**ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**

A red and white logo

Description automatically generated

**BÁO CÁO BÀI TẬP LỚN**

**Lưu trữ và xử lý dữ liệu lớn**

**ĐỀ TÀI:**

**Wikipedia Search Engine**

|  |
| --- |
|  |
| **Nhóm:**  **Mã lớp:** | **11**  **154050** |
| **Giảng viên hướng dẫn:**  **Sinh viên thực hiện:** | **TS. Trần Việt Trung**  **20215390 – Phạm Việt Hùng**  **20215301 – Hoàng Thế Anh**  **20210476 – Nguyễn Nam Khánh**  **20215315 – Đỗ Thị Thanh Bình**  **20215407 – Hà Trung Kiên** |

**MỤC LỤC**

[I. Đặt vấn đề 2](#_Toc189840289)

[1. Tổng quan bài toán 2](#_Toc1493569978)

[2. Phân tích bài toán trong xử lý dữ liệu lớn 2](#_Toc1999996759)

[3. Phạm vi và giới hạn báo cáo 3](#_Toc979839477)

[II. Kiến trúc và thiết kế 3](#_Toc608972856)

[1. Kiến trúc tổng thể 3](#_Toc1374167392)

[1.1 Kiến trúc Lambda 3](#_Toc1681242387)

[1.2 Kiến trúc tổng quan của hệ thống 4](#_Toc530568407)

[2. Chi tiết thành phần hệ thống 5](#_Toc1708118968)

[2.1 Hadoop HDFS 5](#_Toc1084625444)

[2.2 Apache Spark 6](#_Toc97474579)

[2.3 Apache airflow 6](#_Toc956741449)

[III. Chi tiết triển khai 7](#_Toc606250772)

[1. Crawling data 7](#_Toc845275586)

[1.1. Crawl theo tên bài viết 7](#_Toc2031726660)

[1.2. Crawl theo URL 7](#_Toc839178490)

[2. Hadoop HDFS 8](#_Toc707397768)

[2.1. HDFS trong dự án này được triển khai với cấu trúc 8](#_Toc514226405)

[2.2. Mô hình triển khai 8](#_Toc1568641653)

[3. Spark 11](#_Toc1262520651)

[3.1. Lưu dữ liệu vào Hbase 11](#_Toc52312863)

[3.2. Chuyển đổi dữ liệu 13](#_Toc167859709)

[3.3. Cập nhật chỉ mục đảo ngược 15](#_Toc966049113)

[3.4. Tìm kiếm từ khóa 15](#_Toc121316177)

[4. Airflow 17](#_Toc1442852602)

[5. ReactJs 18](#_Toc314766418)

[IV. Bài học kinh nghiệm 18](#_Toc774147418)

[1. Kinh nghiệm merge các bảng 18](#_Toc1624843888)

[2. Kinh nghiệm xây dựng search: 19](#_Toc325693071)

[2. Setup HDFS 21](#_Toc2003415740)

[3. Sử dụng Spark ghi dữ liệu vào Apache Hbase 22](#_Toc277240410)

[4. Lựa chọn giữa Selenium và CrawlAPI 23](#_Toc281255190)

[V. Kết luận 24](#_Toc346654047)

[1. Kết luận 24](#_Toc2066625196)

[2. Hướng phát triển 25](#_Toc2059591829)

# **I. Đặt vấn đề**

## **Tổng quan bài toán**

Wikipedia là một trong những kho tài liệu lớn nhất và phong phú nhất trên thế giới, với hàng triệu bài viết được viết và cập nhật liên tục bởi cộng đồng người dùng toàn cầu. Việc tìm kiếm thông tin trên Wikipedia trở nên cực kỳ quan trọng đối với người dùng muốn truy cập vào dữ liệu hữu ích và chính xác trong thời gian ngắn. Tuy nhiên, với khối lượng dữ liệu khổng lồ và liên tục thay đổi, việc xây dựng một hệ thống tìm kiếm hiệu quả và chính xác trên nền tảng này là một thách thức không nhỏ.

Mục tiêu của hệ thống tìm kiếm Wikipedia là giúp người dùng dễ dàng truy cập thông tin qua các truy vấn tìm kiếm, đảm bảo độ chính xác cao, tốc độ nhanh và khả năng phục vụ hàng triệu yêu cầu tìm kiếm mỗi ngày. Các bài toán chính mà hệ thống cần giải quyết bao gồm tối ưu hóa thuật toán tìm kiếm, xử lý ngữ nghĩa của các truy vấn, và cải thiện hiệu quả sử dụng dữ liệu.

## **Phân tích bài toán trong xử lý dữ liệu lớn**

Với Wikipedia, dữ liệu không chỉ bao gồm hàng triệu bài viết mà còn có cấu trúc phức tạp và không đồng nhất, vì vậy bài toán tìm kiếm và phân tích yêu cầu các phương pháp và công nghệ xử lý dữ liệu lớn mạnh mẽ. Các vấn đề chính bao gồm:

* **Lưu trữ và xử lý dữ liệu khối lượng lớn:** Wikipedia chứa hàng triệu bài viết với dung lượng dữ liệu khổng lồ, yêu cầu hệ thống tìm kiếm phải có khả năng mở rộng để xử lý và lưu trữ một lượng dữ liệu cực kỳ lớn và có thể thay đổi liên tục.
* **Tối ưu hóa thuật toán tìm kiếm:** Các truy vấn tìm kiếm trên Wikipedia có thể yêu cầu xử lý một lượng lớn dữ liệu để tìm ra kết quả phù hợp. Các thuật toán tìm kiếm cần được tối ưu hóa để cung cấp kết quả chính xác và nhanh chóng. Sử dụng các công cụ phân tích như Hadoop MapReduce và Spark có thể giúp tăng tốc quá trình xử lý và phân tích dữ liệu tìm kiếm, từ đó cải thiện tốc độ phản hồi và chất lượng kết quả.
* **Cập nhật và tối ưu hóa dữ liệu liên tục:** Wikipedia là một hệ thống động, với dữ liệu được cập nhật và sửa đổi liên tục. Điều này đòi hỏi hệ thống tìm kiếm phải có khả năng xử lý các bản cập nhật này một cách hiệu quả và không làm giảm chất lượng của kết quả tìm kiếm. Các công nghệ như Spark Streaming có thể hỗ trợ xử lý dữ liệu thay đổi trong thời gian thực, giúp cập nhật thông tin kịp thời và cải thiện kết quả tìm kiếm.

Với sự hỗ trợ của các hệ thống phân tích dữ liệu lớn, việc khai thác dữ liệu từ Wikipedia không chỉ dừng lại ở việc thống kê và tổ chức thông tin mà còn giúp tối ưu hóa quá trình tìm kiếm và cải thiện khả năng phục vụ người dùng. Các công nghệ như Hadoop và Spark có thể xử lý và phân tích một lượng lớn dữ liệu trong thời gian thực, giúp nhận diện các xu hướng tìm kiếm, tối ưu hóa thuật toán tìm kiếm, và cải thiện độ chính xác của kết quả. Điều này không chỉ giúp tăng tốc độ phản hồi mà còn cho phép hệ thống tìm kiếm của Wikipedia phục vụ hàng triệu yêu cầu mỗi ngày một cách hiệu quả và chính xác hơn.

