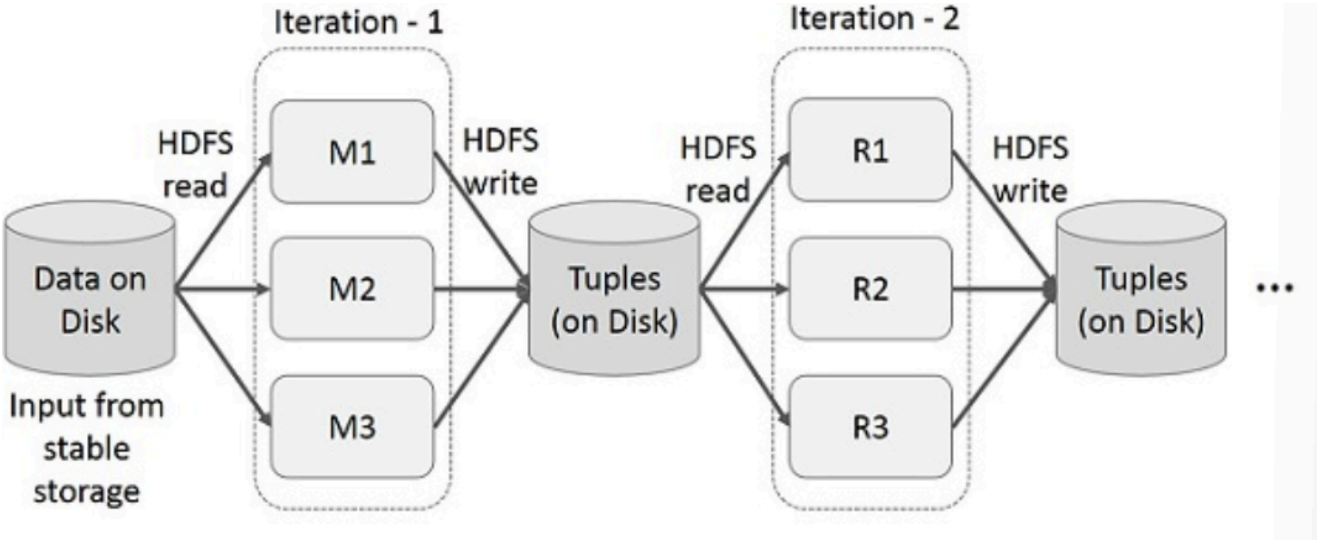
5. Nguyên nhân khiến RDD chạy nhanh hơn và hiệu quả hơn

5.1 Khi không ứng dụng RDD

Data sharing is slow in MapRedue(MR)

- **Vấn đề:** Trong Hadoop MapReduce, dữ liệu phải được lưu vào **HDFS (Hadoop Distributed File System)** giữa các bước xử lý. Điều này làm cho quá trình chia sẻ dữ liệu giữa các công việc trở nên chậm và tốn nhiều tài nguyên.
- **Nguyên nhân:**
  - Sau mỗi bước **Map** hoặc **Reduce**, dữ liệu trung gian phải được ghi vào đĩa (HDFS).
  - Đọc/ghi từ HDFS có độ trễ cao hơn so với truy xuất từ bộ nhớ.
- **Hệ quả:**
  - Ảnh hưởng đến hiệu suất tổng thể.
  - Không phù hợp với các bài toán yêu cầu tính toán lặp (iterative algorithms) hoặc truy vấn tương tác.
- Cả ứng dụng Iterative[2-1] and Interactive[3-1] đều yêu cầu chia sẻ dữ liệu nhanh hơn giữa các tác vụ song song. Chia sẻ dữ liệu chậm trong MapReduce do sao chép, tuần tự hóa và IO đĩa. Về hệ thống lưu trữ, hầu hết các ứng dụng Hadoop đều dành hơn 90% thời gian để thực hiện các hoạt động đọc-ghi HDFS.

