# ĐẠI HỌC BÁCH KHOA HÀ NỘI

# TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG


# BÁO CÁO BÀI TẬP LỚN

## Đề tài : Phân tích dữ liệu website thương mại điện tử

Học phần: IT4931 – Lưu trữ và xử lý dữ liệu lớn

Mã lớp: 154050

GVHD: TS. Trần Viết Trung

Danh sách thành viên nhóm:

| Họ và tên | Mã số sinh viên |
|---|---|
| Nguyễn Đức Duy | 20210275 |
| Đào Anh Quân | 20215631 |
| Nguyễn Đình Tùng | 20215663 |
| Vũ Thị Thanh Hoa | 20210356 |
| Ngân Văn Thiện | 20215647 |

# Mục lục

# DANH MỤC HÌNH ẢNH

## 1. Đặt vấn đề

Dữ liệu là một tài sản quý giá trong kinh doanh ngày nay. Trong lĩnh vực thương mại điện tử, việc hiểu và khai thác dữ liệu có thể mang lại lợi thế cạnh tranh lớn cho những ai biết cách vận dụng nó hiệu quả.

### 1.1. Bài toán được lựa chọn

Trong thời đại công nghệ số hiện nay, việc **phân tích dữ liệu thương mại điện tử** đã trở thành một phần không thể thiếu trong chiến lược kinh doanh của các doanh nghiệp. Dữ liệu thu thập từ hoạt động giao dịch trên nền tảng trực tuyến không chỉ giúp cải thiện trải nghiệm khách hàng mà còn hỗ trợ doanh nghiệp trong việc đưa ra quyết định thông minh và tối ưu hóa quy trình làm việc. Việc vận dụng những công cụ phân tích phù hợp sẽ giúp doanh nghiệp hiểu rõ hơn về hành vi của người tiêu dùng, từ đó phát triển chiến lược kinh doanh hiệu quả hơn.

Khi nói đến **phân tích dữ liệu trong thương mại điện tử**, bài toán chính mà nhiều doanh nghiệp gặp phải là cách thu thập, xử lý và phân tích lượng dữ liệu khổng lồ từ nhiều nguồn khác nhau như website, mạng xã hội, và các sàn thương mại điện tử. Những dữ liệu này có thể bao gồm thông tin về sản phẩm, hành vi mua sắm của khách hàng, cũng như phản hồi từ người tiêu dùng.

Việc giải quyết bài toán này đòi hỏi sự kết hợp giữa công nghệ và chiến lược kinh doanh. Doanh nghiệp cần xây dựng hệ thống quản lý dữ liệu mạnh mẽ để đảm bảo rằng mọi thông tin đều được ghi nhận và phân tích một cách chính xác. Hơn nữa, điều này cũng góp phần tăng cường khả năng dự đoán xu hướng tiêu dùng trong tương lai, từ đó đưa ra quyết định kinh doanh kịp thời và hiệu quả.

Trong bài tập lớn này, nhóm tập trung vào việc phân tích và xử lý dữ liệu từ một website thương mại điện tử. Dữ liệu được lấy từ Kaggle dưới dạng file CSV, chứa thông tin về giao dịch, hành vi người dùng, và các yếu tố khác liên quan đến hoạt động của website. Mục tiêu chính của project là xây dựng một data pipeline tự động để thu thập, biến đổi và biểu diễn dữ liệu nhằm phục vụ cho các bài toán phân tích sau này. Các công nghệ và công cụ chính được sử dụng bao gồm:

1. Thu thập dữ liệu: Dữ liệu được lấy từ Kaggle và đưa vào hệ thống qua Kafka.
2. Xử lý và lưu trữ dữ liệu theo lô:
    - Kafka: Tạo hai luồng dữ liệu, một luồng đẩy vào HDFS để xử lý theo lô.
    - HDFS (Hadoop Distributed File System): Lưu trữ dữ liệu thô trước khi xử lý.
    - Spark: Xử lý dữ liệu từ HDFS và lưu kết quả vào Cassandra.
    - Cassandra: Lưu trữ kết quả xử lý theo lô để dễ dàng truy vấn và phân tích sau này.
3. Xử lý dữ liệu thời gian thực:
    - Kafka: Luồng thứ hai được Spark Streaming xử lý.
    - Spark Streaming: Xử lý dữ liệu thời gian thực từ Kafka và đẩy kết quả vào Elasticsearch.
    - Elasticsearch: Lưu trữ và truy vấn dữ liệu thời gian thực.
    - Kibana: Trực quan hóa dữ liệu từ Elasticsearch để dễ dàng phân tích.
4. Quản lý và triển khai:
    - Docker: Tạo các container để mô phỏng máy thật, giúp môi trường phát triển và triển khai nhất quán.
    - Kubernetes (K8s): Triển khai và quản lý các container, đảm bảo tính linh hoạt và khả năng mở rộng của hệ thống.
    - Airflow: Lập lịch và quản lý các tác vụ xử lý dữ liệu, đảm bảo các pipeline chạy đúng thứ tự và thời gian.

### 1.2. Phân tích tính phù hợp của bài toán với Big Data

Bài toán phân tích dữ liệu thương mại điện tử rất phù hợp với Big Data do nó đáp ứng đủ các yếu tố 5V: Volume, Velocity, Variety, Veracity và Value:

- **Khối lượng dữ liệu lớn (Volume)**: Dữ liệu thương mại điện tử thường bao gồm thông tin về hàng triệu giao dịch, hành vi người dùng, sản phẩm, giá cả, và các thông tin liên quan khác. Những dữ liệu này được thu thập

trên quy mô lớn, có thể lên đến hàng triệu bản ghi mỗi ngày. Dữ liệu này thường được lưu trữ dưới dạng các file CSV chứa hàng nghìn, thậm chí hàng triệu dòng dữ liệu, đáp ứng được đặc trưng "Volume" của Big Data. Lượng dữ liệu khổng lồ này đòi hỏi một hệ thống phân tích dữ liệu có khả năng xử lý và lưu trữ hiệu quả.

- **Tốc độ phát sinh dữ liệu (Velocity)**: Dữ liệu thương mại điện tử phát sinh liên tục từ các hoạt động của người dùng trên website, như việc thêm sản phẩm vào giỏ hàng, thanh toán, đánh giá sản phẩm, và hành vi duyệt web. Tính chất liên tục này yêu cầu hệ thống xử lý dữ liệu phải có khả năng tiếp nhận và phân tích dữ liệu theo thời gian thực hoặc gần thời gian thực. Trong dự án này, dữ liệu được giả lập và xử lý qua Kafka, với một luồng dữ liệu xử lý theo thời gian thực qua Spark Streaming và lưu trữ vào Elasticsearch, đáp ứng đặc trưng "Velocity" của Big Data.
- **Tính đa dạng của dữ liệu (Variety)**: Dữ liệu thương mại điện tử có tính đa dạng cao, bao gồm các loại dữ liệu có cấu trúc (thông tin giao dịch, sản phẩm), dữ liệu bán cấu trúc (danh mục sản phẩm, mã giảm giá), và dữ liệu phi cấu trúc (đánh giá của người dùng, phản hồi của khách hàng). Các định dạng dữ liệu này yêu cầu các phương pháp lưu trữ và xử lý linh hoạt, với khả năng tích hợp dữ liệu từ nhiều nguồn khác nhau. Hệ thống của chúng tôi sử dụng HDFS để lưu trữ dữ liệu thô, Cassandra để lưu trữ dữ liệu đã xử lý, và Elasticsearch để lưu trữ dữ liệu thời gian thực, tất cả đều hỗ trợ các loại dữ liệu đa dạng.
- **Giá trị khai thác từ dữ liệu (Value)**: Dữ liệu thương mại điện tử mang lại giá trị lớn trong việc phân tích hành vi khách hàng, xu hướng tiêu dùng, hiệu suất sản phẩm, và tối ưu hóa chiến lược tiếp thị. Bằng cách phân tích dữ liệu này, các doanh nghiệp có thể cải thiện trải nghiệm khách hàng, tối ưu hóa chiến lược bán hàng, và đưa ra các quyết định chiến lược dựa trên dữ liệu. Các công cụ phân tích dữ liệu như Spark, Elasticsearch, và Kibana giúp biến các dữ liệu thô thành thông tin có giá trị cho các nhà quản lý và nhà phân tích.
- **Tính xác thực của dữ liệu (Veracity)**: Dữ liệu thu thập từ Kaggle và website thương mại điện tử thường được kiểm tra và làm sạch để loại bỏ các sai sót, thiếu sót, hoặc dữ liệu không hợp lệ. Trong dự án này, Spark được sử dụng để làm sạch và biến đổi dữ liệu, đảm bảo rằng chỉ dữ liệu chính xác và đáng tin cậy được đưa vào các bước phân tích tiếp theo.