## **Phạm vi và giới hạn báo cáo**

* **Phạm vi:** Báo cáo tập trung vào việc phân tích và tối ưu hóa hệ thống tìm kiếm Wikipedia trong bối cảnh xử lý dữ liệu lớn, đặc biệt là cải thiện tốc độ và hiệu quả của quá trình tìm kiếm. Nội dung nhấn mạnh vào các phương pháp và công nghệ xử lý dữ liệu lớn nhằm hỗ trợ số lượng lớn yêu cầu tìm kiếm mỗi ngày.
* **Giới hạn:** Báo cáo sử dụng các công cụ phân tích dữ liệu lớn như Hadoop, Hbase và Spark để phân tích dữ liệu từ Wikipedia, nhưng không bao gồm việc thay đổi hoặc phát triển trực tiếp hệ thống tìm kiếm của Wikipedia. Dữ liệu được sử dụng chủ yếu từ các nguồn mở và không bao gồm việc thu thập dữ liệu thực địa. Kết quả phân tích trong báo cáo mang tính tham khảo và đề xuất, không đảm bảo độ chính xác tuyệt đối do các hạn chế về dữ liệu và phương pháp nghiên cứu.

# **II. Kiến trúc và thiết kế**

## **Kiến trúc tổng thể**

### 1.1 Kiến trúc Lambda

A diagram of a software process

Description automatically generated with medium confidence

Lambda Architecture là một mô hình kiến trúc dữ liệu được thiết kế để xử lý khối lượng lớn dữ liệu với độ trễ thấp và khả năng mở rộng cao. Kiến trúc này được chia thành ba lớp chính:

- **Batch Layer**: Xử lý toàn bộ dữ liệu gốc và tạo ra các view dữ liệu đầy đủ

- **Speed Layer**: Xử lý dữ liệu theo thời gian thực để bù đắp độ trễ của Batch Layer

- **Serving Layer**: Phục vụ kết quả từ cả Batch Layer và Speed Layer

**Lý Do Lựa Chọn Lambda Architecture:**

- Đảm bảo tính chính xác và cập nhật của dữ liệu

- Khả năng mở rộng và chịu lỗi cao- Khả năng mở rộng và chịu lỗi cao

### 1.2 Kiến trúc tổng quan của hệ thống

A diagram of a company logo

Description automatically generated

Kiến trúc tổng thể: Hệ thống sử dụng kiến trúc Lambda và tập trung vào xử lý theo batch:

**Batch Layer:**

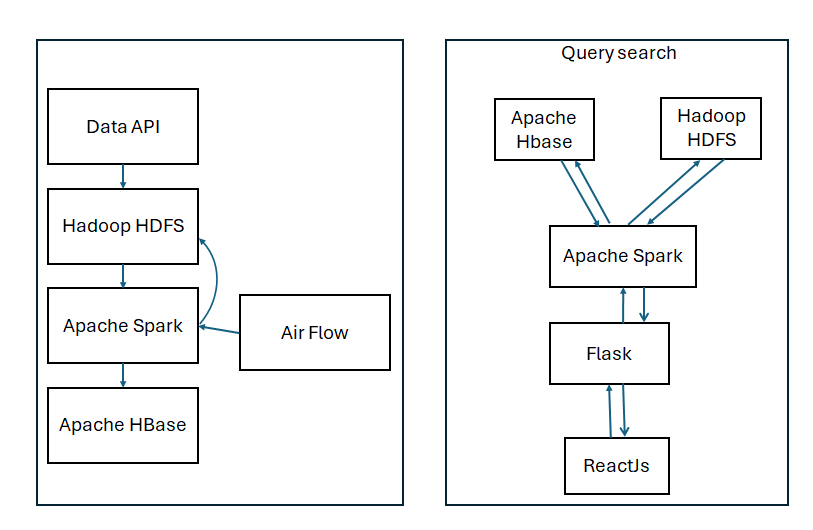
* HDFS lưu trữ toàn bộ dữ liệu gốc và đã được xử lý
* Hbase lưu toàn bộ dữ liệu chi tiết
* Spark thực hiện xử lý batch định kỳ, thực hiện truy vấn dữ liệu

**Serving Layer:**

* Apache Hbase: Lưu trữ thông tin chi tiết của dữ liệu
* Flask cung cấp API gửi dữ liệu truy vấn cho Client
* Dữ liệu được cập nhật theo chu kỳ batch

**Luồng dữ liệu:** Dữ liệu được trích xuất thông qua Data API và Crawl data bằng BeautifulSoup sau đó được lưu trữ và xử lý trong Hadoop HDFS. Apache Spark được sử dụng để thực hiện các tác vụ biến đổi và tải dữ liệu. Dữ liệu đã được xử lý sẽ được lưu trữ trong Apache Hbase. Flask được sử dụng để làm Back-end còn lại NodeJS dùng để visualization và báo cáo dữ liệu. Toàn bộ hệ thống sẽ được triển khai và quản lý trong môi trường docker.

**Biểu đồ tương tác thành phần:**



## **Chi tiết thành phần hệ thống**

### 2.1 Hadoop HDFS

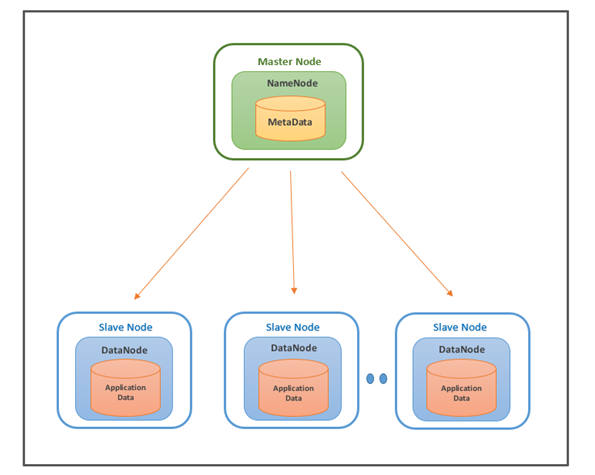
*Đặc điểm chính của HDFS*

* Lưu trữ dữ liệu lớn: HDFS tối ưu hóa để lưu trữ các tập tin lớn, chia nhỏ thành các khối dữ liệu (blocks) và phân tán trên nhiều máy trong cluster.
* Tính chịu lỗi: Dữ liệu được sao chép (replication) trên nhiều máy (DataNodes), đảm bảo khả năng phục hồi khi xảy ra lỗi phần cứng.
* Mô hình master-slave: HDFS bao gồm một NameNode quản lý metadata và nhiều DataNodes chịu trách nhiệm lưu trữ dữ liệu thực tế.
* Khả năng mở rộng: HDFS có thể mở rộng dễ dàng bằng cách thêm DataNodes mà không ảnh hưởng đến hệ thống hiện có.

*Vai trò của HDFS trong dự án*

HDFS được sử dụng như một hệ thống lưu trữ trung tâm

* Lưu trữ dữ liệu gốc (raw data) từ dịch vụ của service Flask và dữ liệu sau khi Apache Spark thực hiện xử lý.
* Tích hợp với Apache Spark để thực hiện xử lý dữ liệu lớn.
* Đảm bảo rằng dữ liệu được lưu trữ an toàn và dễ dàng truy cập từ các thành phần khác trong hệ thống, bao gồm Airflow và Apache Hbase.



### 2.2 Apache Spark

Apache Spark là một framework mã nguồn mở mạnh mẽ để tính toán phân tán, nổi bật với khả năng xử lý dữ liệu tốc độ cao nhờ thực hiện tính toán song song trên nhiều máy và ưu tiên sử dụng bộ nhớ trong (in-memory). Spark hỗ trợ xử lý dữ liệu theo thời gian thực với Spark Streaming, đồng thời tích hợp dễ dàng với nhiều hệ thống file khác nhau như HDFS, Cassandra, và S3.

