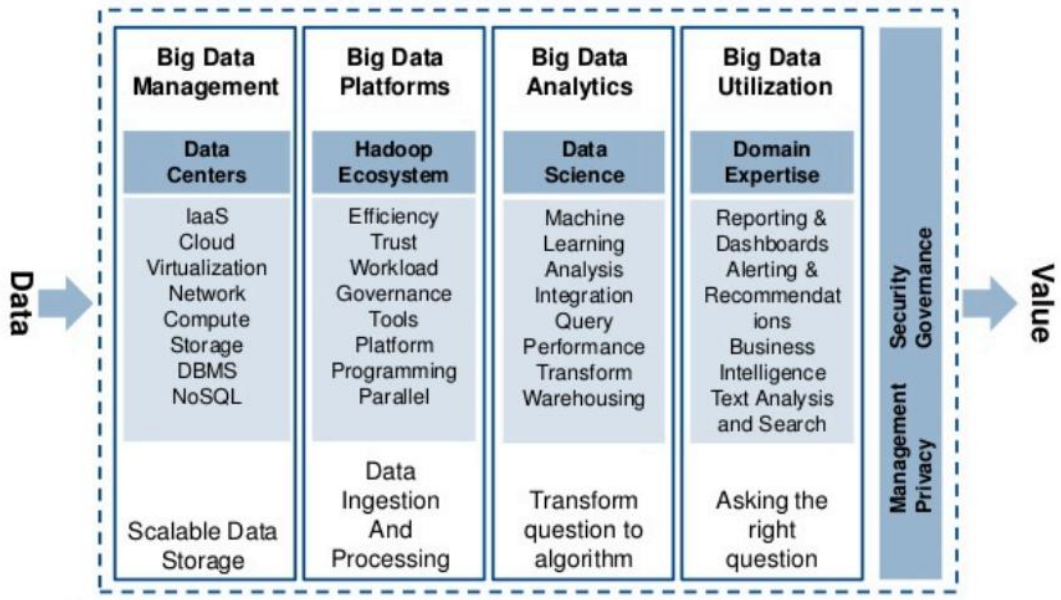
Tổng ôn kiến thức Lưu trữ và Xử lý dữ liệu lớn

I. Giới thiệu về Lưu trữ và xử lý dữ liệu lớn

- Dữ liệu lớn (Big data) là một thuật ngữ cho việc xử lý một tập hợp dữ liệu rất lớn và phức tạp mà các ứng dụng xử lý dữ liệu truyền thống không xử lý được. Dữ liệu lớn bao gồm các thách thức như phân tích, thu thập, giám sát dữ liệu, tìm kiếm, chia sẻ, lưu trữ, truyền nhận, trực quan, truy vấn và tính riêng tư [Wikipedia].
- **5V** trong dữ liệu lớn gồm có:
  - **Velocity** (tốc độ): tốc độ mà dữ liệu được tạo ra và phân tích
  - **Variety** (đa dạng): các kiểu và hình thức DL khác nhau bao gồm lượng lớn DL phi cấu trúc
  - **Value** (giá trị): tiềm năng của dữ liệu lớn đối với phát triển kinh tế và xã hội
  - **Veracity** (xác thực): mức độ về chất lượng, độ chính xác và sự không chắc chắn của dữ liệu và nguồn dữ liệu
  - **Volume** (dung lượng): lượng dữ liệu khổng lồ được tạo ra thông qua quá trình số hóa và dữ liệu hóa thông tin trên quy mô lớn
- **Big data technology stack**



  - Scalable Data Management (Quản lý dữ liệu có thể mở rộng)
    - **Scalability** (khả năng mở rộng): có khả năng quản lý lượng DL ngày càng lớn
    - **Accessibility** (khả năng tiếp cận): có khả năng đọc ghi hiệu quả DL vào hệ thống lưu trữ
    - **Transparency** (trong suốt): truy cập dữ liệu qua mạng dễ dàng, không cần biết vị trí vật lý của DL
    - **Availability** (sẵn sàng): khả năng chịu lỗi (fault tolerance)
  - Scalable data ingestion and processing (Thu thập và xử lý dữ liệu)
    - Data ingestion (thu thập dữ liệu)
    - Data processing (xử lý dữ liệu)

- Scalable analytic algorithms (Thuật toán phân tích)
  - Thách thức: khối lượng lớn, nhiều chiều và xử lý thời gian thực
  - Scaling-up ML algorithms (thuật toán ML mở rộng): xử lý trên một máy duy nhất (in a single machine) và song song (parallelism)
- Làm sạch dữ liệu lớn (**Cleaning big data**): nhiệm vụ KHDL tốn thời gian nhất và ít thú vị nhất
  - **Chuẩn bị dữ liệu (Data preparation)** chiếm khoảng **80%** công việc của các nhà KHDL

II. Hệ sinh thái Hadoop (Hadoop Ecosystem)

- Các thành phần chính của Hadoop:
  - Storage (lưu trữ): Hadoop Distributed File System (HDFS)
  - Processing (xử lý): MapReduce framework
  - System utilities:
    - Hadoop common: hỗ trợ các module khác
    - Hadoop YARN: framework để lập kế hoạch công việc và quản lý tài nguyên cụm
- Scalability: Hadoop có thể chạy trên cụm. Các máy chủ riêng lẻ trong một cụm được gọi nút. Mỗi nút có thể vừa lưu trữ vừa xử lý dữ liệu. Mở rộng quy mô bằng cách thêm nhiều nút hơn.
- Fault tolerance:
  - Tệp được tải vào HDFS được sao chép qua các nút trong cụm. Nếu một nút bị lỗi, dữ liệu của nút đó sẽ được sao chép lại bằng một trong các nút bản sao.
  - Công việc xử lý dữ liệu được chia thành các nhiệm vụ riêng lẻ. Mỗi tác vụ lấy một lượng nhỏ dữ liệu làm đầu vào. Thực thi nhiệm vụ song song. Nhiệm vụ thất bại cũng được lên lịch lại ở nơi khác.
  - Các lỗi thường xuyên (Routine failures) được xử lý tự động mà không làm mất dữ liệu

- **Hadoop Distributed File Systems (HDFS):**

➔ **Kiến trúc của HDFS tuân theo cơ chế Master/Slave architecture**

- **HDFS master: NameNode** có nhiệm vụ quản lý namespace và metadata, giám sát các DataNode
  - Quản lý file system namespace
    - Ánh xạ tên tệp vào một tập hợp các khối
    - Ánh xạ một khối tới các nút dữ liệu nơi nó cư trú
  - NameNode metadata:
    - Toàn bộ metadata nằm trong bộ nhớ chính và không cần phân trang
    - Các kiểu nội dung trong metadata:
      - Danh sách các file
      - Danh sách các block của mỗi file
      - Danh sách các DataNode ứng với mỗi block
      - Kiến trúc file: thời gian tạo, hệ số nhân bản ...
    - Transaction Log: Ghi lại việc tạo tập tin, xóa tập tin ...

        - Quản lý cấu hình cụm (cluster)
        - Công cụ sao chép cho các khối
- **HDFS slaves: DataNode** có nhiệm vụ Xử lý việc đọc/ghi dữ liệu thực tế {chunks}
    - Block server
        - Lưu trữ dữ liệu trong hệ thống tệp cục bộ (ví dụ: ext3)
        - Lưu trữ siêu dữ liệu của một khối (ví dụ CRC)
        - Cung cấp dữ liệu và siêu dữ liệu cho Clients
    - Block report
        - Định kỳ gửi báo cáo về tất cả các khối hiện có tới NameNode
    - Tạo điều kiện thuận lợi cho việc truyền dữ liệu
        - Chuyển tiếp dữ liệu tới các nút dữ liệu được chỉ định khác
    - Heartbeat
        - DataNode gửi heartbeat đến Namenode: cứ 3 giây một lần
        - NameNode sử dụng heartbeat để phát hiện lỗi DataNode

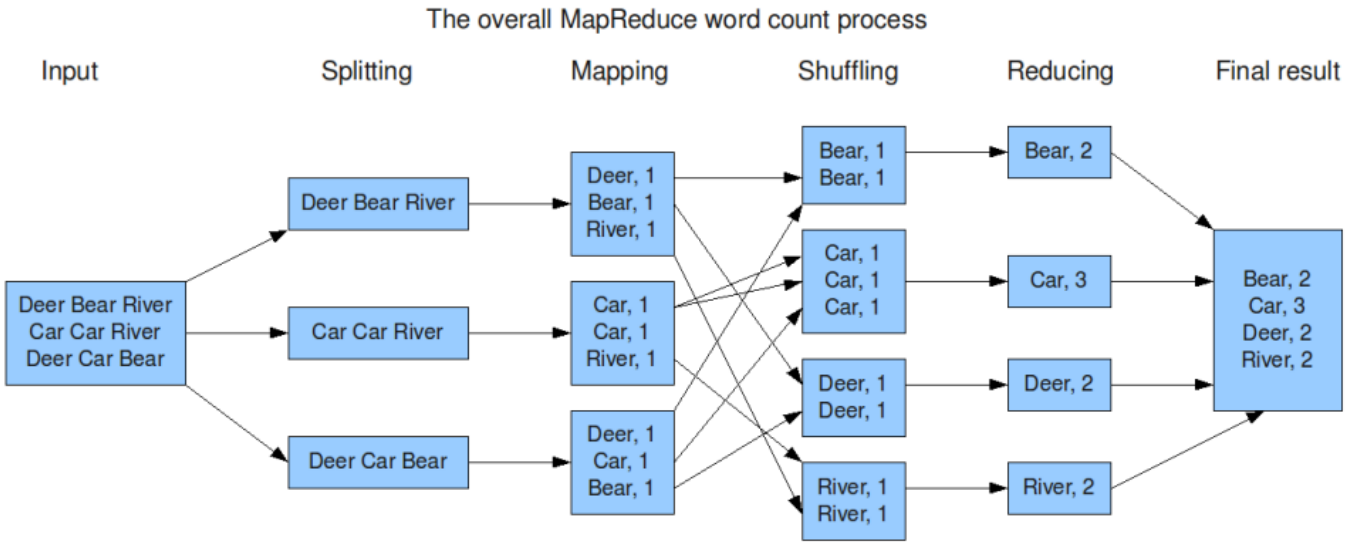
➔ **Nguyên tắc thiết kế chính của HDFS**

- I/O pattern: chỉ nối thêm để giảm thiểu đồng bộ hóa
- Data distribution: file được chia thành các chunk lớn (64MB) để giảm kích thước metadata và giảm truyền thông mạng
- Data replication: mỗi chunk thường được sao chép trên 3 nút khác nhau
- Fault tolerance: DataNode được sao chép, NameNode có Secondary namenode

➔ **Cơ chế hoạt động của HDFS**

- **Quá trình ghi dữ liệu**
    - **Client yêu cầu NameNode**: Khi một ứng dụng hoặc người dùng muốn lưu trữ một file, nó sẽ yêu cầu NameNode cung cấp vị trí của các DataNode khả dụng.
    - **Chia file thành các block**: File được chia thành các block dữ liệu có kích thước cố định (mặc định là 128MB) và mỗi block được lưu trữ trên nhiều DataNode khác nhau (theo mức độ replication).
    - **Ghi block lên DataNode**: Client gửi các block dữ liệu trực tiếp tới các DataNode, đồng thời các DataNode tự động tạo ra bản sao của block để lưu trên các DataNode khác.
- **Quá trình đọc dữ liệu**
    - **Client yêu cầu NameNode**: Client yêu cầu NameNode cung cấp vị trí các block của file trên các DataNode.
    - **Truy xuất dữ liệu**: Dựa trên thông tin vị trí do NameNode cung cấp, client trực tiếp lấy các block từ các DataNode tương ứng và tái cấu trúc lại file.

- **MapReduce framework** (not a language, it's a style of processing data: Simplicity (đơn giản), Flexibility (linh hoạt), Scalability (khả mở))



  - **Splits**: Dữ liệu lớn đầu vào (ví dụ như file, log, hay bảng dữ liệu) được chia thành nhiều phần nhỏ hơn gọi là **Input Splits**. Mỗi split thường là một block dữ liệu trên HDFS (kích thước mặc định là 128MB). Các split này sẽ được xử lý độc lập và song song.
  - **Map**: Mỗi Mapper nhận một Input Split, xử lý từng dòng dữ liệu, và chuyển đổi dữ liệu thành các cặp **key-value** (cặp khóa-giá trị). Đây là cốt lõi của giai đoạn Map: từ dữ liệu thô, Mapper sẽ trích xuất các giá trị có ý nghĩa.
  - **Suffle**: Hệ thống sẽ đảm bảo rằng tất cả các cặp có cùng một key sẽ được gửi đến cùng một Reducer (có thể sử dụng một hàm băm dựa trên key để đưa đúng các cặp vào các node mới). Đây là quá trình shuffle – chuyển dữ liệu giữa các node.
  - **Sort**: Các cặp key-value sẽ được sắp xếp theo thứ tự dựa trên key. Việc sắp xếp giúp giảm bớt gánh nặng cho Reducer và tối ưu hóa quá trình xử lý.
  - **Reducer**: Mỗi Reducer nhận các nhóm cặp key-value có cùng key từ quá trình Shuffle. Với mỗi nhóm, Reducer thực hiện các thao tác tổng hợp hoặc xử lý theo yêu cầu của người dùng.

- **Các thành phần khác trong hệ sinh thái Hadoop**
  - **Hive**: quản lý dữ liệu dựa trên SQL cho Hadoop (**HiveSQL**), phù hợp cho việc xử lý dữ liệu bảng.
  - **HBase**: cơ sở dữ liệu **NoSQL** phân tán và mở rộng, được xây dựng trên HDFS. Nó hỗ trợ lưu trữ dữ liệu có cấu trúc và không cấu trúc với hiệu suất cao, phù hợp cho các ứng dụng yêu cầu truy cập ngẫu nhiên và thời gian thực.
  - **Pig**: cung cấp một ngôn ngữ lập trình kịch bản (**scripting language**) là Pig Latin. Pig phù hợp cho các thao tác phức tạp với dữ liệu bán cấu trúc hoặc không cấu trúc.
  - **Oozie**: công cụ quản lý luồng công việc (**workflow**) và lập lịch công việc.

- **Mahout**: hỗ trợ các thuật toán **học máy** cho Hadoop, bao gồm các thuật toán phân cụm, phân loại, và lọc cộng tác.
- **R Connectors**: cho phép **kết nối** Hadoop với ngôn ngữ lập trình **R**
- **Zookeeper**: dịch vụ đồng bộ hóa phân tán, cung cấp quản lý **cấu hình** và cung cấp khả năng chịu lỗi cho các ứng dụng phân tán trong Hadoop.
- **Sqoop**: dùng để **trao đổi DL** giữa các CSDL quan hệ (như MySQL, PostgreSQL, Oracle) và HDFS.
- **Flume**: thu thập và truyền tải **dữ liệu log theo thời gian thực** vào HDFS. Nó đặc biệt phù hợp cho việc thu thập dữ liệu từ các nguồn như log của web server.

