Báo cáo BigData

# 1. BigData

## 1.1 Khái niệm

Big Data (dữ liệu lớn) là một khái niệm được sử dụng để mô tả các tập dữ liệu có kích thước lớn, phức tạp và đa dạng mà các công cụ và phương pháp truyền thống không thể xử lý hiệu quả.

## 1.2 Đặc trưng

Đặc trưng của Big Data là 3V:

+ Volume (lượng dữ liệu lớn)

+ Velocity (tốc độ xử lý nhanh)

+ Variety (đa dạng dữ liệu)

+ Veracity (tính xác thực)

+ Value (giá trị)

## 1.3 Nguồn dữ liệu

Lượng dữ liệu lớn trong Big Data có thể đến từ nhiều nguồn khác nhau như hệ thống giao dịch, mạng xã hội, thiết bị IoT (Internet of Things) và các nguồn dữ liệu khác. Các công nghệ Big Data được sử dụng để xử lý, lưu trữ và phân tích dữ liệu này nhằm tìm ra thông tin hữu ích, mô hình dự đoán và các hiểu biết mới.

## 1.4 Công nghệ sử dụng

Trong quá trình xử lý dữ liệu lớn, các công nghệ Big Data như Hadoop, Spark và công nghệ cơ sở dữ liệu phi cấu trúc NoSQL (Not Only SQL) thường được sử dụng.

+ Hadoop là một framework mã nguồn mở được sử dụng để lưu trữ và xử lý dữ liệu lớn trên các cụm máy tính.

+ Spark là một hệ sinh thái dữ liệu lớn nhanh chóng và linh hoạt, cho phép thực hiện xử lý và phân tích dữ liệu thời gian thực.

+ Công nghệ cơ sở dữ liệu NoSQL cho phép lưu trữ và truy xuất dữ liệu phi cấu trúc, linh hoạt và có thể mở rộng.

## 1.5 Đánh giá

Big Data mang lại nhiều lợi ích, bao gồm khả năng tìm kiếm thông tin, phân tích dữ liệu và hiểu biết sâu sắc hơn về các mô hình và xu hướng. Nó đã có ứng dụng rộng rãi trong nhiều lĩnh vực, bao gồm kinh doanh, khoa học, y tế, tài chính, xã hội học và quản lý đô thị. Tận dụng Big Data có thể giúp các tổ chức đưa ra quyết định thông minh, tối ưu hóa hoạt động và tạo ra giá trị mới.

# 2. Hadoop ecosystem

## 2.1 Khái niệm Hadoop

Apache Hadoop là một framework cho phép xử lý phân tán các tập dữ liệu lớn trên các cụm máy tính sử dụng mô hình lập trình đơn giản. Nó được thiết kế để mở rộng từ máy chủ đơn lên hàng ngàn máy tính, mỗi máy tính đều cung cấp tính toán và lưu trữ cục bộ. Thay vì phụ thuộc vào phần cứng để đạt được khả năng sẵn có cao, thư viện này được thiết kế để phát hiện và xử lý sự cố ở tầng ứng dụng, đem lại dịch vụ có sẵn lúc nào trên một cụm máy tính, mà mỗi máy tính có thể gặp các sự cố.

## 2.2 Hadoop ecosystem

### 2.2.1 Công cụ tích hợp Hadoop

Nhiều công cụ liên quan tích hợp với Hadoop.

• Phân tích dữ liệu

• Tích hợp cơ sở dữ liệu

• Quản lý luồng công việc (workflow management)

Những công cụ này không được xem là 'Hadoop cốt lõi' (core Hadoop). Thay vào đó, chúng thuộc phạm vi của 'Hệ sinh thái Hadoop' (Hadoop ecosystem).

Nhiều công cụ này cũng là các dự án mã nguồn mở của Apache.

### 2.2.2 Apache Pig

Apache Pig xây dựng trên nền tảng Hadoop để cung cấp xử lý dữ liệu cấp cao.

• Pig đặc biệt tốt trong việc kết hợp và biến đổi dữ liệu.

Trình thông dịch Pig chạy trên máy khách (client machine).

• Biến đổi các kịch bản PigLatin thành các công việc MapReduce.

• Gửi những công việc đó tới cụm (cluster).

### 2.2.3 Apache Hive

Một sự trừu tượng khác phía trên MapReduce

• Giảm thời gian phát triển

• HiveQL: Ngôn ngữ giống SQL

Trình thông dịch Hive chạy trên máy khách (client machine)

• Biến đổi các kịch bản HiveQL thành các công việc MapReduce

• Gửi những công việc đó tới cụm (cluster)

### 2.2.4 Apache Hbase

HBase là một hệ thống lưu trữ dữ liệu phân tán hướng cột được xây dựng trên nền tảng HDFS.

• Được coi là cơ sở dữ liệu Hadoop.

Dữ liệu được tổ chức logic thành các bảng, hàng và cột.

• Có thể lưu trữ hàng terabytes và thậm chí petabytes dữ liệu trong một bảng.

• Các bảng có thể có hàng ngàn cột.

HBase có khả năng mở rộng để cung cấp lưu lượng ghi rất cao.

• Có thể thực hiện hàng trăm nghìn lệnh chèn (insert) mỗi giây.

Tương đối nguyên thủy so với RDBMS (Hệ quản trị cơ sở dữ liệu quan hệ).

• Là một hệ thống NoSQL: Không có ngôn ngữ truy vấn cấp cao.

• Sử dụng API để quét (scan) / lấy (get) / đặt (put) giá trị dựa trên các khóa.

### 2.2.5 Apache sqoop

Sqoop là một công cụ được thiết kế để chuyển dữ liệu hàng loạt giữa Apache Hadoop và các cơ sở dữ liệu có cấu trúc như cơ sở dữ liệu quan hệ.

Nó có thể nhập tất cả các bảng, một bảng duy nhất hoặc một phần của một bảng vào HDFS (Hadoop Distributed File System).

• Thông qua một tác vụ MapReduce hoặc chỉ Map.

• Kết quả là một thư mục trong HDFS chứa các tệp văn bản được phân cách bằng dấu phẩy (comma-delimited).

Sqoop cũng có thể xuất dữ liệu từ HDFS trở lại cơ sở dữ liệu.

### 2.2.6 Apache Oozie

Oozie là một hệ thống lập lịch luồng công việc (workflow scheduler) để quản lý các công việc Apache Hadoop.

Các công việc luồng công việc Oozie là các Đồ Thị Hướng Không Cyclic (Directed Acyclic Graphs - DAGs) của các hành động (actions).

Oozie hỗ trợ nhiều hành động trong luồng công việc, bao gồm:

• Thực thi các công việc MapReduce

• Chạy các tập lệnh Pig hoặc Hive

• Thực thi các chương trình Java hoặc lệnh shell tiêu chuẩn

• Xử lý dữ liệu thông qua các lệnh HDFS

• Chạy các lệnh từ xa bằng SSH

• Gửi các tin nhắn e-mail.

### 2.2.7 Apache Zookeeper

Apache ZooKeeper là một dịch vụ phân phối phối hợp đáng tin cậy cao.

• Thành viên nhóm

• Bầu cử người lãnh đạo

• Cấu hình động

• Giám sát trạng thái

Tất cả các loại dịch vụ này được sử dụng dưới một hình thức nào đó bởi các ứng dụng phân phối.

# 3. HDFS

## 3.1 Tổng quan về HDFS

Cung cấp lưu trữ giá rẻ và đáng tin cậy cho lượng dữ liệu khổng lồ, Thiết kế cho

• Các tệp lớn (kích thước tệp từ 100 MB đến một số TB)

• Viết một lần, đọc nhiều lần (chỉ ghi thêm dữ liệu)

• Chạy trên phần cứng thông dụng (commodity hardware)

Hệ thống tệp theo cấu trúc phân cấp kiểu UNIX

• (ví dụ, /hust/soict/hello.txt)

• Quyền sở hữu và quyền truy cập tệp kiểu UNIX

## 3.2 Nguyên tắc thiết kế chính của HDFS

Mô hình I/O (Input/Output)

• Chỉ ghi thêm dữ liệu (append only) → giảm đồng bộ hóa

Phân phối dữ liệu

• Tệp được chia thành các phần lớn (64 MB)

→ giảm kích thước dữ liệu siêu dữ liệu (metadata)

→ giảm giao tiếp mạng

Sao chép dữ liệu

• Mỗi phần dữ liệu thường được sao chép vào 3 nút khác nhau

Khả năng chịu lỗi

• Data Node: sao chép lại dữ liệu

• Name Node

• Secondary Namenode

• Standby, Active Namenodes

## 3.3 Kiến trúc HDFS

• Kiến trúc Master/slave

• HDFS master: Namenode

• Quản lý không gian tên và siêu dữ liệu (metadata)

• Giám sát Datanode

• HDFS slaves: Datanodes

• Xử lý đọc/ghi dữ liệu thực tế {chunks}

• Các chunk là các tệp cục bộ trong các hệ thống tệp cục bộ (local file systems)

### 3.3.1 Các chức năng của Namenode

• Quản lý không gian tên Hệ thống Tệp

• Ánh xạ một tên tệp thành một tập hợp các khối (blocks)

• Ánh xạ một khối thành các Datanodes mà khối đó nằm trên đó

• Quản lý cấu hình cụm (Cluster Configuration Management)

• Bộ máy sao chép (Replication Engine) cho các khối

### 3.3.2 Metadata (Namenode)

• Metadata trong bộ nhớ chính (main memory)

• Toàn bộ metadata nằm trong bộ nhớ chính

• Không sử dụng trang yêu cầu (demand paging) cho metadata

• Các loại metadata

• Danh sách các tệp

• Danh sách các khối cho mỗi tệp

• Danh sách các DataNodes cho mỗi khối

• Thuộc tính của tệp, ví dụ: thời gian tạo, yếu tố sao chép

• Một bản ghi giao dịch (Transaction Log)

• Ghi lại các tạo tệp, xóa tệp, v.v.

### 3.3.3 Datanode

• Một máy chủ khối (Block Server)

• Lưu trữ dữ liệu trong hệ thống tệp cục bộ (ví dụ: ext3)

• Lưu trữ metadata của một khối (ví dụ: CRC)

• Cung cấp dữ liệu và metadata cho các máy khách (Clients)

• Báo cáo khối (Block Report)

• Định kỳ gửi báo cáo về tất cả các khối hiện có tới Namenode

• Hỗ trợ việc liên kết dữ liệu (Pipelining of Data)

• Chuyển tiếp dữ liệu đến các Datanode được chỉ định khác

• Tín hiệu sinh lực (Heartbeat)

• Datanode gửi tín hiệu (Heartbeat) tới Namenode mỗi 3 giây một lần

• Namenode sử dụng Heartbeats để phát hiện sự cố với Datanode

## 3.4 Các hoạt động cơ bản

### 3.4.4 Sao chép dữ liệu (Data replication)

• Vị trí đặt các phần dữ liệu (chunks)

• Chiến lược hiện tại

• Một bản sao trên nút cục bộ (local node)

• Bản sao thứ hai trên một khay cách xa (remote rack)

• Bản sao thứ ba trên cùng một khay cách xa

• Các bản sao bổ sung được đặt ngẫu nhiên

• Các máy khách (Clients) đọc từ bản sao gần nhất

• Namenode phát hiện sự cố của Datanode

• Chọn Datanode mới cho các bản sao mới

• Cân bằng sử dụng đĩa

• Cân bằng lưu lượng giao tiếp đến các Datanode

### 3.4.5 Cân bằng dữ liệu (Data rebalance)

• Mục tiêu: Tỷ lệ % đầy đủ đĩa trên các Datanode nên tương tự nhau

• Thường chạy khi thêm Datanode mới

• Cụm đang hoạt động khi Cân bằng dữ liệu (Rebalancer) hoạt động

• Cân bằng dữ liệu (Rebalancer) được giới hạn để tránh quá tải mạng

• Công cụ dòng lệnh

### 3.4.6 Độ chính xác dữ liệu

• Sử dụng checksums để xác nhận dữ liệu

• Sử dụng CRC32

• Tạo tệp

• Máy khách tính toán checksum cho mỗi 512 byte

• Datanode lưu trữ checksum

• Truy cập tệp

• Máy khách lấy dữ liệu và checksum từ Node dữ liệu

• Nếu xác minh không thành công, máy khách thử các bản sao khác.

### 3.4.7 Liên kết dữ liệu (Data pipelining)

• Máy khách lấy danh sách các Datanode để đặt các bản sao của một khối

• Máy khách ghi khối vào Datanode đầu tiên

• Datanode đầu tiên chuyển tiếp dữ liệu đến Node tiếp theo trong Pipeline

• Khi tất cả các bản sao được ghi, Máy khách chuyển sang ghi khối tiếp theo trong tệp.

## 3.5 Các khái niệm nâng cao

### 3.5.1 Secondary Name node

• Namenode là một điểm đơn lẻ gây ra sự cố.

• Secondary Name node

• Thực hiện sao lưu (checkpointing) bản sao mới nhất của tập tin FsImage (hình ảnh trạng thái của hệ thống tệp HDFS) và các tệp nhật ký giao dịch (Transaction Log).

• Sao chép FsImage và Transaction Log từ Namenode sang một thư mục tạm thời.

• Khi Namenode được khởi động lại

• Kết hợp FSImage và Transaction Log thành một FSImage mới trong thư mục tạm thời.

• Tải lên FSImage mới lên Namenode

• Nhật ký giao dịch trên Namenode được xoá đi.

### 3.5.2 Namenode high availability (HA)

Nền tảng

Trước khi Hadoop 2.0.0, NameNode là một điểm đơn lẻ gây ra sự cố (Single Point of Failure - SPOF) trong một cụm HDFS. Mỗi cụm có một NameNode duy nhất, và nếu máy chủ hoặc quy trình đó trở nên không khả dụng, cụm HDFS sẽ không thể truy cập dữ liệu cho đến khi NameNode được khởi động lại hoặc được triển khai trên một máy chủ khác.

Điều này ảnh hưởng đến sự sẵn có tổng thể của cụm HDFS theo hai cách chính:

1. Trong trường hợp xảy ra sự kiện không được lên kế hoạch như máy chủ bị sập, cụm sẽ không khả dụng cho đến khi người quản trị khởi động lại NameNode.
2. Các sự kiện bảo trì được lên kế hoạch như nâng cấp phần mềm hoặc phần cứng trên máy chủ NameNode sẽ dẫn đến thời gian không hoạt động của cụm.