Các thành phần chính:

* **Spark Core**: Đảm nhiệm tính toán và xử lý dữ liệu trong bộ nhớ, làm nền tảng cho các thành phần khác.
* **Spark SQL**: Hỗ trợ thao tác trên dữ liệu có cấu trúc và bán cấu trúc, cung cấp giao diện sử dụng ngôn ngữ SQL.
* **Spark NLP**: Hỗ trợ thực hiện các tác vụ NLP quy mô lớn trong môi trường phân tán.

### 2.3 Apache airflow

Apache Airflow là một công cụ mã nguồn mở được sử dụng để lập lịch, quản lý, và giám sát các quy trình xử lý dữ liệu. Airflow cho phép xây dựng và chạy các quy trình công việc, được biểu diễn dưới dạng Directed Acyclic Graphs (DAGs)

Operator trong airflow đại diện cho một công việc cụ thể trong quy trình. Một số operator điển hình:

* **BashOperator:** Chạy các lệnh Bash hoặc script Shell.
* **PythonOperator:** Thực thi các hàm Python.
* **SparkSqlOperator:** Thực hiện các truy vấn Spark SQL.
* **BranchPythonOperator:** Logic phân nhánh trong DAG
* **SimpleHttpOperator:** Thực hiện các yêu cầu HTTP (REST API)

Trong hệ thống này, airflow được sử dụng để lập lịch chạy có các tác vụ lấy data từ api và thực hiện ETL (Extract – Transform – Load) data. Các tác vụ sẽ được thiết lập chạy mỗi ngày để database có thể cập nhật được dữ liệu mới nhất hằng ngày.

# **III. Chi tiết triển khai**

## **Crawling data**

### Crawl theo tên bài viết

Phương thức **crawl\_keywords** đọc danh sách từ khóa, thực hiện thu thập dữ liệu trên trang Wiki bằng phương thức **crawl**, trích xuất dữ liệu từ HTML phản hồi bằng phương thức **scrape\_data** và và lưu file dưới dạng csv bằng phương thức **save\_to\_csv**

def crawl\_keywords(api\_token, keywords\_file, output\_file):

    # Read keywords from file

    with open(keywords\_file, 'r', encoding='utf-8') as file:

        keywords = [line.strip() for line in file.readlines() if line.strip()]

    scraped\_data\_list = []

    cnt=0

    # Crawl data for each keyword

    for keyword in keywords:

        print(cnt)

        cnt+=1

        page\_url = f"https://en.wikipedia.org/wiki/{keyword.replace(' ', '\_')}"

        print(f"Crawling data for: {keyword}")

        scraped\_data = crawl(page\_url, api\_token)

        if scraped\_data:

            scraped\_data['keyword'] = keyword

            scraped\_data\_list.append(scraped\_data)

    # Save all scraped data to a single CSV file

    save\_to\_csv(scraped\_data\_list, output\_file)

### Crawl theo URL

Phương thức **click\_and\_download** tải xuống tệp theo yêu cầu, thực hiện giải nén tệp đã lưu bằng phương thức **extract\_bz2**

def click\_and\_download(url):

try:

# Gửi yêu cầu GET tới URL

response = requests.get(url)

response.raise\_for\_status() # Kiểm tra lỗi HTTP

# Lấy tên file từ tiêu đề hoặc URL

filename = "file.bz2" # Lấy phần cuối của URL làm tên file

with open(filename, 'wb') as file:

file.write(response.content)

print(f"File đã được tải xuống và lưu với tên: {filename}")

except requests.exceptions.RequestException as e:

print(f"Đã xảy ra lỗi: {e}")

def extract\_bz2(input\_path, output\_path):

try:

with bz2.BZ2File(input\_path, 'rb') as file\_in:

with open(output\_path, 'wb') as file\_out:

file\_out.write(file\_in.read())

print(f"File đã được giải nén: {output\_path}")

except Exception as e:

print(f"Đã xảy ra lỗi: {e}")

Phương thức **upload\_to\_hdfs** lưu file đã tải lên HDFS

def upload\_to\_hdfs(local\_path, hdfs\_path):

try:

# Chạy lệnh hdfs dfs -put

subprocess.run(["hdfs", "dfs", "-put", local\_path, hdfs\_path], check=True)

print(f"File {local\_path} đã được upload lên HDFS tại {hdfs\_path}")

except subprocess.CalledProcessError as e:

print(f"Đã xảy ra lỗi khi upload file lên HDFS: {e}")

## **Hadoop HDFS**

### HDFS trong dự án này được triển khai với cấu trúc

* 1 NameNode: Quản lý metadata của hệ thống, chẳng hạn như vị trí lưu trữ của các khối dữ liệu.
* 3 DataNodes: Lưu trữ các khối dữ liệu. Mỗi DataNode có vai trò chịu trách nhiệm lưu trữ và trả dữ liệu theo yêu cầu từ NameNode.
* ResourceManager: Quản lý tài nguyên và điều phối các ứng dụng Hadoop.
* NodeManagers: Theo dõi tài nguyên sử dụng trên node cục bộ, giao tiếp với ResourceManager để nhận nhiệm vụ.
* HistoryServer: Lưu trữ dữ liệu lịch sử từ ResourceManager, cho phép kiểm tra và phân tích các tác phụ đã hoàn thành.

### Mô hình triển khai

HDFS được triển khai trong môi trường Docker Compose với các thành phần chính sau:

* NameNode: Sử dụng image *bde2020/hadoop-namenode*, triển khai trong container **namenode**, với các cổng 9870 (Web UI) và 9000 (giao tiếp với client)
* DataNodes: Ba DataNodes được triển khai bằng *image bde2020/hadoop-datanode*. Chúng lưu trữ các khối dữ liệu và phục vụ yêu cầu từ NameNode.
* Cấu hình: các cấu hình chi tiết được định nghĩa trong file hadoop.env với các điểm nổi bật sau
  + HDFS: WebHDFS enabled, permissions disabled, flexible hostname registration.

*Tích hợp HDFS qua docker-compose*

namenode:

    image: bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-java8

    container\_name: namenode

    restart: always

    ports:

      - 9870:9870

      - 9000:9000

    volumes:

      - hadoop\_namenode:/hadoop/dfs/name

    environment:

      - CLUSTER\_NAME=test

    env\_file:

      - ./hadoop.env

    networks:

      - hadoop-network

# HDFS Datanodes

datanode:

    image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8

    container\_name: datanode

    restart: always

    volumes:

      - hadoop\_datanode:/hadoop/dfs/data

    environment:

      SERVICE\_PRECONDITION: "namenode:9870"

    env\_file:

      - ./hadoop.env

    networks:

      - hadoop-network

datanode2:

    image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8

    container\_name: datanode2

    restart: always

    volumes:

      - hadoop\_datanode2:/hadoop/dfs/data

    environment:

      SERVICE\_PRECONDITION: "namenode:9870"

    env\_file:

      - ./hadoop.env

    networks:

      - hadoop-network

datanode3:

    image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8

    container\_name: datanode3

    restart: always

    volumes:

      - hadoop\_datanode3:/hadoop/dfs/data

    environment:

      SERVICE\_PRECONDITION: "namenode:9870"

    env\_file:

