Kiến trúc lưu trữ và xử lý dữ liệu phân tán (Hadoop, Spark)

Nguyễn Mạnh Linh – Nguyễn Đức Thịnh

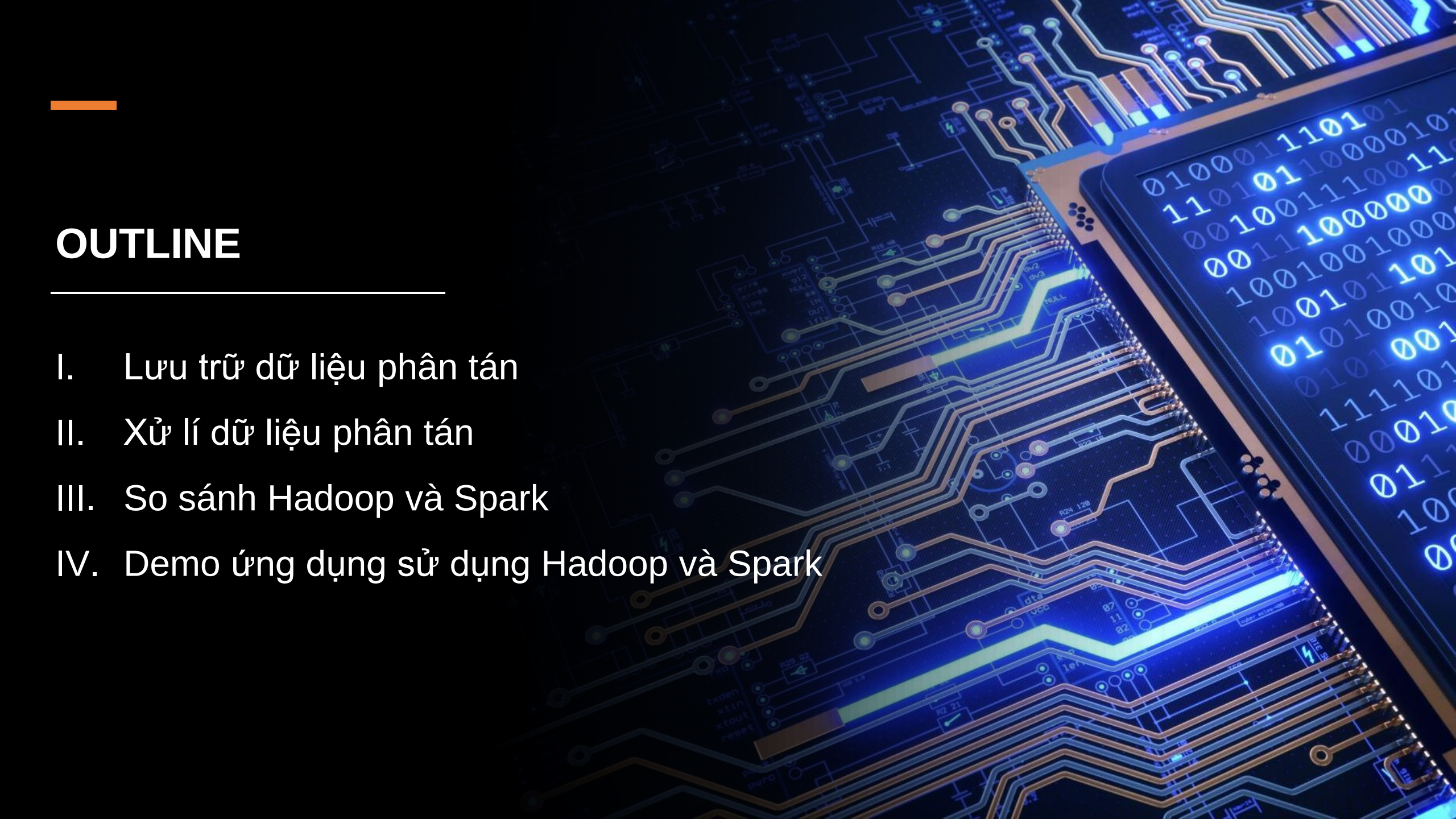
OUTLINE

I. Lưu trữ dữ liệu phân tán

II. Xử lí dữ liệu phân tán

III. So sánh Hadoop và Spark

IV. Demo ứng dụng sử dụng Hadoop và Spark



I. LƯU TRỮ DỮ LIỆU PHÂN TÁN

1. Hadoop Distributed File System (HDFS)

1.1. Kiến trúc và các thành phần chính

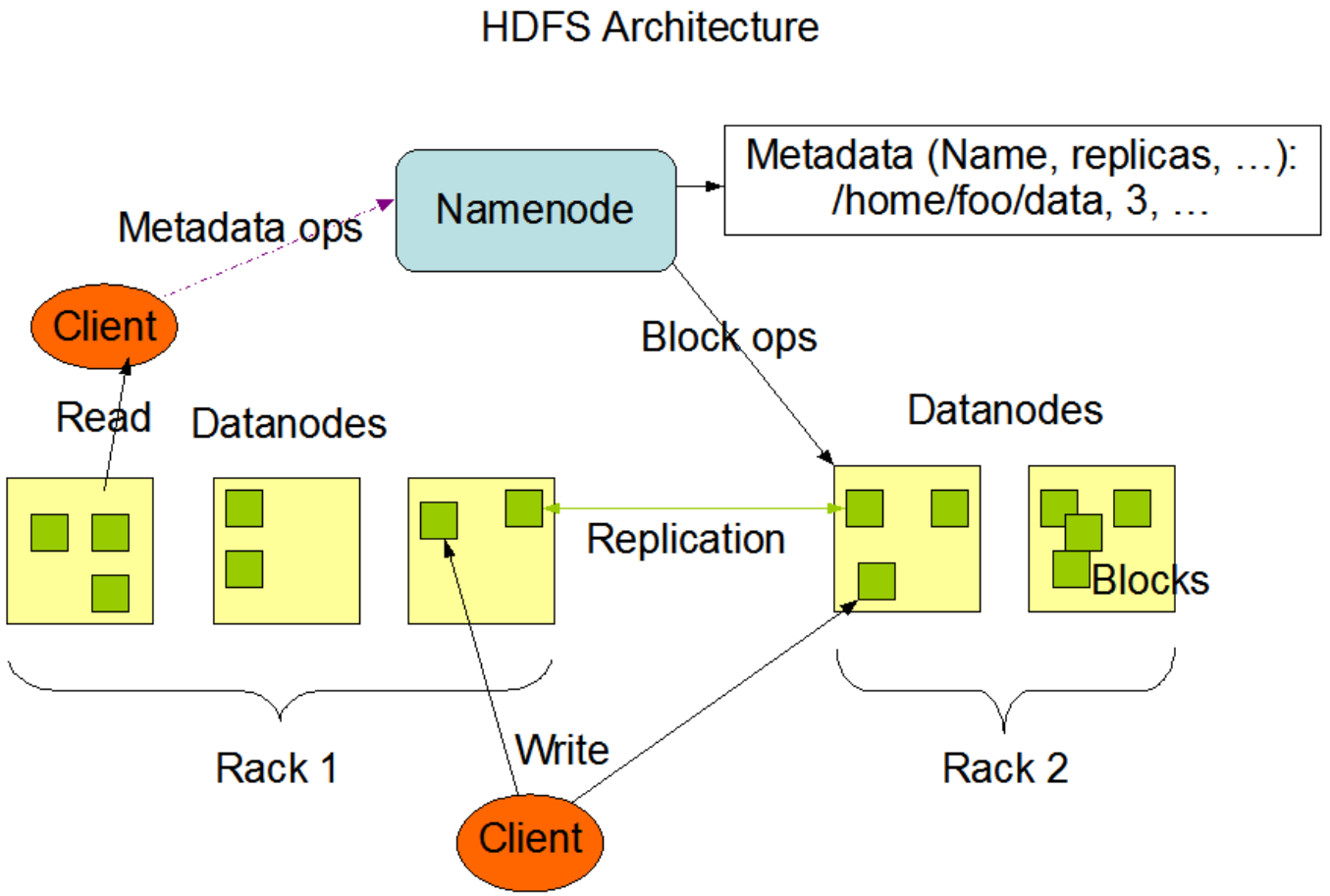
1.2. Lưu trữ và phân phối các block dữ liệu