Tính năng High Availability (HA) của HDFS giải quyết các vấn đề trên bằng cách cung cấp tùy chọn chạy hai (hoặc nhiều hơn, bắt đầu từ Hadoop 3.0.0) NameNodes dự phòng trong cùng một cụm, được cấu hình theo kiểu Active/Passive với một hoặc nhiều bản sao dự phòng nóng (hot standby). Điều này cho phép thay đổi nhanh chóng sang NameNode mới trong trường hợp máy chủ sập, hoặc thay đổi dự phòng do người quản trị khởi tạo một cách đều đặn để thực hiện bảo trì lên kế hoạch.

Kiến trúc

Kiến trúc Trong một cụm HA tiêu chuẩn, hai hoặc nhiều máy riêng biệt được cấu hình như các NameNode. Tại bất kỳ thời điểm nào, chính xác một trong số các NameNode được ở trong trạng thái Active, và các NameNode còn lại ở trạng thái Standby. NameNode Active chịu trách nhiệm cho tất cả các hoạt động của máy khách trong cụm, trong khi Standby đơn giản chỉ đóng vai trò như một slave, duy trì đủ trạng thái để cung cấp chuyển đổi nhanh nếu cần thiết.

Để các nút Standby giữ trạng thái của họ được đồng bộ với nút Active, hiện thực hiện tại yêu cầu các nút có quyền truy cập vào một thư mục trên một thiết bị lưu trữ chung (ví dụ: một gắn kết NFS từ NAS). Ràng buộc này có thể được thả lỏng trong các phiên bản tương lai.

Khi bất kỳ sửa đổi không gian tên nào được thực hiện bởi nút Active, nó sẽ ghi nhật ký bền vững của sửa đổi vào một tệp nhật ký chỉnh sửa được lưu trữ trong thư mục chia sẻ. Các nút Standby liên tục theo dõi thư mục này để theo dõi các chỉnh sửa và khi nhận thấy các chỉnh sửa, chúng áp dụng chúng vào không gian tên của riêng mình. Trong trường hợp chuyển đổi nhanh chóng, các nút Standby sẽ đảm bảo rằng chúng đã đọc tất cả các chỉnh sửa từ bộ nhớ lưu trữ chia sẻ trước khi thăng cấp thành trạng thái Active. Điều này đảm bảo rằng trạng thái không gian tên được đồng bộ hoàn toàn trước khi chuyển đổi nhanh chóng xảy ra.

Để cung cấp chuyển đổi nhanh chóng, cũng cần thiết rằng các nút Standby có thông tin cập nhật về vị trí các khối trong cụm. Để thực hiện điều này, các DataNode được cấu hình với vị trí của tất cả các NameNode và gửi thông tin vị trí khối và nhịp tim đến tất cả các NameNode.

Điều quan trọng cho hoạt động đúng đắn của một cụm HA là chỉ có một trong số các NameNode được Active vào một thời điểm. Nếu không, trạng thái không gian tên sẽ nhanh chóng lệch nhau giữa hai nút, gây nguy cơ mất dữ liệu hoặc kết quả không đúng đắn khác. Để đảm bảo tính chất này và ngăn chặn tình huống "split-brain", người quản trị phải cấu hình ít nhất một phương thức fencing cho lưu trữ chia sẻ. Trong quá trình chuyển đổi nhanh chóng, nếu không thể xác minh rằng nút Active trước đó đã từ bỏ trạng thái Active của mình, quá trình fencing có trách nhiệm cắt đứt quyền truy cập của nút Active trước đó vào lưu trữ chỉnh sửa chia sẻ. Điều này ngăn nó tiếp tục thực hiện bất kỳ chỉnh sửa nào vào không gian tên, cho phép nút Active mới tiếp tục chuyển đổi một cách an toàn.

### 3.5.3 JournalNode

JournalNode là một thành phần quan trọng trong hệ thống tệp phân tán Hadoop (HDFS). Nó đóng vai trò lưu trữ và sao lưu thông tin nhật ký chỉnh sửa (editlog) của HDFS. JournalNode giúp đảm bảo tính nhất quán và khả năng phục hồi của dữ liệu trong trường hợp xảy ra sự cố.

JournalNode hoạt động trong cụm HDFS và có thể được triển khai trên nhiều nút khác nhau để đảm bảo sự dư thừa và khả năng chịu lỗi. Khi Namenode ghi các chỉnh sửa vào editlog, JournalNode nhận và lưu trữ các thông tin này trên các nút JournalNode khác nhau. Việc lưu trữ thông tin nhật ký trên nhiều JournalNode đảm bảo tính sẵn sàng và an toàn của dữ liệu.

JournalNode cung cấp khả năng ghi nhật ký (journaling) cho HDFS để đảm bảo rằng tất cả các thay đổi dữ liệu được ghi lại một cách an toàn và nhất quán. Nó cũng cung cấp khả năng phục hồi dữ liệu trong trường hợp sự cố xảy ra. Khi một Namenode mới được triển khai, hoặc khi một Namenode chính gặp sự cố, JournalNode có thể dung lượng để phục hồi dữ liệu từ thông tin nhật ký.

Với JournalNode, tính sẵn sàng và tính nhất quán của HDFS được cải thiện. Nó là một phần quan trọng của kiến trúc của HDFS để đảm bảo rằng dữ liệu luôn được bảo vệ một cách tin cậy và có khả năng tái tạo trong trường hợp xảy ra sự cố.

### 3.5.4 ZooKeeper

ZooKeeper là một hệ thống phân tán mã nguồn mở được sử dụng để quản lý và giám sát các dịch vụ phân tán trong một môi trường phân tán. ZooKeeper thiết lập và duy trì một dịch vụ tên cung cấp đồng bộ, đồng thời đảm bảo tính nhất quán và tin cậy của các dịch vụ.

ZooKeeper chủ yếu hoạt động dựa trên nguyên lý "Zab" (ZooKeeper Atomic Broadcast). Nó sử dụng một tập hợp các quy tắc và giao thức để đồng bộ và đảm bảo tính nhất quán dữ liệu giữa các nút trong một cụm ZooKeeper.

Chức năng chính của ZooKeeper bao gồm:

1. Quản lý dịch vụ tên: ZooKeeper cung cấp một không gian tên phân tán để đăng ký và truy vấn các dịch vụ. Các ứng dụng có thể đăng ký và xóa bỏ các nút con trong không gian tên để thông báo về sự hiện diện và sẵn sàng của chúng.

2. Đồng bộ hóa dữ liệu: ZooKeeper đồng bộ hóa dữ liệu trên các nút trong cụm bằng cách sử dụng các giao thức đồng bộ và ghi nhật ký. Điều này đảm bảo rằng chỉ có một phiên bản dữ liệu được chấp nhận và tính nhất quán được duy trì.

3. Giám sát và phát hiện lỗi: ZooKeeper theo dõi trạng thái của các nút trong cụm và cung cấp các thông báo về lỗi và sự thay đổi trong cụm. Nó giúp xác định trạng thái lỗi và thực hiện các hành động khắc phục.

4. Khóa và tạo thứ tự: ZooKeeper cung cấp các cơ chế để tạo ra khóa phân tán và xây dựng thứ tự sự kiện trong môi trường phân tán. Điều này giúp đảm bảo tính xác thực và tuân thủ thứ tự trong một cụm phân tán.

ZooKeeper đã trở thành một thành phần quan trọng trong các hệ thống phân tán như Hadoop, Kafka và HBase. Nó giúp kiểm soát, quản lý và đảm bảo tính nhất quán của dữ liệu và dịch vụ phân tán.

### 3.5.5 HDFS HA Using QJM

HDFS High Availability Using the Quorum Journal Manager

NameNode HA với QJM (Quorum Journal Manager) là một phương pháp triển khai tính cao cấp cho NameNode trong môi trường Hadoop. Phương pháp này sử dụng JournalNodes để đảm bảo đồng bộ trạng thái giữa NameNode Active và NameNode Standby.

Khi có sự thay đổi trong không gian tên, NameNode Active ghi lại các sửa đổi vào JournalNodes. JournalNodes là một tập hợp các máy chủ được cấu hình để lưu trữ một bộ nhật ký chỉnh sửa chung. Nhật ký này chứa thông tin về các hoạt động thay đổi không gian tên được thực hiện.

NameNode Standby liên tục đồng bộ hóa và theo dõi JournalNodes để cập nhật trạng thái của mình. Nó đọc thông tin từ JournalNodes và áp dụng các sửa đổi vào không gian tên của mình, đảm bảo rằng trạng thái không gian tên của Standby luôn giống với Active.

Trong quá trình chuyển giao, khi NameNode Standby xác định rằng nó đã đồng bộ hóa đủ thông tin từ JournalNodes, nó có thể tự mình thăng cấp lên trạng thái Active. Điều này đảm bảo rằng không gian tên được đồng bộ hoàn toàn giữa Active và Standby trước khi chuyển giao xảy ra.

Phương pháp NameNode HA với QJM đảm bảo tính cao cấp và độ tin cậy của hệ thống Hadoop bằng cách duy trì việc ghi log và sao chép dữ liệu trạng thái giữa các NameNode, đảm bảo rằng không gian tên luôn đồng bộ và có sẵn cho các hoạt động của người dùng.

### 3.5.6 HDFS HA Using NFS

NameNode HA với NFS (Network File System) là một phương pháp triển khai tính cao cấp cho NameNode trong môi trường Hadoop. Phương pháp này sử dụng một lưu trữ chia sẻ thông qua NFS để đảm bảo đồng bộ trạng thái giữa NameNode Active và NameNode Standby.

Khi có sự thay đổi trong không gian tên, NameNode Active ghi lại các chỉnh sửa vào một tệp chỉnh sửa được lưu trữ trên thiết bị lưu trữ chia sẻ, ví dụ như NFS mount từ NAS. Cả hai NameNode liên tục theo dõi thư mục chia sẻ này để cập nhật trạng thái của mình. NameNode Standby đọc thông tin từ lưu trữ chia sẻ và áp dụng các chỉnh sửa vào không gian tên của mình, đảm bảo rằng trạng thái không gian tên của Standby luôn giống với Active.

Trong quá trình chuyển giao, khi NameNode Standby xác định rằng nó đã đọc đủ thông tin từ lưu trữ chia sẻ, nó có thể tự mình thăng cấp lên trạng thái Active. Điều này đảm bảo rằng không gian tên được đồng bộ hoàn toàn giữa Active và Standby trước khi chuyển giao xảy ra.

Phương pháp NameNode HA với NFS đảm bảo tính cao cấp và độ tin cậy của hệ thống Hadoop bằng cách sử dụng tệp chỉnh sửa chia sẻ và lưu trữ chia sẻ để đồng bộ trạng thái không gian tên giữa các NameNode. Điều này giúp đảm bảo rằng không gian tên luôn có sẵn và đồng bộ cho các hoạt động của người dùng.

### 3.5.7 Editlog

Editlog (nhật ký chỉnh sửa) là một thành phần quan trọng trong Hệ thống Tệp phân tán Hadoop (HDFS). Nó được sử dụng để ghi lại tất cả các thay đổi dữ liệu trong HDFS khi có sự chỉnh sửa xảy ra. Khi dữ liệu trong HDFS thay đổi, các thao tác chỉnh sửa đó sẽ được ghi vào editlog.

Editlog đóng vai trò quan trọng trong việc bảo đảm tính nhất quán của dữ liệu HDFS. Với editlog, ta có thể tái tạo lại tất cả các chỉnh sửa đã xảy ra trên hệ thống tệp. Nhờ vào editlog, trong trường hợp xảy ra sự cố như mất điểm truy cập vào dữ liệu hoặc mất mát dữ liệu, ta có thể khôi phục dữ liệu về trạng thái trước khi sự cố xảy ra.

Editlog thường được chỉnh sửa bằng cách ghi thêm các thao tác vào cuối tệp, giữ cho nó luôn cập nhật với các thay đổi gần nhất trên HDFS. Các thao tác chỉnh sửa được ghi là các ghi chú nhỏ giúp cho việc chỉnh sửa dữ liệu. Khi editlog trở nên quá lớn, ta có thể cắt nhỏ nó và tạo một phiên bản mới để tiếp tục ghi thêm các chỉnh sửa vào đó.

Editlog cùng với fsimage, được sử dụng để khôi phục lại trạng thái và đồ thị của HDFS. Tổng hợp cả hai thành phần này giúp HDFS đảm bảo tính nhất quán dữ liệu và khả năng phục hồi cao trong trường hợp xảy ra sự cố.

### 3.5.8 Standby Namenode

Standby Namenode (Namenode dự phòng) là một thành phần quan trọng trong Hệ thống Tệp phân tán Hadoop (HDFS). Standby Namenode hoạt động như một bản sao dự phòng của Namenode chính (Active Namenode). Nhiệm vụ chính của Standby Namenode là duy trì một bản sao của fsimage và nhận thông tin từ editlog.

Khi Namenode chính gặp sự cố hoặc cần được bảo trì, Standby Namenode có khả năng tiếp nhận vai trò hoạt động dự phòng. Việc chuyển giao từ Active Namenode sang Standby Namenode được xác định bằng cách sử dụng quy tắc "quyền ưu tiên". Standby Namenode sẽ lấy cấu trúc dữ liệu từ fsimage và nhận các chỉnh sửa từ editlog để đồng bộ và tái tạo lại trạng thái của HDFS.

Standby Namenode hoạt động liên tục để duy trì tính sẵn sàng và khả năng phục hồi của HDFS. Cấu trúc dữ liệu fsimage cần được cập nhật định kỳ để giữ cho Standby Namenode luôn được cập nhật với những thay đổi mới nhất. Khi namenode chính trở lại hoạt động, Standby Namenode sẽ trở lại hoạt động dự phòng và chờ đến khi một sự cố xảy ra tiếp theo.

Standby Namenode đóng vai trò quan trọng trong việc tăng cường tính sẵn sàng và tin cậy của HDFS. Với Standby Namenode, việc phục hồi dữ liệu trong trường hợp sự cố trở nên nhanh chóng và đảm bảo tính nhất quán của hệ thống tệp.

### 3.5.9 FS Image

FS Image (fsimage) là một thành phần trong Hệ thống Tệp phân tán Hadoop (HDFS). Nó đại diện cho một bản sao tĩnh của đồ thị tệp HDFS và chứa thông tin về cấu trúc của hệ thống tệp.

