



**LỜI CẢM ƠN**

Trước tiên em xin gửi lời cảm ơn chân thành và sâu sắc đến thầy giáo, TS. Nguyễn Văn Trung, người đã định hướng về mục tiêu của đề tài khóa luận đồng thời cũng là người hướng dẫn, động viên và chỉ bảo cho em với điều kiện tốt nhất từ khi bắt đầu cho đến khi kết thúc đề tài khóa luận tốt nghiệp.

Em xin dành lời cảm ơn chân thành tới các thầy cô giáo khoa Công nghệ thông tin, trường Đại học Khoa Học Huế, đã tận tình đào tạo, chỉ bảo cho em từ những ngày đầu gia nhập trường, cung cấp cho em những kiến thức vô cùng quý giá và tạo kiện tốt nhất cho em trong suốt quá trình học tập và làm việc tại trường.

Cuối cùng, em xin cảm ơn tất cả những người thân yêu trong gia đình và toàn thể bạn bè những người đã luôn kề bên giúp đỡ, động viên khi em gặp khó khăn.

Mặc dù đã hoàn thành đề tài, do giới hạn về thời gian và hiểu biết cá nhân, khó có thể tránh khỏi những thiếu sót. Vì vậy, em rất mong nhận được sự thông cảm và những ý kiến đánh giá khách quan từ thầy cô và các bạn để bài khóa luận này có thể đạt được kết quả tốt hơn.

Em xin chân thành cảm ơn!

LỜI CAM ĐOAN

Em cam đoan đây là công trình do em tự thực hiện dưới sự hướng dẫn của TS. Nguyễn Văn Trung. Các nội dung nghiên cứu, số liệu và kết quả là trung thực. Các số liệu, công trình sử dụng của tác giả khác đều được trích dẫn nguồn gốc rõ ràng.

Nếu phát hiện có bất kỳ sự gian lận nào, em xin chịu hoàn toàn trách nhiệm.

Huế, tháng 5 năm 2024

Sinh viên

Lê Tự Toàn

**BẢNG CHỮ CÁI VIẾT TẮT**

| **Viết tắt** | **Tiếng Anh** | **Tiếng Việt** |
| --- | --- | --- |
| HDFS | Hadoop File System | Hệ thống lưu dữ liệu được sử dụng bởi Hadoop |
| ACID | Atomicity, Consistency, Isolation, Durability | Bốn thuộc tính quan trọng của một hệ quản trị cơ sở dữ liệu: tính nguyên tử, tính nhất quán, tính cô lập, tính bền vững |
| ETL | Extract Transform and Load | Trích xuất, chuyển đổi và tải |
| ELT | Extract Load and Transform | Trích xuất, tải và chuyển đổi |
| IoT | Internet of Things | Internet vạn vật |
| SQL | Structured Query Language | Ngôn ngữ truy vấn có cấu trúc |
| NoSQL | Not Only SQL | Không chỉ SQL |
| ML | Machine Learning | Học máy |
| AI | Artificial Intelligence | Trí tuệ nhân tạo |
| BI | Business Intelligence | Trí tuệ kinh doanh |
| CSV | Comma-Separated Values | Giá trị phân tách bằng dấu phẩy |
| JSON | JavaScript Object Notation | Ký hiệu đối tượng JavaScript |
| XML | eXtensible Markup Language | Ngôn ngữ đánh dấu mở rộng |
| RDBMS | Relational Database Management System | Hệ quản trị cơ sở dữ liệu quan hệ |
| API | Application Programming Interface | Giao diện lập trình ứng dụng |
| CLI | Command Line Interface | Giao diện dòng lệnh |
| K8S | Kubernetes | K – tám chữ cái ở giữa – S,  điều phối viên Kubernetes |
| ITS | Intelligence Transportation System | Hệ thống giao thông thông minh |

**DANH MỤC HÌNH ẢNH**

[Hình 1.1: Bức tranh toàn cảnh về Big Data hiện nay 4](#_heading=h.tyjcwt)

[Hình 1.2: Phạm vi nghiên cứu 7](#_heading=h.3dy6vkm)

[Hình 2.1: Hadoop 1.0 và Hadoop 2.0 10](#_heading=h.17dp8vu)

[Hình 2.2: Quá trình đọc tập tin trên HDFS 16](#_heading=h.44sinio)

[Hình 2.3: Quá trình ghi tập dữ liệu trên HDFS 16](#_heading=h.2jxsxqh)

[Hình 2.4: Các thành phần trong nhiệm vụ xử lý Map-Reduce 18](#_heading=h.4i7ojhp)

[Hình 2.5: Quy trình Hadoop Map-Reduce 19](#_heading=h.1ci93xb)

[Hình 2.6: Quy trình Wordcount trong Map-Reduce 19](#_heading=h.3whwml4)

[Hình 2.7: Thành phần của Apache Spark 27](#_heading=h.147n2zr)

[Hình 2.8: Luồng hoạt động của Spark Streaming 28](#_heading=h.32hioqz)

[Hình 2.9: Cơ chế hoạt động của Spark 30](#_heading=h.vx1227)

[Hình 2.10: Phân bổ tài nguyên trong Spark 31](#_heading=h.4f1mdlm)

[Hình 2.11: Một quy trình thực thi nhiệm vụ của Spark 33](#_heading=h.28h4qwu)

[Hình 2.12: Spark trong chế độ Standalone Cluster Manager 36](#_heading=h.nmf14n)

[Hình 2.13: Kiến trúc của Apache Mesos 36](#_heading=h.37m2jsg)

[Hình 2.14: Spark trong chế độ hoạt động với YARN 37](#_heading=h.1mrcu09)

[Hình 2.15: Quy trình theo dõi đầu ra bản đồ 39](#_heading=h.46r0co2)

[Hình 3.1: Kiến trúc triển khai ứng dụng cổ điển 47](#_heading=h.3ygebqi)

[Hình 3.2: Kiến trúc máy tính ảo hóa 49](#_heading=h.sqyw64)

[Hình 3.3: Kiến trúc Container hóa 53](#_heading=h.3q5sasy)

[Hình 3.4 : Lịch sử phát triển của môi trường triển khai ứng dụng 54](#_heading=h.kgcv8k)

[Hình 3.5: Mức độ phổ biến của Docker container và VM trên toàn thế giới trong vòng 5 năm qua (2019-2024) 56](#_heading=h.34g0dwd)

[Hình 3.6: Kiến trúc Docker 57](#_heading=h.43ky6rz)

[Hình 3.7: Docker Engine 58](#_heading=h.xvir7l)

[Hình 3.8: Phiên bản Docker Compose 65](#_heading=h.2afmg28)

[Hình 3.9: Kiến trúc Docker Swarm 67](#_heading=h.39kk8xu)

[Hình 4.1: Giao diện VirtualBox Manager 73](#_heading=h.2250f4o)

[Hình 4.2: Giao diện tạo Virtual Machine 73](#_heading=h.haapch)

[Hình 4.3: Môi trường thí nghiệm 74](#_heading=h.1gf8i83)

[Hình 4.4: Cấu hình VM máy thứ nhất 75](#_heading=h.40ew0vw)

[Hình 4.5: Cấu hình VM máy thứ hai. 75](#_heading=h.2fk6b3p)

[Hình 4.6: Mạng kết nối. 77](#_heading=h.3ep43zb)

[Hình 4.7: Cụm Swarm trong thí nghiệm. 78](#_heading=h.1tuee74)

[Hình 4.8: Tỷ lệ hủy chuyến bay và chuyển chuyến bay của từng hãng hàng không trong các ngày trong tuần trên tập dữ liệu Airline Delay Analysis (2009-2013). 80](#_heading=h.184mhaj)

[Hình 4.9: Khoảng cách bay của từng hãng hàng không các ngày trong tuần trên tập dữ liệu Airline Delay Analysis (2009-2013). 80](#_heading=h.3s49zyc)

[Hình 4.10: Top 10 hãng hàng không muộn chuyến (2009-2013) 81](#_heading=h.279ka65)

[Hình 4.11: Bảng phân bố thời gian muộn chuyến của hai 20 hãng hàng không trong từng ngày trong tuần trên tập dữ liệu Airline Delay Analysis (2009-2013). 81](#_heading=h.meukdy)

# **MỤC LỤC**

[MỞ ĐẦU 1](#_heading=h.30j0zll)

[CHƯƠNG 1: TỔNG QUAN VỀ BIG DATA 2](#_heading=h.1fob9te)

[1.1 Giới thiệu Big Data 2](#_heading=h.3znysh7)

[1.2 Khái quát về Big Data 3](#_heading=h.2et92p0)

[CHƯƠNG 2: CƠ SỞ LÝ THUYẾT 8](#_heading=h.1t3h5sf)

[2.1 Khái quát về hệ phân tán 8](#_heading=h.4d34og8)

[2.2 Giới thiệu về Apache Hadoop 9](#_heading=h.2s8eyo1)

[2.2.1 Các ứng dụng nền tảng của Apache Hadoop 11](#_heading=h.3rdcrjn)

[2.2.2 Hệ thống quản lý tập tin phân tán Hadoop 13](#_heading=h.26in1rg)

[2.2.2.1 Giới thiệu HDFS 13](#_heading=h.lnxbz9)

[2.2.2.2 Khái niệm nền tảng kiến trúc HDFS 14](#_heading=h.35nkun2)

[2.2.2.3 Đọc ghi dữ liệu trên HDFS 15](#_heading=h.1ksv4uv)

[2.2.3 Giải thuật xử lý song song MAP-REDUCE 17](#_heading=h.z337ya)

[2.2.3.1 Giới thiệu MAP-REDUCE 17](#_heading=h.3j2qqm3)

[2.2.3.2 Kiến trúc của MAP-REDUCE 18](#_heading=h.1y810tw)

[2.2.3.3 Bài toán đếm chữ cái 18](#_heading=h.2xcytpi)

[2.3 Giới thiệu Apache Spark 21](#_heading=h.2bn6wsx)

[2.3.1 Lợi thế của Apache Spark 22](#_heading=h.qsh70q)

[2.3.2 Các khái niệm của Apache Spark 23](#_heading=h.3as4poj)

[2.3.2.1 Resilient Distributed Datasets 23](#_heading=h.1pxezwc)

[2.3.2.2 Khái niệm cốt lõi của Spark 24](#_heading=h.49x2ik5)

[2.3.3 Các thành phần của Apache Spark 27](#_heading=h.2p2csry)

[2.3.3.1 Spark Core 27](#_heading=h.3o7alnk)

[2.3.3.2 Spark SQL 27](#_heading=h.23ckvvd)

[2.3.3.3 Spark Streaming 28](#_heading=h.ihv636)

[2.3.3.4 Spark MLlib 29](#_heading=h.1hmsyys)

[2.3.3.5 Spark GraphX 29](#_heading=h.41mghml)

[2.3.4 Kiến trúc Apache Spark 30](#_heading=h.2grqrue)

[2.3.4.1 Master 30](#_heading=h.3fwokq0)

[2.3.4.2 Worker 30](#_heading=h.1v1yuxt)

[2.3.4.3 Driver 31](#_heading=h.2u6wntf)

[2.3.4.4 Executor 32](#_heading=h.19c6y18)

[2.3.4.5 Thành phần của Spark 33](#_heading=h.3tbugp1)

[2.3.4.6 Khả năng chịu lỗi của Spark 39](#_heading=h.2lwamvv)

[2.4 Cơ sở lý luận 40](#_heading=h.111kx3o)

[2.5. Lý thuyết về các kiến trúc và thuật ngữ 42](#_heading=h.3l18frh)

[2.5.1. Data Warehouse 42](#_heading=h.206ipza)

[2.5.2. Data Lake 44](#_heading=h.4k668n3)

[CHƯƠNG 3: TRIỂN KHAI ỨNG DỤNG 47](#_heading=h.2zbgiuw)

[3.1 Lịch sử triển khai ứng dụng. 47](#_heading=h.1egqt2p)

[3.2 Công nghệ ảo hóa 48](#_heading=h.2dlolyb)

[3.2.1 Lợi ích của công nghệ ảo hóa 50](#_heading=h.3cqmetx)

[3.2.2 Giới thiệu ảo hóa 50](#_heading=h.1rvwp1q)

[3.2.2.1 Các loại ảo hóa 50](#_heading=h.4bvk7pj)

[3.3 Docker 53](#_heading=h.2r0uhxc)

[3.3.1 Giới thiệu Docker 53](#_heading=h.1664s55)

[3.3.2 Docker và VM 54](#_heading=h.25b2l0r)

[3.3.3 Kiến trúc Docker 57](#_heading=h.1jlao46)

[3.3.3.1 Docker Engine 58](#_heading=h.2iq8gzs)

[3.3.3.2 Docker Containers 59](#_heading=h.3hv69ve)

[3.3.3.3 Docker Images 60](#_heading=h.1x0gk37)

[3.3.3.4 Docker Network 61](#_heading=h.4h042r0)

[3.3.3.5 Docker Registry 63](#_heading=h.2w5ecyt)

[3.3.4 Docker Container Orchestration 63](#_heading=h.1baon6m)

[3.3.4.1 Docker Compose 64](#_heading=h.3vac5uf)

[3.3.4.2 Docker Swarm 67](#_heading=h.pkwqa1)

[CHƯƠNG 4: THỰC NGHIỆM VÀ ĐÁNH GIÁ KẾT QUẢ 69](#_heading=h.1opuj5n)

[4.1 Kịch bản triển khai 69](#_heading=h.48pi1tg)

[4.2 Thiết lập môi trường 72](#_heading=h.2nusc19)

[4.2.1 Tải các phần mềm cần thiết cho môi trường nghiên cứu 72](#_heading=h.1302m92)

[4.2.2 Tạo các máy ảo trên hệ điều hành Linux 73](#_heading=h.3mzq4wv)

[4.2.3 Cài đặt Docker lên các máy ảo 74](#_heading=h.319y80a)

[4.2.4 Tạo Cluster Docker Swarm và thêm các Node vào Cluster 77](#_heading=h.upglbi)

[4.2.5 Chạy ứng dụng 78](#_heading=h.4du1wux)

[4.3 Kết quả thực nghiệm xử lý dữ liệu 79](#_heading=h.2szc72q)

[CHƯƠNG 5: KẾT LUẬN 83](#_heading=h.36ei31r)

[5.1. Kết quả nghiên cứu của đề tài 83](#_heading=h.1ljsd9k)

[5.2. Hạn chế luận văn 83](#_heading=h.45jfvxd)

[5.3. Hướng phát triển tiếp theo của đề tài nghiên cứu 83](#_heading=h.2koq656)

[TÀI LIỆU THAM KHẢO 84](#_heading=h.a0oirrlnrzzv)

# 

# 

# 

# 

# 

# 

# 

# **MỞ ĐẦU**

1. **Lý do chọn đề tài**

Trong thời đại kỹ thuật số ngày nay, còn được biết đến dưới các thuật ngữ như "thời đại máy tính", "thời đại thông tin", hoặc "thời đại truyền thông mới", việc để một doanh nghiệp bắt kịp với xu hướng thời đại không chỉ đòi hỏi về vốn và nhân lực mà còn về việc quản lý dữ liệu - một nguồn lực chủ chốt không thể bỏ qua hay có một câu nói kinh điển trong ngành dữ liệu là “In God we trust All others must bring data” của giáo sư kinh tế học William Edwards Deming khi nói về tầm quan trọng của quyết định dựa trên dữ liệu.

1. **Mục tiêu đề tài**

Tìm hiểu về thuật ngữ Big Data

Tìm hiểu công nghệ, các khái niệm xoay quanh thuật ngữ Big Data

Xây dựng một ứng dụng liên quan đến Big Data

1. **Bố cục của khóa luận**

Bố cục của khóa luận được tổ chức thành các chương sau

* Chương 1. Tổng quan về Big Data. Chương này sẽ trình bày một cái nhìn chung về thuật ngữ Big Data.
* Chương 2. Cơ sở lý thuyết. Chương này sẽ trình bày trọng tâm cơ sở lý thuyết có liên quan đến việc xây dựng ứng dụng về cách lưu trữ và xử lý dữ liệu lớn.
* Chương 3. Triển khai ứng dụng. Chương này sẽ trình bày lịch sử quá trình cũng như cơ sở lý thuyết liên quan đến việc xây dựng ứng dụng của luận văn.
* Chương 4. Thực nghiệm và đánh giá kết quả. Chương này sẽ trình bày về nghiên cứu bao gồm cách triển khai và bài toán xử lý cụ thể.
* Chương 5. Kết luận. Bao gồm kết quả, hạn chế và hướng phát triển xa hơn của đề tài.

# **CHƯƠNG 1: TỔNG QUAN VỀ BIG DATA**

## **1.1 Giới thiệu Big Data**

- Theo wikipedia: Dữ liệu lớn (Big data) là một thuật ngữ chỉ bộ dữ liệu lớn hoặc phức tạp mà các phương pháp truyền thống không đủ các ứng dụng để xử lý dữ liệu này. [10]

- Theo Gartner: Dữ liệu lớn là những nguồn thông tin có đặc điểm chung khối lượng lớn, tốc độ nhanh và dữ liệu định dạng dưới nhiều hình thức khác nhau, do đó muốn khai thác được đòi hỏi phải có hình thức xử lý mới để đưa ra quyết định, khám phá và tối ưu hóa quy trình. [11]

Ngày nay dễ nhận thấy nhất với sự xuất hiện của Internet và các nền tảng thương mại điện tử đã tạo ra một hệ thống thông tin khổng lồ về thói quen mua sắm và nhu cầu của người tiêu dùng.

Các trang web này không chỉ gợi ý sản phẩm mà còn thu thập dữ liệu về người dùng thông qua các hoạt động trực tuyến của họ. Những dữ liệu này không chỉ được sử dụng bởi những doanh nghiệp cá nhân đơn thuần mà còn được mua lại hoặc chia sẻ cho các doanh nghiệp khác để phục vụ cho các nhu cầu marketing lớn hơn. Điều này giúp các doanh nghiệp, tổ chức xác định và phân loại khách hàng, hiểu rõ hơn về thị trường và tạo ra các chiến lược marketing hiệu quả. Dữ liệu không chỉ là một cầu nối giữa doanh nghiệp và khách hàng mà còn là yếu tố quan trọng trong việc tối ưu hóa quá trình marketing, giúp tiết kiệm thời gian cho khách hàng và tăng lợi nhuận cho doanh nghiệp bằng cách tối ưu hóa kế hoạch phân phối sản phẩm và dịch vụ trong những bài toán lớn về phân tích dữ liệu. Hơn thế nữa, ở khái niệm vĩ mô, ứng dụng dữ liệu lớn (Big Data) có thể giúp các tổ chức và chính phủ dự đoán được tỉ lệ thất nghiệp, xu hướng nghề nghiệp, hoặc cắt giảm chi tiêu, kích thích tăng trưởng kinh tế… thậm chí là đề ra những chính sách định hướng dựa trên nguồn dữ liệu lớn.

Việc xây dựng các ứng dụng nền tảng Big Data nếu được khai thác hiệu quả sẽ đem lại những lợi thế cạnh tranh và hiệu quả to lớn trong nhiều lĩnh vực, đặc biệt trong bối cảnh thị trường dịch vụ tài chính đang bão hòa như hiện nay, trên cơ sở đó phân tích khái niệm và sử dụng những ứng dụng của Big Data vào mục đích phát triển doanh nghiệp sẽ đưa ra các quyết định tối ưu mang lại hiệu quả hơn việc đưa ra một quyết định mù mờ dựa trên cảm tính.

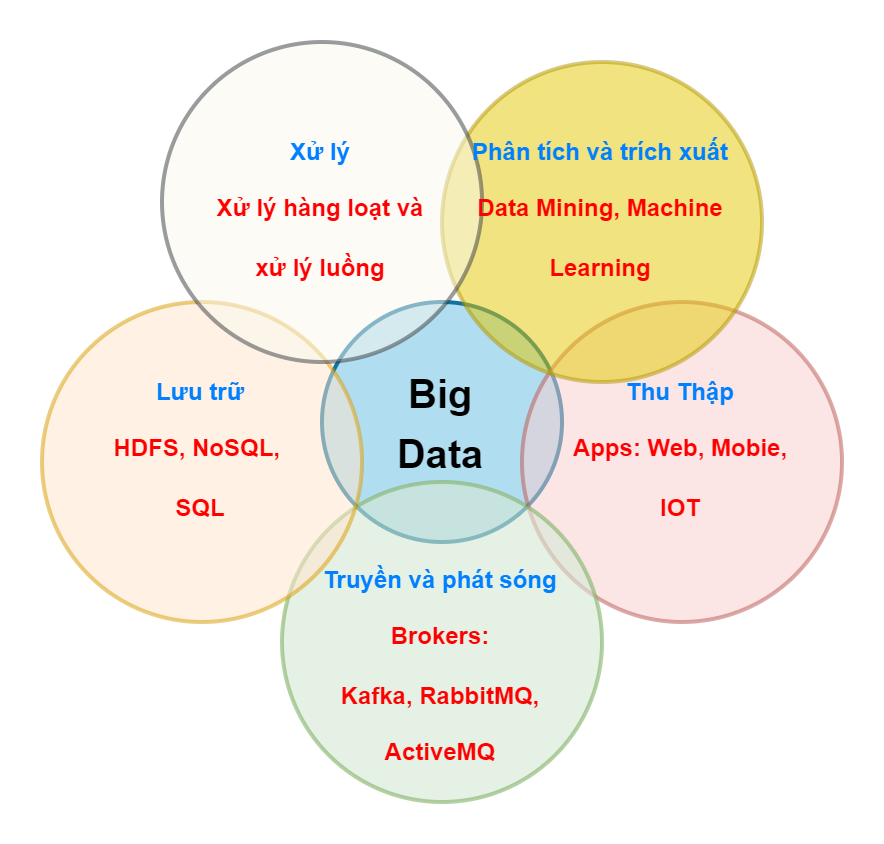
Big Data là thuật ngữ mô tả quá trình xử lý dữ liệu trên một tập dữ liệu lớn bao gồm cả dữ liệu có cấu trúc, bán cấu trúc hay không có cấu trúc - tràn ngập trong các doanh nghiệp hàng ngày, nhấn mạnh việc tập dữ liệu có quá nhiều kiểu. Những bộ dữ liệu này quá lớn và tăng theo từng ngày đến nỗi phần mềm xử lý dữ liệu truyền thống không thể quản lý và xử lý chúng. Bao gồm các cách thức phân tích, thu thập, giám sát dữ liệu, tìm kiếm, chia sẻ, lưu trữ, truyền nhận, trực quan hóa, truy vấn và tính riêng tư. Thuật ngữ này thường đơn giản là sử dụng để phân tích dự đoán hoặc là một số phương pháp tiên tiến khác để trích xuất giá trị từ dữ liệu, mà ít khi đề cập đến kích thước của bộ dữ liệu. Dữ liệu lớn, như tên gọi của nó, có kích thước quá lớn, thay đổi quá nhanh và thường không thể lưu trữ tập trung và không phù hợp với các cấu trúc của kiến trúc cơ sở dữ liệu.

Có một số thống kê thú vị được đưa ra bởi IDC như sau:

* Tạo dữ liệu hàng ngày: IDC báo cáo rằng chúng ta hiện đang tạo ra hơn 2.5 exabyte dữ liệu mỗi ngày, tăng đáng kể so với các năm trước do sự bùng nổ của các thiết bị kết nối và Internet vạn vật (IoT)​ (IDC Blogs)​.
* Dung lượng dữ liệu toàn cầu: Theo IDC, dung lượng dữ liệu toàn cầu đã đạt 64.2 zettabyte vào năm 2020 và dự kiến sẽ tăng lên 180 zettabyte vào năm 2025. Đây là sự tăng trưởng vượt bậc trong không gian dữ liệu kỹ thuật số​ (Datanami)​.
* Tạo dữ liệu cá nhân: Một người trung bình tạo ra khoảng 1.7 megabyte dữ liệu mỗi giây, điều này được thúc đẩy bởi việc sử dụng các thiết bị di động và các dịch vụ trực tuyến ngày càng tăng​ (Datanami)​.
* Tốc độ tăng trưởng dữ liệu: IDC cũng ước tính rằng 90% dữ liệu toàn cầu đã được tạo ra trong vòng hai năm qua, minh chứng cho sự bùng nổ nhanh chóng của dữ liệu trong kỷ nguyên kỹ thuật số​ (Datanami)​.

## **1.2 Khái quát về Big Data**

Sự bùng nổ về lượng dữ liệu được tạo từ mọi nơi ví dụ như từ những ứng dụng mạng xã hội hàng ngày, ứng dụng web di động, nhật ký, những thiết bị kết nối chip cảm biến, ảnh kỹ thuật số, video, hoạt động giao dịch, vệ tinh, v.v. - dưới mọi hình thức khác nhau. Thực tế là các kho dữ liệu hiện tại về dữ liệu lớn được xây dựng trên khắp thế giới đã rất lớn và chúng còn đang tăng trưởng một cách nhanh chóng theo cấp số mũ. Trong đó 3 khía cạnh nhận diện chính của Big Data là: Dữ liệu (Data), Công nghệ (Technology), Quy mô (Size). Thứ nhất là dữ liệu như đã đề cập ở trên xuất phát điểm từ nhiều nguồn trên Internet được cập nhật qua các thiết bị kết nối mạng từng giờ, từng phút, từng giây và khối lượng cập nhật này rất lớn. Hiện nay, Big Data được đo lường theo đơn vị Terabytes (TB), Petabytes (PB) và Exabytes (EB). Yếu tố nhận diện thứ hai là công nghệ (technology). Công nghệ thường được thiết kế và hình thành một hệ sinh thái từ dưới đi lên để có khả năng xử lý các dữ liệu lớn và phức tạp một số công nghệ sẽ được khái quát như sau:



**Hình 1.1: Bức tranh toàn cảnh về Big Data hiện nay**

Và yếu tố nhận diện thứ ba của Big Data là quy mô dữ liệu. Hiện nay vẫn chưa có câu trả lời chính xác cho câu hỏi như thế nào gọi là lớn. Theo ngầm hiểu thì khi dữ liệu vượt quá khả năng xử lý của các hệ thống truyền thống thì sẽ được xếp vào Big Data. Và có rất ứng dụng trong thực tế như [8]:

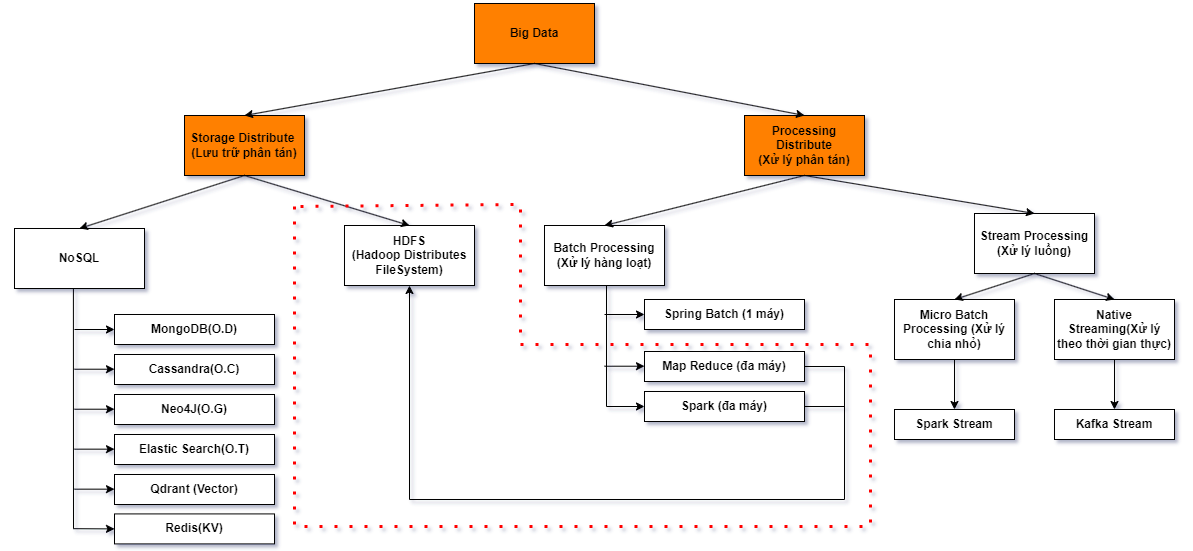
* Ứng dụng trong bầu cử: Tổng thống Mỹ Obama đã sử dụng dữ liệu dữ liệu lớn để phục vụ cho cuộc tranh cử Tổng thống của mình. Ông xây dựng một đội ngũ nhân viên chuyên đi thu thập thông tin và phân tích dữ liệu thu được trong dự án triển khai về dữ liệu lớn. Đội ngũ nhân viên này thu thập tất cả thông tin về người dân ở các khu vực, sau đó phân tích và chỉ ra một số thông tin quan trọng về người dân Mỹ như: Thích đọc sách gì, thích mua loại thuốc gì, thích sử dụng phương tiện gì… Thậm chí còn biết được cả thông tin về mẹ của cử tri đó đã bỏ phiếu tín nhiệm ai ở lần bầu cử trước. Trên cơ sở những thông tin này, Tổng thống Obama đã đưa ra kế hoạch vận động phù hợp, giúp ông tái đắc cử Tổng thống nước Mỹ lần thứ 2. Ngoài ra một số ứng dụng khác trong lĩnh vực chính trị mà dữ liệu lớn được áp dụng như: Hệ thống chính phủ điện tử; phân tích quy định và việc tuân thủ quy định; phân tích, giám sát, theo dõi và phát hiện gian lận, mối đe dọa, an ninh mạng.
* Ứng dụng trong giao thông: Sử dụng số liệu CDR trong quá khứ để ước lượng các dòng giao thông trong thành phố vào các giờ cao điểm, từ đó có những kế hoạch phân luồng giao thông chi tiết, hợp lý giúp giảm thiểu kẹt xe. Ngoài ra còn đưa ra thông tin cho người tham gia giao thông được biết nếu muốn đi từ nơi này đến nơi khác thì nên đi vào giờ nào để tránh kẹt xe, hoặc đi đường nào là ngắn nhất, v.v... Ngoài ra, dữ liệu lớn còn giúp phân tích định vị người dùng thiết bị di động, ghi nhận chi tiết cuộc gọi trong thời gian thực; và giảm thiểu tình trạng ùn tắc giao thông.
* Ứng dụng trong y tế: Trong y học các bác sĩ dựa vào số liệu trong các bệnh án để đưa ra dự đoán về nguy cơ mắc bệnh. Đồng thời cũng đưa ra được xu hướng lây lan của bệnh. Ví dụ, ứng dụng Google Flu Trend là một trong những ứng dụng thành công của Google ứng dụng này dựa trên từ khóa tìm kiếm ở một khu vực nào đó, sau đó bộ máy phân tích của Google sẽ phân tích và đối chiếu kết quả tìm kiếm đó, sau cùng là đưa ra dự báo về xu hướng dịch cúm tại khu vực đó. Qua đó cho biết tình hình cúm tại khu vực đó sẽ diễn ra như thế nào để đưa ra các giải pháp phòng tránh. Những kết quả mà Google Flu Trend đưa ra, hoàn toàn phù hợp với báo cáo của Tổ chức Y tế Thế giới WHO về tình hình bệnh cúm tại các khu vực đó.
* Ứng dụng thống kê: Nhận thấy những lợi ích to lớn và thách thức của Big Data đối với thống kê nhà nước, Ủy ban Thống kê Liên hợp quốc cũng như các tổ chức thống kê khu vực và Cơ quan thống kê quốc gia của nhiều nước đã triển khai hàng loạt các hoạt động về Bigdata như: Hàn Quốc sử dụng ảnh vệ tinh để thống kê nông nghiệp và một số lĩnh vực khác; Australia sử dụng ảnh vệ tinh để thống kê diện tích đất nông nghiệp và năng suất; Italia sử dụng dữ liệu điện thoại di động để thống kê di cư; Bhutan dùng thiết bị di động để tính toán chỉ số giá tiêu dùng; Estonia dùng điện thoại di động định vị vệ tinh để thống kê du lịch; EuroStat sử dụng dữ liệu về sử dụng điện thoại di động để thống kê du lịch. Và còn rất nhiều ứng dụng khác như: trong thể thao, tài chính, thương mại, chính sách, v.v.