III. Hadoop HDFS

- **Cơ chế Data replication**
  - **Chunk Placement (Vị trí đặt khối dữ liệu):** Khối dữ liệu (block) của một tệp được phân phối trên nhiều DataNode để tăng cường độ tin cậy
    - **One Replica on Local Node:** một bản sao trên nút cục bộ, nơi dữ liệu được tải lên
    - **Second Replica on a Remote Rack:** bản sao thứ 2 trên 1 rack (nhóm các datanode) từ xa.
    - **Third Replica on Same Remote Rack:** bản sao thứ 3 trên rack từ xa cùng bản sao thứ 2 nhưng khác datanode.
    - **Additional Replicas Are Randomly Placed:** các bản sao thêm được đặt ngẫu nhiên.
  - **Clients Read from Nearest Replicas:** Khách hàng đọc từ bản sao gần nhất
  - **Namenode Detects Datanode Failures (Namenode phát hiện lỗi của Datanode)**: Namenode liên tục theo dõi các Datanode trong hệ thống. Nếu một Datanode không phản hồi hoặc báo cáo lỗi, Namenode sẽ phát hiện và ghi nhận Datanode đó bị hỏng.

- **Cơ chế Data Correctness (Tính đúng đắn của dữ liệu)**
  - **Use Checksums to Validate Data:** Sử dụng Checksums CRC32 để xác thực dữ liệu
  - **File Creation (Tạo tệp):** Client tính toán checksum cho mỗi 512 byte. Datanode lưu trữ checksum. Điều này giúp Datanode có thể xác thực dữ liệu khi cần thiết.
  - **File Access (Truy cập tệp):** Khi truy cập tệp trong HDFS, Client lấy dữ liệu và checksum từ Datanode để kiểm tra tính toàn vẹn của dữ liệu nhận được. Nếu việc xác thực thất bại, Client thử với các bản sao khác.

- **Secondary Namenode**
  - Giúp NameNode bằng cách quản lý và thực hiện checkpoint (điểm kiểm tra) các tệp FsImage và Transaction Log.
    - **FsImage**: Đây là ảnh chụp toàn bộ hệ thống tệp tại một thời điểm nhất định.

    - **Transaction Log (Nhật ký giao dịch):** Lưu lại tất cả các thay đổi đã diễn ra sau khi tạo FsImage. Mỗi lần một tệp mới được tạo, thay đổi, hoặc xóa, thông tin về thay đổi đó sẽ được ghi vào Transaction Log.
    - **Secondary NameNode** định kỳ sao chép các tệp FsImage và Transaction Log từ Namenode và lưu vào một thư mục tạm thời để tạo điểm kiểm tra, giúp giảm kích thước của Transaction Log và đảm bảo không bị quá tải.
  - Khi Namenode được khởi động lại (sau sự cố hoặc bảo trì):
    - Kết hợp FsImage và Transaction Log thành một FsImage mới trong thư mục tạm thời
    - Tải tệp FsImage mới lên Namenode
    - Transaction Log trên Namenode được xóa bỏ

- **High Availability (HA):** Để tránh tình trạng bottleneck và đảm bảo tính khả dụng, HDFS hỗ trợ cơ chế dự phòng cho NameNode, với một **Active NameNode** và một **Standby NameNode**. Nếu Active NameNode gặp lỗi, Standby NameNode sẽ thay thế. Cách thức hoạt động như sau:
  - **Namespace và metadata đồng bộ:** Khi Active NameNode thực hiện bất kỳ thay đổi nào đối với metadata (cấu trúc của file hệ thống), các thay đổi này sẽ được ghi vào Journal Nodes thông qua "shared edits". Standby NameNode sẽ liên tục đọc và áp dụng các thay đổi từ Journal Nodes để luôn đồng bộ với Active NameNode.
  - **Zookeeper và Failover Controller:** Zookeeper Service sẽ theo dõi tình trạng của cả Active và Standby NameNode. Nếu Active NameNode gặp sự cố hoặc không phản hồi, Zookeeper sẽ báo cho Failover Controller để thực hiện quá trình chuyển đổi sang Standby NameNode. Sau khi chuyển đổi, Standby NameNode sẽ trở thành Active NameNode và tiếp tục quản lý hệ thống.
  - **DataNode:** Các DataNode được cấu hình để gửi thông tin về khối dữ liệu và heartbeat tới cả Active và Standby NameNode. Điều này đảm bảo rằng khi Standby NameNode trở thành Active, nó có thể ngay lập tức tiếp quản việc quản lý mà không bị mất bất kỳ dữ liệu nào.

- **HDFS data format**
  - *Text file:* CSV, TSV, Json records
  - *Sequence file:* có định dạng nhị phân, lưu trữ dưới dạng Key-Value. Một tệp sequence file có 3 phần chính gồm **Header** (chứa metadata về loại dữ liệu của Key và Value), **Records** (mỗi record chứa một cặp Key và Value) và **Sync Markers (g**iúp truy cập ngẫu nhiên và hỗ trợ khả năng đọc từ giữa tệp). Giúp lưu trữ dữ liệu trung gian trong các job MapReduce
  - *Avro*: có định dạng nhị phân, sử dụng **schema-based serialization**. Một tệp Avro bao gồm **Header (c**hứa schema của dữ liệu (dưới dạng JSON)) và **Data Block** (dữ liệu nhị phân được lưu trữ theo schema đã định nghĩa).
  - *Parquet*: có định dạng lưu trữ dữ liệu dạng **cột (columnar)**, nghĩa là dữ liệu trong tệp được lưu trữ theo từng cột thay vì từng hàng. Một tệp Parquet bao gồm **Footer** (metadata của schema và các block dữ liệu) và **Row Group** (các block chứa dữ liệu, được tổ chức theo cột). Thích hợp cho dữ liệu phân tích lớn, nơi chỉ một số cột được truy vấn thường xuyên.

- *Optimized row columnar (ORC):* có định dạng nhị phân được tối ưu hóa để lưu trữ và truy vấn dữ liệu dạng cột. Tệp ORC bao gồm **Footer** (chứa metadata của schema), **Stripe** (chứa các row group, với dữ liệu lưu theo cột) và **Index Data** (lưu thông tin chỉ mục để tăng tốc độ truy vấn).

Định dạng	Cấu trúc	Tối ưu hóa	Trường hợp sử dụng
CSV/TSV	Dạng dòng	Không	Dữ liệu đơn giản, dễ đọc và chia sẻ
JSON	Dữ liệu bán cấu trúc	Không	API log, dữ liệu không cố định schema
Sequence	Key-Value nhị phân	Có	Lưu trữ tuần tự, MapReduce
Avro	Nhị phân, schema	Có	Chuyển dữ liệu giữa hệ thống, Spark
Parquet	Dạng cột	Có	Phân tích dữ liệu lớn, truy vấn nhanh
ORC	Dạng cột	Cao hơn	Truy vấn Hive, hiệu suất cao

IV. NoSQL

1. Các cơ sở dữ liệu NoSQL

1.1. Key-Value Databases (Redis, DynamoDB, Riak)

a. Kiến trúc & Mô hình

- Mô hình: Dữ liệu được lưu trữ dưới dạng cặp Key-Value, trong đó Key là duy nhất và được sử dụng để truy xuất Value tương ứng.
- Kiến trúc:
  - Redis: Là một in-memory database (cơ sở dữ liệu trong bộ nhớ), thường lưu trữ toàn bộ dữ liệu trong RAM để tăng tốc độ truy xuất. Hỗ trợ persistence thông qua snapshot hoặc append-only file (AOF).
  - DynamoDB: Cơ sở dữ liệu NoSQL được quản lý bởi AWS, hoạt động dựa trên kiến trúc distributed hash table để phân vùng dữ liệu.
  - Riak: Dựa trên kiến trúc masterless replication (không có node chủ), sử dụng giao thức gossip protocol để duy trì tính nhất quán giữa các node.

b. Ưu điểm

- Hiệu suất rất cao (đặc biệt là Redis).
- Dễ dàng mở rộng (scalability) bằng cách thêm node vào cụm.
- Linh hoạt, có thể lưu trữ bất kỳ loại dữ liệu nào (chuỗi, số, object).

c. Nhược điểm

- Chỉ phù hợp cho các ứng dụng đơn giản, không hỗ trợ query phức tạp.
- Không tối ưu cho dữ liệu có quan hệ chặt chẽ (không có JOIN).

d. Trường hợp sử dụng

- Redis: Lưu cache, session, leaderboard trong các ứng dụng game, real-time analytics.
- DynamoDB: Hệ thống e-commerce, quản lý giỏ hàng, log tracking.
- Riak: Lưu trữ metadata trong hệ thống phân tán hoặc ứng dụng IoT.

1.2. Column Family Databases (HBase, Cassandra)

a. Kiến trúc & Mô hình

- Mô hình: Dữ liệu được tổ chức dưới dạng bảng nhưng không giống SQL. Thay vì lưu theo hàng, dữ liệu được lưu theo cột (column family).
- Kiến trúc:
  - HBase: Dựa trên HDFS, cung cấp cơ sở dữ liệu dạng bảng với các cột động, hỗ trợ phân tán ngang.
  - Cassandra: Dựa trên kiến trúc peer-to-peer, trong đó tất cả các node trong cụm đều ngang hàng. Sử dụng gossip protocol để trao đổi trạng thái giữa các node.

b. Ưu điểm

- Tối ưu cho các hệ thống lưu trữ dữ liệu lớn, truy xuất theo khóa chính.
- Dễ mở rộng (horizontally scalable).
- HBase: Tích hợp sâu với hệ sinh thái Hadoop, phù hợp với phân tích dữ liệu lớn.
- Cassandra: Hỗ trợ write-heavy workloads với độ trễ thấp.

c. Nhược điểm

- Học tập và triển khai phức tạp hơn so với SQL hoặc key-value.
- Ít hỗ trợ query phức tạp (không hỗ trợ JOIN hoặc giao dịch ACID toàn diện).

d. Trường hợp sử dụng

- HBase: Phân tích dữ liệu lớn (Big Data Analytics), lưu trữ log hệ thống.
- Cassandra: Ứng dụng thời gian thực, IoT, quản lý log sự kiện.

1.3. Document Databases (MongoDB, CouchDB)

a. Kiến trúc & Mô hình

- Mô hình: Lưu trữ dữ liệu dưới dạng document (tài liệu), thường sử dụng định dạng JSON hoặc BSON. Document chứa các cặp key-value có cấu trúc và không cần schema cố định.
- Kiến trúc:
  - MongoDB: Dựa trên replica set và sharding để phân tán và nhân bản dữ liệu.
  - CouchDB: Dựa trên kiến trúc multi-master replication với giao thức RESTful API.

b. Ưu điểm

- Linh hoạt vì không cần schema cố định (schema-less).
- Dễ dàng lưu trữ dữ liệu bán cấu trúc và phi cấu trúc.
- MongoDB: Hỗ trợ query mạnh mẽ và tích hợp tốt với ứng dụng web.
- CouchDB: Hỗ trợ tốt replication đồng bộ và bất đồng bộ.

c. Nhược điểm

- Hiệu suất kém hơn cho các ứng dụng có nhiều thao tác quan hệ dữ liệu (JOIN).
- Dữ liệu có thể lặp lại (denormalization), gây tăng kích thước lưu trữ.

d. Trường hợp sử dụng

- MongoDB: Quản lý nội dung website, ứng dụng IoT, catalog sản phẩm.
- CouchDB: Ứng dụng offline-first (hỗ trợ offline tốt nhờ replication), lưu trữ cấu hình thiết bị.

1.4. Graph Databases (Neo4j)

a. Kiến trúc & Mô hình

- Mô hình: Lưu trữ dữ liệu dưới dạng đồ thị (graph), với các node đại diện cho thực thể và edge đại diện cho mối quan hệ giữa các thực thể. Mỗi node hoặc edge đều có thuộc tính riêng.
- Kiến trúc:
    - Neo4j: Sử dụng đồ thị định hướng, có công cụ truy vấn mạnh mẽ là Cypher Query Language (CQL).

b. Ưu điểm

- Tối ưu hóa cho các truy vấn phức tạp về mối quan hệ.
- Hiệu suất cao khi làm việc với dữ liệu có nhiều mối quan hệ (không cần JOIN).
- Dễ dàng mở rộng để xử lý các mối quan hệ lớn.

c. Nhược điểm

- Không phù hợp cho các ứng dụng yêu cầu tính toán số lượng lớn hoặc không liên quan đến mối quan hệ.
- Học tập và triển khai khó hơn so với SQL hoặc NoSQL cơ bản.

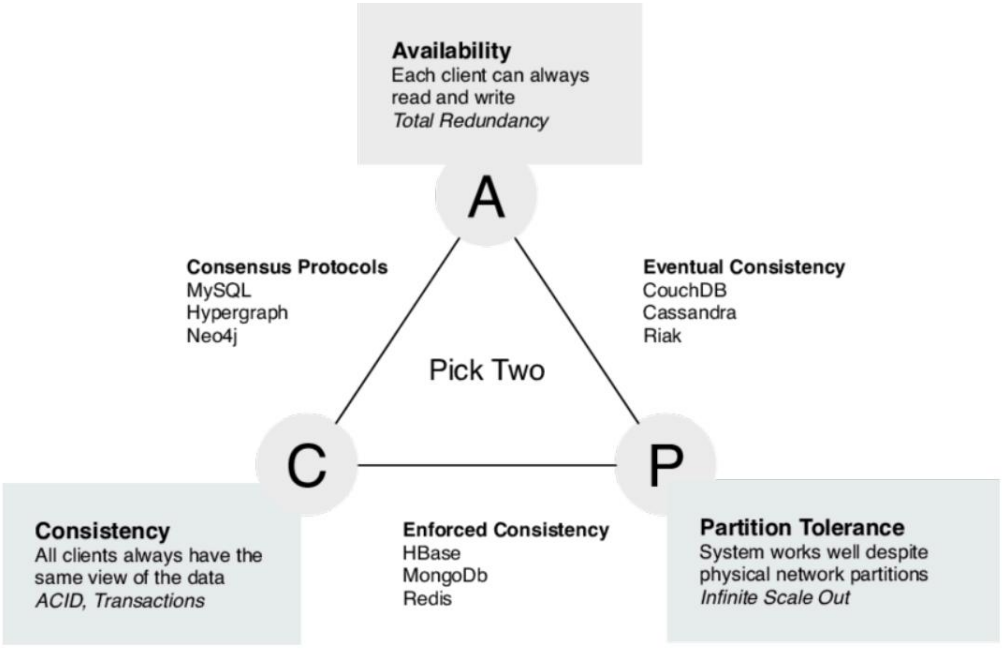
d. Trường hợp sử dụng

    - Mạng xã hội (social network): Phân tích mối quan hệ bạn bè, follower.
    - Phát hiện gian lận: Theo dõi các giao dịch liên quan.
    - Hệ thống gợi ý: Đề xuất sản phẩm hoặc bạn bè dựa trên đồ thị.

Loại CSDL	Ví dụ	Đặc điểm chính	Trường hợp sử dụng
Key-Value	Redis, DynamoDB, Riak	Hiệu suất cao, không cần schema	Cache, session, log tracking
Column Family	HBase, Cassandra	Lưu trữ dạng cột, tối ưu dữ liệu lớn	Big Data, IoT, log storage
Document	MongoDB, CouchDB	Linh hoạt, dữ liệu không cố định schema	Web apps, catalog, offline-first apps
Graph	Neo4j	Tập trung vào mối quan hệ giữa các thực thể	Social networks, fraud detection

2. CAP Theorem

- Những hạn chế của cơ sở dữ liệu phân tán có thể được mô tả trong định lý CAP
  - **Consistency** (Tính nhất quán): mọi nút luôn nhìn thấy cùng một dữ liệu ở bất kỳ phiên bản cụ thể nào (nghĩa là tính nhất quán nghiêm ngặt)
  - **Availability** (Tính sẵn sàng): hệ thống tiếp tục hoạt động ngay cả khi các nút trong cụm gặp sự cố hoặc một số bộ phận phần cứng hoặc phần mềm không hoạt động do nâng cấp
  - **Partition Tolerance** (Dung sai phân vùng): hệ thống tiếp tục hoạt động khi có phân vùng mạng
- **Định lý CAP:** bất kỳ cơ sở dữ liệu phân tán nào có dữ liệu được chia sẻ đều có thể có tối đa hai trong số ba thuộc tính mong muốn là C, A hoặc P. Đây là những sự đánh đổi liên quan đến hệ thống phân tán của Eric Brewer trong PODC 2000.