### 1.3. Phạm vi và giới hạn của project

**Phạm vi của Project:**

- **Mục tiêu chính**: Mục tiêu của project là phân tích và xử lý dữ liệu thương mại điện tử lấy từ Kaggle, với các công cụ và kỹ thuật Big Data như Kafka, Spark, Cassandra, Elasticsearch, và Kibana. Các phân tích sẽ được thực hiện trên dữ liệu giao dịch của người dùng, giúp tìm ra các xu hướng, hành vi mua sắm, và tối ưu hóa chiến lược tiếp thị cho các doanh nghiệp thương mại điện tử.
- **Thu thập và xử lý dữ liệu**: Dữ liệu sẽ được thu thập từ Kaggle dưới dạng các file CSV, sau đó được đưa vào hệ thống qua Kafka và xử lý theo lô qua Spark và HDFS hoặc theo thời gian thực qua Spark Streaming. Kết quả xử lý sẽ được lưu trữ trong Cassandra cho xử lý theo lô và Elasticsearch cho xử lý thời gian thực.
- **Trực quan hóa**: Dữ liệu đã được xử lý sẽ được trực quan hóa qua Kibana, với các biểu đồ và báo cáo giúp người dùng phân tích các xu hướng và hiệu suất kinh doanh.

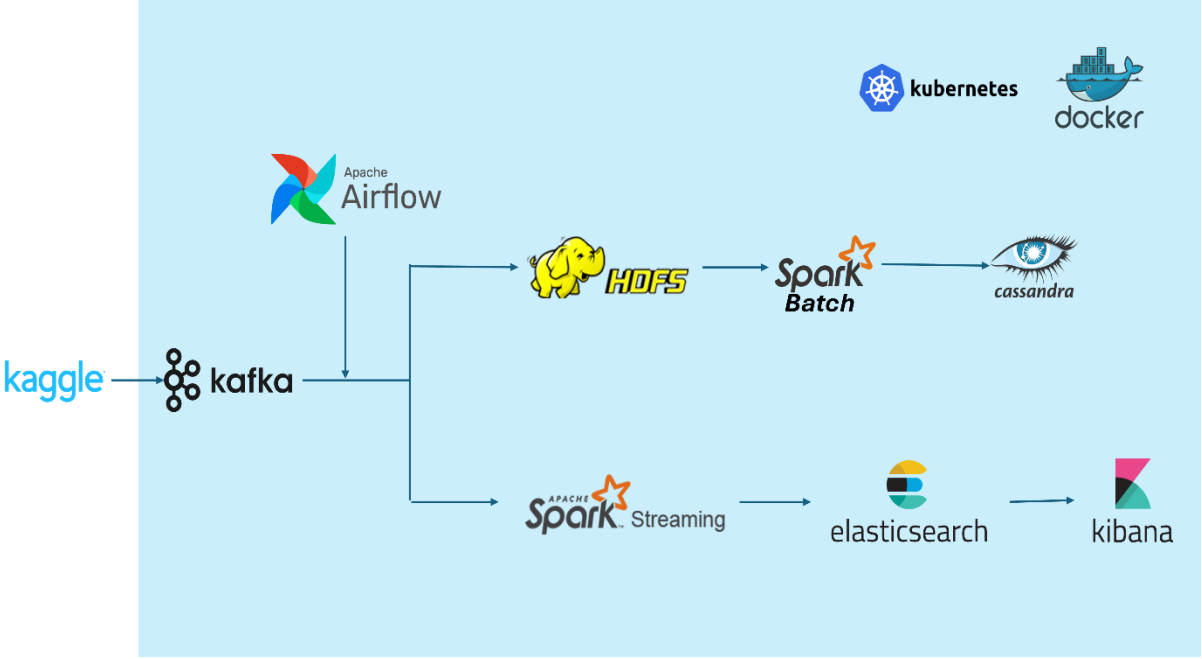
**Giới hạn của Project:**

- **Nguồn dữ liệu**: Dữ liệu chỉ được lấy từ Kaggle và chỉ giới hạn trong bộ dữ liệu đã được cung cấp, không mở rộng thêm vào các nguồn dữ liệu khác từ các hệ thống trực tuyến hay các API khác.
- **Khả năng mở rộng**: Hệ thống được xây dựng để xử lý dữ liệu với quy mô vừa phải trong môi trường học tập, không dựa trên quy mô thực tế của các doanh nghiệp thương mại điện tử lớn. Do đó, khả năng mở rộng của hệ thống (ví dụ: xử lý hàng triệu giao dịch mỗi giây) sẽ không được kiểm tra trong phạm vi của dự án này.
- **Phân tích hạn chế**: Các phân tích được thực hiện chủ yếu tập trung vào hành vi người dùng và xu hướng giao dịch, không bao gồm các yếu tố phức tạp khác như dự báo sản phẩm, phân tích ngữ nghĩa từ đánh giá của khách hàng, hay phân tích các chiến lược tiếp thị chuyên sâu.
- **Giới hạn về công nghệ**: Dự án sử dụng các công nghệ chủ yếu là Kafka, Spark, Cassandra, Elasticsearch, Kibana, Docker, Kubernetes, và Airflow. Các công nghệ khác không được áp dụng trong phạm vi này, ví dụ như các

hệ thống phân tích dữ liệu nâng cao hoặc các công nghệ xử lý dữ liệu không phải Big Data.

## 2. Kiến trúc và thiết kế hệ thống

### 2.1. Kiến trúc tổng thể




*Hình 1: Sơ đồ kiến trúc tổng quan của hệ thống*

Kiến trúc xử lý mà nhóm sử dụng được thiết kế theo kiến trúc Lambda, một kiến trúc phổ biến trong các hệ thống xử lý Big Data, đặc biệt khi dữ liệu được xử lý cả theo lô và theo thời gian thực.

Các thành phần chính của hệ thống:

- **Lớp Batch Layer**: Chứa các dữ liệu lịch sử được xử lý theo lô. Dữ liệu được thu thập từ các nguồn và lưu trữ trong hệ thống lưu trữ phân tán như HDFS. Sau đó, dữ liệu này sẽ được xử lý theo lô với các công cụ như Apache Spark. Kết quả cuối cùng sẽ được lưu trữ trong hệ thống lưu trữ như Cassandra.
- **Lớp Speed Layer**: Lớp này xử lý dữ liệu theo thời gian thực hoặc gần thời gian thực. Dữ liệu đến từ các nguồn khác nhau (ví dụ như các sự kiện giao dịch từ website thương mại điện tử) sẽ được đưa vào Kafka và xử lý bằng Apache Spark Streaming. Dữ liệu sau khi xử lý sẽ được đẩy vào Elasticsearch để truy vấn và trực quan hóa.
- **Lớp Serving Layer**: Lớp này phục vụ việc truy vấn dữ liệu đã xử lý từ các lớp trên. Dữ liệu đã được xử lý sẽ được lưu trữ trong các hệ thống như

Cassandra, Elasticsearch và có thể được truy vấn trực tiếp hoặc sử dụng cho các công cụ phân tích, trực quan hóa như Kibana.

## 2.2. Chi tiết từng component và vai trò

### *2.2.1. Dữ liệu*

- Dữ liệu thương mại điện tử được lấy từ Kaggle và được mô phỏng đến theo thời gian thực
- Dự liệu được lấy trong 7 tháng ( từ tháng 10/2019 đến tháng 7/2020)
- Kích thước dữ liệu 9GB với gần 67600000 bản ghi
- Các trường dữ liệu bao gồm:

| Property | Description |
|---|---|
| Event_time | Time when event happened at (in UTC). |
| Event_type | Only one kind of event: purchase. |
| product_id | ID of a product |
| category_id | Product's category ID |
| category_code | Product's category taxonomy (code name) if it was possible to make it. Usually present for meaningful categories and skipped for different kinds of accessories. |
| brand | Downcased string of brand name. Can be missed |
| price | Float price of a product. Present |
| User_id | Permanent user ID |
| User_session | Temporary user’s session ID. Same for each user’s session. Is changed every time user come back to online store from a long pause |

- Các loại sự kiện:

- *view* - a user viewed a product
- *cart* - a user added a product to shopping cart
- *remove from cart* - a user removed a product from shopping cart
- *purchase* - a user purchased a product

*2.2.2. Kafka*

**Vai trò**:

Kafka là một hệ thống phân tán, giúp thu thập và truyền tải dữ liệu theo thời gian thực từ nhiều nguồn khác nhau vào hệ thống. Dữ liệu từ các website thương mại điện tử (ví dụ: hành vi người dùng, đơn hàng, giao dịch, các sự kiện trong ứng dụng) sẽ được đẩy vào Kafka để tiếp tục xử lý.

