ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN - TIN HỌC

SEMINAR - KHOA HỌC DỮ LIỆU

ĐỀ TÀI:

PHÂN TÍCH HIỆU NĂNG CỦA APACHE HADOOP VÀ APACHE SPARK TRÊN DỮ LIỆU LỚN

Giảng viên hướng dẫn: Hà Văn Thảo

Sinh viên thực hiện:

21110256 - Huỳnh Nguyễn Thế Dân

21280099 - Nguyễn Công Hoài Nam

21280118 - Lê Nguyễn Hoàng Uyên

Tp. Hồ Chí Minh, tháng 1 năm 2025

MỤC LỤC

Lời cảm ơn

Lời đầu tiên em xin cảm ơn đến thầy Hà Văn Thảo đã đưa đề tài và cung cấp hướng dẫn cũng như tài liệu tham khảo để chúng em thực hiện Seminar "Phân tích hiệu năng của Apache Hadoop và Apache Spark trên dữ liệu lớn".

Suốt quá trình thực hiện đồ án, chúng em đã có được những kiến thức, kinh nghiệm mới mẻ, bổ ích và có thể sẽ sử dụng đến trong quá trình làm khóa luận sau này của chúng em. Từ lúc bắt đầu đến lúc hoàn thành seminar, thầy thường xuyên hỏi han và cung cấp những gợi ý, chỉ dẫn nhiệt tình thông qua các buổi họp. Thông qua đó giúp chúng em có được cái nhìn tổng quan nhất cũng như có được những bước đi hợp lý hơn trong quá trình hoàn thiện. Trong quá trình đó, chúng em không thể tránh khỏi những sai sót. Chúng em rất mong nhận được những góp ý của thầy để những kế hoạch tiếp theo của chúng em sẽ càng hoàn thiện hơn và có được những kinh nghiệm quý giá trước khi ra trường đời.

Chúng em xin chân thành cảm ơn!

Các thành viên trong nhóm

Đặt vấn đề

Trong bối cảnh hiện nay, khi dữ liệu lớn (Big Data) trở thành yếu tố cốt lõi thúc đẩy các quyết định kinh doanh, nghiên cứu khoa học và các ứng dụng trí tuệ nhân tạo, việc xử lý và phân tích dữ liệu một cách hiệu quả là một thách thức lớn. Apache Hadoop (MapReduce) và Apache Spark là hai nền tảng hàng đầu được sử dụng rộng rãi để giải quyết các vấn đề liên quan đến dữ liệu lớn. Tuy nhiên, mỗi nền tảng lại có cách tiếp cận và hiệu suất xử lý khác nhau, đặt ra câu hỏi về việc lựa chọn nền tảng phù hợp cho từng loại bài toán và điều kiện hệ thống cụ thể.

Chúng tôi chọn đề tài "Phân tích hiệu năng của Apache Hadoop và Apache Spark trên dữ liệu lớn"nhằm đáp ứng nhu cầu hiểu rõ hơn về khả năng của hai nền tảng này trong việc xử lý dữ liệu lớn. Cả Hadoop và Spark đều có hơn 150 tham số có thể cấu hình, và sự kết hợp các tham số này sẽ ảnh hưởng đáng kể đến hiệu năng của hệ thống. Do đó, một câu hỏi được đặt ra là: Liệu việc tối ưu hóa các tham số có thể cải thiện hiệu năng của hệ thống khi xử lý tập dữ liệu lớn hay không?

Nghiên cứu này tập trung vào các tham số có tác động đáng kể đến hiệu năng, chẳng hạn như cách sử dụng tài nguyên, cách phân chia và trình tự xử lý dữ liệu đầu vào. Chúng tôi áp dụng phương pháp thử và sai để điều chỉnh các tham số, đồng thời tiến hành các thử nghiệm thực tế nhằm trả lời các câu hỏi như:

1. Việc tăng số lượng datanode có làm tăng tốc độ xử lý dữ liệu không?
2. Các tham số hệ thống có tác động cụ thể như thế nào đến hiệu năng?
3. Hiệu năng của MapReduce và Spark khác nhau như thế nào với từng bài toán cụ thể?

Ngoài ra, việc phân tích và so sánh hiệu năng giữa hai nền tảng này không chỉ giúp hiểu rõ hơn về khả năng xử lý dữ liệu mà còn đưa ra những định hướng sử dụng phù hợp với từng bài toán trong thực tế. Điều này có ý nghĩa đặc biệt quan trọng trong việc hỗ trợ các nhà nghiên cứu và doanh nghiệp lựa chọn công nghệ và cấu hình tối ưu nhất cho nhu cầu của mình.

Với mục tiêu đó, đề tài không chỉ mang tính thực tiễn cao mà còn mở ra nhiều hướng nghiên cứu mới, như phát triển các kỹ thuật tự động điều chỉnh tham số hoặc ứng dụng trên các tập dữ liệu lớn hơn, với điều kiện hệ thống liên kết mạng cục bộ hoặc môi trường phân tán phức tạp hơn.

1 Cơ sở lý thuyết

1.1 Hadoop

Hadoop là framework dựa trên 1 giải pháp tới từ Google để lưu trữ và xử lý dữ liệu lớn. Hadoop sử dụng giải thuật MapReduce xử lý song song các dữ liệu đầu vào, để phát triển các ứng dụng có thể thực hiện phân tích thống kê hoàn chỉnh trên dữ liệu số lượng lớn.

Hadoop gồm 2 tầng chính:

- Tầng xử lý và tính toán (MapReduce): MapReduce là một mô hình lập trình song song hóa để xử lý dữ liệu lớn trên một cụm gồm nhiều các máy tính thương mại (commodity hardware).
- Tầng lưu trữ (HDFS): HDFS cung cấp 1 giải pháp lưu trữ phân tán cũng được thiết kế để chạy trên các máy tính thương mại.

Ngoài 2 thành phần được đề cập ở trên thì Hadoop framework cũng gồm 2 module sau:

- Hadoop Common: là các thư viện, tiện ích viết bằng ngôn ngữ Java.
- Hadoop Yarn: hệ thống quản lý tài nguyên và lịch trình cho các ứng dụng.

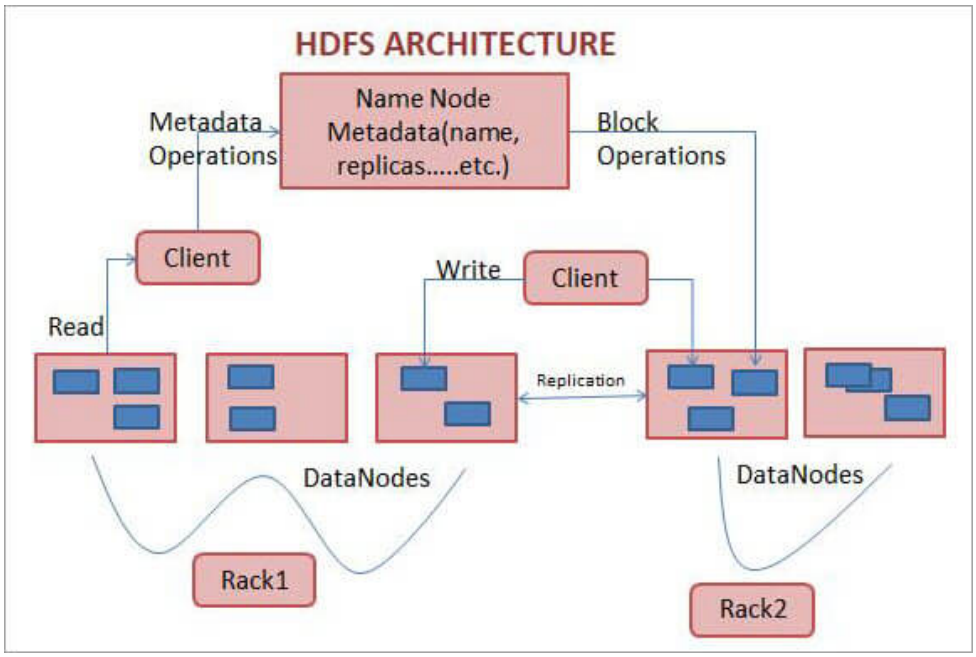
1.2 Hệ thống tệp phân tán Hadoop (HDFS)

Hadoop Distributed File System (HDFS) là một trong những hệ thống lớn nhất trong hệ sinh thái Hadoop và là hệ thống lưu trữ chính của Hadoop.

HDFS cung cấp khả năng lưu trữ tin cậy và chi phí hợp lý cho khối dữ liệu lớn, tối ưu cho các tập tin kích thước lớn (từ vài trăm MB cho tới vài TB). HDFS có không gian cây thư mục phân cấp giống như các hệ điều hành Unix, Linux.

Do các tính chất của dữ liệu lớn và hệ thống tập tin phân tán nên việc chỉnh sửa là rất khó khăn. Vì thế mà HDFS chỉ hỗ trợ việc ghi thêm dữ liệu vào cuối tệp (append), nếu muốn chỉnh sửa ở bất kì chỗ khác chỉ có cách là viết lại toàn bộ tệp với các phần sửa đổi và thay thế lại tệp cũ. HDFS tuân theo tiêu chí "ghi một lần và đọc nhiều lần".

HDFS bao gồm 1 Namenode chính và nhiều Datanode kết nối lại thành một cụm (cluster).



- **Namenode**: HDFS chỉ bao gồm duy nhất 1 namenode được gọi là master node thực hiện các nhiệm vụ:
  - Lưu trữ metadata của dữ liệu thực tế (tên, đường dẫn, block id, cấu hình datanode vị trí block,…).
  - Quản lý không gian tên của hệ thống file (ánh xạ các file name với các block, ánh xạ các block vào các datanode).
  - Quản lý cấu hình của cụm.
  - Chỉ định công việc cho datanode.
- **Datanode**: Chức năng của Datanode:
  - Lưu trữ dữ liệu thực tế.
  - Trực tiếp thực hiện và xử lý công việc (đọc/ghi dữ liệu).
- **Rack**: Theo thứ tự giảm dần từ cao xuống thấp thì ta có Rack > Node > Block. Rack là một cụm datanode cùng một đầu mạng, bao gồm các máy vật lí (tương đương một server hay 1 node) cùng kết nối chung 1 switch.
- **Block**: Block là một đơn vị lưu trữ của HDFS, các data được đưa vào HDFS sẽ được chia thành các block có các kích thước cố định (nếu không cấu hình thì mặc định nó là 128MB).
- **Replication**: Dữ liệu trong HDFS được nhân bản (replicate) trên nhiều DataNode để đảm bảo tính sẵn có và chống lại mất mát dữ liệu.
- **Cơ chế Heartbeat**: Heartbeat là cách liên lạc hay là cách để datanode cho namenode biết là nó còn sống. Định kì datanode sẽ gửi một heartbeat về cho namenode để namenode biết là datanode đó còn hoạt động. Nếu datanode không gửi heartbeat về cho namenode thì namenode coi rằng node đó đã hỏng và không thể thực hiện nhiệm vụ được giao. Namenode sẽ phân công task đó cho một datanode khác.

1.3 MapReduce

MapReduce là một kỹ thuật xử lý và là một mô hình lập trình cho tính toán phân tán để triển khai và xử lý dữ liệu lớn. MapReduce chứa 2 tác vụ quan trọng là **Map** và **Reduce**.