3. Amazon DynamoDB

- Các đặc điểm cơ bản của DynamoDB
  - Giao diện đơn giản: Lưu trữ khóa/giá trị (**Key – Value**)
  - Hy sinh tính nhất quán mạnh mẽ để có được tính sẵn sàng
  - Kho dữ liệu "luôn có thể ghi (**always writeable**)": không có bản cập nhật nào bị từ chối do lỗi hoặc ghi đồng thời
  - Giải quyết xung đột được thực thi trong quá trình đọc thay vì ghi
  - Cơ sở hạ tầng trong một miền quản trị duy nhất trong đó tất cả các nút được coi là đáng tin cậy
  - Tính đối xứng: Mỗi nút trong Dynamo phải có cùng trách nhiệm như các nút ngang hàng.
  - Tính không đồng nhất: Điều này rất cần thiết trong việc thêm các nút mới có dung lượng cao hơn mà không cần phải nâng cấp tất cả các máy chủ cùng một lúc
- Kiến trúc hệ thống
  - Phân vùng
    - Băm nhất quán (Consitency Hashing): phạm vi đầu ra của hàm băm được coi là không gian hình tròn cố định hoặc "vòng". DynamoDB là DHT zero-hop.
    - Mỗi nút vật lý có thể chịu trách nhiệm cho nhiều hơn một nút ảo. Máy mạnh hơn có nhiều nút ảo hơn.
  - Tính sẵn sàng cao cho việc ghi
    - Vector blocks có sự đối chiếu trong quá trình đọc: Nếu bộ đếm trên đồng hồ của đối tượng thứ nhất nhỏ hơn hoặc bằng tất cả các nút trong đồng hồ thứ hai thì nút đầu tiên là nút tổ tiên của nút thứ hai và có thể bị lãng quên.
  - Xử lý các sự cố tạm thời
    - Sloppy Quorum and hinted handoff: $\boldsymbol{R + W > N}$ trong đó N là tổng số bản sao trên mỗi cặp khóa/giá trị, R là số lượng nút tối thiểu phải tham gia đọc thành công, W là số nút tối thiểu phải tham gia viết thành công.
  - Phục hồi sau những thất bại vĩnh viễn
    - Merkle trees: Cây băm trong đó các lá là giá trị băm của các khóa riêng lẻ. Các nút cha cao hơn trong cây là các giá trị băm của các nút con tương ứng của chúng. Mỗi nhánh của cây có thể được kiểm tra độc lập mà không yêu cầu các nút tải xuống toàn bộ cây, giúp giảm lượng dữ liệu cần truyền trong khi kiểm tra sự không nhất quán giữa các bản sao.
  - Phát hiện tư cách thành viên và lỗi

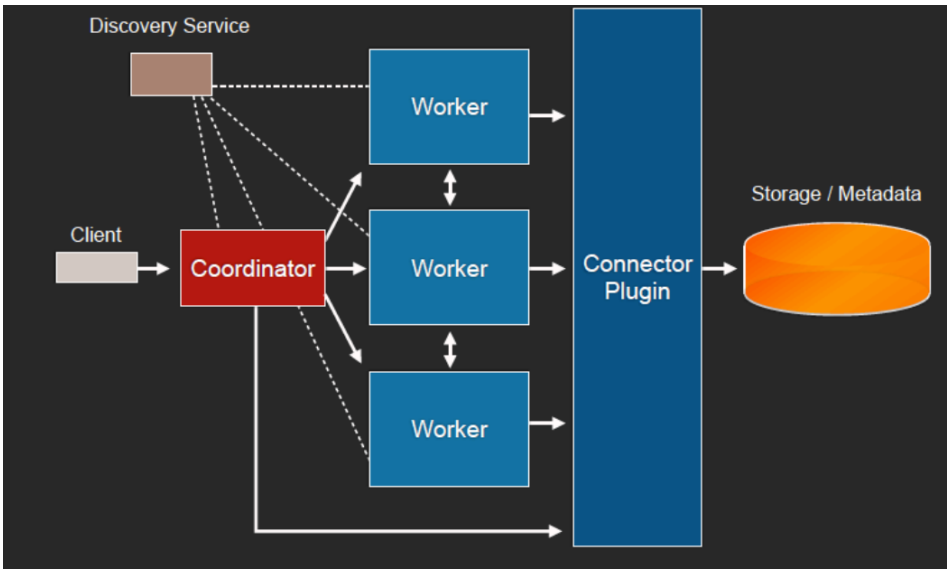
4. Công cụ xử lý truy vấn (Query processing engine)

- Presto là một **công cụ truy vấn SQL phân tán mã nguồn mở** được thiết kế để xử lý các truy vấn tương tác nhanh chóng trên dữ liệu lớn.

- **Công cụ truy vấn SQL mã nguồn mở với hiệu suất cao:** Presto được phát triển để xử lý các truy vấn dữ liệu với hiệu suất cao mà không cần chuyển dữ liệu sang một kho dữ liệu tập trung. Công cụ này hỗ trợ giao diện **SQL tiêu chuẩn ANSI.**
- **Xử lý truy vấn tương tác nhanh:** Presto cho phép xử lý truy vấn trong thời gian thực, từ **mili-giây đến vài phút**, phù hợp cho các tình huống yêu cầu phản hồi nhanh. Tuy nhiên, nó không thay thế hoàn toàn các công cụ như MapReduce hoặc Hive, vốn vẫn cần thiết cho các tác vụ ETL (Extract, Transform, Load) phức tạp.
- **Tích hợp với các công cụ BI thương mại:** Presto hỗ trợ kết nối thông qua **ODBC/JDBC**, giúp các công cụ phân tích kinh doanh (BI) như Tableau, Power BI hoặc các bảng điều khiển tùy chỉnh dễ dàng tích hợp để truy cập dữ liệu.
- **Khả năng truy vấn đa nguồn dữ liệu dựa trên cơ chế Plugin:** Presto hỗ trợ **truy vấn phân tán trên nhiều nguồn dữ liệu khác nhau**, bao gồm hệ thống dữ liệu lớn: Hive, HBase, Cassandra hay cơ sở dữ liệu thương mại: Oracle, MySQL, PostgreSQL.
- **Kết hợp phân tích lô (batch analysis) và trực quan hóa dữ liệu**

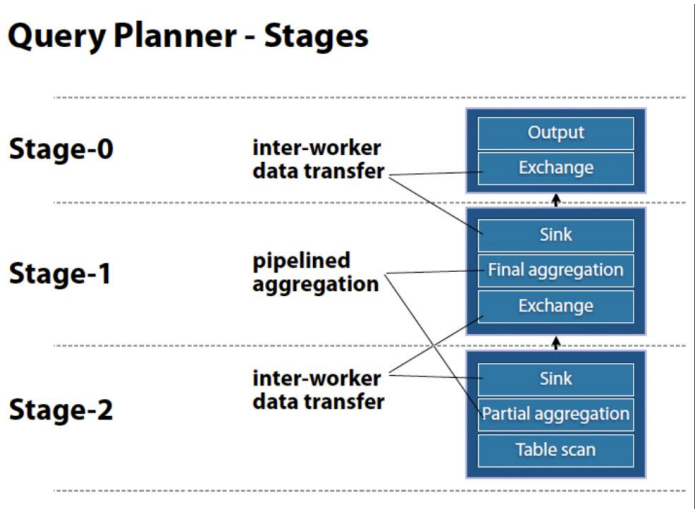
- **Kiến trúc của Presto**

- **Client:** Client là nơi bắt đầu gửi các truy vấn SQL đến hệ thống Presto. Nhận kết quả trả về từ Presto và gửi các câu lệnh truy vấn SQL đến Coordinator để xử lý. Client có thể là người dùng cuối, công cụ BI (Business Intelligence), hoặc các ứng dụng phân tích dữ liệu khác.
- **Coordinator:** Coordinator (bộ điều phối) là thành phần trung tâm trong hệ thống Presto, chịu trách nhiệm điều phối và quản lý các truy vấn. Phân tích câu lệnh SQL, Lập kế hoạch thực thi, Điều phối, Quản lý tài nguyên.
- **Worker:** Các Worker là thành phần thực hiện công việc xử lý dữ liệu thực tế.
- **Discovery Service:** Discovery Service là dịch vụ giúp các thành phần trong hệ thống Presto tìm thấy và giao tiếp với nhau. Duy trì danh sách các Worker đang hoạt động. Đảm bảo rằng Coordinator có thể phân phối công việc đến các Worker phù hợp.
- **Connector Plugin:** Connector Plugin là thành phần chịu trách nhiệm kết nối với các nguồn dữ liệu khác nhau.
    - Tương tác với các hệ thống lưu trữ dữ liệu như Hive, HBase, Cassandra, hoặc các cơ sở dữ liệu thương mại (MySQL, PostgreSQL).
    - Lấy dữ liệu từ các nguồn tương ứng để phục vụ truy vấn.
    - Trừu tượng hóa nguồn dữ liệu để Presto có thể xử lý chúng một cách đồng nhất.
- **Storage/Metadata:** Đây là nơi lưu trữ dữ liệu và thông tin metadata cần thiết cho các truy vấn.



- **Query Planner**

- **Stage – 2:** Đọc và xử lý dữ liệu thô từ nguồn, giảm kích thước dữ liệu bằng cách thực hiện các phép toán tổng hợp cơ bản.
  - **Table scan**: Đọc dữ liệu từ nguồn lưu trữ (ví dụ: Hive, HDFS).
  - **Partial aggregation**: Thực hiện tổng hợp dữ liệu sơ bộ ngay sau khi đọc dữ liệu.
  - **Sink**: Gửi dữ liệu trung gian đã tổng hợp lên các giai đoạn tiếp theo.
- **Stage – 1:** Tối ưu hóa kết quả trung gian từ nhiều nguồn (Workers), đảm bảo rằng dữ liệu được tổng hợp đúng chuẩn trước khi đưa ra kết quả.
  - **Exchange**: Nhận dữ liệu từ giai đoạn trước (Stage-2) thông qua giao tiếp giữa các Worker.
  - **Final aggregation**: Thực hiện tổng hợp DL cuối cùng, kết hợp từ nhiều Worker khác nhau.
  - **Sink**: Chuẩn bị dữ liệu để chuyển tiếp lên giai đoạn cuối cùng.
- **Stage – 0:** Kết hợp all DL từ các Worker, đảm bảo kết quả cuối cùng sẵn sàng để trả về cho Client.
  - **Exchange**: Thu thập dữ liệu đã tổng hợp từ giai đoạn trước (Stage-1).
  - **Output**: Xuất dữ liệu cuối cùng ra cho người dùng hoặc ứng dụng.



- **Query Optimization**
  - Cải tiến hiệu suất truy vấn bằng cách tối ưu hóa bố cục dữ liệu (data layouts)
    - Tận dụng bố cục vật lý của dữ liệu: Bộ tối ưu hóa (optimizer) sử dụng thông tin về cách dữ liệu được lưu trữ để cải thiện hiệu suất xử lý truy vấn. Các kỹ thuật Partitioning, Sorting, Grouping và Indexes
    - Đa dạng bố cục dữ liệu: Một bảng có thể có nhiều cách bố trí dữ liệu (layouts) khác nhau
    - Lựa chọn bố cục tốt nhất: Bộ tối ưu hóa tự động chọn bố cục dữ liệu phù hợp nhất cho từng truy vấn, dựa trên loại truy vấn (lọc, nhóm, sắp xếp, v.v.), các cột và thuộc tính được truy vấn.
    - Điều chỉnh truy vấn bằng cách tạo bố cục mới
  - Tối ưu hóa bằng cách đẩy các điều kiện lọc (predicate) xuống cấp thấp hơn của hệ thống, giúp giảm khối lượng dữ liệu cần xử lý.
    - Đẩy điều kiện lọc xuống connector: Điều kiện lọc (predicates) như **range** (phạm vi) hoặc **equality** (so sánh bằng) được đẩy xuống tầng **connector** (tầng giao tiếp với nguồn DL).
    - Bộ lọc hai phần (Two-part constraint):
      - **Domain of values** nhằm xác định phạm vi giá trị hợp lệ (domain)
      - **"Black box" predicate l**à một bộ lọc tùy chỉnh được áp dụng ở tầng connector, không bị ràng buộc bởi các quy tắc cụ thể của engine, cho phép connector tự quyết định cách tối ưu hóa hiệu quả nhất.

V. Kafka

Toàn bộ nội dung nằm trong file "Kafka với Spring Boot"

VI. Batch processing

5. MapReudce

- **Sorting algrithm**

- Được sử dụng để kiểm tra tốc độ tho ocura Hadoop
- Đầu vào: Một tập hợp các tập tin, một giá trị trên mỗi dòng. Mapper key là tên file, số dòng. Giá trị của Mapper là nội dung của dòng.
- Các cặp (key, value) từ mapper được chuyển đến reducer thông qua hàm hash(key). Phải chọn hàm băm sao cho $k1 < k2 => hash(k1) < hash(k2)$

- Search algorithm

- Thuật toán này được sử dụng để tìm kiếm các dòng trong tập hợp file văn bản phù hợp với một mẫu (pattern) cụ thể. Mục tiêu là xác định những file nào chứa thông tin quan trọng (interesting files).
- Đầu vào: Một tập tin chứa các dòng văn bản và Một mẫu tìm kiếm để tìm.
- Mapper key là tên file, số dòng. Giá trị của Mapper là nội dung của dòng. Mẫu tìm kiếm được gửi dưới dạng tham số đặc biệt.
- Với mỗi cặp (filename, line_content), kiểm tra xem line_content có khớp với pattern hay không.
- Trong quá trình Mapper phát ra kết quả (filename, _), nếu nhiều dòng trong cùng một file phù hợp với mẫu tìm kiếm, Mapper sẽ phát ra nhiều bản ghi (filename, _). Combiner được sử dụng để giảm dữ liệu dư thừa ngay trong mỗi Mapper bằng cách nhóm các bản ghi (filename, _) tương tự thành một bản ghi duy nhất.