Phạm vi nghiên cứu:

Như đã đề cập ở [Hình 1.1] có 5 yếu tố về công nghệ cơ bản để cấu thành thuật ngữ Big Data ở thời điểm hiện tại có thể bao gồm như sau:

* Lưu trữ: HDFS, NoSQL, SQL.
* Xử lý: Xử lý hàng loạt (Batch Processing) và xử lý luồng (Stream Processing).
* Truyền và phát sóng: Brokers .
* Thu thập: Các ứng dụng Apps.
* Phân tích và trích xuất: Data Mining và Machine Learning.

Luận văn này cung cấp một cái nhìn tổng quan về một phần nhỏ trong quá trình xử lý và lưu trữ dữ liệu lớn. Mục đích để hiểu cách xây dựng một khung dữ liệu lớn nhằm phục vụ cho những bài toán có quy mô lớn hơn. Trong đó có 2 yếu tố cơ bản và cốt lõi nhất cấu thành thuật ngữ Big Data là yếu tố lưu trữ và xử lý. Luận án sẽ tập trung vào hai phần này theo như biểu diễn sau:



**Hình 1.2: Phạm vi nghiên cứu**

Theo như [Hình 2] mô tả thì đề tài luận án sẽ xoay quanh việc tìm hiểu cách thức lưu trữ và xử lý phân tán (phần in đỏ trong hình ảnh) trên các máy hay nói cách khác là nghiên cứu về một quy trình xử lý dữ liệu lớn cơ bản hiện nay: Lập lịch, xử lý, lưu trữ và biểu diễn dữ liệu.

# **CHƯƠNG 2: CƠ SỞ LÝ THUYẾT**

## **2.1 Khái quát về hệ phân tán**

Khi ta đề cập đến Dữ liệu lớn Big Data thì ta phải nói đến hệ phân tán trên nhiều máy, có thể hình dung việc xử lý và lưu trữ dữ liệu trên một máy là điều bất khả thi và để thực hiện điều đó thì phải thực hiện nó trên một hệ phân tán đúng nghĩa. Hệ thống phân tán (Distributed System) là một hệ thống phần mềm trong đó các thành phần cấu tạo nên nó nằm trên các máy tính khác nhau và được kết nối thành mạng lưới. Các máy tính này phối hợp hoạt động với nhau để hoàn thành một nhiệm vụ chung bằng cách trao đổi thông điệp. [1]

Có một số mô hình phát triển hệ thống phân tán:

* Hệ thống máy khách/máy chủ: Đây là mô hình cơ bản nhất. Máy khách cung cấp đầu vào cho máy chủ và máy chủ trả lời bằng đầu ra. Máy khách muốn thực hiện một tác vụ trên máy chủ và máy chủ sẽ phân bổ và thực hiện tác vụ đó rồi gửi kết quả dưới dạng phản hồi. Loại máy chủ này có thể áp dụng cho nhiều máy chủ khác. [2]
* Hệ thống ngang hàng: Trong hệ thống này, mỗi nút thực thi nhiệm vụ của mình trên bộ nhớ được cấp phát cục bộ và chia sẻ dữ liệu thông qua phương tiện hỗ trợ. Các ứng dụng mạng máy tính sử dụng hệ thống ngang hàng để quản lý các bộ xử lý giao tiếp với nhau, duy trì cơ sở bộ nhớ độc lập. [3]
* Middleware: Đây là một ứng dụng nằm giữa hai ứng dụng khác nhau và cung cấp dịch vụ cũng như lợi ích cho cả hai.
* Ba tầng: Hệ thống ba tầng sử dụng một lớp và máy chủ riêng biệt cho từng chức năng của chương trình. Dữ liệu của khách hàng được lưu trữ ở tầng giữa, bao gồm một lớp ứng dụng, một lớp dữ liệu và một lớp trình bày. Hệ thống ba tầng này phổ biến trong các ứng dụng web hoặc trực tuyến.
* N-tier: Nó còn được gọi là hệ thống phân tán nhiều tầng. Hệ thống này có thể chứa bất kỳ số lượng chức năng nào, tương tự như hệ thống ba cấp. Hệ thống N-tier này phổ biến hơn trong các ứng dụng web và hệ thống dữ liệu.
* Mô hình Master-Slave: Đây là mô hình phổ biến trong các hệ thống phân tán lớn, trong đó một nút chủ (Master) quản lý và điều phối các nút phụ (Slave). Nút chủ chịu trách nhiệm phân công công việc, theo dõi tiến độ và thu thập kết quả từ các nút phụ. Các nút phụ thực hiện các tác vụ cụ thể và báo cáo lại cho nút chủ. Mô hình này giúp quản lý hiệu quả và phân bổ tài nguyên đồng đều trong hệ thống.

Các đặc tính của hệ thống phân tán:

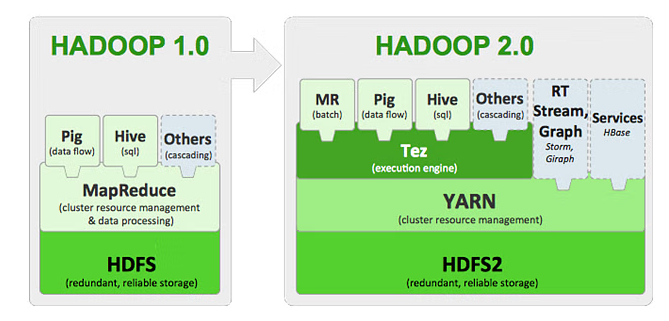
* Chia sẻ tài nguyên: Các tài nguyên trên máy tính được quản lý bởi chương trình quản lý tài nguyên. Việc chia sẻ tài nguyên giúp tiết kiệm chi phí đầu tư và tăng khả năng sẵn sàng của hệ thống.
* Tính trong suốt: Hệ thống phân tán phải hoạt động liên tục mà không bị gián đoạn.
* Tính mở: Hệ thống phân tán phải có khả năng mở rộng để đáp ứng nhu cầu tăng trưởng.
* Tính co giãn: Hệ thống phân tán phải có khả năng thích nghi với sự thay đổi của môi trường và yêu cầu của người dùng.

## **2.2 Giới thiệu về Apache Hadoop**

Nhiều giao diện lập trình ứng dụng điện toán song song như MPI chia các vấn đề tính toán lớn thành các vấn đề nhỏ hơn được phân bổ giữa các đơn vị xử lý trung tâm (CPU) khác nhau và được thực thi đồng thời. Tuy nhiên, nhu cầu liên lạc giữa các bộ xử lý khiến khả năng chịu lỗi trở thành một thách thức đối với tính toán song song vì mỗi nút được yêu cầu không bị lỗi. Sự thiếu hụt này được khắc phục trong Hadoop bằng cách hạn chế các loại thông tin liên lạc này. Chương này tập trung thảo luận về các thành phần HDFS và MapReduce trong Hadoop.

Apache Hadoop là một framework cho phép xử lý phân tán các tập dữ liệu lớn trên một cụm (cluster) các máy tính phân tán đúng nghĩa, sử dụng mô hình lập trình đơn giản. Nó được thiết kế để có khả năng mở rộng từ một đến hàng nghìn máy tính, nơi mỗi máy tính trong cụm phân tán đảm nhận vai trò tính toán và lưu trữ dữ liệu cục bộ.

Một điểm đặc biệt của Apache Hadoop là khả năng tự phát hiện và quản lý lỗi tại tầng ứng dụng, thay vì sử dụng cơ chế sẵn sàng cao (High Availability - HA) thông qua phần cứng. Điều này được thực hiện bằng cách thiết lập các ứng dụng (node) phụ trợ dưới dạng hoạt động hoặc chế độ chờ (active/standby) trên cụm máy tính, để đảm bảo tính sẵn sàng cao (HA) tại các điểm có thể xảy ra lỗi.



**Hình 2.1: Hadoop 1.0 và Hadoop 2.0**

Apache Hadoop được tạo bởi 2 thành phần chủ chốt được phát triển từ những thời kỳ đầu tiên bao gồm một hệ thống file phân tán HDFS (Hadoop Distributed Filesystem) và một Layer để tính toán theo mô hình xử lý Map Reduce. Hadoop là một open source cho phép xử lý dữ liệu theo lô (Batch Processing) và có khả năng xử lý khối lượng dữ liệu cực lớn nhưng không phải sự lựa chọn tối ưu cho các truy cập yêu cầu độ trễ thấp.

Hadoop sử dụng một cụm các server thông thường hay với cái tên khác là các node để lưu trữ, tính toán. Việc tính toán trên HDFS (Hadoop Distributed Filesystem) được thực hiện một cách song song và trừu tượng với các nhà phát triển giúp họ tránh được việc lập trình mạng và xử lý bài toán đồng bộ phức tạp. Không giống như nhiều hệ thống phân tán khác, Hadoop cung cấp việc xử lý logic trên nơi lưu dữ liệu mà không phải lấy dữ liệu từ các máy khác giúp tăng hiệu năng một cách mạnh mẽ.

Hadoop bao gồm những module sau:

* Hadoop Common: Các tiện ích cơ bản hỗ trợ Hadoop.
* HDFS (Hadoop Distributed Filesystem): Hệ thống file phân tán cung cấp khả năng truy vấn song song tối đa hóa theo đường truyền truy cập bởi ứng dụng.
* Hadoop YARN: Framework quản lý lập lịch tác vụ và tài nguyên trên cụm (có từ phiên bản Hadoop 2.0 trở lên).
* Hadoop MapReduce: Hệ thống YARN-based để xử lý tập dữ liệu lớn.

### **2.2.1 Các ứng dụng nền tảng của Apache Hadoop**

Apache phát triển một số ứng dụng để việc sử dụng MapReduce một cách dễ dàng hơn đồng thời hỗ trợ việc ghi và lấy dữ liệu trên HDFS được đơn giản. Sau đây là một số ứng dụng thông dụng được giới thiệu rộng rãi đi kèm mà chúng ta cần quan tâm đến như sau:

* **Apache Hive**

Hive tạo ra một cơ sở dữ liệu quan hệ dạng trừu tượng cho phép các nhà phát triển có thể truy vấn dữ liệu bằng SQL. Thực chất đây là việc thi hành một hoặc nhiều job MapReduce trên các cụm. Hive thực hiện việc tạo ra một schema dạng bảng trên tập các file đang tồn tại trên HDFS và quản lý các bản ghi được trích xuất khi chạy một query. Bản thân dữ liệu trên đĩa không thay đổi mà chỉ được lấy ra tại thời điểm query. Các câu lệnh HiveQL được dịch và thực thi trên các lớp map và reduce có sẵn tương ứng với câu lệnh SQL đó. thông thường truy vấn bằng Hive được sử dụng cho các tác vụ Batch Processing chứ không phải là Stream Processing vì truy vấn Hive tương đối mất thời gian. [5]

* **Apache Pig**

Tương tự như Hive, Apache Pig được phát triển để giúp đơn giản hóa việc sử dụng MapReduce job mà không cần phải viết mã Java. Thay vào đó, người dùng có thể viết các công việc xử lý dữ liệu bằng ngôn ngữ kịch bản cấp cao mà Pig cung cấp. Trong trường hợp cần thực hiện các thao tác xử lý dữ liệu đặc biệt mà Pig không hỗ trợ, chúng ta có thể mở rộng các kịch bản Pig bằng Java. [5]

* **Apache Sqoop**

Việc chuyển đổi dữ liệu từ dữ liệu quan hệ sang dữ liệu trên Hadoop là một trong những thách thức quan trọng và phổ biến trong ngành ngày nay. Sqoop, viết tắt của "SQL to Hadoop", là một công cụ được phát triển để thực hiện chuyển đổi dữ liệu hai chiều giữa Hadoop và hầu hết các cơ sở dữ liệu sử dụng JDBC driver.

Sử dụng MapReduce, Sqoop thực hiện các hoạt động một cách song song mà không cần phải viết mã code. Sqoop hỗ trợ các plugin cho từng loại cơ sở dữ liệu cụ thể, cung cấp các tính năng cơ bản của hệ quản trị cơ sở dữ liệu quan hệ đó. [5]

Hiện nay, Sqoop có hỗ trợ sẵn các kết nối (connectors) cho MySQL và PostgreSQL và đi kèm với một số connector miễn phí để hỗ trợ Teradata, Netezza, SQL Server và Oracle. Điều này giúp người dùng dễ dàng tích hợp dữ liệu từ các nguồn dữ liệu quan trọng vào Hadoop để thực hiện các công việc xử lý và phân tích dữ liệu phức tạp.

* **Apache Flume**

Apache Flume là một hệ thống để tập hợp các dữ liệu streaming được thiết kế để truyền khối lượng lớn dữ liệu vào hệ thống ví dụ như Hadoop. Nó hỗ trợ kết nối và ghi dữ liệu trực tiếp tới HDFS.

* **Apache Oozie**

Trong thực tế, sẽ có rất nhiều job Map-Reduce được chạy do đó Apache Oozie được sinh ra với mục đích trở thành một workflow engine để điều phối các job này trên các Hadoop cluster. Workflow có thể hoạt động dựa trên cơ chế thời gian hoặc cơ chế sự kiện. Oozie là một REST service để quản lý workflow và trạng thái hoạt động của luồng.

* **Apache HBase**

Apache HBase là cơ sở dữ liệu phân tán (non relational), độ trễ thấp được xây dựng trên nền HDFS. HBase đưa ra một mô hình dữ liệu linh hoạt có các thuộc tính scale-out với API rất đơn giản. Dữ liệu trên HBase được lưu trữ dưới dạng bán cột (semi-columnar) phân chia bởi các hàng trong các region. Các bảng trên HBase có thể lên tới hàng trăm terabyte hoặc một số trường hợp lên đến petabytes. Hiện nay, HBase được sử dụng để phục vụ một khối lượng lớn dữ liệu trong các hệ thống real-time.

* **Apache ZooKeeper**

Apache ZooKeeper hỗ trợ các tính năng của hệ thống phân tán. Trong thực tế nhiều dự án sử dụng Hadoop có sử dụng ZooKeeper để hỗ trợ các tính năng cần thiết cho hệ thống phân tán (leader election, locking, group membership, service location, configuration services).

* **Apache HCatalog**

Apache HCatalog là một dịch vụ cung cấp khả năng chia sẻ schema và các dịch vụ truy cập dữ liệu giữa các ứng dụng trong cùng một hệ thống. Về lâu dài, HCatalog sẽ cung cấp việc kết hợp các tool ví dụ như Hive và Pig để chúng có thể chia sẻ các thông tin về dataset và metadata.

* **Apache Whirr**

Apache Whirr được xây dựng để đơn giản hóa việc tạo và deploy các clusters trên môi trường cloud ví dụ như Amazon’s AWS.

### **2.2.2 Hệ thống quản lý tập tin phân tán Hadoop**

Khi tập dữ liệu hay kích thước của dữ liệu vượt quá khả năng lưu trữ trên một máy, tất yếu sẽ dẫn đến việc tập dữ liệu bị phân chia và lưu trữ trên nhiều máy tính. Các hệ thống tập tin quản lý việc lưu trữ dữ liệu trên một mạng máy tính gọi là hệ thống tập tin phân tán. Việc hoạt động trên một môi trường liên mạng, nên các hệ thống tập tin phân tán phức tạp hơn rất nhiều so với một hệ thống tập tin cục bộ. Chẳng hạn như việc quản lý trạng thái hoạt động (live/dead) của các máy chủ tham gia vào hệ thống quản lý tập tin cũng đã là một thách thức lớn. May thay Hadoop HDFS lại giúp chúng ta giải quyết vấn đề đó.

#### **2.2.2.1 Giới thiệu HDFS**

HDFS (viết tắt từ Hadoop Distributed File System) với sự nỗ lực tạo ra một nền tảng lưu trữ dữ liệu đáp ứng cho một khối lượng dữ liệu lớn và chi phí rẻ. HDFS ra đời trên nhu cầu lưu trữ dữ liệu của Nutch, một dự án về máy tính tìm kiếm (Search Engine) nguồn mở. HDFS kế thừa các mục tiêu chung của các hệ thống tập tin phân tán trước đó như dữ liệu tin cậy, khả năng mở rộng và hiệu suất ổn định và hỗ trợ tốt cho các ứng dụng xử lý dữ liệu lớn.

#### **2.2.2.2 Khái niệm nền tảng kiến trúc HDFS**

Block: Mỗi tệp nhỏ được lưu trữ theo khối (block) và ứng với siêu dữ liệu (metadata) cho từng khối sẽ được lưu trữ trong NameNode con các tệp nhỏ sẽ được lưu trữ trên các DataNode và được sao chép ở nhiều nút. Kích thước mặc định là 128MB đối với phiên bản từ 2.0 trở lên và 64MB từ các phiên bản trở xuống. Một số trường hợp tệp nhỏ được tạo ra:

File System Namespace: HDFS tổ chức tập tin phân cấp theo mô hình truyền thống. Với HDFS người dùng cũng có thể tạo, xóa, di chuyển, đổi tên tập tin thư mục như các hệ thống tập tin truyền thống thông thường.

Data Replication: HDFS được thiết kế để lưu trữ các tập tin cực lớn. Trên một cụm Hadoop mỗi tập tin được chia thành nhiều block có thứ tự lưu trữ trên nhiều máy. Việc nhân bản các block nhằm tăng khả năng chịu lỗi cho hệ thống. Mỗi block sẽ được nhân bản bao nhiêu lần tùy theo cấu hình của hệ thống. Các DataNode có nhiệm vụ lưu trữ các block mà nó được NameNode phân công.

NameNode và DataNode: HDFS là một sự phát triển riêng tách biệt so với các hệ thống tập tin phân tán truyền thống và là một kiến trúc Master/Slave. HDFS lưu trữ các tập tin về dữ liệu của ứng dụng và các tập tin chứa siêu dữ liệu (metadata) của hệ thống một cách riêng biệt. Siêu dữ liệu hay các định nghĩa tập tin được lưu trữ trên các máy chủ chuyên dụng gọi là NameNode và dữ liệu của các ứng dụng được lưu trữ trên các máy con khác được gọi là DataNode. Hiện nay từ phiên bản Hadoop 3.0 trở lên việc NameNode có thể mở rộng theo chiều ngang là việc khả thi (nghĩa là thêm nút NameNode) và các NameNode độc lập, không cần phối hợp với nhau và chỉ giao tiếp với các DataNode qua Block Pools từ đó đảm bảo được việc dữ liệu bền vững, nội dung của tập tin được nhân rộng trên nhiều DataNode khác nhau. Hay có thể biểu diễn như sau:

* Một cụm HDFS có một hoặc nhiều NameNode (master – nút chủ)
* Một cụm HDFS có một hoặc nhiều các DataNode (worker - nút thợ)

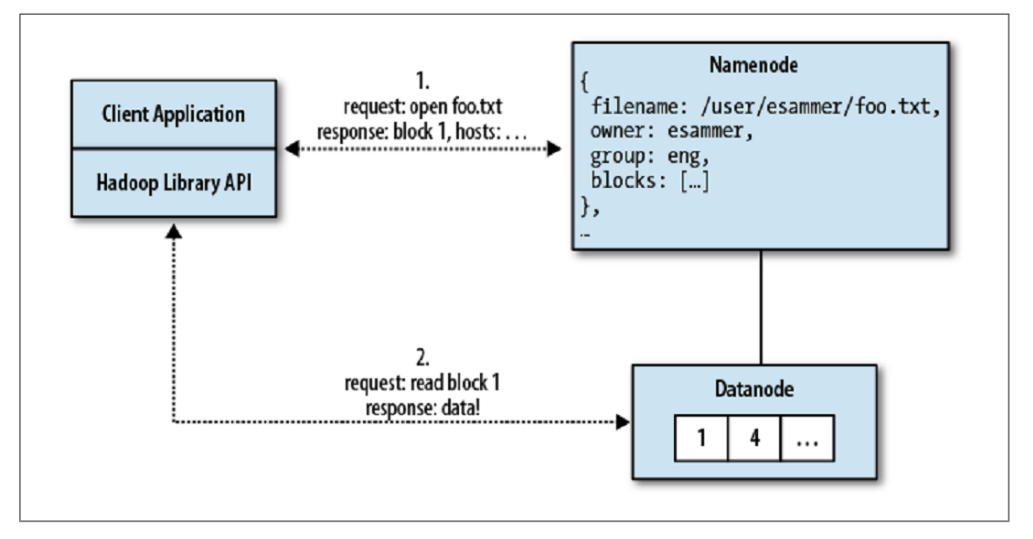
NameNode quản lý các namespace filesystem. Nó quản lý một filesystem tree và các metadata cho tất cả các file và thư mục trên tree. Thông tin này được lưu trữ trên đĩa vật lý dưới dạng không gian tên ảnh và nhật ký (edit log). NameNode còn quản lý thông tin các tệp nhỏ (block) được lưu trên các DataNode và có thể sao chép ở nhiều nút. Yêu cầu cài đặt duy nhất chỉ là chạy hệ điều hành GNU/Linux. HDFS được xây dựng trên ngôn ngữ Java nên bất kỳ máy nào hỗ trợ Java đều có thể thực thi cài đặt.

SecondaryNameNode: Trong HDFS cluster có duy nhất một NameNode, hệ thống không thể hoạt động nếu như không có NameNode. Vì tính chất quan trọng đó, việc sao lưu dự phòng cho NameNode là rất cần thiết. Đó chính là nhiệm vụ của Secondary NameNode. Định kỳ sau mỗi khoảng thời gian, Secondary NameNode sẽ kết nối đến NameNode để cập nhật các tập tin về cấu hình, trạng thái của hệ thống, toàn bộ các tập tin này sẽ được lưu lại thành một CheckPoint, dùng cho việc khôi phục lại NameNode nếu NameNode bị lỗi. Đồng thời, NameNode là một dịch vụ phải phục vụ rất nhiều DataNode trong hệ thống. Secondary NameNode còn có khả năng chịu tải với NameNode. Ngoài ra, việc định kỳ tạo CheckPoint giúp hệ thống chuyển sang trạng thái hoạt động nhanh hơn.

#### **2.2.2.3 Đọc ghi dữ liệu trên HDFS**

1. **Đọc dữ liệu**

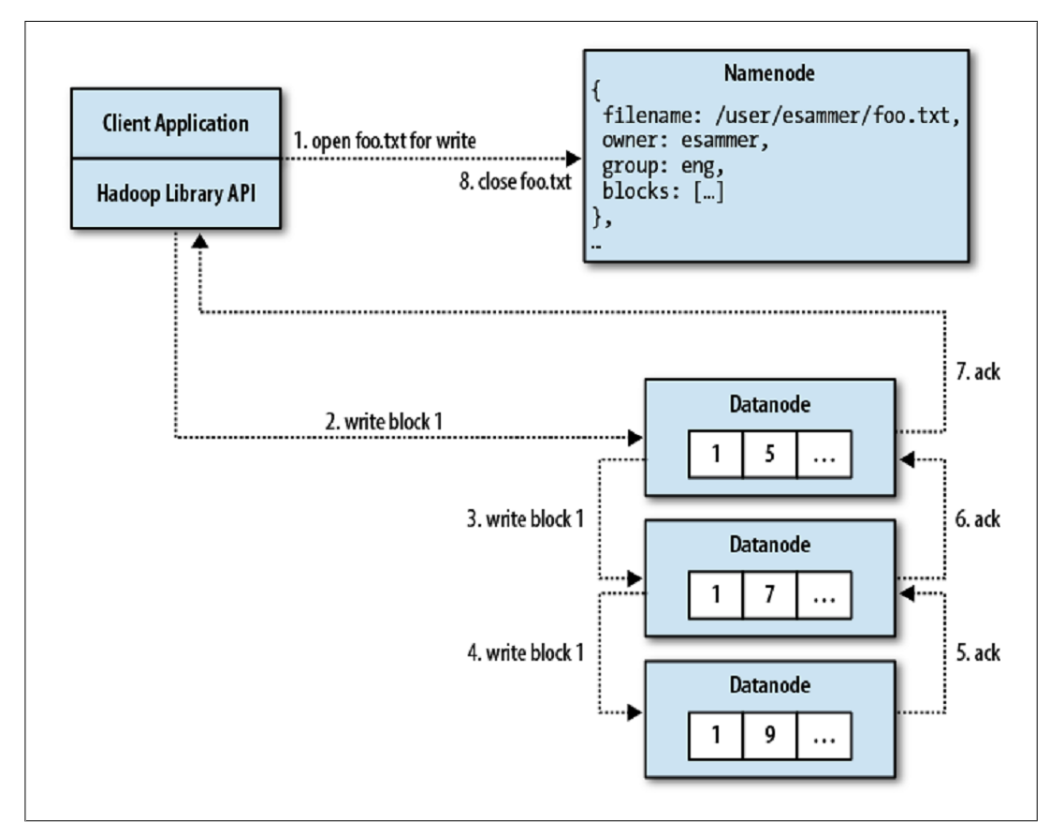
Khối dữ liệu block ID và địa chỉ IP đích máy chủ DataNode lưu trữ block đó, máy khách (client) có thể liên lạc với các DataNode để đọc các block cần thiết. Máy khách sẽ sử dụng các API Hadoop core cầm thiết để yêu cầu quyền đọc và tập dữ liệu muốn đọc từ NameNode, Khi đó NameNode sẽ thực hiện quá trình kiểm duyệt và trích xuất địa chỉ đích của các DataNodes chứa tập dữ liệu (blocks) muốn đọc của client .Quá trình lặp lại cho đến khi tất cả các khối trong file được đọc và máy khách đóng luồng đọc khi kết thúc.



**Hình 2.2: Quá trình đọc tập tin trên HDFS**

Hình 2.2 mô tả quá trình đọc dữ liệu của NameNode từ DataNode thông qua các API được cung cấp từ Library của Hadoop

1. **Ghi dữ liệu**



**Hình 2.3: Quá trình ghi tập dữ liệu trên HDFS**

Máy khách sẽ gửi yêu cầu tạo một file bằng việc sử dụng các FileSystem APIs do Hadoop cung cấp. Một yêu cầu được gửi đến NameNode để tạo tập tin metadata. Nếu được cho phép, thông tin metadata cho tập tin đó được khởi tạo; tuy nhiên lúc này chưa có một block nào liên kết với tập tin này. Một tiến trình trả kết quả được gửi lại cho máy khách xác nhận yêu cầu tạo file đã hoàn thành và bắt đầu có thể ghi dữ liệu khi đó dữ liệu từ máy khách sẽ được ghi vào một luồng stream và được chia ra thành các gói lưu trong queue của bộ nhớ. Khi đó một tiến trình riêng biệt sẽ hệ với NameNode để yêu cầu một tập DataNode phục vụ cho việc sao lưu dữ liệu và các khối (block). Máy khách sẽ tạo một kết nối trực tiếp đến datanode đầu tiên trong danh sách. DataNode đầu tiên đó sẽ kết nối lần lượt các DataNode khác. Các gói dữ liệu được ghi dần vào các DataNode. Mỗi DataNode sẽ phản hồi dữ liệu ghi thành công hay không. Quá trình này kết thúc khi toàn bộ các gói dữ liệu đã được ghi hết xuống và được lưu tại các khối (block) của các DataNode.

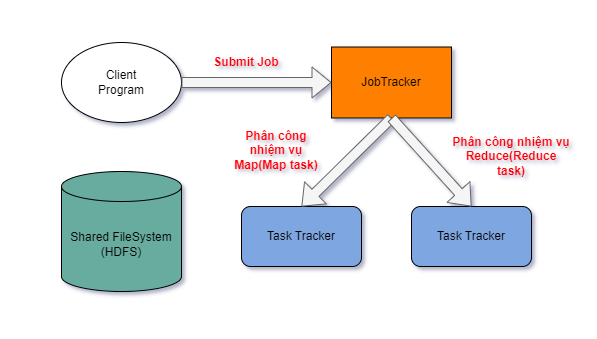
### **2.2.3 Giải thuật xử lý song song MAP-REDUCE**

#### **2.2.3.1 Giới thiệu MAP-REDUCE**

Map-Reduce là một mô hình lập trình để biểu diễn các tính toán phân bố trên một lượng lớn dữ liệu và một khung thực thi phân tán xử lý dữ liệu quy mô lớn trên các cụm máy chủ. Map-Reduce được phát triển bởi Google và được xây dựng trên các nguyên tắc nổi tiếng trong xử lý song song và phân tán. Map-Reduce đã được áp dụng rộng rãi thông qua việc hiện thực dưới dạng nguồn mở trên hệ thống Hadoop. Hiện nay có hai nhánh về xử lý dữ liệu đó là xử lý hàng loạt và xử lý luồng truyền phát, Map-Reduce thuộc vào xử lý hàng loạt.

Xử lý hàng loạt liên quan đến việc xử lý một khối lượng dữ liệu lớn cùng một lúc, theo các khoảng thời gian đã lên lịch xử lý. Ngược lại, xử lý luồng liên quan đến việc xử lý dữ liệu được sinh liên tục theo thời gian thực. Việc xử lý luồng tập trung vào việc thời gian phản ứng với dữ liệu sinh ra phải đáp ứng gần như là với thời gian thực. Mô hình dựa trên phương pháp chia để trị: một bài toán lớn được chia thành nhiều bài toán con đủ nhỏ để có thể giải quyết trên một máy tính. Các bài toán con này sau đó được xử lý độc lập song song ở các nút khác nhau của cụm. Cuối cùng, lời giải của các bài toán con này được hợp nhất để có được kết quả của bài toán lớn.