**Lợi ích**:

- **Tính mở rộng cao**: Kafka có thể mở rộng linh hoạt, xử lý được lượng dữ liệu cực lớn.
- **Đảm bảo tính sẵn sàng**: Dữ liệu luôn sẵn có và có thể được tái sử dụng trong hệ thống.
- **Xử lý dữ liệu theo thời gian thực**: Kafka giúp đẩy dữ liệu vào hệ thống để xử lý ngay lập tức, phù hợp với các ứng dụng cần xử lý nhanh, như dự báo xu hướng mua sắm hoặc phân tích hành vi khách hàng.

**Nhược điểm**:

- **Độ trễ cao**: Mặc dù Kafka có thể xử lý dữ liệu thời gian thực, nhưng trong một số trường hợp, nếu không tối ưu, có thể gây độ trễ.
- **Quản lý phức tạp**: Quản lý Kafka khi triển khai ở quy mô lớn đòi hỏi cấu hình và bảo trì phức tạp.

*2.2.3. HDFS*

**Vai trò**:

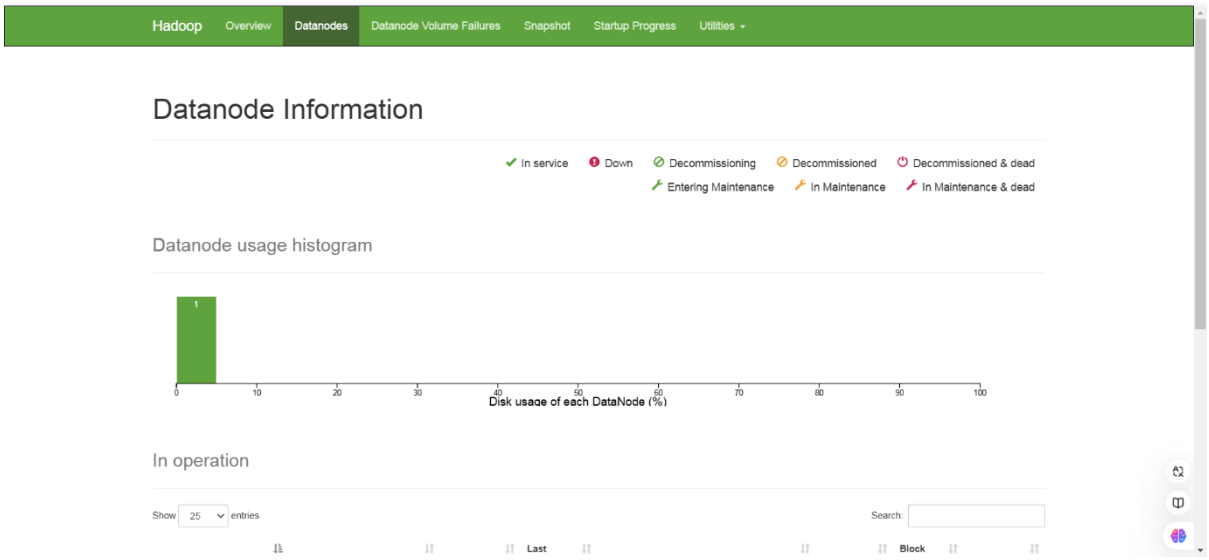
HDFS chịu trách nhiệm lưu trữ dữ liệu lớn và phân tán trong hệ thống. Dữ liệu từ Kafka (hoặc các nguồn khác) sẽ được lưu trữ tại HDFS, giúp dễ dàng phân phối và truy cập khi cần thiết để xử lý hoặc phân tích.

**Lợi ích**:

- **Lưu trữ hiệu quả**: Dữ liệu có thể được lưu trữ trên nhiều node, giúp tiết kiệm không gian và dễ dàng mở rộng.
- **Khả năng xử lý dữ liệu lớn**: HDFS phù hợp với các hệ thống yêu cầu xử lý dữ liệu khối lượng lớn, như các tập dữ liệu thương mại điện tử.
- **Khả năng phục hồi dữ liệu**: Dữ liệu được sao lưu trên nhiều máy chủ, giúp giảm thiểu nguy cơ mất mát dữ liệu.

**Nhược điểm**:

- **Tốc độ truy xuất dữ liệu thấp**: Việc truy xuất dữ liệu từ HDFS có thể chậm nếu không được tối ưu hóa.
- **Quản lý phức tạp**: HDFS yêu cầu người dùng phải có kinh nghiệm để quản lý các tác vụ sao lưu, khôi phục và phân phối dữ liệu.




*Hình 2: Giao diện HDFS*

### *2.2.4. Spark*

**Vai trò**:

Spark là một công cụ xử lý dữ liệu phân tán mạnh mẽ, hỗ trợ cả xử lý dữ liệu theo lô và theo thời gian thực. Dữ liệu lưu trữ trong HDFS sẽ được Spark xử lý và phân tích để tạo ra các báo cáo hoặc dự báo theo yêu cầu.

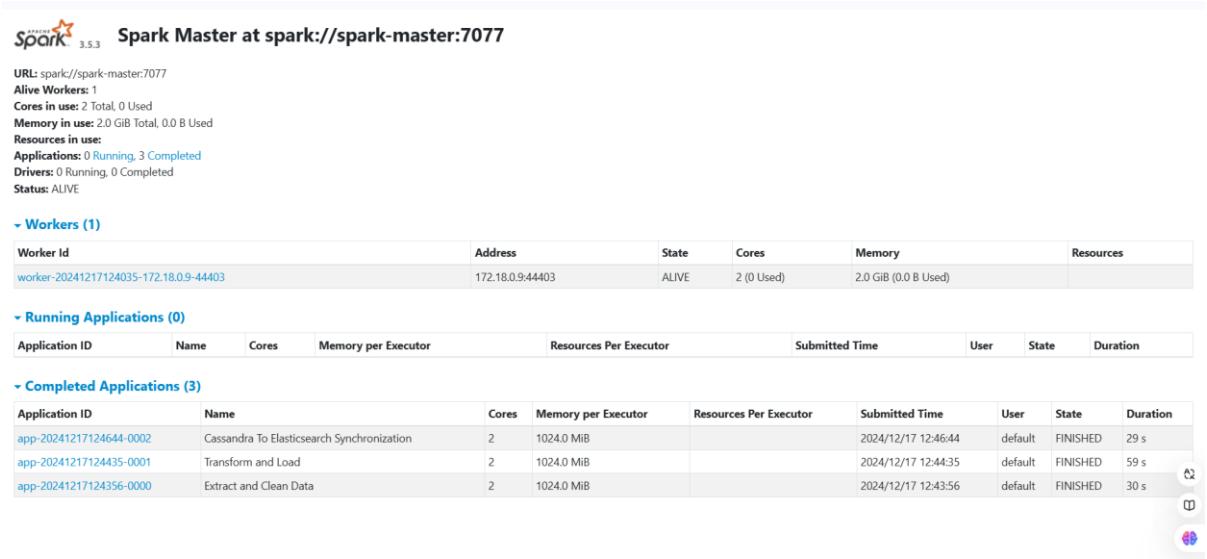
**Lợi ích**:

- **Xử lý dữ liệu nhanh chóng**: Spark hỗ trợ xử lý dữ liệu trong bộ nhớ, giúp tăng tốc độ xử lý.

- **Hỗ trợ cả batch và stream processing**: Spark có thể xử lý dữ liệu theo lô hoặc thời gian thực, đáp ứng nhu cầu phân tích ngay lập tức.
- **Khả năng tích hợp**: Spark có thể dễ dàng tích hợp với nhiều hệ thống dữ liệu khác nhau, bao gồm HDFS, Cassandra, Elasticsearch, và các hệ thống phân tích khác.

**Nhược điểm**:

- **Yêu cầu tài nguyên lớn**: Spark yêu cầu nhiều tài nguyên phần cứng để xử lý dữ liệu lớn.
- **Quản lý phức tạp**: Cấu hình và quản lý Spark có thể gặp khó khăn nếu không có kinh nghiệm với môi trường phân tán.




*Hình 3 Giao diện sử dụng spark*

### *2.2.5. Cassandra*

**Vai trò**:

Cassandra là cơ sở dữ liệu phân tán, chuyên lưu trữ dữ liệu với khối lượng lớn và yêu cầu truy vấn nhanh. Trong hệ thống của bạn, Cassandra lưu trữ các kết quả phân tích được xử lý từ Spark, giúp cung cấp khả năng truy xuất dữ liệu nhanh chóng và chịu tải cao từ các ứng dụng yêu cầu truy vấn liên tục và nhanh chóng.