- TF – IDF

- TF-IDF xác định mức độ quan trọng của một từ trong một tài liệu cụ thể trong mối quan hệ với toàn bộ tập hợp tài liệu (corpus). Công thức cụ thể của TF – IDF như sau:

$$tf(t,d) = \frac{f(t,d)}{\max\{f(w,d): w \in d\}};\ idf(t,D) = log(\frac{|D|}{|d \in D: t \in d|});\ tf - idf = tf \times idf$$

- TF-IDF được tính qua **4 job chính**, mỗi job tương ứng với một bước trong quá trình xử lý:
  - Job 1: Tính tần suất từ trong từng tài liệu (Word Frequency in Each Document)
    - Mapper nhận input là $(docname, contents)$, thực hiện tách nội dung của tài liệu thành các từ → $((word, docname), 1)$
    - Reducer nhận các cặp $((word, docname), 1)$ từ mapper và tính tổng số lần xuất hiện → $((word, docname), n)$
  - Job 2: Tính tổng số từ trong mỗi tài liệu
    - Mapper nhận input là $((word, docname), n)$ → $(docname, (word, n))$
    - Reducer nhận $(docname, [(word1, n1), (word2, n2), \dots])$ từ mapper và tính tổng số từ $N = n1 + n2$ ... → $((word, docname), (n, N))$
  - Job 3: Tính số tài liệu chứa từng từ
    - Mapper nhận input là ((word, docname),(n,N)) → (word, (docname, n, N, 1)).
    - Reducer nhận input (word,[(docname1,n1,N1,1),(docname2,n2,N2,1),...]) từ Mapper sau khi Combiner, tính số tài liệu d → ((word,docname),(n,N,d))

➢ Job 4: Tính TF-IDF

- Mapper nhận input là $((word, docname), (n, N, d))$ và tính $TF = n/N$ và $IDF = D/d$ → ((word,docname),TF×IDF)
- Reducer nhận kết quả từ mapper và phát ra đầu ra

- Các tối ưu hóa
  - ➢ **Gộp Job 3 và Job 4**: Kết hợp tính d và TF−IDF trong cùng một Reducer để giảm số lần đọc/ghi dữ liệu.
  - ➢ **Sử dụng Combiner**: Dùng Combiner để giảm số lượng bản ghi (word,docname,1) trước khi đến Reducer trong Job 3.
  - ➢ **Quản lý bộ nhớ**: Khi dữ liệu quá lớn, cần đảm bảo sử dụng bộ nhớ phẳng (flat memory usage) để tránh tình trạng quá tải.

- Best First Search

- Lặp lại qua MapReduce – ánh xạ một số nút, kết quả bao gồm các nút bổ sung được đưa vào các bước MapReduce tiếp theo.

- PageRank

- Ý tưởng của PageRank: Nếu một người dùng bắt đầu tại một trang web ngẫu nhiên và duyệt bằng cách nhấp vào các liên kết hoặc nhập ngẫu nhiên URL mới, xác suất họ đến một trang cụ thể là gì? PageRank của một trang nắm bắt ý niệm này. Các trang "phổ biến" hoặc "có giá trị" hơn sẽ nhận được thứ hạng cao hơn.
- Công thức của PageRank với trang A và các trang T_1 đến T_n liên kết đến A. Trong công thức này C(P) là tổng số liên kết ra từ trang P (out-degree) và d là hệ số giảm (tính "nhảy URL ngẫu nhiên").

$$PR(A) = (1-d) + d\left(\frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_n)}{C(T_n)}\right)$$

- Chi tiết cách MapReduce thực hiện thuật toán PageRank
  - ➢ **Giai đoạn 1: Phân tích HTML**
    - **Mục đích:** Chuẩn bị dữ liệu, phân tích đồ thị web để tạo ra cấu trúc cần thiết cho PageRank.
    - **Đầu vào:** Dữ liệu dạng (URL, nội dung trang).
    - **Tác vụ Map:** Phân tích nội dung trang để xác định danh sách các URL mà trang hiện tại trỏ đến. Tạo đầu ra ở dạng: $(URL, (PR_{init}, danh\ sách\ URL))$ trong đó PR_{init} là giá trị PageRank khởi tạo (thường đặt là 1/N, với N là tổng số trang), Danh sách URL là các trang mà URL hiện tại trỏ đến.
    - **Tác vụ Reduce:** Chỉ thực hiện hàm đồng nhất, tức là không thay đổi dữ liệu.
  - ➢ **Giai đoạn 2: Phân phối PageRank**
    - **Mục đích:** Phân phối giá trị PageRank từ một trang đến các trang mà nó liên kết đến.

- **Đầu vào:** Dữ liệu từ giai đoạn 1: $(URL, (PR_{cur}, danh\ sách\ URL))$.
- **Tác vụ Map:** Với mỗi URL trong danh sách liên kết, tính toán giá trị PageRank phân phối: $PR_{phân\ phối} = \frac{PR_{cur}}{|danh\ sách\ URL|}$ và xuất dữ liệu dạng $(u, PR_{phân\ phối})$ với u là một URL trong danh sách liên kết. Đảm bảo xuất thêm cặp (URL, danh sách URL) để giữ lại cấu trúc đồ thị.
- **Tác vụ Reduce:** Nhận dữ liệu từ Map là một URL cụ thể có thể nhận nhiều giá trị $PR_{phân\ phối}$. Tính tổng tất cả các giá trị $PR_{phân\ phối}$: $PR_{mới} = (1-d) + d.\sum PR_{phân\ phối}$ với d là hệ số giảm (thường là 0.85). → Xuất dữ liệu: $(URL, (PR_{mới}, danh\ sách\ URL))$

➢ **3. Lặp lại đến khi hội tụ**

- **Điều kiện hội tụ:** Giá trị PageRank giữa hai lần lặp liên tiếp không thay đổi đáng kể (dưới một ngưỡng ε). Hoặc sau một số vòng lặp cố định.
- Nếu chưa hội tụ, đầu ra từ Giai đoạn 2 sẽ được sử dụng làm đầu vào cho một vòng lặp mới.

6. Apache Spark

- Resilient Distributed Dataset (RDD)

- RDD là **cấu trúc dữ liệu phân tán, chịu lỗi** (fault-tolerant, distributed data structure).
- Cho phép người dùng:
  - ➢ **Lưu trữ tạm thời (persist)** các kết quả trung gian trong bộ nhớ.
  - ➢ **Kiểm soát cách phân vùng dữ liệu** (partitioning) để tối ưu hóa vị trí dữ liệu và xử lý.
  - ➢ Sử dụng các **toán tử phong phú** (rich set of operators) để thao tác dữ liệu (như map, filter, join).
- **Tính năng chính của RDD:**
  - ➢ **Chịu lỗi (Fault-tolerant):** Có khả năng phục hồi dữ liệu dựa trên lineage (dòng truy xuất nguồn gốc dữ liệu).
  - ➢ **Phân tán (Distributed):** Dữ liệu được phân bổ trên nhiều nút (node) trong cluster.
  - ➢ **Xử lý song song (Parallel Processing):** Áp dụng cùng một thao tác cho nhiều mục dữ liệu cùng lúc.

- Các phép biến đổi trong RDD

- **Biến đổi thô (Coarse-grained transformations):** Là các phép biến đổi áp dụng trên toàn bộ tập dữ liệu, áp dụng **cùng một thao tác** cho nhiều mục dữ liệu. Ví dụ: map, filter, join.
- **Cập nhật tinh (Fine-grained updates):** Là các cập nhật chi tiết, thực hiện trên các phần tử cụ thể của tập dữ liệu. RDD **không hỗ trợ** cập nhật tinh vì nó không phù hợp cho xử lý phân tán.

- **Hai loại thao tác trên dữ liệu phân tán**
  - **Phép biến đổi (Transformations):**
    - **Đặc điểm:**
      - **Lazy (trì hoãn):** Không thực thi ngay khi được gọi, chỉ thực thi khi có một phép hành động (Action) được gọi.
      - Cho phép tối ưu hóa việc xử lý bằng cách **gộp các phép biến đổi** trước khi thực thi.
    - **Ví dụ: map b**iến đổi từng phần tử trong tập dữ liệu, **filter** lọc các phần tử theo điều kiện, **join** kết hợp hai tập dữ liệu dựa trên khóa chung.
  - **Phép hành động (Actions):**
    - **Đặc điểm:**
      - Là các thao tác kích hoạt việc thực thi các phép biến đổi.
      - Trả kết quả về driver hoặc ghi dữ liệu ra bộ nhớ/disk.
    - **Ví dụ: collect t**hu thập tất cả các phần tử của RDD về driver, **count đ**ếm số lượng phần tử trong RDD, **saveAsTextFile l**ưu RDD ra file văn bản.

- **DAG (Directed Acyclic Graph - Đồ thị có hướng không chu trình)**
  - DAG là một biểu diễn logic của các thao tác được thực hiện trên dữ liệu và cách dữ liệu luân chuyển giữa các bước xử lý.
  - DAG được sử dụng để biểu diễn và tối ưu hóa các thao tác trên RDD trong Spark. Khi thực hiện các thao tác như map, filter, hoặc join trên RDD, Spark xây dựng một DAG để mô hình hóa thứ tự và mối quan hệ giữa các thao tác này.
  - DAG gồm các nút (Node) đại diện cho một tập dữ liệu (RDD) hoặc một bước xử lý cụ thể và cạnh (Edge) đại diện cho mối quan hệ giữa các nút, tức là dữ liệu được truyền từ một bước xử lý sang bước kế tiếp.
  - Quá trình hoạt động của DAG trong Spark
    - Xây dựng DAG:
      - Khi định nghĩa các thao tác trên RDD (transformations), Spark không thực thi ngay mà chỉ xây dựng DAG để biểu diễn các phép biến đổi cần thực hiện.
    - Tối ưu hóa DAG:
      - Spark tối ưu hóa DAG để giảm thiểu số lần đọc/ghi dữ liệu vào bộ nhớ hoặc đĩa.
      - Các phép biến đổi được gộp lại thành các stages (giai đoạn), mỗi stage tương ứng với một nhóm các phép tính có thể thực hiện song song.

    - Thực thi DAG:
      - Khi một action (như collect hoặc saveAsTextFile) được gọi, Spark kích hoạt việc thực thi DAG.
      - DAG được chia thành nhiều stages và mỗi stage được thực thi trên các worker nodes.

- **Quá trình lưu trữ và chịu lỗi**

  - **Quá trình lưu trữ dữ liệu RDD:**
    - **RDD ban đầu** được tải từ hệ thống tệp phân tán như HDFS hoặc nguồn dữ liệu khác (S3, Cassandra, v.v.).
    - **RDD trung gian** được lưu trữ trong RAM để tăng tốc độ truy xuất, hoặc lưu trên disk nếu RAM không đủ.
  - **Cơ chế chịu lỗi (Fault-tolerance)** → Dựa trên **lineage (dòng truy xuất nguồn gốc):**
    - Lineage là thông tin về các phép biến đổi được áp dụng để tạo ra một RDD từ dữ liệu ban đầu.
    - Nếu một phần dữ liệu bị mất, Spark có thể tái tạo lại RDD từ các phép biến đổi đã lưu trữ trong lineage.

  Ví dụ: Nếu RDD1 được tạo từ RDD0 bằng phép map, thì lineage của RDD1 lưu trữ thông tin rằng nó được sinh ra từ phép map trên RDD0. Khi RDD1 bị lỗi, Spark có thể tái tạo nó bằng cách thực hiện lại phép map trên RDD0.

- **DataFrame**

  - Đặc điểm:
    - **Abstraction chính trong Spark 2.0:** DataFrame là cấu trúc dữ liệu cấp cao, đại diện cho một tập dữ liệu có cấu trúc.
    - **Immutable (Không thay đổi):** Một khi được tạo ra, DataFrame không thể bị thay đổi. Nếu cần thay đổi dữ liệu, phải tạo một DataFrame mới.
    - **Track Lineage Information:** Spark lưu trữ thông tin lineage (quá trình tạo ra dữ liệu) để có thể tính toán lại dữ liệu bị mất trong trường hợp gặp sự cố.
    - **Hoạt động song song:** DataFrame hỗ trợ xử lý dữ liệu trên tập hợp lớn các phần tử một cách song song, tận dụng tối đa tài nguyên của cluster.
  - Các bước xây dựng Dataframe
    - Parallelizing từ các danh sách Python: Có thể sử dụng danh sách hoặc các cấu trúc dữ liệu có sẵn trong Python để tạo DataFrame.
    - Chuyển đổi từ một DataFrame Spark hoặc pandas có sẵn
    - Tạo từ các tệp tin lưu trữ: DataFrame có thể được tạo từ các tệp lưu trữ trên HDFS hoặc các hệ thống lưu trữ khác (như Amazon S3, Google Cloud Storage).

- **SparkContext và Master Parameter trong Spark**

  - **SparkContext**
    - SparkContext là đối tượng trung tâm trong mọi chương trình Spark. Nó đại diện cho kết nối với cụm Spark và cung cấp các phương tiện để tương tác với nó.
    - Chương trình Spark đầu tiên cần tạo SparkContext để xác định cách thức và vị trí truy cập vào cụm Spark.
    - Vai trò của SparkContext

- **Quản lý kết nối với cụm:** SparkContext giao tiếp với cụm Spark và cung cấp thông tin cần thiết như nguồn tài nguyên, cấu hình, và môi trường thực thi.
- **Tạo RDD, DataFrame và Dataset:** Thông qua SparkContext có thể tạo và quản lý các cấu trúc dữ liệu phân tán (RDD, DataFrame, Dataset) trong Spark.
- **Quản lý ứng dụng:** SparkContext chịu trách nhiệm gửi các tác vụ (tasks) đến các worker node trong cụm để thực thi.

- **Master Parameter**
  - **Master parameter** xác định loại và kích thước của cụm Spark sẽ được sử dụng để chạy chương trình. Các giá trị của Master parameter quy định chế độ thực thi (local hoặc cluster) và cách kết nối đến cụm.

Master Parameter	Mô tả
`local`	Chạy Spark trên máy cục bộ với **1 luồng worker** (không có song song).
`local[K]`	Chạy Spark trên máy cục bộ với **K luồng worker** (thường K = số lõi CPU để tận dụng tối đa tài nguyên).
`spark://HOST:PORT`	Kết nối tới một cụm Spark độc lập, nơi `HOST` và `PORT` là địa chỉ của cụm.
`mesos://HOST:PORT`	Kết nối tới cụm Mesos.
`yarn`	Kết nối tới cụm YARN.