      - ./hadoop.env

    networks:

      - hadoop-network

# ResourceManager

resourcemanager:

    image: bde2020/hadoop-resourcemanager:2.0.0-hadoop3.2.1-java8

    container\_name: resourcemanager

    restart: always

    environment:

      SERVICE\_PRECONDITION: "namenode:9000 namenode:9870 datanode:9864"

    env\_file:

      - ./hadoop.env

    networks:

      - hadoop-network

# NodeManagers

nodemanager1:

    image: bde2020/hadoop-nodemanager:2.0.0-hadoop3.2.1-java8

    container\_name: nodemanager

    restart: always

    environment:

      SERVICE\_PRECONDITION: "namenode:9000 namenode:9870 datanode:9864 resourcemanager:8088"

    env\_file:

      - ./hadoop.env

    networks:

      - hadoop-network

# HistoryServer

historyserver:

    image: bde2020/hadoop-historyserver:2.0.0-hadoop3.2.1-java8

    container\_name: historyserver

    restart: always

    environment:

      SERVICE\_PRECONDITION: "namenode:9000 namenode:9870 datanode:9864 resourcemanager:8088"

    volumes:

      - hadoop\_historyserver:/hadoop/yarn/timeline

    env\_file:

      - ./hadoop.env

    networks:

      - hadoop-network

## **Spark**

### Lưu dữ liệu vào Hbase

#### Tách dữ liệu từ file xml

Đọc dữ liệu từ các thẻ xml sử dụng thư viện spark-xml vào Dataframe Spark, lựa chọn các trường cần thiết trước khi lưu vào HBase

df = spark.read.format("xml").option("rowTag", "page").option("inferSchema", "true").load('hdfs://namenode:9000' + PATH\_TO\_FILE)

df = df.withColumn(

"title",

lower(trim(regexp\_replace(regexp\_replace(col("title"), r'[^\w\s]', ' '), r'\s+', ' ')))

)

df\_flat = df.select(

col("id"),

col("ns"),

col("redirect.\_VALUE").alias("redirect\_value"),

col("redirect.\_title").alias("redirect\_title"),

col("revision.comment").alias("revision\_comment"),

col("revision.contributor.id").alias("contributor\_id"),

col("revision.contributor.ip").alias("contributor\_ip"),

col("revision.contributor.username").alias("contributor\_username"),

col("revision.format").alias("revision\_format"),

col("revision.id").alias("revision\_id"),

col("revision.minor").alias("revision\_minor"),

col("revision.model").alias("revision\_model"),

col("revision.origin").alias("revision\_origin"),

col("revision.parentid").alias("revision\_parentid"),

col("revision.sha1").alias("revision\_sha1"),

col("revision.text.\_VALUE").alias("content"),

col("revision.text.\_bytes").alias("text\_bytes"),

col("revision.text.\_sha1").alias("text\_sha1"),

col("revision.text.\_xml:space").alias("text\_xml\_space"),

col("revision.timestamp").alias("timestamp"),

col("title")

)

#### Lưu dữ liệu vào Hbase

Phương thức write\_to\_hbase tạo kết nối và lưu giá trị của từng hàng vào Hbase

def write\_to\_hbase(partition):

connection = happybase.Connection('hbase-master')

connection.open()

table = connection.table('wikipedia')

for row in partition:

row\_key = str(row["title"]).encode('utf-8') if row["title"] else b'unknown'

table.put(row\_key, {

b'cf:title': str(row["title"]).encode('utf-8') if row["title"] else b'',

b'cf:id': str(row["id"]).encode('utf-8') if row["id"] else b'',

b'cf:ns': str(row["ns"]).encode('utf-8') if row["ns"] else b'',

b'cf:redirect\_value': str(row["redirect\_value"]).encode('utf-8') if row["redirect\_value"] else b'',

b'cf:redirect\_title': str(row["redirect\_title"]).encode('utf-8') if row["redirect\_title"] else b'',

b'cf:revision\_comment': str(row["revision\_comment"]).encode('utf-8') if row["revision\_comment"] else b'',

b'cf:contributor\_id': str(row["contributor\_id"]).encode('utf-8') if row["contributor\_id"] else b'',

b'cf:contributor\_ip': str(row["contributor\_ip"]).encode('utf-8') if row["contributor\_ip"] else b'',

b'cf:contributor\_username': str(row["contributor\_username"]).encode('utf-8') if row["contributor\_username"] else b'',

b'cf:revision\_format': str(row["revision\_format"]).encode('utf-8') if row["revision\_format"] else b'',

b'cf:revision\_id': str(row["revision\_id"]).encode('utf-8') if row["revision\_id"] else b'',

b'cf:revision\_minor': str(row["revision\_minor"]).encode('utf-8') if row["revision\_minor"] else b'',

b'cf:revision\_model': str(row["revision\_model"]).encode('utf-8') if row["revision\_model"] else b'',

b'cf:revision\_origin': str(row["revision\_origin"]).encode('utf-8') if row["revision\_origin"] else b'',

b'cf:revision\_parentid': str(row["revision\_parentid"]).encode('utf-8') if row["revision\_parentid"] else b'',

b'cf:revision\_sha1': str(row["revision\_sha1"]).encode('utf-8') if row["revision\_sha1"] else b'',

b'cf:revision\_timestamp': str(row["timestamp"]).encode('utf-8') if row["timestamp"] else b'',

b'cf:revision\_text\_VALUE': str(row["content"]).encode('utf-8') if row["content"] else b'',

b'cf:revision\_text\_bytes': str(row["text\_bytes"]).encode('utf-8') if row["text\_bytes"] else b'',

b'cf:revision\_text\_sha1': str(row["text\_sha1"]).encode('utf-8') if row["text\_sha1"] else b'',

b'cf:revision\_text\_xmlspace': str(row["text\_xml\_space"]).encode('utf-8') if row["text\_xml\_space"] else b'',

})

connection.close()

print("Data written to HBase\n")

### Chuyển đổi dữ liệu

#### Làm sạch dữ liệu

**Từ dữ liệu trích xuất từ file xml, thực hiện làm sạch (loại bỏ ký tự đặc biệt, số, khoảng trắng dư thừa) và chuyển về dạng chữ thường**

df\_flat = df\_flat.withColumn(

"content",

lower(trim(regexp\_replace(regexp\_replace(regexp\_replace(col("content"), r'[^\w\s]', ' '), r'\s+', ' '), r'\d+', '')))

).withColumn(

"revision\_comment",

lower(trim(regexp\_replace(regexp\_replace(col("revision\_comment"), r'[^\w\s]', ' '), r'\s+', ' '))) # Xử lý comment

).withColumn(

"title",

lower(trim(regexp\_replace(regexp\_replace(col("title"), r'[^\w\s]', ' '), r'\s+', ' '))) # Xử lý comment

)

#### Trích xuất từ khóa và hợp nhất các trường

Thực hiện trích xuất các từ từ 2 cột “title” và “revision\_comment” và gộp thành 1 dataframe duy nhất (thêm cột “location” để xác định nguồn gốc từ)

df\_new = df\_flat.select(

col("title"),

explode(

split(col("title"), r"\s+") # Tokenize Column1

).alias("word")

).withColumn("location", lit("title")) \

.union(df\_flat.select(

col("title"),

explode(

split(col("revision\_comment"), r"\s+") # Tokenize Column2

).alias("word")

).withColumn("location", lit("revision\_comment")))