1.3. Data replication và khả năng chịu lỗi

1.4. Những ưu điểm của HDFS

1.1. Kiến trúc và các thành phần chính của HDFS

- NameNode
- DataNode
- Khối dữ liệu (Blocks)
- Rack
- NameNode phụ
- HDFS Federation và High Availability
- HDFS Command-Line Interface (CLI) và APIs



1.1. Kiến trúc và các thành phần chính của HDFS

a. NameNode:

- Là thành phần trung tâm của HDFS và hoạt động như nút chủ (**master node**).
- Lưu trữ **metadata** về các tệp và thư mục, bao gồm vị trí và các block tương ứng của chúng.
- Quản lý các hoạt động không gian tên hệ thống tệp như tạo, xóa và sửa đổi tệp.

b. DataNode:

- Phân tán trên cụm và hoạt động như các node con (**slave nodes**).
- Chịu trách nhiệm lưu trữ và quản lý các khối dữ liệu thực tế.
- Mỗi DataNode định kỳ báo cáo trạng thái của mình và không gian lưu trữ khả dụng cho NameNode.

1.1. Kiến trúc và các thành phần chính của HDFS

c. Khối dữ liệu (Blocks): trong HDFS, các tệp dữ liệu được chia thành các khối (block) có kích thước cố định (thường mặc định là 64 hoặc 128 MB).

d. Rack: là một bộ chuyển đổi mạng vật lý hoặc một nhóm các máy được kết nối chặt chẽ nằm trong cùng một trung tâm dữ liệu. Các DataNode trên cùng một rack giao tiếp với nhau bằng các kết nối có băng thông cao, độ trễ thấp.

e. NameNode phụ:

- Hỗ trợ NameNode chính trong việc quản lý không gian tên hệ thống tệp.
- Thực hiện kiểm tra metadata định kỳ để sao lưu.
- Giúp giảm thiểu thời gian khởi động lại NameNode trong trường hường hợp xảy ra sự cố.

1.1. Kiến trúc và các thành phần chính của HDFS

f. HDFS Federation và High Availability:

- HDFS Federation cho phép quản lý nhiều cụm độc lập (Namespaces) bằng một cụm Hadoop duy nhất.
- Khả năng sẵn có cao (HA) đảm bảo tính liên tục của NameNode bằng cách sử dụng một NameNode dự phòng.

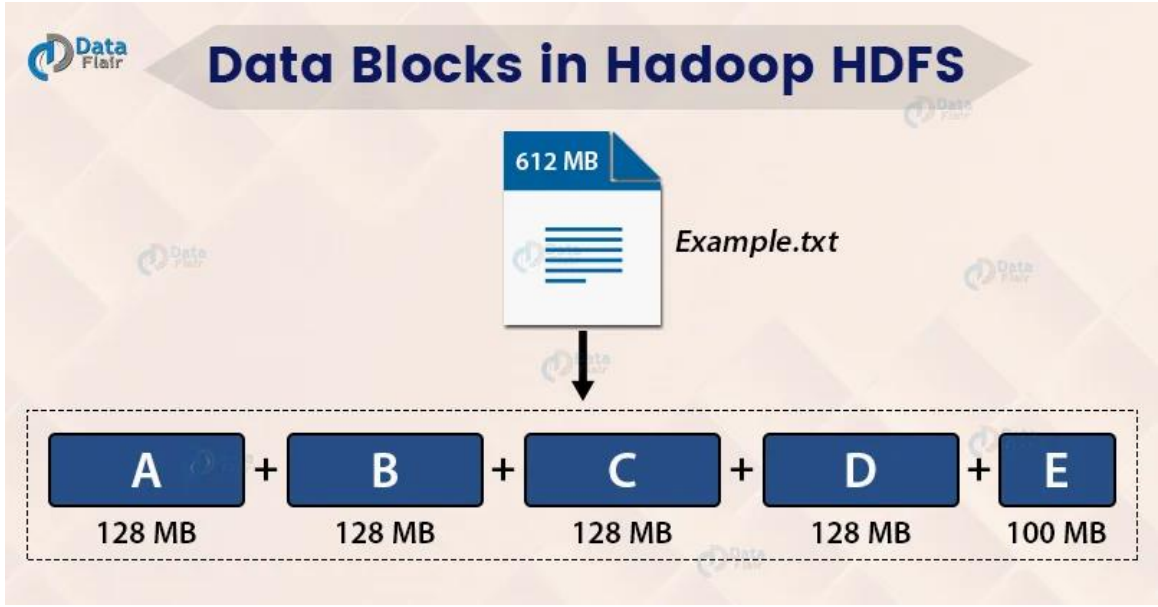
g. Giao diện dòng lệnh (CLI) và API của HDFS:

- HDFS cung cấp các công cụ dòng lệnh (ví dụ: hdfs dfs) và API (Java, Hadoop FileSystem API) để tương tác với hệ thống tệp.

1.2. Lưu trữ và phân phối các Block dữ liệu

a. Lưu trữ khối (blocks):