Ví dụ:

```python
from pyspark import SparkContext

# Chạy Spark cục bộ với 1 worker thread
sc = SparkContext(master="local", appName="LocalExample")

# Chạy Spark cục bộ với số luồng bằng số lõi CPU
sc = SparkContext(master="local[4]", appName="LocalWithThreads")

# Kết nối Spark tới cụm độc lập
sc = SparkContext(master="spark://192.168.1.100:7077", appName="StandaloneClusterExample")

# Kết nối tới cụm YARN
sc = SparkContext(master="yarn", appName="YARNClusterExample")
```

VII. Stream processing

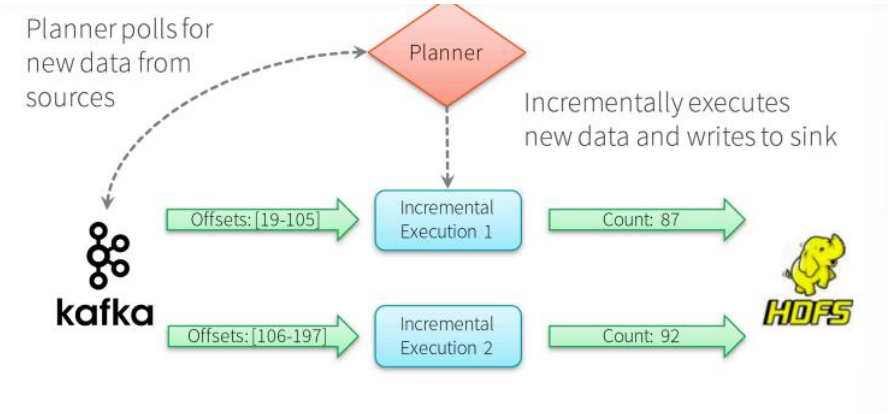
- Hạn chế của Dstreams (một mô hình xử lý dòng cũ)
  - Khó tích hợp với thời gian sự kiện: do sử dụng thời gian batch, khó khăn trong xử lý late data
  - API phức tạp và khó quản lý: việc dịch đổi giữa RDD và DStream gây tốn thời gian và dễ lỗi.
  - Khó đảm bảo độ tin cậy đầu-cuối (end-to-end guarantees)

- Mô hình mới cho stream processing
  - **Dữ liệu đầu vào**
    - Dữ liệu được xem như **một bảng chỉ thêm dữ liệu** (append-only table).
    - Đây là cách biểu diễn hợp lý trong xử lý dòng vì dữ liệu luôn được thêm mới theo thời gian.
  - **Kích hoạt (Trigger)**
    - **Trigger** xác định tần suất kiểm tra nguồn dữ liệu để tìm dữ liệu mới (ví dụ: mỗi 5 giây hoặc 1 phút).
    - Đây là cách kiểm soát chu kỳ xử lý, giúp cân bằng giữa hiệu suất và độ trễ.
  - **Truy vấn**
    - **Các phép toán quen thuộc**: map, filter, reduce.
    - **Các phép toán mới**:
      - Cửa sổ thời gian (windowing): Gom nhóm dữ liệu theo khoảng thời gian.
      - Phiên làm việc (session ops): Gom nhóm dựa trên hoạt động của người dùng.
  - **Kết quả**
    - Kết quả là một **bảng kết quả cuối cùng**, được Spark cập nhật ở mỗi chu kỳ kích hoạt.
  - **Đầu ra**
    - Có ba chế độ đầu ra tùy theo yêu cầu:
      - **Complete Output**: Ghi toàn bộ bảng kết quả ở mỗi chu kỳ.
      - **Delta Output**: Ghi chỉ những hàng bị thay đổi.
      - **Append Output**: Ghi chỉ các hàng mới.
  - **Structured Streaming**
    - API luồng cấp cao dựa trên Datasets/DataFrames
      - Thời gian sự kiện, cửa sổ, phiên làm việc, nguồn và "sink".
      - Đảm bảo chính xác từng bước từ đầu đến cuối.
    - Hợp nhất các truy vấn streaming, tương tác và batch
      - Tổng hợp dữ liệu trong một luồng, sau đó phân phát bằng JDBC
      - Thêm, xóa, thay đổi truy vấn khi chạy
      - Xây dựng và ứng dụng mô hình ML

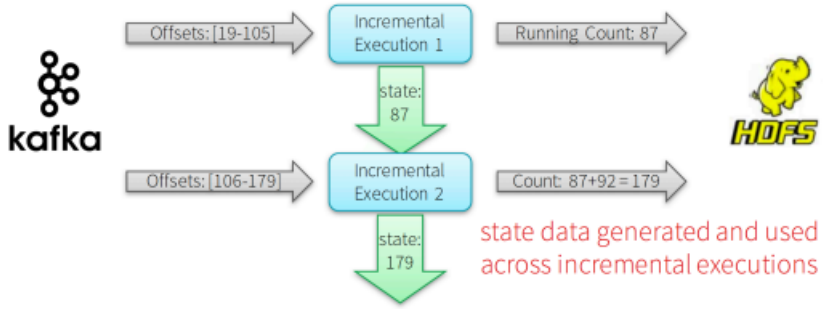
- Internal execution (thực thi nội bộ)
  - **Batch Execution on Spark SQL (thực thi theo lô** trên Spark SQL)
    - DataFrame/Dataset: Đây là lớp đầu vào mà người dùng tương tác. DataFrame /Dataset chứa dữ liệu có cấu trúc (structured data) được tổ chức dưới dạng bảng (columns/rows).
    - Logical Plan: Khi người dùng chạy truy vấn, Spark tạo ra kế hoạch logic (Logical Plan). Đây là bản thiết kế trừu tượng, không tối ưu hóa, biểu diễn tất cả các phép toán mà hệ thống cần thực hiện. Ví dụ: lọc, gom nhóm, tính tổng.

- Planner: chuyển kế hoạch logic thành kế hoạch thực thi (Execution Plan). Trong quá trình này, Spark áp dụng các chiến lược tối ưu hóa dựa trên chi phí (Physical Plan) để tìm cách thực thi hiệu quả nhất.
- Execution Plan: Đây là bản kế hoạch cụ thể mà Spark sử dụng để thực thi công việc trên các node trong cụm. Các công việc Spark (Spark jobs) siêu tối ưu hóa được thực thi dựa trên kế hoạch này. → **Project Tungsten**
  - Tối ưu hóa mã (Code Optimizations): Bytecode generation, JVM Intrinsics, Vectorization, Operations on Serialized Data (thao tác với DL đã tuần tự)
  - Tối ưu hóa bộ nhớ (Memory Optimizations): Compact and Fast Encoding (mã hóa nhỏ gọn và nhanh), Off-Heap Memory (bộ nhớ ngoài heap của JVM)

- **Continuous Incremental Execution (thực thi tăng dần liên tục** – DL được xử lý ngay khi nó đến thay vì xử lý toàn bộ như batch)
  - Khác với Batch Execution, Planner sẽ chuyển đổi kế hoạch logic thành **một chuỗi các kế hoạch thực thi tăng dần (Incremental Execution Plans)**. Mỗi kế hoạch thực thi xử lý một "chunk" (khối) dữ liệu mới đến từ luồng dữ liệu streaming.
  - Ví dụ xử lý DL luồng từ Kafka:
    - Planner liên tục kiểm tra (poll) nguồn dữ liệu như Kafka để lấy các phần dữ liệu mới (offsets).
    - Planner tạo ra các kế hoạch thực thi tăng dần (Incremental Execution). Ví dụ: Incremental Execution 1 xử lý offsets từ 19-105, Incremental Execution 2 xử lý offsets từ 106-197.
  - Kết quả xử lý được ghi vào HDFS hoặc một nơi lưu trữ khác. Ví dụ: Execution 1 tạo ra kết quả Count: 87, Execution 2 tạo ra Count: 92.



- **Continuous Aggregations** (Tổng hợp liên tục)
  - Hệ thống duy trì các giá trị tổng hợp (aggregate) **dạng trạng thái trong bộ nhớ (in-memory state)**. Để đảm bảo khả năng khôi phục khi gặp lỗi, trạng thái trong bộ nhớ được sao lưu bằng **WAL (Write-Ahead Log)** trên hệ thống tệp.
  - Trong sơ đồ, Execution 1 tạo ra trạng thái ban đầu (state = 87), được ghi vào HDFS. Execution 2 sử dụng trạng thái trước đó (state = 87) và cập nhật thành state = 179.

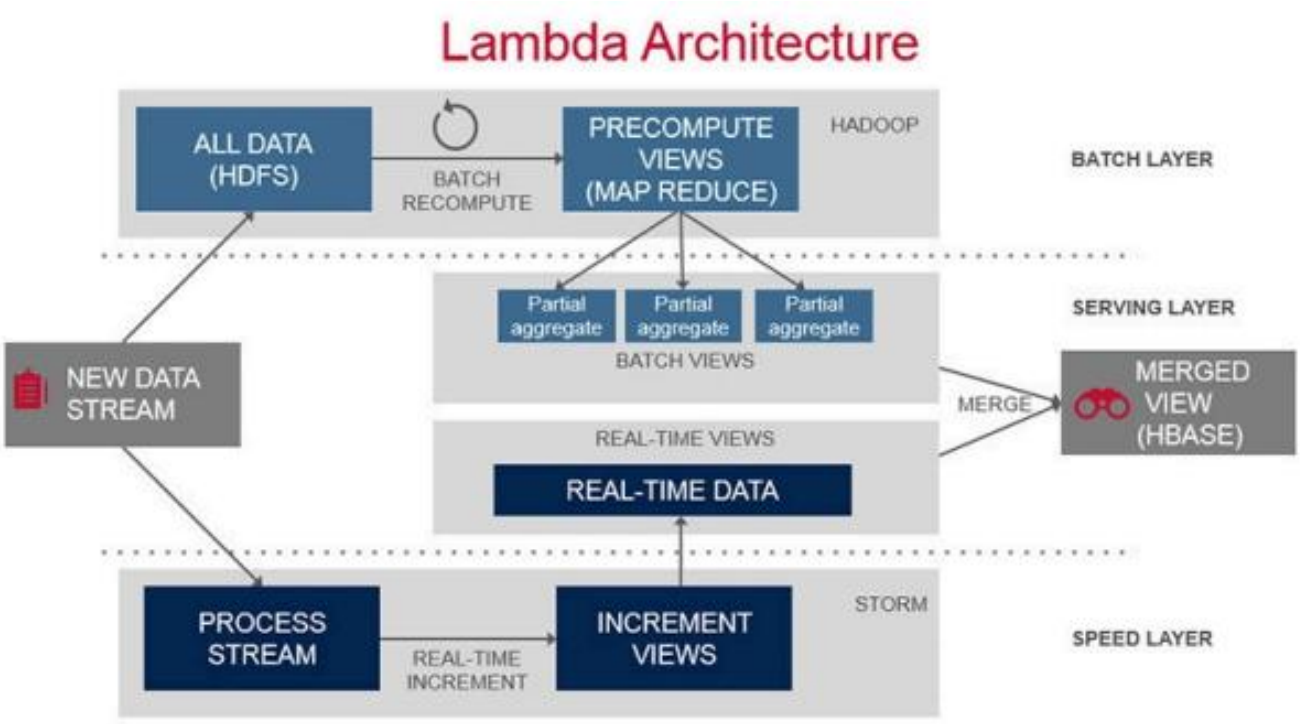


- **Khả năng chịu lỗi (Fault tolerance)**
  - Tất cả DL và siêu dữ liệu trong hệ thống cần có khả năng khôi phục hoặc phát lại.
  - Fault-tolerant Planner
    - Theo dõi các offset bằng cách ghi phạm vi offset của mỗi lần thực thi vào một log (WAL - Write Ahead Log) trong HDFS.
    - Đọc log để khôi phục sau lỗi và thực hiện lại chính xác phạm vi của offset
  - Fault-tolerant Sources
    - Các nguồn của Structured Streaming được thiết kế để có thể phát lại (replayable) như Kafka, Kinesis, hoặc file. Chúng tạo ra cùng một dữ liệu chính xác dựa trên offset được bộ lập kế hoạch khôi phục.
  - Fault-tolerant State
    - Dữ liệu trạng thái trung gian được duy trì trong các bản đồ key-value có phiên bản tại các Spark workers, được hỗ trợ bởi HDFS.
    - Planner đảm bảo sử dụng đúng phiên bản trạng thái khi thực thi lại sau lỗi.
  - Fault-tolerant Sink
    - Đầu ra (sink) được thiết kế để bất biến (idempotent) và xử lý các lần thực thi lại để tránh ghi đè hai lần lên kết quả.

offset tracking in WAL + state management + fault-tolerant sources and sinks = end-to-end exactly-once guarantees

VIII. Big data architecture

- **Kiến trúc Lambda:** là một kiến trúc xử lý dữ liệu phổ biến được thiết kế để xử lý khối lượng lớn dữ liệu theo cả cách xử lý lô (batch processing) và thời gian thực (real-time processing).



- **Batch Layer (Lớp xử lý lô):**
  - **Mô tả:**
    - Dữ liệu gốc (raw data) được lưu trữ lâu dài trong HDFS (Hadoop Distributed File System) hoặc các hệ thống lưu trữ lớn tương tự.
    - Sử dụng MapReduce hoặc các framework tương tự để tính toán các kết quả tổng hợp (batch views) từ dữ liệu lịch sử.
    - Batch Layer thực hiện các tính toán lại định kỳ (batch recompute) để đảm bảo tính toàn vẹn và chính xác của dữ liệu tổng hợp.
  - **Chức năng:**
    - Lưu trữ dữ liệu đầy đủ từ quá khứ.
    - Tạo ra các kết quả xử lý tổng hợp lớn (precomputed views) để phục vụ cho các truy vấn phức tạp.
- **Speed Layer (Lớp tốc độ):**
  - **Mô tả:**
    - Xử lý dòng dữ liệu mới trong thời gian thực bằng các công cụ như Storm, Kafka Streams hoặc Spark Streaming.
    - Tạo các kết quả tạm thời (increment views hoặc real-time views) từ dữ liệu thời gian thực.
  - **Chức năng:**
    - Đảm bảo độ trễ thấp (low-latency) trong việc xử lý dữ liệu.
    - Cung cấp các kết quả gần đúng ngay lập tức trước khi batch layer hoàn tất tính toán toàn bộ dữ liệu.

- **Serving Layer (Lớp phục vụ):**
  - **Mô tả:**
    - Tổng hợp và kết hợp (merge) các kết quả từ Batch Layer và Speed Layer để tạo ra một kết quả thống nhất (merged view) mà người dùng cuối hoặc ứng dụng có thể truy vấn.
    - HBase hoặc các cơ sở dữ liệu khác thường được sử dụng làm lớp lưu trữ cho merged view.
  - **Chức năng:**
    - Cung cấp dữ liệu nhanh chóng và chính xác dựa trên các kết quả từ cả batch và real-time.

Quá trình hoạt động tổng quát:

- **Dữ liệu mới (New Data Stream):** Dữ liệu từ các luồng (stream) được gửi đến cả Batch Layer và Speed Layer.
- **Batch Layer:** Lưu dữ liệu đầy đủ vào HDFS và thực hiện tính toán tổng hợp định kỳ.
- **Speed Layer:** Xử lý dòng dữ liệu thời gian thực, cung cấp kết quả gần như tức thì.
- **Serving Layer:** Hợp nhất kết quả từ cả hai lớp và cung cấp giao diện truy vấn cho người dùng.

- **Spark streaming**
  - Đặc điểm của Spark streaming
    - **Scalable**: Có khả năng mở rộng quy mô để xử lý lượng dữ liệu lớn.
    - **Fault-tolerance**: Đảm bảo độ tin cậy, có khả năng phục hồi trong trường hợp gặp lỗi.
    - **Stream processing system**: Là hệ thống xử lý dữ liệu dạng luồng.
  - Quá trình tính toán dữ liệu luồng (Streaming Computation): được thực hiện dưới dạng **một chuỗi các job nhỏ, xác định (deterministic batch jobs)**.
    - Chia luồng dữ liệu trực tiếp thành các batch nhỏ với độ dài X giây.
    - Xử lý từng batch dữ liệu dưới dạng RDD (Resilient Distributed Dataset) bằng các phép toán RDD.
    - Kết quả xử lý RDD được trả về theo batch.

  Dữ liệu đầu vào (Input Data Stream) → xử lý qua Spark Streaming (chia DL thành batch nhỏ) → xử lý bằng Spark Engine (các phép toán RDD) → xuất ra các batch kết quả đã được xử lý.