#### **2.2.3.2 Kiến trúc của MAP-REDUCE**



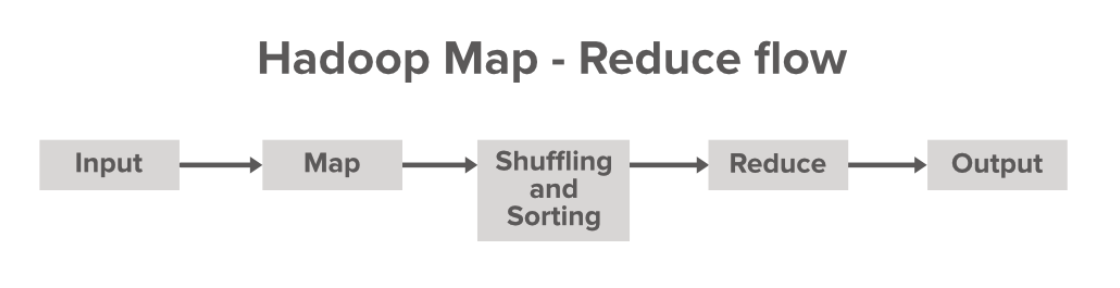
**Hình 2.4: Các thành phần trong nhiệm vụ xử lý Map-Reduce**

Kiến trúc Map-Reduce được mô tả như sau:

* Client Program: là một chương trình Hadoop Map-Reduce được cài đặt được sử dụng và tiến hành chạy một công việc Map-Reduce.
* JobTracker: Có chức năng tiếp nhận và điều phối các công việc (job) và là thành phần điều khiển chủ chốt của Map-Reduce.
* TaskTracker: Có chức năng tiếp nhận các task từ JobTracker để thực hiện nhiệm vụ.
* HDFS: là hệ thống quản lý tập tin phân tán được dùng cho việc chia sẻ các tập dữ tin và Map-Reduce hoạt động rất tốt trên HDFS.

#### **2.2.3.3 Bài toán đếm chữ cái**

Một trong những ví dụ kinh điển mở đầu của giải thuật xử lý song song Map-Reduce đó chính là ví dụ đếm từ. Trong đó Map-Reduce sẽ chia ra hai giai đoạn chính:



**Hình 2.5: Quy trình Hadoop Map-Reduce**

Cả hai giai đoạn này đều có đầu vào và đầu ra ở dạng key, value. Người phát triển cần viết 2 hàm chức năng để có thể xử lý dữ liệu đó là hai hàm Map và Reduce. Với ví dụ kinh điển đếm từ ta có thể biểu diễn như sau:

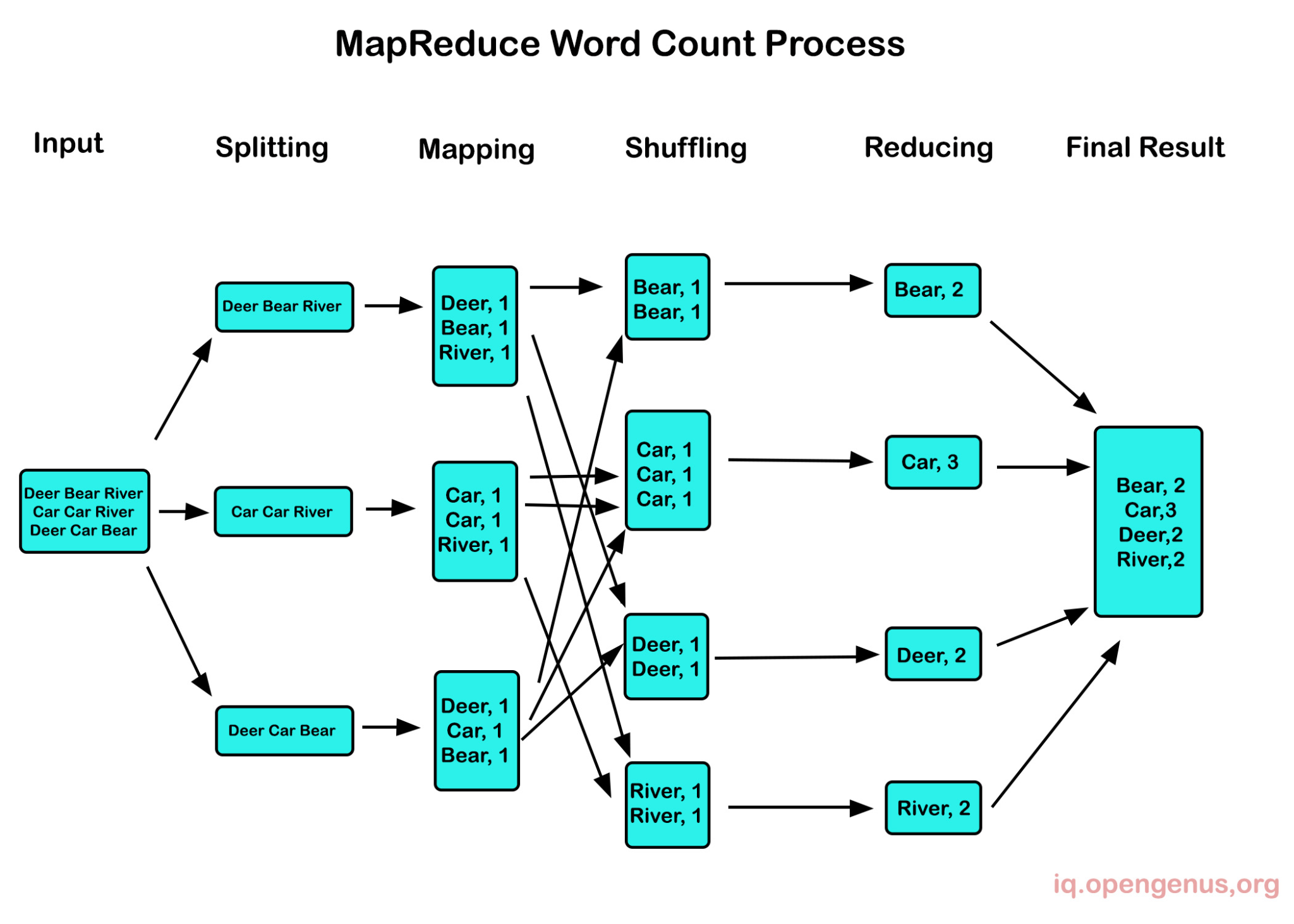
Input: dữ liệu dưới dạng Key → Value

Lập trình viên viết 2 thủ tục:

Map(k, v) → <k', v'>\*

Reduce(k', <v'>\*) → <k', v''>\*

Map biến mỗi key k thu được bằng thành cặp <k', v'>. Reduce nhận đầu vào là khóa k' và danh sách các giá trị v' và trả về kết quả là cặp <k', v''>. Ví dụ như hình bên dưới đây biểu diễn quy trình thực hiện của bài toán trên.



**Hình 2.6: Quy trình Wordcount trong Map-Reduce**

Mô hình lập trình MapReduce chỉ có hai giai đoạn: ánh xạ(Map) và/hoặc thu gọn(Reduce). Trong giai đoạn đầu tiên, dữ liệu được phân phối đến tất cả các máy trong cụm để xử lý và tạo ra kết quả dưới dạng cặp khóa-giá trị. Ở giai đoạn thứ hai, các kết quả được nhóm theo khóa và tất cả các cặp có cùng khóa sẽ được thu thập và phân tích trên cùng một máy tính. Người dùng chỉ phải cung cấp một chức năng để chạy trong mỗi giai đoạn để hoạt động trên đầu vào, trong khi mọi thứ khác đều do khung xử lý. Các ứng dụng và luồng dữ liệu phức tạp có thể được triển khai bằng cách xâu chuỗi các giai đoạn này, chuỗi này tạo thành một “biểu đồ” hoạt động - được gọi là “đồ thị chu kỳ có hướng” hoặc DAG, DAG chứa chuỗi hành động được kết nối với nhau trong quy trình làm việc. Trong trường hợp MapReduce, DAG là một chuỗi các nhiệm vụ ánh xạ(Map) và thu gọn(Reduce) được sử dụng để triển khai ứng dụng và nhiệm vụ của nhà phát triển là xác định từng tác vụ và xâu chuỗi chúng lại với nhau, điều này phát sinh rất nhiều chi phí do chi phí duy trì đầu ra của mỗi chương trình Map-Reduce vào đĩa. Kết quả là nhiều framework mới hơn đã được tạo ra để khắc phục những nhược điểm này như Dryad, Tez và Spark.

Dryad là một khung tính toán cụm khác được phát triển bởi Microsoft. Ứng dụng Dryad được mô hình hóa dưới dạng đồ thị có hướng (DAG) trong đó các tác vụ tính toán được thực hiện trong các đỉnh và kênh truyền thông được biểu diễn dưới dạng các cạnh. Để sử dụng Dryad, người viết phải viết các chương trình chạy ở các đỉnh và kết nối chúng với nhau qua các kênh truyền. Việc này giúp Dryad có thể chỉ định nhiều giao tiếp khác nhau đồng thời ứng dụng Dryad có thể có nhiều toán tử có nghĩa là thuật toán trong Dryad dễ triển khai hơn so với Map-Reduce. Nhưng thật không may, Microsoft đã ngừng phát triển Dryad và quyết định tập trung vào Hadoop Map-Reduce vào năm 2011.

Tez là một khung công tác mã nguồn mở để xây dựng thời gian chạy tập trung vào luồng dữ liệu phức tạp, thay vì chỉ hỗ trợ cung cấp các công cụ viết ứng dụng cho người dùng, mục đích chính của Tez là cung cấp môi trường và thư viện để viết chính công cụ thực thi. Tez cho phép người dùng lập mô hình tính toán dưới dạng DAG - tương tự như Dryad, nhưng với khả năng kiểm soát và tùy chỉnh tốt hơn các cạnh và đỉnh. Ngoài ra, nó còn hiển thị API để truy cập và sửa đổi DAG để hỗ trợ tối ưu hóa động. Xây dựng trên nền tảng Hadoop Yarn, một thành phần quan trọng Ngoài Hadoop Map-Reduce, Tez còn cung cấp nhiều tính năng tiên tiến như bảo mật, dữ liệu nhận thức của địa phương, tái sử dụng tài nguyên và khả năng chịu lỗi.

## **2.3 Giới thiệu Apache Spark**

Ngày nay hầu hết rất nhiều hệ thống xử lý dữ liệu thông tin lớn đều đang sử dụng Hadoop để phân tích và lưu trữ dữ liệu lớn. Ưu điểm lớn nhất của Hadoop là được dựa trên một mô hình lập trình song song với kiến trúc xử lý song song là Map-Reduce, mô hình này cho phép khả năng tính toán có thể mở rộng, linh hoạt, khả năng chịu lỗi, chi phí rẻ [5]. Điều này cho phép tăng tốc thời gian xử lý các dữ liệu lớn nhằm duy trì tốc độ, giảm thời gian chờ đợi khi dữ liệu ngày càng lớn. Hadoop đã được sử dụng cho rất nhiều bài toán xử lý dữ liệu lớn. Đặc biệt các công ty công bố áp dụng Hadoop rất nhiều điển hình như: Google, Yahoo, FaceBook, Amazon,... . Dù có rất nhiều điểm mạnh nhưng Map-Reduce lại có một nhược điểm chí mạng đó là tất cả các thao tác đều phải thực hiện trên ổ đĩa cứng điều này làm giảm tốc độ tính toán đi gấp nhiều lần nghĩa là các kết quả trung gian thu được trong mỗi lần lặp được ghi vào HDFS và được tải lên từ đó(thao tác với disk) gây tốn quá nhiều I/O của đĩa và tài nguyên trên các máy. Để khắc phục nhược điểm này thì Apache Spark ra đời. Apache Spark có thể chạy nhanh hơn 10 lần so với Map-Reduce ở trên đĩa cứng và 100 lần khi chạy trên bộ nhớ RAM [4] theo (Nguồn <https://spark.apache.org/>). Và Spark sẽ chỉ ghi vào đĩa khi bộ nhớ được cấp để xử lý tác vụ đầy.

Apache Spark là một Open Source Cluster Computing Framework được phát triển vào năm 2009 bởi AMPLab tại đại học California, Berkeley. Sau này, Spark đã được trao cho Apache Software Foundation vào năm 2013 và được phát triển cho đến nay. Apache Spark được phát triển nhằm tăng tốc khả năng tính toán xử lý trên tập dữ liệu lớn của Hadoop.

Spark cho phép xây dựng và phân tích nhanh các mô hình dự đoán, cung cấp khả năng truy xuất dữ liệu lớn cùng lúc. Thêm vào đó, Spark còn cung cấp tính năng streaming hay micro batch processing để xử lý luồng, được sử dụng làm hậu xử lý cho các ứng dụng real time ngày nay bằng cách nạp toàn bộ dữ liệu vào bộ nhớ và nó chỉ được xếp vào loại gần sát với thời gian thực(mili giây thay vì nano giây).

Với nhu cầu phân tích dữ liệu song song ngày càng tăng, Spark đã xuất hiện như một phần mở rộng cho Hadoop Map-Reduce như một giải pháp thân thiện hơn với người dùng. Khi ta có một tác vụ lớn cần xử lý trên nhiều máy, Spark cho phép ta phân chia tác vụ này thành những phần dễ quản lý hơn. Sau đó, Spark sẽ chạy các tác vụ này trong bộ nhớ, trên các cluster của nhiều server khác nhau để khai thác tốc độ truy xuất nhanh từ RAM. Spark sử dụng API RDD (Resilient Distributed Dataset) để xử lý dữ liệu. Spark hiện nay được coi là một khung tính toán phân tán nhanh và lớn nhất với nhiều thư viện hữu ích đi kèm như Spark SQL (với kiểu dữ liệu DataFrames), Spark Streaming, MLlib(machine learning: classification, regression, clustering, collaborative filtering, và dimensionality reduction) và GraphX (tính toán song song trên dữ liệu đồ thị).

### **2.3.1 Lợi thế của Apache Spark**

Apache Spark có các điểm mạnh sau đây.

* Tốc độ: Spark thường nhanh hơn MapReduce nhờ việc lưu trữ dữ liệu xử lý trực tiếp trong bộ nhớ (in-memory processing). Việc này giảm thiểu thời gian cần thiết để đọc và ghi dữ liệu trên đĩa cứng, dẫn đến tăng tốc độ xử lý dữ liệu. Spark cũng có khả năng tối ưu hóa thực thi song song và có thể chạy trên cụm máy tính với hiệu suất cao.
* Hỗ trợ đa ngôn ngữ và cực kỳ linh hoạt: Spark cung cấp API cho nhiều ngôn ngữ lập trình như Java, Scala và Python, cho phép các nhà phát triển sử dụng ngôn ngữ yêu thích của họ để phát triển ứng dụng. Điều này tạo điều kiện thuận lợi cho việc tích hợp và phát triển ứng dụng trên nền tảng Spark.
* Phân tích nâng cao: Spark không chỉ hỗ trợ các phép toán Map và Reduce, mà còn cung cấp các thư viện và API cho phép thực hiện nhiều loại phân tích phức tạp như truy vấn SQL, xử lý dữ liệu theo luồng (stream processing), học máy (machine learning) và xử lý đồ thị (graph processing). Điều này làm cho Spark trở thành một nền tảng linh hoạt và mạnh mẽ cho các ứng dụng phân tích dữ liệu đa dạng. Ngoài ra cơ chế phân tích và xử lý các tập dữ liệu lớn rất tối ưu và thân thiện với người dùng.
* Hỗ trợ đa nền tảng: Spark có thể linh hoạt lựa chọn cho việc triển khai trên nhiều môi trường khác nhau, từ hệ thống máy chủ doanh nghiệp(on-premises) xoay quanh hệ sinh thái dữ liệu lớn Apache Hadoop đến các môi trường đám mây, đa đám mây hoặc hay thậm chí tất cả cùng cộng hưởng đáp ứng được hầu hết nhu tất cả các nhu cầu của doanh nghiệp.
* Tận dụng tối đa tài nguyên sẵn có: Từ việc tính toán trên bộ nhớ, xử lý theo thời gian thực và hỗ trợ được hầu hết các phân tích thông minh cần thiết của một doanh nghiệp.

### **2.3.2 Các khái niệm của Apache Spark**

#### **2.3.2.1 Resilient Distributed Datasets**

Cốt lõi Spark dựa trên khái niệm Bộ dữ liệu phân tán linh hoạt RDD (Resilient Distributed Datasets, nó cho phép các lập trình viên thực hiện xử lý dữ liệu có khả năng chịu lỗi trên toàn bộ cụm máy tính. Đặc tính của RDD được thể hiện dưới tên của nó. [7]

* Khả năng phục hồi(Resilient) : Có khả năng chịu lỗi, sự cố xảy ra và có khả năng xây dựng lại dữ liệu khi có lỗi
* Phân tán(Distributed) : Dữ liệu của RDD được phân phối trên tất cả các máy tham gia vào cụm. Việc phân vùng dữ liệu này đảm bảo toàn bộ tài nguyên của cụm được sử dụng để xử lý dữ liệu và giảm thiểu tác động tiêu cực của sự cố máy.
* Tập dữ liệu(Dataset): RDD hoạt động như một tập hợp các bản ghi. Mỗi bản ghi độc lập với bản ghi khác và không có ràng buộc nào về việc có thể được sử dụng làm bản ghi. Yêu cầu này là cần thiết vì bản ghi phải được chuyển sang nhị phân trước khi truyền qua mạng đến một máy khác.

Có thể thấy hai tính chất quan trọng của RDD đó là phân tán dữ liệu(Partitions) trên các cụm và có khả năng chịu lỗi. Thuật ngữ Khả năng chịu lỗi xác định khả năng tạo dữ liệu tự động hoặc dữ liệu quay trở lại trạng thái ban đầu khi xảy ra lỗi bất ngờ có khả năng mất dữ liệu. Dữ liệu được ghi vào RDD được phân vùng và lưu trữ thành nhiều nút thực thi. Nếu một nút thực thi bị lỗi trong thời gian chạy thì nó sẽ ngay lập tức được sao lưu từ nút thực thi tiếp theo. RDD có thể lưu trữ dữ liệu có cấu trúc, không cấu trúc và bán cấu trúc. Thông thường, RDD chỉ cho phép đọc, phân mục tập hợp của các bản ghi. RDDs có thể được tạo ra qua Driver, RDD là một tập hợp có khả năng chịu lỗi mỗi thành phần có thể được tính toán song song. Có hai cách để tạo RDDs

* Tạo từ một tập hợp dữ liệu có sẵn trong ngôn ngữ sử dụng như Java, Python, Scala
* Lấy từ dataset hệ thống lưu trữ bên ngoài như HDFS, Hbase hoặc các cơ sở dữ liệu quan hệ.

RDD là một tập hợp các bản ghi bất biến và chỉ có thể được tạo thông qua các hoạt động xác định từ dữ liệu đầu vào hoặc các RDD khác. Kết quả là, miễn dữ liệu đầu vào ổn định thì nội dung của RDD được chỉ định bởi các phép biến đổi này luôn tốt.

Ngoài ra, Spark cho phép thực hiện tính toán lười biếng(lazy-computation): RDD chỉ được thực hiện khi các actions được gọi. Vì cụm thường phải theo dõi nhiều RDD cùng một lúc nên tính toán RDD nếu không cần thiết, Spark có thể lưu giữ nhiều nội dung của các RDD có liên quan khác trong bộ nhớ chính càng lâu càng tốt. Bộ nhớ đệm dữ liệu thông minh này cải thiện đáng kể hiệu suất của hệ thống.

Cuối cùng, người dùng cũng có thể yêu cầu Spark lưu vào bộ nhớ đệm hoặc kiểm tra các RDD cụ thể. Đây là một khả năng rất mạnh mẽ, bởi vì người dùng có thể đưa ra phán đoán tốt hơn về việc RDD nào cần lưu vào bộ nhớ đệm nhờ kiến thức về thuật toán của họ. Đồng thời đây cũng là điểm kiểm tra nội dung của RDD có thể cho phép khôi phục nhanh hơn trong trường hợp có lỗi.

#### **2.3.2.2 Khái niệm cốt lõi của Spark**

Transformations: Là các phép biến đổi tạo ra một RDD mới bằng cách chạy một số thao tác trên RDD hiện tại. Các phép biến đổi trong Spark được đánh giá Lazy có nghĩa là chúng chỉ được tính toán khi chúng được yêu cầu tính toán kết quả bằng một Action, có hai loại chính là Wide Transformations và Narrow Transformations chúng khác nhau ở chỗ Wide Transformations sẽ tạo ra hiện tượng Shuffle dữ liệu ở trên các nút nghĩa là trên tất cả các nút dữ liệu sẽ bị xáo trộn và thay đổi gây nên hiện tượng tốn tài nguyên nên phải cân nhắc và sắp đặt các hàm này đúng vị trí khi sử dụng.

* Narrow (pipeline table): Mỗi phân vùng của RDD mẹ được sử dụng bởi tối đa một phân vùng của RDD con. Nó cho phép thực thi theo đường ống trên một nút cụm. Phục hồi lỗi hiệu quả hơn vì chỉ cần tính toán lại các phân vùng gốc bị mất.
* Wide (shuffle): Nhiều phân vùng con có thể phụ thuộc vào một phân vùng cha. Yêu cầu dữ liệu từ tất cả các phân vùng gốc phải có sẵn và được xáo trộn trên các nút. Nếu một số phân vùng bị xóa khỏi tất cả các phân vùng trước thì cần phải tính toán lại toàn bộ.

Actions: Trong Spark RDD là một loại hoạt động (hoặc hành động) kích hoạt quá trình tính toán trên dữ liệu trong RDD và trả về kết quả, chúng thường là điểm kết thúc của một chuỗi các biến đổi dữ liệu nghĩa là khi có Action cụ thể thì các Transformations mới được thực thi và Spark sẽ thực hiện các Transformations theo một cách thông minh nhất. Một Action trả về giá trị cho chương trình điều khiển hoặc nó có thể ghi dữ liệu vào bộ nhớ ngoài.

Key-Value Pair Datasets: Có một số thao tác và hành động nhất định chỉ khả dụng trên RDD với dữ liệu dạng key-value (ví dụ: reduceByKey()). Để các biến đổi này hoạt động, cấu trúc của RDD phải được lưu trữ theo kiểu Key-Value .

Persistence: Theo mặc định, RDD được đánh giá lazy, điều này có thể không phải lúc nào cũng được mong muốn nếu kế hoạch sử dụng lại RDD đã tính toán nhiều lần. Tuy nhiên, Spark cung cấp khả năng lưu trữ (hoặc lưu vào bộ đệm) RDD để truy cập nhanh vào bộ nhớ, đĩa hoặc cả hai (tùy thuộc vào mức lưu trữ đã chọn). Dữ liệu được lưu trữ cũng có thể được tuần tự hóa để làm cho các đối tượng chiếm ít không gian hơn với chi phí tiềm tàng là tăng thời gian CPU xử lý.

Biến dùng chung (Shared Variables), Spark hỗ trợ hai biến chia sẽ đó là Accumulators và Broadcast Variables. Đầu tiên, chúng ta xem xét khái niệm closure và lý do tại sao chúng ta cần các cơ chế đặc biệt để phân phối dữ liệu được chia sẻ trong Spark.

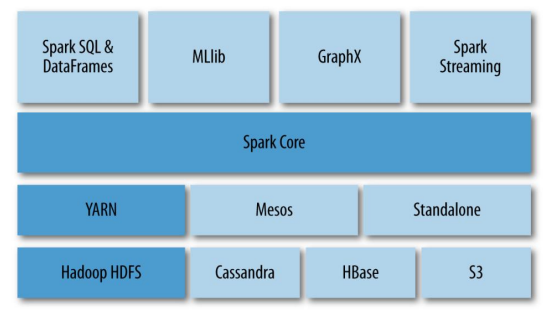
Closures: Ở chế độ cụm, trình điều khiển Spark sẽ tự động chia các hoạt động RDD thành các tác vụ và phân phối tính toán của chúng trên các nút cụm. Trước mỗi lần thực thi tác vụ, một bao đóng được tuần tự hóa được tính toán có chứa các biến và phương thức mà tác vụ sắp được tính toán cần truy cập vào. Việc đóng được tính toán thường chứa các bản sao của các biến trình điều khiển, khiến không thể thay đổi biến ban đầu trên nút trình điều khiển bằng cách thay đổi giá trị được truyền của nó trên nút cụm.



Accumulators: Bộ tích lũy (Accumulators) là các biến được thiết kế để an toàn và hiệu quả. Một Accumulators được tạo trong nút trình điều khiển và có thể được truyền xuống các nút cụm và các thay đổi của nó sẽ được truyền trở lại nút trình điều khiển. Các nút cụm chỉ có thể thêm vào Accumulators đã truyền thông qua thao tác cộng kết hợp và giao hoán và nút trình điều khiển là nút duy nhất có thể truy cập vào giá trị của Accumulators. Spark có hỗ trợ riêng cho các Accumulators kiểu số và bộ sưu tập, Accumulators các loại bổ sung cũng có thể được triển khai. Vì các Accumulators có tên Spark phiên bản 2.0 có thể được tạo để làm cho các giá trị của chúng được hiển thị trong giao diện người dùng web của Spark.

Broadcast Variables: Các Broadcast Variables có thể được sử dụng để truyền dữ liệu lớn một cách hiệu quả đến các nút cụm để truy cập chỉ đọc (read-only), đóng vai trò như một bộ đệm để tránh truyền dữ liệu mỗi khi yêu cầu tính toán. Trình điều khiển Spark khởi tạo Broadcast Variable, có thể là bất kỳ loại có thể tuần tự hóa nào và mỗi nút cụm sẽ nhận được bản sao của biến đó một lần. Khi đó, các tác vụ đang chạy sẽ có quyền truy cập nhanh, chỉ đọc vào dữ liệu phát sóng trên bất kỳ nút cụm nào.

### **2.3.3 Các thành phần của Apache Spark**



**Hình 2.7: Thành phần của Apache Spark**

Nền tảng của Apache Spark sẽ được chia làm 2 layer chính:

* Layer thứ nhất đó chính là: Spark Core chủ yếu làm việc với RDD như đã đề cập ở trên về lý thuyết nền tảng của Spark
* Layer thứ hai là: Spark SQL, Spark Structured Streaming, Spark Streaming, Spark MLib, Spark GraphX
* Layer thứ ba là môi trường làm việc cùng với Spark trong đó độ ưu tiên của YARN và Hadoop HDFS là ưu tiên đầu tiên

#### **2.3.3.1 Spark Core**

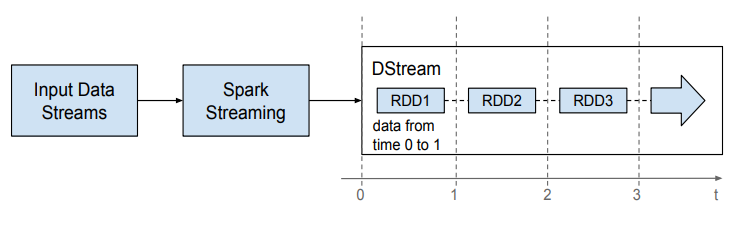
Chức năng cơ bản của Spark Core bao gồm lập lịch tác vụ, quản lý bộ nhớ, khôi phục lỗi, tương tác với hệ thống lưu trữ, v.v. Trình bày khái niệm chính của Spark về resilient distributed datasets để làm việc với dữ liệu thông qua API RDD của Spark.

#### **2.3.3.2 Spark SQL**

Spark SQL là một mô-đun để xử lý dữ liệu có cấu trúc thông qua giao diện SQL. Dữ liệu có thể được truy cập từ nhiều nguồn dữ liệu khác nhau (ví dụ: JSON, Parquet, Hive Tables và JDBC). Spark SQL cung cấp chức năng thao tác dữ liệu thông qua SQL, HQL hoặc API bộ dữ liệu cấp cao tùy chỉnh và kết hợp SQL với các thao tác dữ liệu do API Spark RDD cung cấp. Khái niệm chính của Spark SQL là Bộ dữ liệu đại diện cho một tập hợp dữ liệu phân tán (các đối tượng có lược đồ được xác định). Bộ dữ liệu mang đến sự tiện lợi của RDD với các tối ưu hóa hiệu suất bổ sung được thực hiện bởi Spark SQL(RDD cộng thêm schema). Tập dữ liệu có thể được tổ chức thành các cột được đặt tên tạo thành một Data Frame, đơn giản là một Tập dữ liệu kiểu Hàng. DataFrame tương đương với bảng cơ sở dữ liệu (lược đồ có các hàng xuất phát từ truy vấn mà DataFrame mô tả). Lưu ý có 2 dạng bảng chính trong Spark SQL đó là Managed Table và External Table.

#### **2.3.3.3 Spark Streaming**

Spark streaming là một mô-đun dành cho các ứng dụng cần xử lý dữ liệu khi chúng xuất hiện, nó cung cấp các luồng rời rạc trừu tượng hóa cấp cao (DStream) của riêng nó để đơn giản hóa hoạt động với luồng dữ liệu liên tục. Nó xử lý dữ liệu theo micro batch, tùy thuộc vào khoảng thời gian phát trực tuyến đã định cấu hình, trong đó mỗi batch nội bộ là một RDD dữ liệu được xử lý trong một khoảng thời gian đã đặt kể từ batch cuối cùng. Nói cách khác, DStream bên trong là một chuỗi RDD liên tục trong đó mỗi RDD biểu thị dữ liệu từ một khoảng thời gian nhất định.



**Hình 2.8: Luồng hoạt động của Spark Streaming**

DStream có thể được chuyển đổi bằng cách sử dụng các phép biến đổi tương tự như API RDD, chẳng hạn như bản đồ (func) và bộ lọc (func). DStreams cũng hỗ trợ windowed computation bổ sung liên quan đến thời gian.

Spark streaming có thể tích hợp với nhiều nguồn dữ liệu thời gian thực khác nhau (ví dụ: Apache Flume, Apache Kafka và HDFS) và hỗ trợ điểm kiểm tra, một cơ chế chịu lỗi hoạt động bằng cách lưu trữ đủ lượng thông tin trong bộ lưu trữ đáng tin cậy để phục vụ cho việc khôi phục trong trường hợp thất bại.

* Window Operations

Trong xử lý luồng (stream processing), mặc định chúng ta chỉ có thể thao tác với các tập dữ liệu chỉ chứa dữ liệu trong một khoảng thời gian nhất định (khoảng thời gian xử lý hàng loạt đã thiết lập). Nếu chúng ta xử lý DStream bằng cách sử dụng hàm foreachRDD(func) mà áp dụng hàm được truyền vào cho mỗi RDD được tạo ra bởi DStream của chúng thì không có thông tin về dữ liệu đã được xử lý ở các RDD trước đó.