- **Map**: Đây là pha đầu tiên của chương trình. Có hai bước trong pha này: splitting and mapping. Một tập dữ liệu được chia thành các đơn vị bằng nhau được gọi là chunks trong bước phân tách (splitting). Hadoop bao gồm một RecordReader sử dụng TextInputFormat để chuyển đổi các phân tách đầu vào thành các cặp key-value.
- **Shuffle**: Đây là pha thứ hai diễn ra sau khi hoàn thành Mapper. Nó bao gồm hai bước chính: sắp xếp và hợp nhất (sorting and merging). Trong pha này, các cặp key-value được sắp xếp bằng cách sử dụng các key. Việc hợp nhất đảm bảo rằng các cặp key-value được kết hợp.
- **Reduce**: Trong pha Reduce, đầu ra của pha Shuffle được sử dụng làm đầu vào. Reducer xử lý đầu vào này hơn nữa để giảm các giá trị trung gian thành các giá trị nhỏ hơn. Nó cung cấp một bản tóm tắt của toàn bộ tập dữ liệu, ví dụ như là tính tổng, tìm max, min,… . Đầu ra của pha này được lưu trữ trong HDFS.

1.4 Spark

Trước khi Spark ra đời, Hadoop đang là một công cụ mạnh mẽ và phổ biến, tuy nhiên Hadoop có những hạn chế nhất định và Spark ra đời để cải thiện các hạn chế đó.

Mô hình lập trình của MapReduce hiện tại dựa trên acyclic data flow (dòng dữ liệu tuần hoàn) từ một bộ nhớ ổn định tới một bộ nhớ ổn định, nôm na là Hadoop nhận đầu vào từ một bộ nhớ ổn định và sau mỗi Task hoàn thành kết quả được lưu lại trên một bộ nhớ ổn định. Điều này khiến cho việc nếu muốn dùng lại dữ liệu đó chúng ta phải đọc lại dữ liệu từ bộ nhớ.

Machine learning phát triển dẫn tới việc khi chúng ta chạy một thuật toán machine learning trên Hadoop sẽ mất thời gian. Vì các thuật toán Machine learning đa số là các thuật toán thuộc dạng Iterative algorithms (lặp đi lặp lại) nên việc sau mỗi lần lặp dữ liệu được lưu tại một bộ nhớ ổn định và được đọc lại để làm đầu vào cho vòng lặp tiếp theo là tốn rất nhiều thời gian.

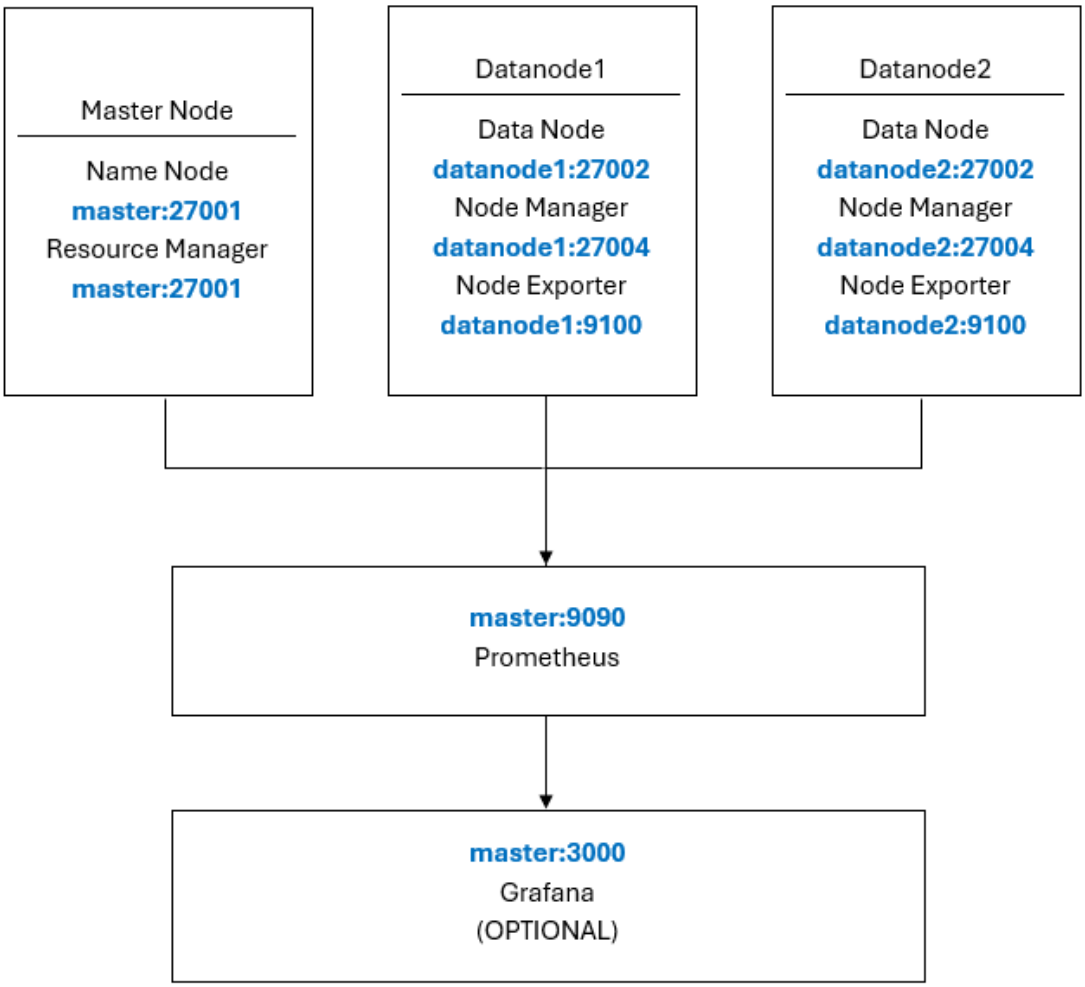
Spark được tạo ra để giải quyết các hạn chế của MapReduce bằng cách thực hiện xử lý trong bộ nhớ, giảm số bước trong một công việc và bằng cách sử dụng lại dữ liệu trên nhiều thao tác song song. Với Spark, chỉ cần thực hiện một bước là dữ liệu được đọc vào bộ nhớ, các thao tác được thực hiện và kết quả được ghi lại, từ đó giúp việc thực thi nhanh hơn nhiều. Spark cũng sử dụng lại dữ liệu bằng cách sử dụng bộ nhớ đệm nằm trong bộ nhớ để tăng tốc đáng kể các thuật toán máy học liên tục gọi một hàm trên cùng một tập dữ liệu. Việc tái sử dụng dữ liệu được thực hiện thông qua việc tạo DataFrames, một khung trừu tượng trên Tập dữ liệu phân tán linh hoạt (RDD), là một tập hợp các đối tượng được lưu trong bộ nhớ đệm và được tái sử dụng trong nhiều hoạt động của Spark. Điều này làm giảm đáng kể độ trễ khiến Spark có tốc độ nhanh hơn nhiều lần so với MapReduce, đặc biệt là khi thực hiện máy học và phân tích tương tác.

1.5 Hệ giám sát Hadoop

Prometheus là hệ thống giám sát mã nguồn mở hỗ trợ giám sát hiệu suất hệ thống và ứng dụng.

Khi tích hợp với Hadoop, hệ thống định kỳ truy cập vào các endpoint HTTP - các thành phần của Hadoop (HDFS, YARN, MapReduce,...) cung cấp thông qua các exporter (như Node Exporter hoặc JMX Exporter).

Dữ liệu thu thập được lưu trữ ở dạng chuỗi thời gian (time-series) và sử dụng ngôn ngữ truy vấn PromQL để trích xuất và phân tích các số liệu như hiệu suất hệ thống, tài nguyên máy chủ và thông tin Hadoop (HDFS, YARN).

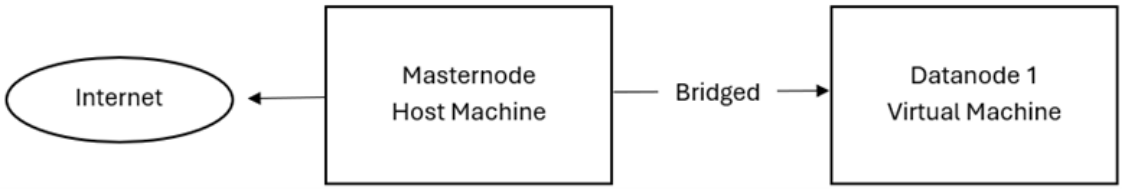


Hình 1: Sơ đồ giám sát hệ thống

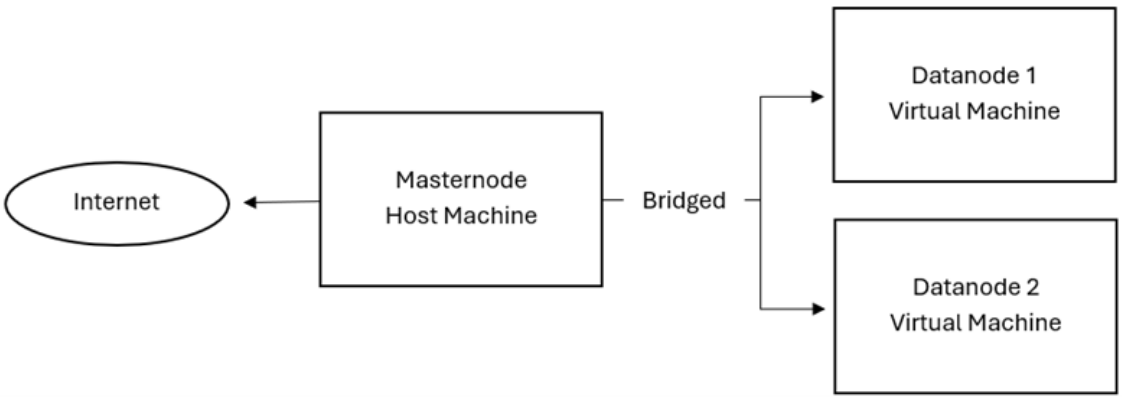
2 Thực nghiệm

2.1 Kiến trúc hệ thống

Do hạn chế về tài nguyên máy tính (như CPU, RAM), hệ thống sẽ được thực nghiệm trên hai trường hợp: cấu hình với 1 masternode và 1 datanode, cấu hình với 1 masternode và 2 datanodes; cùng được cài đặt trên một máy tính cá nhân liên kết với các máy ảo.



Hình 2: Hệ thống gồm 1 masternode và 1 datanode



Hình 3: Hệ thống gồm 1 masternode và 2 datanodes

Các thông số kỹ thuật chi tiết được trình bày trong bảng sau:

Cấu hình masternode	CPU	Intel i7-1065G7 (8) @ 3.900GHz
	Operating System	Ubuntu 24.04 LTS x86_64
	Main memory	12GB
	Local Storage	SSD 512GB
Cấu hình datanode	CPU	Intel i7-1065G7 (8) @ 3.900GHz
	Operating System	Ubuntu Server 24.04.1
	Number of nodes	1 hoặc 2
	Datanode	Memory: 2560MB
		Local Storage: 100GB
		CPU cores: 2
Phần mềm	JDK	8
	Hadoop	3.2
	Spark	3.0
Bài toán (Workload)	Micro Benchmarks	WordCount, và TeraSort

Bảng 1: Cấu hình tài nguyên hệ thống

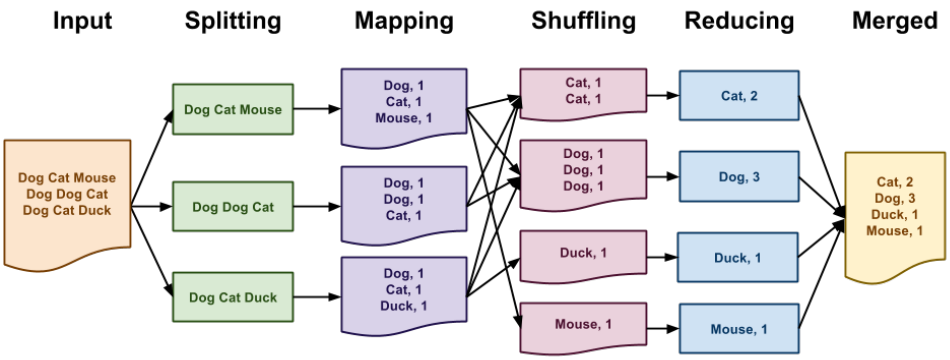
Quá trình thực nghiệm sẽ được chạy trên cả MapReduce và Spark. Chúng tôi chọn Yarn làm trình quản lý tài nguyên, để theo dõi tình hình của từng nút đang hoạt động và theo dõi chi tiết của từng công việc cùng với lịch sử của nó. Chúng tôi đã sử dụng Prometheus để theo dõi và lập các Dashboard tự động cho các bài toán chạy trên Spark và MapReduce.