- Các công cụ xử lý luồng khác

Tiêu chí	Apache Storm	Spark Streaming	Apache Samza	Apache Flink

Đặc điểm	- Xử lý luồng thực sự (true streaming).	- Xử lý luồng dựa trên nền tảng batch của Spark.	- Xử lý luồng thực sự, tích hợp chặt với Apache Kafka.	- Xử lý luồng thực sự, linh hoạt về độ trễ và thông lượng.
	- Độ trễ rất thấp, thông lượng thấp hơn.	- Độ trễ cao hơn nhưng thông lượng rất cao.	- Hỗ trợ xử lý trạng thái (stateful) như yếu tố chính.	- Hỗ trợ rich semantics như xử lý cửa sổ dữ liệu (windowing).
	- Tối ưu cho các hệ thống yêu cầu phản ứng nhanh.	- Chạy theo cách xử lý lô nhỏ (micro-batch).	- Hỗ trợ mạnh xử lý trạng thái và tích hợp hệ thống phân tán.	- Điều chỉnh tốt giữa hiệu năng và độ phức tạp xử lý.
API	- Cấp thấp: Bolts, Spouts.	- DStreams (Discretized Streams).	- API cấp thấp, cung cấp khái niệm luồng khác biệt.	- API chức năng mạnh mẽ, hỗ trợ runtime chuyên sâu.
	- Giao diện Trident hỗ trợ xử lý phức tạp hơn.	- Hạn chế do mô hình batch runtime.		
Trường hợp sử dụng	- Ứng dụng cần xử lý real-time, độ trễ cực thấp.	- Ứng dụng yêu cầu xử lý tải cao với thông lượng lớn.	- Xử lý trạng thái phức tạp với Kafka làm backend.	- Ứng dụng cần xử lý nâng cao như phân tích dữ liệu thời gian thực.
	- Hệ thống phát hiện gian lận, giám sát dữ liệu tức thì.	- Phân tích dữ liệu hàng loạt kết hợp streaming.	- Quản lý luồng trạng thái phức tạp trong doanh nghiệp.	- Phân tích dữ liệu thời gian thực yêu cầu linh hoạt.

IX. Big data analytics with SparkML

- Pipeline
  - **Pipeline** trong SparkML là một chuỗi các bước xử lý dữ liệu, từ tiền xử lý, đặc trưng hóa (feature engineering), đến huấn luyện và dự đoán mô hình.
  - **Một số hàm liên quan:**
    - Pipeline(): Tạo một đối tượng Pipeline.
    - fit(): Dùng để huấn luyện Pipeline với dữ liệu huấn luyện.
    - transform(): Áp dụng Pipeline để biến đổi dữ liệu hoặc tạo dự đoán.
- Transformer
  - **Transformer** là một thành phần trong Pipeline dùng để **chuyển đổi dữ liệu** từ dạng này sang dạng khác.
  - Transformer không thay đổi sau khi được cấu hình.
  - Ví dụ các Transformer phổ biến:
    - **StringIndexer**: Mã hóa dữ liệu danh mục thành số.
    - **VectorAssembler**: Kết hợp nhiều cột đặc trưng thành một vector.
    - **StandardScaler**: Chuẩn hóa dữ liệu (scaling).

- Estimator
  - **Estimator** đại diện cho một **thuật toán học máy** hoặc **một bước trong Pipeline** yêu cầu dữ liệu huấn luyện để tạo ra một mô hình.
  - Ví dụ các Estimator phổ biến:
    - **LogisticRegression**: Thuật toán hồi quy logistic.
    - **DecisionTreeClassifier**: Bộ phân loại cây quyết định.
    - **KMeans**: Phân cụm K-Means.

- Pipeline và PipelineModel

Tiêu chí	Pipeline	PipelineModel
Vai trò	Định nghĩa quy trình xử lý dữ liệu và huấn luyện.	Áp dụng quy trình đã huấn luyện lên dữ liệu mới.
Loại đối tượng	Estimator	Transformer
Hàm chính	fit()	transform()
Đầu vào	Dữ liệu huấn luyện	Dữ liệu mới cần dự đoán
Đầu ra	PipelineModel	Kết quả dự đoán

X. Giải thích chi tiết một số câu hỏi trắc nghiệm

Câu 1: Hadoop giải quyết bài toán khả mở bằng cách nào? Chọn đáp án SAI

A. Thiết kế phân tán ngay từ đầu, mặc định triển khai trên cụm máy chủ
B. Các node tham gia vào cụm Hadoop được gán vai trò hoặc là node tính toán hoặc là node lưu trữ dữ liệu
C. Các node tham gia vào cụm đóng cả 2 vai trò tính toán và lưu trữ
D. Các node thêm vào cụm có thể có cấu hình, độ tin cậy cao

Giải thích: Hadoop giải quyết bài toán khả mở (scalability) bằng các cơ chế sau:

- **Thiết kế phân tán từ đầu:**
  - Hadoop được thiết kế để hoạt động trên một cụm máy chủ (cluster) gồm nhiều node, mỗi node có thể là một máy chủ độc lập.
  - Việc **phân tán dữ liệu** và **phân tán công việc** được tích hợp sẵn, giúp hệ thống mở rộng dễ dàng bằng cách thêm node mới vào cụm mà không cần thay đổi cấu trúc hoặc gây gián đoạn.
- **Kết hợp vai trò lưu trữ và tính toán:**
  - Hadoop tận dụng nguyên tắc **lưu trữ và tính toán cục bộ**. Các node trong cụm thường đóng cả hai vai trò. Bằng cách này, Hadoop giảm thiểu chi phí truyền dữ liệu qua mạng (data locality), tăng hiệu suất khi cụm mở rộng.
- **Thêm node linh hoạt:** Khi cần tăng khả năng xử lý hoặc lưu trữ, chỉ cần thêm node mới vào cụm Hadoop:
  - Node mới được tự động gán vai trò và hòa nhập vào cụm mà không cần can thiệp thủ công.
  - Không có yêu cầu các node phải đồng nhất về cấu hình, miễn là đáp ứng tối thiểu các tiêu chuẩn phần cứng.
- **Quản lý bởi HDFS và YARN:**
  - **HDFS (Hadoop Distributed File System)**: Phân tán dữ liệu và nhân bản (replication) các mảnh nhỏ của tệp tin trên nhiều node. Điều này giúp khả năng lưu trữ mở rộng theo số lượng node.
  - **YARN (Yet Another Resource Negotiator)**: Quản lý tài nguyên trong cụm, phân bổ tài nguyên hiệu quả khi thêm node mới.
- **Làm việc trên phần cứng giá rẻ:**
  - Hadoop được tối ưu hóa để làm việc với **commodity hardware** (phần cứng phổ thông), không cần các máy chủ chuyên dụng đắt tiền. Điều này giúp mở rộng cụm với chi phí thấp.

Tóm lại, Hadoop giải quyết bài toán khả mở bằng cách **thiết kế hệ thống phân tán**, **kết hợp lưu trữ và tính toán**, và hỗ trợ việc thêm node linh hoạt mà không yêu cầu đồng nhất về cấu hình.

Câu 2: Hadoop giải quyết bài toán chịu lỗi thông qua kỹ thuật gì? Chọn đáp án **SAI**

A. Kỹ thuật dư thừa
B. Các tệp tin được phân mảnh, các mảnh được nhân bản ra các node khác trên cụm
C. Các tệp tin được phân mảnh, các mảnh được lưu trữ tin cậy trên ổ cứng theo cơ chế RAID
D. Các công việc cần tính toán được phân mảnh thành các tác vụ độc lập

Giải thích: Hadoop sử dụng **kỹ thuật dư thừa** để chịu lỗi, không dựa trên RAID. Cụ thể:

- (a) **Đúng**: Dư thừa dữ liệu bằng cách nhân bản các mảnh dữ liệu (replication) trên các node khác nhau.
- (b) **Đúng**: Tệp tin được phân thành nhiều mảnh nhỏ và lưu trữ trên nhiều node khác nhau với cơ chế nhân bản.
- (d) **Đúng**: Công việc tính toán được chia thành các tác vụ nhỏ, độc lập để giảm thiểu sự ảnh hưởng khi một node gặp lỗi.

Câu 3: Các đặc trưng của HDFS. Chọn đáp án SAI

A. Tối ưu cho các tệp tin có kích thước lớn
B. Hỗ trợ thao tác đọc ghi tương tranh tại chunk (phân mảnh) trên tệp tin
C. Hỗ trợ nén dữ liệu để tiết kiệm chi phí
D. Hỗ trợ cơ chế phân quyền và kiểm soát người dùng của UNIX

Giải thích: HDFS được tối ưu hóa cho các **quy trình đọc tuần tự**, không phải đọc ghi đồng thời (concurrent read/write) nên đáp án B sai:

- (a) **Đúng**: HDFS được thiết kế để xử lý các tệp tin lớn (hàng GB đến TB).
- (c) **Đúng**: HDFS hỗ trợ nén để tiết kiệm chi phí lưu trữ.
- (d) **Đúng**: HDFS hỗ trợ cơ chế phân quyền và kiểm soát người dùng dựa trên hệ thống UNIX.

Câu 4: Phát biểu nào đúng về Presto

A. Các stage được thực thi theo cơ chế pipeline, không có thời gian chờ giữa các stage như Map Reduce
B. Presto cho phép xử lý kết tập dữ liệu mà kích thước lớn hơn kích thước bộ nhớ trong
C. Presto có cơ chế chịu lỗi khi thực thi truy vấn

Giải thích:

- A **đúng** vì Presto sử dụng cơ chế thực thi **pipeline**:
  - Dữ liệu được xử lý và truyền giữa các stage (các giai đoạn của truy vấn) một cách liên tục.
  - Không có thời gian chờ giữa các stage như trong MapReduce (MapReduce cần ghi dữ liệu trung gian xuống ổ cứng trước khi chuyển sang giai đoạn tiếp theo).
  - Điều này giúp Presto có độ trễ thấp và hiệu suất cao hơn.
- B **sai** vì Presto được thiết kế để xử lý dữ liệu **trong bộ nhớ (in-memory)**, tối ưu hóa cho các truy vấn nhanh.
  - Nếu dữ liệu vượt quá kích thước bộ nhớ, Presto không xử lý hiệu quả vì không ghi dữ liệu trung gian xuống ổ đĩa.
  - Điều này khác với Spark, có khả năng lưu trữ dữ liệu trung gian trên ổ đĩa nếu bộ nhớ không đủ.
- C **sai** vì Presto không có cơ chế chịu lỗi (fault tolerance) mạnh mẽ như Hadoop hoặc Spark:
  - Nếu một query (truy vấn) gặp lỗi, Presto không có khả năng tự động tiếp tục từ điểm dừng mà phải thực thi lại toàn bộ query.

- Điều này là do Presto không lưu dữ liệu trung gian xuống ổ đĩa và không có cơ chế checkpoint.

➔ Cơ chế **pipeline execution** là đặc điểm nổi bật của Presto, giúp nó tối ưu hóa hiệu suất và giảm độ trễ so với các công nghệ truyền thống như MapReduce.

Câu 5: Phát biểu nào sau đây sai về Kafka

A. nhiều consumer có thể cùng đọc 1 topic
B. 1 message có thể được đọc bởi nhiều consumer khác nhau
C. số lượng consumer phải ít hơn hoặc bằng số lượng partitions
D. 1 message chỉ có thể được đọc bởi 1 consumer trong 1 consumer group

Giải thích:

- A **đúng** vì trong Kafka, nhiều consumer có thể đồng thời đọc dữ liệu từ cùng một topic. Các consumer có thể thuộc các consumer group khác nhau hoặc cùng một group.
- B **đúng** vì một message có thể được đọc bởi nhiều consumer nếu các consumer này thuộc các **consumer group khác nhau**. Trong cùng một group, mỗi message chỉ được phân phối đến **một consumer** (theo nguyên tắc của Kafka).
- C **sai** vì không có yêu cầu rằng số lượng consumer phải ít hơn hoặc bằng số lượng partitions. Tuy nhiên:
  - **Nếu số lượng consumer vượt quá số lượng partitions**, một số consumer sẽ không nhận được dữ liệu vì mỗi partition chỉ có thể được tiêu thụ bởi một consumer trong cùng một group tại một thời điểm.
  - **Nếu số lượng consumer ít hơn số lượng partitions**, một số partitions sẽ được gán cho cùng một consumer.
- D **đúng** vì Trong một **consumer group**, Kafka đảm bảo rằng mỗi partition của topic chỉ được gán cho một consumer duy nhất. Điều này giúp phân phối công việc giữa các consumer trong group.

Câu 6: Hệ thống nào cho phép đọc ghi dữ liệu tại vị trí ngẫu nhiên, thời gian thực tới hàng terabyte dữ liệu

A. Hbase
B. Flume
C. Pig
D. HDFS

Giải thích:

HBase:

- Là một cơ sở dữ liệu NoSQL hoạt động trên HDFS, hỗ trợ:
  - **Đọc/ghi dữ liệu ngẫu nhiên** tại các vị trí bất kỳ.
  - **Thời gian thực** (real-time) trên dữ liệu lớn, thường lên đến hàng terabyte hoặc hơn.
- **Đặc điểm nổi bật**:
  - HBase sử dụng mô hình cột (column-oriented) để lưu trữ dữ liệu.
  - Thích hợp cho các ứng dụng yêu cầu truy cập dữ liệu nhanh và cập nhật thường xuyên (như bảng thời gian thực, dữ liệu cảm biến).

Flume:

- Là một công cụ chuyên dùng để **thu thập và truyền dữ liệu** từ nhiều nguồn khác nhau đến các hệ thống lưu trữ, như HDFS.
- Không hỗ trợ **đọc/ghi ngẫu nhiên** hay **thời gian thực** trên dữ liệu lớn.

Pig:

- Là một công cụ xử lý dữ liệu dựa trên ngôn ngữ **Pig Latin**, chủ yếu được dùng cho **xử lý dữ liệu hàng loạt (batch processing)** trên Hadoop.
- Không hỗ trợ truy cập dữ liệu thời gian thực hay đọc/ghi ngẫu nhiên.

HDFS (Hadoop Distributed File System):

- Là hệ thống lưu trữ phân tán của Hadoop, tối ưu cho việc lưu trữ và xử lý các tệp dữ liệu lớn.
- Không hỗ trợ **đọc/ghi ngẫu nhiên**. HDFS chỉ hỗ trợ:
    - Ghi tuần tự (write-once).
    - Đọc tuần tự (sequential reads).

Câu 7: Công cụ nào có thể sử dụng để hỗ trợ import, export dữ liệu vào ra hệ sinh thái Hadoop?

A. Oozie (lên lịch và quản lý)
B. Flume (thu thập dữ liệu)
C. Sqoop (trung gian tương tác với SQL)
D. Hive (truy vấn)

Giải thích: Sqoop dùng để trao đổi DL giữa các CSDL quan hệ (như MySQL, PostgreSQL, Oracle) và HDFS. → hỗ trợ import và export dữ liệu.

Câu 8: Cơ chế nhân bản dữ liệu trong HDFS?

A. Namenode quyết định vị trí các nhân bản của các chunk trên datanode.
B. Datanode là primary quyết định vị trí các nhân bản của các chunk tại các secondary datanode.
C. Client quyết định vị trí lưu trữ các nhân bản với từng chunk

Giải thích: NameNode là thành phần quản lý metadata của HDFS, bao gồm thông tin về tệp, phân chia thành các khối (chunk), và vị trí lưu trữ các bản sao (replica) trên DataNode. Khi một tệp được lưu trữ trong HDFS, NameNode quyết định vị trí các bản sao của từng chunk trên các DataNode để đảm bảo cân bằng tải và tính sẵn sàng.