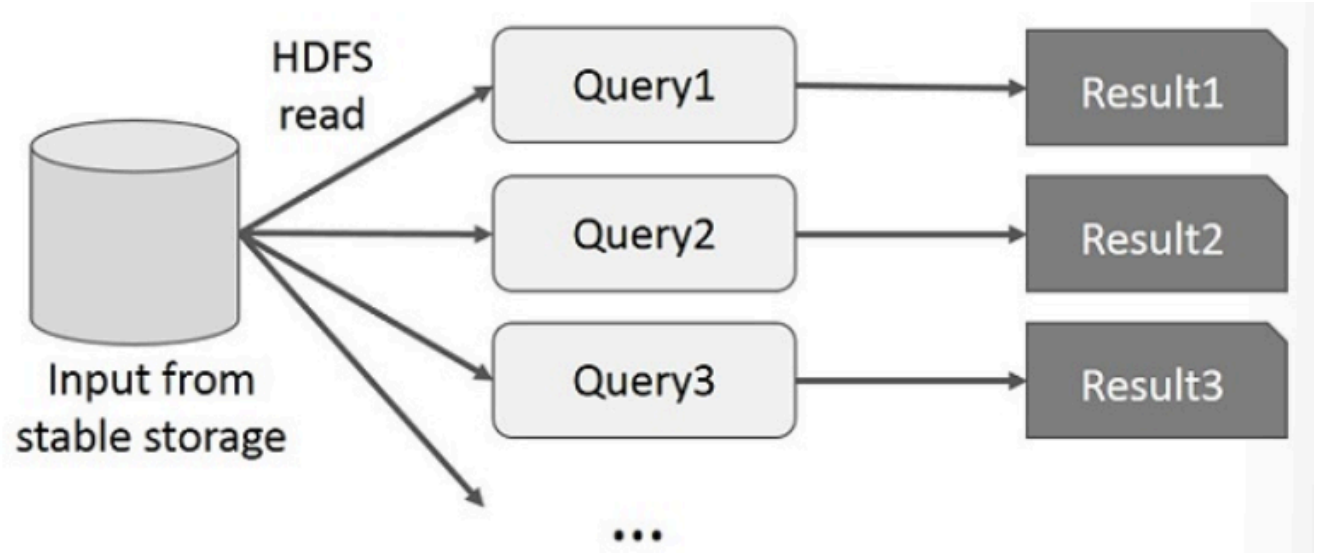
Iterative Operations on MapReduce



- **Vấn đề:**
  - MapReduce không được thiết kế cho các thuật toán lặp lại.
  - Sau mỗi vòng lặp, nó phải ghi dữ liệu ra HDFS và đọc lại cho vòng lặp tiếp theo.
- **Hệ quả:**
  - Các thuật toán học máy hoặc xử lý đồ thị (như PageRank) trên Hadoop **rất chậm** so với Spark.
- **Ví dụ:**

- Nếu bạn chạy thuật toán **K-Means Clustering** trên Hadoop, mỗi vòng lặp sẽ lưu dữ liệu ra HDFS, gây lãng phí tài nguyên và thời gian.

Intervactive operations on MapReduce



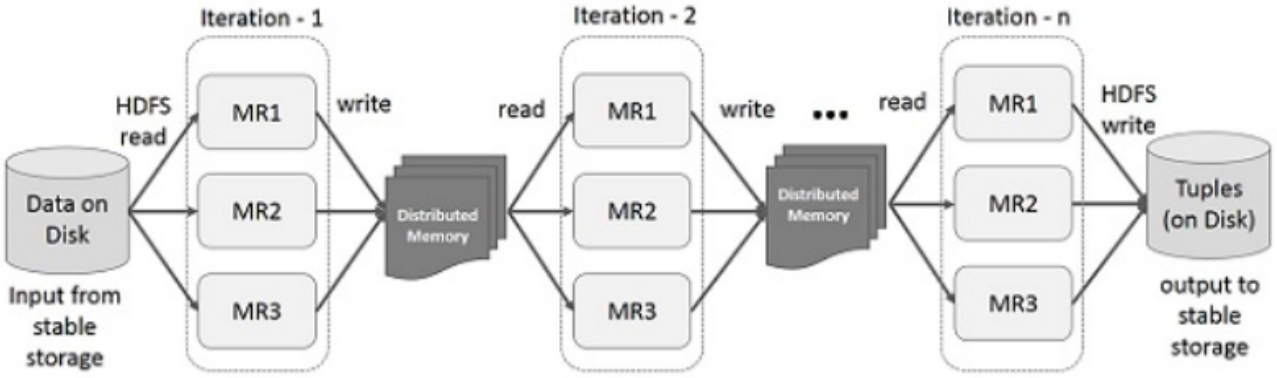
- **Hạn chế của MapReduce:**
  - Không hỗ trợ tương tác nhanh vì **mỗi lần chạy một truy vấn mới, nó phải đọc lại dữ liệu từ HDFS**.
  - **MapReduce không có cơ chế lưu trữ trong bộ nhớ**, mỗi thao tác mới đều phải đọc và ghi vào đĩa.
- **Hệ quả:**
  - Độ trễ cao, không phù hợp cho truy vấn dữ liệu động hoặc nhiều lần.
- **Ví dụ:**
  - Nếu bạn muốn đếm số từ trong một tập dữ liệu lớn, MapReduce phải đọc toàn bộ dữ liệu từ đầu mỗi lần thực thi.