Bằng cách sử dụng các phép toán cửa sổ (window operations), chúng ta có thể truy cập vào nhiều tập dữ liệu cùng một lúc, số lượng tập dữ liệu mà chúng ta có thể truy cập mỗi lần được đặt bởi tham số độ dài cửa sổ và số lượng tập dữ liệu chúng ta bỏ qua sau khi một cửa sổ được xử lý được đặt bởi tham số khoảng trượt. Điều này cho phép chúng ta xử lý dữ liệu trong một khoảng thời gian là bội số của khoảng thời gian xử lý hàng loạt của chúng ta.

#### **2.3.3.4 Spark MLlib**

MLlib là thư viện học máy có khả năng mở rộng, được tích hợp trong Apache Spark, bao gồm các chức năng học máy. MLlib chứa các thuật toán học máy phổ biến như:

* Classification: logistic regression, naive Bayes, v.v...
* Regression: generalized linear regression, survival regression, v.v...
* Decision trees, random forests, and gradient-boosted trees
* Recommendation: alternating least squares (ALS), v.v...
* Clustering: K-means, Gaussian mixtures (GMMs) v.v...

Ý tưởng là sử dụng MLlib để có thể thử nghiệm các thuật toán ML trong bối cảnh Dữ liệu lớn, tập trung hơn hết vào tiền xử lý dữ liệu thay vì nghiên cứu và tối ưu hóa chính các thuật toán. Vì các giá trị đích được dự đoán là các biến nhị phân nên một bộ phân loại nhị phân đã được chọn: Vector hỗ trợ (Support Vector Classifier).

#### **2.3.3.5 Spark GraphX**

GraphX là một thành phần dành cho việc xử lý đồ thị và tính toán song song trên đồ thị. GraphX mở rộng API RDD của Spark và sử dụng trừu tượng đồ thị đa hướng với các thuộc tính trên mỗi đỉnh và cạnh để biểu diễn dữ liệu. GraphX cung cấp các toán tử cấu trúc và kết nối (như sub-graph, joinVertices) để thao tác đồ thị và bao gồm các thuật toán đồ thị phổ biến (như PageRank quá nổi tiếng của Google, thành phần liên thông và triangle counting) để làm cho phân tích dữ liệu dễ dàng hơn.

### **2.3.4 Kiến trúc Apache Spark**



**Hình 2.9: Cơ chế hoạt động của Spark**

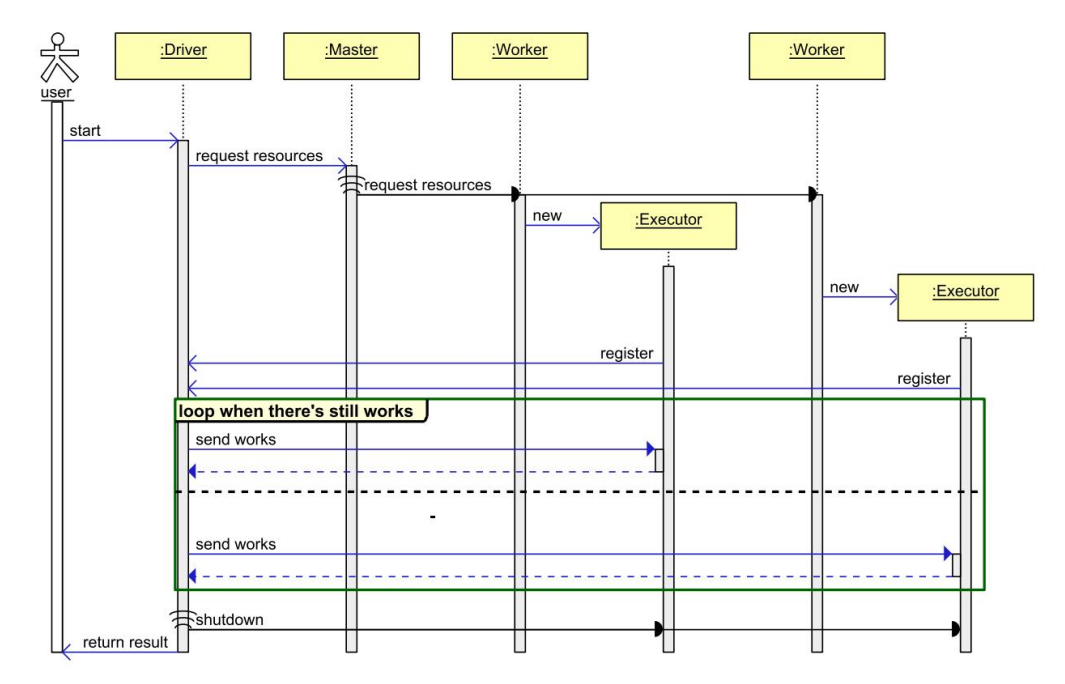
Apache Spark tuân theo kiến ​​trúc chủ/nô lệ có một nút chủ và nhiều nút công nhân phục vụ quanh nó.

#### **2.3.4.1 Master**

Nút Master được kết nối với tất cả các nút công nhân để quản lý tài nguyên của cụm và cung cấp cổng duy nhất vào các cụm. Khi chạy, chương trình điều khiển phải liên hệ với nút Master để yêu cầu CPU và bộ nhớ cần thiết. Master sẽ lần lượt phân bổ tài nguyên cho Worker dưới dạng các trình thực thi (executor process) và kết nối các trình thực thi này với trình điều khiển (driver)

#### **2.3.4.2 Worker**

Các nút Worker là nơi diễn ra quá trình tính toán thực tế. Sau khi khởi động, mỗi công nhân cố gắng kết nối với máy chủ và đăng ký số lõi CPU và dung lượng RAM có sẵn trên máy. Sau đó, cứ sau 15 giây, người công nhân sẽ gửi nhắn theo từng đợt để thông báo cho nút Master rằng nó vẫn ổn.



**Hình 2.10: Phân bổ tài nguyên trong Spark**

Khi nhận được yêu cầu tài nguyên từ nút Master, nút Worker sẽ sinh ra các quy trình thực thi(executors) mới theo yêu cầu. Các bộ thực thi không được sử dụng lại hoặc chia sẽ giữa các ứng dụng: mỗi chương trình sẽ có quy trình thực thi riêng ưu điểm các tác vụ được thực thi đa luồng (multithread) và thực thi song song có thể tùy chỉnh cấp phép. Các tác vụ của các ứng dụng khác nhau được chạy trong các JVM (Java Virtual Machine) khác nhau, giúp tách biệt việc lập lịch và thực thi các ứng dụng. Sau khi tạo thành, trình thực thi sẽ được nhân viên giám sát và sẽ kết thúc khi nút Master yêu cầu. Điều này thường xảy ra khi các ứng dụng hoàn tất, hoặc là một số lỗi không thể phục hồi trong cụm.

Ngoài các quy trình thực thi, các nút công nhân cũng có thể chứa trình điều khiển nếu cần. Đây là trường hợp khi các nhà phát triển muốn sử dụng ở chế đó triển khai “cluster”. Ở chế độ này, nút Master sẽ chọn một nút Worker ngẫu nhiên trong cụm để chạy trình điều khiển.

#### **2.3.4.3 Driver**

Trình điều khiển là bộ não của ứng dụng Spark. Nó có thể được chạy từ bên ngoài cụm bằng máy khác ở chế độ deployment mode hoặc từ trong cụm bằng chế độ cluster employment mode. Nó quản lý cách dữ liệu được phân phối và xử lý trên các trình thực thi khác nhau. Vì trình điều khiển cần liên lạc liên tục với trình thực thi trên nút Worker, lý tưởng nhất nên triển khai trên máy có kết nối mạng nhanh đến cụm.

Để thực hiện công việc của mình, trình điều khiển chứa nhiều thành phần như sau:

* The RPC environment: Môi trường, trình điều khiển chứa nhiều thành phần như sau: kết nối mạng nhanh đến
* The scheduler: Bent: Môi trường, trình điều khiển chứa nhiều thành phần như sau: kết nối mạng nhanh đến cụm của bộ lập lịch bao gồm DAGScheduler, TaskScheduler, TaskSetManager và phần phụ trợ của trình lập lịch.
* The Map’s output tracker: Trình theo dõi đu khiển chứa nhiều thành phần như sau: kết nối mạng nhanh đến cụm.của bộ lập lịch bao gồm
* The block manager: Trình quản lý khối của trình điều khiển chứa thông tin về khối dữ liệu là gì và ở đâu được lưu trữ trên toàn bộ cụm.

#### **2.3.4.4 Executor**

Trình thực thi là phần cơ bản của ứng dụng Spark: nó nhận các tác vụ từ trình điều khiển, chạy các tác vụ đó và báo cáo kết quả và số liệu thống kê cho trình điều khiển. Trình thực thi được sinh ra trong nút công nhân theo yêu cầu từ bản gốc và thường tồn tại trong toàn bộ ứng dụng trừ khi chúng bị lỗi hoặc bị loại bỏ sớm. Nếu một trình thực thi gặp sự cố ngay lập tức một trình thực thi khác sẽ ngay lập tức được tạo ra để thay thế nó. Khi trình thực thi khởi động, nó sẽ liên hệ trực tiếp với trình điều khiển để đăng ký số lõi CPU trống mà nó có và chờ tác vụ cụ thể. Trong suốt vòng đời của nó, trình thực thi sẽ được trình điều khiển yêu cầu rất nhiều nhiệm vụ.

Các thành phần chính của trình thực thi là:

* The RPC environment: Môi trường này được sử dụng để liên lạc với các trình điều khiển để đăng ký tài nguyên, chấp nhận nhiệm vụ từ trình điều khiển và cập nhật trạng thái của chúng trở lại trình điều khiển.
* The scheduler: Được sử dụng để chạy một tác vụ bên trong nhóm luồng và thu thập số liệu thống kê của nó.
* The Map’s output tracker: Trình theo dõi đầu ra của Bản đồ của trình thực thi lưu trữ một bản sao được lưu trong bộ nhớ cache của trình điều khiển.
* The block manager: Trình quản lý khối của trình thực thi có thông tin về vị trí của dữ liệu các khối được lưu trữ trong công nhân đó.

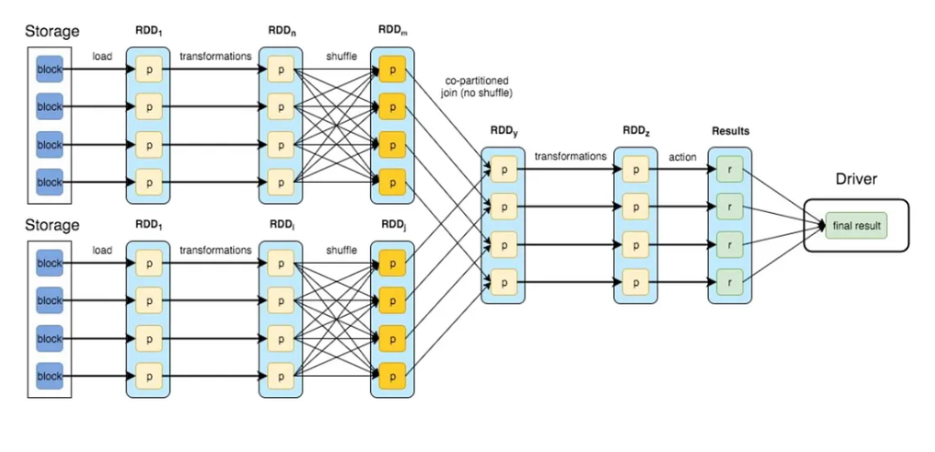
#### **2.3.4.5 Thành phần của Spark**

Phần này sẽ thảo luận về một số thành phần chính của trình điều khiển và bộ thực thi của Spark mà chúng tôi đã trình bày ở phần trước:

Từ chương trình điều khiển do người lập trình viết, Spark tạo ra một kế hoạch thực hiện gồm nhiều công việc (jobs), một công việc cho mỗi action được thực hiện. Một công việc được tạo bằng cách nhóm tất cả các phép biến đổi (transformations) được thực hiện từ đầu chương trình cho đến action kết thúc. Công việc sau đó được gửi tới DAGScheduler để thực thi.

1. **Bộ lập lịch DAGScheduler**

DAGScheduler là bộ lập lịch cấp cao của Spark, thực hiện lập lịch theo từng giai đoạn. Nó tính toán DAG (đồ thị chu kỳ có hướng) của các giai đoạn cho từng công việc và tìm ra lịch trình tối ưu nhất để chạy công việc. Lịch trình tối ưu này liên quan đến đầu ra của các giai đoạn RDD từ các công việc trước đó để tránh việc tính toán lại không cần thiết. Để tạo ra một DAG của các giai đoạn, Spark phá vỡ đồ thị phụ thuộc RDD tại các ranh giới shuffle.



**Hình 2.11: Một quy trình thực thi nhiệm vụ của Spark**

Khi DAG được tạo ra, Spark sẽ bắt đầu thực thi các giai đoạn theo thứ tự topological. Ví dụ, với chương trình trong hình 2.12, có 2 giai đoạn 1 và 2 luôn thực hiện trước giai đoạn 3. Tuy nhiên, giai đoạn 1 và giai đoạn 2 có thể được thực thi đồng thời hoặc tuần tự tùy thuộc vào khả năng tài nguyên của cụm, vì không có sự phụ thuộc giữa chúng. Việc thực thi các giai đoạn được thực hiện bằng cách tạo một TaskSet cho mỗi giai đoạn sắp chạy và gửi TaskSet đó đến TaskScheduler cơ bản. Mỗi TaskSet bao gồm nhiều tác vụ độc lập có thể được chạy trên bất kỳ executor nào tại các nút worker. Vị trí ưu tiên để khởi chạy mỗi tác vụ được tính toán dựa trên lượng dữ liệu đầu vào tại mỗi executor: các executor có nhiều dữ liệu đầu vào cho tác vụ sẽ được ưu tiên cao hơn. Chiến lược này được chọn để giảm thiểu lượng dữ liệu chuyển qua mạng. Vì kích thước của một tác vụ thường nhỏ hơn nhiều so với dữ liệu nên việc gửi tác vụ đến trình thực thi thường nhanh hơn nhiều so với việc tìm nạp dữ liệu từ xa.

Công việc cuối cùng của DAGScheduler là xử lý các lỗi do mất các tệp đầu ra của map, trong trường hợp này các giai đoạn cũ cần phải được chạy lại để tái tạo những tệp đó. Vì DAGScheduler biết những tệp đầu ra của map nào vẫn còn sử dụng được hay khả dụng, nên các giai đoạn chạy lại không cần phải được thực thi hoàn toàn: chỉ những tác vụ tạo ra đầu ra bị mất mới cần thiết phải thực hiện lại.

1. **Bộ lập lịch tác vụ (TaskScheduler)**

Bộ lập lịch tác vụ là trình lập lịch ở cấp thấp của ứng dụng Spark, nhận các tác vụ từ DAGScheduler và phân phối chúng tới các executors. Nó cũng giám sát các executors để theo dõi tiến trình của các tác vụ đang chạy, thông qua tín hiệu heartbeat gửi đều đặn mỗi 10 giây.

Trong Spark, tác vụ là đơn vị thực thi nhỏ nhất nó hoàn toàn độc lập, được xác định bởi ID theo giai đoạn, ID lần thử giai đoạn và ID phân vùng. Mỗi tác vụ đại diện cho việc tính toán của một phân vùng dữ liệu để thực thi một giai đoạn trong công việc Spark. Tất cả các tác vụ có cùng ID giai đoạn và ID lần thử giai đoạn thuộc về cùng một TaskSet, do DAGScheduler tạo và gửi tới TaskScheduler.

TaskScheduler quản lý các TaskSet độc lập bằng các TaskSetManager riêng biệt. Khi nhiều TaskSet được gửi cùng lúc, TaskScheduler sẽ cố gắng chạy chúng theo thứ tự xác định bởi chính sách lập lịch, mặc định là dựa trên ID giai đoạn và độ ưu tiên của TaskSet.

TaskSetManager theo dõi từng tác vụ và khớp chúng với các executors ưu tiên bằng kỹ thuật delay scheduling. Có 4 mức độ locality, từ cao đến thấp:

* PROCESS LOCAL: Tác vụ chạy trong cùng một tiến trình executor với dữ liệu đầu vào.
* NODE LOCAL: Tác vụ chạy trên cùng một nút nhưng khác tiến trình executor với dữ liệu đầu vào.
* RACK LOCAL: Tác vụ chạy trên nút khác nhưng cùng rack.
* ANY: Tác vụ chạy trên bất kỳ executor nào còn trống.

Mọi tác vụ bắt đầu với mức ưu tiên cao nhất (PROCESS LOCAL hoặc NODE LOCAL) và chờ một thời gian ngắn để khớp với executor phù hợp. Nếu không khớp, mức độ ưu tiên sẽ giảm dần để có thêm nhiều executor phù hợp hơn. TaskSetManager cũng giám sát các lỗi của tác vụ và thử lại tối đa 3 lần nếu có thể. Trong trường hợp lỗi không thể phục hồi, TaskSetManager sẽ tự dọn dẹp và thoát để DAGScheduler khởi động lại toàn bộ giai đoạn.

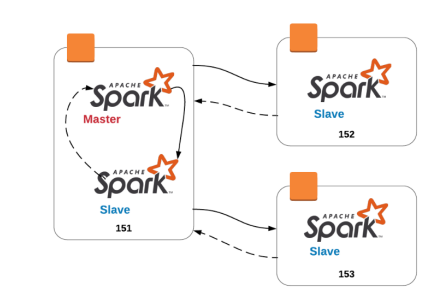
1. **The CoarseGrainedSchedulerBackend**

Để hỗ trợ nhiều loại trình quản lý cụm như Apache Mesos, Apache YARN, K8S hoặc trình quản lý cụm Standalone của Spark, Spark có một cơ chế backend riêng có thể cắm thêm được, cho phép ánh xạ tài nguyên của cụm tới các executors của Spark một cách linh hoạt đây là cơ chế phổ biến nhất và tích hợp tốt nhất với Spark. CoarseGrainedSchedulerBackend hoạt động như kho lưu trữ executors cho một ứng dụng Spark. Nó theo dõi việc đăng ký, hủy đăng ký của các executors và duy trì bản ghi các executors và CPU còn trống trong cụm. Bộ lập lịch backend này cũng tái sử dụng các executors trong suốt thời gian của một ứng dụng Spark, thay vì khởi chạy executor mới cho mỗi tác vụ. Các chế độ chạy cụm của Spark như sau:

* Standalone Scheduler - Trình điều khiển được tích hợp của Spark

Đây là một cách triển khai cục bộ cho Spark, nơi một máy tính hoặc một số máy tính kết hợp thành một cụm Spark riêng.

Scheduler (lập lịch) được tích hợp trong chính cụm Spark này và quản lý việc phân phối công việc giữa các máy tính thành viên. Trong những dự án lớn cần cấu hình nhiều node thì việc sử dụng Standalone Scheduler không được khuyến khích theo như chính đề xuất của Spark mà phải sử dụng các cơ chế khác như YARN, MESOS hoặc ưu tiên nhất là KUBERNETES .

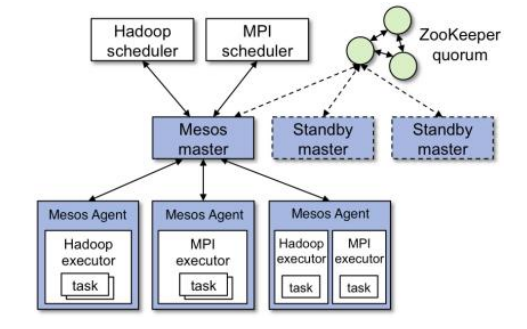


**Hình 2.12: Spark trong chế độ Standalone Cluster Manager**

* MESOS - trình điều khiển sẽ sinh ra các trình thực thi trên cụm Mesos (chế độ triển khai: client | cluster)

MESOS là một hệ thống phân phối tài nguyên mã nguồn mở và đa dạng hóa, giúp quản lý tài nguyên từ các máy chủ trong một cụm.

Spark cũng có khả năng tích hợp với MESOS, cho phép nó sử dụng các tài nguyên trên cụm được quản lý bởi MESOS và lên lịch thực thi công việc theo các yêu cầu tài nguyên của ứng dụng Spark.

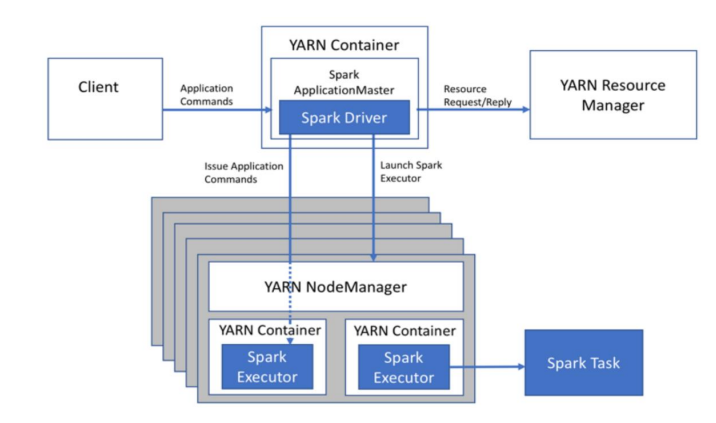


**Hình 2.13: Kiến trúc của Apache Mesos**

* YARN - ý tưởng tương tự như với Mesos (chế độ triển khai: client | cluster)

YARN (Yet Another Resource Negotiator) là một cơ chế phân phối tài nguyên trong hệ thống Hadoop.

Spark có thể tích hợp với YARN để sử dụng các nguồn tài nguyên của Hadoop và lên lịch thực thi các công việc trên cụm Hadoop sử dụng cả xác định dựa trên các yêu cầu tài nguyên của ứng dụng.



**Hình 2.14: Spark trong chế độ hoạt động với YARN**

* KUBERNETES (K8S) - trình quản lý container phân vùng tài nguyên động.

K8S là một hệ thống mã nguồn mở để quản lý và triển khai các ứng dụng container, được phát triển bởi Google.

Spark có thể tích hợp với K8S thông qua K8S Scheduler Backend để triển khai các ứng dụng Spark trên cụm K8S.

1. **The Map’s Output Tracker**

MapOutputTracker là thành phần theo dõi vị trí của các kết quả shuffle được tạo ra bởi một giai đoạn. Đối với mỗi giai đoạn, kết quả map được chia thành các bucket, mỗi bucket được tạo ra bởi một tác vụ và tương ứng với một phân vùng trong RDD đầu vào. Có hai phiên bản của MapOutputTracker:

* MapOutputTrackerMaster: Chạy trên driver và chứa vị trí của mọi bucket đầu ra map cho tất cả các shuffle trong cụm.
* MapOutputTrackerWorker: Chạy trên các executor và lưu trữ tạm thời vị trí đầu ra map để tránh việc liên hệ với driver mỗi khi một tác vụ mới bắt đầu trên các executor. Nếu không tìm thấy trong bộ nhớ tạm, nó sẽ yêu cầu lại vị trí đầu ra map từ MapOutputTrackerMaster.

Như được chỉ ra trong hình 2.16, mỗi khi một tác vụ hoàn thành xử lý, nó phải liên hệ với driver, nơi sẽ đăng ký vị trí đầu ra map của tác vụ đó trong MapOutputTracker. Vị trí đầu ra map được xác định bởi:

* ID của shuffle: mỗi phụ thuộc rộng giữa hai giai đoạn được gán một định danh duy nhất, cũng đóng vai trò như ID của shuffle.
* ID của phân vùng dữ liệu mà tác vụ tính toán.
* ID của block manager đang chạy trên executor đã xử lý tác vụ.

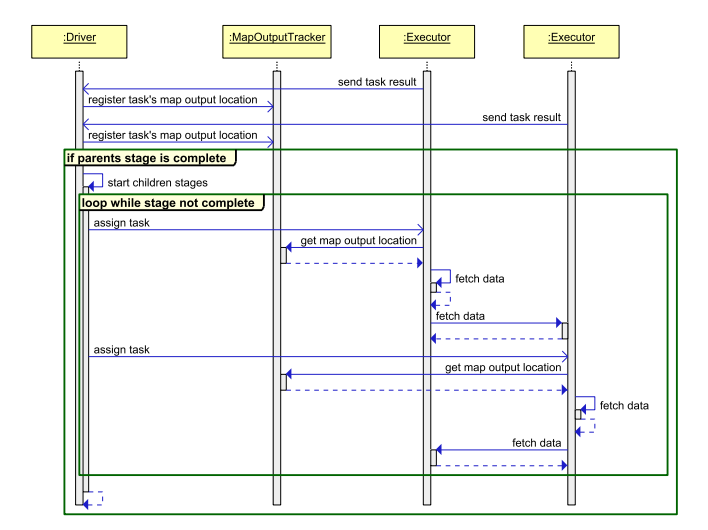
Sau đó, khi các tác vụ của giai đoạn con được khởi chạy, chúng sẽ liên hệ với MapOutputTracker để lấy vị trí đầu ra map của các giai đoạn cha để có thể lấy và xử lý dữ liệu này.

1. **The Block Manager**

Block Manager là kho lưu trữ dạng key-value chạy trên driver và các executors của một ứng dụng Spark. Nó cung cấp giao diện thống nhất để lưu trữ và truy xuất các khối dữ liệu cả tại chỗ (local) và từ xa (remote), từ bộ nhớ trong (in-memory) cũng như từ ổ đĩa (on-disk).

Khi một tác vụ hoàn thành, đầu ra của nó được sắp xếp theo khóa của mỗi bản ghi và lưu vào đĩa cục bộ dưới dạng tệp dữ liệu shuffle và tệp chỉ mục. Tệp chỉ mục này chứa ánh xạ từ ID phân vùng của RDD con tới vùng của tệp dữ liệu chứa dữ liệu cho ID đó. Vị trí của các tệp này sau đó được lưu vào block manager của executor. Chỉ khi quá trình này hoàn tất, tác vụ mới có thể báo cáo lại cho driver rằng nó đã được thực thi thành công.

Sau khi mỗi tác vụ của RDD con lấy vị trí đầu ra map từ driver, nó liên hệ với các block managers được chỉ định trong các vị trí đó để truy xuất dữ liệu thuộc về phân vùng của nó. Đối với dữ liệu đã tồn tại trong block manager cục bộ, nó được đọc trực tiếp từ tệp dữ liệu shuffle và tệp chỉ mục. Ngược lại, dữ liệu cần tải từ các executors từ xa sẽ được đệm trong bộ nhớ cho đến khi bộ đệm đầy. Kích thước mặc định của bộ đệm là 48 MB. Khi dữ liệu từ bộ đệm này được tiêu thụ, dữ liệu mới sẽ được tải vào bộ đệm cho đến khi mọi thứ được xử lý.



**Hình 2.15: Quy trình theo dõi đầu ra bản đồ**

#### **2.3.4.6 Khả năng chịu lỗi của Spark**

Như đã trình bày ở phần 2.3.2.1, Spark cung cấp khả năng tính toán chịu lỗi bằng cách tính toán lại các RDD dựa trên dòng dõi (lineage) của chúng. Tuy nhiên, việc truy vết toàn bộ lineage từ phép biến đổi đầu tiên có thể mất nhiều thời gian, đặc biệt khi có liên quan đến data-shuffling. Do đó, Spark tận dụng các tệp shuffle trên đĩa như một cơ chế checkpoint. Vì các tệp này không bị xóa cho đến khi chương trình kết thúc, nên khi một tác vụ trong giai đoạn cần khởi động lại do lỗi thực thi, nó có thể đọc lại các tệp shuffle đầu vào để tính toán lại phân vùng tương ứng. Việc thử lại này được thực hiện tự động tối đa ba lần bởi TaskScheduler mà không cần thay đổi bất kỳ thành phần nào.

Tuy nhiên, trong trường hợp lỗi nghiêm trọng như crash hoặc mất kết nối của một tiến trình executor, dữ liệu shuffle trên đĩa sẽ không còn khả dụng. Lỗi này sẽ kích hoạt FetchFailedException khi các tác vụ đang chạy cố gắng tải đầu vào shuffle của chúng. Trong trường hợp này, runtime không thể chỉ khởi động lại các tác vụ gây ra lỗi, mà phải tính toán lại dữ liệu shuffle bị mất. Để làm điều này, thiết kế hiện tại yêu cầu hủy giai đoạn lỗi và khởi động lại các giai đoạn cha của nó. Vì chỉ có hai lớp phụ thuộc: giai đoạn cha và giai đoạn con, khi chúng ta hủy các giai đoạn con, toàn bộ cụm sẽ được giải phóng để chạy các giai đoạn cha.

Ngược lại, khi loại bỏ rào cản giai đoạn bằng chính sách truyền dữ liệu theo pipeline, chúng ta có thể có ba hoặc nhiều lớp phụ thuộc hơn: giai đoạn cha (S1), giai đoạn con (S2) và giai đoạn cháu (S3). Khi S2 gặp FetchFailedException, nó sẽ bị hủy và S1 sẽ được khởi động lại. Tuy nhiên, do sự tồn tại của S3, có thể xảy ra deadlock. Khi S2 bị hủy, tất cả các slot CPU của nó được giải phóng để runtime có thể dùng chúng để chạy các tác vụ của S1. Tuy nhiên, runtime cũng có thể dùng các slot CPU này để chạy các tác vụ của S3. Deadlock xảy ra khi tất cả các slot CPU được phân bổ cho S3 thay vì S1. Vì S2 chưa hoàn thành, S3 sẽ bị kẹt chờ thêm dữ liệu từ S2. Tuy nhiên, S2 cần kết quả của S1, không thể thực thi do thiếu slot CPU.

Deadlock này có khả năng xảy ra vì có một khoảng thời gian chờ giữa lúc S2 bị hủy và S1 được lập lịch lại. Để tránh deadlock này, khi S2 gặp lỗi FetchFailedException và phải hủy, chúng ta cũng hủy S3. Hành động này giải phóng tài nguyên cụm để khởi động lại S1. Tuy nhiên, điều này cũng có nghĩa là thời gian CPU đã sử dụng cho S3 bị lãng phí. Không giống như S2, S3 không có tác vụ nào hoàn thành trước lỗi và phải khởi động lại hoàn toàn. Tuy nhiên, điều này có thể tránh được với một scheduler thông minh hơn. Vì chỉ các phần lỗi của S1 và S2 cần tính toán lại, không cần phải hủy hoàn toàn S3. Chúng ta chỉ cần hủy một số tác vụ nhỏ trong S3 để có đủ slot CPU khởi động lại S1 và S2. Đáng tiếc, điều này khó thực hiện với kiến trúc scheduler hiện tại của Spark, vì vậy hủy hoàn toàn S3 là lựa chọn tốt nhất của chúng ta.