Câu 9: Cơ chế mà NoSQL sử dụng để tăng khả năng chịu lỗi

A. Phân mảnh và phân tán dữ liệu ra nhiều máy chủ
B. Nhân bản (Replication)
C. Giao diện truy vấn đơn giản hơn so với CSDL quan hệ truyền thống

Giải thích: Replication là cơ chế chính giúp tăng khả năng chịu lỗi trong NoSQL. Dữ liệu được sao chép trên nhiều máy chủ (replica nodes). Nếu một node gặp sự cố, các replica khác có thể được sử dụng để đảm bảo dịch vụ không bị gián đoạn và không mất dữ liệu.

Một số mô hình nhân bản phổ biến:

- **Master-Slave Replication**: Một node chính (master) nhận ghi (write), các node phụ (slave) sao chép và phục vụ đọc (read). VD như MongoDB, Redis
- **Peer-to-Peer Replication**: Mọi node đều có quyền ghi và dữ liệu được đồng bộ hóa giữa các node. VD như Cassandra, Riak

Câu 10: Chọn phát biểu đúng về NoSQL

A. Không hỗ trợ các truy vấn SQL
B. Không thể được sử dụng kết hợp với các CSDL quan hệ
C. Rất phù hợp cho các tập dữ liệu phân tán quy mô lớn
D. Đáp ứng khả năng xử lý giao dịch với tính nhất quán chặt

Giải thích:

A. Không hỗ trợ các truy vấn SQL

- **Sai**: Một số cơ sở dữ liệu NoSQL (như Cassandra với CQL - Cassandra Query Language) hỗ trợ truy vấn kiểu SQL. Hơn nữa, các công cụ NoSQL như MongoDB có thể sử dụng cú pháp truy vấn tương tự SQL.
- Tuy nhiên, không phải tất cả các NoSQL đều hỗ trợ SQL.

B. Không thể được sử dụng kết hợp với các CSDL quan hệ

- **Sai**: NoSQL có thể được sử dụng kết hợp với CSDL quan hệ trong các hệ thống lai (hybrid). VD:
  - Một hệ thống dùng **MySQL** cho quản lý giao dịch và **MongoDB** để lưu trữ dữ liệu phi cấu trúc.
  - Công cụ tích hợp dữ liệu như Apache Sqoop có thể di chuyển dữ liệu giữa NoSQL và SQL.

C. Rất phù hợp cho các tập dữ liệu phân tán quy mô lớn

- **Đúng**: NoSQL được thiết kế để xử lý dữ liệu lớn (Big Data) và phân tán trên nhiều máy chủ (horizontal scaling). Kiến trúc của NoSQL giúp nó dễ dàng quản lý và phân phối dữ liệu trong môi trường phân tán.

D. Đáp ứng khả năng xử lý giao dịch với tính nhất quán chặt

- **Sai**: NoSQL thường tuân theo mô hình **CAP theorem** và tối ưu hóa cho **tính sẵn sàng (availability)** và **phân vùng chịu lỗi (partition tolerance)**, đôi khi đánh đổi tính nhất quán mạnh (strong consistency).
- Tuy nhiên, một số NoSQL như Couchbase hoặc Cosmos DB có thể cung cấp tùy chọn để đạt tính nhất quán chặt trong một số trường hợp.

Câu 11: NoSQL có đặc điểm nào dưới đây?

A. Mở rộng theo chiều ngang, tinh chỉnh được tính sẵn sàng của hệ thống
B. Không thể sử dụng SQL để truy vấn dữ liệu NoSQL
C. Mở rộng theo chiều dọc, thiết kế đơn giản, khó tinh chỉnh tính sẵn sàng của hệ thống
D. Mở rộng theo chiều dọc, thiết kế phức tạp, tinh chỉnh được tính sẵn sàng của hệ thống

Giải thích:

- **Mở rộng theo chiều ngang (Horizontal Scaling):** NoSQL được thiết kế để dễ dàng thêm nhiều máy chủ (nodes) nhằm mở rộng dung lượng lưu trữ và khả năng xử lý. Đây là ưu điểm lớn so với các CSDL quan hệ truyền thống, vốn thường mở rộng theo chiều dọc (Vertical Scaling) bằng cách nâng cấp phần cứng.
- **Tinh chỉnh tính sẵn sàng:** Nhiều hệ thống NoSQL (như Cassandra, DynamoDB, MongoDB) cho phép tùy chỉnh các mức độ nhất quán và tính sẵn sàng, thường dựa trên **CAP theorem**. Người dùng có thể cân bằng giữa **Consistency (nhất quán)**, **Availability (sẵn sàng)** và **Partition Tolerance (chịu lỗi phân vùng)** theo yêu cầu.

Câu 12: Phát biểu nào sai về Hbase

A. Hbase có lệ thuộc vào các dịch vụ cung cấp bởi Zookeeper
B. Hbase có hệ thuộc vào các dịch vụ cung cấp bởi HDFS
C. Hbase không hỗ trợ versioning
D. Hbase hỗ trợ truy vấn dạng SQL

Giải thích:

A: HBase có lệ thuộc vào các dịch vụ cung cấp bởi Zookeeper

- Đúng: HBase sử dụng Zookeeper để quản lý metadata, điều phối các Region Server, và đảm bảo khả năng chịu lỗi. Zookeeper là một phần quan trọng trong kiến trúc của HBase.

B: HBase có lệ thuộc vào các dịch vụ cung cấp bởi HDFS

- Đúng: HBase lưu trữ dữ liệu trên HDFS. HDFS cung cấp khả năng lưu trữ phân tán và chịu lỗi, điều này giúp HBase hoạt động ổn định.

C: HBase không hỗ trợ versioning

- Sai: HBase hỗ trợ versioning. Mỗi giá trị trong cột của một bảng có thể lưu nhiều phiên bản (versions) dựa trên timestamp. Số lượng phiên bản mặc định là 3, và người dùng có thể thay đổi cấu hình này.

D: HBase hỗ trợ truy vấn dạng SQL

- Sai (một phần): HBase không hỗ trợ truy vấn SQL trực tiếp. Tuy nhiên, với sự tích hợp của Apache Phoenix, người dùng có thể thực hiện các truy vấn SQL trên dữ liệu HBase.

Câu 13: Phát biểu nào sai về Hfile trong Hbase? (??? A or B or C)

A. Hfile chứa một tập hợp các dòng bản ghi trong Hbase table
B. Nhiều Hfile có thể được gộp lại thành 1 Hfile lớn theo những khoảng thời gian nhất định
C. Một version của 1 dòng hay 1 bản ghi trong Hbase table có thể được phân rã trên nhiều Hfile khác nhau
D. Nhiều Hfile có thể được gộp lại thành 1 Hfile lớn khi cần thiết

Giải thích: Mỗi phiên bản (version) của một dòng (được xác định bởi row key, column qualifier và timestamp) được lưu trữ trọn vẹn trong một HFile.

Câu 14: Các đặc điểm của virtual node trên AmazonDB. Chọn phương án sai
A. Mỗi node vật lý có thể được ánh xạ thành nhiều node ảo, nằm liên tiếp nhau trong vòng tròn không gian khoá.
B. Số lượng các node ảo đối với mỗi node vật lý là khác nhau tùy vào từng node vật lý.
C. Số lượng các node ảo bắt buộc cần phải căn cứ vào khả năng lưu trữ của node vật lý.
D. Node ảo đóng vai trò quan trọng trong bài toán cân bằng tải và hiệu năng khi một node vật lý ra hoặc kết nối vào cụm.

Giải thích:

A. Mỗi node vật lý có thể được ánh xạ thành nhiều node ảo, nằm liên tiếp nhau trong vòng tròn không gian khoá.

- **Đúng:** Trong hệ thống phân tán của Amazon DynamoDB, mỗi node vật lý có thể được ánh xạ thành nhiều node ảo (virtual nodes). Các node ảo này nằm trong một không gian khoá (keyspace) và được phân phối đều để giúp cân bằng tải.

B. Số lượng các node ảo đối với mỗi node vật lý là khác nhau tùy vào từng node vật lý.

- **Đúng:** Số lượng node ảo không nhất thiết phải giống nhau giữa các node vật lý. Mỗi node vật lý có thể được ánh xạ thành một số lượng node ảo khác nhau, tùy thuộc vào việc cân bằng tải và khả năng của từng node vật lý.

C. Số lượng các node ảo bắt buộc cần phải căn cứ vào khả năng lưu trữ của node vật lý.

- **Sai:** Số lượng các node ảo không nhất thiết phải căn cứ vào khả năng lưu trữ của node vật lý. Các node ảo thường được xác định theo cách khác, chẳng hạn như số lượng dữ liệu và yêu cầu về hiệu suất, thay vì dựa hoàn toàn vào dung lượng lưu trữ của từng node vật lý.

D. Node ảo đóng vai trò quan trọng trong bài toán cân bằng tải và hiệu năng khi một node vật lý ra hoặc kết nối vào cụm.

- **Đúng:** Các node ảo đóng vai trò quan trọng trong việc cân bằng tải và hiệu suất, đặc biệt khi các node vật lý gia nhập hoặc rời khỏi hệ thống. Các node ảo giúp phân phối lại dữ liệu một cách linh hoạt và đồng đều, giúp hệ thống duy trì tính ổn định và hiệu suất cao.

Câu 15: Phát biểu nào sau đây sai về Kafka

A. Các topic gồm nhiều partition
B. Partition được nhân bản ra nhiều brokers.
C. Kafka bảo đảm thứ tự của các message với mỗi topics.
D. Message sau khi được tiêu thụ (consume) thì không bị xoá

Giải thích:

A. Các topic gồm nhiều partition

- **Đúng:** Trong Kafka, một **topic** có thể chia thành nhiều **partition**. Điều này giúp Kafka phân tán dữ liệu và cho phép các consumer xử lý song song, từ đó cải thiện hiệu suất và khả năng mở rộng.

B. Partition được nhân bản ra nhiều brokers

- **Đúng:** Các **partition** của topic trong Kafka có thể được **nhân bản (replicated)** ra nhiều **brokers**. Điều này giúp tăng tính khả dụng và chịu lỗi, vì nếu một broker gặp sự cố, các bản sao của partition vẫn có thể phục vụ yêu cầu.

C. Kafka bảo đảm thứ tự của các message với mỗi topic

- **Sai:** Kafka bảo đảm thứ tự của các message trong mỗi **partition** của topic. Tuy nhiên, Kafka không đảm bảo thứ tự của các message giữa các partition, mà chỉ trong một partition cụ thể.

D. Message sau khi được tiêu thụ (consume) thì không bị xoá

- **Đúng:** Kafka không xoá message ngay sau khi chúng được tiêu thụ. Các message trong Kafka có thể được lưu trữ trong một khoảng thời gian xác định hoặc cho đến khi đạt đến giới hạn dung lượng, bất kể liệu chúng đã được tiêu thụ hay chưa. Điều này cho phép các consumer đọc lại các message cũ nếu cần.

Câu 16: Phát biểu nào sau đây sai về Kafka? (A or B)

A. Mỗi partition có 1 leader và nhiều followers.
B. Tất cả các thao tác ghi, đọc được xử lý bởi leader, follower làm theo leader.
C. Nếu leader bị lỗi, 1 follower sẽ thay thế trở thành leader mới

Giải thích:

- A sai vì mỗi partition luôn có 1 leader nhưng có thể có 1 hoặc nhiều followers chứ không phải luôn có nhiều followers

Câu 17: Phát biểu nào sai về Kafka?

A. Các message trên Kafka được lưu lại theo thời gian (time-based)
B. Các message trên Kafka được lưu lại theo kích thước partition (size-based)
C. Các message trên Kafka được lưu lại trước khi thực hiện compaction
D. Message sau khi được tiêu thụ bởi tất cả các consumer thì bị xoá.

Giải thích:

A. Các message trên Kafka được lưu lại theo thời gian (time-based)

- **Đúng:** Kafka cho phép cấu hình thời gian lưu trữ các message trong topic, có thể dựa trên thời gian, ví dụ như giữ các message trong một khoảng thời gian cụ thể (thời gian sống của message). Sau thời gian này, các message sẽ bị xoá đi.

B. Các message trên Kafka được lưu lại theo kích thước partition (size-based)

- **Đúng:** Kafka có thể cấu hình lưu trữ message theo kích thước partition. Khi partition đạt đến kích thước nhất định, Kafka sẽ thực hiện việc xoá hoặc chuyển các message cũ đi để giải phóng không gian.

C. Các message trên Kafka được lưu lại trước khi thực hiện compaction

- **Đúng:** Trước khi thực hiện **log compaction** (quá trình xóa các message cũ có phiên bản mới hơn), các message được lưu trữ trong Kafka. Quá trình compaction chỉ xoá các message cũ, giữ lại các message mới nhất cho mỗi khóa (key) trong log.

D. Message sau khi được tiêu thụ bởi tất cả các consumer thì bị xoá.

- **Sai:** Kafka không xoá message ngay sau khi chúng được tiêu thụ bởi tất cả các consumer. Các message sẽ được lưu trữ trong Kafka theo các chính sách lưu trữ thời gian (time-based) hoặc kích thước (size-based) cho đến khi chúng hết hạn hoặc hệ thống thực hiện compaction. Kafka không xoá message dựa trên việc tiêu thụ của consumer.

Câu 18: Phát biểu nào sai về Kafka?

A. Kafka producer quyết định message sẽ được gửi đến partition nào trong topic.
B. Thứ tự của message trong mỗi partition do key của message quyết định.
C. Kafka producer có thể gửi message đến nhiều broker khác nhau.

Giải thích:

A. Kafka producer quyết định message sẽ được gửi đến partition nào trong topic.

- **Đúng:** Kafka producer có thể xác định partition mà message sẽ được gửi đến trong một topic. Điều này có thể được thực hiện thông qua **partitioning key** (ví dụ, thông qua key của message) hoặc thông qua một hàm phân phối để xác định partition cụ thể.

B. Thứ tự của message trong mỗi partition do key của message quyết định.

- **Sai:** Thứ tự của message trong mỗi partition được xác định bởi thứ tự ghi của các producer, không phải do key của message quyết định. Key chỉ giúp xác định partition mà message sẽ được gửi đến, nhưng thứ tự các message trong partition vẫn phụ thuộc vào thứ tự chúng được ghi vào Kafka, không phải key.

C. Kafka producer có thể gửi message đến nhiều broker khác nhau.

- **Đúng:** Kafka producer có thể gửi message đến nhiều broker khác nhau. Các broker trong Kafka cụm (cluster) chịu trách nhiệm lưu trữ các partition của topic, và producer sẽ gửi các message đến các broker chứa partition tương ứng.

Câu 19: Đâu là cơ chế chịu lỗi của Apache Spark?

A. Chịu lỗi qua cơ chế huyết thống
B. Chịu lỗi qua cơ chế nhân bản
C. Chịu lỗi qua cơ chế lưu lại lịch sử nhiều phiên bản

Giải thích: Spark chủ yếu dựa vào cơ chế huyết thống (**lineage**) và lưu trữ các thông tin về các phép toán đã thực hiện để chịu lỗi.

→ RDD lưu lại các phiên bản cũ của mình → Có lỗi thì tính lại từ phiên bản cũ

Câu 20: Đâu là ưu điểm của Spark so với MapReduce?

A. Hỗ trợ tốt cho xử lý chuỗi các biến đổi

B. Có thể khai phá dữ liệu trong thời gian tương tác

C. Khai thác bộ nhớ trong thay vì sử dụng hệ thống lưu trữ ngoài như HDFS

D. Có khả năng chịu lỗi

Câu 21: Các biến đổi (transformation) trên Spark có đặc điểm gì?