5.2 Khi úng dụng RDD

Data Sharing using Spark RDD

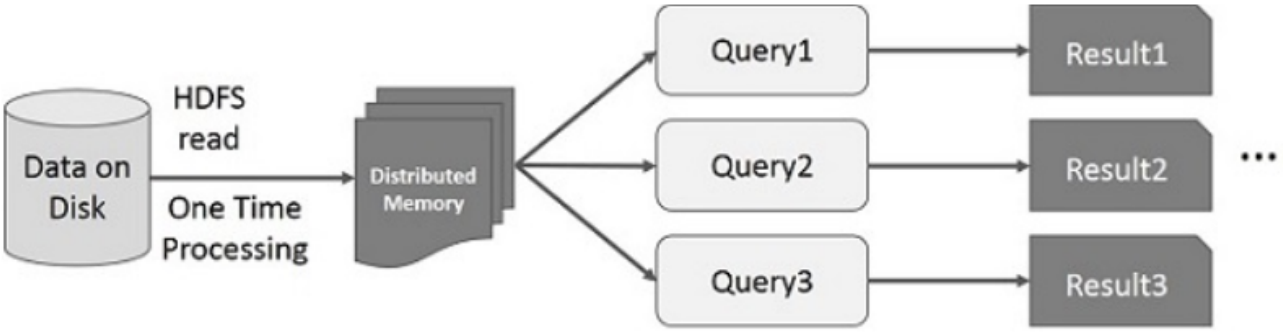
- **Ưu điểm:**
  - Spark hỗ trợ **chia sẻ dữ liệu hiệu quả hơn** so với Hadoop MapReduce.
  - RDD có thể được **cache (bộ nhớ đệm)** trong RAM, giúp giảm thời gian đọc/ghi.
- **Lợi ích:**
  - Các công việc xử lý trên cùng một tập dữ liệu có thể tái sử dụng dữ liệu mà không cần tải lại.
  - Tăng tốc độ xử lý nhiều lần so với MapReduce.
  - Bộ nhớ chia sẻ nhanh hơn 10 - 100 lần só mới mạng và ổ đĩa

Iterative Operations on Spark RDD



- **Spark tối ưu hóa cho các thuật toán lặp (iterative algorithms) như Machine Learning hoặc PageRank.**
- **Lý do:**
  - RDD có thể được **tái sử dụng (reused)** trong bộ nhớ mà không cần ghi lại vào HDFS sau mỗi vòng lặp.
- **Ứng dụng:**
  - Học máy (Machine Learning) với thuật toán Gradient Descent.
  - Thuật toán PageRank (tính mức độ quan trọng của trang web).
- **Ví dụ:**
  - Nếu thực hiện một thuật toán **Gradient Descent**, Spark có thể giữ dữ liệu trong bộ nhớ và thực hiện nhiều vòng lặp mà không cần đọc lại từ HDFS mỗi lần.

Interactive Operations on Spark RDD



- **Spark RDD hỗ trợ thao tác tương tác nhanh hơn so với MapReduce.**
- **Lý do:**
  - Spark **lưu trữ dữ liệu trong bộ nhớ (in-memory computing)** thay vì đọc/ghi liên tục vào HDFS.
  - Dữ liệu có thể được truy vấn lại nhiều lần mà không cần tải lại từ đầu.
- **Ứng dụng:**
  - Phân tích dữ liệu theo thời gian thực.
  - Truy vấn dữ liệu nhiều lần với độ trễ thấp.

So Sánh

Tính năng	Spark RDD	Hadoop MapReduce
Chia sẻ dữ liệu	Lưu trong ram	Ghi vào HDFS giữa các bước
truy vấn tương tác	thông qua caching mà xử lý dữ liệu liên tục	đọc dữ liệu từ hdfs nhiều lần
Tính toán lặp	Hiệu quả cao do dữ liệu được lưu trữ trong bộ nhớ	Chậm do phải đọc ghi liên tục vào hdfs

→ **Kết luận:** Spark vượt trội hơn MapReduce trong các ứng dụng yêu cầu xử lý nhanh và tính toán lặp lại, nhờ vào khả năng lưu trữ dữ liệu trong bộ nhớ và giảm số lần đọc/ghi vào đĩa.