#### Thực hiện lemmatization trên dữ liệu

Tạo Spark NLP Pipeline để thực hiện **lemmatization trên các từ của cột “word”**

# Sử dụng mô hình đã lưu

lemmatizer = LemmatizerModel.pretrained("lemma\_antbnc") \

.setInputCols(["token"]) \

.setOutputCol("lemma")

# Tạo DocumentAssembler

document\_assembler = DocumentAssembler() \

.setInputCol("word") \

.setOutputCol("document")

# Tạo Tokenizer

tokenizer = Tokenizer() \

.setInputCols(["document"]) \

.setOutputCol("token")

# Xây dựng Pipeline

pipeline = Pipeline(stages=[

document\_assembler,

tokenizer,

lemmatizer

])

# Thực thi pipeline

model = pipeline.fit(df\_new)

result = model.transform(df\_new)

# Thêm cột mới 'first\_lemma' với phần tử đầu tiên của mảng 'lemma.result'

result = result.withColumn("first\_lemma", col("lemma.result").getItem(0))

result = result.drop("token")

result = result.drop("document")

result = result.drop("lemma")

result = result.drop("word")

result = result.withColumnRenamed("first\_lemma", "word")

#### Tính tần suất xuất hiện của mỗi từ và tạo Chỉ mục đảo ngược

Sử dụng groupBy() và agg() để nhóm các các từ lại và đếm số lần xuất hiện

# Đếm số lần xuất hiện của mỗi từ trong từng vị trí

df\_count = result.groupBy("word", "title", "location").agg(

count("\*").alias("count")

)

# Tổng hợp thành Inverted Index với số lần xuất hiện

inverted\_index\_with\_count = df\_count.groupBy("word").agg(

collect\_list(

struct(

col("title").alias("doc\_id"),

col("location"),

col("count").alias("frequency")

)

).alias("locations")

)

Tạo chỉ mục của từ theo cấu trúc: [{doc\_id: "doc1", location: "title", frequency: 3}]

# Tổng hợp thành Inverted Index với số lần xuất hiện

inverted\_index\_with\_count = df\_count.groupBy("word").agg(

collect\_list(

struct(

col("title").alias("doc\_id"),

col("location"),

col("count").alias("frequency")

)

).alias("locations")

)

#### Lưu file chỉ mục đảo ngược vào HDFS

Lưu kết quả của inverted index vào HDFS dưới định dạng Parquet theo đường dẫn

inverted\_index\_with\_count.write.parquet("hdfs://namenode:9000/user/root/index/final.parquet")

### Cập nhật chỉ mục đảo ngược

Khi bổ sung dữ liệu mới, thực hiện cập nhật chỉ mục đảo ngược và lưu lại vào HDFS.

old\_df = spark.read.parquet("hdfs://namenode:9000/user/root/index/final.parquet")

inverted\_index\_with\_count = spark.read.parquet("hdfs://namenode:9000/user/root/index/mini\_index.parquet")

combined\_df = old\_df.union(inverted\_index\_with\_count)

combined\_df.show(truncate=False)

# Group theo cột "word" và gom các "locations" thành một danh sách duy nhất

combined\_df\_grouped = combined\_df.groupBy("word").agg(

flatten(collect\_list("locations")).alias("locations")

)

# Hiển thị kết quả

combined\_df\_grouped.show(truncate=False)

combined\_df\_grouped.write.mode("overwrite").parquet("hdfs://namenode:9000/user/root/index/final.parquet")

### Tìm kiếm từ khóa

#### Xử lý từ (cụm từ) cần tìm kiếm nhận được từ API

user\_query = request.json.get("search", "")

user\_query = user\_query.lower()

user\_query = re.sub(r"[^a-zA-Z0-9]", " ", user\_query)

#### Tìm kiếm dữ liệu trong Hbase

Kết nối đến Hbase thông qua happybase

connection = happybase.Connection('hbase-master')

connection.open()

table2 = connection.table('top\_scores')

table = connection.table('wikipedia')

Kiểm tra bảng top\_scores (lưu trữ các truy vấn đã thực hiện trước đó), thực hiện lấy row\_key

row2 = table2.row(user\_query.encode('utf-8'))

row\_key = row2.get(b'cf:top1').decode('utf-8') # Lấy giá trị cột 'top1'

Tìm kiếm khi truy vấn được thực hiện lần đầu tiên, và lưu giá trị tìm được vào bảng top\_scores

key\_words = user\_query.split()

df = spark.read.parquet("hdfs://namenode:9000/user/root/index/final.parquet")

result\_df = df.filter(col("word").isin(key\_words))

# Tách cột locations (explode để xử lý từng phần tử)

exploded\_df = result\_df.withColumn("location\_item", explode(col("locations")))

# Trích xuất các trường từ StructType trong cột locations

processed\_df = exploded\_df.select(

col("word"),

col("location\_item.doc\_id").alias("doc\_id"),

col("location\_item.location").alias("position"),

col("location\_item.frequency").alias("count")

)

# Tính điểm dựa trên vị trí

scored\_df = processed\_df.withColumn(

"score",

when(col("position") == "revision\_comment", col("count") \* 5)

.when(col("position") == "title", col("count") \* 150)

.otherwise(col("count") \* 1)

)

# Tính tổng điểm cho mỗi bài viết (doc\_id)

aggregated\_df = scored\_df.groupBy("doc\_id").agg(sum("score").alias("total\_score"))

# Sắp xếp theo tổng điểm và lấy top 3 bài viết có điểm cao nhất

top\_3\_ids = aggregated\_df.orderBy(col("total\_score").desc()).limit(5)

# Get the table object

first\_row = top\_3\_ids.first() # Lấy phần tử đầu tiên

row\_key = first\_row["doc\_id"]

# Lưu vào HBase

table2.put(user\_query.encode('utf-8'), {

b'cf:top1': str(top\_3\_ids.collect()[0]["doc\_id"]).encode('utf-8'), # top1

b'cf:top2': str(top\_3\_ids.collect()[1]["doc\_id"]).encode('utf-8'), # top2

b'cf:top3': str(top\_3\_ids.collect()[2]["doc\_id"]).encode('utf-8'), # top3

b'cf:top4': str(top\_3\_ids.collect()[3]["doc\_id"]).encode('utf-8'), # top3

b'cf:top5': str(top\_3\_ids.collect()[4]["doc\_id"]).encode('utf-8') # top3

})

Gửi dữ liệu cho Client thông qua REST API của Flask

row.get(b'cf:revision\_text\_VALUE') else 'No Content'

return jsonify({"title": title, "text": content})

## Airflow

Sử dụng airflow để thiết lập lịch chạy định kỳ cho các tác vụ:

* Crawl dữ liệu từ trang Wiki
* Thực hiện lưu dữ liệu vào Hbase
* Thực hiện transform dữ liệu, cập nhật danh sách chỉ mục và lưu vào HDFS

Thiết lập các tham số cho Dags

default\_args = {

'owner': 'airflow', # Người sở hữu DAG

'retries': 1, # Số lần thử lại nếu job thất bại

'retry\_delay': timedelta(minutes=5), # Độ trễ giữa các lần thử lại

'start\_date': datetime(2024, 12, 17), # Thời gian bắt đầu DAG

'catchup': False, # Không chạy các task đã bỏ lỡ trước thời điểm start\_date

'depends\_on\_past': True, # Phụ thuộc vào task trước đó

'email\_on\_failure': False, # Không gửi email khi task thất bại

'email\_on\_retry': False, # Không gửi email khi task thử lại

}

# Tạo DAG

dag = DAG(

'spark\_jobs\_dag', # Tên DAG

default\_args=default\_args, # Tham số mặc định

description='DAG để chạy các Spark job',

schedule\_interval='@daily', # Lịch chạy DAG, ví dụ: chạy mỗi ngày

catchup=False, # Không chạy các task đã bỏ lỡ

)

Ngày bắt đầu chạy dag là 17/12/2024. Dag sẽ được chạy vào 0 giờ mỗi ngày. Số lần retry là 1 và khoảng thời gian giữa các lần retry là 5 phút.