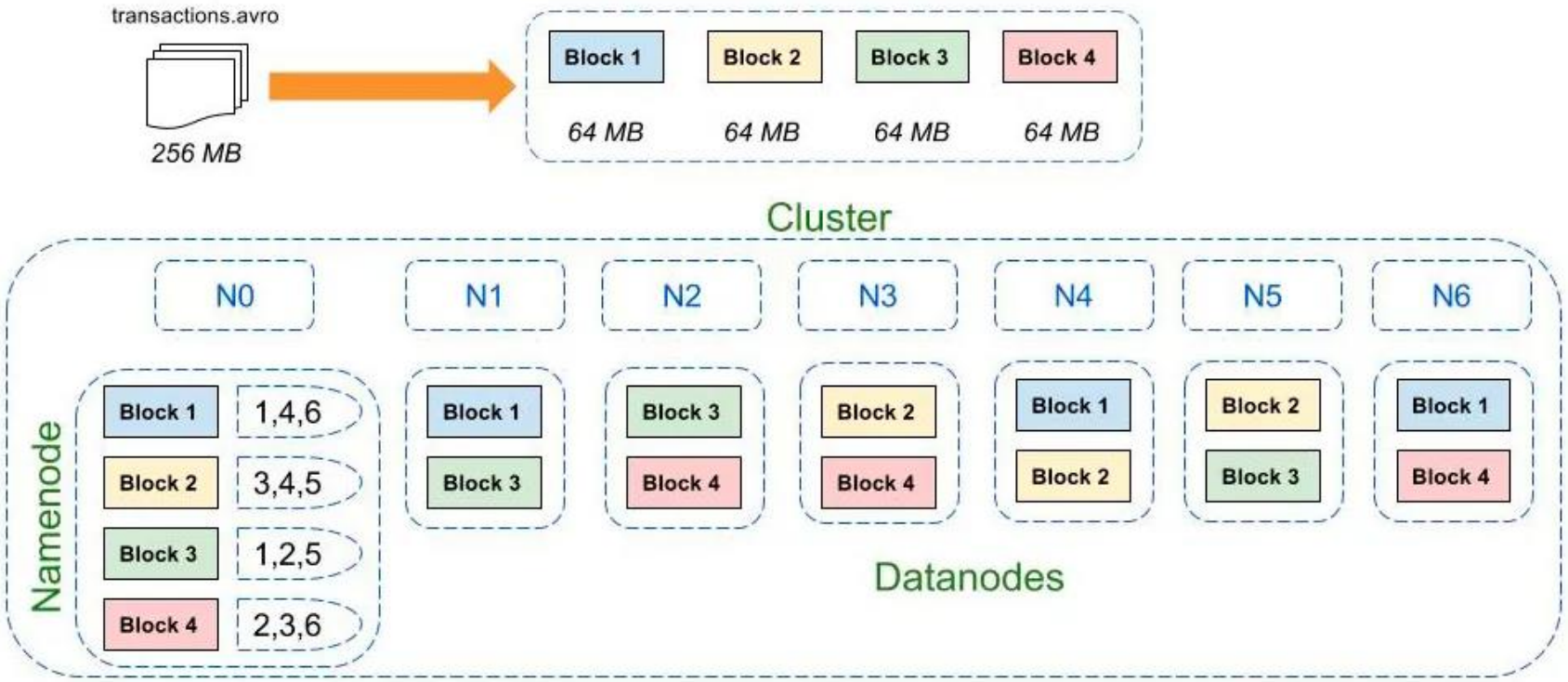
- HDFS **chia nhỏ tệp thành các khối** có kích thước đã được xác định (thường là 64 hoặc 128 MB).
- Mỗi khối **được lưu trữ trên các DataNode khác nhau** trong cụm.
- Mỗi khối được **lưu trữ như một đơn vị riêng biệt** và được **quản lý độc lập bởi các DataNode**.



1.2. Lưu trữ và phân phối các Block dữ liệu

b. Phân phối khối:

- Khi một tệp được ghi vào HDFS, **NameNode xác định các DataNode để lưu trữ từng khối**.
- **Các khối của một tệp được phân phối trên nhiều DataNode**, tăng khả năng song song hóa.
- Các DataNode được chọn dựa trên sự gần gũi với ứng dụng sử dụng dữ liệu hoặc chính sách nhận biết rack.



1.3. Data replication và khả năng chịu lỗi

a. Sao chép (Data replication):

- HDFS **sao chép mỗi khối dữ liệu sang nhiều DataNode khác nhau** trong cụm.
- Yếu tố nhân bản (**replication factor**) có thể tùy chỉnh: số lượng bản sao được tạo ra.
- Việc sao chép cho phép dự phòng dữ liệu và **cho phép khôi phục dữ liệu trong trường hợp DataNode gặp sự cố**.

b. Vị trí bản sao:

- HDFS sử dụng một chính sách đặt các block (**block placement policy**) để xác định nơi lưu trữ bản sao. Chính sách này nhằm mục tiêu **tối đa hóa tính sẵn có của dữ liệu**, **tăng khả năng chịu lỗi** và **tận dụng băng thông mạng**.
- Mặc định, HDFS đảm bảo rằng **các bản sao được đặt trên các rack khác nhau** để giảm thiểu nguy cơ mất dữ liệu trong trường hợp rack gặp sự cố.

1.3. Data replication và khả năng chịu lỗi

c. Quản lý sao chép:

- **NameNode theo dõi vị trí của mỗi khối và các bản sao** của nó trong cụm.

- Khi một **DataNode gặp sự cố** hoặc không khả dụng, **NameNode khởi động sao chép để tạo bản sao mới** trên các DataNode khác (không gặp sự cố).

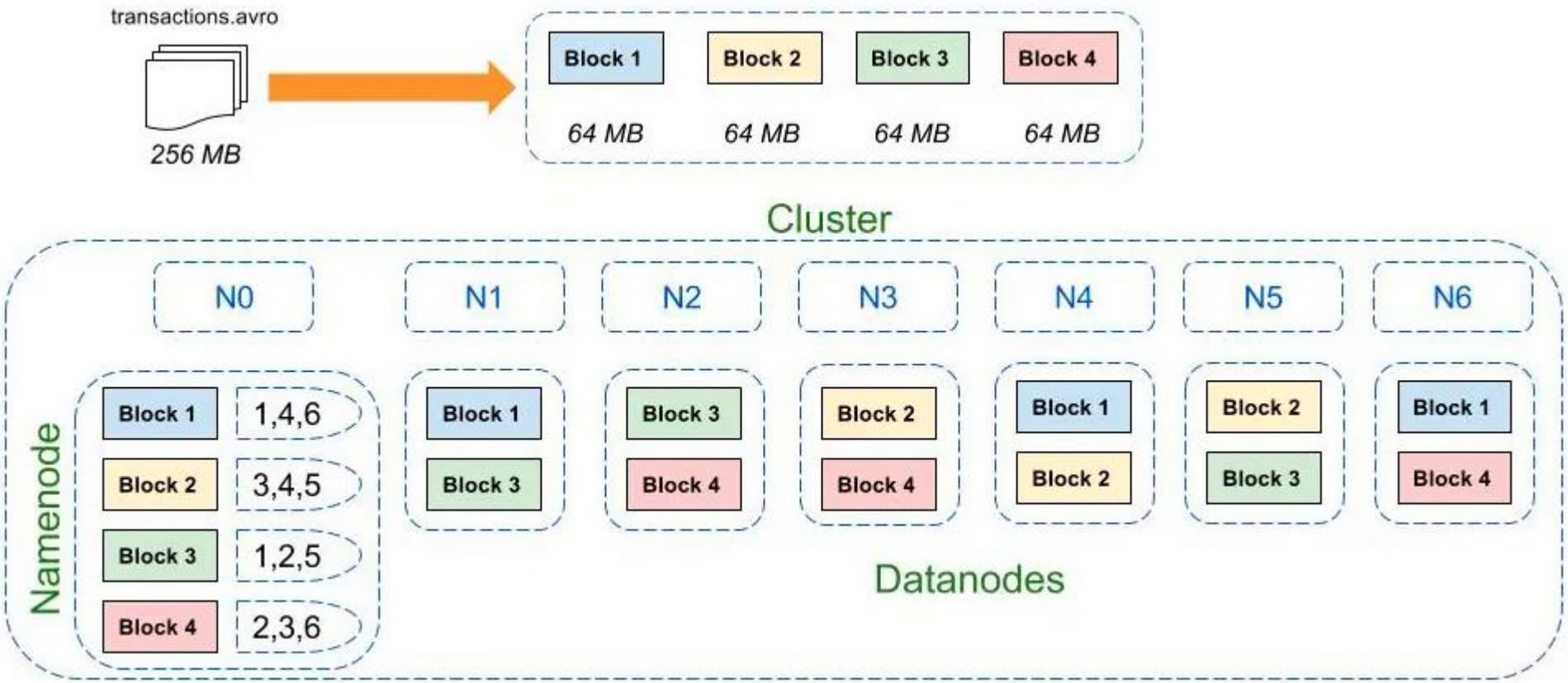
d. Khả năng chịu lỗi:

- HDFS cung cấp khả năng chịu lỗi bằng cách **tự động phát hiện và khôi phục từ sự cố**.