Apache Spark Core Programming

RDD Transformations

S.No	Transformation	Ý nghĩa
1	`map(func)`	Trả về một tập dữ liệu phân tán mới bằng cách áp dụng hàm `func` lên từng phần tử của tập dữ liệu nguồn.
2	`filter(func)`	Trả về một tập dữ liệu mới bao gồm các phần tử trong tập nguồn thỏa mãn điều kiện của `func`.
3	`flatMap(func)`	Tương tự như `map`, nhưng mỗi phần tử đầu vào có thể ánh xạ thành nhiều phần tử đầu ra.
4	`mapPartitions(func)`	Tương tự như `map`, nhưng chạy riêng trên từng phân vùng (block) của RDD.
5	`mapPartitionsWithIndex(func)`	Tương tự `mapPartitions`, nhưng cung cấp thêm chỉ mục của phân vùng.
6	`sample(withReplacement, fraction, seed)`	Lấy mẫu một phần của dữ liệu với hoặc không

S.No	Transformation	Ý nghĩa
		có thay thế, sử dụng bộ sinh số ngẫu nhiên.
7	`union(otherDataset)`	Trả về một tập dữ liệu mới là hợp của tập dữ liệu nguồn và một tập dữ liệu khác.
8	`intersection(otherDataset)`	Trả về tập dữ liệu chứa các phần tử xuất hiện trong cả hai tập dữ liệu.
9	`distinct([numTasks])`	Trả về một tập dữ liệu mới chỉ chứa các phần tử duy nhất của tập nguồn.
10	`groupByKey([numTasks])`	Khi được gọi trên tập dữ liệu (K, V), nhóm các giá trị có cùng khóa vào tập (K, Iterable).
11	`reduceByKey(func, [numTasks])`	Tương tự `groupByKey`, nhưng áp dụng hàm `func` để tổng hợp giá trị.
12	`aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])`	Tổng hợp các giá trị theo khóa bằng cách sử dụng giá trị khởi tạo, giảm thiểu việc cấp phát bộ nhớ.
13	`sortByKey([ascending], [numTasks])`	Sắp xếp tập dữ liệu theo khóa theo thứ tự tăng dần hoặc giảm dần.
14	`join(otherDataset, [numTasks])`	Kết hợp hai tập dữ liệu (K, V) và (K, W) để tạo ra (K, (V, W)).
15	`cogroup(otherDataset, [numTasks])`	Nhóm nhiều tập dữ liệu có cặp khóa-giá trị.
16	`cartesian(otherDataset)`	Trả về tập dữ liệu chứa tất cả các cặp (T, U) giữa hai tập dữ liệu T và U.
17	`pipe(command, [envVars])`	Chuyển đổi từng phân vùng của RDD bằng cách sử dụng một lệnh ngoài, ví dụ như shell script.
18	`coalesce(numPartitions)`	Giảm số lượng phân vùng của RDD mà không cần phân phối lại dữ liệu.
19	`repartition(numPartitions)`	Phân phối lại dữ liệu để tăng hoặc giảm số lượng

S.No	Transformation	Ý nghĩa
		phân vùng.
20	`repartitionAndSortWithinPartitions(partitioner)`	Tương tự `repartition`, nhưng sắp xếp dữ liệu trong từng phân vùng để tối ưu hóa quá trình shuffle.

RDD Actions

S.No	Action	Ý nghĩa
1	`reduce(func)`	Tổng hợp các phần tử trong tập dữ liệu bằng cách sử dụng hàm `func`, nhận hai đối số và trả về một kết quả.
2	`collect()`	Trả về tất cả các phần tử của tập dữ liệu dưới dạng một mảng tại chương trình điều khiển.
3	`count()`	Trả về số lượng phần tử trong tập dữ liệu.
4	`first()`	Trả về phần tử đầu tiên của tập dữ liệu.
5	`take(n)`	Trả về một mảng chứa `n` phần tử đầu tiên của tập dữ liệu.
6	`takeSample(withReplacement, num, [seed])`	Trả về một mẫu ngẫu nhiên gồm `num` phần tử từ tập dữ liệu.
7	`takeOrdered(n, [ordering])`	Trả về `n` phần tử đầu tiên theo thứ tự mặc định hoặc tùy chỉnh.
8	`saveAsTextFile(path)`	Lưu tập dữ liệu thành các tệp văn bản tại một thư mục nhất định.
9	`saveAsSequenceFile(path)`	Lưu tập dữ liệu dưới dạng tệp Hadoop SequenceFile.
10	`saveAsObjectFile(path)`	Lưu tập dữ liệu dưới dạng các đối tượng Java đã tuần tự hóa.
11	`countByKey()`	Chỉ áp dụng cho RDD chứa (K, V), trả về số lần xuất hiện của mỗi khóa dưới dạng hashmap.
12	`foreach(func)`	Chạy hàm `func` trên mỗi phần tử của tập dữ liệu, thường được sử dụng để cập nhật các hệ thống bên ngoài.