Thiết lập các task:

* Task\_1: thực hiện gọi API để lấy dữ liệu từ Wiki
* Task\_2: Xử lý dữ liệu được crawl và lưu vào Hbase
* Task\_3: Xử lý dữ liệu, tạo chỉ mục đảo ngược, lưu vào HDFS

Task task\_1 sử dụng SparkSubmitOperato:

def run\_spark\_job\_1():

"""Chạy Spark job 1"""

subprocess.run(['spark-submit', 'crawl.py'], check=True)

task\_1 = PythonOperator(

task\_id='crawl', # Tên của task

python\_callable=run\_spark\_job\_1, # Hàm Python gọi Spark job 1

dag=dag, # DAG mà task thuộc về

)

Task task\_2 sử dụng SparkSubmitOperator:

def run\_spark\_job\_2():

"""Chạy Spark job 2"""

subprocess.run(['spark-submit', 'put\_to\_hbase.py'], check=True)

task\_2 = PythonOperator(

task\_id='put\_to\_hbase', # Tên của task

python\_callable=run\_spark\_job\_2, # Hàm Python gọi Spark job 2

dag=dag, # DAG mà task thuộc về

)

Task task\_3 sử dụng SparkSubmitOperator:

def run\_spark\_job\_3():

"""Chạy Spark job 3"""

subprocess.run(['spark-submit', 'extract.py'], check=True)

task\_3 = PythonOperator(

task\_id='extract', # Tên của task

python\_callable=run\_spark\_job\_3, # Hàm Python gọi Spark job 3

dag=dag, # DAG mà task thuộc về

)

Thiết lập thứ tự thực hiện các task trong DAG

task\_1 >> task\_2 >> task\_3

## **ReactJs**

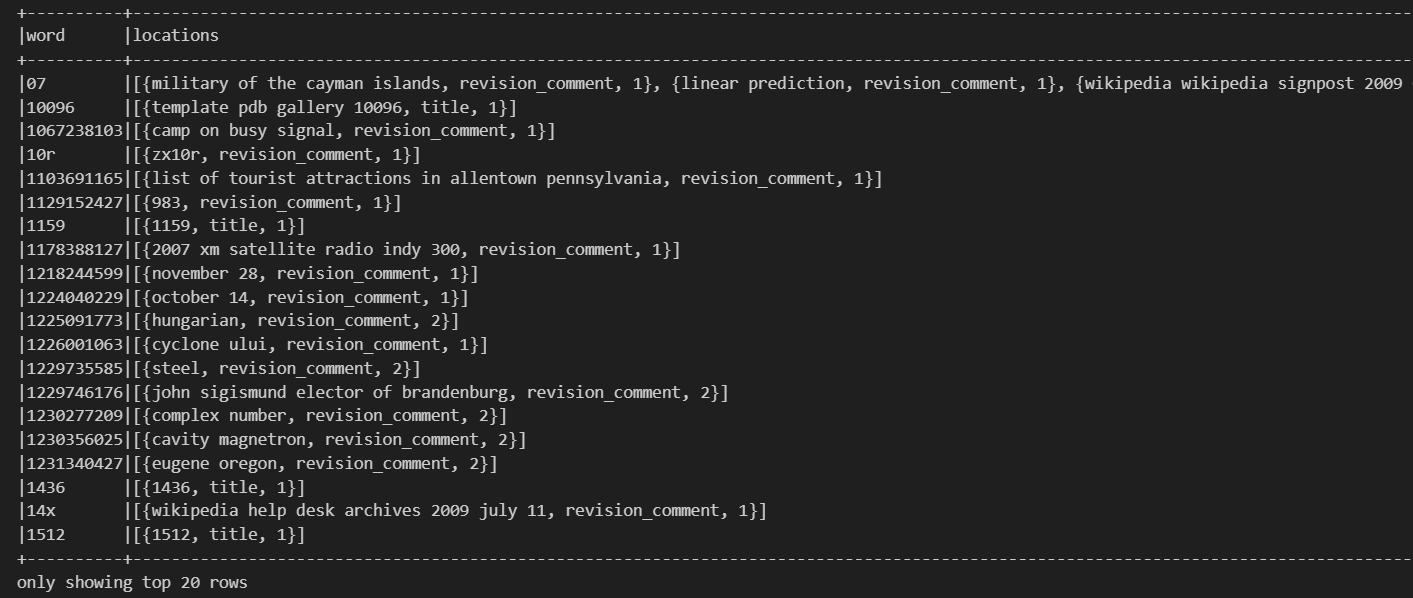
Cung cấp giao diện và gọi API tìm kiếm đến Flask App và hiển thị kết quả

# **IV. Bài học kinh nghiệm**

## **1. Kinh nghiệm merge các bảng**

Hoàn cảnh:

Trong quá trình xây dựng inverted\_index, những bảng dữ liệu gốc có hàng triệu bản ghi, trong khi những dữ liệu mới đến thì chỉ có vài chục, vài trăm ngàn bản ghi và nhiệm vụ của chúng ta là merge 2 bản ghi này lại thành một bản ghi duy nhất.



Phương pháp đã thử:

1. Union 2 bảng lại với nhau, sau đó flat cột locations rồi merge bằng groupBy => Đơn giản, dễ hiểu nhưng kích thước dữ liệu tăng, đặc biệt các chi phí broadcast, suffle tăng cao. Trong khi locations rất dài (nếu tách riêng ra các thành phần bên trong của locations thì có thể lên đến hàng tỷ dòng) => việc nhóm lại các dòng với groupBy mất thời gian và tài nguyên tính toán. => không hiệu quả, bị error 137: out of memory (OOM)

2. Repartition theo cột word + sử dụng phép toán join(full outer join): Thời gian có giảm đáng kể so với cách ban đầu.

3. Sử dụng broadcast join(full outer join) + chia parititon cho bảng nhỏ để tối ưu việc merge 2 bảng này: sử dụng bảng nhỏ hơn đánh dấu là broadcast để join => Hiệu quả hơn chút so với cách 2 nhưng lại chậm hơn cách không chia partition (dữ liệu bảng nhỏ khoảng 50MB) và chúng ta chỉ có 3 worker node nên k gặp vấn đề overhead và tận dụng được tối đa khả năng của broadcast join. => Đây là cách được sử dụng

Một số bài học rút ra:

Khi join các bảng theo key mà trong đó 1 bảng lớn 1 bảng nhỏ thì có thể sử dụng broadcast join( gửi bảng nhỏ tới tất cả các node trong cụm worker và nó giảm được việc suffle). Tuy nhiên khi dữ liệu của bảng nhỏ không còn đủ nhỏ hoặc trong cụm có khá nhiều worker thì sẽ sinh ra chi phí broadcast, dẫn đến sử dụng bộ nhớ kém hiệu quả của các worker (VD: dataframe 200MB mà có 100 worker => chi phí sẽ là 200\*100= 20GB)

## **2. Kinh nghiệm xây dựng search:**

Search dữ liệu là khi server nhận được truy vấn từ Front-end, sẽ tìm ra những văn bản có “liên quan” nhất bằng việc search các từ tính số lần xuất hiện và vị trí xuất hiện => tính điểm và trả về cho bên front-end.