2.2 Dữ liệu thực nghiệm

Trong mỗi lần thực nghiệm, chúng tôi đã sử dụng các tập dữ liệu với kích thước tăng dần (1GB, 5GB, 10GB, 15GB, 20GB, 25GB, 30GB) với cùng cấu hình tham số, để theo dõi hiệu năng của hệ thống thông qua các tiêu chí như thời gian thực thi, thông lượng, speedup. Báo cáo này sẽ tiến hành thực nghiệm trên 2 bài toán WordCount và TeraSort để đo hiệu năng của Hadoop và Spark.

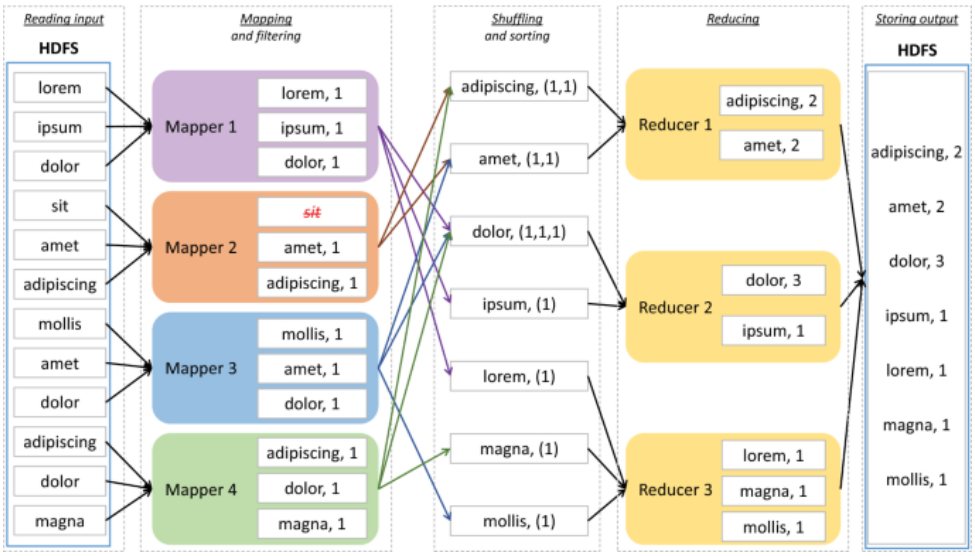
- Bài toán WordCount phụ thuộc vào số lượng từ khác nhau và tần suất xuất hiện của chúng. Dữ liệu đầu vào được tạo tự động bởi hàm *RandomTextWriter* - một hàm có sẵn trong Java. Dựa trên cách thức hoạt động của Mapreduce, hàm Map sẽ chia dữ liệu thành các từ riêng lẻ và tạo dữ liệu

trung gian cho hàm Reduce dưới dạng key-value; sau đó hàm Reduce sẽ sử dụng dữ liệu trung gian để tính tổng các value tương ứng với mỗi key. Cuối cùng, bài toán trả về danh sách các từ riêng lẻ cùng tần suất xuất hiện của chúng.



Hình 4: Bài toán WordCount

- Bài toán TeraSort, là bài toán sắp xếp dữ liệu dựa trên cơ chế hoạt động của Mapreduce. Dữ liệu đầu vào được tạo ra bởi hàm *TeraGen* - hàm được triển khai trong Java. Hàm Map sẽ chia dữ liệu thành các từ riêng lẻ và tạo dữ liệu trung gian cho hàm Reduce dưới dạng key-value; sau đó hàm Reduce sẽ sử dụng dữ liệu trung gian để sắp xếp dựa trên key tương ứng. Cuối cùng, hàm TeraValidate sẽ xác thực lại dữ liệu đã sắp xếp.



Hình 5: Bài toán TeraSort

2.3 Điều chỉnh các tham số

Các tham số này trong Apache Hadoop và Apache Spark được điều chỉnh theo phương pháp thử và sai, qua đó chúng tôi muốn tìm ra những tham số quan trọng có ảnh hưởng đến hiệu năng của hệ thống.

Chúng tôi đã tiến hành sử dụng Apache Hadoop và Apache Spark với các thiết lập tham số khác nhau liên quan đến Resource utilization, Input Split, và Shuffle. Cụ thể như sau:

Các tham số cấu hình Hadoop

- **Resource utilization**

  - **mapreduce.reduce.memory.mb**: Tham số chỉ mức bộ nhớ tối đa trong pha Reduce nhằm đảm bảo tác vụ không sử dụng quá nhiều bộ nhớ, tránh trường hợp hệ thống bị treo hoặc lỗi do thiếu bộ nhớ.
  - **mapred.job.reduces**: Tham số này chỉ số lượng task mà một pha Reduce cần làm.
  - **mapreduce.reduce.cpu.vcores**: Tham số này xác định số lượng vCore (logical CPU cores) mà mỗi task sẽ sử dụng. Cấu hình này giúp tối ưu hóa hiệu suất khi số lượng core càng lớn.

- **Input Split**

  - **mapreduce.input.fileinputformat.split.minsize**: Tham số này xác định kích thước tối thiểu của các phần dữ liệu được phân chia trong mỗi block khi sử dụng hệ thống lưu trữ phân tán HDFS. VD: mapreduce.input.fileinputformat.split.minsize có giá trị mặc định là 128MB, có nghĩa là toàn bộ dữ liệu đầu vào sẽ được phân chia thành các block có kích thước 128MB.

- **Shuffle**

  - **mapreduce.task.io.sort.mb**: Tham số này quy định mức bộ nhớ tối đa mà mỗi task sẽ sử dụng khi sắp xếp dữ liệu trong quá trình shuffle (chuyển đổi dữ liệu từ map tới reduce).
  - **mapreduce.reduce.shuffle.parallelcopies**: Tham số này chỉ số lượng bản sao song song trong quá trình shuffle khi chuyển dữ liệu từ pha Map tới Reduce. Sử dụng nhiều bản sao song song có thể giúp giảm thiểu độ trễ trong quá trình Reduce.
  - **mapreduce.task.io.sort.factor**: Tham số này quy định số lượng luồng (threads) được sử dụng để hợp nhất (merge) dữ liệu trong quá trình Shuffle đến pha Reduce.

Các tham số cấu hình Spark

- **Resource utilization**

  - **num-executors**: Tham số này chỉ định số lượng executors (các tiến trình xử lý) sẽ được khởi tạo Spark. Mỗi executor là một tiến trình độc lập, chịu trách nhiệm thực thi các task và lưu trữ dữ liệu.
  - **executor-cores**: Tham số này xác định số lượng vCore (logical CPU cores) mà mỗi executor sẽ sử dụng. Cấu hình này ảnh hưởng đến khả năng xử lý song song của mỗi executor. Mỗi core sẽ thực hiện một task trong quá trình tính toán, và giá trị này càng cao thì khả năng xử lý song song của executor càng mạnh.
  - **executor-memory**: Tham số này chỉ định lượng bộ nhớ mà mỗi executor sẽ sử dụng. Cấu hình này ảnh hưởng đến lượng dữ liệu mà executor có thể giữ trong bộ nhớ để xử lý. Nếu bộ nhớ được cấu hình quá nhỏ, executor sẽ phải sử dụng bộ nhớ ngoài (disk), điều này có thể làm giảm hiệu suất của hệ thống.
  - **driver-memory**: Tham số này xác định lượng bộ nhớ mà driver của Spark sẽ sử dụng. Driver chịu trách nhiệm khởi tạo và quản lý các executor, đồng thời duy trì thông tin điều phối và quản lý tiến trình. Việc cấu hình bộ nhớ hợp lý cho driver là rất quan trọng để đảm bảo rằng quá trình điều phối không bị gián đoạn.

- **Input Split**

  - **spark.hadoop.MapReduce.input.fileinputformat.split.minsize**: Tham số này xác định kích thước tối thiểu của các phần dữ liệu được phân chia trong mỗi block khi sử dụng hệ thống lưu trữ phân tán HDFS. VD: mapreduce.input.fileinputformat.split.minsize có giá trị mặc định là 128MB, có nghĩa là toàn bộ dữ liệu đầu vào sẽ được phân chia thành các block có kích thước 128MB. Cấu hình này giúp phân chia dữ liệu hợp lý hơn khi thực hiện các tác vụ tính toán.

- **Shuffle**
  - **spark.shuffle.file.buffer** Tham số này xác định bộ đệm (buffer) file trong quá trình shuffle. Bộ đệm được sử dụng để lưu trữ tạm thời dữ liệu trong quá trình chuyển dữ liệu giữa các task. Giá trị này ảnh hưởng đến độ trễ và hiệu suất của quá trình shuffle.
  - **spark.reducer.maxSizeInFlight**: Tham số này xác định kích thước tối đa của dữ liệu mà một task Reduce có thể xử lý trong một lượt. Nếu giá trị này quá nhỏ, quá trình shuffle có thể bị chậm, còn nếu quá lớn, có thể gây ra vấn đề về bộ nhớ.
  - **spark.default.parallelism**: Tham số này chỉ định mức độ song song (parallelism) mặc định trong các tác vụ Spark, đặc biệt là trong quá trình shuffle. Cấu hình này giúp Spark xác định số lượng task sẽ chạy song song trong khi xử lý dữ liệu.
  - **dfs.replication**: Tham số này quy định số lượng bản sao của mỗi block dữ liệu được lưu trữ trong HDFS. Cấu hình này giúp tăng độ bền và độ tin cậy của dữ liệu trong trường hợp một hoặc nhiều bản sao bị mất.

Các thông số kỹ thuật chi tiết về cấu hình tham số của Hadoop và Spark được trình bày như sau:

Resource utilization	mapreduce.reduce.memory	256MB
	mapred.job.reduces	16
	mapreduce.reduce.cpu.vcores	4
Input Split	mapreduce.input.fileinputformat.split.minsize	128MB (default), 256MB, 512MB, 1024MB
Shuffle	mapreduce.task.io.sort.mb	25 (default), 50, 75, 100
	mapreduce.reduce.shuffle.parallelcopies	50 (default), 100, 150, 200
	mapreduce.task.io.sort.factor	15 (default), 30, 45, 60

Bảng 2: Các tham số cấu hình Hadoop

Resource utilization	num-executors	2, 4, 6
	executor-cores	1
	executor-memory	640MB
	driver-memory	640MB
Input Split	spark.hadoop.MapReduce.input. fileinputformat.split.minsize	128MB (default), 256MB, 512MB, 1024MB
Shuffle	spark.shuffle.file.buffer	16KB, 32KB (default), 48KB, 64KB
	spark.reducer.maxSizeInFlight	32MB, 48MB (default), 64MB, 96MB
	spark.default.parallelism	4, 8, 10, 12
	dfs.replication	1

Bảng 3: Các tham số cấu hình Spark

Chúng tôi chọn Yarn làm trình quản lý tài nguyên, để theo dõi tình hình của từng nút đang hoạt động và theo dõi chi tiết của từng công việc cùng với lịch sử của nó. Chúng tôi đã sử dụng Prometheus để theo dõi và lập các Dashboard tự động các khối lượng công việc được chọn đang chạy trên Spark và MapReduce.