## **2.4 Cơ sở lý luận**

Thuật ngữ dữ liệu lớn (Big Data) hiện nay vẫn chưa có câu trả lời chính xác cụ thể cho câu hỏi dữ liệu thế nào gọi là lớn đối với một số tổ chức, đây có thể là hàng chục terabyte dữ liệu. Đối với những tổ chức khác nó có thể lên tới hàng trăm petabyte thậm chí hàng nghìn petabyte dữ liệu. Câu trả lời còn tùy vào dữ liệu mà chúng ta đang xử lý và mức độ quan trọng của tập dữ liệu đó ngay tại thời điểm xử lý. Và có những tập dữ liệu tuy “nhỏ” nhưng lại “lớn”, và trái lại tuy “lớn” nhưng mà rất “nhỏ”. Có thể nói Big Data là một khái niệm tương đối theo nhiều mặt và không thể ràng buộc ở một con số nhất định cụ thể. Để dữ liệu có thể được coi là Big Data thì phải đáp ứng đủ ba yếu tố hay ba đặc điểm chính của dữ liệu lớn.

Big Data đã xuất hiện rất nhiều đặc điểm để có thể nhận dạng được rằng liệu dữ liệu đó có phải nằm trong thuật ngữ Big Data hay không. Ba đặc điểm chính của Big Data hay còn được gọi là 3V để xác định rằng dữ liệu đó có phải là dữ liệu lớn hay không, 3V bao gồm: Dung lượng (volume), Tốc độ (velocity), Tính đa dạng (variety). Ngoài ra còn có nhiều chữ V khác và hiện nay thuật ngữ Big Data đang được đánh giá trên tiêu chí 5V nhưng trong đó cốt lõi vẫn đáp ứng đủ 3 tiêu chí trên thì chúng ta có thể coi đó chính là Big Data. Trong tiêu chí 5V các chữ V đó có các tính chất như sau: (1) Volume – dung lượng, (2) Variety - đa dạng, (3) Velocity – tốc độ, (4) Veracity – tính xác thực, và (5) Value - giá trị.

• Volume - Dung lượng dữ liệu do hệ thống ITS tạo ra tăng theo cấp số nhân. Với số lượng ngày càng tăng của các ứng dụng công nghệ giám sát, IoT, Apps cho phép việc thu thập dữ liệu ngày càng đa dạng và phức tạp. Vì thế lượng dữ liệu liên quan đến tất cả mọi hoạt động xung quanh được tạo ra mỗi giây và lớn dần theo cấp số mũ là một điều không thể tránh khỏi.

• Variety - Dữ liệu được thu thập ở nhiều định dạng và theo nhiều phương cách khác nhau. Mức độ định dạng của dữ liệu này cũng có thể thay đổi từ dữ liệu bán cấu trúc (ví dụ: nhật ký sửa chữa, hình ảnh, video và tệp âm thanh) đến dữ liệu có cấu trúc (ví dụ: dữ liệu từ hệ thống cảm biến, dữ liệu sự cố giao thông, v.v…). Các bộ dữ liệu khác nhau có các định dạng khác nhau về kích thước tệp, độ dài bản ghi và lược đồ mã hóa; nội dung của chúng có thể đồng nhất hoặc không đồng nhất. Các tập dữ liệu không đồng nhất này, được tạo ra bởi các nguồn khác nhau ở các định dạng khác nhau, đã đặt ra những thách thức đáng kể cho việc tích hợp và phân tích.

• Velocity – Sự đa dạng nguồn thu thập dữ liệu và yêu cầu giám sát thực tiễn về giám sát, kinh doanh, xây dựng, sức khỏe, v.v. đã kéo theo tốc độ tạo ra dữ liệu lớn. Trong đó nhiều dữ liệu được thu thập liên tục, theo thời gian thực, trong khi các dữ liệu khác được thu thập định kỳ. Ví dụ các dữ liệu cảm biến lắp đặt trên phương tiện giao thông hay dữ liệu giám sát camera sẽ được thu thập thời gian thực với tần suất có thể tính 19 theo đơn vị giây. Tuy nhiên các dữ liệu về bản đồ có thể được cập nhật định kỳ trong khoảng thời gian dài hơn.

• Veracity - Được sử dụng để mô tả tính chắc chắn hoặc độ tin cậy của dữ liệu ITS. Ví dụ: bất kỳ quyết định nào được đưa ra từ một dữ liệu cảnh báo nào đó phải được phân tích dựa trên tính toàn vẹn của nguồn dữ liệu, nghĩa là, phải có các hiệu chuẩn chính xác dữ liệu truyền về từ cảm biến và có những giải thích chính xác về bất kỳ dữ liệu bị thiếu.

• Value – Đề cập đến vấn đề thời gian thu thập dữ liệu, tốc độ lấy mẫu của dữ liệu. Ví dụ, dữ liệu cũ vài giờ có thể không có giá trị đối với ứng dụng phân tích y học, nhưng có thể hữu ích trong ứng dụng chẩn đoán ngắn. Như vậy hầu hết dữ liệu y học được thu thập đều có giá trị phục vụ cho bài toán phân tích đa chiều về cơ thể người, dữ liệu càng được làm mới, lưu trữ trong khoảng thời gian càng dài thì càng có giá trị.

## **2.5. Lý thuyết về các kiến trúc và thuật ngữ**

Cơ sở lưu trữ và phân tích dữ liệu trung tâm dựa trên các giải pháp về dữ liệu lớn. Truy xuất dữ liệu lớn không chỉ đơn giản là sao chép, mà còn là một quy trình phức tạp để đảm bảo tính truy xuất dữ liệu, sự trao đổi thông tin và khả năng tái sử dụng của dữ liệu đúng cách và có quy chuẩn nhất định [1]. Thách thức đặt ra cho các giải pháp kết nối là duy trì cấu trúc và tính nhất quán của dữ liệu tại điểm vào của hệ thống lưu trữ.

### **2.5.1. Data Warehouse**

Nhà kho dữ liệu (Data Warehouse) là khái niệm được giới thiệu lần đầu vào năm 1988 bởi 2 nhà nghiên cứu của IBM là Barry Devlin và Paul Murphy. Đây là nơi lưu trữ dữ liệu bằng thiết bị điện tử của một tổ chức, doanh nghiệp, nhằm hỗ trợ việc phân tích dữ liệu và lập báo cáo [3]. Về cơ bản, có thể hiểu, Data Warehouse là một tập hợp dữ liệu hoặc thông tin có chung một chủ đề, được tổng hợp từ nhiều nguồn khác nhau trong nhiều mốc thời gian. Quá trình tập hợp và thao tác trên dữ liệu mang đặc tính ACID:

• Atomicity (Tính nguyên tử): dữ liệu được tập hợp từ nhiều nguồn khác nhau, khi tập hợp phải thực hiện làm sạch, sắp xếp, rút gọn dữ liệu.

• Consistency (Tính nhất quán): chỉ lấy những dữ liệu có ích (các dữ liệu có cùng chủ đề).

• Isolation (Tính cô lập): Các dữ liệu truy xuất không bị ảnh hưởng bởi các dữ liệu khác hoặc tác động lên nhau.

• Durable (Tính bền vững): Dữ liệu không thể tạo thêm, xóa hay sửa đổi.

Dữ liệu trong kho dữ liệu là rất lớn và không có những thao tác như sửa đổi hay tạo mới đây là điểm phân biệt rõ ràng giữa kho dữ liệu với những database thông thường. Kho dữ liệu dựa trên cơ sở mô hình dữ liệu đa chiều, được mô hình hóa vào đối tượng gọi là datacube. Datacube bao gồm các dữ kiện(fact), và những dữ kiện này tạo ra nhiều dữ kiện khác nhau (dimension). Đối với kiến trúc Data Warehouse, dữ liệu có cấu trúc từ các database thông qua quá trình trích xuất, biến đổi sẽ được lưu vào “nhà kho”. Dữ liệu trong nhà kho này sẽ được sử dụng để xuất báo cáo, trực quan cho người sử dụng Mục đích của kiến trúc Data Warehouse là phục vụ các yêu cầu phân tích, hoặc khai phá cụ thể được gọi là chủ đề. Ví dụ chủ đề giao thông có thể bao gồm tổng số phương tiện trong ngày, thống kê số lượng xe nội tỉnh/ngoại tỉnh hoặc mật độ phương tiện qua các nút giao thông.

• Ưu điểm:

– Kiến trúc hệ thống xây dựng theo hướng phân tích thông tin quản trị thông minh BI (Business Intelligence) để có thể trình bày thông tin trên các báo cáo quản trị giúp điều hành doanh nghiệp. Đối tượng thụ hưởng hệ thống là những người phân tích thông tin và đưa ra các kế hoạch dài hạn hoặc điều hành ngắn hạn.

– Là một kiến trúc mạnh về quản lý các chức năng ACID transaction.

– Hiện nay có các Data Warehouse hỗ trợ khoa học dữ liệu và học máy rất tiến bộ như các Data Warehouse trên nền tảng public cloud như Snowflake, Amazon Redshift, IBM DB2, v.v.

• Nhược điểm:

– Không hỗ trợ những liệu có đặc tính phi cấu trúc, bán cấu trúc;

– Hạn chế trong việc đáp ứng dữ liệu biến đổi nhanh về mặt số lượng;

– Chi phí cao cho việc lưu trữ những tập dữ liệu lớn;

Nhà kho dữ liệu (Data Warehouse) giúp các nhà lãnh đạo có được những phân tích chuyên sâu bằng việc thu thập dữ liệu từ những cơ sở dữ liệu vận hành về một nhà kho, nơi mà sau này có thể được sử dụng như một nguồn thông tin hỗ trợ việc ra quyết định và các tác vụ phân tích kinh doanh (Business Intelligence). Dữ liệu trong kho có thể được ghi lại bằng phương pháp schema-on-write, có nghĩa là mẫu dữ liệu được tối ưu hóa cho luồng tiêu thụ. Đây là thế hệ đầu tiên của các nền tảng phân tích dữ liệu Khoảng một thập kỷ trước, hệ thống thế hệ đầu tiên sử dụng Data Warehouse bắt đầu đối mặt với nhiều thách thức mới:

• Thứ nhất đó là chúng kết hợp giữa tính toán và lưu trữ trên cùng một máy. Điều này khiến cho doanh nghiệp càng phải chi nhiều tiền hơn khi dữ liệu phình to ra.

• Thứ hai, ngoài việc dữ liệu phình to về mặt kích thước, chúng còn trở nên đa dạng về mặt chủng loại, và hầu hết là những dữ liệu không cấu trúc. Đặc biệt là những dữ liệu dạng tài liệu văn bản, ảnh, audio… khiến cho Data Warehouse không còn khả năng lưu trữ và truy vấn nữa.

• Thứ ba, với việc dữ liệu thay đổi và phát triển quá nhanh, nhu cầu nhận được giá trị từ dữ liệu một cách nhanh chóng đã trở nên cấp bách hơn bao giờ hết. Hầu như các nhà lãnh đạo không đủ kiên nhẫn cho nhiều tháng phân tích và mô hình hóa dữ liệu cho mục đích sử dụng. Vì nhu cầu phát sinh, hệ thống thế hệ thứ hai đã được nghiên cứu phát triển để giải quyết những khó khăn mà Data Warehouse gặp phải.

### **2.5.2. Data Lake**

Vào thời điểm dữ liệu trong Data Warehouse có thể sử dụng được, nhu cầu về dữ liệu có thể đã thay đổi. Trong một nhánh tương tự như Data Warehouse, các kho dữ liệu cục bộ (data mart) nổi lên với một mục đích sử dụng cụ thể hoặc được phân loại theo một chất lượng nhất định. Data mart đã thành công hơn vì việc sử dụng dữ liệu được hiểu rõ hơn, và kết quả có thể được cung cấp nhanh hơn. Tuy nhiên, tính chất ngăn cách của các Data Mart đã khiến chúng trở nên ít hữu ích hơn đối với các bài toán có lượng dữ liệu khổng lồ, và cần sử dụng dữ liệu đó một cách đa chức năng.

Vì lý do này, các hồ dữ liệu (Data Lake) đã phát triển do khả năng đáp ứng nhu cầu dữ liệu ở mọi quy mô. Chúng có thể tăng tốc mọi thứ, làm cho dữ liệu dễ sử dụng hơn cho các nhu cầu chưa được xác định trước đó. Sự xuất hiện của điện toán đám mây quy mô lớn với sức mạnh tính toán khổng lồ và khả năng lưu trữ gần như vô hạn đã khiến phương pháp tiếp cận hồ dữ liệu này trở nên khả thi.

Mục đích cuối cùng của hồ dữ liệu là để giải quyết sự thất vọng mà hệ thống công nghệ thông tin gặp phải khi cố gắng tạo ra một kho lưu trữ dữ liệu chiến lược có tổ chức, để đưa ra những quyết định kinh doanh quan trọng. Việc sử dụng này có thể bao gồm từ việc phân tích dữ liệu để hiểu rõ hơn nhu cầu của người sử dụng, cho đến trí tuệ nhân tạo để giải quyết các thách thức trong thời gian thực

• Ưu điểm:

– Chi phí lưu trữ thấp để có thể lưu trữ tất cả những dữ liệu thô;

– Dữ liệu có thể đọc trực tiếp từ các thư viện học máy của Apache Spark nhờ các định dạng mở Apache Parquet hay Apache ORC.

• Nhược điểm:

– Nhiều phân hệ lưu trữ phát sinh khiến hệ thống trở nên phức tạp, gây khó khăn trong việc quản lý dữ liệu;

– Phát sinh nhiều tác vụ ETL gây khó khăn trong việc quản lý lỗi;

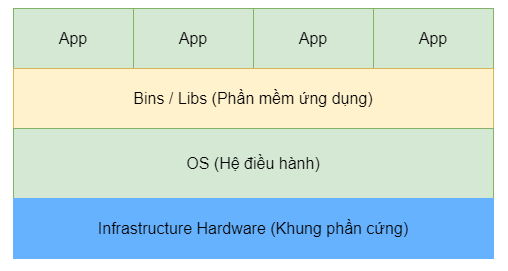
– Khó khăn trong việc đồng bộ dữ liệu giữa các tác vụ ETL và ELT.

Để giải quyết vấn đề của Data Warehouse, thế hệ hệ thống thứ hai đã ra đời, tải toàn bộ dữ liệu thô vào trong một hồ dữ liệu (Data Lake) với chi phí lưu trữ thấp, các API lưu trữ dữ liệu dưới định dạng mở như Apache Parquet, ORC. Cách tiếp cận này khởi đầu bằng sự chuyển dịch của hệ sinh thái Apache Hadoop, sử dụng HDFS để tối thiểu chi phí lưu trữ. Các hồ dữ liệu sử dụng kiến trúc schema-on-read cho phép sự linh động trong lưu trữ dữ liệu với chi phí thấp, nhưng bù lại đó là chất lượng dữ liệu và sự quản lý luồng lưu trữ. Trong kiến trúc này, một phân hệ của dữ liệu sẽ thông qua quá trình ETL và lưu trữ xuống nhà kho dữ liệu, sử dụng cho việc hỗ trợ ra quyết định và các ứng dụng BI. Việc sử dụng các định dạng mở như Parquet, ORC cũng mở ra một phương hướng mới cho các cơ chế phân tích khác, điển hình là các hệ thống học máy.

Từ năm 2018 trở đi, các hồ dữ liệu đám mây như Amazon S3 bắt đầu thay thế dần HDFS. Cùng với sự phát triển của rất nhiều nền tảng lưu trữ kho dữ liệu mới như Amazon Redshift. Nhưng nhìn chung HDFS(Data Lake), Apache Hive(được xem như Data Warehouse của HDFS) cũng là nền tảng căn bản của các kiến trúc lưu trữ dữ liệu lớn hiện giờ nếu không kể đến các dịch vụ đám mây. Hiện nay cũng đang nổi lên một phong trào kết hợp cả hai khái niệm kho chứa trên thành một khái niệm chung đó chính là Lakehouse và nó cũng có khả năng kết hợp với các dịch vụ đám mây một cách nhanh chóng đồng thời mang hết tất cả ưu điểm của Lake và Warehouse, nhưng vẫn đang dừng lại ở mức lý thuyết và chưa có chứng minh thực tiễn nổi trội nào.

# **CHƯƠNG 3: TRIỂN KHAI ỨNG DỤNG**

## **3.1 Lịch sử triển khai ứng dụng.**



**Hình 3.1: Kiến trúc triển khai ứng dụng cổ điển**

Ở thời điểm đầu việc triển khai ứng dụng cổ điển được thực hiện trên một phần cứng cố định một hệ điều hành và nhiều lớp ứng dụng trên đó, phải trải qua rất nhiều năm khi công nghệ đã phát triển thì việc triển khai này đã được thay đổi và tối ưu hơn.

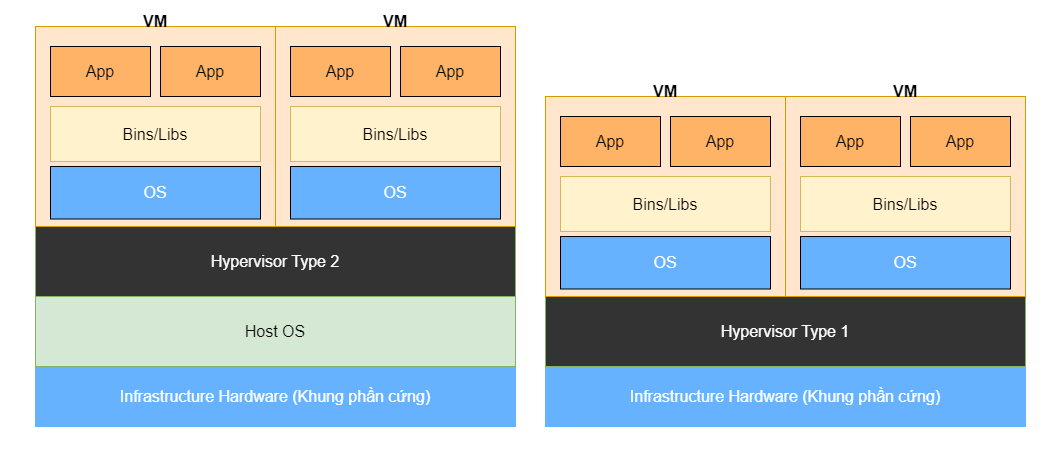
Trong đó lịch sử máy tính cá nhân cổ điển là một quá trình phát triển không ngừng và đầy thử thách đối với thế giới. Vi xử lý hay CPU là một trong những thành phần quan trọng nhất của máy tính hay được gọi là “Trái tim của tất cả thiết bị điện tử”. Trong những năm đầu phát triển máy tính, chúng ta đã luôn cố gắng tăng gấp đôi tần số của bộ vi xử lý (1980-1986). Các bộ vi xử lý do Intel ra mắt (công ty về phần cứng lớn nhất thế giới lúc đó) luôn chú trọng điều này. Nghĩa là vào thời điểm này với sự phát triển về công nghệ người ta ước tính rằng công nghệ có thể cho phép kích thước của bóng bán dẫn mỗi năm có thể giảm đi một nửa đồng nghĩa với việc số lượng bóng bán dẫn tăng gấp đôi trong cùng bộ vi xử lý cùng kích thước do đó chúng ta có thể tăng tần suất cho bộ vi xử lý là một điều tất yếu (định luật Moore), mọi thứ điều hoạt động tốt cho đến khi thế hệ chip pentium 4 với tần số cao nhưng nó đã xảy ra một số vấn đề khi nó hoạt động trên các thiết bị máy tính, quá trình nóng lên diễn ra quá nhanh dẫn đến quá nhiều việc cần phải giải quyết vào lúc đó và gần như là không thể giải quyết hết những vấn đề xoay quanh vào bối cảnh công nghệ lúc bấy giờ. Công thức định nghĩa cho tần số này đã sai và đánh dấu điểm cuối cùng của định luật Moore cùng lúc đó cũng là điểm bắt đầu của khái niệm mới về chip xử lý đó là khái niệm đa lõi, tăng số lõi thay vì tăng tần số và hướng tới một kiến trúc xử lý song song.

Trong cùng thời điểm này (Năm 1999) đã có một bước chuyển mình lớn trong ngành phần cứng đó là sự ra đời của “GPU đầu tiên trên thế giới” GeForce 256(kiến trúc xử lý đồ họa riêng biệt) do Nvidia công bố đã giúp cho máy tính ngày một phát triển mạnh hơn về mặt tính toán(quan trọng trong học máy ngày nay). Nghĩa là bấy giờ máy tính đã rất mạnh về mặt xử lý và không còn trở ngại gì nữa.

Ngoài ra các công nghệ khác như bộ nhớ, đĩa cứng hay các thiết bị phần cứng khác điều phát triển một cách vượt bậc. Lúc này việc ảo hóa hay máy ảo đã không còn quá xa vời để thực hiện trên các máy tính như trước vì quá trình ảo hóa đòi hỏi máy tính phải rất mạnh mẽ trong việc xử lý và việc triển khai cổ điển cũng dần đi vào lạc hậu khi không tối ưu được hết hiệu năng của máy chủ. Hay nói xa hơn là bài toán về điện toán đám mây đã tiến tới rất gần với chúng ta vì ảo hóa được đánh giá là một công nghệ cốt lõi của điện toán đám mây lúc bấy giờ, lúc này công nghệ ảo hóa bắt đầu nở rộ và trở thành một trong những ý tưởng đột phá của nhân loại trong thế kỷ 20.

## **3.2 Công nghệ ảo hóa**

Thực tế công nghệ ảo hóa đã bắt đầu từ những năm 1960 nhưng mãi đến những năm về sau công nghệ này mới được cho là một đóng góp to lớn trong thế kỷ 20. Vì khi ấy máy tính rất đắt, vi xử lý chậm và kĩ thuật ảo hóa lúc này đòi hỏi quá nhiều chi phí bỏ ra hay có thể nói là công nghệ máy tính chưa thể đạt đến trình độ để thực hiện việc ảo hóa một cách tối ưu nên việc triển khai cổ điển vẫn là lựa chọn an toàn và tối ưu, cách triển khai cổ điển thường theo mô hình một máy chủ vật lý, một hệ điều hành, một tập thư viện, phụ thuộc và cài đặt các ứng dụng lên trên nó. Khi máy tính trở nên tốt hơn và các chương trình ảo hóa ngày một phát triển và tối giản đi các “lớp” hay rõ nhất vào những năm 1990, chúng ta đã chứng kiến sự ra đời của Cơ sở hạ tầng máy tính để bàn ảo (VDI), cho phép truy cập từ xa vào máy tính để bàn, và được sử dụng trong các quy trình triển khai dự án của các công ty ngày nay giúp tăng bảo mật dự án trong những dự án khép kín. Khi công nghệ tốt hơn nữa hay ảo hóa máy tính để bàn cũng vậy (những năm 2000) nó trở nên phổ biến hơn và các công ty bắt đầu sử dụng nó để quản lý hệ thống máy tính của họ một cách hiệu quả và hiện nay cụm từ ảo hóa là một phần không thể thiếu trong tất cả các doanh nghiệp làm về công nghệ. Vậy công nghệ ảo hóa là gì?



**Hình 3.2: Kiến trúc máy tính ảo hóa**

Ảo hóa là việc sử dụng phần mềm chia phần cứng vật lý thành nhiều phần cứng ảo. Vì vậy, có thể nói ảo hóa là việc chia một máy vật lý thành nhiều máy con ảo.

Công nghệ ảo hóa là một công nghệ thực hiện ảo hóa trên máy tính, bao gồm các kỹ thuật và quy trình thực hiện ảo hóa. Các kỹ thuật và quy trình này để tạo ra một tầng trung gian giữa hệ thống phần cứng máy tính và phần mềm chạy trên nó. Mỗi máy ảo cũng được cấp phát tài nguyên phần cứng như máy thật gồm RAM, CPU, Card mạng, ổ cứng, các tài nguyên khác và có một hệ điều hành riêng. Việc này dẫn đến làm lu mờ đi cách thiết kế hệ thống thông tin cũ thường thiết kế để chạy một hệ điều hành và một ứng dụng. Điều này không khai thác triệt để hiệu năng của hệ thống máy chủ rất lớn. Ảo hóa cho phép ta vận hành nhiều máy chủ ảo trên cùng một máy chủ vật lý, dùng chung tài nguyên của máy chủ đó qua nhiều môi trường khác nhau. Giả sử một công ty chạy rất nhiều hệ thống như hệ thống cơ sở dữ liệu,hệ thống mạng, hệ thống email, hệ thống thanh toán, hệ thống Apps, … Dĩ nhiên các hệ thống này có thể đặt chung vào một máy chủ nhưng vì một danh sách dài các lý do đưa ra bắt buộc phải tách riêng các hệ thống này ra, như vậy với việc thiết kế hệ thống cũ thì ta phải thiết lập rất nhiều máy chủ con không đáng có và gây nên tình trạng hao phí tài nguyên của công ty, trong tình huống này công nghệ ảo hóa là một cứu cánh của tất cả các công ty có tình huống tương tự trên. Hiện nay các công ty dẫn đầu trong ngành ảo hóa này là các cái tên như: Xen, Microsoft, VMware, Red Hat.

### **3.2.1 Lợi ích của công nghệ ảo hóa**

* Hợp nhất được nhiều hệ thống (Server) ứng dụng trên một máy vật lý
* Linh hoạt có thể chia sẻ tài nguyên động của phần cứng chung giữa các hệ thống (Server) ảo
* Tính sẵn sàng cao (High Availability), phục hồi sau thảm họa(Disaster Recovery)
* Tiết kiệm:
* Không gian lắp đặt
* Năng lượng (hệ thống làm lạnh, vận hành nhiều hệ thống)
* Tài nguyên phần cứng

### **3.2.2 Giới thiệu ảo hóa**

#### **3.2.2.1 Các loại ảo hóa**

Có rất nhiều cách phân loại ảo hóa nhưng quy chuẩn chung sẽ có các loại chính như sau:

* Phân loại theo mục đích: gồm rất nhiều loại, chỉ liệt kê 5 loại phổ biến hay được nhắc đến hiện nay.
* **Ảo hóa Desktop**

Ảo hóa desktop là quá trình tạo ra một môi trường máy tính ảo trên một máy tính vật lý, cho phép người dùng chạy nhiều hệ điều hành và ứng dụng trên cùng một thiết bị.

Điều này cho phép người dùng có thể cài đặt và chạy các ứng dụng không tương thích với hệ điều hành chính của họ, hoặc thử nghiệm các phiên bản mới của hệ điều hành mà không ảnh hưởng đến môi trường làm việc chính.

* **Ảo hóa ứng dụng**

Ảo hóa ứng dụng là quá trình chạy ứng dụng trong một môi trường ảo độc lập, cách ly hoàn toàn khỏi hệ điều hành chính của máy tính.

Điều này giúp giảm thiểu xung đột giữa các ứng dụng và tăng tính linh hoạt trong việc triển khai và quản lý ứng dụng.

* **Ảo hóa máy chủ (Server)**

Ảo hóa máy chủ là quá trình tạo ra các máy chủ ảo trên một máy chủ vật lý, cho phép một máy chủ vật lý có thể chạy nhiều hệ điều hành và ứng dụng độc lập.

Điều này giúp tối ưu hóa sử dụng tài nguyên máy chủ, giảm chi phí vận hành và quản lý hệ thống, và cung cấp tính linh hoạt trong triển khai và quản lý ứng dụng.

* **Ảo hóa lưu trữ**

Ảo hóa lưu trữ là quá trình tạo ra các khối lưu trữ ảo từ các tài nguyên lưu trữ vật lý, cho phép quản lý lưu trữ linh hoạt hơn và tối ưu hóa sử dụng tài nguyên.

Điều này bao gồm việc tạo ra các ổ đĩa ảo, thư mục ảo và phân vùng ảo, cho phép tạo ra sao lưu, di chuyển dữ liệu và mở rộng không gian lưu trữ một cách dễ dàng.

* **Ảo hóa mạng**

Ảo hóa mạng là quá trình tạo ra mạng ảo từ cơ sở hạ tầng mạng vật lý, cho phép tách biệt và cách ly các luồng dữ liệu và tài nguyên mạng.

Điều này giúp tạo ra mạng ảo riêng biệt cho mỗi ứng dụng hoặc máy chủ, cải thiện bảo mật, hiệu suất và quản lý của mạng.

* Phân loại theo kiến trúc Hypervisor (phân loại quan trọng).
* **Hypervisor (Trình giám sát ảo)** , là một phần mềm chương trình cơ sở hoặc phần cứng máy tính tạo và chạy các máy ảo. Nó cung cấp quyền kiểm soát bộ vi xử lý và tài nguyên của máy chủ vật lý, nó phân bổ cho mỗi máy ảo những tài nguyên mà nó cần đảm bảo rằng các máy ảo không bị xung đột và can thiệp lẫn nhau. Có hai loại Hypervisor
* **Loại I (native/bare-metal hypervisors)**: Hoạt động trực tiếp trên phần cứng máy tính vật lý, và được cài đặt trực tiếp trên máy tính hoặc server. Loại hypervisor này có hiệu suất tốt hơn và được sử dụng chủ yếu cho các môi trường ảo hóa máy chủ, nơi mà hiệu suất và độ ổn định là rất quan trọng. Type 1 hypervisor có thể được cài đặt trực tiếp trên máy chủ hoặc thiết bị, mà không cần hệ điều hành. Được sử dụng trong môi trường thực tế. Ví dụ: Vmware WS, Vmware Fusion, Virtualbox, Hyper-V
* **Loại II (hosted hypervisors)**: Hoạt động trong máy chủ hệ điều hành, bộ ảo hóa này chạy ở cấp độ thứ ba phía trên phần cứng. Loại hypervisor này phổ biến trong việc ảo hóa trên máy tính cá nhân và người dùng Laptop. Type 2 hypervisor đòi hỏi một hệ điều hành chạy trên máy tính, và do đó, nó có thể ảnh hưởng đến hiệu suất của máy tính. Loại này thường được sử dụng trong các môi trường thử nghiệm và không sử dụng trong môi trường thực tế. Ví dụ: XEN, Oracle VM, VMware ESX
* Phân loại theo hãng
* **Vmware**: ESX/ESXi, Vmware WorkStation
* **Microsoft**: Hyper-V

Có hai môi trường máy chủ ảo hóa: ảo hóa toàn phần (Full virtualization) và ảo hoá một nửa (Paravirtualization).