Numeric RDD Operations in Spark

Spark hỗ trợ các thao tác trên dữ liệu số bằng cách sử dụng các phương thức API được định nghĩa sẵn. Các phép toán này được tính toán và trả về dưới dạng một đối tượng `StatusCounter` thông qua phương thức `status()`.

Danh sách các phương thức xử lý dữ liệu số trong RDD

S.No	Method	Ý nghĩa
1	`count()`	Trả về số lượng phần tử trong RDD.
2	`mean()`	Trả về giá trị trung bình của các phần tử trong RDD.
3	`sum()`	Tính tổng tất cả các phần tử trong RDD.
4	`max()`	Trả về giá trị lớn nhất trong RDD.
5	`min()`	Trả về giá trị nhỏ nhất trong RDD.
6	`variance()`	Tính phương sai của các phần tử trong RDD.
7	`stdev()`	Tính độ lệch chuẩn của các phần tử trong RDD.

#LuuVinhTuong #HCMUTE

---

1. **Batch Applications (Xử lý theo lô)**:
   - Là cách xử lý dữ liệu theo từng khối lớn, thay vì xử lý ngay lập tức từng phần nhỏ.
   - Dữ liệu được thu thập, lưu trữ rồi mới được xử lý cùng một lúc.
   - Phù hợp với các tác vụ như xử lý nhật ký hệ thống, phân tích dữ liệu lớn theo chu kỳ (hàng ngày, hàng giờ).
   - Ví dụ: Hệ thống tính toán doanh số cuối ngày từ dữ liệu bán hàng.

   ↩
2. **Iterative Algorithms (Thuật toán lặp)**:
   - Là các thuật toán cần lặp đi lặp lại trên cùng một tập dữ liệu để hội tụ về kết quả tốt nhất.
   - Thường dùng trong Machine Learning (Học máy) và Graph Processing (Xử lý đồ thị).
   - Ví dụ: Thuật toán phân cụm K-Means hoặc thuật toán PageRank của Google.

   ↩ ↩
3. **Interactive Queries (Truy vấn tương tác)**:
   - Cho phép người dùng gửi truy vấn và nhận kết quả gần như ngay lập tức.
   - Thường được dùng trong các hệ thống phân tích dữ liệu thời gian thực, nơi người dùng muốn kiểm tra dữ liệu một cách linh hoạt.
   - Ví dụ: Truy vấn SQL trên dữ liệu lớn bằng Spark SQL để tìm kiếm thông tin nhanh chóng.

   ↩ ↩
4. **Streaming (Xử lý dòng dữ liệu liên tục)**
   - Là cách xử lý dữ liệu ngay khi nó xuất hiện, thay vì đợi gom đủ thành lô lớn.

- Thích hợp cho các ứng dụng thời gian thực như giám sát giao dịch ngân hàng, phân tích lưu lượng truy cập web.
- Ví dụ: Phát hiện gian lận trong giao dịch thẻ tín dụng theo thời gian thực.

↩

5. **AMPLab** (Algorithms, Machines, and People Lab) là một phòng thí nghiệm nghiên cứu tại Đại học California, Berkeley (UC Berkeley). Đây là nơi đã phát triển nhiều công nghệ quan trọng trong lĩnh vực **xử lý dữ liệu lớn (Big Data)** và **trí tuệ nhân tạo (AI)**. ↩
6. **YARN (Yet Another Resource Negotiator)** là một thành phần quản lý tài nguyên và job scheduling của Hadoop. Nó phân bổ tài nguyên (CPU, bộ nhớ) cho nhiều ứng dụng chạy trên cụm và quản lý việc thực thi của chúng một cách hiệu quả. ↩
7. Apache Mesos là một **trình quản lý tài nguyên (resource manager)** mã nguồn mở, giúp phân phối và quản lý tài nguyên máy tính trong các hệ thống phân tán, đặc biệt là trong các cụm máy chủ (clusters). Nó cho phép chạy nhiều ứng dụng khác nhau như **Apache Spark, Hadoop, Kubernetes, Docker** trên cùng một cụm mà không cần quản lý tài nguyên thủ công ↩