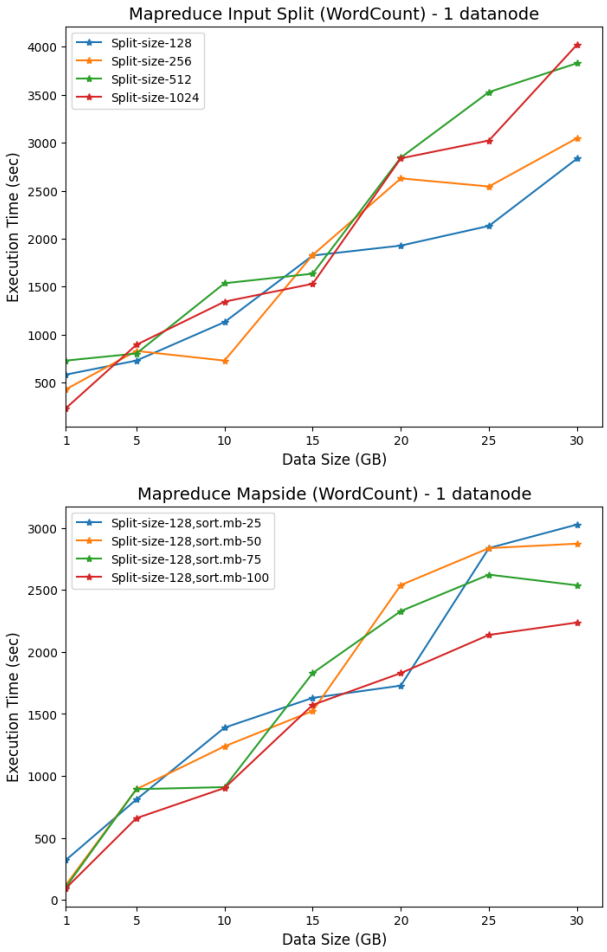
3 Kết quả

3.1 Phân tích MapReduce

3.1.1 MapReduce - 1 datanode

Bài toán được thiết lập với dữ liệu đầu vào có kích thước tăng dần từ (1G-30GB), đang hoạt động với 1 datanode, reduce memory là 256MB, jobs reduce là 16, và reduce cpu vcores là 4. Thực nghiệm tiến hành thay đổi các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle) để chứng minh rằng chúng có ảnh hưởng đến hiệu năng của hệ thống.

Bài toán WordCount



Hình 6: Biểu đồ thời gian thực thi của MapReduce cho bài toán WordCount

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Sort (MB)**: là quy định mức bộ nhớ tối đa mà mỗi task sẽ sử dụng khi sắp xếp dữ liệu trong quá trình shuffle.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

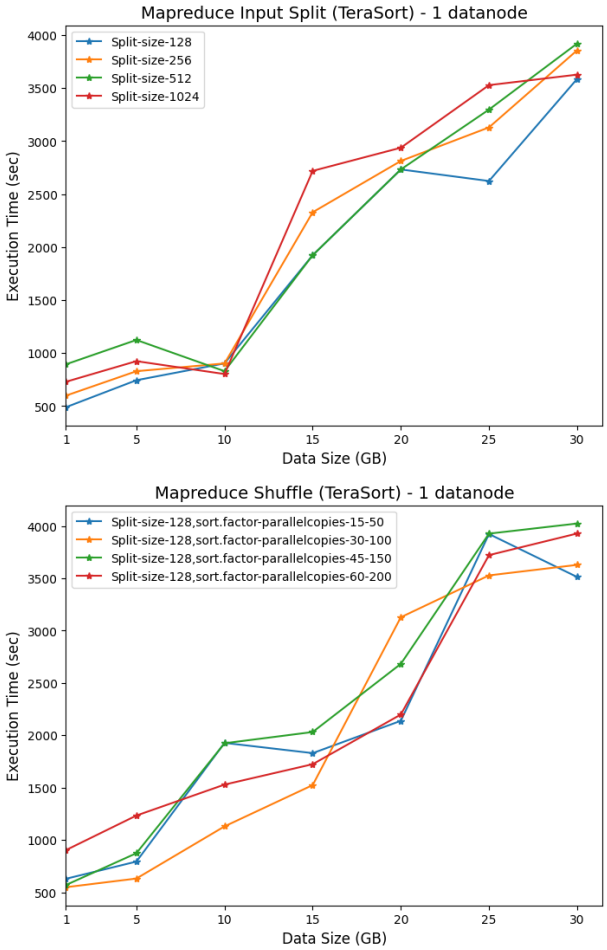
	Default		Input Split		**Optimization**
Input	Input Split	Duration(s)	Input Split	Duration(s)	
1GB	128MB	582	**1024MB**	**232**	**60.14%**
5GB	128MB	728	**128MB**	**728**	
10GB	128MB	1130	**256MB**	**728**	**35.58%**
15GB	128MB	1824	**1024MB**	**1529**	**16.17%**
20GB	128MB	1927	**128MB**	**1927**	
25GB	128MB	2133	**128MB**	**2133**	
30GB	128MB	2835	**128MB**	**2835**	

	Default		Mapside		Optimization
Input	Sort	Duration(s)	Sort	Duration(s)	
1GB	25MB	321	**100MB**	**93**	**71.03%**
5GB	25MB	809	**100MB**	**659**	**18.54%**
10GB	25MB	1389	**100MB**	**902**	**35.06%**
15GB	25MB	1629	**50MB**	**1524**	**6.45%**
20GB	25MB	1728	**25MB**	**1728**	
25GB	25MB	2839	**100MB**	**2138**	**24.69%**
30GB	25MB	3028	**100MB**	**2237**	**26.12%**

Hình trên cho thấy thời gian thực thi của MapReduce phụ thuộc vào kích thước của dữ liệu đầu vào và các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle). Thời gian thực thi của MapReduce WordCount với tham số **InputSplit(1024MB)** và tham số **Shuffle(Sort 100MB)** đạt được phần trăm tối ưu cao nhất trên toàn bộ tập dữ liệu (**60.14%** và **71.03%**). Trên thực tế, tham số **InputSplit(128MB)** - giá trị mặc định đạt được hiệu quả tối ưu hơn trên phần lớn các mức dữ liệu đầu vào (Input) nên MapReduce chạy với giá trị Input Split mặc định là rất phù hợp.

Do đó, chúng ta có thể kết luận rằng việc các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle), có thể được coi là một yếu tố quan trọng trong việc nâng cao hiệu năng của MapReduce WordCount khi thực thi các tập dữ liệu có kích thước nhỏ.

Bài toán TeraSort



Hình 7: Biểu đồ thời gian thực thi của MapReduce cho bài toán WordCount

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Sort factor**: tham số chỉ số lượng luồng (threads) để hợp nhất dữ liệu trong quá trình shuffle, giá trị mặc định là 15.
- **Parallelcopies**: tham số chỉ số lượng bản sao song song trong quá trình shuffle, giá định mặc định là 50.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

	Default		Input Split		**Optimization**
Input	Input Split	Duration(s)	Input Split	Duration(s)	
1GB	128MB	489	**128MB**	**489**	
5GB	128MB	744	**128MB**	**744**	
10GB	128MB	903	**1024MB**	**802**	**11.18%**
15GB	128MB	1924	**128MB**	**1924**	
20GB	128MB	2733	**128MB**	**2733**	
25GB	128MB	2624	**128MB**	**2624**	
30GB	128MB	3587	**128MB**	**3587**	

	Default		Mapside		**Optimization**
Input	Sort Factor - Parallelcopies	Duration(s)	Sort Factor - Parallelcopies	Duration(s)	
1GB	15-50	628	**30-100**	**548**	**12.74%**
5GB	15-50	793	**30-100**	**631**	**20.43%**
10GB	15-50	1927	**30-100**	**1129**	**41.41%**
15GB	15-50	1829	**30-100**	**1524**	**16.68%**
20GB	15-50	2138	**15-50**	**2138**	
25GB	15-50	3924	**30-100**	**3528**	**10.09%**
30GB	15-50	3513	**15-50**	**3513**	

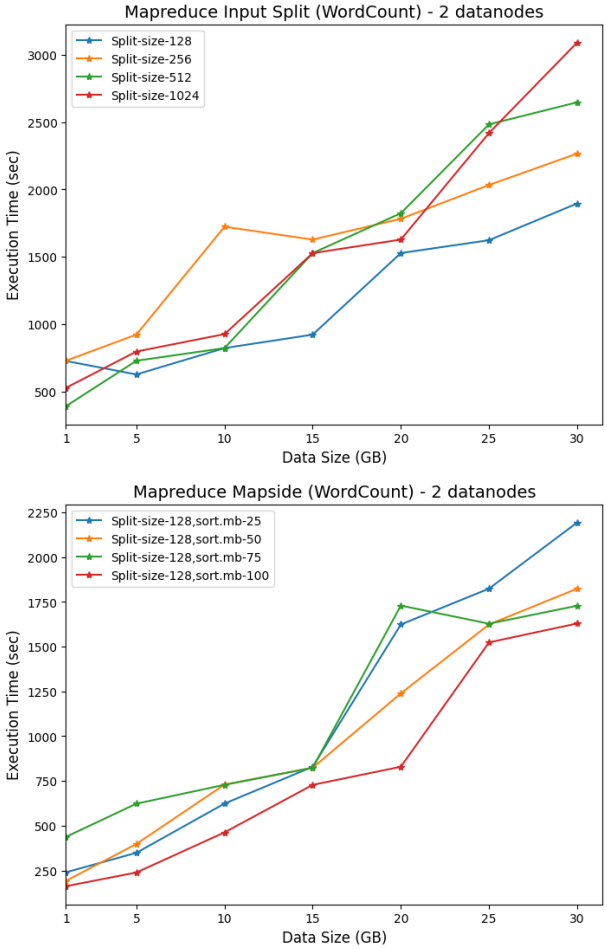
Hình trên cho thấy thời gian thực thi của MapReduce phụ thuộc vào kích thước của dữ liệu đầu vào và các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle). Thời gian thực thi của MapReduce TeraSort với tham số **InputSplit(1024MB)** và tham số **Shuffle(Sort Factor 30, Parallelcopies 100)** đạt được phần trăm tối ưu cao nhất trên toàn bộ tập dữ liệu (**11.18%** và **41.41%**). Trên thực tế, tham số **InputSplit(128MB)** - giá trị mặc định đạt được hiệu quả tối ưu hơn trên phần lớn các mức dữ liệu đầu vào (Input) nên MapReduce chạy với giá trị Input Split mặc định là rất phù hợp.

Do đó, chúng ta có thể kết luận rằng việc các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle), có thể được coi là một yếu tố quan trọng trong việc nâng cao hiệu năng của MapReduce TeraSort khi thực thi các tập dữ liệu có kích thước nhỏ.

3.1.2 MapReduce - 2 datanode(s)

Bài toán được thiết lập với dữ liệu đầu vào có kích thước tăng dần từ (1G-30GB), đang hoạt động với 2 datanode(s), reduce memory là 256MB, jobs reduce là 16, và reduce cpu vcores là 4. Thực nghiệm tiến hành thay đổi các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle) để chứng minh rằng chúng có ảnh hưởng đến hiệu năng của hệ thống.