Chẳng hạn như tìm kiếm các từ khóa [“Computer”, “vision”, “bigdata”] trong bảng inverted\_index thì sẽ tìm các dòng có cột word có những giá trị này, sau đó sẽ tính điểm dựa vào cột locations thu được.

Bảng inverted\_index có nhiều dòng(do phần làm sạch chưa tốt hẳn nên có khá nhiều bản ghi ~ vài triệu dòng) dẫn đến việc tìm kiếm thông thường (where, filter) không hiệu quả nếu không có phương pháp!

Chúng ta sẽ bật spark-session và load dataframe sẵn, cache luôn toàn bộ dữ liệu lại trước khi bắt đầu cho thực hiện truy vấn!

Phương pháp đã thử:  
1. Lọc thông thường: sử dụng các phép toán where - filter hay isin(cho chuỗi) để lọc các dòng cần => Đơn giản nhất nhưng lâu nhất

2. Lưu inverted\_index partition theo cột word và dùng binary\_seach để lấy trực tiếp dữ liệu => tốc độ tìm kiếm từ xuất hiện chỉ là O(logn) nên nhanh, tuy nhiên có hàng triệu bản ghi mỗi bản ghi thì đơn vị chỉ tính bằng KB, lưu trữ như thế vào hdfs không hiệu quả, hiệu suất truy vấn (I/O) lại bị chậm và lag hoàn toàn ở máy namenode => Không hợp lý

3. Repatition hay bucket theo cột word => dẫn đến có hàng triệu partition => cũng không hợp lý.

4. Repatition theo chữ cái đầu: Lưu trữ trên HDFS không bị Metadata overhead đồng thời quá trình search cũng cải thiện rõ rệt so với phương pháp 1 ban đầu. Tuy nhiên thì thời gian truy vấn thực tế vẫn khá lâu: Có thể lên tới chục giây cho một truy vấn thông thường => đang cân nhắc dùng demo so sánh hiệu quả.

5. Xây dựng thuật toán Binary search cho cột word nhưng gặp vấn đề bộ nhớ khi chuyển cột word thành list. Hơn nữa trong môi trường phân tán thì nó cần suffle khá nhiều trong quá trình sắp xếp theo cột word => Không khả thi

6. Sử dụng Hbase làm nơi lưu trữ thay vì hdfs dưới dạng file parquet:

Hbase có ưu điểm truy vấn khá tốt theo key và thời gian truy vấn giảm xuống chỉ còn 0,x giây => Hoạt động ấn tượng.

Một số bài học rút ra:

Tránh lưu file quá nhỏ vào hdfs => giảm hiệu suất hệ thống hdfs, tìm ra “chiến thuật” hợp lý để partition theo mục đích và lưu trữ.

Trong quá trình biến đổi

Spark phù hợp xử lý những tác vụ lớn, phức tạp. Tuy nhiên nếu yêu cầu không phức tạp và dữ liệu không “lớn” thì hãy tìm cách khác

**3. Một số kinh nghiệm khác:**

**- Không nên dùng các UDF vì hiệu suất thực sự không cao**

**- Trong quá trình truy vấn hạn chế các thao tác action không cần thiết gây tốn kém như collect(), show(), count() để tránh ảnh hưởng hiệu suất.**

**- Tại các bước mà RDD phân ra nhiều nhánh thì nên caching lại dữ liệu.**

**- Quá nhiều hay quá ít partition cũng không hợp lý, kích thước hợp lý là** 100MB đến 200MB và có thể cân nhắc chia theo bội số của số executor.

## **2. Setup HDFS**

**Mô tả vấn đề:** Khi sử dụng HDFS một số tham số cần quan tâm như số datanode, replication, blocksize, việc chia rack việc sử dụng tham số replication có thể gây ra sự đánh đổi (trade-off) giữa Reliability, Availability và Network bandwidth utilization.

**Giải pháp đã tìm hiểu:**

* Replication mặc định là 3 (với số lượng datanode >= 3)
* Sử dụng rack chỉ phù hợp cho dự án có nhiều datanode (vài chục) sẽ khiến cho hệ thống bền vững và tăng khả năng chịu lỗi.
* Sử dụng Replication lớn tăng tính tin cậy của hệ thống (relicability) và tính khả dụng (availability) tuy nhiên sẽ làm giảm hiệu năng khi phải sử dụng đến khả năng kết nối giữa các datanode đặc biết là với write operation
* Sử dụng Replication nhỏ thì ngược lại giảm tính tin cậy và khả dụng tuy nhiên sẽ tăng hiệu năng của write operation và tận dụng được network bandwidth utilization.
* Sử dụng rack sẽ gây phức tạp và không phù hợp với cá project cá nhân có số lượng datanode nhỏ

**Giải pháp cuối cùng:**

* Với dự án quy mô nhỏ như project môn học sử dụng số lượng datanode nhỏ (3 datanodes) và replication nhỏ (2 or 3) sẽ phù hợp, không chia rack vì sẽ gây phức tạp lớn và không cần thiết cũng như giảm hiệu năng khi phải comunication giữa các rack

**Bài học rút ra:**

* Best practice: Sử dụng replication lớn nếu hệ thống cần khả năng tin cậy cao mà hiệu năng thì vừa phải, replication nhỏ hơn nếu hệ thống cần tối ưu write operation và network bandwidth utilization
* Sử dụng rack khi hệ thống scale lớn quy mô vài chục data node tuy nhiên với hệ thống nhỏ thì không cần thiết chia rack.

## **3. Sử dụng Spark ghi dữ liệu vào Apache Hbase**

**Mô tả vấn đề:**

* Cần ghi DataFrame lớn từ Spark vào Apache Hbase database
* Các issues hay gặp: connection timeout, memory overflow, performance kém, định dạng dữ liệu
* Impact: Ảnh hưởng đến thời gian hoàn thành job và resource usage

**Các giải pháp đã thử**

Sử dụng REST API của Hbase để truy vấn lưu trữ dữ liệu

import requests

def save\_to\_hbase (data):

response = requests.post('http://hbase-master:8080/table\_name', json=data)

if response.status\_code == 200:

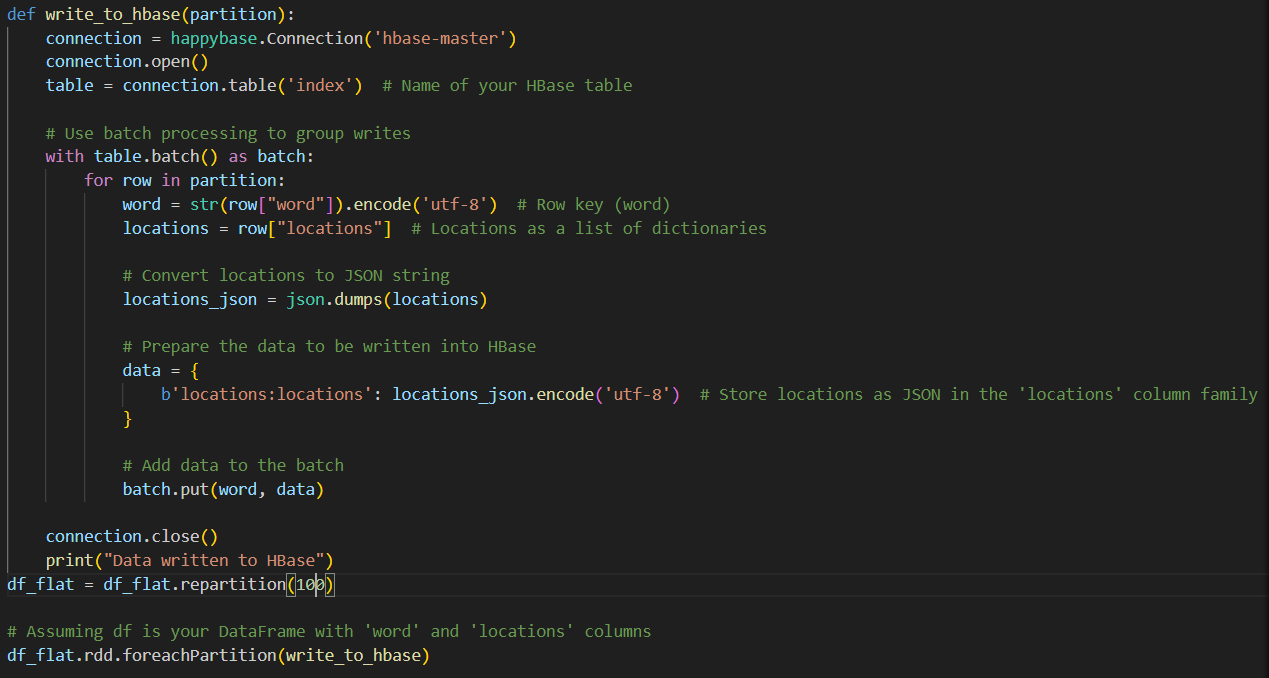
print("Data written successfully")

else:

print("Error writing data")

Đơn giản những dễ độ trễ cao, hạn chế chức năng sử dụng

**Ghi theo từng bacth sẽ giảm lượng lớn số lượng connection được tạo ra, điều chỉnh số lượng phù hợp với khả năng kết nối**



## 4**. Lựa chọn giữa Selenium và CrawlAPI**

#### Mô tả vấn đề: Khi thu thập dữ liệu từ các trang web, cần xem xét các yếu tố như tốc độ, khả năng xử lý nội dung động (JavaScript), tài nguyên hệ thống, và độ phức tạp của việc triển khai. Selenium và CrawlAPI là hai công cụ phổ biến nhưng có sự khác biệt lớn trong cách thức hoạt động và khả năng phù hợp với từng bài toán cụ thể.

#### Giải pháp đã tìm hiểu:

* **Selenium:**
  + **Ưu điểm:**
    - Hỗ trợ tốt các trang web yêu cầu JavaScript hoặc AJAX để hiển thị nội dung.
    - Linh hoạt khi xử lý các tương tác phức tạp như cuộn trang, nhấp chuột, hoặc xử lý CAPTCHA.
    - Có khả năng hiển thị giao diện trực quan giúp kiểm tra và debug dễ dàng.
  + **Nhược điểm:**
    - Chạy chậm do phải tải toàn bộ nội dung của trang web (CSS, hình ảnh, JavaScript).
    - Tiêu thụ nhiều tài nguyên hệ thống (RAM và CPU).
    - Phức tạp khi triển khai trên máy chủ không có giao diện đồ họa, yêu cầu cấu hình trình duyệt headless.
* **CrawlAPI:**
  + **Ưu điểm:**
    - Tốc độ nhanh hơn do chỉ tải nội dung HTML, không cần render JavaScript hoặc các thành phần giao diện khác.
    - Dễ triển khai trên môi trường máy chủ headless, không yêu cầu cài đặt thêm trình duyệt.
    - Tiêu thụ ít tài nguyên hệ thống.
  + **Nhược điểm:**
    - Không thể xử lý nội dung động (JavaScript, AJAX).
    - Hạn chế khi cần thao tác phức tạp hoặc crawl các trang có cơ chế bảo vệ bot mạnh.
    - Phụ thuộc vào giới hạn và chi phí của dịch vụ API.

#### Giải pháp cuối cùng:

* **CrawlAPI** được chọn khi:
  + Dữ liệu cần thu thập nằm trên các trang web tĩnh, không phụ thuộc vào JavaScript.
  + Yêu cầu tốc độ nhanh và hiệu suất cao.
  + Dự án cần triển khai trên máy chủ có tài nguyên hạn chế.

#### Bài học rút ra:

* Sử dụng **CrawlAPI** cho các bài toán cần hiệu suất cao, không yêu cầu xử lý nội dung động.
* Sử dụng **Selenium** khi thu thập dữ liệu từ các trang động hoặc cần thao tác phức tạp.
* Đánh giá kỹ yêu cầu và nguồn lực của dự án trước khi lựa chọn công cụ crawl.

# **V. Kết luận**

## **1. Kết luận**

Qua quá trình thực hiện đề tài "Wikipedia Search Engine", nhóm đã đạt được những kết quả sau:

* Xây dựng thành công một hệ thống tìm kiếm với kiến trúc hiện đại, tận dụng các công nghệ như Hadoop, Hbase và Spark.
* Thiết lập được quy trình thu thập, xử lý và phân tích dữ liệu tự động, giúp tối ưu hóa việc tổ chức và tìm kiếm thông tin trong một hệ thống dữ liệu lớn và phức tạp.
* Áp dụng thành công các kỹ thuật ETL (Extract-Transform-Load) thông qua Spark để xử lý dữ liệu thô và chuyển đổi thành thông tin có giá trị.

## **2. Hướng phát triển**

Để tiếp tục nâng cao hiệu quả của hệ thống, nhóm đề xuất các hướng phát triển sau:

* **Mở rộng và cải thiện nguồn dữ liệu:** Kết nối với các cơ sở dữ liệu ngoài Wikipedia, như dữ liệu từ các hệ thống tìm kiếm khác hoặc các nền tảng kiến thức mở, nhằm làm phong phú thêm kết quả tìm kiếm và mở rộng phạm vi của hệ thống.
* **Nâng cao hiệu suất tìm kiếm:** Tiếp tục tối ưu hóa thuật toán tìm kiếm, cải thiện khả năng phân tích dữ liệu lớn hơn nữa, và ứng dụng các kỹ thuật mới trong Hadoop và Spark để nâng cao hiệu quả tìm kiếm trong môi trường có dữ liệu khổng lồ.
* **Cải thiện độ chính xác và tốc độ tìm kiếm:** Tối ưu hóa quy trình xử lý dữ liệu và các thuật toán tìm kiếm, sử dụng các phương pháp phân vùng dữ liệu và bộ nhớ đệm để giảm thiểu thời gian xử lý và tăng tốc độ phản hồi cho người dùng.
* **Cải thiện hiệu năng hệ thống:** Tối ưu hóa quy trình ETL để giảm thời gian xử lý, áp dụng các kỹ thuật cache và phân vùng dữ liệu hiệu quả.
* **Ứng dụng phân tích thời gian thực:** Phát triển các phương pháp phân tích dữ liệu thời gian thực, giúp cập nhật nhanh chóng các thay đổi và cải thiện kết quả tìm kiếm ngay khi dữ liệu trên Wikipedia được thay đổi hoặc cập nhật.