**Full-virtualization**: Phần cứng được mô phỏng để mở rộng chạy những hệ điều hành khách trên nền tảng ảo hóa. Cho phép chạy bất kể hệ điều hành nào như một máy ảo trên máy chủ vật lý. Hệ điều hành khách không thể nhận biết được mình đang được ảo hóa và chạy trên một máy ảo, nơi các lệnh từ máy ảo được chuyển đổi từ các thành phần ảo sang các phần cứng thực sự trên máy chủ vật lý. Hệ điều hành khách sử dụng trực tiếp các thành phần ảo của máy ảo.

**Paravirtualization**: là một phương pháp ảo hóa máy chủ khác. Với phương pháp ảo hóa này, thay vì mô phỏng một môi trường phần cứng hoàn chỉnh, phần mềm ảo hóa này là một lớp mỏng (Hypervisor), hệ điều hành được sửa đổi để chạy trên bộ ảo hóa một nửa và việc điều chỉnh này liên quan đến việc chèn trình điều khiển cho phép chuyển hướng các cuộc gọi hệ thống thay vì dịch chúng. Phần mềm được trực tiếp cài đặt trong hệ điều hành ảo hóa một nửa. Do đó thật thật tiện lợi và thông minh khi sử dụng cơ chế như vậy để truy cập phần cứng có khả năng được sử dụng nhiều.

## **3.3 Docker**

### **3.3.1 Giới thiệu Docker**



**Hình 3.3: Kiến trúc Container hóa**

Docker là một nền tảng mã nguồn mở được thiết kế để hỗ trợ các lập trình viên và các quản trị hệ thống trong việc xây dựng, vận chuyển và triển khai các ứng dụng phân tán. Docker đã đưa ra một giải pháp mới cho vấn đề ảo hóa và cũng là tương lai trong ngành ảo hóa đó là thay vì tạo ra các máy ảo con chạy độc lập kiểu hypervisors (tạo phần cứng ảo và cài đặt hệ điều hành riêng tùy thuộc vào môi trường ứng dụng) để tạo và quản lý môi trường cho các ứng dụng, giờ đây các ứng dụng sẽ được đóng gói và cô lập thành các thùng chứa Container. Và các Container này chạy chung trên nhân hệ điều hành qua LXC (Linux Containers), chia sẻ tài nguyên của máy chủ, do đó, hoạt động nhẹ và nhanh hơn các máy ảo dạng hypervisor. Docker hướng đến một giải pháp tối ưu hóa cho môi trường phát triển ứng dụng.

Docker vẫn tiếp tục là một trong những nhà cung cấp hàng đầu về môi trường Container Runtime và đóng gói ứng dụng. Mặc dù đã có sự xuất hiện của nhiều công nghệ khác trong lĩnh vực này, Docker vẫn giữ vững vị thế của mình do các lý do sau:

Độ phổ biến và sự chấp nhận của cộng đồng: Docker đã trở thành một tiêu chuẩn trong việc đóng gói và triển khai ứng dụng, thu hút một cộng đồng lớn của các nhà phát triển, doanh nghiệp và người dùng cuối.

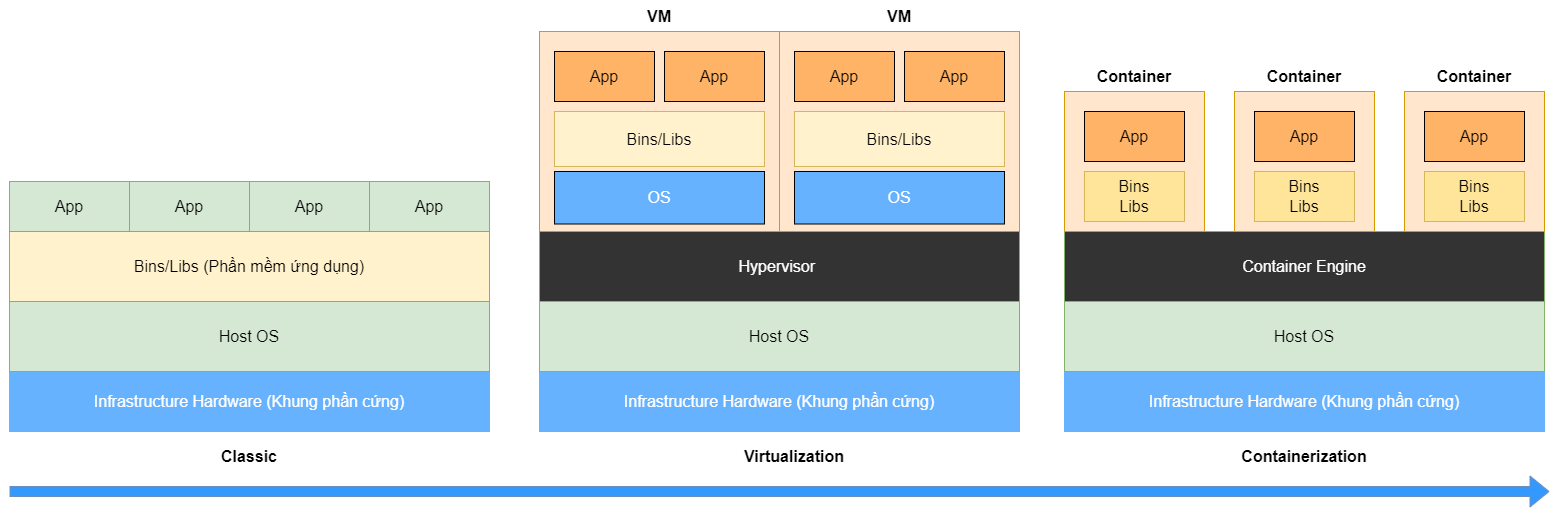
Dễ sử dụng: Docker cung cấp một giao diện dễ sử dụng cho việc đóng gói và quản lý các Container và với lượt truy cập từ cộng đồng tăng dần theo từng năm.

Hệ sinh thái phong phú: Docker không chỉ cung cấp công cụ để đóng gói container mà còn có một hệ sinh thái phong phú bao gồm Docker Hub (nơi chia sẻ và lưu trữ các image container), Docker Swarm (dịch vụ quản lý container dựa trên cụm), và Docker Compose (công cụ cho phép định nghĩa và quản lý các ứng dụng đa-container ở mức độ dễ tiếp cận).

Hỗ trợ đa nền tảng: Docker hỗ trợ nhiều nền tảng khác nhau bao gồm Linux, Windows và macOS, cho phép người dùng triển khai ứng dụng container trên nhiều môi trường khác nhau.

Docker là một công nghệ tương đối mới nên nó vẫn còn một số hạn chế. Thứ nhất, nó chỉ hỗ trợ ứng dụng có thể chạy trên Linux, ít nhất là ở thời điểm hiện tại. Docker cũng chỉ chạy trên Linux và mặc dù có thể sử dụng phần mềm bổ sung để chạy trên Windows hoặc OSX nhưng nó vẫn yêu cầu máy ảo để làm việc đó. Cụ thể máy ảo đó sẽ hoạt động trên một môi trường ảo Linux khi tải Docker về sẽ luôn đi kèm. Mặc dù đã có tiến bộ trong việc hỗ trợ các hệ điều hành khác nhưng nó vẫn chưa hoàn thiện và tương thích hoàn toàn với hệ điều hành.

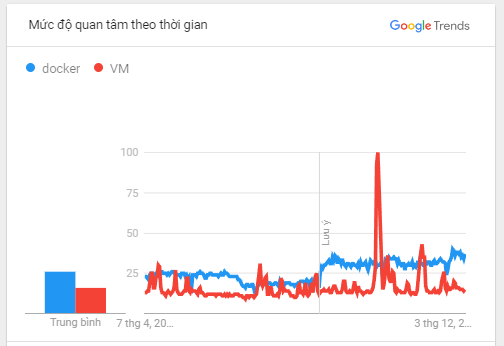
### **3.3.2 Docker và VM**



**Hình 3.4 : Lịch sử phát triển của môi trường triển khai ứng dụng**

**Bảng so sánh ảo hóa và Container**

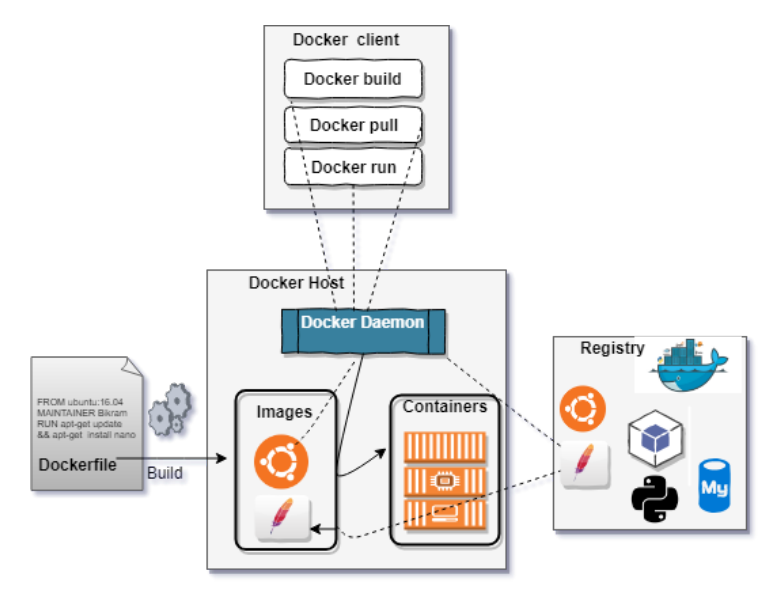
| Thông số | Máy ảo(VM) | Docker Container |
| --- | --- | --- |
| Guest OS | Mỗi VM điều yêu cầu phần cứng ảo và kernel được tải vào bộ nhớ của nó. | Hình ảnh kernel được tải vào bộ nhớ vật lý và tất cả các vùng chứa đều có chung hệ điều hành và kernel. |
| Communication | Giao tiếp thông qua các thiết bị Ethernet | Nó sử dụng các cơ chế IPC như sockets, signals, pipes, v.v. |
| Security | Phụ thuộc vào trình ảo hóa (hypervisor) | Kiểm soát các truy cập bắt buộc (MAC). |
| Performance | Cơ sở hạ tầng được truyền từ máy khách đến máy chủ, điều này ảnh hưởng đến hiệu suất. | Cung cấp hiệu suất gần như nguyên bản so với Hệ điều hành máy chủ cơ bản. |
| Isolation | Không thể chia sẻ thư viện, tập tin, v.v. giữa máy khách và máy chủ cũng như giữa các máy khách với nhau. | Thư mục có thể được gắn kết và chia sẻ giữa các container với nhau. |
| Startup time | Mất vài phút để khởi động | Mất vài giây để khởi động |
| Storage | Vì toàn bộ kernel hệ điều hành và chương trình liên quan của nó cần được cài đặt và chạy nên cần nhiều dung lượng lưu trữ hơn. | Vì hệ điều hành cơ sở được chia sẻ nên chiếm rất ít dung lượng lưu trữ. |



**Hình 3.5: Mức độ phổ biến của Docker container và VM trên toàn thế giới trong vòng 5 năm qua (2019-2024)**

“Docker bắt đầu với một dự án mã nguồn mở tại dotCloud, một công ty dịch vụ nền tảng dựa trên đám mây, vào đầu năm 2013. Ban đầu, Docker là một phần mở rộng tự nhiên của công nghệ mà công ty đã phát triển để chạy doanh nghiệp đám mây của mình triển khai trên hàng ngàn máy chủ. Nó được viết bằng Go, một ngôn ngữ lập trình kiểu tĩnh được phát triển bởi Google với cú pháp dựa trên ngôn ngữ C. Tiến triển mạnh sau sáu đến chín tháng, và công ty đã tuyển dụng một CEO mới, tham gia vào đó là Linux Foundation, đổi tên thành Docker Inc., và công bố rằng họ đang dời trọng tâm của mình sang việc phát triển Docker và hệ sinh thái Docker. Theo như Google đã tiết lộ rằng hầu hết cơ sở hạ tầng của họ điều được thiết kế trên hệ điều hành Linux” [6]. Và hiện nay việc triển khai môi trường ứng dụng container runtime của các doanh nghiệp đa số hầu hết sử dụng Docker cho phát triển ứng dụng. Và nó được đánh giá là Top 1 về độ phổ biến cũng như hiệu quả mang lại. Tiếp sau đó có thể kể đến CRI-O(hỗ trợ tốt nhất cho K8S) và ContainerD.

### **3.3.3 Kiến trúc Docker**



**Hình 3.6: Kiến trúc Docker**

Docker sử dụng kiến trúc client-server đơn giản. Các container client từ máy chủ hoặc máy ảo tương tác với Docker daemon, người có trách nhiệm xây dựng, chạy và phân phối các container. Giao tiếp giữa Docker daemon và các container được thực hiện thông qua mạng bridge. Máy khách Docker và daemon có thể chạy trên cùng một hệ thống hoặc có thể kết nối máy khách Docker với daemon Docker từ xa. Máy khách Docker và daemon giao tiếp bằng API REST, qua ổ cắm UNIX hoặc giao diện mạng. Một ứng dụng khách Docker khác là Docker Compose, cho phép chúng ta làm việc với các ứng dụng bao gồm một bộ vùng chứa. Nó bao gồm các thành phần khác nhau như Docker images, Docker registries và Docker containers.

Quá trình trên có thể mô tả ngắn như sau: Nhà phát triển bắt đầu bằng việc tạo một tập tin Dockerfile và sau đó tiến hành xây dựng (build) tập tin Dockerfile. Quá trình này tạo ra các images, sau đó các images này được gọi tới Docker daemon để thực hiện các API cần thiết cho việc tạo ra các containers từ các images này.

Một số lệnh cơ bản:

docker version: Lệnh này dùng để kiểm tra version docker.

docker pull: Lệnh này dùng để tải các image trên docker hub về.

docker images: Lệnh này dùng để liệt kê các image có trên máy tính.

docker run: Lệnh này dùng để tạo 1 container với images của bạn.

docker ps: Lệnh này để liệt kê các container đang chạy.

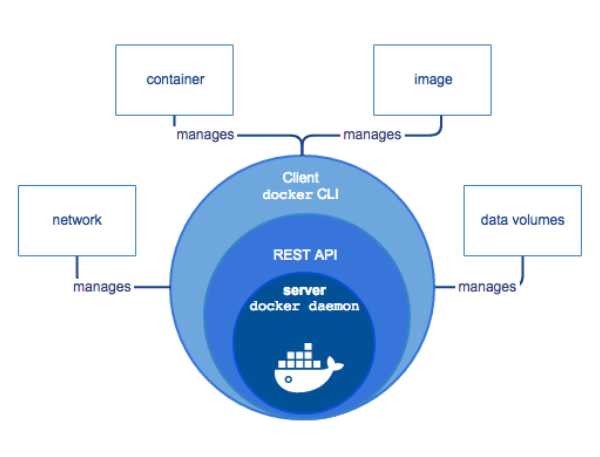
docker stop: Lệnh này dùng để dừng 1 container đang chạy.

docker start: Để start lại container đã dừng trước đó, bạn dùng lệnh docker start.

docker rm: Lệnh này dùng để xóa container đã tạo trước đó, nếu container đang chạy, cần thêm tham số -f.

docker exec: Lệnh này dùng để chạy 1 lệnh trong container, ví dụ bên dưới là liệt kê các tệp tin, thư mục trong folder root của container.

#### **3.3.3.1 Docker Engine**



**Hình 3.7: Docker Engine**

Docker Engine là một nền tảng cơ bản được đặt ở trên Docker để tạo và chạy các Container Docker. Nó sử dụng Dockerfile để tạo ra Container. Hình ảnh cơ bản được kéo từ Docker Hub hoặc chúng ta có thể cài đặt ứng dụng riêng và chạy dịch vụ của chúng. Docker Engine sử dụng các thành phần khác nhau cho phép chúng ta phát triển, lắp ráp, vận chuyển và chạy ứng dụng.

* Docker Daemon: Đây là một tiến trình nền luôn lắng nghe các yêu cầu API Docker và xử lý chúng. Daemon cũng quản lý hình ảnh Docker, container, mạng, lưu trữ và ổ đĩa.
* REST API: Một API để tương tác với Docker daemon.
* Docker CLI: Đây là giao diện dòng lệnh cho các client tương tác với Docker daemon.

#### **3.3.3.2 Docker Containers**

Những cải tiến trong nhân Linux đã dẫn tới sự phát triển của ảo hóa dựa trên Container. Container cho phép các nhà phát triển đóng gói và cô lập các thành phần, phụ thuộc, thư viện của ứng dụng với tất cả những gì ứng dụng cần để hoạt động. Khái niệm này mang lại tính di động và tính linh hoạt chưa từng có vì một Container có thể được triển khai trên bất kỳ máy chủ nào có hỗ trợ Container Engine, cho dù đó có là môi trường phát triển, thử nghiệm, hoặc thương mại (dev,test,prod). Việc sử dụng Kernel của hệ điều hành chủ động đảm bảo hiệu suất tối ưu và tính tương thích cao. Chúng cũng là công cụ để cung cấp phần mềm tập trung vào nền tảng như một dịch vụ (PaaS).

Một trong những đặc điểm chính của containers là khả năng hoạt động một cách độc lập trong khi vẫn được chạy trực tiếp trên máy chủ. Mỗi container có không gian quy trình tuần tự rõ ràng và giao diện mạng riêng biệt, đảm bảo sự cách ly hiệu quả đồng thời tối ưu hóa tài nguyên có sẵn. Sự cách ly này cho phép phân tách cơ sở hạ tầng ứng dụng thành các phần nhỏ nhẹ dễ triển khai và tái sử dụng.

Containers cung cấp một giải pháp thông minh để giải quyết các thách thức liên quan đến việc triển khai và quản lý ứng dụng. Khả năng đóng gói ứng dụng với những phụ thuộc của chúng, đảm bảo sự cách ly hiệu quả và tính di động tối đa, là một công cụ không thể thiếu đối với các nhà phát triển và quản trị. Với sự xuất hiện của containers, các doanh nghiệp có thể tận hưởng sự linh hoạt tăng cao, tối ưu hóa sử dụng tài nguyên và giảm chi phí liên quan đến cơ sở hạ tầng vốn dĩ đã thuộc về các công nghệ VM hiện có. Các lợi ích của Container mang lại là:

* Hiệu suất tốt hơn so với các máy ảo (VMs): Containers có thể khởi động ngay lập tức và chia sẻ nhân hệ điều hành với máy chủ, giúp tối ưu hóa sử dụng tài nguyên và giảm thiểu thời gian khởi động. So với việc khởi động và chạy một máy ảo, việc sử dụng container có thể cung cấp hiệu suất cao hơn và tiết kiệm thời gian.
* Khả năng di động giữa các môi trường: Containers cho phép di chuyển ứng dụng từ một môi trường sang một môi trường khác một cách dễ dàng, bao gồm cả việc triển khai trên nhiều đám mây. Điều này tạo ra sự linh hoạt và lựa chọn rộng rãi trong việc lựa chọn nơi triển khai ứng dụng, giúp tối ưu hóa chi phí và tăng cường sự ổn định.
* Tính nhất quán giữa các môi trường phát triển, thử nghiệm và sản phẩm: Containers cung cấp một cách tiếp cận thống nhất giữa các môi trường khác nhau. Điều này đảm bảo rằng các ứng dụng được triển khai và chạy giống nhau trên mọi môi trường, từ phát triển đến sản xuất, giảm thiểu rủi ro và đảm bảo tính ổn định và nhất quán.
* Khả năng phân tách ứng dụng một cách dễ dàng: Containers cho phép modularize ứng dụng thành các phần nhỏ hơn và độc lập, giúp quản lý và mở rộng ứng dụng một cách hiệu quả hơn. Điều này làm cho việc phát triển và bảo trì ứng dụng trở nên dễ dàng hơn, đồng thời tăng cường tính linh hoạt và tái sử dụng.
* Quản lý kế thừa kỹ thuật (ứng dụng cũ) thông qua sự cách ly: Containers cung cấp một môi trường cách ly cho các ứng dụng, giúp quản lý và triển khai các ứng dụng cũ một cách an toàn và hiệu quả. Điều này giúp hạn chế tác động của các thay đổi và cập nhật trên các ứng dụng hiện có, đồng thời giữ cho hệ thống hoạt động một cách ổn định và tin cậy.

#### **3.3.3.3 Docker Images**

Trong môi trường Docker, các ứng dụng được cung cấp dưới dạng hình ảnh ứng dụng chỉ đọc (read-only application images). Một hình ảnh như vậy chứa ứng dụng cùng với một tham chiếu đến hình ảnh cha của nó (quan hệ kế thừa giữa các ảnh), tạo ra một kiến trúc có nhiều tầng. Nếu một hình ảnh không có hình ảnh cha, nó được gọi là hình ảnh gốc. Ví dụ phổ biến về hình ảnh gốc là của các bản phân phối Linux. Quá trình tạo hình ảnh ứng dụng cho Docker thường được gọi là "dockerizing".

Hiện nay, có hai cách khả thi để tạo hình ảnh ứng dụng. Một cách là chạy một hình ảnh hiện có như một container và thực hiện các thay đổi. Việc sử dụng lệnh docker commit lưu các thay đổi và có thể được tiếp theo bằng lệnh docker push, lưu trữ các thay đổi trong một kho lưu trữ. Cách khác là tận dụng việc xây dựng tự động của hình ảnh. Điều này được xử lý bằng một lệnh docker build, yêu cầu các chỉ thị được cung cấp trong một Dockerfile. Các chỉ thị được hỗ trợ bao gồm sao chép và tải xuống tệp, thực thi các tập lệnh, chỉ định hình ảnh cha và các thư mục dữ liệu, tiết lộ cổng và một số tùy chọn khác. Cũng có thể tạo ra một hình ảnh gốc, bằng cách cung cấp một bản sao tar của một hệ thống tệp hiện có cho lệnh docker import, hoặc chỉ định một hình ảnh đặc biệt được gọi là scratch (nó chứa một hệ thống tệp rỗng) như là hình ảnh cha. Một chuỗi các chỉ thị Dockerfile để Dockerize một ứng dụng có thể nhìn như sau:

FROM repository:image #hình ảnh cha

MAINTAINER name email #người duy trì Dockerfile

ENV key value #thiết lập biến môi trường

RUN/ADD/COPY commands #cài đặt ứng dụng, sao chép và tải xuống dữ liệu

EXPOSE port1, port2 #tiết lộ cổng (public port)

ENTRYPOINT /path/to/app #đường dẫn đến tập tin nhị phân

VOLUME /path/to/data/volume #thư mục dữ liệu

#### **3.3.3.4 Docker Network**

Docker network sẽ đảm nhiệm nhiệm vụ kết nối mạng giữa các container với nhau, kết nối giữa container với bên ngoài, cũng như kết nối giữa các cụm (swarm) docker containers. Với container và service của Docker, bạn có thể kết nối chúng lại với nhau hoặc kết nối chúng với các mạng khác nằm ngoài docker. Docker network có thể cung cấp hầu hết các chức năng mà một hệ thống mạng bình thường cần có, các loại network drivers của Docker.

* **BRIDGE**

Đây là driver mạng default của Docker. Nếu không chỉ định driver thì bridge sẽ là driver mạng mặc định khi khởi tạo.

Khi chúng ta cài đặt Docker, virtual bridge docker0 sẽ được tạo ra, docker tìm một subnet chưa được dùng trên host và gán một địa chỉ cho docker0.

Bridge network thường được sử dụng khi cần chạy ứng dụng dưới dạng các container độc lập cần giao tiếp với nhau. Các container trong cùng mạng có thể giao tiếp với nhau qua địa chỉ IP. Docker không hỗ trợ nhận diện host ở mạng này, vì vậy muốn connect thì phải dùng options links để docker có thể hiểu được địa chỉ của các service.

Bridge là driver tốt nhất cho việc giao tiếp multiple containers ở một host đơn.

* **HOST**

Dùng khi container cần giao tiếp với host và sử dụng luôn mạng ở host, vì sử dụng mạng của máy chủ đang chạy nên không còn lớp mạng nào giữa container với Docker Host phù hợp khi cần connect từ container ra thẳng ngoài host

* **OVERLAY**

Mạng lớp phủ - Overlay network tạo một mạng phân tán giữa nhiều máy chủ Docker. Kết nối nhiều Docker daemons với nhau và cho phép các cụm services giao tiếp với nhau. Chúng ta có thể sử dụng overlay network để giao tiếp dễ dàng giữa cụm các services với một container độc lập, hay giữa 2 container với nhau ở khác máy chủ Docker daemons.

Nhờ Overlay network, không cần các công việc thiết lập routing giữa các container thông qua hệ điều hành. Overlay network tạo nên một lớp phủ trên mạng của máy chủ và cho phép container kết nối đến (bao gồm cả các cụm containers) để giao tiếp một cách bảo mật. Docker đảm bảo định tuyến các gói tin đến và đi đúng container đích.

* **MACVLAN**

Mạng Macvlan cho phép chúng ta gán địa chỉ MAC cho container, điều này làm cho mỗi container như là một thiết bị vật lý trong mạng. Docker daemon định tuyến truy cập tới container bởi địa chỉ MAC. Sử dụng driver macvlan là lựa chọn tốt khi các ứng dụng khác cần phải connect đến theo địa chỉ vật lý hơn là thông qua các lớp mạng của máy chủ.

* **NONE**

Với container không cần networking hoặc cần disable đi tất cả mọi networking, chúng ta sẽ chọn driver này. Thường được dùng với mạng tùy chỉnh. Driver này không thể dùng trong cụm swarm.

#### **3.3.3.5 Docker Registry**

Khi một nhà phát triển muốn che mờ, giấu đi hình ảnh của mình, cách đơn giản nhất là tải chúng lên một registry Docker. Ta có thể chọn sử dụng một registry công cộng, có sẵn trên Docker Hub, hoặc sử dụng một registry riêng tư. Bên trong registry, các hình ảnh khác nhau của một ứng dụng đều nằm trong một kho (repository). Làm việc với kho tương tự như việc sử dụng hệ thống kiểm soát phiên bản git. Điều này có nghĩa là các lệnh như push (tải lên phiên bản mới) và pull (tải về phiên bản mới nhất) được sử dụng để tương tác với registry. Registries cũng đi kèm với một số tính năng giúp trong quá trình tạo ra các phiên bản mới của hình ảnh. Tự động xây dựng (Automatic builds) cung cấp một cách tự động xây dựng hình ảnh ứng dụng trong một kho từ nguồn github hoặc bitbucket và một Dockerfile, trong khi Webhooks và Webhook chains cho phép gửi một hoặc nhiều yêu cầu HTTP với các payload JSON cụ thể. Điều này có thể được sử dụng như một cách để gửi thông báo về một cập nhật mới của hình ảnh ứng dụng. Hiện có hai phiên bản của dự án Docker registry, cả hai đều là mã nguồn mở. Phiên bản cũ hơn (v1), viết bằng Python, đã được sử dụng trong Docker cho đến phiên bản v1.6, cho đến khi phiên bản ổn định của phiên bản hiện tại, Distribution (v2), được phát hành, được viết bằng Go. Distribution tuyên bố cung cấp các yêu cầu push và pull nhanh hơn, đồng thời có cách triển khai hiệu quả hơn. Mặc dù các phiên bản cố gắng duy trì tính tương thích ngược với các phiên bản càng nhiều càng tốt, một số điểm kết thúc của registry vẫn có sự khác biệt nhỏ.

### **3.3.4 Docker Container Orchestration**

Container Orchestration là quá trình tự động hóa, điều phối và quản lý các container trong một môi trường hệ thống phân tán. Các công cụ điều phối container giúp tự động hóa việc triển khai, mở rộng và quản lý các ứng dụng chạy trong các container. Điều này giúp tăng cường hiệu suất, độ tin cậy và linh hoạt cho ứng dụng và hạ tầng.

Container Orchestration giải quyết các vấn đề như:

Triển khai tự động: Tự động triển khai ứng dụng vào các container.

Mở rộng tự động: Tự động mở rộng ứng dụng bằng cách thêm hoặc giảm số lượng container dựa trên nhu cầu tài nguyên.

Cân bằng tải: Phân phối lưu lượng truy cập đến các container đang chạy để đảm bảo hiệu suất và độ tin cậy.

Tự phục hồi: Tự động khôi phục các container hoặc các service khi có lỗi xảy ra.

Quản lý tài nguyên: Phân phối và quản lý tài nguyên máy chủ, CPU, bộ nhớ và lưu trữ cho các container.

Hiện nay có các công cụ điều phối Container nổi tiếng đã đề cập ở trên như:

Kubernetes(nổi tiếng nhất) : Phát triển bởi Google và hiện được quản lý bởi Cloud Native Computing Foundation (CNCF). K8S là một nền tảng mã nguồn mở được sử dụng rộng rãi và phổ biến nhất để quản lý container và tự động hóa việc triển khai, mở rộng và quản lý ứng dụng. Hỗ trợ tất cả các Container Runtime đạt chuẩn CRI (Container Runtime Interface) để chạy trên nhà điều phối này.

Docker Swarm(đi kèm tích hợp sẵn): Là một công cụ quản lý container tích hợp sẵn trong Docker. Dễ sử dụng triển khai và học tập, Swarm cung cấp khả năng tự động hóa việc triển khai các ứng dụng Docker trên một cụm máy chủ. Chỉ cần học Docker Compose là có thể tiến đến triển khai được Docker Swarm, thích hợp cho những hạ tầng ứng dụng vừa và nhỏ.

Apache Mesos: Mesos không chỉ hỗ trợ Docker mà còn hỗ trợ nhiều loại container khác nhau. Nó cung cấp một cơ chế chia sẻ tài nguyên mạnh mẽ và cho phép triển khai ứng dụng phân tán trên các hạ tầng lớn.

#### **3.3.4.1 Docker Compose**

Docker Compose Dự án này là một dự án mã nguồn mở chính thức của Docker. Nó chịu trách nhiệm điều phối nhanh chóng các cụm Container Docker, quản lý các Container một cách dễ dàng và hiệu quả cũng như xác định và chạy nhiều Container. Docker đã giúp việc thiết lập môi trường phát triển cục bộ trở nên dễ dàng hơn. Tuy nhiên, nếu bạn muốn tạo nhiều vùng chứa cho ứng dụng của mình, bạn phải tạo một số tệp Docker. Điều này làm tăng thêm gánh nặng cho việc duy trì chúng và cũng khá tốn thời gian.

Docker Compose giải quyết vấn đề này bằng cách cho phép bạn sử dụng tệp YAML để vận hành các ứng dụng nhiều vùng chứa cùng một lúc. Bạn có thể đặt số lượng vùng chứa mong muốn, bản dựng và thiết kế lưu trữ của chúng, sau đó bằng một bộ lệnh duy nhất, bạn có thể xây dựng, chạy và định cấu hình tất cả các vùng chứa.