Bài toán WordCount



Hình 8: Biểu đồ thời gian thực thi của MapReduce cho bài toán WordCount

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Sort (MB)**: là quy định mức bộ nhớ tối đa mà mỗi task sẽ sử dụng khi sắp xếp dữ liệu trong quá trình shuffle.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

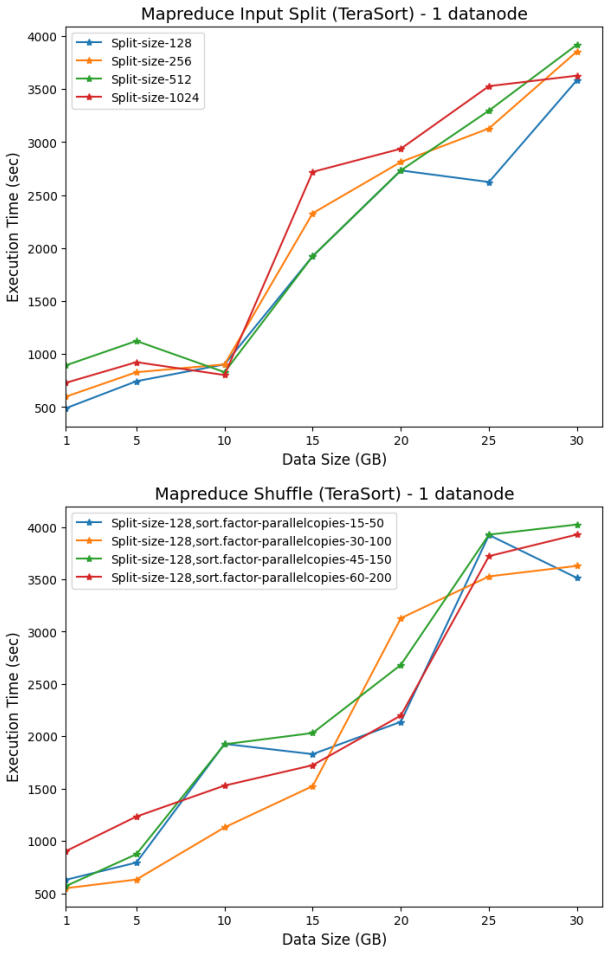
	Default		Mapside		Optimization
Input	Input Split	Duration(s)	Input Split	Duration(s)	
1GB	128MB	728	**512MB**	**393**	**46.02%**
5GB	128MB	628	**128MB**	**628**	
10GB	128MB	824	**128MB**	**824**	
15GB	128MB	924	**128MB**	**924**	
20GB	128MB	1528	**128MB**	**1528**	
25GB	128MB	1624	**128MB**	**1624**	
30GB	128MB	1896	**128MB**	**1896**	

	Default		Input Split		Optimization
Input	Sort	Duration(s)	Sort	Duration(s)	
1GB	25MB	240	**100MB**	**162**	**32.5%**
5GB	25MB	349	**100MB**	**239**	**31.52%**
10GB	25MB	624	**100MB**	**463**	**25.80%**
15GB	25MB	829	**100MB**	**728**	**12.18%**
20GB	25MB	1624	**100MB**	**829**	**48.95%**
25GB	25MB	1824	**100MB**	**1524**	**16.45%**
30GB	25MB	2193	**100MB**	**1629**	**25.72%**

Hình trên cho thấy thời gian thực thi của MapReduce phụ thuộc vào kích thước của dữ liệu đầu vào và các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle). Thời gian thực thi của MapReduce WordCount với tham số **InputSplit(512MB)** và tham số **Shuffle(Sort 100MB)** đạt được phần trăm tối ưu cao nhất trên toàn bộ tập dữ liệu (**46.02%** và **48.95%**). Trên thực tế, tham số **InputSplit(128MB)** - giá trị mặc định đạt được hiệu quả tối ưu trên tất cả các mức dữ liệu đầu vào (Input) nên MapReduce chạy với giá trị Input Split mặc định là rất phù hợp.

Do đó, chúng ta có thể kết luận rằng việc các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle), có thể được coi là một yếu tố quan trọng trong việc nâng cao hiệu năng của MapReduce WordCount khi thực thi các tập dữ liệu có kích thước nhỏ.

Bài toán TeraSort



Hình 9: Biểu đồ thời gian thực thi của MapReduce cho bài toán WordCount

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Sort factor**: tham số chỉ số lượng luồng (threads) để hợp nhất dữ liệu trong quá trình shuffle, giá trị mặc định là 15.
- **Parallelcopies**: tham số chỉ số lượng bản sao song song trong quá trình shuffle, giá định mặc định là 50.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

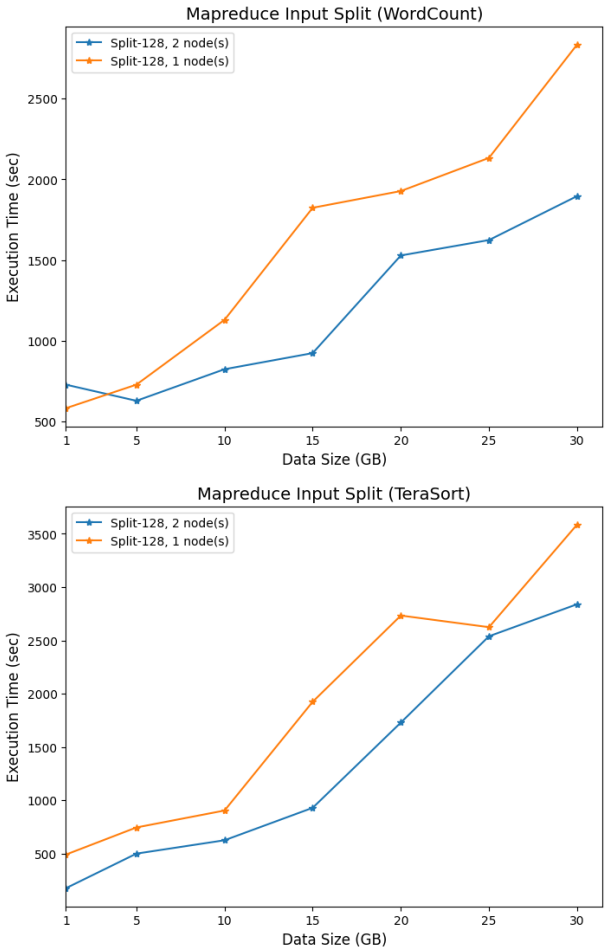
	Default		Input Split		Optimization
Input	Input Split	Duration(s)	Input Split	Duration(s)	
1GB	128MB	174	**1024MB**	**83**	**52.30%**
5GB	128MB	498	**256MB**	**282**	**43.37%**
10GB	128MB	624	**128MB**	**624**	
15GB	128MB	928	**128MB**	**928**	
20GB	128MB	1728	**1024MB**	**1628**	**5.79%**
25GB	128MB	2539	**128MB**	**2539**	
30GB	128MB	2839	**1024MB**	**2712**	**4.47%**

	Default		Input Split		Optimization
Input	Sort Factor-Parallelcopies	Duration(s)	Sort Factor-Parallelcopies	Duration(s)	
1GB	15-50	259	**30-100**	**109**	**57.92%**
5GB	15-50	624	**60-200**	**424**	**32.05%**
10GB	15-50	824	**30-100**	**762**	**7.52%**
15GB	15-50	1529	**60-200**	**1239**	**18.97%**
20GB	15-50	1724	**30-100**	**1627**	**5.63%**
25GB	15-50	2438	**60-200**	**2219**	**8.98%**
30GB	15-50	3130	**30-100**	**2938**	**6.13%**

Hình trên cho thấy thời gian thực thi của MapReduce phụ thuộc vào kích thước của dữ liệu đầu vào và các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle). Thời gian thực thi của MapReduce TeraSort với tham số **InputSplit(1024MB)** và tham số **Shuffle(Sort Factor 30, Parallelcopies 100)** - tham số **Shuffle(Sort Factor 60, Parallelcopies 200)** đạt được phần trăm tối ưu cao nhất trên toàn bộ tập dữ liệu hơn **50%**. Trên thực tế, tham số **InputSplit(128MB)** - giá trị mặc định đạt được hiệu quả tối ưu hơn trên phần lớn các mức dữ liệu đầu vào (Input) nên MapReduce chạy với giá trị Input Split mặc định là rất phù hợp.

Do đó, chúng ta có thể kết luận rằng việc các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle), có thể được coi là một yếu tố quan trọng trong việc nâng cao hiệu năng của MapReduce TeraSort khi thực thi các tập dữ liệu có kích thước nhỏ.

3.1.3 So sánh



Hình 10: Biểu đồ thời gian thực thi của MapReduce khi thay đổi số lượng datanode

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

WordCount	**1 Datanode**		**2 Datanodes**		**Speed up**
Input	Input Split	Duration(s)	Input Split	Duration(s)	**(1datanode/2datanodes)**
1GB	128MB	582	128MB	728	0.8
5GB	128MB	728	128MB	628	1.16
10GB	128MB	1130	128MB	824	1.37
15GB	128MB	1824	128MB	924	1.97
20GB	128MB	1927	128MB	1528	1.26
25GB	128MB	2133	128MB	1624	1.31
30GB	128MB	2835	128MB	1896	1.5

TeraSort	**1 Datanode**		**2 Datanodes**		**Speed up**
Input	Input Split	Duration(s)	Input Split	Duration(s)	**(1datanode/2datanodes)**
1GB	128MB	489	128MB	174	2.81
5GB	128MB	744	128MB	498	1.49
10GB	128MB	903	128MB	624	1.45
15GB	128MB	1924	128MB	928	2.07
20GB	128MB	2733	128MB	1728	1.58
25GB	128MB	2624	128MB	2539	1.03
30GB	128MB	3587	128MB	2839	1.26

Hình trên so sánh thời gian thực thi của MapReduce hoạt động với 1 datanode và 2 datanodes khi thực nghiệm trên 2 bài toán WordCount, TeraSort với các tham số phân chia dữ liệu đầu vào khác nhau (InputSplit). Đối với bài toán *WordCount*, ta tiến hành phân tích kết quả của InputSplit có giá trị **128MB**, vì tham số này thu được hiệu năng cao nhất trong quá trình thực nghiệm của MapReduceWordCount trên 1 và 2 datanode(s). Đối với bài toán *TeraSort*, ta tiến hành phân tích kết quả của InputSplit có giá trị **128MB** vì tham số này thu được hiệu năng cao nhất trong quá trình thực nghiệm của MapReduceTeraSort trên 1 và 2 datanode(s). Chúng ta nhận thấy rằng khi tăng số lượng datanode, hiệu năng của hệ thống được cải thiện rõ rệt **(gấp khoảng 2 lần)**.

Do đó, chúng ta có thể kết luận rằng việc tăng số lượng datanode có thể cải thiện hiệu năng của hệ thống và tăng tốc độ xử lý.