- Việc **lưu trữ nhiều bản sao của các block** tăng cường **tính sẵn có của dữ liệu** khi xảy ra sự cố trên các DataNode.

- Trong trường hợp **NameNode gặp sự cố**, **NameNode dự phòng sẽ được thay thế** để đảm bảo khả năng vận hành liên tục của HDFS cluster.

1.3. Data replication và khả năng chịu lỗi



1.4. Ưu điểm của HDFS

a) Khả năng mở rộng và công suất xử lý cao: Mở rộng dễ dàng theo chiều ngang bằng cách thêm DataNode vào cụm để phân phối dữ liệu và công việc.

b) Data locality và mô hình truy cập dữ liệu:

- **HDFS tận dụng data locality** giúp **tăng tốc độ truy xuất dữ liệu**.
- **Thích hợp cho các công việc cần đọc dữ liệu nhiều lần** (read-intensive) và phù hợp với **mô hình xử lý phân tán**.

c) Xử lý và quản lý dữ liệu quy mô lớn một cách hiệu quả:

- Bằng cách chia tệp thành các khối nhỏ, HDFS cho phép **xử lý song song trên nhiều khối dữ liệu cùng một lúc**.
- HDFS **cung cấp các cơ chế sao chép và phân tán dữ liệu** để **đảm bảo tính toàn vẹn và khả dụng** của dữ liệu.

2. Các kiến trúc lưu trữ dữ liệu phân tán khác

- **Key-value stores**
- **Document stores**
- **Columnar stores**
- **Graph databases**

II. XỬ LÝ DỮ LIỆU PHÂN TÁN

1. Xử lý dữ liệu phân tán với Hadoop

1.1. Mô hình MapReduce

1.2. Xử lý dữ liệu lớn trong hệ sinh thái Apache Hadoop

1.1. MapReduce Paradigm

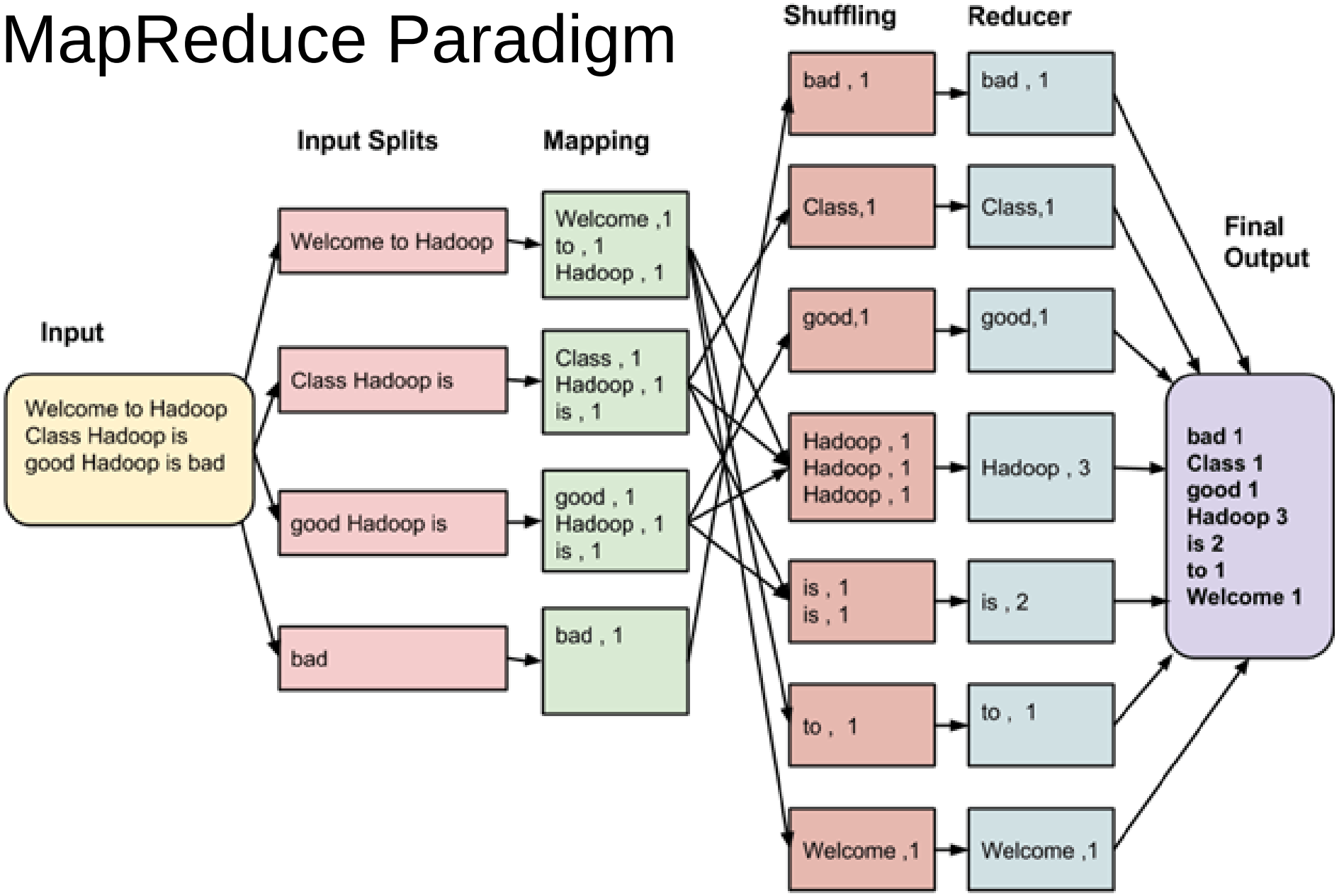
a) Hàm Map (Map function):

- **Áp dụng** một phép toán hoặc biến **đổi cho mỗi phần tử trong một tập hợp dữ liệu**.
- Chia tập dữ liệu thành các phần nhỏ hơn và **xử lý mỗi phần độc lập** => Cho phép song song và đồng thời xử lý dữ liệu.
- **Trả về một tập hợp mới** với các phần tử đã được biến đổi.

b) Hàm Reduce (Reduce function):