Docker Compose rất phù hợp cho các môi trường phát triển, thử nghiệm và chạy thử cũng như các quy trình tích hợp liên tục(trên một máy) chứ không phải trên một môi trường thực tế(nhiều máy tính), việc triển khai trên những môi trường thực tế thì sẽ cần những Docker Orchestration Tools tùy thuộc vào nhu cầu của dự án dễ tiếp cận hay phức tạp tùy tình huống, ưa chuộng nhất hiện nay đối với phức tạp vẫn là Kubernetes và dễ triển khai là Docker Swarm bằng cách triển khai các tệp Docker Compose lên trên nó.

Có một số phiên bản của định dạng tệp của Docker Compose: 1, 2, 2.x và 3.x. Bảng sau đây hiển thị các bản phân phối docker cụ thể được tệp Compose hỗ trợ như sau:



**Hình 3.8: Phiên bản Docker Compose**

Những trường cần lưu ý khi xây dựng tệp Docker Compose YAML:

* build: Trường cho phép chỉ định vị trí của tệp Dockerfile để có thể xây dựng image bằng Docker Compose.
* image: Trường chỉ định image Docker sẽ sử dụng.
* networks: Trường chỉ định các mạng sẽ sử dụng mặc định mạng có tên default.
* volumes: Trường chỉ định khối lượng dữ liệu sử dụng.
* environment: Trường chỉ định biến môi trường được cài đặt.
* ports: Trường chỉ định cổng được ánh xạ từ máy chủ đến vùng chứa (Host port:Container Port).
* expose: Tùy chọn để hiển thị các cổng bên trong bộ chứa Docker, cho phép các bộ chứa khác kết nối với các cổng này nhưng không ánh xạ chúng tới máy chủ Docker.
* depends\_on: Trường chỉ định sự phụ thuộc giữa các dịch vụ.
* restart: Cung cấp một chiến lược khởi động lại nếu vùng chứa bị lỗi.
* command: Trường lệnh được thực thi khi vùng chứa khởi động.
* healthcheck: Chỉ định cấu hình kiểm tra sức khỏe của vùng chứa.
* configs: Chỉ định tệp cấu hình được sử dụng bởi vùng chứa.
* secrets: Chỉ định dữ liệu bí mật được sử dụng bởi vùng chứa.
* hostname và container\_name: tên máy chủ và tên vùng chứa, tên máy chủ được sử dụng để đặt tên máy chủ của vùng chứa đây là tên được sử dụng bên trong vùng chứa, còn tên vùng chứa là tên được sử dụng trên máy chủ Docker.

Một số lệnh cơ bản của Docker Compose:

Cú pháp:

$ docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]

$ docker-compose -h|--help

Một số câu lệnh thường dùng:

Kiểm tra phiên bản docker-compose: $ docker-compose --version

Hiển thị danh sách containers: $ docker-compose ps

Tạo và khởi động containers: $ docker-compose up

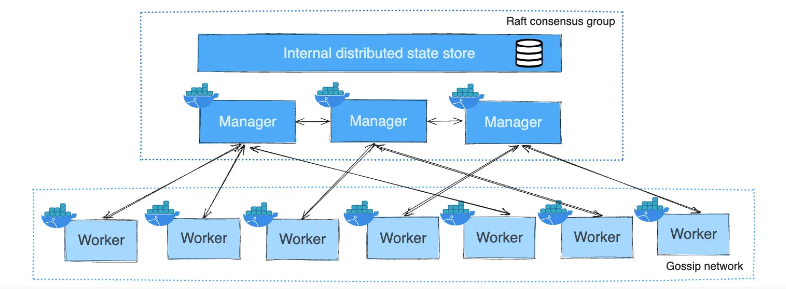
Xây dựng hoặc tái xây dựng dịch vụ: $ docker-compose build

Xác nhận hoặc hiển thị file config Compose: $ docker-compose config

Loại bỏ, ngừng Containers: $ docker-compose rm

#### **3.3.4.2 Docker Swarm**

Docker Swarm là một công cụ quản lý điều phối đã được đi kèm tích hợp cài đặt với Docker. Docker Swarm rất dễ cài đặt và sử dụng, nó sử dụng Docker Compose YAML để quản lý các cấu hình của ứng dụng. Chỉ cần thành thạo sử dụng Docker Compose thì có thể dễ dàng làm quen với Docker Swarm. Docker Swarm cho phép tạo một cụm gồm một hoặc nhiều nút Docker được gọi là bầy đàn(Swarm). Một Swarm bao gồm một hoặc nhiều nút: máy vật lý hoặc máy ảo chạy Docker Engine. Có thể sử dụng Docker Command để quản lý và thao tác với các cluster Docker Swarm.



**Hình 3.9: Kiến trúc Docker Swarm**

Trong Docker Swarm gồm 2 thành phần chính: Nút quản lý (Manager Node), Nút công nhân (Worker Node).

Nút quản lý (Manager Node):

* Nút quản lý là các máy chủ Docker chịu trách nhiệm quản lý toàn bộ trạng thái cluster Docker Swarm.
* Các nút quản lý đảm bảo rằng các dịch vụ và các container được triển khai và chạy đúng cách trên toàn bộ cluster.
* Chúng lưu trữ trạng thái của toàn bộ cluster và quản lý việc phân phối công việc (workload) đến các nút công nhân.

Nút công nhân (Worker Node):

* Nút công nhân là các máy chủ Docker được sử dụng để chạy các container.
* Các nút công nhân không quản lý cluster nhưng chấp nhận công việc được gửi từ các nút quản lý và chạy các container tương ứng. Trong mỗi nút công nhân sẽ có các Agent chịu trách nhiệm gửi tín hiệu cho nút quản lý.
* Chúng chịu trách nhiệm thực thi các tác vụ (tasks) mà nút quản lý gửi cho chúng, bao gồm việc triển khai và chạy các container.

Và đặc biệt trong Docker Swarm bạn có thể thay đổi vai trò bằng cách thăng cấp, tiến cử cho một nút Worker thành một nút Manager bằng lệnh $ docker node promote. Ví dụ: bạn có thể muốn thăng cấp nút công nhân khi đưa nút người quản lý ngoại tuyến để bảo trì. .và ngược lại cũng như hạ cấp chúng, bạn cũng có thể hạ cấp nút người quản lý xuống nút công nhân.

# **CHƯƠNG 4: THỰC NGHIỆM VÀ ĐÁNH GIÁ KẾT QUẢ**

## **4.1 Kịch bản triển khai**

Trong Chương này sẽ trình bày kịch bản nghiên cứu trong luận văn về Big Data ứng với hai công nghệ là Hadoop và Spark trong một quy trình xử lý Batch Processing cơ bản nhất. Mục tiêu của luận văn này đạt được:

* Xây dựng được Hadoop HDFS Cluster lưu trữ dữ liệu trên Docker Swarm.
* Xây dựng được Spark Cluster và xử lý tập dữ liệu ở HDFS trên một bài toán cơ bản bằng Docker Swarm và lên lịch xử lý bằng Apache Airflow.
* Trong đó Apache Airflow chỉ có tác dụng kết nối tới cụm Apache Spark và tự động hóa quy trình Spark Job được chỉ định bằng DAGs(Directed Acyclic Graphs) nên sẽ không được mô tả chi tiết và không nằm trong đề tài nghiên cứu mà chỉ là một phần công cụ bổ trợ cho việc lập lịch tự động được thêm vào.
* Thực hiện xử lý mô phỏng một bài toán đơn giản và trình chiếu được dữ liệu của bài toán đó trên Apache Superset thông qua kết nối với Apache Hive.

Trong lĩnh vực xử lý dữ liệu lớn, các bài toán về Big Data thường quy về việc xử lý các ma trận lớn hoặc cực lớn. Các ma trận này thường được chia nhỏ thành các ma trận nhỏ hơn, phù hợp để xử lý. Nhiệm vụ của người nghiên cứu là áp dụng các kỹ thuật toán học để xử lý và phân tích các ma trận này theo mục đích cụ thể. Thực tế cho thấy, những ma trận này có thể thiếu sót, chứa sai số, nhiễu ở mức cao, hoặc có nhiều dữ liệu quan hệ đồ thị phức tạp. Trong trường hợp này, việc sử dụng Spark RDD (Resilient Distributed Dataset) rất hiệu quả vì RDD cho phép xử lý dữ liệu một cách linh hoạt và mạnh mẽ đối với dữ liệu phi cấu trúc hoặc bán cấu trúc. Ngược lại, đối với Spark DataFrame và Spark SQL, chúng yêu cầu dữ liệu phải đáp ứng một dạng cấu trúc cụ thể. DataFrame và SQL thường thích hợp hơn cho dữ liệu đã được cấu trúc rõ ràng, bởi vì chúng cung cấp các API dễ sử dụng và có khả năng tối ưu hóa truy vấn rất mạnh mẽ. Các công cụ này cho phép người dùng thực hiện các thao tác dữ liệu một cách dễ dàng và hiệu quả hơn, nhờ vào việc tự động hóa nhiều khía cạnh của quá trình xử lý và tối ưu hóa dữ liệu.

Trong nghiên cứu này thực hiện tính toán trên một tập dữ liệu CSV, cụ thể là ma trận dạng bảng biểu diễn độ trễ của hãng hàng không từ Airline Delay Analysis (kaggle.com). Bộ dữ liệu này bao gồm các chuyến bay nội địa ở Hoa Kỳ từ năm 2009 đến năm 2020, được thu thập và quản lý bởi “Cục Thống Kê Giao Thông Vận Tải của DOT”, và có nhiều thuộc tính khác nhau. Ban đầu, một nỗ lực đã được thực hiện để tiền xử lý và chuẩn hóa tập dữ liệu cho năm 2020. Tuy nhiên, việc không thể suy ra một sự kết hợp có ý nghĩa cho các hồ sơ này dẫn đến quyết định không sử dụng toàn bộ dữ liệu của năm đó trong quá trình phân tích. Do đó, dữ liệu cho các năm 2020 và 2019 đã bị loại khỏi phân tích do thiếu một số trường dữ liệu quan trọng. Trong nghiên cứu chỉ thực hiện tính toán từ một phần trong mẫu dữ liệu trên từ năm 2009 đến 2013.

Tập dữ liệu ma trận đã được tổng hợp hoàn chỉnh, không chứa dữ liệu nhiễu, sai số và có cấu trúc rõ ràng, ngoại trừ một số trường không được công bố dữ liệu thực tế. Những trường này sẽ được loại bỏ trong quá trình xử lý để đảm bảo tính chính xác và hiệu quả của phân tích. Nên tập dữ liệu này sẽ được xử lý bằng Spark DataFrame.

**Bảng mô tả trường dữ liệu:**

| **Cột** | **Miêu tả** |
| --- | --- |
| FL\_DATE | Ngày bay ở định dạng yyyy/mm/dd |
| OP\_CARRIER | Mã định danh duy nhất của hãng hàng không |
| OP\_CARRIER\_FL\_NUM | Số chuyến bay của hãng hàng không |
| ORIGIN | Mã duy nhất của sân bay khởi hành |
| DEST | Mã duy nhất của sân bay điểm đến |
| CRS\_DEP\_TIME | Thời gian khởi hành dự kiến |
| DEP\_TIME | Giờ khởi hành thực tế |
| DEP\_DELAY | Tổng thời gian trễ khi khởi hành, tính bằng phút |
| TAXI\_OUT | Thời gian tính từ lúc khởi hành từ sân bay xuất phát đến thời điểm WHEELS\_OFF |
| WHEELS\_OFF | Ngay khi máy bay rời khỏi mặt đất |
| WHEELS\_ON | Ngay khi máy bay chạm đất |
| TAXI\_IN | Thời gian từ lúc đến WHEELS\_ON đến khi đến sân bay nơi đến |
| CRS\_ARR\_TIME | Thời gian đến dự kiến |
| ARR\_TIME | Thời gian đến thực tế |
| ARR\_DELAY | Tổng độ trễ khi đến nơi (âm nếu đến sớm), tính bằng phút |
| CANCELLED | Chuyến bay bị hủy (1 = bị hủy) |
| CANCELLATION\_CODE | Lý do hủy chuyến:  A - Hãng hàng không/Hãng vận chuyển;  B – Thời tiết;  C - Hệ thống hàng không quốc gia;  D - Bảo mật |
| DIVERTED | Máy bay hạ cánh ở sân bay khác với sân bay nơi đến |
| CRS\_ELAPSED\_TIME | Thời gian bay dự kiến |
| ACTUAL\_ELAPSED\_TIME | Thời gian bay thực tế (AIR\_TIME + TAXI\_IN + TAXI\_OUT) |
| AIR\_TIME | Khoảng thời gian máy bay đang bay. Thời gian giữa WHEELS\_OFF và WHEELS\_ON |
| DISTANCE | Khoảng cách giữa sân bay ORIGIN và DESTINATION sân bay |
| CARRIER\_DELAY | Sự chậm trễ do hãng hàng không gây ra, tính bằng phút |
| WEATHER\_DELAY | Sự chậm trễ do điều kiện thời tiết |
| NAS\_DELAY | Sự chậm trễ do Hệ thống Hàng không Quốc gia gây ra |
| SECURITY\_DELAY | Sự chậm trễ do kiểm tra an ninh |
| LATE\_AIRCRAFT\_DELAY | Sự chậm trễ do máy bay đến muộn |
| Unnamed: 27 | Cột không được sử dụng (chỉ hiện diện trong tệp CSV từ 2009 đến 2018 và 2020) |
| Unnamed: 20 | Cột không được sử dụng (chỉ hiện diện trong 2019.csv) |

## **4.2 Thiết lập môi trường**

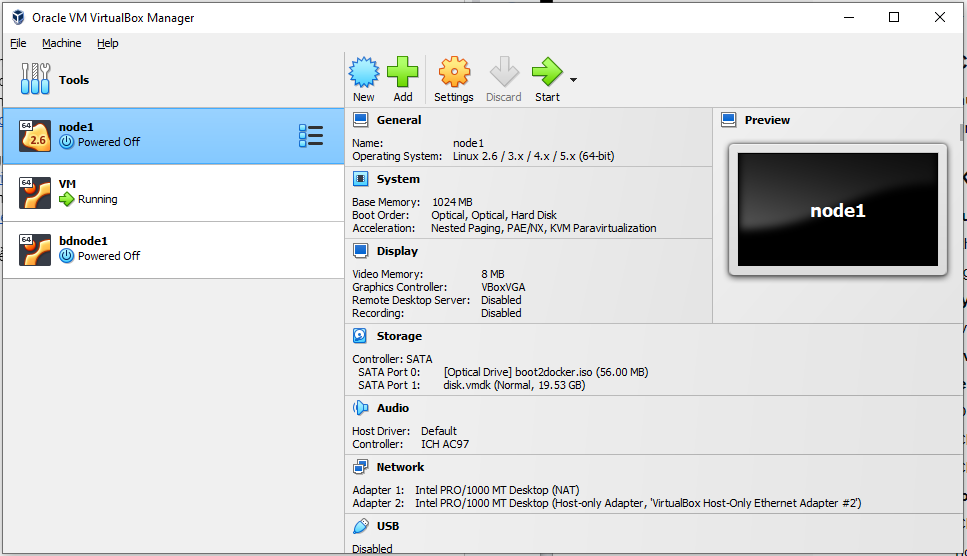
### **4.2.1 Tải các phần mềm cần thiết cho môi trường nghiên cứu**

Việc thiết lập chỉ xoay quanh nghiên cứu về xử lý lớn đa máy nên những triển khai đơn giản và cơ bản sẽ được ưu tiên thực hiện như cách triển khai: Docker Swarm dễ nhất trong các cách phù hợp trong môi trường sản xuất vừa và nhỏ phù hợp cho việc thiết kế cụm nghiên cứu, Oracle VM VirtualBox phần mềm ảo hóa dễ tiếp cận cho người mới bắt đầu.

* Tải hệ điều hành mã nguồn mở Linux với phiên bản phân phối Ubuntu (<https://ubuntu.com/download/desktop/thank-you?version=22.04.4&architecture=amd64>).
* Yêu cầu tắt Hyper-V để tải phần mềm ảo hóa mã nguồn mở của Oracle (<https://www.virtualbox.org/wiki/Downloads>).

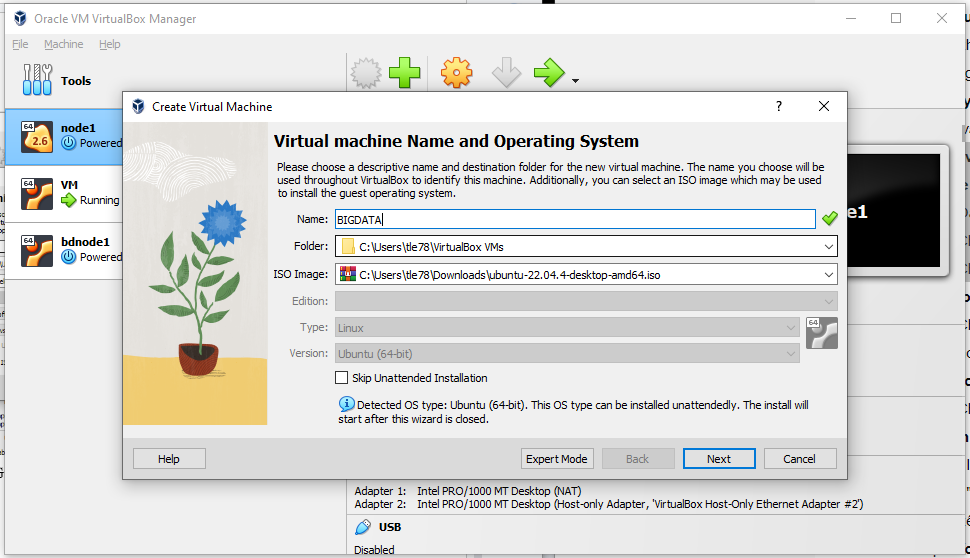
### **4.2.2 Tạo các máy ảo trên hệ điều hành Linux**

* Sau khi tải xong các phần mềm cần thiết thì tiến hành tạo mới máy ảo như sau:



**Hình 4.1: Giao diện VirtualBox Manager**

Trên thanh cài đặt ta nhấn Machine và tiếp tục nhấn “New” để tiến tạo một máy ảo mới.



**Hình 4.2: Giao diện tạo Virtual Machine**

Đặt tên và cấu hình máy ảo:

Name and operating system:

Đặt tên cho máy ảo (ví dụ: BIGDATA).

Chọn nơi lưu trữ máy ảo (ví dụ: default ở ổ C).

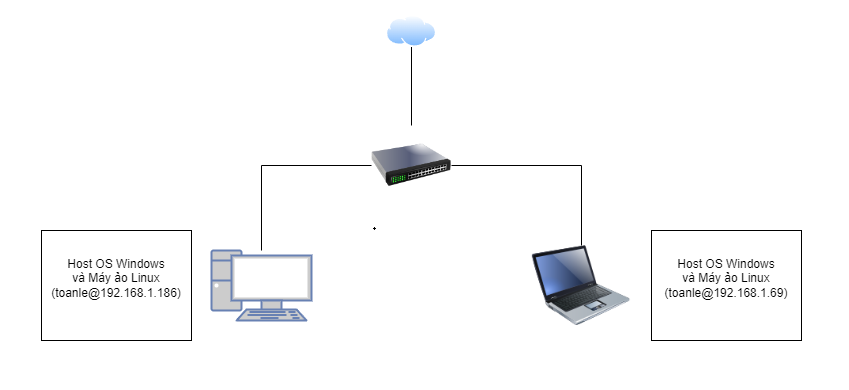
Chọn phiên bản hệ điều hành (ví dụ: Ubuntu 64-bit một nhánh trong hệ điều hành Linux).

Tiến hành tạo thông tin cho máy ảo như tên đăng nhập, mật khẩu, Hostname, Domain Name, Memory size, CPU Process, Hard disk, cài đặt loại ổ cứng và nhấn Start để tiến hành khởi.

Sau khi đã hoàn thành các thủ tục tạo cơ bản thì sẽ tiến hành Install hệ điều hành về máy ảo và tiếp tục quá trình tạo như thông thường và hoàn thành cài đặt.

### **4.2.3 Cài đặt Docker lên các máy ảo**

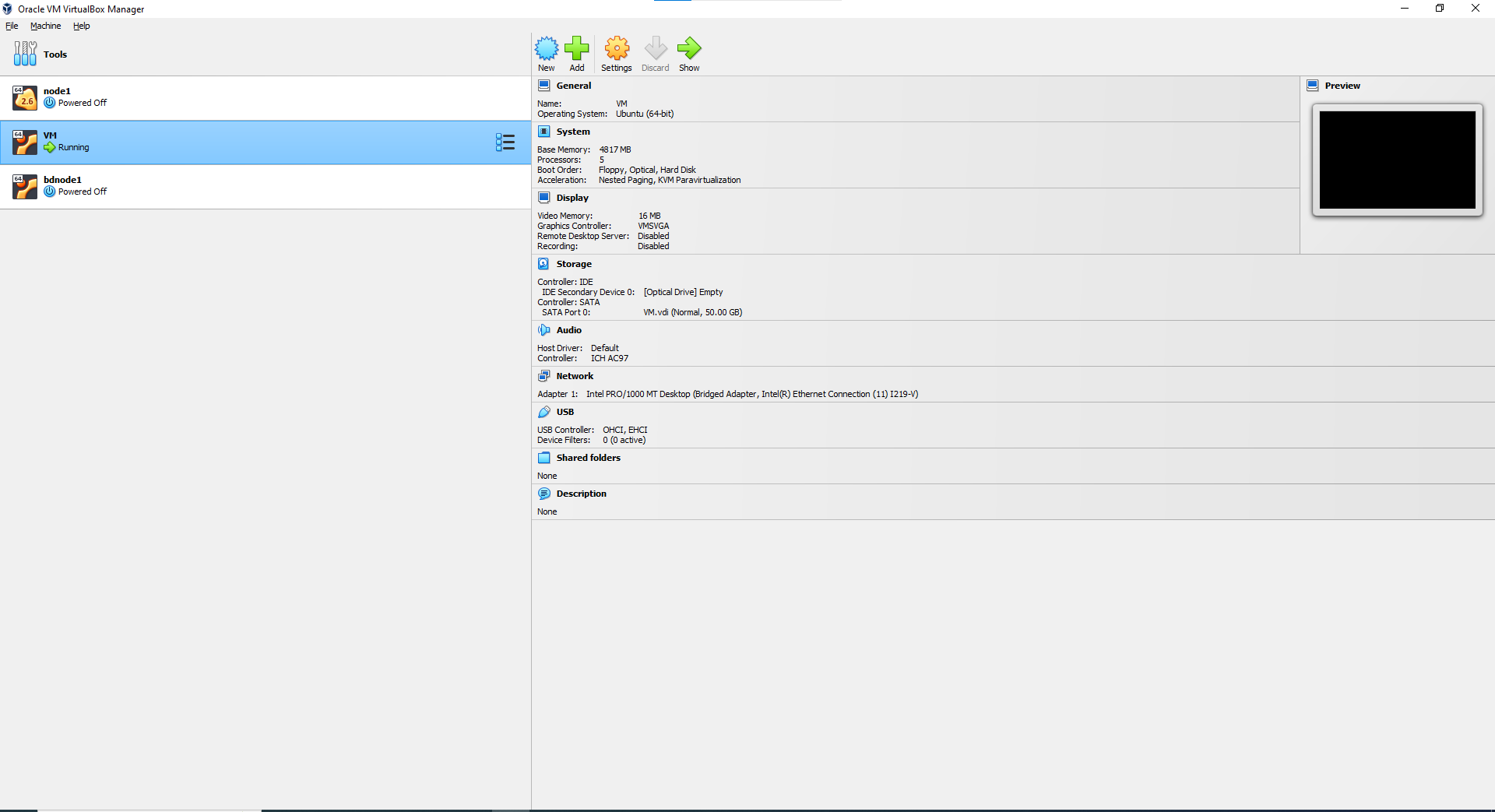
Sơ bộ về môi trường nghiên cứu được thiết kế rất cơ bản bao gồm 2 máy tính và những setup cực kỳ cơ được thiết kế để kết nối giữa 2 máy tính:



**Hình 4.3: Môi trường thí nghiệm**

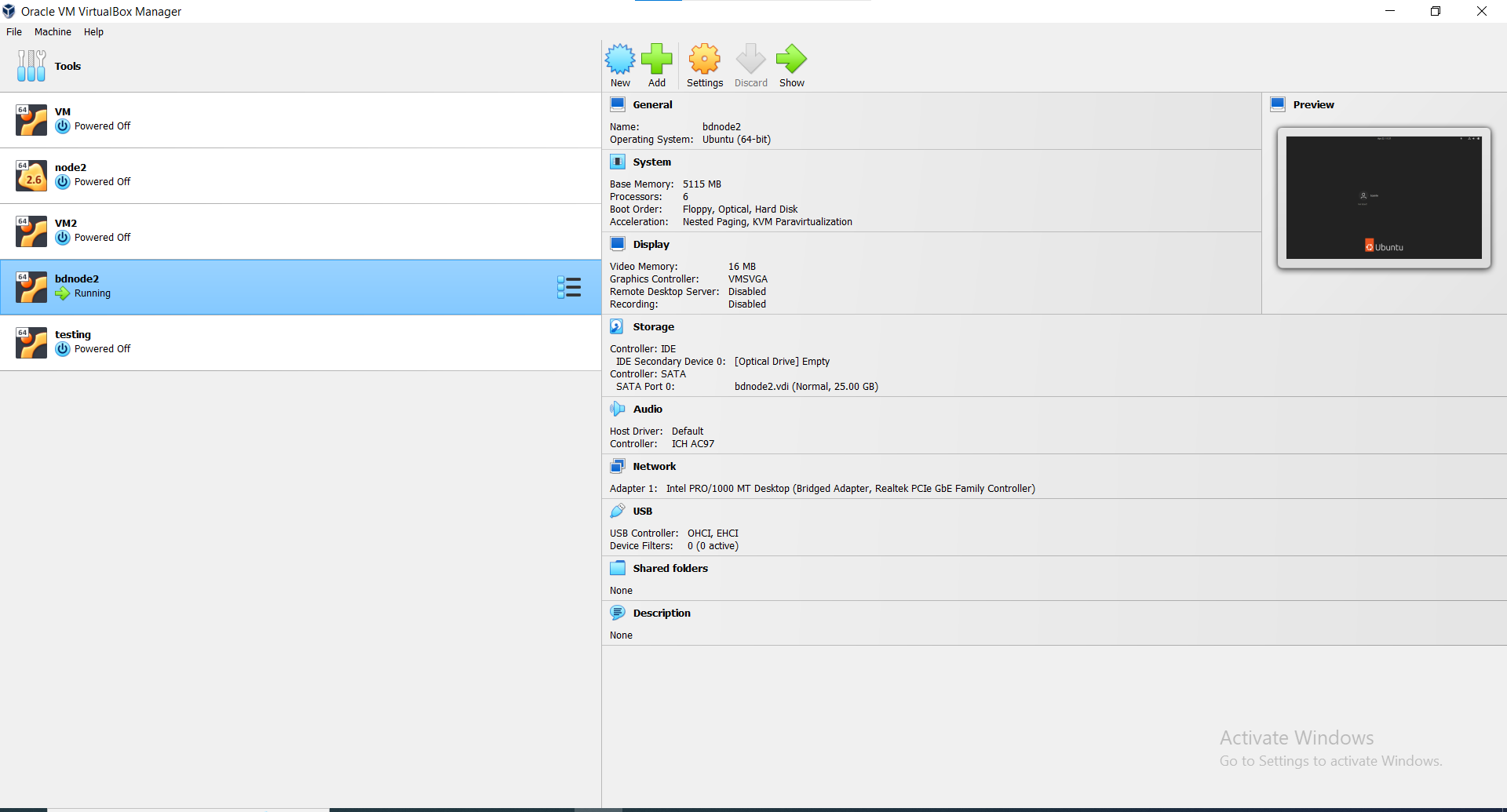
Cả hai máy ảo được cài đặt trên 2 máy vật lý khác nhau này điều được Attached Network Bridged Adapter và sử dụng chung đường mạng Ethernet với Host OS. Cách cấu hình dễ nhất ở môi trường thử nghiệm và nghiên cứu.

Trong môi trường trên node chính là máy ảo chạy trên máy tính để bàn tên là VM với thông số như sau:



**Hình 4.4: Cấu hình VM máy thứ nhất**

và node phụ là máy ảo chạy trên laptop tên là bdnode2 với thông số như sau:



**Hình 4.5: Cấu hình VM máy thứ hai.**

Tổng phần cứng 12 Core Intel Core I5 11900k và 16GB RAM trên hai máy ảo được phân phối.