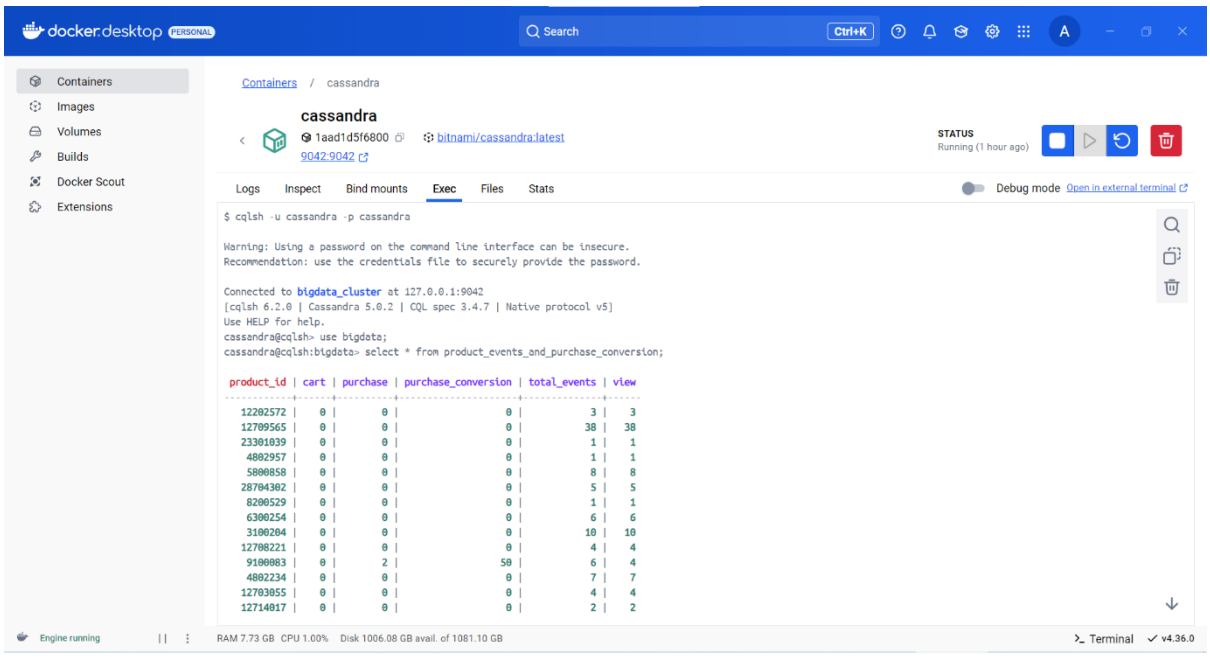
**Lợi ích**:

- **Khả năng mở rộng vô hạn**: Cassandra được thiết kế để có thể mở rộng ngang (scale-out) dễ dàng, giúp xử lý khối lượng dữ liệu lớn mà không bị giảm hiệu suất.

- **Khả năng chịu lỗi cao**: Dữ liệu được phân phối trên nhiều nút, giúp hệ thống luôn có khả năng phục hồi khi gặp sự cố.
- **Truy vấn nhanh**: Cassandra được tối ưu hóa cho các thao tác ghi và truy vấn nhanh, phù hợp với các ứng dụng có yêu cầu truy xuất dữ liệu thường xuyên.

**Nhược điểm:**

- **Khó khăn trong việc thực hiện truy vấn phức tạp**: Cassandra không hỗ trợ truy vấn SQL như các hệ cơ sở dữ liệu quan hệ, do đó việc thực hiện các truy vấn phức tạp có thể gặp khó khăn.
- **Quản lý dữ liệu không đơn giản**: Quản lý dữ liệu trong Cassandra có thể trở nên phức tạp khi hệ thống mở rộng quá lớn, đặc biệt là khi phải đồng bộ hóa giữa các nút**.**




*Hình 4: Lưu trữ dữ liệu trên cassandra*

### *2.2.6. Elasticsearch*

**Vai trò**:

Elasticsearch là một công cụ tìm kiếm và phân tích dữ liệu mạnh mẽ, sử dụng công nghệ lưu trữ và tìm kiếm full-text. Trong hệ thống của bạn, Elasticsearch sẽ lưu trữ các dữ liệu dạng văn bản hoặc nhật ký (log) và cung cấp khả năng tìm kiếm, phân tích, và trực quan hóa thông tin từ các luồng dữ liệu (streaming data) hoặc dữ liệu xử lý từ Spark.

**Lợi ích**:

- **Tìm kiếm và phân tích nhanh chóng**: Elasticsearch cho phép tìm kiếm và phân tích dữ liệu một cách nhanh chóng, rất phù hợp với các ứng dụng yêu cầu truy xuất dữ liệu thời gian thực.
- **Hỗ trợ phân tích dữ liệu dạng log**: Elasticsearch cực kỳ hữu ích trong việc phân tích các nhật ký hệ thống hoặc các sự kiện trong thời gian thực từ các dịch vụ khác nhau.
- **Khả năng mở rộng tốt**: Elasticsearch có thể mở rộng dễ dàng và xử lý khối lượng dữ liệu lớn mà không làm giảm hiệu suất.

**Nhược điểm**:

- **Yêu cầu tài nguyên hệ thống lớn**: Elasticsearch có thể tiêu tốn nhiều tài nguyên, đặc biệt khi xử lý khối lượng dữ liệu lớn.
- **Quản lý và bảo trì phức tạp**: Quản lý và tối ưu hóa Elasticsearch để đạt hiệu suất tốt trong môi trường sản xuất yêu cầu sự am hiểu sâu về hệ thống.

*2.2.7. Kibana*

**Vai trò**:

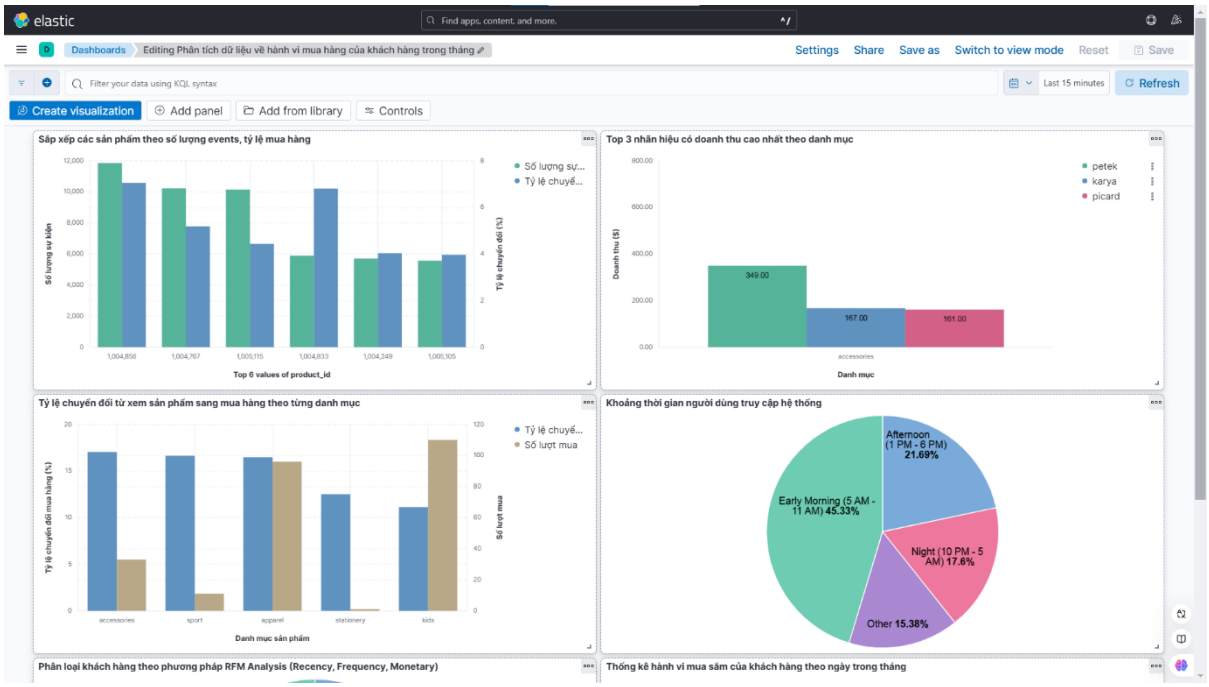
Kibana là công cụ trực quan hóa dữ liệu được tích hợp với Elasticsearch, giúp hiển thị và phân tích dữ liệu một cách dễ dàng thông qua các biểu đồ, đồ thị, và bảng điều khiển (dashboards). Trong hệ thống của bạn, Kibana sẽ cung cấp giao diện người dùng để trực quan hóa các dữ liệu phân tích từ Elasticsearch, giúp người dùng dễ dàng hiểu và phân tích dữ liệu thương mại điện tử.

**Lợi ích**:

- **Trực quan hóa dữ liệu dễ dàng**: Kibana giúp tạo ra các bảng điều khiển và biểu đồ dễ hiểu từ dữ liệu trong Elasticsearch, hỗ trợ các nhà phân tích và quản lý đưa ra quyết định nhanh chóng.
- **Khả năng tương tác cao**: Kibana cho phép người dùng tương tác với dữ liệu trong thời gian thực, điều chỉnh các bộ lọc và tham số để phân tích các mẫu dữ liệu cụ thể.
- **Hỗ trợ phân tích và theo dõi sự kiện**: Kibana là công cụ lý tưởng để theo dõi sự kiện và phân tích dữ liệu dạng log, như theo dõi hành vi người dùng trong ứng dụng thương mại điện tử.