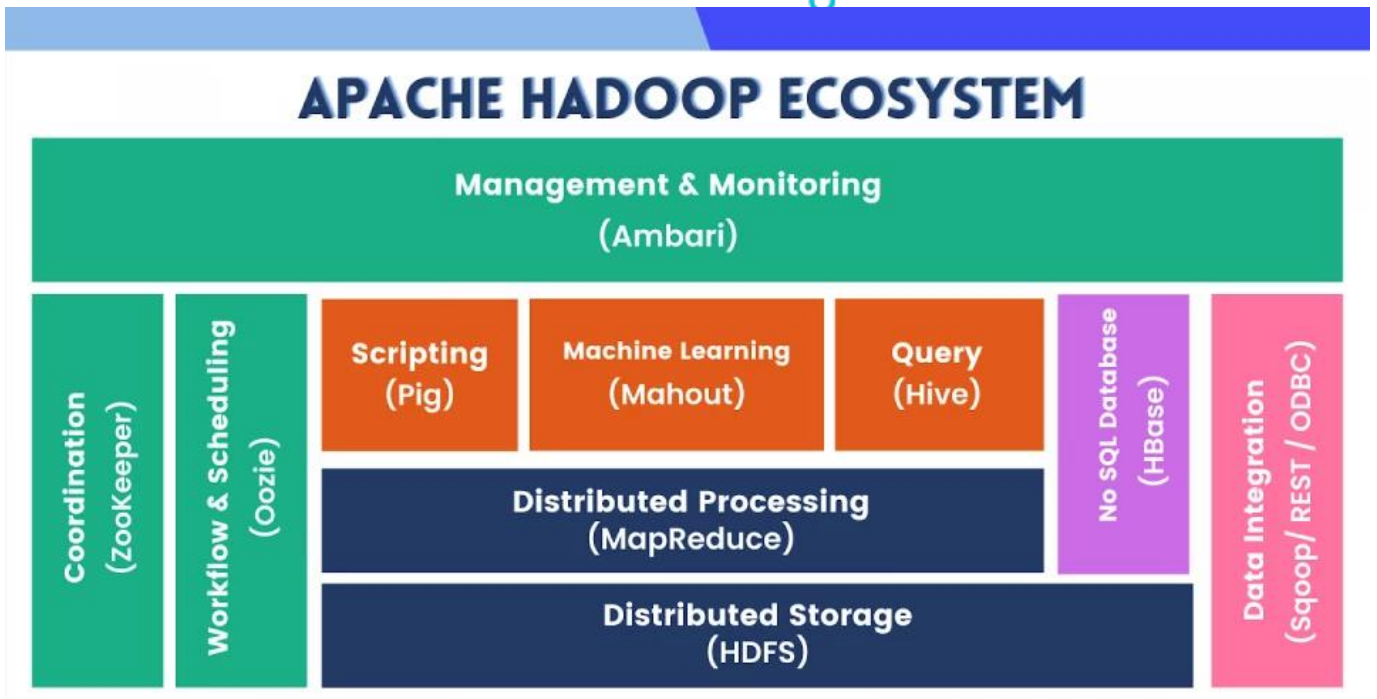
- **Kết hợp hoặc tổng hợp** các phần tử của một tập hợp **thành một giá trị duy nhất**.
- Thường được sử dụng để tính tổng, tìm giá trị lớn nhất/nhỏ nhất hoặc tính toán các thống kê tổng hợp.
- Có thể song song hóa bằng cách tính toán các kết quả phần tử riêng biệt và kết hợp chúng lại.

1.1. MapReduce Paradigm



1.2. Xử lý dữ liệu lớn trong hệ sinh thái Apache Hadoop

- **Apache Hive**: SQL-like query
- **Apache Pig**: High-level data flow language
- **Apache Mahout:** Machine Learning



2. Xử lý dữ liệu phân tán với Spark

2.1. Resilient Distributed Datasets (RDDs)

2.2. Mô hình thực thi Directed Acyclic Graph (DAG)

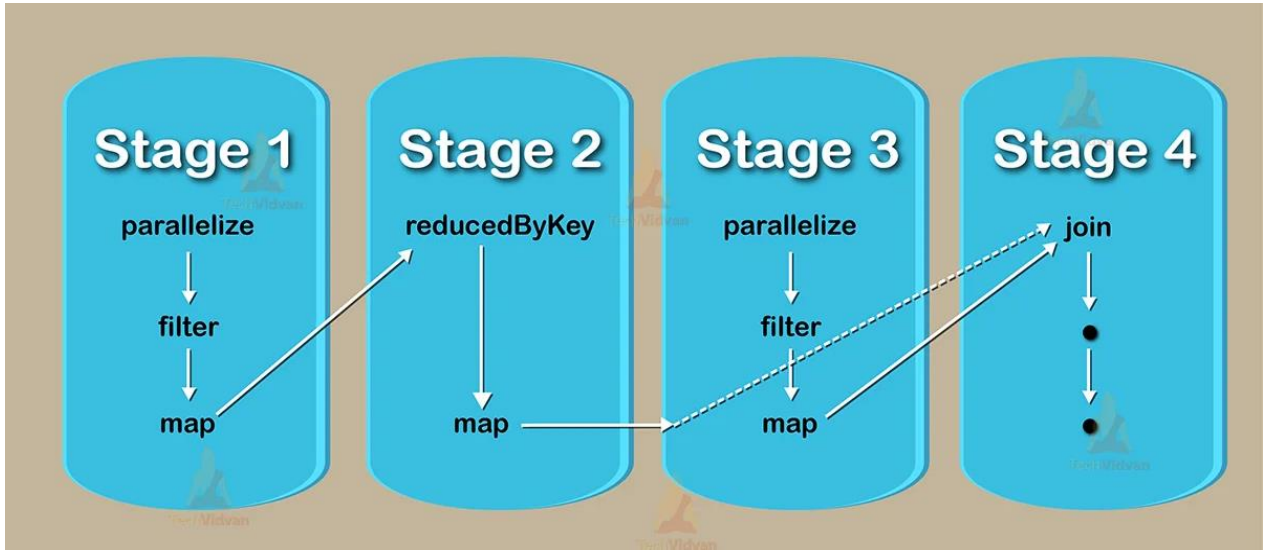
2.3. Các thành phần chính của Spark

2.1. Resilient Distributed Datasets (RDDs)

- **RDD** là **một cấu trúc dữ liệu cơ bản trong Apache Spark**. RDD là tập hợp **các đối tượng được phân vùng, không thể thay đổi (immutable)** và có thể **được xử lý song song trên nhiều nodes** trong một cụm.
- **RDD có thể được tạo từ nhiều nguồn dữ liệu khác nhau như:** HDFS, hệ thống tệp cục bộ (local), hoặc bằng cách biến đổi các RDD hiện có.
- **Đặc trưng của RDDs:**
  - ✓ Xử lý phân tán
  - ✓ Lazy Evaluation
  - ✓ In-Memory Computing
  - ✓ Khả năng phục hồi (Resilience)
  - ✓ Khả năng chịu lỗi

2.2. Directed Acyclic Graph (DAG) execution model

- DAG là một khái niệm cơ bản trong ApacheSpark, **điều phối việc thực thi các tác vụ xử lý dữ liệu** (RDD hay DataFrames).
- Trong mô hình thực thi DAG, Spark xây dựng một **đồ thị có hướng không có chu kỳ (DAG)** của các phép biến đổi được áp dụng cho RDD hoặc DataFrames.
- DAG thể hiện logic thực hiện của các tác vụ tính toán, làm rõ sự phụ thuộc giữa các phép biến đổi khác nhau. **Mỗi nút trong DAG đại diện cho một bước tính toán** và các **cạnh đại diện cho luồng dữ liệu giữa các bước**.