Sau khi thực hiện cài đặt xong thì ta thử Ping kết nối giữa hai máy ảo với nhau sau đó tiến hành cài đặt OpenServer SSH cho các máy ảo theo Link cài đặt sau (<https://ubuntu.com/server/docs/service-openssh>) một số ghi chép về SSH là gì (<https://github.com/hoangdh/ghichep-SSH>) và tiến hành tạo kết nối SSH giữa máy ảo trên 2 máy tính vật lý khác nhau [7]:

Node chính sẽ sử dụng kết nối ssh để kết nối với các nút khác trong cụm bằng xác thực cặp khóa, để quản lý cụm. do đó hãy tạo cặp khóa bằng

ssh-keygen -t rsa -P '' -f ~/.ssh/id\_rsa

Lệnh ssh-keygen tạo các tập tin bên dưới bao gồm một Private Key và một Public Key.

toanle@192.168.1.186:~$ ls -lrt .ssh/

-rw-r--r-- 1 ubuntu ubuntu 397 Dec 9 00:17 id\_rsa.pub

-rw------- 1 ubuntu ubuntu 1679 Dec 9 00:17 id\_rsa

Sao chép id\_rsa.pub vào ủy authorized\_keys trong thư mục ~/.ssh. Bằng cách sử dụng >> nó sẽ nối thêm nội dung của tệp id\_rsa.pub vào ủy quyền

cat .ssh/id\_rsa.pub >> ~/.ssh/authorized\_keys

Bây giờ hãy sao chép ủy quyền vào tất cả các nút dữ liệu trong một cụm(1 nút).

scp .ssh/authorized\_keys toanle@192.168.1.69:/home/ubuntu/.ssh/authorized\_keys

Sau khi thực hiện tạo và sao chép khóa công khai qua các node phụ thì việc kết nối SSH đã được các node phụ hoàn toàn ủy quyền. Sau khi tạo giao tiếp giữa các máy xong thì ta tiến hành cài đặt môi trường Docker cho các máy ảo theo thứ tự như sau:

Link cài đặt Docker trên máy ảo Linux trong môi trường nghiên cứu: (<https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04>)

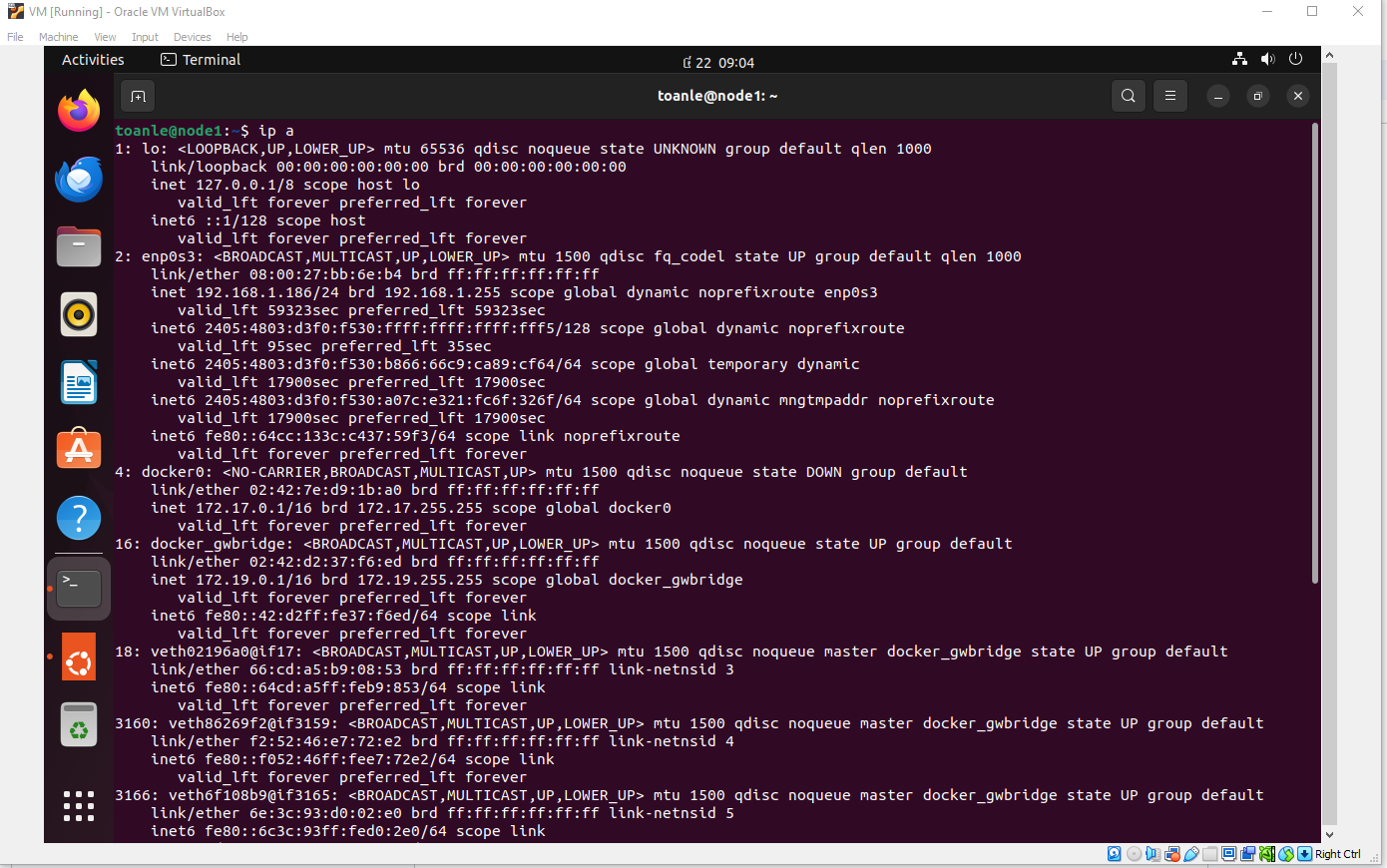
Link cài đặt Docker Compose trong môi trường nghiên cứu:

(<https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04>)

Link cài đặt Docker Swarm trên máy ảo Linux trong môi trường nghiên cứu: (<https://www.linuxtechi.com/how-to-deploy-docker-swarm-on-ubuntu/>)

### **4.2.4 Tạo Cluster Docker Swarm và thêm các Node vào Cluster**

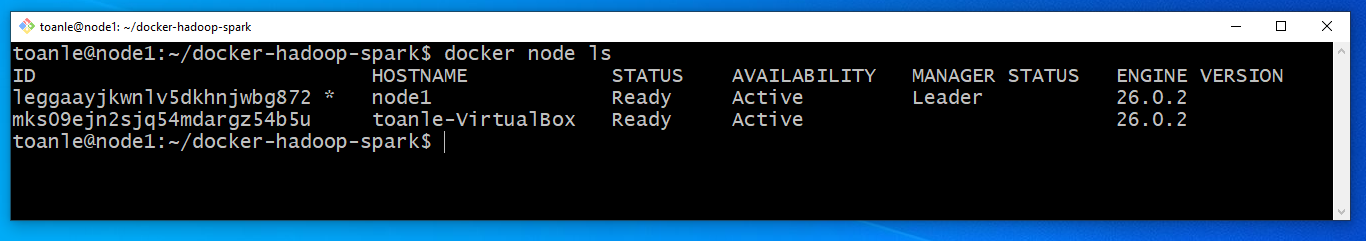
Ta tiến hành tạo Cluster Docker Swarm cho bài luận văn nghiên cứu, bài nghiên cứu chỉ bao gồm 2 nút trong đó 1 nút là Manager và 1 nút là Worker trên 2 máy ảo ở hai phần cứng vật lý khác nhau. Tiến hành kiểm tra ip của máy ảo chính trong môi trường nghiên cứu này là 192.168.1.186



**Hình 4.6: Mạng kết nối.**

Tiến hành tạo một cụm Docker Swarm:

* Đứng ở Node quản lý khi đã được cài đặt đủ Docker, Docker Compose, Docker Swarm ta thực hiện lệnh docker swarm init --advertise-addr 192.168.1.186 để khởi tạo một cụm Swarm.
* Sau khi hoàn tất khởi tạo thì ta sẽ nhận về một mã gia nhập vào Docker Swarm tương ứng ( ví dụ: docker swarm join --token SWMTKN-1-1ui7odb6pxhbf7nna2x2smmcjhreef08bl2l5y8647ltdszdeu-dqw9t4i49240ehimzgi5ku0nu 192.168.1.186:2377).
* Ta tiến hành đứng ở Node quản lý sau khi tạo thành công thì thực hiện kết nối SSH tới nút phụ (ssh toanle@192.168.1.69) và gõ lại mã gia nhập đã note ở trên để tiến hành gia nhập vào node quản lý. Sau khi gia nhập xong ta thoát khỏi Node công nhân và kiểm tra.



**Hình 4.7: Cụm Swarm trong thí nghiệm.**

Sau khi gia nhập thành công thì ta tiến hành Tạo file điều phối Docker Compose YAML(với các Container trong phạm vi nghiên cứu) và thực hiện quá trình Deploy Docker Compose lên môi trường Docker Swarm.

### **4.2.5 Chạy ứng dụng**

Các ứng dụng trong phạm vi nghiên cứu bao gồm:

* Airflow: Công cụ lập lịch.
* Hadoop HDFS: Sử dụng để lưu trữ dữ liệu được xem như là Data Lake.
* Spark: Công cụ xử lý dữ liệu lớn cốt lõi.
* Hive: Được xem như là Data Warehouse của Hadoop HDFS.
* Superset: Hiển thị trực quan dữ liệu bằng các bảng dashboards thông qua kết nối tới Hive.

Sau khi thiết lập môi trường hoàn tất và tải xuống mã nguồn cài đặt và mã nguồn xử lý thì ta tiến hành chạy docker-compose trên cụm swarm như sau.

Triển khai Stack với Docker Swarm:

* docker stack deploy -c docker-compose.yml [tên dịch vụ]

Kiểm tra trạng thái các dịch vụ:

* docker stack services [tên dịch vụ]

Quản lý stack và các dịch vụ:

Để xem các container đang chạy trong một stack, sử dụng lệnh:

* docker stack ps [tên dịch vụ]

Để xóa stack có thể sử dụng lệnh:

* docker stack rm [tên dịch vụ]

Sau khi triển khai hoàn tất và trạng thái của các dịch vụ đều hoạt động bình thường thì ứng dụng sẽ tự động chạy các Spark Job bằng Airflow.

## **4.3 Kết quả thực nghiệm xử lý dữ liệu**

Sau khi dùng Spark để thực hiện xử lý dữ liệu từ tập dữ liệu chuyến bay trễ với phần cứng được mô tả như trên thời gian để xử lý phân tán trên HDFS(Data Lake) và lưu vào Apache Hive(Data warehouse) gồm các quá trình:

* Làm sạch dữ liệu: Thời gian thực hiện xấp xỉ 5 phút 32 giây, cho thấy quá trình làm sạch dữ liệu đã được tối ưu hóa đáng kể. Vì bộ dữ liệu đã được làm sạch và chỉ cần bỏ đi một số cột đã nêu ở trên để tiến hành lưu vào HDFS và Apache Hive. Quá trình này diễn tương dài vì trình ghi là dạng nén Parquet.
* Biến đổi, kiểm tra dữ liệu và lưu trữ: Mất 14 phút 57 giây, điều này cho thấy việc biến đổi và kiểm tra dữ liệu có thể đòi hỏi tài nguyên tính toán lớn hơn do các phép biến đổi phức tạp hơn. Trong đó việc lưu trữ dữ liệu tốn đến hơn 3 phút.
* Các ước lượng thời gian xấp xỉ thu được điều xuất phát từ việc đọc biểu đồ thời lượng thực hiện của từng Spark Job trên Apache Airflow.

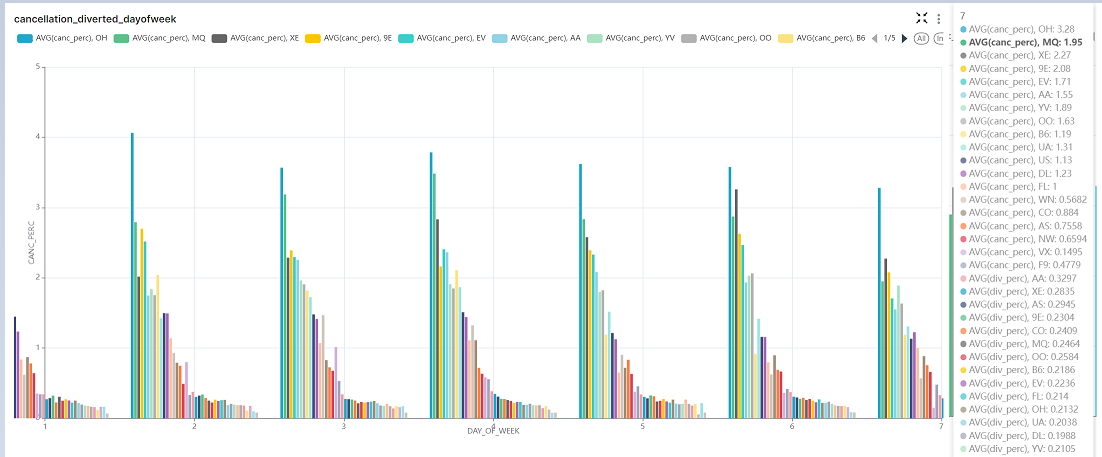
Lưu trữ và hiển thị dữ liệu:

* Dữ liệu sau khi xử lý được lưu trữ trong Apache Hive và HDFS, một hệ thống kho dữ liệu mạnh mẽ, hỗ trợ việc lưu trữ và truy vấn dữ liệu hiệu quả.
* Superset được sử dụng để hiển thị trực quan các dữ liệu dưới dạng bảng và biểu đồ. Điều này cho phép người dùng dễ dàng tương tác và phân tích dữ liệu một cách trực quan.

Hiệu suất hệ thống:

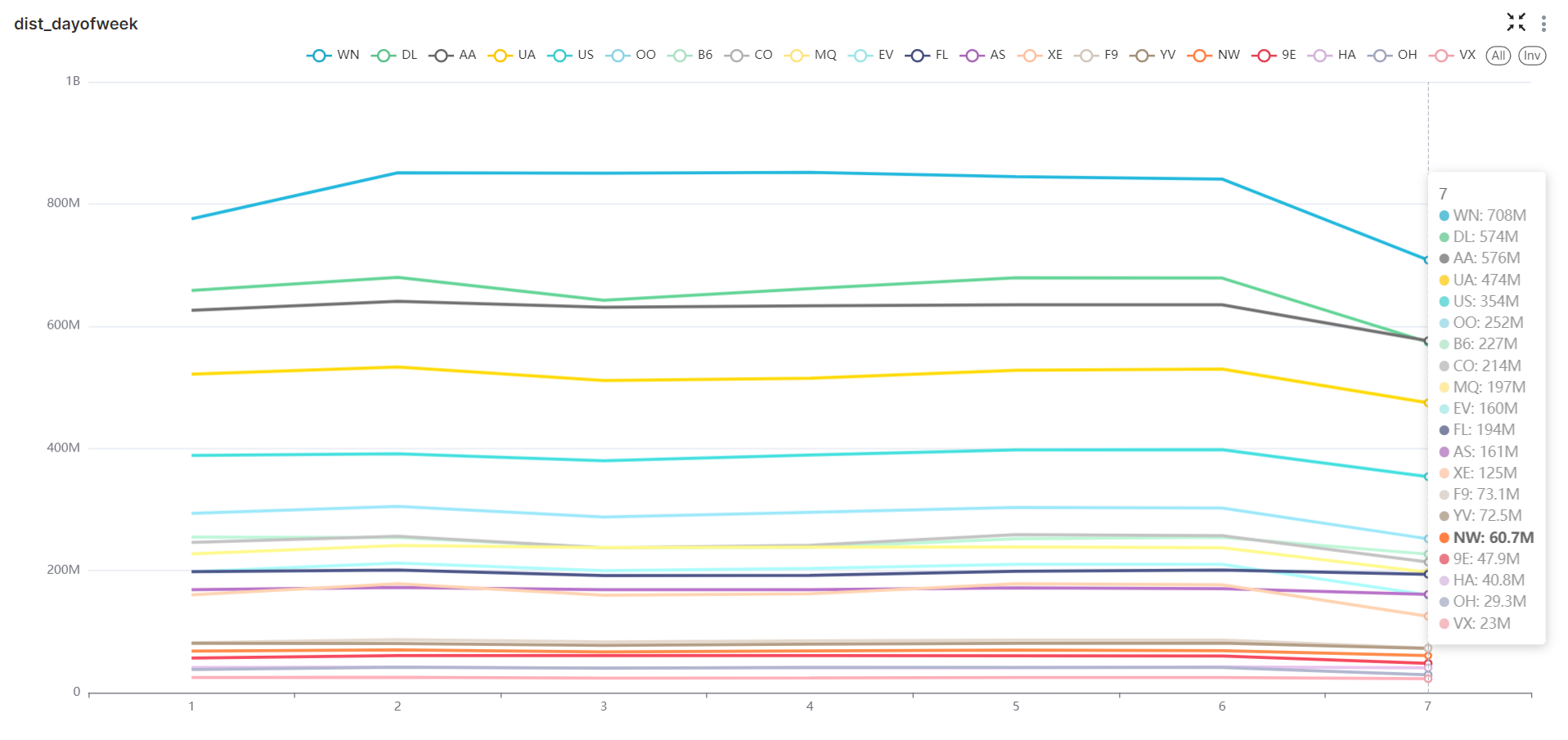
* Hệ thống được triển khai trên Docker Swarm cho phép tận dụng khả năng mở rộng, quản lý linh hoạt và cơ chế phục hồi tốt khi xảy ra lỗi chết máy đột ngột bởi kiến trúc linh hoạt của nó nhưng chỉ phù hợp cho môi trường cụm thử nghiệm hoặc các cụm tương đối, điều này được do chính cộng đồng Docker xác nhận.
* Việc sử dụng Spark cho thấy hiệu suất xử lý dữ liệu tốt trên nền tảng phân tán HDFS, đáp ứng được yêu cầu của các bài toán xử lý dữ liệu lớn.

Sau khi hoàn tất thì thu được một số kết quả tiêu biểu như những hình sau:



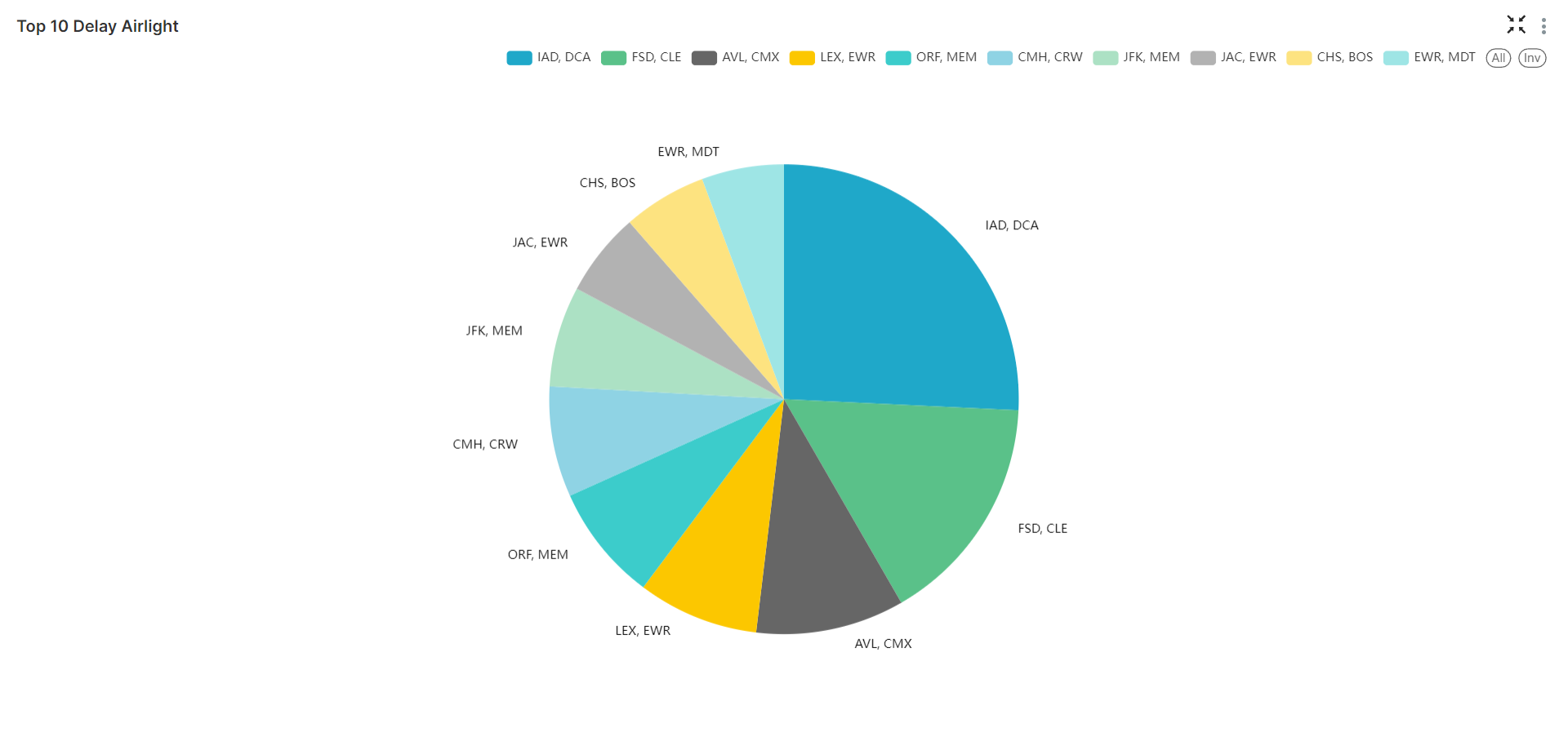
**Hình 4.8: Tỷ lệ hủy chuyến bay và chuyển chuyến bay của từng hãng hàng không trong các ngày trong tuần trên tập dữ liệu Airline Delay Analysis (2009-2013).**

Biểu đồ này giúp so sánh tỷ lệ phần trăm chuyến bay bị hủy hoặc chuyển hướng của các hãng hàng không khác nhau theo từng ngày trong tuần trong những năm từ 2009 đến 2013. Từ đó, có thể nhận ra các xu hướng, như hãng hàng không nào có tỷ lệ hủy chuyến hoặc chuyển hướng cao nhất vào những ngày cụ thể trong tuần.



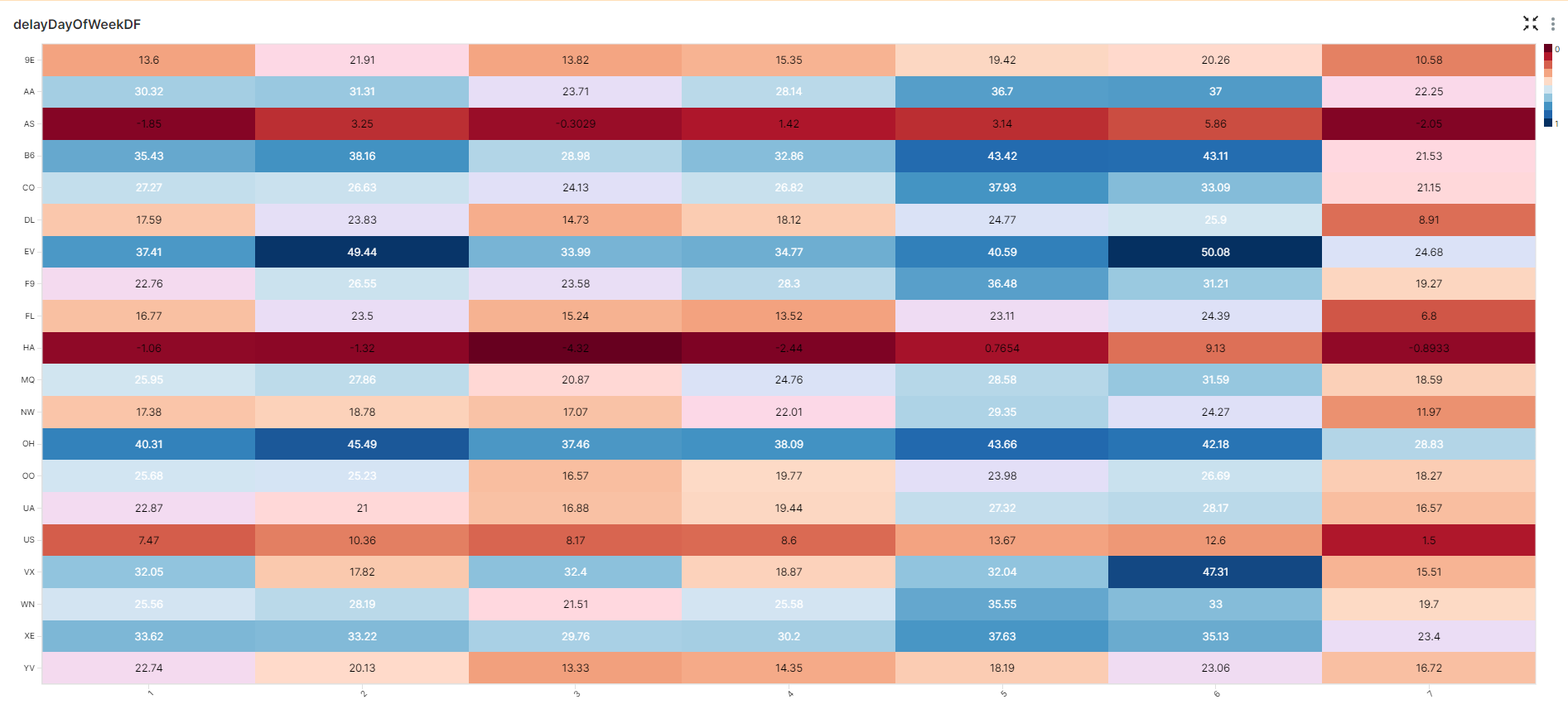
**Hình 4.9: Khoảng cách bay của từng hãng hàng không các ngày trong tuần trên tập dữ liệu Airline Delay Analysis (2009-2013).**

Biểu đồ so sánh tổng khoảng cách bay của các hãng hàng không khác nhau vào các ngày trong tuần trong những năm từ 2009 đến 2013. Điều này có thể giúp các hãng hàng không hiểu rõ hơn về hiệu suất của mình trong từng ngày cụ thể.



**Hình 4.10: Top 10 hãng hàng không muộn chuyến (2009-2013)**

Biểu đồ cho thấy danh sách đứng đầu của 10 hãng hàng không trễ chuyến nhiều nhất trong tập dữ liệu từ 2009 đến 2013. Trong đó hãng hàng không IAD trễ nhất trong tổng số 10 hãng hàng không khi so sánh tỷ lệ trễ chuyến với top 10 hãng trễ chuyến thì chiếm tỷ lệ trễ chuyến là 25,75%.



**Hình 4.11: Bảng phân bố thời gian muộn chuyến của hai 20 hãng hàng không trong từng ngày trong tuần trên tập dữ liệu Airline Delay Analysis (2009-2013).**

Dữ liệu giúp trực quan hóa phát hiện những hãng hàng không trễ chuyến dựa vào mẫu dữ liệu màu sắc được trải dài từ 0 đến 1 như trong hình. Từ hình ảnh có thể nhận biết được xu hướng hãng hàng không ít trễ chuyến ví dụ như trên hình có thể thấy hãng hàng không AS với phân vùng màu sắc thiên hướng đỏ đậm chứng tỏ hãng rất ít trễ chuyến với khách hàng.

Kết quả thực nghiệm cho thấy hệ thống xử lý dữ liệu lớn dựa trên Hadoop HDFS và Spark hoạt động hiệu quả trong môi trường Docker Swarm. Quá trình làm sạch và biến đổi dữ liệu diễn ra nhanh chóng, và dữ liệu được lưu trữ và hiển thị một cách trực quan bằng Apache Hive và Superset. Hệ thống này không chỉ đảm bảo hiệu suất xử lý cao mà còn mang lại tính linh hoạt và khả năng mở rộng, phù hợp cho các ứng dụng xử lý dữ liệu lớn phức tạp.

# **CHƯƠNG 5: KẾT LUẬN**

## **5.1. Kết quả nghiên cứu của đề tài**

Trong nghiên cứu về lưu trữ và xử lý dữ liệu lớn trên một tập dữ liệu sẵn có. Kết quả nghiên cứu thí nghiệm trong luận văn là cơ sở nền tảng của lưu trữ và xử lý dữ liệu lớn hàng loạt và cũng đồng thời làm rõ được một quy trình xử lý dữ liệu lớn cơ bản sẽ được diễn ra như thế nào như lập lịch, lưu trữ, xử lý, trình bày trực quan hóa dữ liệu.

## **5.2. Hạn chế luận văn**

Hạn Chế về Dữ liệu: Nghiên cứu này chỉ làm việc trên một tập dữ liệu có sẵn, do đó, mẫu dữ liệu có thể chưa phản ánh chính xác các vấn đề đặt ra và chưa định hình đúng về một bài toán lớn cụ thể. Điều này dẫn đến hạn chế về tính toàn diện và khả năng ứng dụng của kết quả nghiên cứu.

Hạn Chế về Triển Khai: Việc xử lý và lưu trữ dữ liệu lớn yêu cầu sự triển khai trên các hệ thống có tính mở rộng cao như K8S hoặc các hệ thống lưu trữ và xử lý trên đám mây. Tuy nhiên chưa thể học tập và triển khai đầy đủ các giải pháp này, ảnh hưởng đến khả năng mở rộng và hiệu quả của quá trình xử lý dữ liệu.

Hạn Chế về Ngôn Ngữ Lập Trình: Mặc dù Spark Job thường được viết bằng Scala hoặc Python, cuốn sách dạy về Spark chỉ chủ yếu tập trung viết mã bằng Scala và Python để tận dụng tối đa các thư viện và tính năng hỗ trợ, trong nghiên cứu này, mã nguồn chỉ được viết bằng Java. Điều này có thể ảnh hưởng khả năng tương thích với các công cụ và thư viện hiện có trong hệ sinh thái xử lý dữ liệu lớn.

## **5.3. Hướng phát triển tiếp theo của đề tài nghiên cứu**

Tiếp tục nghiên cứu triển khai tích hợp trên K8S, viết cơ sở hạ tầng mã nguồn để đưa lên đám mây, nghiên cứu thêm các ứng dụng xử lý và lưu trữ trên đám mây lớn khác. Thực hiện một số đánh giá chi phí và hiệu năng hệ thống.

Nghiên cứu thêm về những bài toán thực tế khác về các Layer khác của Spark công cụ cốt lõi của xử lý dữ liệu lớn.

# **TÀI LIỆU THAM KHẢO**

[1]. Ristea, A. L., & Ciucu, A. V. (2016). Big Data in Healthcare: Challenges and Opportunities. Journal of Big Data, Vol. 3, Issue 2, pp. 45-58.

[2]. McAfee, A., & Brynjolfsson, E. (2018). The Role of Big Data in Finance. International Journal of Financial Studies, Vol. 6, Issue 3, pp. 34-47.

[3]. Ralph Kimball, Margy Ross (2002). The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, 2nd Edition. John Wiley & Sons, Indianapolis.

[4]. Bill Chambers, Matei Zaharia (2018). Spark: The Definitive Guide. O'Reilly Media, Sebastopol.

[5]. Tom White (2015). Hadoop: The Definitive Guide, 4th Edition. O'Reilly Media, Sebastopol.

[6]. Docker, Inc. (2015). Docker: Up & Running. O'Reilly Media, Sebastopol.

[7]. Spark By Examples (n.d.). Apache Hadoop Installation. Truy cập tại<https://sparkbyexamples.com/hadoop/apache-hadoop-installation/>. (truy cập lần cuối 5/26/2024)

[8]. Viện Thống Kê (2016). Báo cáo thống kê năm 2016. Truy cập tại<https://vienthongke.vn/wp-content/uploads/2021/04/Bai4.So5_.2016.pdf>. (truy cập lần cuối 5/26/2024)

[9] IDC Blogs. (n.d.). Daily Data Generation. Retrieved from <https://www.idc.com/blogs/daily-data> (truy cập lần cuối 5/26/2024)

[10] Wikipedia. (n.d.). Big Data. Retrieved from <https://en.wikipedia.org/wiki/Big_data> (truy cập lần cuối 5/26/2024)

[11] Gartner. (n.d.). Definition of Big Data. Retrieved from <https://www.gartner.com/en/research/big-data> (truy cập lần cuối 5/26/2024)

[12] Data Lake and Data Warehouse diffrenent <https://aws.amazon.com/vi/compare/the-difference-between-a-data-warehouse-data-lake-and-data-mart/> (truy cập lần cuối 5/26/2024)