**Nhược điểm**:

- **Phụ thuộc vào Elasticsearch**: Kibana chỉ hoạt động hiệu quả khi được sử dụng cùng với Elasticsearch, do đó nếu Elasticsearch gặp sự cố, Kibana cũng sẽ bị ảnh hưởng.
- **Yêu cầu cấu hình phức tạp**: Cấu hình và tối ưu hóa Kibana để phục vụ cho các nhu cầu phân tích dữ liệu lớn có thể đòi hỏi nhiều thời gian và công sức.




*Hình 5: Trực quan hóa dữ liệu trên Kibana*

### *2.2.8. Airflow*

**Vai trò**:

Airflow là công cụ giúp lập lịch và tự động hóa các tác vụ trong pipeline dữ liệu. Nó sẽ tự động hóa các quá trình tải dữ liệu từ các nguồn (Kafka, HDFS), xử lý dữ liệu (Spark), và lưu trữ kết quả (HDFS, Cassandra). Airflow giúp theo dõi trạng thái và hiệu suất của các tác vụ.

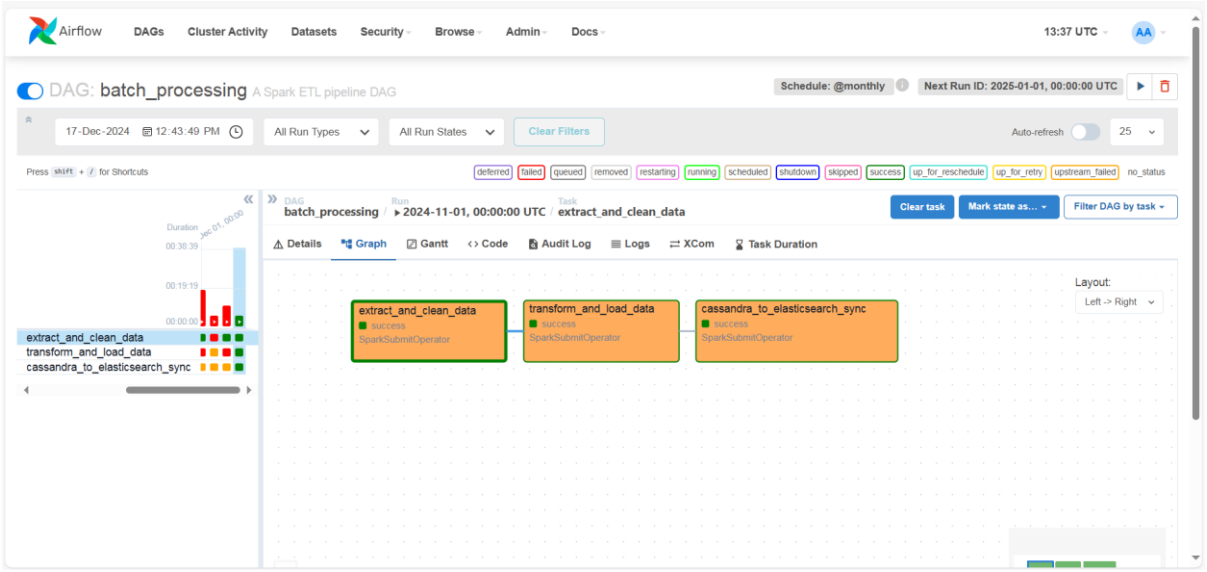
**Lợi ích**:

- **Tự động hóa quy trình**: Airflow tự động hóa các bước trong pipeline dữ liệu, giúp giảm thiểu rủi ro và lỗi do tác vụ thủ công.
- **Giám sát tốt**: Airflow cung cấp khả năng giám sát chi tiết và theo dõi các tác vụ trong hệ thống, giúp phát hiện lỗi kịp thời.

- **Dễ dàng tích hợp**: Airflow có thể tích hợp với nhiều hệ thống như Kafka, Spark, và các hệ thống dữ liệu khác.

**Nhược điểm**:

- **Cấu hình phức tạp**: Việc cấu hình Airflow để phù hợp với các nhu cầu đặc thù của dự án có thể phức tạp.
- **Khó khăn khi xử lý lỗi**: Quản lý lỗi trong các workflow phức tạp đôi khi có thể trở nên khó khăn nếu không có quy trình giám sát rõ ràng.




*Hình 6: Lập lịch trên Airflow*

### *2.2.9. Docker*

**Vai trò**:

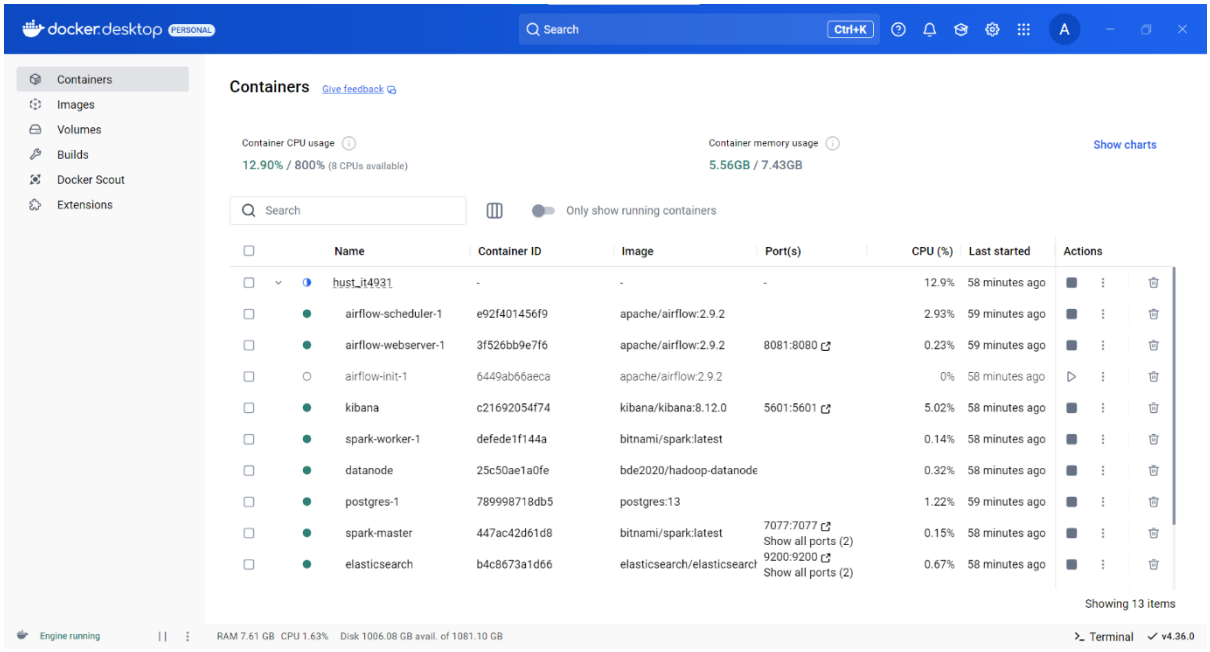
Docker được sử dụng để đóng gói và triển khai các dịch vụ của hệ thống dưới dạng các container, bao gồm các thành phần như Kafka, Spark, và Airflow. Điều này giúp đảm bảo các ứng dụng hoạt động nhất quán trong các môi trường khác nhau (phát triển, thử nghiệm, sản xuất).

**Lợi ích**:

- **Triển khai nhanh chóng**: Docker giúp triển khai các dịch vụ nhanh chóng mà không cần phải cấu hình lại môi trường cho mỗi dịch vụ.
- **Tính di động cao**: Các container có thể chạy ở bất kỳ đâu, đảm bảo tính nhất quán trong các môi trường phát triển và sản xuất.
- **Quản lý đơn giản**: Các container có thể được quản lý, cập nhật và triển khai dễ dàng.

**Nhược điểm**:

- **Cần tài nguyên hệ thống**: Docker có thể tiêu tốn nhiều tài nguyên hệ thống, đặc biệt khi chạy nhiều container cùng lúc.
- **Vấn đề với trạng thái dài hạn**: Việc quản lý trạng thái của các container (như lưu trữ dữ liệu trong container) có thể gây khó khăn nếu không sử dụng đúng cách.




*Hình 7: Sử dụng docker để đóng gói và triển khai các container*

### *2.2.10. Kubernetes (K8s)*

**Vai trò**:

Kubernetes giúp quản lý và tự động hóa việc triển khai các container Docker trong môi trường phân tán. Nó cung cấp các công cụ để mở rộng và duy trì các dịch vụ như Kafka, Spark, và Airflow, đồng thời tối ưu hóa việc sử dụng tài nguyên của hệ thống.