2.2. Directed Acyclic Graph (DAG) execution model

- **Cách thức hoạt động của mô hình thực thi DAG**
  - ✓ **Transformations:** Khi ta sử dụng các phép biến đổi đối với RDDs hay DataFrames, S**park không thực thi các tác vụ này ngay lập tức**. Thay vào đó, Spark ghi lại những phép biến đổi này dưới dạng một chuỗi các tác vụ cần thực thi trên dataset. Những biển đổi này **chỉ thực sự được thực thi khi một action được gọi (lazy evaluated)**.
  - ✓ **Lineage:** Spark lưu thông tin về "lineage" - lịch sử các phép biến đổi được thực hiện đối với RDD hay DataFrame. Thông tin này cho phép Spark **phục hồi những phân hoạch bị mất của RDDs trong trường hợp xảy ra sự cố** (dựa trên dữ liệu hiện có và các phép biến đổi ngược).

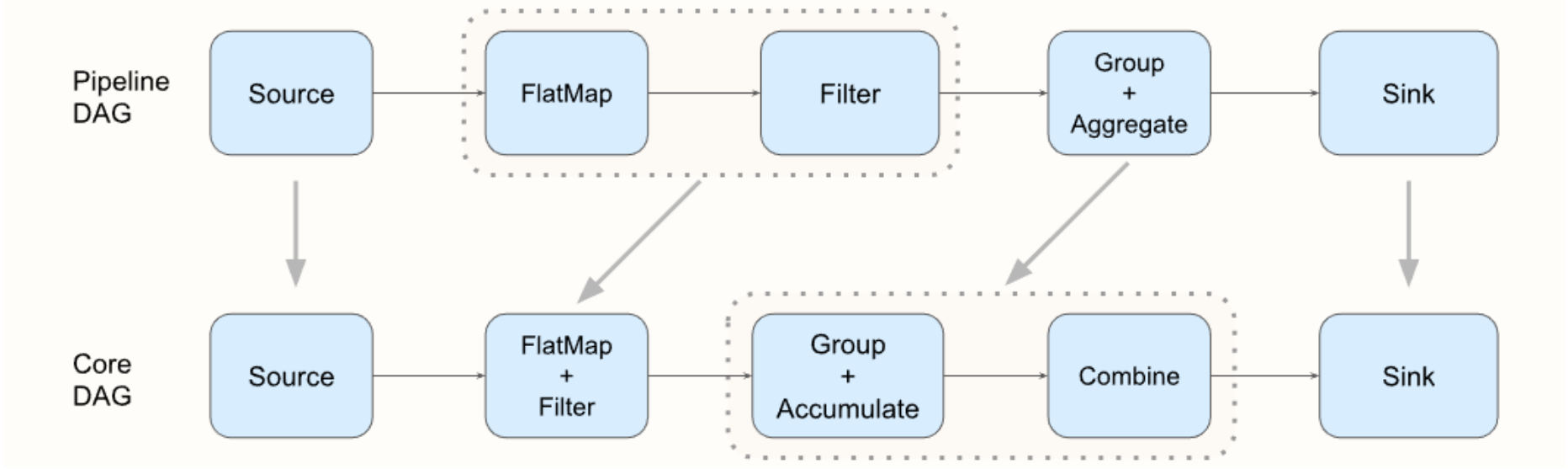
2.2. Directed Acyclic Graph (DAG) execution model

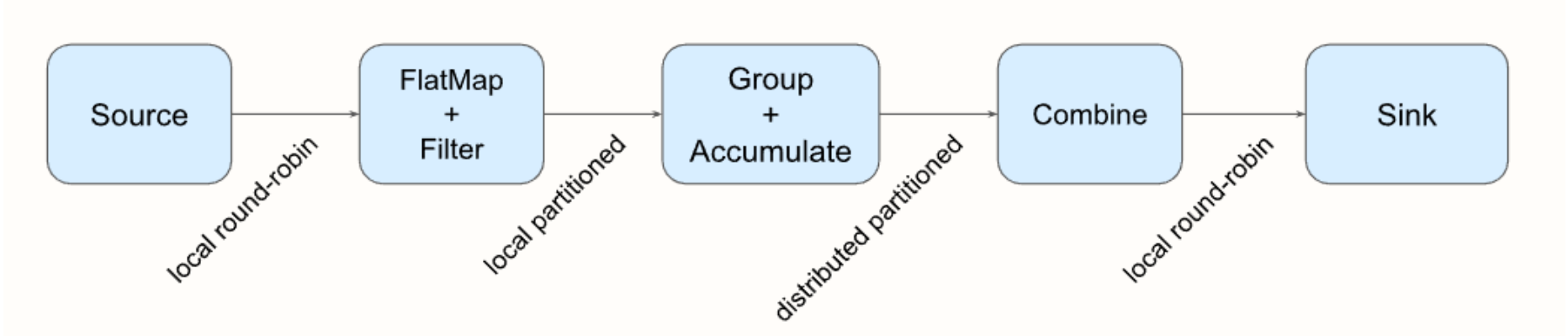
- **Cách thức hoạt động của mô hình thực thi DAG:**
  - ✓ **Optimization: Spark Catalyst Optimizer** sẽ phân tích DAG để **tối ưu hóa trình tự thực thi** các phép biến đổi, **sử dụng nhiều kỹ thuật tối ưu khác nhau** như: predicate pushdown, projection pruning, và join reordering,... Mục tiêu là **giảm thiểu việc shuffling data** và từ đó giúp giảm chi phí tính toán chung.
  - ✓ **Thực thi theo giai đoạn:** Khi một action được kích hoạt, Spark sẽ chia DAG thành một tập hợp các giai đoạn. **Một giai đoạn đại diện cho một tập hợp các phép biến đổi có thể được thực hiện cùng nhau** mà không cần xáo trộn dữ liệu giữa các nodes. Các giai đoạn được xác định dựa trên sự phụ thuộc về dữ liệu (data dependencies) và nhu cầu di chuyển dữ liệu (data movement).