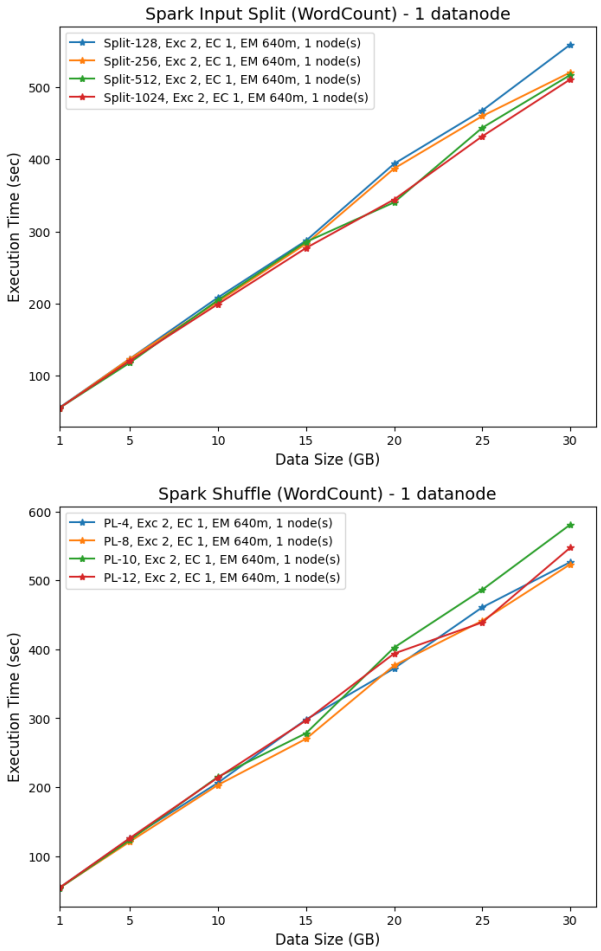
3.2 Phân tích Spark

3.2.1 Spark - 1 datanode

Bài toán được thiết lập với dữ liệu đầu vào có kích thước tăng dần từ (1G-30GB), đang hoạt động với 1 datanode, executor là 2, executor core là 1, và executor memory là 640m. Thực nghiệm tiến hành thay đổi các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle) để chứng minh rằng chúng có ảnh hường đến hiệu năng của hệ thống.

Bài toán WordCount

Bài toán được thiết lập với dữ liệu đầu vào có kích thước tăng dần từ (1G-30GB), đang hoạt động với 1 datanode, executor là 2, executor core là 1, và executor memory là 640m. Thực nghiệm tiến hành thay đổi các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle) để chứng minh rằng chúng có ảnh hưởng đến hiệu năng của hệ thống.



Hình 11: Biểu đồ thời gian thực thi của Spark cho bài toán WordCount

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Parallelism**: là số luồng xử lý song song, giá trị mặc định là 4.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

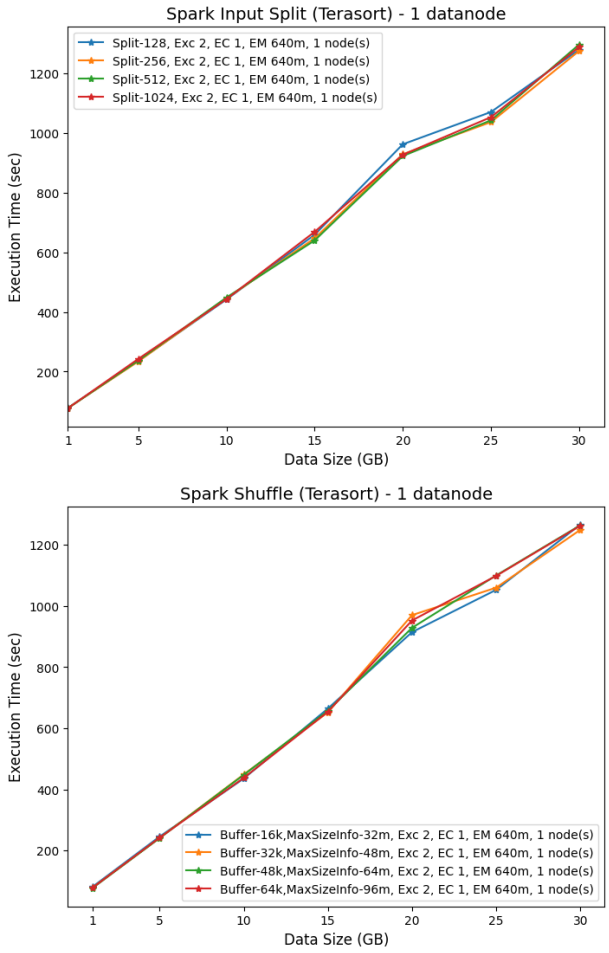
	Default		Input Split		Optimization
Input	Input Split	Duration(s)	Input Split	Duration(s)	
1GB	128MB	56	**256MB**	**53**	**5.36%**
5GB	128MB	124	**512MB**	**118**	**4.84%**
10GB	128MB	208	**1024MB**	**199**	**4.33%**
15GB	128MB	287	**1024MB**	**277**	**3.48%**
20GB	128MB	394	**512MB**	**340**	**13.71%**
25GB	128MB	468	**1024MB**	**432**	**7.69%**
30GB	128MB	559	**1024MB**	**511**	**8.59%**

	Default		Shuffle		Optimization
Input	Parallelism	Duration(s)	Parallelism	Duration(s)	
1GB	4	54	**10**	**53**	**1.85%**
5GB	4	125	**8**	**121**	**3.20%**
10GB	4	206	**8**	**203**	**1.46%**
15GB	4	298	**8**	**270**	**9.40%**
20GB	4	372	**4**	**372**	
25GB	4	461	**12**	**439**	**4.77%**
30GB	4	527	**8**	**524**	**0.457%**

Hình 11 cho thấy thời gian thực thi của Spark phụ thuộc vào kích thước của dữ liệu đầu vào và các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle). Thời gian thực thi của Spark WordCount với tham số **InputSplit(512MB)** và tham số **Shuffle(Parallelism 8)** đạt được phần trăm tối ưu cao nhất trên toàn bộ tập dữ liệu (**13.71%** và **9.4%**). Trên thực tế, tham số **InputSplit(1024MB)** đạt được hiệu quả tối ưu hơn trên phần lớn các mức dữ liệu đầu vào (Input) - trên 10GB, cải thiện hiệu năng xử lý cao hơn 3.48%.

Do đó, chúng ta có thể kết luận rằng việc các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle), có thể được coi là một yếu tố quan trọng trong việc nâng cao hiệu năng của Spark WordCount khi thực thi các tập dữ liệu có kích thước nhỏ.

Bài toán TeraSort



Hình 12: Biểu đồ thời gian thực thi của Spark cho bài toán TeraSort

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là

128MB.

- **Buffer(KB)**: Bộ đệm để lưu trữ dữ liệu tạm thời, giá trị mặc định là 32KB.
- **MaxSizeInfo**: Kích thước dữ liệu tối đa có thể xử lý, giá định mặc định là 48MB.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

	Default		Input Split		Optimization
Input	Input Split	Duration(s)	Input Split	Duration(s)	
1GB	128MB	79	**1024MB**	**78**	**1.27%**
5GB	128MB	236	**256MB**	**234**	**0.85%**
10GB	128MB	442	**128MB**	**442**	
15GB	128MB	659	**512MB**	**640**	**2.88%**
20GB	128MB	962	**512MB**	**922**	**4.16%**
25GB	128MB	1070	**256MB**	**1036**	**3.18%**
30GB	128MB	1279	**256MB**	**1274**	**0.39%**

	Default		Shuffle		Optimization
Input	Buffer	Duration(s)	Buffer	Duration(s)	
1GB	32KB	78	**48KB**	**77**	**1.28%**
5GB	32KB	241	**32KB**	**241**	
10GB	32KB	449	**16KB**	**436**	**2.90%**
15GB	32KB	652	**32KB**	**652**	
20GB	32KB	970	**16KB**	**915**	**5.67%**
25GB	32KB	1060	**16KB**	**1053**	**0.66%**
30GB	32KB	1250	**32KB**	**1250**	

Hình trên cho thấy thời gian thực thi của Spark phụ thuộc vào kích thước của dữ liệu đầu vào và các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle). Thời gian thực thi của Spark TeraSort với tham số **InputSplit(512MB)** và tham số **Shuffle(Buffer 16KB, MaxSizeInfo 48MB)** đạt được phần trăm tối ưu cao nhất trên toàn bộ tập dữ liệu (**4.16%** và **5.67%**). Trên thực tế, tham số **InputSplit(256MB)** đạt được hiệu quả tối ưu hơn trên phần lớn các mức dữ liệu đầu vào (Input) - trên 15GB, cải thiện hiệu năng xử lý cao hơn **2%**

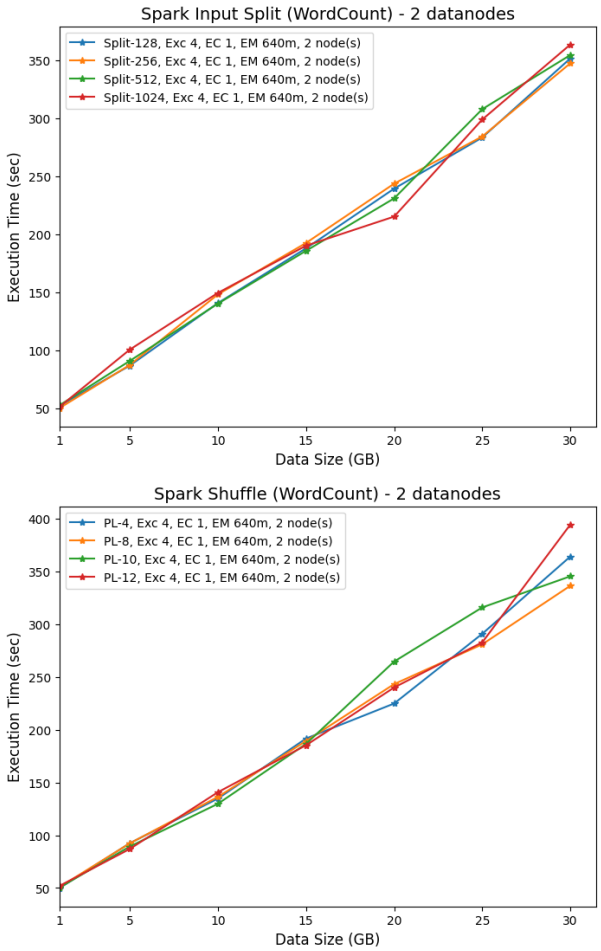
Do đó, chúng ta có thể kết luận rằng việc các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle), có thể được coi là một yếu tố quan trọng trong việc nâng cao hiệu năng của Spark TeraSort khi thực thi các tập dữ liệu có kích thước nhỏ.

3.2.2 Spark - 2 datanode(s)

Bài toán được thiết lập với dữ liệu đầu vào có kích thước tăng dần từ (1G-30GB), đang hoạt động với 2 datanode(s), executor là 4, executor core là 1, và executor memory là 640m. Thực nghiệm tiến hành thay đổi các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle) để chứng minh rằng chúng có ảnh hưởng đến hiệu năng của hệ thống.

Bài toán WordCount

Bài toán được thiết lập với dữ liệu đầu vào có kích thước tăng dần từ (1G-30GB), đang hoạt động với 1 datanode, executor là 4, executor core là 1, và executor memory là 640m. Thực nghiệm tiến hành thay đổi các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle) để chứng minh rằng chúng có ảnh hưởng đến hiệu năng của hệ thống.