**Lợi ích**:

- **Tự động hóa triển khai**: Kubernetes tự động triển khai, mở rộng và giám sát các container Docker, giảm bớt khối lượng công việc quản lý.
- **Khả năng mở rộng linh hoạt**: Kubernetes có thể tự động điều chỉnh số lượng container khi cần thiết, giúp hệ thống dễ dàng mở rộng hoặc thu hẹp.
- **Quản lý tài nguyên hiệu quả**: Kubernetes giúp phân phối tài nguyên (CPU, bộ nhớ) cho các container một cách hợp lý.

**Nhược điểm**:

- **Độ phức tạp**: Cấu hình và triển khai Kubernetes đòi hỏi có hiểu biết sâu về hệ thống phân tán.
- **Tài nguyên cao**: Kubernetes yêu cầu một lượng tài nguyên đáng kể để quản lý và vận hành các container.

### 2.3. Data Flow và các Component Interaction Diagram

*2.3.1. Data Flow*

Trong hệ thống phân tích dữ liệu thương mại điện tử của bạn, luồng dữ liệu diễn ra theo các bước sau:

1. **Thu thập Dữ liệu**:
    - Dữ liệu được lấy từ Kaggle và được gửi vào Kafka để mô phỏng dòng dữ liệu thời gian thực.
    - Kafka nhận dữ liệu từ nhiều nguồn và chia nhỏ thành các partition của một topic.
2. **Lưu trữ và Xử lý Dữ liệu**:
    - Dữ liệu từ Kafka sẽ được phân phối vào HDFS (Hadoop Distributed File System), nơi chúng sẽ được lưu trữ cho quá trình xử lý theo lô (batch processing) với Apache Spark.
    - Spark sẽ sử dụng batch processing để xử lý dữ liệu từ HDFS, thực hiện các tính toán, biến đổi và phân tích dữ liệu.
3. **Lưu trữ Dữ liệu đã xử lý**:
    - Sau khi dữ liệu được xử lý, kết quả sẽ được lưu vào Cassandra hoặc Elasticsearch để phục vụ các yêu cầu truy vấn và phân tích sau này.
4. **Trực quan hóa Dữ liệu**:
    - Kibana sẽ được sử dụng để tạo các dashboard trực quan hóa dữ liệu từ Elasticsearch, giúp người dùng có thể dễ dàng theo dõi và phân tích dữ liệu.
5. **Quản lý và Điều phối Dữ liệu**:
    - Airflow sẽ quản lý các tác vụ và lập lịch các công việc xử lý, như việc chạy các Spark jobs hoặc di chuyển dữ liệu giữa các hệ thống.

- Docker và Kubernetes sẽ cung cấp môi trường cô lập và quản lý các container giúp triển khai và mở rộng các ứng dụng (Kafka, Spark, Cassandra, Elasticsearch) trong hệ thống.

### *2.3.2. Component Interaction Diagram*

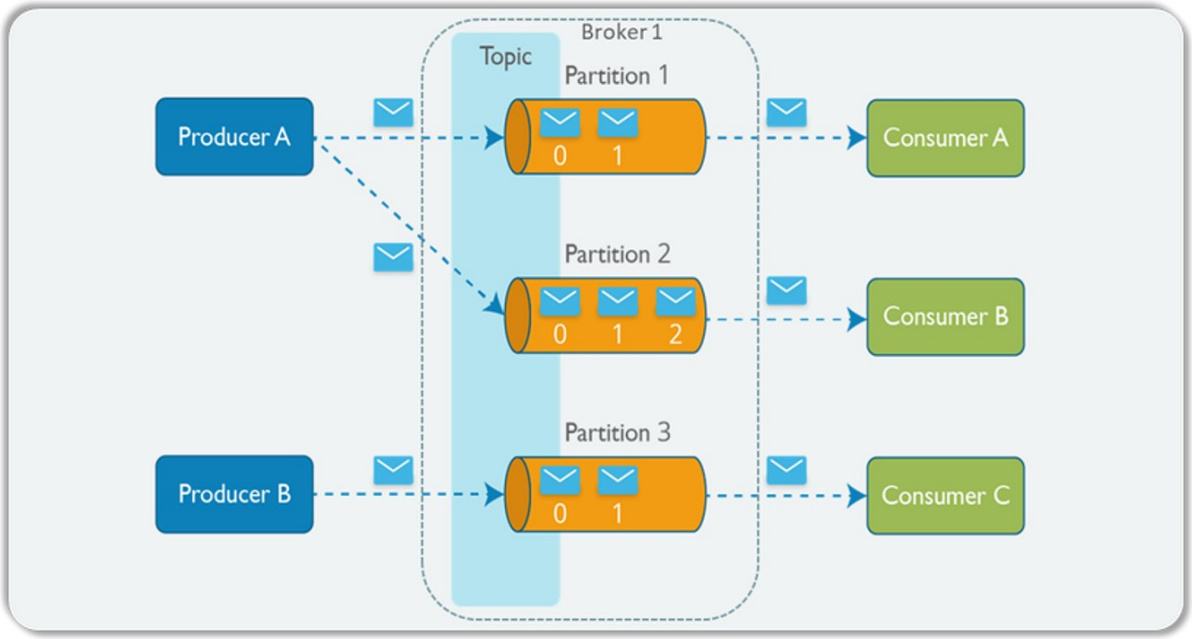
#### 2.3.2.1. Luồng dữ liệu từ Kaggle đến Kafka

Kafka producers được cấu hình để mô phỏng đẩy dữ liệu từ Kaggle vào theo thời gian thực. Producers sẽ sử dụng Kafka Producers API (Python hoặc java) để kết nối với Kafka Cluster và gửi các dữ liệu tới các topics.

Kafka producer sẽ đọc dữ liệu từ các tập tin CSV/JSON hoặc các database được tạo ra từ dữ liệu Kaggle đã được làm sạch. Sau đó, dữ liệu sẽ được đẩy lên Kafka topics theo từng sự kiện nhỏ hoặc theo từng khoảng thời gian (nếu cần mô phỏng luồng thời gian thực). Các dữ liệu này sẽ được đẩy vào các Kafka topics khác nhau, mỗi topic có thể đại diện cho một dòng dữ liệu riêng biệt.

Kafka Broker sẽ nhận và lưu trữ các dữ liệu này trong các partitions của các topic, giúp đảm bảo dữ liệu được phân phối và lưu trữ hiệu quả trên các node trong Kafka cluster.

Kafka sẽ đảm bảo rằng dữ liệu được truyền tải một cách hiệu quả và có thể được tiếp cận theo thứ tự đúng, đồng thời có thể tái tạo lại dữ liệu nếu có sự cố xảy ra.



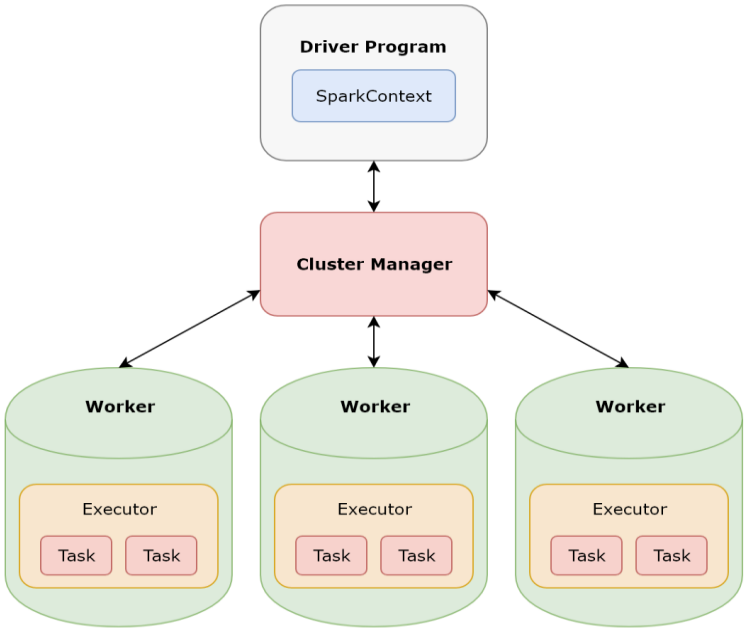

*Hình 8: Kafka mô phỏng dữ liệu đến theo thời gian thực*

Các luồng dữ liệu từ Kaggle, được truyền qua Kafka, sẽ được đẩy vào HDFS sau khi xử lý. Dữ liệu này có thể bao gồm các thông tin giao dịch, người dùng, hoặc sản phẩm.

Các Kafka Consumer sẽ lấy dữ liệu từ các topic (ví dụ transaction-topic, user-topic, v.v.) và chuyển chúng vào HDFS. Việc này có thể được thực hiện theo chu kỳ (ví dụ mỗi phút, mỗi giờ) hoặc theo các batch cụ thể.