Trong fsimage, mỗi đối tượng như thư mục, tệp và phân vùng dữ liệu được mã hóa và lưu trữ dưới dạng một cấu trúc hỗn hợp. Các thông tin bao gồm tên, ID, quyền truy cập, kích thước, thời gian tạo và sửa đổi của các đối tượng. Cấu trúc fsimage giữ cho HDFS khả năng phục hồi dữ liệu trong trường hợp xảy ra sự cố và giúp tái tạo lại trạng thái của hệ thống tệp.

Fsimage được tạo và cập nhật bởi Namenode (hoặc Standby Namenode trong trường hợp sao lưu). Khi có bất kỳ thay đổi nào trong tệp HDFS, fsimage sẽ được cập nhật để phản ánh trạng thái mới nhất của dữ liệu. Nếu fsimage quá lớn, nó có thể được cắt nhỏ và tạo ra phiên bản mới để tiếp tục ghi thêm cập nhật vào đó.

Fsimage chủ yếu được sử dụng để khôi phục lại trạng thái của HDFS trong trường hợp xảy ra sự cố, ví dụ như khi mất điểm truy cập vào dữ liệu hoặc khi có lỗi xảy ra. Pạng kết hợp với nhật ký chỉnh sửa (editlog), fsimage giúp đảm bảo tính nhất quán và khả năng phục hồi cao của dữ liệu trong HDFS.

## 3.6 Kiểu dữ liệu trong HDFS

### 3.6.1 Tệp văn bản

• Bao gồm các bản ghi CSV, TSV, Json

• Định dạng thuận tiện để trao đổi giữa các ứng dụng hoặc kịch bản

• Dễ đọc và dễ phân tích cú pháp

• Không hỗ trợ nén khối

• Không hiệu quả khi truy vấn

• Tốt cho mục đích sơ khai, nhưng không đủ tốt cho thực tế.

### 3.6.2 Tệp chuỗi (Sequence file)

• Cung cấp một cấu trúc dữ liệu liên tục cho các cặp khóa-giá trị nhị phân

• Thông thường được sử dụng để chuyển dữ liệu giữa các công việc Map Reduce

• Có thể được sử dụng như một lưu trữ để đóng gói các tệp nhỏ trong Hadoop

• Dựa trên dòng (row-based)

• Nén

• Có thể chia thành các phần nhỏ (splittable)

• Hỗ trợ chia nhỏ ngay cả khi dữ liệu đã được nén

### 3.6.3 Avro

• Dựa trên dòng (row based)

• Hỗ trợ nén và chia nhỏ (splitting) dữ liệu (object)

• Lược đồ dữ liệu linh hoạt

• Bao gồm lược đồ (JSON) trong tệp

• Các loại dữ liệu

• Nguyên thủy: null, boolean, int, long, ...

• Phức tạp: bản ghi, mảng, bản đồ, ...

• Trình trạng dữ liệu dạng nhị phân và JSON

• Phát hiện lỗi dữ liệu bị hỏng

### 3.6.4 Parquet

• Định dạng tệp nhị phân hướng cột

• Hiệu quả trong việc đọc/ghi dữ liệu trên đĩa khi chỉ cần truy vấn các cột cụ thể

• Hỗ trợ nén và chia nhỏ (splitting) dữ liệu (theo trang - page)

• Hỗ trợ các cột lồng nhau (sử dụng mã hóa Dremel)

### 3.6.5 Optimized Row Columnar (ORC)

• RCFile

• Mỗi cột được nén riêng lẻ trong nhóm hàng (row group)

• Tệp ORC

• Nén theo chế độ khối (block-mode compression)

• Hỗ trợ các loại dữ liệu

• Dữ liệu được sắp xếp (trong một stripe)

• Lưu trữ các bộ sưu tập hàng và trong bộ sưu tập, dữ liệu hàng được lưu trữ theo định dạng cột

• Giới thiệu một chỉ mục nhẹ cho phép bỏ qua các khối hàng không liên quan

• Có thể chia thành các phần nhỏ (splittable) để cho phép xử lý song song các bộ sưu tập hàng

• Chỉ mục với các giá trị được tổng hợp cấp cột (như tối thiểu, tối đa, tổng và số lượng)

# 4. MapReduce

## 4.1 Khái niệm

MapReduce là khung xử lý dữ liệu mặc định của Hadoop.

MapReduce là một mô hình lập trình cho việc xử lý dữ liệu.

Đó không phải là một ngôn ngữ, mà là một kiểu xử lý dữ liệu được tạo ra bởi Google.

## 4.2 Đặc điểm

Ưu điểm của MapReduce:

• Đơn giản

• Linh hoạt

• Khả năng mở rộng.

MapReduce chia công việc thành nhiều tác vụ độc lập và lên lịch thực hiện chúng trên các nút của cụm.

• Mỗi tác vụ thực hiện công việc độc lập với nhau vì lý do mở rộng quy mô.

• Áp lực giao tiếp để luôn đồng bộ dữ liệu trên các nút sẽ ngăn mô hình hoạt động đáng tin cậy và hiệu quả ở quy mô lớn.

## 4.3 Hoạt động

Trong một cụm MapReduce, dữ liệu thường được quản lý bằng các hệ thống tập tin phân tán.

Người lập trình trong MapReduce phải chỉ định hai hàm, hàm "map" và hàm "reduce", để thực hiện vai trò của Mapper và Reducer trong chương trình MapReduce.

• Trong MapReduce, các phần tử dữ liệu luôn được cấu trúc dưới dạng cặp key-value (ví dụ: (K, V)).

• Các hàm map và reduce nhận và xuất ra các cặp (K, V).

Một tập con khác nhau của không gian khóa trung gian được gán cho mỗi Reducer.

Những tập con này được gọi là các phân vùng (partitions).

MapReduce gồm 2 pha

### 4.3.1 Pha Map

Hadoop chia công việc thành nhiều tác vụ "map" riêng lẻ.

• Số lượng tác vụ "map" được xác định bởi lượng dữ liệu đầu vào.

• Mỗi tác vụ "map" nhận một phần của dữ liệu đầu vào tổng thể để xử lý.

• Các tác vụ "map" xử lý từng bản ghi đầu vào một lúc.

• Đối với mỗi bản ghi đầu vào, chúng xuất ra không hoặc nhiều bản ghi đầu ra.

Trong trường hợp này, tác vụ "map" đơn giản chỉ phân tích bản ghi đầu vào,

• Sau đó, xuất ra các trường tên và giá cho mỗi bản ghi đó làm đầu ra.

Hadoop tự động sắp xếp và hợp nhất đầu ra từ tất cả các tác vụ "map".

• Quá trình trung gian này được gọi là shuffle và sort.

• Kết quả được cung cấp cho các tác vụ "reduce".

### 4.3.2 Pha Reduce

Đầu vào của tác vụ "reduce" đến từ quá trình shuffle và sort.

• Tương tự như với tác vụ "map", hàm "reduce" nhận một bản ghi mỗi lần.

• Một tác vụ "reduce" cụ thể nhận tất cả các bản ghi cho một khóa cụ thể.

• Đối với mỗi bản ghi đầu vào, tác vụ "reduce" có thể xuất ra không hoặc nhiều bản ghi đầu ra.

Hàm "reduce" của chúng ta đơn giản là tính tổng tổng số tiền của mỗi người,

• Và xuất ra tên nhân viên (là khóa) và tổng số tiền (là giá trị) làm đầu ra.

## 4.4 Thuật toán trong MapReduce

### 4.4.1 Thuật toán sắp xếp

• Được sử dụng để kiểm tra tốc độ xử lý của Hadoop

• Thực chất là "cuộc đua kéo IO"

• Đầu vào

• Một tập hợp các tệp, mỗi giá trị trên một dòng

• Khóa của Mapper là tên tệp và số dòng

• Giá trị của Mapper là nội dung của dòng đó

Ý tưởng

• Tận dụng các tính chất của reducer: cặp (key, value) được xử lý theo thứ tự theo key; các reducer cũng được sắp xếp

• Mapper: Hàm định danh cho giá trị (k, v) à (v, \_)

• Reducer: Hàm định danh (k’, \_) -> (k’, “”)

Cặp (key, value) từ mapper được chuyển tới reducer cụ thể dựa trên giá trị hash(key)

Hàm băm phải được chọn sao cho với k1 < k2 thì hash(k1) < hash(k2)

### 4.4.2 Thuật toán tìm kiếm

• Đầu vào

• Tập hợp các tệp chứa các dòng văn bản

• Một mẫu tìm kiếm để tìm

• Khóa mapper là tên tệp, số dòng

• Giá trị mapper là nội dung của dòng

• Mẫu tìm kiếm được gửi dưới dạng tham số đặc biệt

+ Mapper

• Đưa ra (tên tệp, một số văn bản) và “pattern”, nếu “text” khớp "pattern" đầu ra, (tên tệp, \_)

+ Reducer

• Hàm định danh

+ Tối ưu hóa

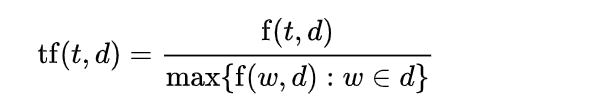
• Một khi một tệp tin được xác định là đáng chú ý, chúng ta chỉ cần đánh dấu nó một lần

• Sử dụng chức năng Combiner để gộp các cặp (tên tệp, \_) trùng lặp thành một cặp duy nhất

• Giảm thiểu I/O mạng

### 4.4.3 Thuật toán TF-IDF

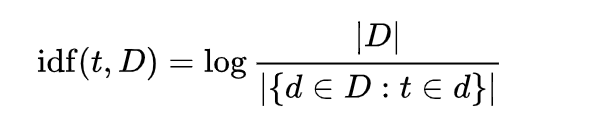
**TF**: Term Frequency(Tần suất xuất hiện của từ) là số lần từ xuất hiện trong văn bản. Vì các văn bản có thể có độ dài ngắn khác nhau nên một số từ có thể xuất hiện nhiều lần trong một văn bản dài hơn là một văn bản ngắn. Như vậy, term frequency thường được chia cho độ dài văn bản( tổng số từ trong một văn bản).

[](https://blog.luyencode.net/wp-content/uploads/2019/01/tf.png)

Trong đó:

* tf(t, d): tần suất xuất hiện của từ t trong văn bản d
* f(t, d): Số lần xuất hiện của từ t trong văn bản d
* max({f(w, d) : w ∈ d}): Số lần xuất hiện của từ có số lần xuất hiện nhiều nhất trong văn bản d

**IDF**: Inverse Document Frequency (Nghịch đảo tần suất của văn bản), giúp đánh giá tầm quan trọng của một từ. Khi tính toán TF, tất cả các từ được coi như có độ quan trọng bằng nhau. Nhưng một số từ như “is”, “of” và “that” thường xuất hiện rất nhiều lần nhưng độ quan trọng là không cao. Như thế chúng ta cần giảm độ quan trọng của những từ này xuống.

[](https://blog.luyencode.net/wp-content/uploads/2019/01/idf.png)

Trong đó:

* idf(t, D): giá trị idf của từ t trong tập văn bản
* |D|: Tổng số văn bản trong tập D
* |{d ∈ D : t ∈ d}|: thể hiện số văn bản trong tập D có chứa từ t.

Cơ số logarit trong công thức này không thay đổi giá trị idf của từ mà chỉ thu hẹp khoảng giá trị của từ đó. Vì thay đổi cơ số sẽ dẫn đến việc giá trị của các từ thay đổi bởi một số nhất định và tỷ lệ giữa các trọng lượng với nhau sẽ không thay đổi. (nói cách khác, thay đổi cơ số sẽ không ảnh hưởng đến tỷ lệ giữa các giá trị IDF). Việc sử dụng logarit nhằm giúp giá trị tf-idf của một từ nhỏ hơn, do chúng ta có công thức tính tf-idf của một từ trong 1 văn bản là tích của tf và idf của từ đó.

Cụ thể, chúng ta có **công thức tính tf-idf** hoàn chỉnh như sau: **tfidf(t, d, D) = tf(t, d) x idf(t, D)**

Khi đó:

Những từ có giá trị TF-IDF cao là những từ xuất hiện nhiều trong văn bản này, và xuất hiện ít trong các văn bản khác. Việc này giúp lọc ra những từ phổ biến và giữ lại những từ có giá trị cao (từ khoá của văn bản đó).

### 4.4.4 Thuật toán BFS và MapReduce

• Vấn đề: Thuật toán này không "phù hợp" với MapReduce

• Giải pháp: Thực hiện nhiều vòng lặp qua MapReduce - ánh xạ một số nút, kết quả bao gồm các nút bổ sung được chuyển vào các vòng lặp MapReduce tiếp theo

• Vấn đề: Gửi toàn bộ đồ thị cho một tác vụ ánh xạ (hoặc hàng trăm/ngàn tác vụ ánh xạ) đòi hỏi lượng bộ nhớ khổng lồ

• Giải pháp: Cân nhắc kỹ càng cách chúng ta biểu diễn đồ thị

1. Thảo luận

• Nhiệm vụ MapReduce này có thể tiến xa đến một bước tiếp theo của biên biết

• Để thực hiện toàn bộ BFS, một thành phần không phải MapReduce sau đó đưa đầu ra của bước này trở lại nhiệm vụ MapReduce để lặp lại

• Vấn đề: Danh sách chỉ đến điểm đã đi đâu?

• Giải pháp: Mapper ánh xạ (n, chỉ đến điểm) cũng như vậy

2. Mở rộng và kết thúc

• Thuật toán này bắt đầu từ một nút

• Các vòng lặp tiếp theo bao gồm nhiều nút hơn của đồ thị khi biên tiến

• Liệu điều này có bao giờ kết thúc không?

• Có! Cuối cùng, các đường đi giữa các nút sẽ không còn được khám phá và không tìm thấy khoảng cách tốt hơn nữa. Khi khoảng cách bằng nhau, chúng ta dừng lại

• Mapper nên ánh xạ (n, D) để đảm bảo "khoảng cách hiện tại" được chuyển vào reducer

3. Thêm trọng số

• Đường đi ngắn nhất với cạnh có trọng số hữu ích hơn so với phương pháp chi phí == 1

• Thay đổi đơn giản: danh sách chỉ đến điểm trong tác vụ ánh xạ bao gồm trọng số 'w' cho mỗi nút đã chỉ đến

• Phát (p, D+wp) thay vì (p, D+1) cho mỗi nút p

• Hoạt động cho đồ thị có trọng số dương

4. So sánh với thuật toán Dijkstra

• Thuật toán Dijkstra hiệu quả hơn vì ở bất kỳ bước nào nó chỉ theo đuổi các cạnh từ đường đi có chi phí nhỏ nhất trong biên giới

• Phiên bản MapReduce khám phá tất cả các đường đi song song; không hiệu quả như Dijkstra nhưng kiến trúc có tính mở rộng hơn

• Tương đương với Dijkstra với trọng số 1

### 4.4.5 PageRank

#### 4.4.5.1 Sơ lược

PageRank: Các bước đi ngẫu nhiên trên Web

• Nếu một người dùng bắt đầu từ một trang web ngẫu nhiên và duyệt bằng cách nhấp vào liên kết và ngẫu nhiên nhập địa chỉ URL mới, xác suất rằng anh ta sẽ đến được một trang cụ thể là bao nhiêu?