A. Thực hiện theo cơ chế lười biếng, khi nào một hành động (action) cần tới phép biến đổi trước đó phải thực hiện thì mới phải thực hiện
B. Mỗi phép biến đổi trên RDD được thực thi bởi một hay nhiều Spark worker
C. Các biến đổi (transformation) luôn tạo ra RDD mới có cùng số partition với RDD đầu vào

Giải thích:

A. Thực hiện theo cơ chế lười biếng, khi nào một hành động (action) cần tới phép biến đổi trước đó phải thực hiện thì mới phải thực hiện

- **Đúng:** Spark thực hiện các phép biến đổi (transformations) theo cơ chế **lười biếng (lazy evaluation)**. Điều này có nghĩa là các phép biến đổi sẽ không được thực thi ngay lập tức khi được gọi, mà chỉ khi một hành động (action) như *collect(), count(), hoặc save()* được gọi. Khi đó, Spark sẽ thực thi tất cả các phép biến đổi trước đó trong một lần.

B. Mỗi phép biến đổi trên RDD được thực thi bởi một hay nhiều Spark worker

- **Đúng:** Các phép biến đổi trên RDD (Resilient Distributed Dataset) được phân phối và thực thi trên nhiều **Spark worker**. Mỗi worker có thể xử lý một phần của dữ liệu, và các phép biến đổi sẽ được áp dụng cho từng phần dữ liệu tương ứng trên các worker này.

C. Các biến đổi (transformation) luôn tạo ra RDD mới có cùng số partition với RDD đầu vào

- **Sai:** Không phải tất cả các phép biến đổi trên RDD tạo ra một RDD mới có cùng số partition với RDD đầu vào. Ví dụ, một số phép biến đổi như *repartition() hoặc coalesce()* có thể thay đổi số lượng partition của RDD. Do đó, số partition có thể thay đổi tùy thuộc vào phép biến đổi cụ thể.

Câu 22: Kiến trúc xử lý dữ liệu Lambda có đặc điểm gì?

A. Kết hợp xử lý dữ liệu theo lô và theo luồng
B. Giúp giải quyết vấn đề độ trễ từ khi dữ liệu được thập tới kết quả phân tích của mô hình xử lý theo lô

C. Giúp giải quyết vấn đề nhược điểm của xử lý theo luồng là kết quả phân tích không khai thác được toàn bộ dữ liệu trong lịch sử.
D. Có kiến trúc gồm 2 tầng: tầng xử lý theo lô và tầng xử lý theo luồng
E. Bao gồm các tiến trình ETL (extract, transform, load) đưa dữ liệu vào hồ dữ liệu (data lake)

Câu 23: Chế độ nào sau đây không phải là chế độ hoạt động của Hadoop?

A. Pseudo distributed mode
B. Globally distributed mode
C. Stand alone mode
D. Fully-Distributed mode

Giải thích: Không phải Globally distributed mode mà phải là Fully-Distributed mode.

Câu 24: Cơ chế nào sau đây không phải là cơ chế hàng rào cho NameNode đã hoạt động trước đó?

A. Tắt cổng mạng của nó thơng qua lệnh quản lý từ xa.
B. Thu hồi quyền truy cập của nó vào thư mục lưu trữ được chia sẻ.
C. Định dạng ổ đĩa của nó.
D. STONITH

Giải thích: Cơ chế hàng rào (fencing) cho NameNode trong Hadoop là các phương pháp được sử dụng để ngăn chặn NameNode gặp sự cố hoặc tiếp tục hoạt động sai trong trường hợp có sự cố xảy ra. Các cơ chế hàng rào này đảm bảo rằng không có hai NameNode nào có thể cùng truy cập và điều khiển hệ thống đồng thời, điều này rất quan trọng trong môi trường Hadoop để tránh mất mát dữ liệu.

Câu 25: Giao tiếp giữa các quá trình (processes) giữa các nút khác nhau trong cluster chủ yếu sử dụng

A. REST API
B. RPC
C. RMI
D. IP Exchange

Giải thích: RPC (**Remote Procedure Call**) cho phép các node trong hệ thống phân tán (như các node trong Hadoop) gọi các hàm hoặc phương thức từ xa, điều này cho phép chúng giao tiếp và tương tác với nhau.

Câu 26: Job tracker chạy trên

A. Namenode
B. Datanode
C. Secondary namenode
D. Secondary datanode

Giải thích: **JobTracker** là một thành phần quan trọng chịu trách nhiệm quản lý và theo dõi tiến trình thực hiện của các job MapReduce. **JobTracker chạy trên Namenode**, nơi nó quản lý việc phân phối và theo dõi các job MapReduce tới các **TaskTrackers trên các Datanode** trong cluster.

Câu 27: Khi lưu trữ tệp Hadoop, câu nào sau đây là đúng? (Chọn hai câu trả lời)

1. Các tệp đã lưu trữ sẽ hiển thị với phần mở rộng .arc.
2. Nhiều tệp nhỏ sẽ trở thành ít tệp lớn hơn.
3. MapReduce xử lý tên tệp gốc ngay cả sau khi tệp được lưu trữ.
4. Các tệp đã lưu trữ phải được lưu trữ tại Liên hợp quốc cho HDFS và MapReduce để truy cập các tệp nhỏ, gốc.
5. Lưu trữ dành cho các tệp cần được lưu nhưng HDFS không còn truy cập được nữa.

A. 1 & 3 B. 2 & 3 C. 2 & 4 D. 3 & 5

Câu 28: Spark có thể chạy ở chế độ nào khi chạy trên nhiều máy

A. chạy trên YARN
B. chạy trên ZooKeeper
C. phương án a và b đều sai
D. phương án a và b đều đúng

Giải thích:

- Spark có thể chạy trên **YARN** (Yet Another Resource Negotiator), một trình quản lý tài nguyên được sử dụng phổ biến trong hệ sinh thái Hadoop. Khi chạy trên YARN, Spark có thể tận dụng hạ tầng phân tán của Hadoop để quản lý tài nguyên và xử lý dữ liệu.
- ZooKeeper không phải là một nền tảng để chạy Spark. ZooKeeper được sử dụng chủ yếu để quản lý cấu hình, đồng bộ hóa, và cung cấp các dịch vụ điều phối trong các hệ thống phân tán, không phải để quản lý tài nguyên hay chạy Spark.

Câu 29: Data Pipeline nào sau đây là đúng trên Spark

A. Spark → RabbitMQ → Elasticsearch => Hiển thị

B. Dữ liệu sensor => RabbitMQ =>Elasticsearch=>Spark=>Hiển thị

C. Dữ liệu sensor => Elasticserach => RabbitMQ => Spark => Hiển thị

D. Spark => Elasticsearch => Hiển thị

Câu 30: Mục đích của sử dụng RabbitMQ là gì?

A. Lưu trữ dữ liệu
B. Tránh dữ liệu bị mất mát
C. Hiển thị dữ liệu
D. Phân tích dữ liệu

Câu 31: Phát biểu nào sai về cơ chế scheduling của Presto?

A. Một task có thể được lập lịch chạy trên bất kỳ worker nào
B. Stage có thể được lập lịch all-at-once
C. Stage có thể được lập lịch theo giai đoạn
D. Split được gán cho task theo cơ chế lazy

Giải thích:

- **A. Một task có thể được lập lịch chạy trên bất kỳ worker nào**: Đúng, Presto có thể lập lịch task trên bất kỳ worker nào trong cluster.
- **B. Stage có thể được lập lịch all-at-once**: Đúng, Presto có thể lập lịch toàn bộ stage cùng một lúc (all-at-once) nếu cần.
- **C. Stage có thể được lập lịch theo giai đoạn**: Đúng, Presto có thể lập lịch stage theo từng giai đoạn (pipelined) để tối ưu hóa hiệu suất.
- **D. Split được gán cho task theo cơ chế lazy**: Sai, Presto không sử dụng cơ chế lazy để gán split cho task. Thay vào đó, split được gán ngay lập tức khi task được lập lịch để đảm bảo xử lý dữ liệu hiệu quả.

Câu 32: Ưu điểm của hệ thống tệp tin phân tán là gì?

A. Đơn giản hoá việc chia sẻ dữ liệu.
B. Tập trung hoá việc quản trị.
C. Cho phép người dùng có cái nhìn hợp nhất (như nhau) về toàn bộ dữ liệu trong hệ thống

Giải thích:

- **A. Đơn giản hoá việc chia sẻ dữ liệu**: Mặc dù hệ thống tệp tin phân tán có thể hỗ trợ chia sẻ dữ liệu, nhưng đây không phải là ưu điểm chính hoặc đặc trưng nhất của nó.
- **B. Tập trung hoá việc quản trị**: Sai. Hệ thống tệp tin phân tán thường phân tán việc quản trị và lưu trữ dữ liệu trên nhiều node, không tập trung hoá.
- **C. Cho phép người dùng có cái nhìn hợp nhất (như nhau) về toàn bộ dữ liệu trong hệ thống**: Đúng. Một trong những ưu điểm chính của hệ thống tệp tin phân tán là cung cấp một cái nhìn thống nhất và hợp nhất về dữ liệu, dù dữ liệu được lưu trữ trên nhiều node khác nhau. Người dùng không cần biết dữ liệu được lưu trữ ở đâu, mà chỉ cần truy cập thông qua một giao diện thống nhất.

XI. Một số câu hỏi về đoạn mã code

Câu 1:

```
rdd = sc.parallelize([“hello”, “world”, “good”, “hello”], 2)
rdd = rdd.map(lambda w : (w, 1))
```

Đưa ra kết quả của rdd.glom().collect()

Giải thích chi tiết code:

- Tạo một RDD từ danh sách *[“hello”, “world”, “good”, “hello”]* và chia nó thành 2 partion. Khi đó ta có 2 partition là [“hello”, “world”] và [“good”, “hello”]
- map sẽ áp dụng lên từng phần tử. Khi đó 2 partion trở thành [(“hello”, 1), (“world”, 1)] và [(“good”, 1), (“hello”, 1)]
- glom chuyển đổi thành 1 danh sách và collect() sẽ trả về dưới dạng danh sách các partition

$$[[(“hello”, 1), (“world”, 1)] , \ [(“good”, 1), (“hello”, 1)]]$$

Các phép biến đổi (Transformations)

Hàm (Tham số)	Ý nghĩa	Ví dụ minh họa
`map(func)`	Áp dụng một hàm lên từng phần tử trong RDD, tạo một RDD mới.	`rdd.map(lambda x: x * 2)` → `[2, 4, 6, 8]` từ `[1, 2, 3, 4]`
`flatMap(func)`	Áp dụng hàm trên từng phần tử và làm phẳng kết quả (flatten).	`rdd.flatMap(lambda x: x.split(" "))` → `["Hello", "World"]` từ `["Hello World"]`
`filter(func)`	Giữ lại các phần tử thỏa mãn điều kiện.	`rdd.filter(lambda x: x % 2 == 0)` → `[2, 4]` từ `[1, 2, 3, 4]`
`distinct()`	Loại bỏ các phần tử trùng lặp trong RDD.	`rdd.distinct()` → `[1, 2, 3]` từ `[1, 2, 2, 3]`
`union(otherRDD)`	Hợp hai RDD, bao gồm tất cả phần tử từ cả hai.	`rdd1.union(rdd2)` → `[1, 2, 3, 4]` từ `rdd1 = [1, 2]` và `rdd2 = [3, 4]`
`intersection(otherRDD)`	Giữ lại các phần tử chung của hai RDD.	`rdd1.intersection(rdd2)` → `[2]` từ `rdd1 = [1, 2]` và `rdd2 = [2, 3]`
`subtract(otherRDD)`	Loại bỏ các phần tử trong RDD khác.	`rdd1.subtract(rdd2)` → `[1]` từ `rdd1 = [1, 2]` và `rdd2 = [2, 3]`

`cartesian(otherRDD)`	Tạo tích Descartes giữa hai RDD.	`rdd1.cartesian(rdd2)` → `[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]` từ `rdd1 = [1, 2]` và `rdd2 = ['a', 'b']`
`groupByKey()`	Gom nhóm các giá trị theo khóa.	`rdd.groupByKey()` từ `[(1, 2), (1, 3)]` → `{1: [2, 3]}`
`reduceByKey(func)`	Kết hợp các giá trị có cùng khóa bằng hàm `func`.	`rdd.reduceByKey(lambda x, y: x + y)` → `[(1, 5)]` từ `[(1, 2), (1, 3)]`
`sortByKey()`	Sắp xếp các phần tử theo khóa.	`rdd.sortByKey()` → `[(1, 'a'), (2, 'b')]` từ `[(2, 'b'), (1, 'a')]`
`join(otherRDD)`	Thực hiện phép nối (join) giữa hai RDD dựa trên khóa.	`rdd1.join(rdd2)` → `[(1, (2, 'a'))]` từ `rdd1 = [(1, 2)]` và `rdd2 = [(1, 'a')]`
`coalesce(numPartitions)`	Giảm số lượng phân vùng của RDD.	`rdd.coalesce(1)` → Giảm phân vùng từ 2 xuống 1.
`repartition(numPartitions)`	Tăng hoặc giảm số lượng phân vùng, đảm bảo cân bằng hơn.	`rdd.repartition(4)` → Tăng số phân vùng lên 4.

Các hành động (Actions)

Hàm (Tham số)	Ý nghĩa	Ví dụ minh họa
`collect()`	Thu thập tất cả phần tử trong RDD và trả về dưới dạng danh sách.	`rdd.collect()` → `[1, 2, 3, 4]` từ RDD `[1, 2, 3, 4]`
`count()`	Đếm tổng số phần tử trong RDD.	`rdd.count()` → `4` từ RDD `[1, 2, 3, 4]`
`take(n)`	Lấy `n` phần tử đầu tiên trong RDD.	`rdd.take(2)` → `[1, 2]` từ RDD `[1, 2, 3, 4]`
`first()`	Trả về phần tử đầu tiên trong RDD.	`rdd.first()` → `1` từ RDD `[1, 2, 3, 4]`

`reduce(func)`	Kết hợp tất cả phần tử trong RDD bằng hàm `func`.	`rdd.reduce(lambda x, y: x + y)` → `10` từ RDD `[1, 2, 3, 4]`
`countByValue()`	Đếm số lần xuất hiện của từng phần tử trong RDD.	`rdd.countByValue()` → `{1: 1, 2: 1, 3: 1, 4: 1}` từ `[1, 2, 3, 4]`
`saveAsTextFile(path)`	Lưu RDD dưới dạng file văn bản tại đường dẫn `path`.	`rdd.saveAsTextFile("output")` → Lưu file tại thư mục `output`.
`takeOrdered(n, key=func)`	Lấy `n` phần tử theo thứ tự sắp xếp từ nhỏ đến lớn (hoặc dựa trên khóa `key`).	`rdd.takeOrdered(2)` → `[1, 2]` từ `[3, 1, 2, 4]`
`foreach(func)`	Áp dụng một hàm `func` cho từng phần tử trong RDD (không thu thập kết quả).	`rdd.foreach(lambda x: print(x))` → In từng phần tử.
`takeSample(withReplacement, num, seed)`	Lấy mẫu ngẫu nhiên từ RDD.	`rdd.takeSample(False, 2)` → `[3, 1]` (ngẫu nhiên từ `[1, 2, 3, 4]`).