Hình 13: Biểu đồ thời gian thực thi của Spark cho bài toán WordCount

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Parallelism**: là số luồng xử lý song song, giá trị mặc định là 4.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

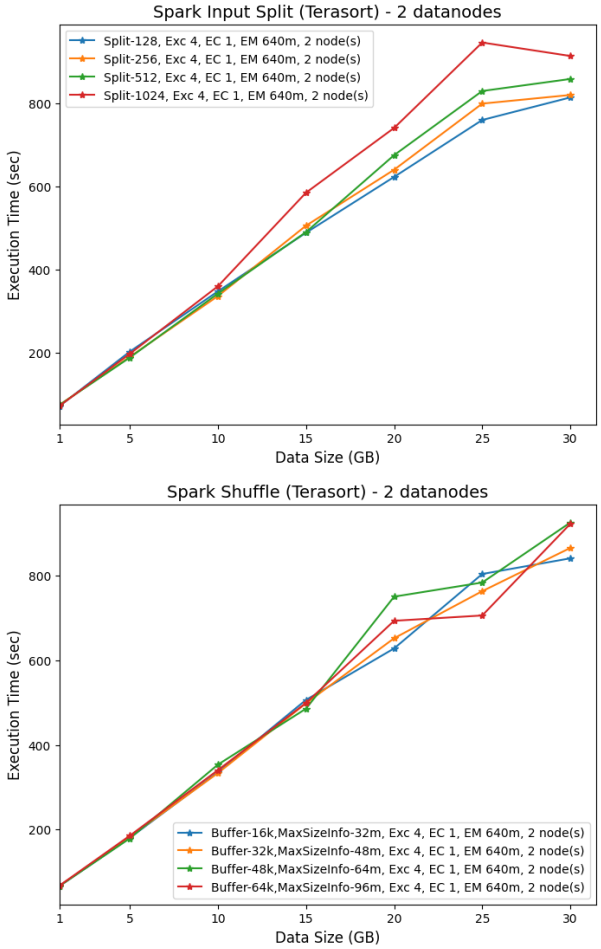
	Default		Input Split		Optimization
Input	Input Split	Duration(s)	Input Split	Duration(s)	
1GB	128MB	52	**256MB**	**50**	**3.85%**
5GB	128MB	87	**128MB**	**87**	
10GB	128MB	141	**512MB**	**140**	**0.71%**
15GB	128MB	188	**512MB**	**186**	**1.06%**
20GB	128MB	239	**1024MB**	**215**	**10.04%**
25GB	128MB	283	**128MB**	**283**	
30GB	128MB	352	**256MB**	**347**	**1.42%**

	Default		Shuffle		Optimization
Input	Parallelism	Duration(s)	Parallelism	Duration(s)	
1GB	4	50	**4**	**50**	
5GB	4	92	**12**	**87**	**5.43%**
10GB	4	135	**10**	**130**	**3.70%**
15GB	4	192	**12**	**185**	**3.65%**
20GB	4	225	**4**	**225**	
25GB	4	291	**8**	**281**	**3.44%**
30GB	4	364	**8**	**336**	**7.69%**

Hình trên cho thấy thời gian thực thi của Spark phụ thuộc vào kích thước của dữ liệu đầu vào và các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle). Thời gian thực thi của Spark WordCount với tham số **InputSplit(1024MB)** và tham số **Shuffle(Parallelism 8)** đạt được phần trăm tối ưu cao nhất trên toàn bộ tập dữ liệu (**10.04%** và **7.69%**).

Do đó, chúng ta có thể kết luận rằng việc các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle), có thể được coi là một yếu tố quan trọng trong việc nâng cao hiệu năng của Spark WordCount khi thực thi các tập dữ liệu có kích thước nhỏ.

Bài toán TeraSort



Hình 14: Biểu đồ thời gian thực thi của Spark cho bài toán TeraSort

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Buffer(KB)**: Bộ đệm để lưu trữ dữ liệu tạm thời, giá trị mặc định là 32KB.
- **MaxSizeInfo**: Kích thước dữ liệu tối đa có thể xử lý, giá định mặc định là 48MB.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

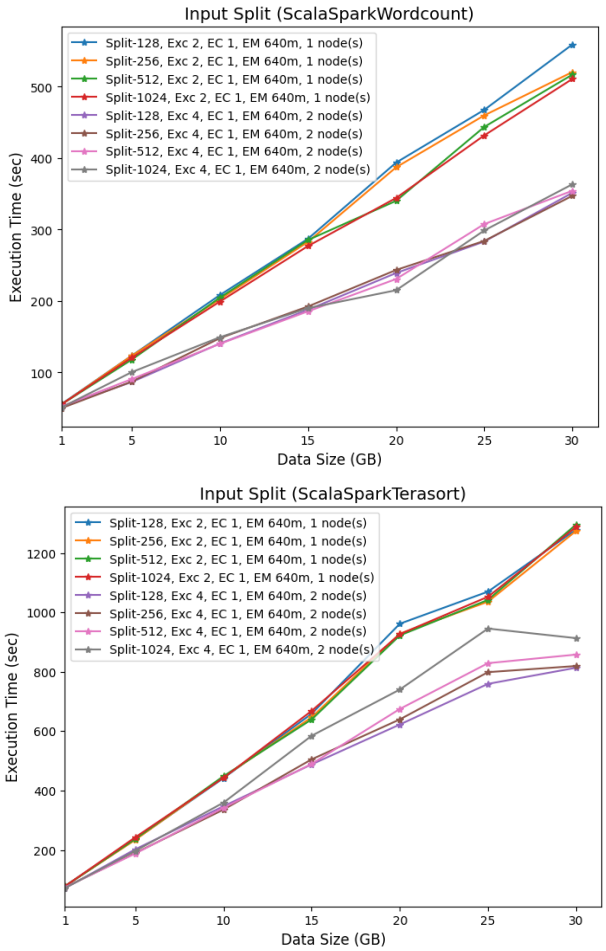
	Default		Input Split		Optimization
Input	Input Split	Duration(s)	Input Split	Duration(s)	
1GB	128MB	71	**128MB**	**71**	
5GB	128MB	202	**512MB**	**188**	**6.93%**
10GB	128MB	347	**256MB**	**336**	**3.17%**
15GB	128MB	488	**128MB**	**488**	
20GB	128MB	622	**128MB**	**622**	
25GB	128MB	759	**128MB**	**759**	
30GB	128MB	813	**128MB**	**813**	

	Default		Shuffle		Optimization
Input	Buffer	Duration(s)	Buffer	Duration(s)	
1GB	32KB	67	**16KB**	**66**	**1.49%**
5GB	32KB	183	**48KB**	**179**	**2.19%**
10GB	32KB	333	**32KB**	**333**	
15GB	32KB	498	**48KB**	**486**	**2.41%**
20GB	32KB	652	**16KB**	**628**	**3.68%**
25GB	32KB	763	**64KB**	**706**	**7.47%**
30GB	32KB	865	**16KB**	**840**	**2.89%**

Hình trên cho thấy thời gian thực thi của Spark phụ thuộc vào kích thước của dữ liệu đầu vào và các tham số phân chia dữ liệu đầu vào (InputSplit), tham số xáo trộn dữ liệu đầu vào (Shuffle). Thời gian thực thi của Spark TeraSort với tham số **InputSplit(512MB)** và tham số **Shuffle(Buffer 64KB, MaxSizeInfo 96MB)** đạt được phần trăm tối ưu cao nhất trên toàn bộ tập dữ liệu (**6.93%** và **7.47%**). Trên thực tế, tham số **InputSplit(128MB)** - tham số mặc định và tham số **Shuffle(Buffer 48KB, MaxSizeInfo 64MB)**, đạt được hiệu quả tối ưu hơn trên phần lớn các mức dữ liệu đầu vào (Input).

Do đó, chúng ta chỉ có thể kết luận rằng các tham số xáo trộn dữ liệu đầu vào (Shuffle) là một yếu tố quan trọng trong việc nâng cao hiệu năng của Spark TeraSort khi thực thi các tập dữ liệu có kích thước nhỏ.

3.2.3 So sánh



Hình 15: Biểu đồ thời gian thực thi của Spark khi thay đổi số lượng datanode

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

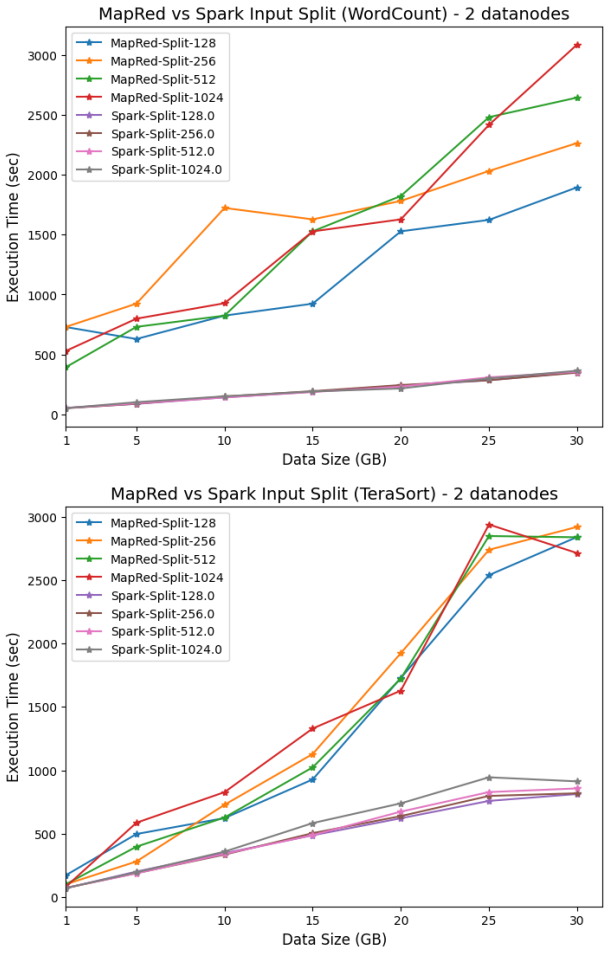
WordCount	1 Datanode		2 Datanodes		**Speed up**
Input	Input Split	Duration(s)	Input Split	Duration(s)	**(1datanode/2datanodes)**
1GB	1024MB	55	1024MB	51	1.08
5GB	1024MB	121	1024MB	100	1.21
10GB	1024MB	199	1024MB	149	1.34
15GB	1024MB	277	1024MB	190	1.46
20GB	1024MB	344	1024MB	215	1.6
25GB	1024MB	432	1024MB	299	1.44
30GB	1024MB	511	1024MB	363	1.41

TeraSort	1 Datanode		2 Datanodes		**Speed up**
Input	Input Split	Duration(s)	Input Split	Duration(s)	**(1datanode/2datanodes)**
1GB	256MB	78	128MB	71	1.1
5GB	256MB	234	128MB	202	1.16
10GB	256MB	445	128MB	347	1.28
15GB	256MB	647	128MB	488	1.33
20GB	256MB	928	128MB	622	1.49
25GB	256MB	1036	128MB	759	1.36
30GB	256MB	1274	128MB	813	1.57

Hình trên so sánh thời gian thực thi của Spark hoạt động với 1 datanode và 2 datanodes khi thực nghiệm trên 2 bài toán WordCount, TeraSort với các tham số phân chia dữ liệu đầu vào khác nhau (InputSplit). Đối với bài toán *WordCount*, ta tiến hành phân tích kết quả của InputSplit có giá trị **1024MB**, vì tham số này thu được hiệu năng cao nhất trong quá trình thực nghiệm của SparkWordCount trên 1 và 2 datanode(s). Đối với bài toán TeraSort, ta tiến hành phân tích kết quả của InputSplit có giá trị 256MB, 128MB vì tham số này thu được hiệu năng cao nhất trong quá trình thực nghiệm của SparkTeraSort trên 1 và 2 datanode(s). Chúng ta nhận thấy rằng khi tăng số lượng datanode, hiệu năng của hệ thống được cải thiện rõ rệt (cao nhất lên tới **37.5%**).

Do đó, chúng ta có thể kết luận rằng việc tăng số lượng datanode có thể cải thiện hiệu năng của hệ thống và tăng tốc độ xử lý.