• PageRank của một trang web ghi nhận khái niệm này

• Các trang web "phổ biến" hoặc "đáng giá" hơn sẽ có một thứ hạng cao hơn

#### 4.4.5.2 Công thức

• Với trang A, và các trang T1 đến Tn liên kết với A, PageRank được định nghĩa như sau:

• PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))

• C(P) là số phần tử (độ ra) của trang P

• d là yếu tố lập dẫn ("URL ngẫu nhiên")

#### 4.4.5.3 Trực quan

• Tính toán là lặp đi lặp lại: PRi+1 được dựa trên PRi

• Mỗi trang phân phối PRi cho tất cả các trang mà nó liên kết tới. Những trang được liên kết thêm vào phần chúc thăm của họ để tìm PRi+1

• d là mộtam số điều chỉnh (thường = 0,85) bao gồm yếu tố "nhảy ngẫu nhiên"

PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))

PageRank: Triển khai đầu tiên

• Tạo hai bảng 'current' và 'next' chứa PageRank cho mỗi trang. Gieo giống 'current' với giá trị PR ban đầu

• Lặp qua tất cả các trang trong đồ thị, phân phối PR từ 'current' vào 'next' của trang được liên kết

• current := next; next := fresh\_table();

• Quay lại bước lặp hoặc kết thúc nếu hội tụ

#### 4.4.5.4 Phân phối của thuật toán

• Những hiểu biết quan trọng cho phân mảnh:

• Bảng 'next' phụ thuộc vào 'current', nhưng không phụ thuộc vào bất kỳ hàng khác của 'next' nào

• Các hàng riêng lẻ của ma trận kề có thể được xử lý song song

• Các hàng ma trận thưa là tương đối nhỏ

• Hậu quả của những hiểu biết:

• Chúng ta có thể ánh xạ từng hàng của 'current' sang một danh sách các "phân đoạn" PageRank để gán cho các trang được liên kết

• Những phân đoạn này có thể được thu gọn thành một giá trị PageRank duy nhất cho một trang bằng cách tổng hợp

• Biểu diễn đồ thị có thể thậm chí được ghi nhỏ hơn; vì mỗi phần tử chỉ đơn giản là 0 hoặc 1, chỉ cần truyền số cột nơi nó là 1

Giai đoạn 1: Phân tích HTML

• Tác vụ ánh xạ nhận các cặp (URL, nội dung trang) và ánh xạ chúng thành (URL, (PRkhởi tạo, danh sách-url))

• PRkhởi tạo là PageRank "hạt giống" cho URL

• danh sách-url chứa tất cả các trang mà URL chỉ đến

• Tác vụ giảm chỉ là chức năng đồng nhất

Giai đoạn 2: Phân phối PageRank

• Tác vụ ánh xạ nhận (URL, (thứ hạng-hiện-tại, danh sách-url))

• Đối với mỗi u trong danh sách-url, phát ra (u, thứ hạng-hiện-tại/|danh sách-url|)

• Phát ra (URL, danh sách-url) để đưa danh sách liên kết qua các vòng lặp

• Tác vụ giảm nhận (URL, danh sách-url) và nhiều giá trị (URL, val)

• Tổng các giá trị và sửa chữa bằng d

• Phát ra (URL, (thứ hạng-mới, danh sách-url))

PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))

Hoàn thiện...

• Một thành phần sau đó xác định xem đã đạt được sự hội tụ chưa (Số lần lặp cố định? So sánh các giá trị quan trọng?)

• Nếu có, ghi ra danh sách PageRank - đã hoàn thành!

• Nếu không, đưa đầu ra của Giai đoạn 2 vào một vòng lặp Giai đoạn 2 khác

#### 4.4.5.5 Ghi chú

• MapReduce thực hiện "công việc nặng" trong tính toán lặp lại

• Yếu tố chính trong việc song song hóa là tính toán PageRank độc lập trong mỗi bước

• Việc song song hóa đòi hỏi suy nghĩ về các phân vùng dữ liệu tối thiểu để truyền (ví dụ: biểu diễn gọn của các hàng trong đồ thị)

• Thậm chí việc triển khai được trình bày hôm nay thực tế không mở rộng được cho toàn bộ Internet; nhưng nó hoạt động với đồ thị có kích thước trung bình.

# 5. Yarn

## 5.1 Khái niệm

YARN (Yet Another Resource Negotiator) là một thành phần quan trọng trong hệ sinh thái Hadoop, quản lý và phân phối tài nguyên hệ thống như bộ nhớ, CPU và lưu lượng mạng.

## 5.2 Thành phần

YARN gồm ba thành phần chính:

1. ResourceManager (RM): Đây là thành phần trung tâm của YARN, quản lý tài nguyên của toàn bộ cụm và lập kế hoạch cho quá trình phân phối công việc, căn cứ vào yêu cầu từ các ứng dụng.

2. NodeManager (NM): Là thành phần chạy trên mỗi nút trong cụm, NodeManager quản lý tài nguyên cục bộ trên nút đó. Nó liên tục giao tiếp với ResourceManager để báo cáo trạng thái và yêu cầu tài nguyên cần thiết để thực hiện ứng dụng. ResourceManger có hai thành phần quan trọng đó là Scheduler và ApplicationManager

* Scheduler có trách nhiệm phân bổ tài nguyên cho các ứng dụng khác nhau. Đây là Scheduler thuần túy vì nó không thực hiện theo dõi trạng thái cho ứng dụng. Nó cũng không sắp xếp lại các tác vụ bị lỗi do lỗi phần cứng hoặc phần mềm. Bộ lập lịch phân bổ các tài nguyên dựa trên các yêu cầu của ứng dụng
* ApplicationMaster (AM): Đây là một thành phần ứng dụng đặc biệt, nằm trong quyền điều hành của ứng dụng, AM quyết định việc yêu cầu và quản lý các tài nguyên cụm cho ứng dụng. Mỗi ứng dụng chạy trên YARN có một AM riêng.

ApplicationManager có chức năng sau:

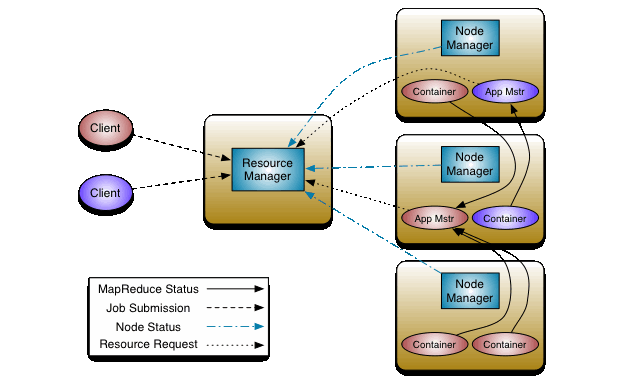
* Chấp nhận nộp công việc.
* Đàm phán container đầu tiên để thực thi ApplicationMaster. Một nơi chứa kết hợp các yếu tố như CPU, bộ nhớ, đĩa và mạng.
* Khởi động lại container ApplicationMaster khi không thành công.

## 5.3 Chức năng

YARN mở rộng khả năng của Hadoop bằng cách cho phép chạy nhiều ứng dụng khác nhau trên cùng một hệ thống, không chỉ là MapReduce. Nó cũng chịu trách nhiệm lập lịch và quản lý công việc cho mỗi ứng dụng trong cụm.

Nói một cách khác, YARN chính là nền tảng cho phép Hadoop chuyển từ một hệ thống xử lý batch không gian đơn lẻ dựa trên MapReduce tới một hệ thống xử lý dữ liệu đa mô hình hoàn chỉnh có khả năng thực hiện nhiều loại xử lý dữ liệu cùng một lúc.

## 5.4 Sơ đồ hoạt động của một ứng dụng trên Yarn



Quá trình 1 ứng dụng chạy trên YARN được mô tả bằng sơ đồ trên qua các bước sau:

* Client giao 1 task cho Resource Manager
* Resource Manager tính toán tài nguyên cần thiết theo yêu cầu của ứng dụng và tạo 1 App Master (App Mstr). Application Master được chuyển đến chạy 1 một node tính toán. Application Master sẽ liên lạc với các NodeManager ở các node khác để ra yêu cầu công việc cho node này.
* Node Manager nhận yêu cầu và chạy các task trên container
* Các thông tin trạng thái thay vì được gửi đến JobTracker sẽ được gửi đến App Master.

## 5.5 Spark và Yarn

Spark có thể chạy trên YARN. Khi chạy trên YARN, Spark sử dụng ResourceManager để yêu cầu tài nguyên và lập kế hoạch công việc trong cụm. Spark sẽ tạo ra một ứng dụng Spark, điều hành bởi một ApplicationMaster, và ApplicationMaster sẽ giao tiếp với ResourceManager để yêu cầu tài nguyên và báo cáo trạng thái công việc. NodeManagers sẽ quản lý và giám sát tài nguyên cục bộ trên mỗi nút trong cụm và thực hiện các công việc Spark trên các tài nguyên đó. YARN cho phép Spark chạy song song trên cụm Hadoop phân tán, tận dụng hiệu quả nguồn tài nguyên có sẵn và cung cấp khả năng xử lý dữ liệu lớn.

# 6. Spark

## 6.1 Hạn chế của Map Reduce

Map Reduce: Công việc lặp

• Công việc lặp bao gồm nhiều hoạt động đọc/ghi từ đĩa cho mỗi vòng lặp

• Đọc/ghi từ đĩa rất chậm!

Yêu cầu một hệ thống phân tích dữ liệu thống nhất cho xử lý dữ liệu quy mô lớn

hỗ trợ tốt hơn cho:

• Thuật toán lặp

• Khai thác dữ liệu tương tác

• Khả năng chịu lỗi, tối ưu dữ liệu cục bộ, khả năng mở rộng

• Ẩn đi sự phức tạp: giúp người dùng tránh việc viết mã cho cơ chế phân tán.

## 6.2 Khái niệm

Spark là một động cơ phân tích thống nhất cho việc xử lý dữ liệu quy mô lớn

• Tốc độ: thực hiện công việc nhanh gấp 100 lần

• Hiệu suất cao cho cả dữ liệu batch và streaming

• Tính toán chạy trong bộ nhớ

## 6.3 Đặc điểm

Dễ sử dụng: viết ứng dụng nhanh chóng bằng Java, Scala, Python, R, SQL

• Cung cấp hơn 80 toán tử cấp cao

• Sử dụng chúng theo cách tương tác từ Scala, Python, R và SQL

Tính tổng quát: kết hợp SQL, Streaming và phân tích phức tạp

• Cung cấp các thư viện bao gồm SQL và DataFrames, Spark Streaming, MLib, GraphX,

• Hỗ trợ một loạt các công việc, ví dụ: ứng dụng batch, thuật toán tương tác, truy vấn tương tác, streaming.

Dễ dàng chạy trên mọi nền tảng:

• Chạy trên Hadoop, Apache Mesos, Kubernetes, độc lập hoặc trên đám mây.

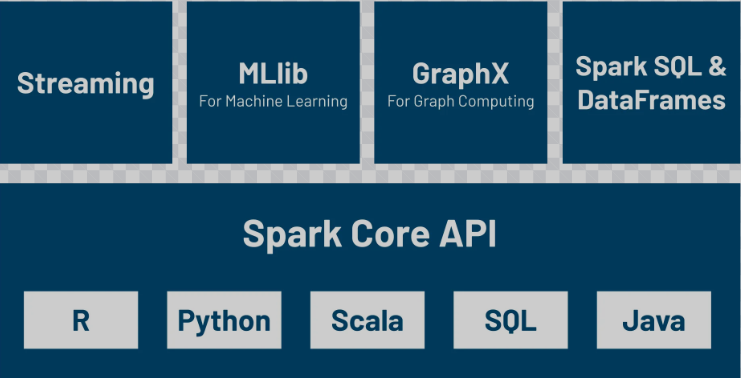
• Truy cập dữ liệu trên HDFS, Aluxio, Apache Cassandra, Apache Hbase, Apache Hive, v.v.

## 6.4 So sánh Spark và Map Reduce

Sự khác nhau giữa Spark and Map Reduce

|  |  |  |
| --- | --- | --- |
|  | Apache Hadoop MR | Apache Spark |
| Lưu trữ | Chỉ trên đĩa | Trên bộ nhớ hoặc đĩa |
| Hoạt động | Map và reduce | Các transformations và actions, bao gồm Map và Reduce |
| Mô hình thực thi | Batch (theo lô) | Batch (theo lô), Iterative (lặp lại) và streaming (trực tuyến) |
| Ngôn ngữ | Java | Scala, Java, Python và R |

## 6.5 Thành phần của Spark stack



+Spark Core:

• Bao gồm các chức năng cơ bản của Spark bao gồm lập lịch công việc, quản lý bộ nhớ, phục hồi lỗi, v.v.

• Cung cấp API để xây dựng và điều chỉnh RDDs.

+SparkSQL

• Cho phép truy vấn dữ liệu có cấu trúc qua SQL, Hive Query Language.

• Cho phép kết hợp truy vấn SQL và thao tác dữ liệu bằng Python, Java, Scala.

+ Spark Streaming: Cho phép xử lý dữ liệu trực tiếp thông qua API.

+Spark Mlib:

• Chứa các chức năng thông thường của máy học.

• Cung cấp nhiều loại thuật toán: phân loại, hồi quy, gom cụm, v.v.

+ GraphX:

• Thư viện để điều chỉnh đồ thị và thực hiện tính toán song song trên đồ thị.

• Mở rộng Spark RDD API.

+ Cluster Managers

• Hadoop Yarn

• Apache Mesos, và

• Standalone Schedular (quản lý đơn giản trong Spark).

## 6.6 Resilient Distributed Dataset (RDD)

• RDDs là cấu trúc dữ liệu phân tán bền vững, chịu lỗi và song song cho phép người dùng lưu trữ rõ ràng các kết quả trung gian trong bộ nhớ, kiểm soát việc phân vùng để tối ưu vị trí dữ liệu và điều khiển chúng bằng cách sử dụng một tập hợp phong phú các toán tử.