2.2. Directed Acyclic Graph (DAG) execution model

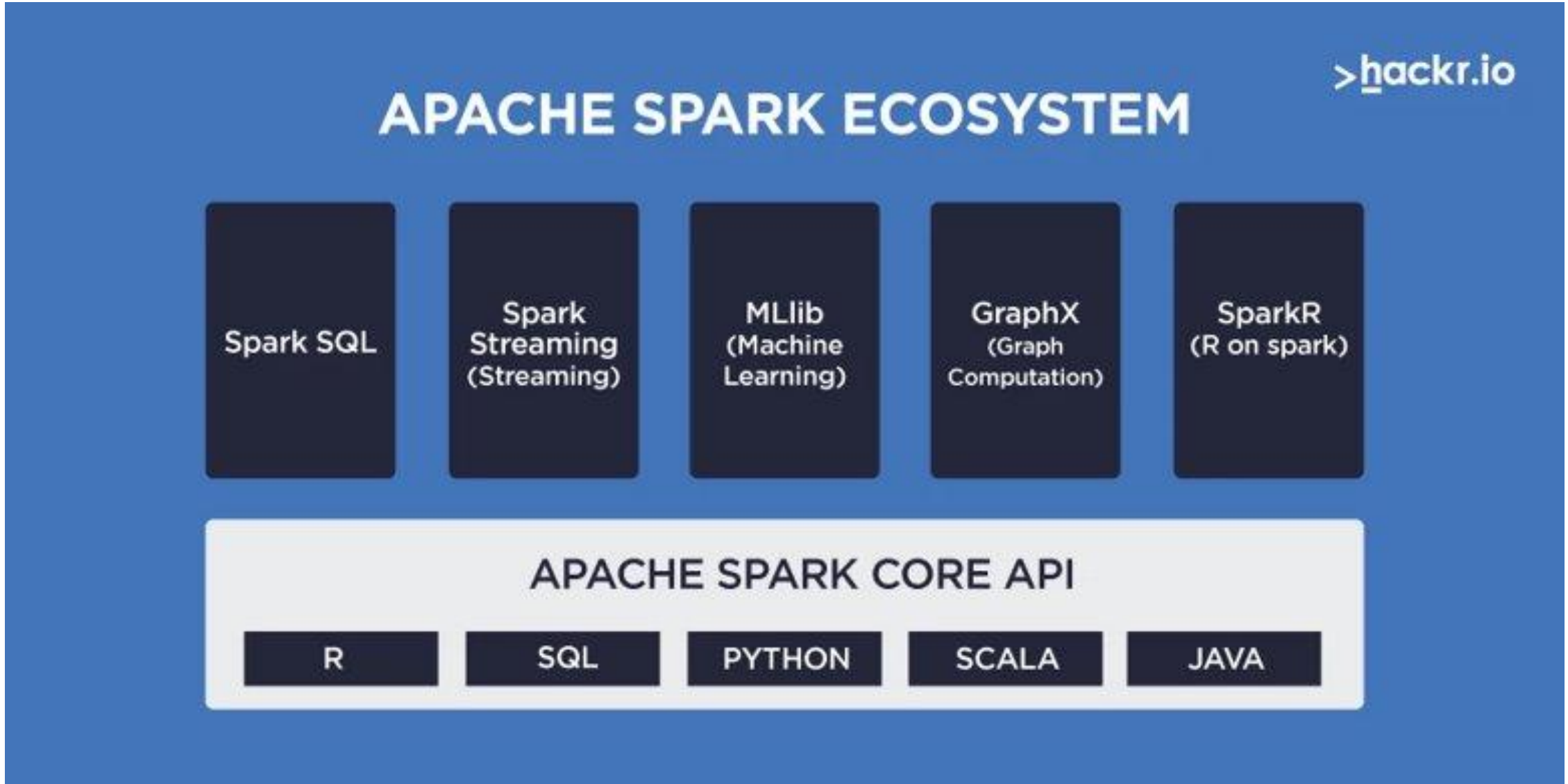
- **Cách thức hoạt động của mô hình thực thi DAG**
  - ✓ **Thực thi tác vụ:** Mỗi giai đoạn lại được chia thành các **tác vụ**, là các công việc riêng lẻ có thể được thực thi trên các nút riêng biệt trong cụm. Spark phân nhiệm vụ cho các nodes dựa trên vị trí dữ liệu (data locality), giảm thiểu việc truyền dữ liệu qua mạng. Các **tác vụ được thực hiện song song trên các nodes khác nhau**, xử lý các phân vùng dữ liệu tương ứng của chúng.
  - ✓ **Thu thập kết quả:** Khi các tác vụ được thực thi, các kết quả trung gian được tính toán và lưu trữ trong bộ nhớ hoặc ổ đĩa, tùy thuộc vào bản chất của dữ liệu và các tài nguyên sẵn có. Cuối cùng, **khi tất cả các tác vụ đã hoàn thành, Spark sẽ thu thập kết quả từ các nodes phân tán** và trả về các kết quả này.

2.2. Directed Acyclic Graph (DAG) execution model





2.3. Các thành phần chính của Spark



III. SO SÁNH HADOOP VÀ SPARK

So sánh Hadoop và Spark

- **Hiệu năng: Spark nhanh hơn** vì sử dụng RAM thay vì đọc ghi từ ổ đĩa. Hadoop lưu trữ dữ liệu trên nhiều nguồn khác nhau và xử lí theo từng batch thông qua MapReduce.
- **Chi phí: Hadoop nhìn chung ít tốn kém hơn** vì nó có thể xử lí dữ liệu trên bất kì dạng ổ đĩa lưu trữ dữ liệu nào. Spark tốn kém hơn vì cần đến tính toán ở bộ nhớ trong (in-memory) để xử lí dữ liệu, đòi hỏi số lượng lớn RAM.
- **Xử lý:** Dù cả 2 đều có thể xử lí dữ liệu phân tán, **Hadoop phù hợp hơn với việc xử lý theo batch và biến đổi dữ liệu tuyến tính**. **Spark phù hợp hơn với việc xử lí real-time và xử lí stream dữ liệu phi cấu trúc**.

So sánh Hadoop và Spark

- **Khả năng mở rộng:** Khi khối lượng dữ liệu tăng nhanh, **Hadoop có thể nhanh chóng mở rộng** để đáp ứng được nhu cầu thông quá HDFS. Còn Spark dựa trên khả năng chịu lỗi của HDFS để tính toán với dữ liệu lớn.
- **Tính bảo mật:** Spark tăng cường tính bảo mật thông qua shared secret hoặc event logging, trong khi Hadoop lại sử dụng các phương thức xác thực và kiểm soát truy cập. Vì vậy, **nhìn chung Hadoop có tính bảo mật cao hơn** (Spark có thể tích hợp với Hadoop để đạt được mức độ bảo mật cao hơn).
- **Machine learning (ML): Spark hỗ trợ tốt hơn cho các tác vụ ML nhờ vào MLlib** - thực hiện các thuật toán ML trên bộ nhớ trong. MLib hỗ trợ đa dạng các thuật toán từ hồi quy, phân loại, xây dựng pipeline, đánh giá,...

IV. CÀI ĐẶT VÀ DEMO ỨNG DỤNG

1. Hadoop, spark cluster

- Cài đặt **Hadoop cluster** với **1 master** và **2 slaves**
- **Physical server:** 56 x Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz (2 Sockets), 64 GB RAM, 1TB SSD
- **Ảo hóa**: proxmox 7.3, LXC ubuntu 22.04

Hostname	IP	CPU (core)	RAM (GB)	Disk (GB)
hadoop-master	94.10.10.11	2	4	32
hadoop-slave-1	94.10.10.11	2	4	32
hadoop-slave-2	94.10.10.11	2	4	32
spark-master	94.10.10.121	2	4	32
spark-slave-1	94.10.10.122	2	4	32
spark-slave-2	94.10.10.123	2	4	32