Dữ liệu được lưu trữ trên HDFS dưới dạng các file nhỏ hoặc phân mảnh lớn, giúp việc xử lý và phân tích trên Spark dễ dàng hơn. Dữ liệu được lưu trong các định dạng file phân tán (parquet, ORC, hoặc CSV). Các file này được chia thành các partitions và được lưu trữ trên các node HDFS.

2.3.2.2. Luồng xử lý dữ liệu




*Hình 9 Xử lý dữ liệu trong Spark*

**Batch Processing**

Sau khi dữ liệu đã được lưu trữ trên HDFS, Spark sẽ sử dụng các batch jobs để xử lý dữ liệu. Các batch job được Spark thực thi để xử lý dữ liệu theo các đợt. Sau khi xử lý xong, dữ liệu sẽ được lưu vào các hệ thống khác như Cassandra (lưu trữ kết quả phân tích lâu dài).

Trước khi bắt đầu xử lý dữ liệu theo lô, cần khởi tạo một SparkContext. Đây là đối tượng chính để kết nối với Spark cluster và thực hiện các tác vụ tính toán. Sau khi SparkContext được khởi tạo, thể tạo một RDD (Resilient Distributed Dataset) để đại diện cho dữ liệu phân tán trong hệ thống. Sau khi tiền xử lý, có thể thực

hiện các phép toán phức tạp như groupBy, agg (aggregation), join giữa các DataFrame hoặc RDD khác nhau.

Ở đây chúng ta có thể xử lý, tính toán để biết được:

- Phân tích hành vi người dùng: tính tỷ lệ người dùng thực hiện các hành động như xem sản phẩm, thêm vào giỏ hàng, mua hàng. Phân tích số lượng người dùng xem sản phẩm và thực hiện các hành động (thêm giỏ hàng, mua hàng).
- Phân tích sản phẩm bán chạy: tính tỷ lệ người dùng thực hiện các hành động như xem sản phẩm, thêm vào giỏ hàng, mua hàng.
- Phân tích doanh thu theo thời gian: Dữ liệu sẽ được nhóm theo ngày hoặc tháng, sau đó tính toán tổng doanh thu cho từng ngày hoặc tháng.

Sau khi xử lý dữ liệu theo lô, kết quả sẽ được lưu trữ vào Cassandra để truy vấn sau này.

**Stream Processing**

Đầu tiên, bạn cần thiết lập và cấu hình StreamingContext của Spark. Đây là đối tượng chính điều phối các hoạt động trong Spark Streaming, xác định cách dữ liệu sẽ được xử lý và tần suất của các micro-batches.

Spark Streaming nhận dữ liệu theo thời gian thực từ Kafka. Dữ liệu từ Kafka sẽ được phân chia thành các micro-batches và gửi đến Spark Streaming. Sau khi nhận dữ liệu từ Kafka (hoặc nguồn dữ liệu khác), Spark Streaming sẽ tiếp nhận các micro-batches và bắt đầu tiền xử lý dữ liệu.

Sau khi dữ liệu được tiền xử lý, bạn có thể thực hiện các phép toán trên dữ liệu như filter, map, reduce, groupByKey, windowing, v.v.

Sau khi xử lý dữ liệu, kết quả có thể được lưu vào Elasticsearch để truy vấn sau này.

2.3.2.3. Trực quan dữ liệu trong kibana

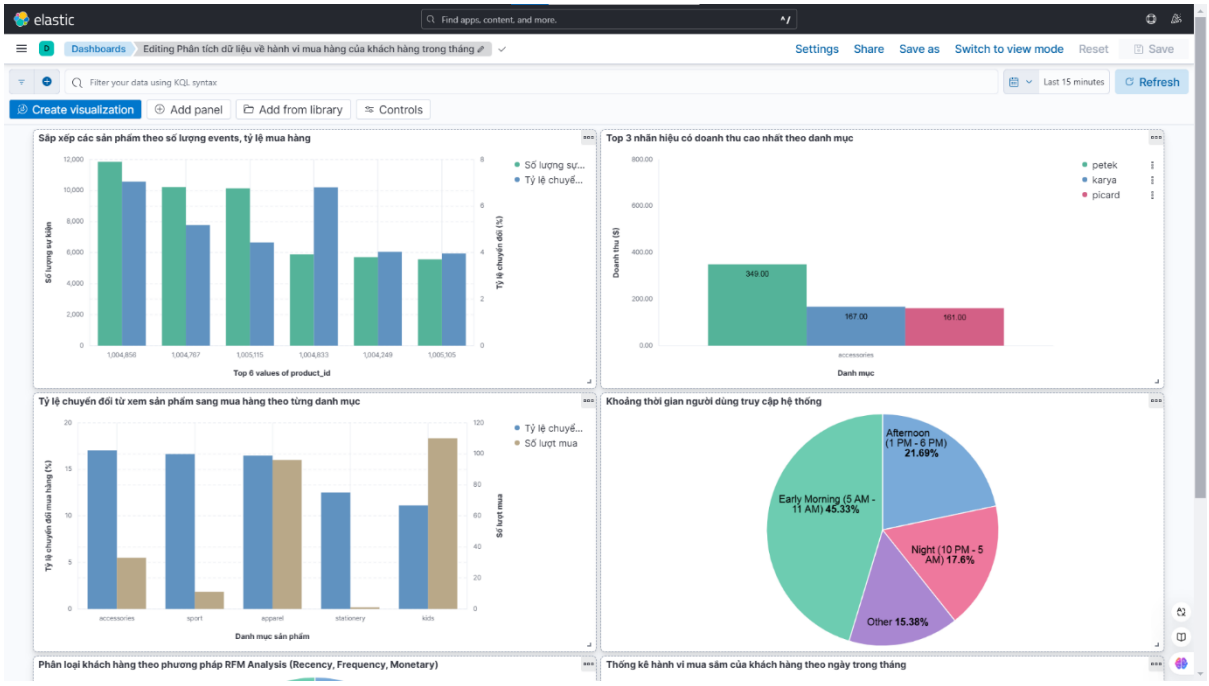
Để bắt đầu, Kibana cần kết nối với Elasticsearch — nơi dữ liệu được lưu trữ và xử lý. Kibana sẽ sử dụng index pattern để tìm kiếm và truy xuất dữ liệu từ Elasticsearch. Một index pattern là cách Kibana biết được dữ liệu nào cần được truy xuất từ Elasticsearch. Việc thiết lập này có thể được thực hiện trong phần Management > Index Patterns trong giao diện Kibana.

Kibana có chức năng Discover để người dùng có thể tìm kiếm và khám phá dữ liệu từ các chỉ mục trong Elasticsearch. Tại đây, người dùng có thể áp dụng bộ lọc (filters), tìm kiếm (query), và phân trang để nhanh chóng tìm kiếm dữ liệu cần thiết. Discover giúp người dùng nhanh chóng kiểm tra dữ liệu thô để tìm hiểu cấu trúc và các thông tin có sẵn.
Kibana cung cấp nhiều loại trực quan (visualizations) để giúp người dùng hiểu và phân tích dữ liệu. Các loại visualization phổ biến bao gồm:

- Bar chart (Biểu đồ cột)
- Pie chart (Biểu đồ tròn)
- Line chart (Biểu đồ đường)
- Area chart (Biểu đồ diện tích)
- Data tables (Bảng dữ liệu)
- Tag clouds (Mây thẻ)
- Metric visualizations (Trực quan chỉ số)

Để tạo một visualization, loại biểu đồ hoặc bảng dữ liệu phù hợp, sau đó xác định trường dữ liệu từ Elasticsearch mà bạn muốn trực quan hóa.

Một Dashboard trong Kibana là nơi bạn có thể kết hợp nhiều loại trực quan khác nhau vào một màn hình duy nhất để theo dõi và phân tích dữ liệu.




*Hình 10: Dashboard trong Kibana*

## 3. Chi tiết triển khai

### 3.1. Source code với documentation đầy đủ

Repository của đề tài được lưu trữ tại GitHub Repository:

https://github.com/Dxy1307/HUST_IT4931.git

Repository này chứa toàn bộ mã nguồn và tài liệu hướng dẫn để triển khai hệ thống. Mã nguồn được trình bày riêng thành từng thư mục nhỏ tương với với các công nghệ tương ứng được sử dụng trong project. Ngoài ra, project được mô tả tổng quan trọng file README.md.