• Sự chuyển đổi thô (coarse-grained) so với các cập nhật tinh vi (fine-grained)

• Ví dụ: map, filter và join) là những phép toán áp dụng cùng một thao tác cho nhiều mục dữ liệu cùng một lúc.

RDD có thể được tạo ra theo hai cách:

- Tạo song song một bộ sưu tập (Parallelize)

- Đọc dữ liệu từ nguồn bên ngoài (S3, C\*, HDFS, v.v.)

1. Tạo song song (Parallelize)

Lấy một bộ sưu tập tồn tại trong bộ nhớ và truyền nó vào phương thức parallelize của SparkContext

Thường không được sử dụng bên ngoài mục đích prototyping và testing vì nó đòi hỏi toàn bộ tập dữ liệu trong bộ nhớ trên một máy tính

2. Đọc từ Tệp Văn Bản (Read from Text File)

Có các phương pháp khác để đọc dữ liệu từ HDFS, C\*, S3, HBase, v.v.

Các hoạt động trên Dữ liệu Phân tán

• Hai loại hoạt động: biến đổi (transformations) và hành động (actions)

• Các biến đổi được thực hiện chậm (không tính toán ngay lập tức)

• Các biến đổi được thực hiện khi một hành động được chạy

• Lưu trữ (cache) dữ liệu phân tán trong bộ nhớ hoặc đĩa

### 6.6.1 Các loại RDD

+ logLinesRDD: Đây là một RDD (Resilient Distributed Dataset) chứa các dòng log. Được sử dụng để lưu trữ và thao tác với các tệp log hoặc dữ liệu log.

+ errorsRDD: Đây là một RDD chứa các bản ghi log được phát hiện là lỗi. Được tạo ra thông qua việc lọc (filter) các dòng log và chỉ giữ lại những dòng log có nội dung đánh dấu là lỗi.

+ cleanedRDD: Đây là một RDD chứa các bản ghi log đã được làm sạch, tức là loại bỏ các dòng log không cần thiết hoặc dữ liệu log không hợp lệ. Thông thường, quá trình làm sạch (cleaning) có thể bao gồm việc xóa các dòng trắng, loại bỏ ký tự đặc biệt không mong muốn hoặc xử lý các thành phần dữ liệu log để chuẩn hóa chúng.

Các RDD trên giúp thực hiện các thao tác và xử lý dữ liệu log trong môi trường phân tán sử dụng Spark.

### 6.6.2 DAG

DAG execution (Directed Acyclic Graph execution) là quá trình thực hiện các phép biến đổi và action trên Spark RDDs (Resilient Distributed Datasets) theo mô hình DAG.

Trong DAG execution, các RDDs và các phép biến đổi (transformations) được tổ chức thành một đồ thị không chu trình (DAG). Các phép biến đổi là các bước biến đổi dữ liệu từ RDD gốc thông qua các phép biến đổi toán học và logic được áp dụng. Trên đồ thị DAG, mỗi đỉnh đại diện cho một RDD và mỗi cạnh thể hiện quan hệ phụ thuộc giữa các RDD dựa trên các phép biến đổi.

Khi quá trình DAG execution bắt đầu, Spark sẽ tối ưu hóa thứ tự thực hiện các phép biến đổi dựa trên dependency trong đồ thị DAG, nhằm tối đa hiệu suất tính toán. Spark cũng có thể thực hiện các phép biến đổi song song (parallel) khi có thể để tận dụng sức mạnh tính toán của nhiều worker node trong hệ thống phân tán.

Quá trình DAG execution cho phép Spark tối ưu hóa tính toán và tận dụng khả năng song song, giúp thực hiện các phép biến đổi trên dữ liệu một cách hiệu quả và nhanh chóng.

### 6.6.3 Triển khai RDD

Resilient Distributed Dataset (RDD)

• RDD ban đầu trên đĩa (HDFS, v.v.)

• RDD trung gian trong bộ nhớ RAM

• Khôi phục lỗi dựa trên lineage

• Các hoạt động trên RDD được phân tán

## 6.7 DataFrame

• DataFrame là một trừu tượng chính trong Spark 2.0.

• Một khi được tạo, DataFrame là bất biến (immutable).

• Theo dõi thông tin lineage để tính lại dữ liệu bị mất một cách hiệu quả.

• Cho phép thực hiện các hoạt động trên tập hợp các phần tử song song.

• Để tạo DataFrame:

- Bằng cách song song hóa các bộ sưu tập Python hiện có (các danh sách).

- Bằng cách biến đổi một DataFrame Spark hoặc pandas hiện có.

- Từ các tệp tin trong HDFS hoặc hệ thống lưu trữ khác.

### 6.7.1 Các biến đổi (Transformations)

• Tạo ra DataFrame mới từ một DataFrame hiện có.

• Sử dụng đánh giá lười biếng (lazy evaluation).

• Không có gì được thực thi ngay lập tức.

• Spark lưu trữ công thức cho nguồn biến đổi.

Các Transformatons phổ biến:

select(\*cols) - Chọn cột từ DataFrame hiện tại.

drop(col) - Trả về một DataFrame mới bỏ đi cột cụ thể.

filter(func) - Trả về một DataFrame mới được tạo bằng cách lựa chọn các dòng của nguồn mà func trả về true.

where(func) - Where là bí danh của filter.

distinct() - Trả về một DataFrame mới chứa các dòng duy nhất của nguồn DataFrame.

sort(\*cols, \*\*kw) - Trả về một DataFrame mới được sắp xếp theo các cột đã chỉ định và theo thứ tự sắp xếp được chỉ định bởi kw.

### 6.7.2 Các hành động (Actions)

• Chạy công thức biến đổi nguồn để Spark thực thi.

• Cung cấp các cơ chế để lấy kết quả ra khỏi Spark.

Các Actions phổ biến

show(n, truncate) - In ra n dòng đầu tiên của DataFrame này.

take(n) - Trả về n dòng đầu tiên dưới dạng danh sách các Row.

collect() - Trả về tất cả các bản ghi dưới dạng danh sách các Row (\*).

count() - Trả về số lượng dòng trong DataFrame này.

describe(\*cols) - Hàm phân tích dữ liệu thống kê tính toán các thống kê (số lượng, trung bình, độ lệch chuẩn, giá trị nhỏ nhất, giá trị lớn nhất) cho các cột số học.

(\*) Lưu ý: collect() có thể không nên sử dụng trên các DataFrame lớn vì nó có thể gây áp lực lên bộ nhớ và gây gián đoạn trong việc thực thi.

### 6.7.3 Quy trình lập trình Spark

• Tạo DataFrames từ dữ liệu bên ngoài hoặc tạo DataFrame từ một bộ sưu tập trong chương trình trình điều khiển (driver program).

• Chuyển đổi chúng một cách lười biếng (Lazily transform) thành các DataFrame mới.

• Lưu trữ bộ nhớ đệm (cache()) một số DataFrames để sử dụng lại.

• Thực hiện các hành động để thực thi tính toán song song và tạo ra kết quả.

### 6.7.4 DataFrames so với RDDs

• Đối với người dùng mới quen với khái niệm các bảng dữ liệu (data frames) trong các ngôn ngữ lập trình khác, API (Application Programming Interface - Giao diện lập trình ứng dụng) này sẽ giúp họ dễ dàng tiếp cận và sử dụng.

• Đối với người dùng Spark hiện có, API này sẽ giúp lập trình Spark dễ dàng hơn so với việc sử dụng RDDs.

• Đối với cả hai nhóm người dùng, DataFrames sẽ cải thiện hiệu suất thông qua tối ưu hóa thông minh và tạo mã tự động.

### 6.7.5 Tiện ích của DataFrames

Ghi ít mã hơn: Input và Output

Giao diện thống nhất để đọc/ghi dữ liệu trong nhiều định dạng khác nhau

Ghi ít mã hơn: Các hoạt động cấp cao

• Giải quyết các vấn đề thông thường một cách ngắn gọn với các hàm DataFrame:

• chọn các cột và lọc dữ liệu

• kết hợp các nguồn dữ liệu khác nhau

• tóm tắt (đếm, tổng, trung bình, v.v.)

• biểu đồ kết quả (ví dụ: với Pandas)

### 6.7.6 Dataset

1. Định nghĩa

+ Dataset là một cấu trúc dữ liệu trong SparkSQL được xác định rõ kiểu dữ liệu và ánh xạ tới một schema quan hệ. Nó biểu diễn các truy vấn có cấu trúc với các bộ mã hóa. Nó là một phần mở rộng của API data frame. Spark Dataset cung cấp cả tính an toàn kiểu và giao diện lập trình hướng đối tượng. Chúng ta biết đến việc phát hành dataset trong Spark 1.6.

+ Bộ mã hóa là khái niệm chính trong framework (SerDes) của việc tuần tự hóa và giải tuần tự hóa trong Spark SQL. Bộ mã hóa dịch giữa các đối tượng JVM và định dạng nhị phân nội bộ của Spark. Spark tích hợp sẵn bộ mã hóa rất tiên tiến. Chúng tạo bytecode để tương tác với dữ liệu ngoài heap.

+ Bộ mã hóa cung cấp truy cập cần thiết vào các thuộc tính riêng lẻ mà không cần giải tuần tự hóa toàn bộ đối tượng. Để tối ưu thời gian và không gian nhập-xuất, Spark SQL sử dụng framework SerDe. Vì bộ mã hóa biết về schema của bản ghi, nó có thể thực hiện tuần tự hóa và giải ứng.

+ Spark Dataset là biểu diễn truy vấn có cấu trúc và trì hoãn (lazy) gây ra hành động. Bên trong, dataset biểu diễn một logical plan. Logical plan cho biết truy vấn tính toán mà chúng ta cần tạo ra dữ liệu. Logical plan là một logical query plan của toán tử logic để tạo thành kế hoạch truy vấn logic. Khi chúng ta phân tích và giải quyết điều này, chúng ta có thể tạo ra một physical query plan.

Dataset kết hợp các tính năng của RDD và DataFrame. Nó cung cấp:

* Sự tiện lợi của RDD.
* Tối ưu hóa hiệu suất của DataFrame.
* An toàn kiểu tĩnh của Scala.

Do đó, Dataset cung cấp một giao diện lập trình hướng chức năng hơn để làm việc với dữ liệu có cấu trúc.

2. Tầm quan trọng của Dataset trong Spark

Để vượt qua nhược điểm của RDD và Dataframe, Dataset đã ra đời. Trong Dataframe, không có sẵn tính an toàn kiểu dữ liệu vào thời điểm biên dịch. Dữ liệu không thể thay đổi mà không biết cấu trúc của nó. Trong RDD, không có sự tối ưu hóa tự động. Do đó, chúng ta phải tối ưu nó thủ công khi cần thiết.

3. Các tính năng của Dataset trong Spark

Sau khi đã được giới thiệu về Dataset, bây giờ hãy thảo luận về các tính năng khác nhau của Dataset trong Spark-

a. Truy vấn tối ưu hóa

Dataset trong Spark cung cấp truy vấn tối ưu hóa bằng cách sử dụng Catalyst Query Optimizer và Tungsten. Catalyst Query Optimizer là một framework không phụ thuộc vào việc thực thi. Nó biểu diễn và chỉnh sửa một đồ luồng dữ liệu. Bằng cách tối ưu hóa Spark job, Tungsten cải thiện quá trình như thực thi. Tungsten tập trung vào kiến trúc phần cứng của nền tảng mà Apache Spark chạy trên đó.

b. Phân tích tại thời điểm biên dịch

Sử dụng Dataset, chúng ta có thể kiểm tra cú pháp và phân tích tại thời điểm biên dịch. Điều này không thể thực hiện bằng Dataframe, RDD hoặc các truy vấn SQL thông thường.

c. Lưu trữ bền vững

Dataset trong Spark có thể được lưu trữ cả ở dạng serializable và có thể truy vấn được. Do đó, chúng ta có thể lưu trữ nó vào bộ nhớ lưu trữ

d. Chuyển đổi qua lại

Chúng ta có thể chuyển đổi Dataset với kiểu dữ liệu an toàn sang DataFrame "không có kiểu". Để thực hiện tác vụ này, Dataset cung cấp ba phương thức để chuyển đổi từ các loại Seq[T] hoặc RDD[T] sang Dataset[T]:

toDS(): Dataset[T]

toDF(): DataFrame

toDF(colNames: String\*): DataFrame

e. Tính toán nhanh hơn

Thực hiện Dataset nhanh hơn cách thực hiện RDD. Điều này gia tăng hiệu xuất của hệ thống. Để đạt được hiệu xuất tương tự sử dụng RDD, người dùng phải xem xét thủ công cách biểu diễn tính toán để tối ưu hóa phân tán.

f. Tiêu thụ ít bộ nhớ hơn

Khi lưu cache, nó tạo ra bố cục tối ưu hơn. Spark biết cấu trúc dữ liệu trong Dataset.

g. API đơn cho Java và Scala

Nó cung cấp một giao diện đơn cho Java và Scala. Sự đồng nhất này đảm bảo chúng ta có thể sử dụng giao diện Scala, ví dụ mã từ cả hai ngôn ngữ. Nó cũng giảm bớt gánh nặng của các thư viện. Vì bây giờ các thư viện không cần xử lý hai loại đầu vào khác nhau nữa.

4. Để tạo một Dataset

a. SparkSession

SparkSession là điểm vào cho SparkSQL. Đây là đối tượng đầu tiên chúng ta tạo khi phát triển ứng dụng Spark SQL sử dụng trừu tượng dữ liệu Dataset có kiểu đầy đủ. Chúng ta có thể tạo một phiên bản SparkSession bằng cách sử dụng SparkSession.Builder và có thể dừng SparkSession bằng cách sử dụng phương thức stop (spark.stop).

b. QueryExecution

Chúng ta biểu diễn quá trình thực thi truy vấn có cấu trúc của dataset bằng cách sử dụng QueryExecution. Để truy cập QueryExecution của một Dataset, chúng ta sử dụng thuộc tính QueryExecution. Bằng cách thực thi logical plan trong Spark Session, chúng ta nhận được QueryExecution.

executePlan(plan: LogicalPlan): QueryExecution

Hàm executePlan thực thi logical plan đầu vào để tạo ra một QueryExecution trong SparkSession hiện tại.

c. Encoder

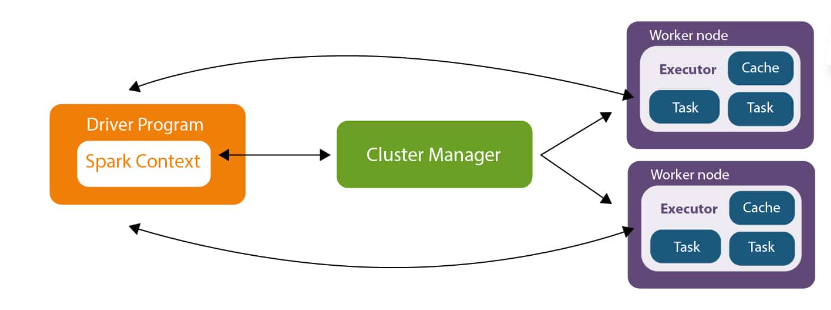
Một encoder cung cấp chức năng chuyển đổi giữa biểu diễn dạng bảng và các đối tượng JVM. Sử dụng encoder, chúng ta tuần tự hóa đối tượng. Encoder tuần tự hóa đối tượng để xử lý hoặc truyền qua mạng.