1. Hadoop, spark cluster

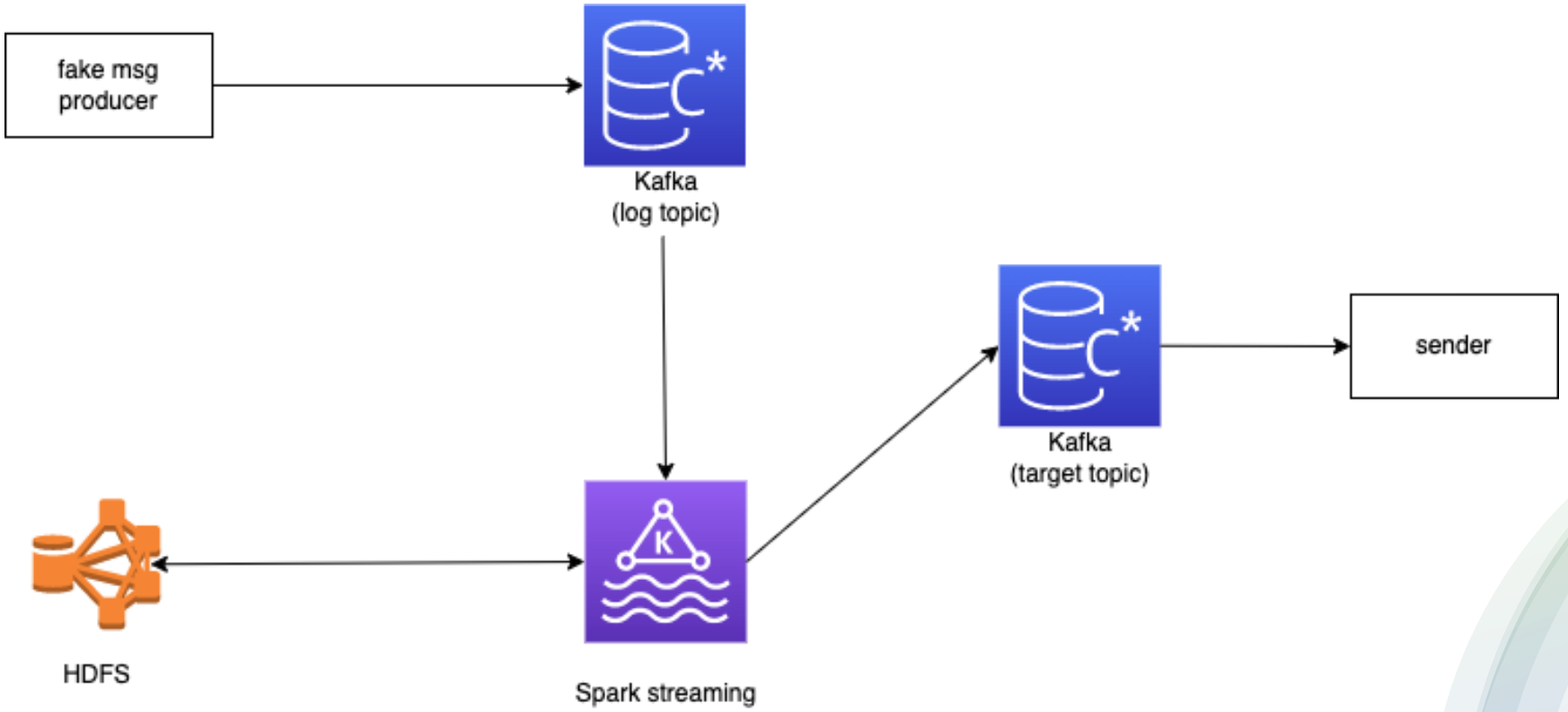
- **Hadoop**: github
- **Spark**: github

2. Real world application - Intro

- **Fenio**
  - Ứng dụng marketing thời gian thực cho phép **nhắn tin đến một tập người dùng đi vào, đi ra hoặc ở trong 1 vùng địa lý** nhất định
  - Nhà quảng cáo cấu hình các features tĩnh (tuổi, giới tính, …) và các features động (1 vùng địa lý, đi vào, đi ra hay ở trong)

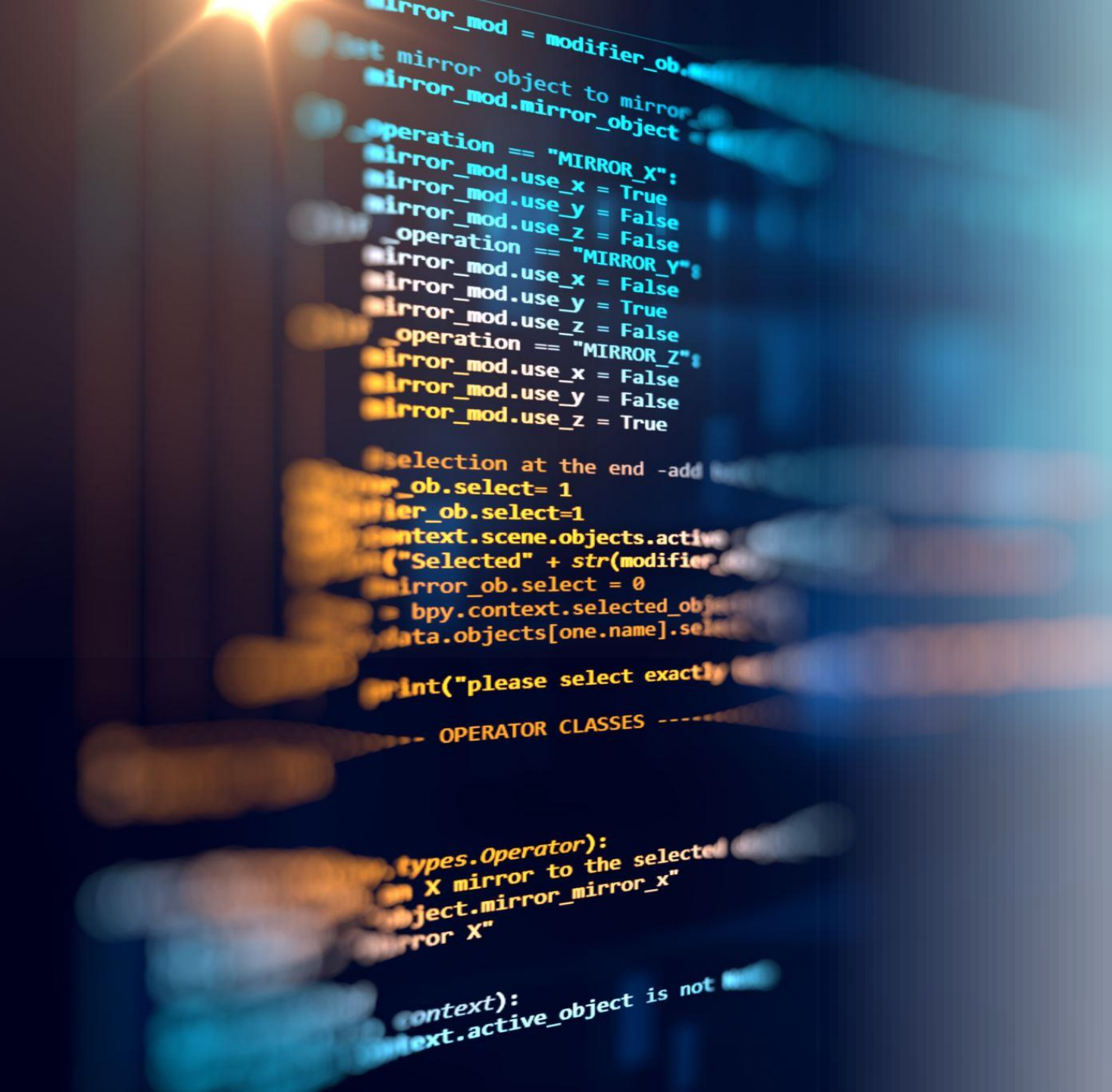
2. Real world application - Architech

- **Architech**



2. Real world application – Data description

- **Static Dataset:**
  - Dữ liệu được sinh ngẫu nhiên gồm các thuộc tính: ID, gender (giới tính, số: 1-Nam, 0-Nữ), birth_year (năm sinh, số: 1960-2000)
- **Streaming data:**
  - ID, source (location trước đó, số: 1-63), target (location hiện tại, số: 1-63)
- **Goal:**
  - Nhắn tin đến những khách hàng là **Nam, đi vào Hà Nội.**



2. Real world application

- **Coding time!**