3.3 So sánh MapReduce - Spark



Hình 16: Biểu đồ thời gian thực thi của MapReduce và Spark

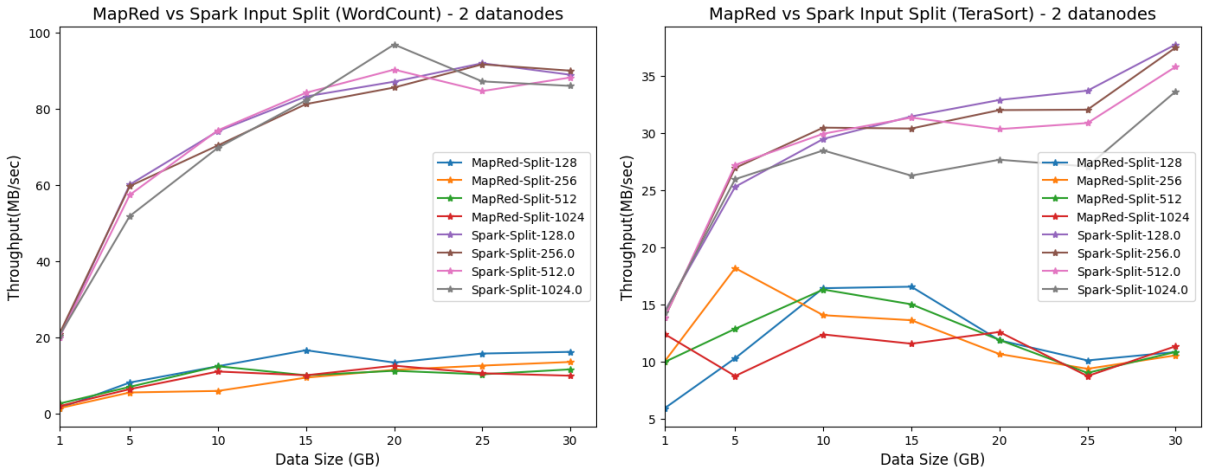
Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

WordCount	**MapReduce**		**Spark**		**Speed up**
Input	Input Split	Duration(s)	Input Split	Duration(s)	**(MapReduce/Spark)**
1GB	128MB	728	128MB	52	14
5GB	128MB	628	128MB	87	7
10GB	128MB	824	128MB	141	6
15GB	128MB	924	128MB	188	5
20GB	128MB	1528	128MB	239	6
25GB	128MB	1624	128MB	283	6
30GB	128MB	1896	128MB	352	5

TeraSort	**MapReduce**		**Spark**		**Speed up**
Input	Input Split	Duration(s)	Input Split	Duration(s)	**(MapReduce/Spark)**
1GB	128MB	174	128MB	71	2
5GB	128MB	498	128MB	202	2
10GB	128MB	624	128MB	347	2
15GB	128MB	928	128MB	488	2
20GB	128MB	1728	128MB	622	3
25GB	128MB	2539	128MB	759	3
30GB	128MB	2839	128MB	813	3

Hình trên so sánh giữa Spark và MapReduce khi thực hiện hai bài toán WordCount, TeraSort với các tham số phân chia dữ liệu đầu vào (Input Split) khác nhau. Đối với bài toán **WordCount**, **Spark có hiệu năng gấp 6 lần so với MapReduce**, trường hợp cao nhất lên tới 14 lần. Đối với bài toán **TeraSort**, **Spark có hiệu năng gấp 3 lần so với MapReduce**. Bên cạnh đó, hiệu năng của Spark tuyến tính và lớn hơn khi khích thước dữ liệu tăng lên; ngược lại, hiệu năng của MapReduce không tuyến tính nhưng có xu hướng hội tụ đến một hằng số.



Hình 17: Biểu đồ throughput của MapReduce và Spark

Trong đó,

- **Input Split (MB)**: là kích thước phân chia dữ liệu đầu vào, đơn vị là MB, có giá trị mặc định là 128MB.
- **Throughput (MBs)**: tham số chỉ lượng dữ liệu được truyền trong một đơn vị thời gian, trong trường hợp này throughput có đơn vị là MB/s.
- **Execution Time (s)**: là thời gian thực thi, đơn vị là giây.

WordCount	**MapReduce**		**Spark**		**Speed up (MapReduce/Spark)**
Input	Input Split	Throughput(MB/s)	Input Split	Throughput(MB/s)	Speed up
1GB	128MB	1.41	128MB	20.01	14
5GB	128MB	8.15	128MB	60.15	7
10GB	128MB	12.43	128MB	74.16	6
15GB	128MB	16.63	128MB	83.29	5
20GB	128MB	13.40	128MB	87.18	7
25GB	128MB	15.76	128MB	92.02	6
30GB	128MB	16.20	128MB	88.99	6

TeraSort	**MapReduce**		**Spark**		**Speed up (MapReduce/Spark)**
Input	Input Split	Throughput(MB/s)	Input Split	Throughput(MB/s)	Speed up
1GB	128MB	5.90	128MB	14.38	2
5GB	128MB	10.28	128MB	25.30	2
10GB	128MB	16.41	128MB	29.49	2
15GB	128MB	16.54	128MB	31.46	2
20GB	128MB	11.85	128MB	32.90	3
25GB	128MB	10.08	128MB	33.72	3
30GB	128MB	10.82	128MB	37.76	3

Hình trên so sánh throughput giữa Spark và MapReduce khi thực hiện hai bài toán WordCount, TeraSort với các tham số phân chia dữ liệu đầu vào (Input Split) khác nhau. Đối với bài toán **WordCount**, **Spark có hiệu năng gấp 6 lần so với MapReduce**, trường hợp cao nhất lên tới 14 lần; Nhưng throughput của MapReduce có xu hướng ổn định hơn khi kích thước dữ liệu tăng dần. Đối với bài toán **TeraSort**,**Spark có hiệu năng gấp 3 lần so với MapReduce**. Và nhìn chung cả hai bài toán, throughput đều có xu hướng hội tụ đến một hằng số.

Kết luận

Báo cáo **"Phân tích hiệu năng của Apache Hadoop và Apache Spark trên dữ liệu lớn"** này trình bày quá trình thử nghiệm đo hiệu năng của Apache Hadoop (MapReduce) và Apache Spark, trên tập dữ liệu có kích thước từ 1GB đến 30GB. Trong quá trình thực nghiệm, chúng tôi đã điều chỉnh các tham số mặc định của hệ thống, bao gồm tham số dữ liệu đầu vào InputSplit và tham số trong quá trình Shuffle, theo phương pháp thử và sai nhằm tìm ra cấu hình tối ưu cho từng bài toán. Hai bài toán được sử dụng để đo lường hiệu năng là **WordCount** và **TeraSort**.

Kết quả thực nghiệm cho thấy hiệu năng của cả Apache Hadoop và Apache Spark phụ thuộc đáng kể vào các tham số hệ thống. Việc điều chỉnh các tham số phù hợp có thể mang lại hiệu năng rất cao, giúp tối ưu hóa thời gian xử lý dữ liệu. Trong đó, các tham số như kích thước **InputSplit** ảnh hưởng trực tiếp đến cách dữ liệu được phân chia và xử lý trong hệ thống. Ngoài ra, tham số liên quan đến **Shuffle** quyết định cách các khối dữ liệu được tổ chức và truyền giữa các tác vụ, ảnh hưởng lớn đến hiệu suất tổng thể của hệ thống.

Tuy nhiên, do hạn chế về tài nguyên máy tính, các thí nghiệm được thực hiện trên hệ thống máy tính cá nhân, sử dụng các máy ảo liên kết với nhau để giả lập môi trường phân tán. Điều này có thể gây ra một số sai số trong quá trình đo lường hiệu năng. Bộ dữ liệu thử nghiệm được giới hạn từ 1GB đến 30GB, do đó, kết luận rút ra từ báo cáo này chỉ phản ánh một phần nhỏ khả năng xử lý của MapReduce và Spark trong thực tế.

Một điểm nổi bật từ các thí nghiệm là việc **tăng số lượng node thực thi giúp tăng tốc độ xử lý đáng kể**. Cụ thể, khi tăng số lượng từ 1 datanode lên 2 datanodes, tốc độ xử lý của hệ thống tăng gần gấp đôi, điều này chứng minh tầm quan trọng của việc tận dụng tài nguyên phân tán để nâng cao hiệu quả xử lý dữ liệu.

Ngoài ra, kết quả so sánh giữa Apache Spark và MapReduce cho thấy Spark có hiệu suất vượt trội hơn. *Trong bài toán WordCount, Spark xử lý nhanh hơn MapReduce khoảng 6 lần, và đối với bài toán TeraSort, Spark cũng vượt trội với tốc độ nhanh hơn khoảng 2 lần.* Đặc biệt, kết quả đo **throughput** (lượng dữ liệu được xử lý trong một đơn vị thời gian) cho thấy **Spark hoạt động ổn định** và duy trì khả năng xử lý nhanh hơn so với MapReduce. Trong khi throughput của Spark duy trì ở mức cao, throughput của **MapReduce có xu hướng hội tụ về một hằng số khi kích thước dữ liệu tăng**.

Nhìn chung, cả **Hadoop và Spark đều hoạt động ổn định trên tập dữ liệu thử nghiệm [1GB-30GB]**. Tuy nhiên, ưu điểm của Spark nằm ở khả năng xử lý dữ liệu nhanh hơn và hiệu quả hơn, đặc biệt khi kích thước dữ liệu lớn hơn. Điều này làm cho Spark trở thành lựa chọn phù hợp hơn trong các ứng dụng yêu cầu tốc độ xử lý cao và hiệu suất ổn định.

Trong tương lai, chúng tôi dự kiến mở rộng nghiên cứu bằng cách triển khai thực nghiệm trên hệ thống máy tính liên kết với nhau qua mạng cục bộ (LAN) để mô phỏng môi trường thực tế tốt hơn. Ngoài ra, các thí nghiệm sẽ được thực hiện trên nhiều bài toán khác trong bộ công cụ HiBench, bao gồm các bài toán phân tích dữ liệu phức tạp hơn và có tính ứng dụng thực tiễn cao.

Một trong những trọng tâm chính sẽ là nghiên cứu các phương pháp tự động điều chỉnh tham số cho cả MapReduce và Spark nhằm thay thế các giá trị mặc định. Việc này giúp hệ thống tối ưu hóa hiệu năng một cách tự động, giảm thiểu công sức điều chỉnh thủ công. Đồng thời, chúng tôi sẽ nghiên cứu thêm các khía cạnh khác như khả năng song song hóa, quản lý tài nguyên và hiệu năng trên các tập dữ liệu thực tế với quy mô lớn hơn.

Báo cáo này là bước đầu trong việc đánh giá và so sánh hiệu năng của Hadoop và Spark. Kỳ vọng rằng, trong các nghiên cứu tiếp theo, chúng tôi có thể mang đến những đóng góp giá trị hơn cho cộng đồng các nhà phát triển trong lĩnh vực xử lý dữ liệu lớn.

Tham khảo

Tài liệu

[1] N. Ahmed, "A comprehensive performance analysis of apache hadoop and apache spark for large scale data sets using hibench," *Journal of Big Data*, 2020.

[2] "Apache hadoop documentation 3.2.0," https://hadoop.apache.org/, Apache Software Foundation, 2019.

[3] "Apache spark documentation 3.2.0," https://spark.apache.org/, Apache Software Foundation, 2019.

[4] "Hibench: A benchmark suite for big data," https://github.com/Intel-bigdata/HiBench, Intel BigData, 2021.

[5] "Prometheus, jmx, grafana monitor hdp-hadoop," https://www.programmersought.com/article/81846796665/, ProgrammerSought, 2019.