## 6.8 Kiến trúc Spark

• Một kiến trúc loại master-worker

• Một nút trình điều khiển (driver) hoặc master node

• Các worker node



• master nodes gửi công việc cho các worker node và chỉ định chúng để lấy dữ liệu từ bộ nhớ hoặc từ đĩa cứng (hoặc từ nguồn khác như S3 hoặc HDFS)

• Chương trình Spark đầu tiên tạo một đối tượng SparkContext

• SparkContext cho biết Spark sẽ truy cập và dùng một cụm xử lý nào và ở đâu

• Tham số master cho một SparkContext xác định loại và kích thước của cụm xử lý sẽ được sử dụng

### 6.8.1 Các tham số master

local - Chạy Spark một cách cục bộ với một luồng làm việc (không có song song hóa)

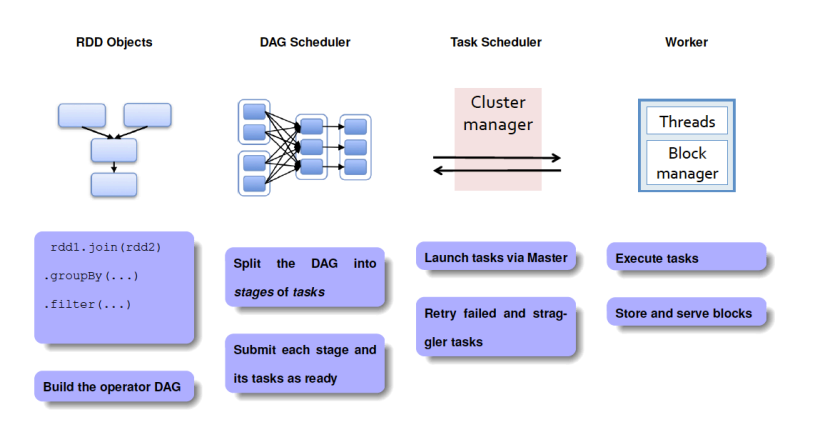
local[K] - Chạy Spark một cách cục bộ với K luồng làm việc (lý tưởng là số lõi CPU)

spark://HOST:PORT - Kết nối với cụm xử lý Spark độc lập

mesos://HOST:PORT - Kết nối với cụm xử lý Mesos

yarn - Kết nối với cụm xử lý YARN

### 6.8.2 Vòng đời một Job trong Spark



RDD Objects (Resilient Distributed Datasets - Tập dữ liệu phân tán chịu lỗi)

+ Xây dựng DAG của các toán tử

DAG Scheduler (Lập lịch DAG)

+Chia DAG thành các giai đoạn của các tác vụ

+Gửi từng giai đoạn và tác vụ tương ứng khi sẵn sang

Task Scheduler

+Khởi chạy tác vụ thông qua Master

+Thử lại các tác vụ thất bại và trì trệ

Worker

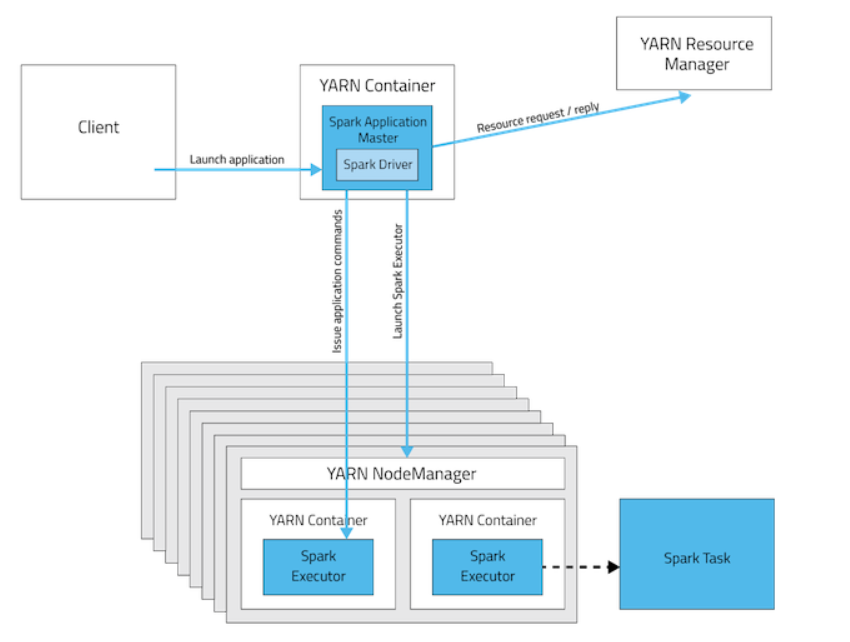
+Thực hiện các tác vụ

+Lưu trữ và phục vụ các khối dữ liệu

6.8.3 Chạy một công việc spark trên Yarn

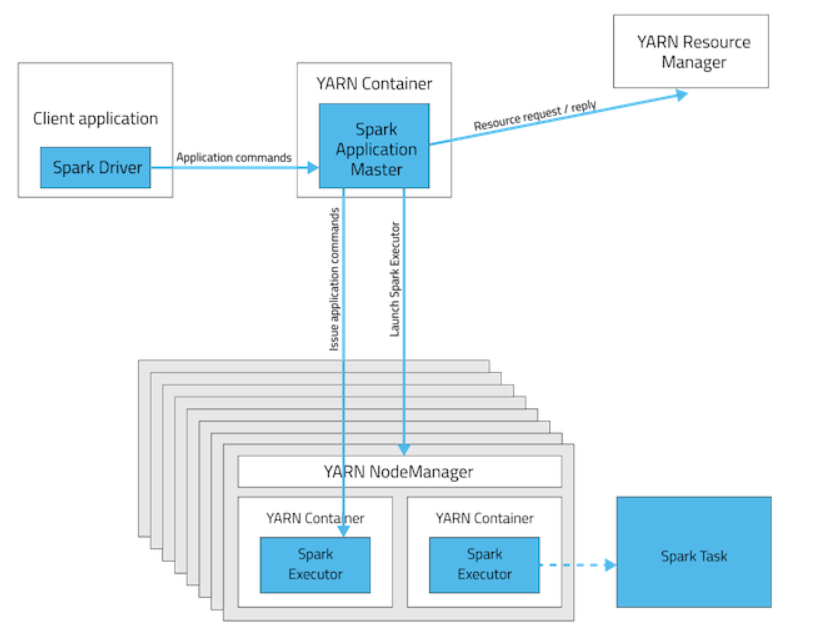
+ Triển khai trên cụm cluster

* Trong chế độ Cluster, trình điều khiển Spark chạy trong ApplicationMaster trên một máy chủ trong cụm. Một quá trình đơn trong một container YARN chịu trách nhiệm cho cả việc điều khiển ứng dụng và yêu cầu tài nguyên từ YARN. Khách hàng khởi chạy ứng dụng không cần chạy trong suốt thời gian tồn tại của ứng dụng.
* Chế độ Cluster không phù hợp cho việc sử dụng Spark tương tác. Các ứng dụng Spark yêu cầu đầu vào từ người dùng, như spark-shell và pyspark, yêu cầu trình điều khiển Spark chạy bên trong quá trình khách hàng khởi tạo ứng dụng Spark.



+ Triển khai chế độ Client

* Trong chế độ khách hàng, trình điều khiển Spark chạy trên máy chủ nơi công việc được gửi. ApplicationMaster chỉ chịu trách nhiệm yêu cầu container thực thi từ YARN. Sau khi các container khởi động, khách hàng truyền thông với các container để lập lịch công việc.



6.9 Hạn chế của Spark

a. Không hỗ trợ xử lý thời gian thực

Đối với Spark, xử lý gần thời gian thực chỉ tạm thời diễn ra trong Spark Streaming. Điều này có nghĩa là Spark không được coi là một hệ thống xử lý thời gian thực hoàn toàn.

b. Vấn đề với các tệp nhỏ

Trong RDD, mỗi tệp được chia thành các phân vùng nhỏ. Điều này có nghĩa là có một lượng lớn các phân vùng nhỏ trong RDD. Do đó, nếu chúng ta muốn hiệu suất trong quá trình xử lý, RDDs cần được phân vùng lại thành một định dạng có thể quản lý được. Điều này đòi hỏi sử dụng phương thức tráo đổi thông qua mạng rất phức tạp.

c. Thiếu hệ thống quản lý tệp

Một vấn đề lớn là Spark không có hệ thống quản lý tệp riêng. Thực tế, Spark phụ thuộc vào một nền tảng khác như Hadoop hoặc một nền tảng trên đám mây khác.

d. Đắt đỏ

Trong khi chúng ta mong muốn xử lý dữ liệu lớn với chi phí hiệu quả, Spark lại rất đắt đỏ. Do việc giữ dữ liệu trong bộ nhớ mất rất nhiều chi phí. Bên cạnh đó, tiêu thụ bộ nhớ rất cao và không được quản lý một cách thân thiện với người dùng. Hơn nữa, chúng ta cần nhiều RAM để chạy trong bộ nhớ, do đó chi phí của Spark cao hơn nhiều.

e. Số lượng thuật toán được hỗ trợ ít

Spark MLlib chỉ có số lượng thuật toán hỗ trợ ít. Ví dụ, khoảng cách Tanimoto.

f. Tối ưu hóa thủ công

Công việc Spark phải được tối ưu hóa thủ công và phải phù hợp với các bộ dữ liệu cụ thể. Hơn nữa, để phân vùng và lưu cache trong Spark đúng đắn, chúng ta phải điều khiển nó thủ công.

g. Xử lý lặp lại

Cơ bản, dữ liệu lặp đi lặp lại theo từng lô. Mỗi lần lặp lại được lên lịch và thực thi riêng biệt.

h. Độ trễ

So sánh với Flink, Apache Spark có độ trễ cao hơn.

i. Tiêu chí cửa sổ

Spark chỉ hỗ trợ tiêu chí cửa sổ dựa trên thời gian, không hỗ trợ tiêu chí cửa sổ dựa trên bản ghi.

# 7 Apache Kafka

## 7.1 Khái niệm Kafka

Apache Kafka là hệ thống gửi nhận tin nhắn theo hình thức publish-subscribe nhanh, linh hoạt, bền vững và chịu được lỗi

• Gửi và nhận tin nhắn theo dạng các luồng ghi chú (streams of records)

• Lưu trữ bền vững trong trường hợp có lỗi xảy ra

• Sao chép các phân vùng nhật ký chủ đề (Topic Log Partitions) đến nhiều máy chủ

• Xử lý các bản ghi khi chúng xảy ra

• Tốc độ IO nhanh, hiệu quả, đóng gói, nén dữ liệu và nhiều tính năng khác

• Sử dụng để tách rời các luồng dữ liệu

• Thường được sử dụng thay cho JMS, RabbitMQ và AMQP

• Tăng công suất, đáng tin cậy và sao chép dữ liệu nhiều hơn

## 7.2 Khả năng của Kafka

+ Xây dựng ứng dụng xử lý dữ liệu phản ứng thời gian thực

• Cung cấp dữ liệu cho hệ thống phân tích thời gian thực

• Chuyển đổi, phản ứng, tổng hợp, kết hợp các luồng dữ liệu thời gian thực (ví dụ: thu thập số liệu)

• Cung cấp sự kiện cho hệ thống xử lý sự kiện phức tạp (CEP)

• Cung cấp dữ liệu phân tích hàng ngày hoặc hàng giờ có độ trễ cao vào Spark, Hadoop, v.v.

• (ví dụ: Nhật ký cam kết bên ngoài cho hệ thống phân tán. Sao chép dữ liệu giữa các nút, đồng bộ lại để khôi phục trạng thái của nút)

• Bảng tổng hợp và bảng điều khiển cập nhật

+ Xây dựng đường ống dữ liệu streaming thời gian thực

• Hỗ trợ các microservices lưu trong bộ nhớ (actors, Akka, Vert.x, Qbit,RxJava)

## 7.3 Sự ứng dụng của Kafka

• 1/3 số công ty trong danh sách Fortune 500

• 7 trong số 10 công ty du lịch hàng đầu, 8 trong số 10 ngân hàng hàng đầu, 9 trong số 10 công ty bảo hiểm hàng đầu, 7 trong số 10 công ty viễn thông hàng đầu

• LinkedIn, Microsoft và Netflix xử lý 1 tỷ tin nhắn mỗi ngày bằng Kafka

• Luồng dữ liệu thời gian thực, được sử dụng để thu thập dữ liệu lớn hoặc để thực hiện phân tích thời gian thực (hoặc cả hai)

Tại sao Kafka phổ biến?

• Hiệu suất tuyệt vời

• Đơn giản trong vận hành, dễ thiết lập và sử dụng, dễ hiểu

• Ổn định, đáng tin cậy, bền vững

• Cung cấp tính năng đăng ký-đăng xuất/ xếp hàng (publish-subscribe/queue) linh hoạt (mở rộng theo số lượng nhóm tiêu thụ)

• Sao chép mạnh mẽ

• Đảm bảo tính nhất quán có thể điều chỉnh cho nhà sản xuất (producer)

• Thứ tự được duy trì ở cấp shard (partition topic)

• Hoạt động tốt với các hệ thống có luồng dữ liệu cần xử lý, tổng hợp, chuyển đổi và tải vào các kho lưu trữ khác.

## 7.4 Thuật ngữ quan trọng

• Kafka duy trì nguồn thông điệp trong các danh mục được gọi là topics.

• Luồng ghi chú (" /orders", " /user-signups"), tên feed

• Lưu trữ chủ đề nhật ký (Log topic) trên đĩa

• Partition / Segments (phần của Topic Log)

• Ghi chú có key (tùy chọn), giá trị và thời điểm, không thể thay đổi

• Các quy trình gửi tin nhắn vào chủ đề Kafka được gọi là (producers).

• Các quy trình đăng ký chủ đề và xử lý luồng dữ liệu các tin nhắn đã được gửi được gọi là người tiêu thụ (consumers).

• Kafka được chạy dưới dạng một cụm bao gồm một hoặc nhiều máy chủ, mỗi máy chủ được gọi là broker.

## 7.5 Kiến trúc Kafka

• Cụm Kafka bao gồm nhiều brokers và ZooKeeper.

• Giao tiếp giữa tất cả các thành phần được thực hiện thông qua một API nhị phân đơn giản và hiệu suất cao qua giao thức TCP.

• ZooKeeper cung cấp một cái nhìn đồng bộ về cấu hình cụm Kafka.

• Bầu cử lãnh đạo cho cặp Kafka Broker và Topic Partition,

• Quản lý khám phá dịch vụ cho Brokers Kafka tạo thành cụm

• ZooKeeper gửi các thay đổi cho Kafka.

• Thêm Broker mới, Broker bị ngừng hoạt động, v.v.

• Xóa Topic, Thêm Topic, v.v.

## 7.6 Cơ chế hoạt động của Kafka

Kafka hoạt động dựa trên mô hình tin nhắn gửi và nhận. Cơ chế hoạt động của Kafka được mô tả từ input đến output như sau:

7.6.1 Producer (Người sản xuất):

- Producer tạo ra các tin nhắn và gửi chúng đến Kafka.

- Tin nhắn được gom nhóm lại thành các khối (batch) để giảm chi phí mạng và tối ưu việc ghi vào Kafka.

- Producer gắn thẻ tin nhắn với một chủ đề (topic) cụ thể, cho biết nơi mà tin nhắn sẽ được lưu trữ.

7.6.2 Kafka cluster (Tụ hợp Kafka):

- Kafka cluster bao gồm nhiều broker (máy chủ), mỗi broker đảm nhận vai trò lưu trữ và xử lý tin nhắn.

- Tin nhắn được phân chia thành các phân vùng (partition), mỗi phân vùng được lưu trữ trên một broker.

- Kafka sử dụng cơ chế phân phối tin nhắn để đảm bảo tính cân bằng tải giữa các broker và các phân vùng.

7.6.3 Consumer (Người tiêu thụ):

- Consumer đăng ký để đọc tin nhắn từ một chủ đề cụ thể.

- Kafka giữ theo dõi vị trí đọc (offset) của từng consumer để đảm bảo không có tin nhắn nào bị bỏ qua hoặc đọc lại.

- Consumer có thể đọc tin nhắn từ Kafka theo các cơ chế tiêu thụ khác nhau, bao gồm hàng đợi đơn (single consumer) hoặc nhóm tiêu thụ (consumer group).

7.6.4 Output (Đầu ra):

- Tin nhắn được đọc từ Kafka bởi consumer và được xử lý theo logic đặc trưng của ứng dụng.

- Các kết quả xử lý có thể được gửi tới hệ thống khác hoặc lưu trữ lại.

Kafka đảm bảo tính bền vững và khả năng mở rộng trong việc xử lý hàng triệu tin nhắn mỗi giây qua các phân vùng và các consumer khác nhau. Cơ chế hoạt động này giúp Kafka trở thành một nền tảng quan trọng trong việc xử lý dữ liệu trực tuyến (real-time data processing) và xây dựng hệ thống phân tán lớn.

## 7.7 Kiến trúc topic Kafka

Kafka chủ đề (topic) nhật ký(log), và phân vùng(partition)

• Kafka topics là một luồng ghi chú (stream of records)

• Topic được lưu trữ trong nhật ký (log)

• Topic là một danh mục hoặc tên luồng hoặc feed

• Topics được sử dụng theo mô hình publish-subscribe

• Một topic có thể có không hoặc nhiều subscriber - consumer groups

### 7.7.1 Phân vùng của chủ đề (Topic partitions)

• Topic được chia thành các phân vùng, thường dựa trên khóa của bản ghi

• Các phân vùng được sử dụng để mở rộng Kafka ra nhiều máy chủ

• Bản ghi được gửi đến phân vùng đúng thông qua khóa

• Phân vùng có thể được sao chép đến nhiều máy chủ

### 7.7.2 Nhật ký phân vùng của chủ đề (Topic partition log)

• Thứ tự chỉ được duy trì trong một phân vùng duy nhất

• Phân vùng là một chuỗi ghi chú có thứ tự, bất biến và liên tục được thêm vào - một nhật ký cam kết có cấu trúc

• Các bản ghi trong phân vùng có một số id tuần tự gọi là offset

### 7.7.3 Sao chép phân vùng của Kafka

• Mỗi phân vùng có một máy chủ đóng vai trò leader và không hoặc nhiều máy chủ đóng vai trò follower

• Leader xử lý tất cả các yêu cầu đọc và ghi cho phân vùng

• Follower sao chép từ leader

• Một replica đang đồng bộ (in-sync replica - ISR) là replica đã sao chép đầy đủ dữ liệu từ leader

• Nếu leader phân vùng gặp lỗi, một ISR mới sẽ được chọn làm leader

• Các phân vùng của nhật ký được phân tán trên các máy chủ trong cụm Kafka, trong đó mỗi máy chủ xử lý dữ liệu và yêu cầu cho một phần phân vùng

• Mỗi phân vùng có thể được sao chép trên một số máy chủ Kafka có thể cấu hình

• Được sử dụng để chống lỗi

• Bản ghi được coi là "đã xác nhận" khi tất cả ISR của phân vùng đã ghi vào nhật ký của mình.

• Chỉ các bản ghi đã xác nhận mới có thể được đọc bởi consumer.

• Một phân vùng khác có thể thuộc sở hữu của một leader khác trên một broker Kafka khác.

### 7.7.4 Tính bảo đảm

• Các tin nhắn được gửi bởi producer vào một phân vùng chủ đề nhất định sẽ được ghi nối theo thứ tự chúng được gửi đi

• Có thể cấu hình tối thiểu ISR có sẵn để trả về lỗi nếu không có đủ replica có sẵn để sao chép dữ liệu

• Một instance consumer nhìn thấy các tin nhắn theo thứ tự chúng được lưu trữ trong nhật ký

• Đối với một chủ đề có yếu tố sao chép N, Kafka có thể chịu được lỗi của tối đa N-1 máy chủ mà không "mất" bất kỳ tin nhắn nào đã được cam kết trong nhật ký

### 7.7.5 Giữ lại bản ghi Kafka

• Cụm Kafka giữ lại tất cả các bản ghi đã được xuất bản

• Dựa trên thời gian - có thể cấu hình khoảng thời gian lưu giữ

• Dựa trên kích thước - có thể cấu hình dựa trên kích thước

• Tính chính xác - giữ lại bản ghi mới nhất

• Chính sách lưu giữ ba ngày, hai tuần hoặc một tháng

• Dữ liệu vẫn có sẵn để tiêu thụ cho đến khi được loại bỏ theo thời gian, kích thước hoặc tính chính xác

• Tốc độ tiêu thụ không bị ảnh hưởng bởi kích thước

### 7.7.6 Ghi bền vững

• Producer có thể chọn đánh đổi (trade-off) giữa hiệu suất và độ bền của việc ghi

• Lưu ý: hiệu suất cũng có thể được tăng lên với nhiều broker hơn...

### 7.7.7 Hoạt động của Producers

• Producers truy nhập vào một chủ đề mà họ chọn (push)

• Producer(s) thêm Bản ghi vào cuối Log topic

• Tải có thể được phân phối trong số phân vùng

• Thông thường theo phương pháp "round-robin"

• Cũng có thể thực hiện "phân vùng ngữ nghĩa" dựa trên một khóa trong tin nhắn

• Ví dụ: các sự kiện của một 'employeeId' cụ thể được đưa vào cùng một phân vùng

• Quan trọng: Producer chọn phân vùng

• Tất cả các node có thể trả lời các yêu cầu về dữ liệu siêu dữ liệu về:

• Các máy chủ còn sống

• Vị trí của leader cho các phân vùng của một chủ đề

### 7.7.8 Hoạt động của Consumer

• Nhiều Consumer có thể đọc từ cùng một chủ đề

• Mỗi Consumer chịu trách nhiệm quản lý offset của riêng mình

• Tin nhắn vẫn còn trên Kafka... chúng không bị xóa sau khi được tiêu thụ

• Consumer có thể biến mất

• Và sau đó quay lại

### 7.7.9 Nhóm Consumer

• Các Consumer được nhóm thành các Nhóm Consumer

• Mỗi nhóm consumer có một ID duy nhất

• Mỗi nhóm consumer là một subscriber

• Mỗi nhóm consumer duy trì offset riêng của mình

• Nhiều subscriber = nhiều nhóm consumer

• Mỗi nhóm có chức năng khác nhau: một nhóm có thể giao bản ghi cho các microservices trong khi một nhóm khác đang gửi dữ liệu đến Hadoop

• Một bản ghi được gửi đến một Consumer trong một Nhóm Consumer

• Mỗi Consumer trong nhóm consumer lấy các bản ghi và chỉ có một Consumer trong nhóm nhận được cùng một bản ghi

• Các Consumer trong Nhóm Consumer cân bằng việc tiêu thụ bản ghi

### 7.7.10 Mô hình phổ biến của nhóm consumer

• Tất cả các thể hiện consumer trong cùng một nhóm

• Hoạt động giống như một hàng đợi truyền thống với cân bằng tải

• Tất cả các tin nhắn được phát sóng đến tất cả các thể hiện consumer trong cùng một nhóm.

• Tất cả các thể hiện consumer trong các nhóm khác nhau

• Mỗi thể hiện consumer trong các nhóm khác nhau không nhận được các tin nhắn được phát sóng trong nhóm khác.

• "Người đăng ký Logic" - Nhiều thể hiện consumer trong một nhóm

• Các consumer được thêm vào để mở rộng và đảm bảo tính chịu lỗi

• Mỗi thể hiện consumer đọc từ một hoặc nhiều phân vùng cho một chủ đề

• Không thể có nhiều thể hiện consumer hơn số phân vùng

### 7.7.11 Chia sẻ tải của Kafka consumer

• Thành viên của nhóm Consumer được xử lý bởi giao thức Kafka một cách động

• Nếu những Consumer mới tham gia nhóm Consumer, nó sẽ nhận một phần phân vùng

• Nếu một Consumer bị lỗi, các phân vùng của nó được chia thành các Consumer hoạt động trong nhóm Consumer còn lại

### 7.7.12 Kafka consumer khôi phục hệ thống

• Các Consumer thông báo cho broker khi xử lý thành công một bản ghi

• Tiến bộ offset ("\_\_consumer\_offset")

• Nếu một Consumer gặp sự cố trước khi gửi offset đã xác nhận cho Kafka broker

• Consumer khác có thể tiếp tục từ offset đã xác nhận cuối cùng

• Một số bản ghi Kafka có thể được xử lý lại

• Hành vi ít nhất một lần

• Tin nhắn nên có tính chất idempotent, tức là việc xử lý lại các tin nhắn không làm thay đổi kết quả cuối cùng.

### 7.7.13 Có thể tiêu thụ những gì (What can be consumed)

• "Log end offset" là offset của bản ghi cuối cùng được ghi vào partition log và nơi mà Producer sẽ ghi tiếp

• "High watermark" là offset của bản ghi cuối cùng được sao chép thành công đến tất cả các partition follower

• Consumer chỉ đọc đến "high watermark". Consumer không thể đọc dữ liệu chưa được sao chép

### 7.7.14 Số lượng Consumer đối với số lượng phân vùng

• Chỉ một Consumer từ cùng một nhóm consumer có thể truy cập vào một Partition cụ thể

• Nếu số lượng nhóm consumer vượt quá số lượng Partition:

• Các Consumer dư thừa sẽ ở trạng thái không hoạt động; có thể sử dụng cho mục đích chuyển đổi lỗi

• Nếu có nhiều Partition hơn số lượng thể hiện nhóm consumer:

• Một số Consumer sẽ đọc từ nhiều phân vùng

### 7.7.15 Broker Kafka

• Cụm Kafka được tạo thành từ nhiều Broker Kafka

• Mỗi Broker có một ID (số)

• Broker chứa các partition log của chủ đề

• Kết nối với một broker sẽ khởi động client cho toàn bộ cụm

• Bắt đầu với ít nhất ba broker, cụm có thể có 10, 100, 1000 broker nếu cần thiết

## 7.8 Tăng cường và tốc độ của Kafka

• Làm thế nào Kafka có thể mở rộng nếu có nhiều Producer và Consumer đọc/ghi vào cùng một log chủ đề của Kafka?

• Ghi nhanh: Ghi tuần tự vào hệ thống tệp nhanh (700MB hoặc hơn mỗi giây)

• Tăng cường ghi và đọc bằng cách chia tách:

• Partition log của chủ đề thành các Partition (các phần của log chủ đề)

• Partition chủ đề có thể được chia thành nhiều Partition khác nhau trên các máy tính/khác nhau ổ cứng

• Nhiều Producer có thể ghi vào các Partition khác nhau của cùng một chủ đề

• Nhiều nhóm Consumer có thể đọc từ các partition khác nhau một cách hiệu quả

• Tốc độ xử lý cao và độ trễ thấp

• Gom nhóm các tin nhắn riêng lẻ để giảm khối lượng mạng và gom nhóm các phần tử để ghi nhật ký và tiêu thụ cùng nhau

• Đảm bảo hiệu quả từ đầu đến cuối từ Producer đến hệ thống tệp tin đến Consumer

• Cung cấp nén dữ liệu hiệu quả hơn. Giảm độ trễ I/O

• Sử dụng I/O không sao chép bằng cách sử dụng sendfile (phương thức transferTo của Java’s NIO FileChannel).

• Triển khai cuộc gọi hệ thống sendfile() của Linux giúp tránh sao chép không cần thiết

• Phụ thuộc nhiều vào Linux PageCache

• Bộ điều lập I/O sẽ gom nhóm các ghi nhỏ liên tiếp thành các ghi lớn hơn về mặt vật lý để cải thiện tốc độ xử lý.

• Bộ điều lập I/O sẽ cố gắng tái sắp xếp các ghi để giảm thiểu sự di chuyển của đầu đọc đĩa và cải thiện tốc độ xử lý.

• Tự động sử dụng toàn bộ bộ nhớ trống trên máy tính

## 7.9 Phân phối dữ liệu

• Ít nhất một lần

• Tin nhắn không bao giờ bị mất nhưng có thể được gửi lại

• Tối đa một lần

• Tin nhắn bị mất nhưng không bao giờ được gửi lại

• Đúng một lần

• Tin nhắn được gửi đúng một lần duy nhất

+ Cách đạt được tính chất đúng một lần

• Phải cân nhắc hai thành phần

• Đảm bảo tính bền vững khi xuất bản một tin nhắn

• Đảm bảo tính bền vững khi tiêu thụ một tin nhắn

• Producer

• Điều gì sẽ xảy ra khi một yêu cầu ghi được gửi nhưng gặp lỗi mạng trước khi nhận được phản hồi (ack)?

• Sử dụng một writer duy nhất cho mỗi phân vùng và kiểm tra giá trị đã ghi nhận mới nhất sau khi gặp lỗi mạng

• Consumer

• Bao gồm một ID duy nhất (ví dụ: UUID) để loại bỏ trùng lặp.

• Xem xét lưu trữ vị trí (offset) cùng với dữ liệu

## 7.10 Vị trí của Kafka

• Đối với việc chuyển tệp tin thực sự lớn

• Có lẽ không, nó được thiết kế cho "tin nhắn" chứ không phải cho việc chuyển tệp tin. Nếu bạn cần chuyển tệp tin lớn, hãy xem xét việc chuyển tệp tin thông qua phương pháp truyền truyền thống hoặc chia nhỏ tệp thành từng dòng để chuyển sang Kafka.

• Như một sự thay thế cho MQ/Rabbit/Tibco

• Có lẽ có. Các con số hiệu suất cải thiện đáng kể. Ngoài ra, đưa ra khả năng cho các Consumer tạm thời. Xử lý lỗi khá tốt.

• Nếu bảo mật trên broker và qua mạng là quan trọng?

• Hiện tại chưa. Chúng ta không thể áp dụng nhiều biện pháp bảo mật. (KAFKA-1682)

• Để thực hiện biến đổi dữ liệu

• Không đúng với bản thân nó

# 8. Hệ phân tán

## 8.1 Sơ lược

Hệ thống tập tin (File system)

• Trừu tượng hóa các thiết bị lưu trữ

Hệ thống tập tin phân phối (Distributed file system)

• Có sẵn cho các quy trình từ xa trong hệ thống phân phối

## 8.2 Lợi ích

• Chia sẻ tập tin

• Tầm nhìn thống nhất của hệ thống từ các khách hàng khác nhau

• Quản trị tập trung

## 8.3 Mục tiêu

Mục tiêu: Độ rõ mạng

* Độ rõ mạng (Truy cập)
* Người dùng nên có thể truy cập vào các tập tin trên mạng một cách dễ dàng như truy cập vào các tập tin được lưu trữ cục bộ.
* Người dùng không cần phải biết vị trí vật lý của một tập tin để truy cập vào nó.
* Độ rõ mạng có thể được giải quyết thông qua các cơ chế đặt tên và gắn kết tập tin.
* Độ rõ vị trí: tên tập tin không chỉ định vị trí vật lý
* Độc lập vị trí: các tập tin có thể được di chuyển đến vị trí vật lý mới, không cần phải thay đổi các tham chiếu tới chúng. (Một tên không phụ thuộc vào địa chỉ của nó)
* Độc lập vị trí → độ rõ vị trí, nhưng ngược lại không nhất thiết phải đúng.

Mục tiêu: Khả dụng (Availability)

• Khả dụng: các tập tin nên dễ dàng và nhanh chóng được truy cập.

• Số lượng người dùng, các lỗi hệ thống hoặc các hậu quả khác của việc phân phối không nên ảnh hưởng đến khả dụng.

• Được giải quyết chủ yếu thông qua sao chép (replication).

## 8.4 Kiến trúc (Architectures)

Kiểu máy khách-phục vụ (Client-Server)

• Hệ thống tệp mạng của Sun Microsystem (NFS), Hệ thống tệp Google (GFS)

• Kiến trúc

• Một hoặc nhiều máy (máy chủ tệp) quản lý hệ thống tệp.

• Tập tin được lưu trữ trên các ổ đĩa của các máy chủ.

• Các yêu cầu về các hoạt động tệp được thực hiện từ các máy khách tới các máy chủ.

• Hệ thống máy khách-phục vụ tập trung lưu trữ và quản lý; các hệ thống P2P phân tán nó.

Kiểu đối xứng (Symmetric)

• Hoàn toàn phân tán; dựa trên công nghệ ngang hàng-to-peer (peer-to-peer technology)

• Ví dụ, Ivy (sử dụng phương pháp Chord DHT).

## 8.5 Vấn đề trong thiết kế hệ phân tán

Đặt tên và giải quyết tên (Naming and name resolution)

• Một không gian tên (name space) - tập hợp các tên

• Giải quyết tên (name resolution) - ánh xạ một tên thành một đối tượng

• 3 cách truyền thống

• Kết hợp tên máy chủ với tên các tập tin được lưu trữ trên máy chủ đó

Gắn các thư mục từ xa vào các thư mục cục bộ

• Cung cấp một thư mục toàn cầu duy nhất

## 8.6 Ý nghĩa chia sẻ tập tin

• Vấn đề: Khi làm việc với hệ thống tệp phân tán, chúng ta cần xem xét việc sắp xếp các hoạt động đọc/ghi đồng thời và ý nghĩa mong đợi (= tính nhất quán) của chúng.

• Giả sử mở; đọc/ghi; đóng

• Ý nghĩa UNIX: giá trị đọc là giá trị được lưu bởi việc ghi cuối cùng

Các việc ghi vào một tệp mở sẽ ngay lập tức hiển thị cho các người dùng khác có tệp này đang mở cùng một lúc. Dễ triển khai nếu chỉ có một máy chủ và không có bộ nhớ cache.

• Ý nghĩa phiên:

Các việc ghi vào một tệp mở bởi một người dùng sẽ không được hiển thị ngay lập tức cho các người dùng khác đã mở tệp. Một khi tệp đã được đóng, các thay đổi được thực hiện bởi nó sẽ hiển thị cho các phiên bắt đầu sau đó.

• Ý nghĩa Tập tin chia sẻ không thay đổi:

Một tệp có thể chia sẻ không thể được sửa đổi.

Tên tệp không thể được tái sử dụng và nội dung của nó không thể bị thay đổi.

Dễ triển khai.

• Giao dịch: Tất cả các thay đổi có thuộc tính toàn bộ hoặc không gì cả.

W1, R1, R2, W2 không

## 8.7 Bộ nhớ đệm (Caching)

Bộ nhớ đệm trên máy chủ: trong bộ nhớ chính (main memory)

• Vấn đề quản lý bộ nhớ đệm, bao nhiêu nội dung nên được lưu trong bộ nhớ đệm, chiến lược thay thế

• Vẫn chậm do độ trễ mạng

• Được sử dụng trong các máy chủ công suất cao của các hệ thống tìm kiếm web

Bộ nhớ đệm trên máy khách trong bộ nhớ chính

• Có thể được sử dụng bởi các máy trạm không có đĩa

• Truy cập nhanh hơn từ bộ nhớ chính so với đĩa

• Cạnh tranh với hệ thống bộ nhớ ảo vì không gian bộ nhớ vật lý

Bộ nhớ đệm của máy khách trên đĩa cục bộ

• Có thể lưu trữ các tập tin lớn trong bộ nhớ đệm

• Quản lý bộ nhớ ảo đơn giản hơn

• Máy trạm có thể hoạt động ngay cả khi không kết nối với mạng

Sự đánh đổi về bộ nhớ đệm (Caching tradeoffs)

• Giảm số lượng truy cập từ xa => giảm lưu lượng mạng và tải máy chủ

• Tổng chi phí hoạt động mạng thấp hơn đối với các phần dữ liệu lớn (bộ nhớ đệm) hơn là một loạt các yêu cầu cụ thể.

• Truy cập đĩa có thể được tối ưu hơn cho các yêu cầu lớn hơn là các khối đĩa ngẫu nhiên

• Vấn đề nhất quán của bộ nhớ đệm là hạn chế chính. Nếu có nhiều việc ghi thường xuyên, chi phí do vấn đề nhất quán là đáng kể.

## 8.8 Nhân bản (Replication)

Dữ liệu tập tin được nhân bản đến nhiều máy chủ lưu trữ.

Mục tiêu

• Tăng độ tin cậy

• Nâng cao khả dụng

• Cân bằng công việc của máy chủ

Làm cho việc nhân bản trở nên rõ ràng như thế nào?

Làm thế nào để giữ cho các bản sao nhất quán?

• một bản sao không được cập nhật do lỗi máy chủ của nó

• mạng bị phân mảnh

# 9. NoSQL

## 9.1 Tại sao chọn NoSQL

Các ứng dụng web có nhu cầu khác nhau

• Khả năng mở rộng theo chiều ngang - giảm chi phí

• Phân tán địa lý

• Đàn hồi

• Không có cấu trúc, cấu trúc linh hoạt cho dữ liệu bán cấu trúc

• Dễ dàng cho các nhà phát triển

• Lưu trữ dữ liệu không đồng nhất

• Khả năng sẵn có/Phục hồi sau thảm họa cao

Ứng dụng web không luôn cần

• Giao dịch

• Tính nhất quán mạnh

• Truy vấn phức tạp

Kiểu dữ liệu Key/Values

+ giao diện đơn giản (Get, Put, Delete)

+ Dữ liệu có thể bao gồm nhiều kiểu khác nhau

+ Rất nhanh và dễ dàng mở rộng (không joins)

## 9.2 Lý thuyết CAP

### 9.2.1 Định lượng Cơ sở dữ liệu truyền thống

RDBMS truyền thống có thể được mở rộng theo hai cách:

Theo chiều dọc (hoặc Tăng)

• Có thể đạt được bằng cách nâng cấp phần cứng (ví dụ: CPU nhanh hơn, bộ nhớ nhiều hơn hoặc ổ đĩa lớn hơn)

• Giới hạn bởi lượng CPU, RAM và đĩa mà có thể cấu hình trên một máy duy nhất

Theo chiều ngang (hoặc Out)

• Có thể đạt được bằng cách thêm nhiều máy tính

• Yêu cầu phân vùng cơ sở dữ liệu và có thể sao chép

• Giới hạn bởi tỷ lệ Đọc-Ghi và gánh nặng giao tiếp

Phân vùng dữ liệu

• Dữ liệu thường được phân vùng (hoặc tạo vạt) để cho phép truy cập đồng thời / song song

• Liệu nó có phù hợp với xử lý truy vấn phức tạp không?

Sao chép dữ liệu

Sao chép dữ liệu qua các máy chủ giúp:

• Tránh tình trạng hạn chế hiệu suất

• Tránh điểm hỏng duy nhất

• Và do đó, nâng cao khả năng mở rộng và sẵn có.

Nhưng, tính nhất quán trở thành một thách thức

• Ví dụ:

• Trong một ứng dụng thương mại điện tử, cơ sở dữ liệu ngân hàng đã được sao chép trên hai máy chủ

• Việc duy trì sự nhất quán của dữ liệu sao chép là một thách thức.

### 9.2.2 Định lý CAP

Những hạn chế của cơ sở dữ liệu phân tán có thể được mô tả trong cái gọi là định lý CAP (Consistency, Availability, Partition Tolerance - Tính nhất quán, Sẵn có, Kháng chia mạng)

• Tính nhất quán: mọi nút luôn nhìn thấy cùng một dữ liệu trong bất kỳ trạng thái nào (tính nhất quán nghiêm ngặt)

+Mỗi lần đọc sẽ nhận được dữ liệu mới nhất hoặc lỗi (error).

• Tính sẵn có: hệ thống tiếp tục hoạt động, ngay cả khi các nút trong cụm gặp sự cố hoặc một số phần cứng hoặc phần mềm bị ngừng hoạt động do nâng cấp

+Mỗi truy vấn luôn nhận được một trả lời không lỗi (non-error response), nhưng không đảm bảo dữ liệu trả về chứa thông tin mới nhất.

• Tính chống chịu lỗi: hệ thống tiếp tục hoạt động trong trường hợp có lỗi

+Hệ thống tiếp tục hoạt động bình thường mặc dù một số lượng bất kỳ các thông điệp (message) bị mất (hoặc trễ) vì lỗi mạng giữa các node.

Định lý CAP: bất kỳ cơ sở dữ liệu phân tán nào với dữ liệu chia sẻ, tối đa có hai trong ba thuộc tính mong muốn, C, A hoặc P. Đây là sự đánh đổi tồn tại trong hệ thống phân tán theo Eric Brewer trong PODC 2000.

### 9.2.3 Thuộc tính BASE

Các thuộc tính BASE (Basically Available, Soft-State, Eventual Consistency) là một phần hợp lý tiếp theo của lý thuyết CAP (Consistency, Availability, Partition tolerance). Đây là một cơ sở lý thuyết cho các cơ sở dữ liệu được thiết kế để hoạt động trong môi trường mạng phân tán. Dưới đây là phân tích của các thuộc tính BASE:

1. Basically Available: Hệ thống đảm bảo tính sẵn có (Availability). Điều này có nghĩa là hệ thống luôn sẵn sàng phục vụ truy vấn và yêu cầu từ người dùng mà không cần cấu hình hoàn hảo hay hoạt động ổn định hết thời gian. Thay vì đảm bảo tính nhất quán tuyệt đối (Consistency) vào mọi thời điểm, hệ thống chấp nhận một số hiệu ứng từ việc không đồng nhất trong dữ liệu.

2. Soft-State: Trạng thái của hệ thống có thể thay đổi theo thời gian. Điều này cho phép hệ thống tồn tại những trạng thái tạm thời và thích ứng với các biến đổi trong mạng phân tán hoặc tải công việc.

3. Eventual Consistency: Hệ thống sẽ cuối cùng đạt được sự nhất quán (Consistency). Theo đó, nếu không có truy vấn mới được thêm vào và không có thay đổi nào trong hệ thống trong một khoảng thời gian, dữ liệu sẽ dần đạt được sự nhất quán trên toàn bộ hệ thống.

Với các thuộc tính BASE, các cơ sở dữ liệu cho phép linh hoạt hơn trong việc đảm bảo tính nhất quán và sẵn có trong môi trường mạng phân tán. Chúng chấp nhận một số mức đồng nhất tạm thời và đảm bảo cuối cùng hệ thống sẽ đạt được sự nhất quán.