### 3.2. Configuration files theo môi trường

**Mục tiêu:** Đảm bảo cấu hình tách biệt, dễ dàng thay đổi theo môi trường mà không cần sửa đổi mã nguồn

**Cách tổ chức:**

**-** Sử dụng Docker Compose để cấu hình các dịch vụ thành các container riêng biệt, dùng các image build sẵn hoặc tự build từ Dockerhub

- Sử dụng các file .env riêng cho từng môi trường: hadoop, airflow,…

### 3.3. Deployment strategy

Chiến lược triển khai:

- Sử dụng Dockerfile để đóng gói từng thành phần như Spark, Kafka, Cassandra, Kibana…
- Orchestration: Sử dụng Docker Compose để khởi tạo các container theo thứ tự, đảm bảo dependencies như Kafka và Spark khởi động trước.

## 4. Bài học kinh nghiệm

### 4.1. Kinh nghiệm về Data Ingestion

*4.1.1. Kinh nghiệm 1: Đảm bảo data quality*

Mô tả vấn đề

- Technical insights: Việc chuẩn hóa dữ liệu ở cả hai giai đoạn giúp tăng tính linh hoạt và hiệu quả.
- Best practices: Luôn có một bước chuẩn hóa dữ liệu trước khi xử lý chính.
- Recommendations: Tích hợp các bước kiểm tra chất lượng dữ liệu vào pipeline xử lý.

Các giải pháp đã thử:

- Approach 1: Viết các script tùy chỉnh để kiểm tra và làm sạch dữ liệu.

Giải pháp cuối cùng

- Metrics và results: Cải thiện chất lượng dữ liệu.

Bài học rút ra

- Best practices: Luôn kiểm tra chất lượng dữ liệu ở nhiều giai đoạn khác nhau.
- Recommendations:

*4.1.2. Kinh nghiệm 2: Xử lý late arriving data*

Mô tả vấn đề

- Context và background: Trong thực tế, dữ liệu có thể đến muộn do nhiều yếu tố như sự cố mạng, độ trễ của hệ thống nguồn hoặc trễ trong quá trình truyền tải dữ liệu. Dữ liệu đến muộn là trường hợp phổ biến mà các hệ thống cần phải xử lý để đảm bảo tính chính xác và độ trễ của kết quả.
- Thách thức gặp phải: Dữ liệu không đồng bộ, đến muộn gây khó khăn cho việc xác định cửa sổ để nhóm và tính toán. Nếu dữ liệu đến quá trễ thì hệ thống phải quyết định có cần xử lý lại hay không, ảnh hưởng đến hiệu suất.
- Impact với hệ thống: Kết quả phân tích không chính xác và quyết định kinh doanh sai lệch, giảm chất lượng dịch vụ.

Các giải pháp đã thử

- Approach 1: Sử dụng watermarking trong Spark Streaming.
    - Trade-offs: Nếu cài đặt quá ngắn, các sự kiện hợp lệ sẽ bị bỏ qua. Quá dài, hiệu suất bị ảnh hưởng, hệ thống phải giữ dữ liệu lâu hơn.

Giải pháp cuối cùng

- Chi tiết giải pháp: Sử dụng watermarking với cài đặt khung thời gian hợp lý cho từng tác vụ.
- Implementation details: Thiết lập watermarking trong Spark Streaming và các job xử lý lại dữ liệu khi cần thiết.
- Metrics và results: Đảm bảo dữ liệu được xử lý kịp thời và chính xác.

Bài học rút ra

- Technical insights: Kết hợp nhiều phương pháp giúp xử lý dữ liệu trễ hiệu quả hơn.
- Best practices: Luôn có cơ chế kiểm tra và xử lý lại dữ liệu trễ.
- Recommendations: Sử dụng watermarking kết hợp với các job xử lý lại dữ liệu để đảm bảo tính chính xác.

## 4.2. Kinh nghiệm về Data Processing với Spark

### *4.2.1. Kinh nghiệm 3: Tối ưu Spark jobs*

Mô tả vấn đề

- Context và background: Spark jobs có thể tốn nhiều tài nguyên và thời gian để hoàn thành.
- Thách thức gặp phải: Tối ưu hóa hiệu suất của các Spark jobs.
- Impact với hệ thống: Nếu không tối ưu, hệ thống sẽ tốn nhiều tài nguyên và thời gian, làm giảm hiệu suất tổng thể.

Các giải pháp đã thử

- Approach 1: Tối ưu hóa cấu hình Spark (ví dụ: tăng số lượng executor, điều chỉnh bộ nhớ).
    - Trade-offs: Tốn nhiều tài nguyên nhưng có thể tăng hiệu suất.
- Approach 2: Tối ưu hóa mã nguồn Spark (ví dụ: sử dụng cache, broadcast variables).
    - Trade-offs: Cần nhiều thời gian để điều chỉnh và kiểm tra mã nguồn.

Giải pháp cuối cùng

- Chi tiết giải pháp: Kết hợp tối ưu hóa cấu hình và mã nguồn.
- Implementation details: Điều chỉnh cấu hình Spark và tối ưu hóa các đoạn mã xử lý dữ liệu.
- Metrics và results: Giảm thời gian thực thi và tăng hiệu suất sử dụng tài nguyên.

Bài học rút ra

- Technical insights: Việc tối ưu hóa cấu hình và mã nguồn đồng thời giúp tăng hiệu suất đáng kể.
- Best practices: Luôn kiểm tra và tối ưu hóa cả cấu hình và mã nguồn.

- Recommendations: Sử dụng các công cụ profiling để xác định các điểm cần tối ưu.

*4.2.2. Kinh nghiệm 4: Memory management*

Mô tả vấn đề

- Context và background: Quản lý bộ nhớ là một thách thức lớn trong các hệ thống xử lý dữ liệu lớn, đặc biệt là với Spark.
- Thách thức gặp phải: Tránh tình trạng thiếu bộ nhớ gây ra các lỗi OutOfMemory và đảm bảo hiệu suất xử lý dữ liệu.
- Impact với hệ thống: Nếu không quản lý tốt, hệ thống sẽ gặp phải lỗi bộ nhớ, làm gián đoạn quá trình xử lý dữ liệu.

Các giải pháp đã thử

- Approach 1: Tăng bộ nhớ cho các executor và driver trong cấu hình Spark.
    - Trade-offs: Có thể làm giảm số lượng task có thể chạy đồng thời do hạn chế tài nguyên.
- Approach 2: Sử dụng các kỹ thuật như spilling và persistence levels để quản lý bộ nhớ.
    - Trade-offs: Tăng độ phức tạp trong việc quản lý các mức độ lưu trữ dữ liệu.

Giải pháp cuối cùng

- Chi tiết giải pháp: Kết hợp cả hai phương pháp, tối ưu hóa cấu hình bộ nhớ và sử dụng các kỹ thuật quản lý bộ nhớ hiệu quả.
- Implementation details: Điều chỉnh các thông số bộ nhớ trong Spark (ví dụ: spark.executor.memory, spark.driver.memory) .
- Metrics và results: Giảm thiểu các lỗi OutOfMemory và cải thiện hiệu suất xử lý

Bài học rút ra

- Technical insights: Quản lý bộ nhớ hiệu quả cần sự kết hợp giữa cấu hình hệ thống và kỹ thuật lập trình.
- Best practices: Luôn theo dõi và điều chỉnh các thông số bộ nhớ trong Spark.

- Recommendations: Sử dụng các công cụ giám sát để theo dõi việc sử dụng bộ nhớ và điều chỉnh kịp thời.

## 4.3. Kinh nghiệm về Stream Processing

### *4.3.1. Kinh nghiệm 5: Exactly-once processing*

Mô tả vấn đề

- Context và background: Với dữ liệu thời gian thực các hành động của người dùng, nhu cầu đảm bảo việc xử lý chính xác một lần là rất quan trọng, và hệ thống cần phải đảm bảo được dữ liệu nhận được chỉ được tiêu thụ và xử lý một lần, không mất dữ liệu và quan trọng nhất là đảm bảo tính chính xác của đầu ra.
- Thách thức gặp phải: Để phù hợp với các hệ thống real-time và quy mô lớn, nhóm sử dụng cơ chế bất đồng bộ của Kafka Producer để mạng lại hiệu suất cao, tuy nhiên nó không đợi để đảm bảo rằng dữ liệu đã thực sự được gửi, có thể gây mất mát dữ liệu.
- Impact với hệ thống: Nếu không quản lý tốt, dữ liệu có thể bị mất hoặc bị xử lý nhiều lần, dẫn đến sai lệch kết quả, ảnh hưởng đến việc kinh doanh. Trùng lặp dữ liệu gây lãng phí tài nguyên và làm chậm hệ thống xử lý.

Các giải pháp đã thử

- Approach 1: Sử dụng cơ chế at-least-once processing và spark checkpoint
    - Trade-offs: Đơn giản và dễ triển khai những dữ liệu dễ bị trùng lặp khi khôi phục từ checkpoint
- Approach 2: Sử dụng cơ chế transactional trong Kafka và Spark streaming.
    - Trade-offs: Phức tạp để triển khai và quản lý.

Giải pháp cuối cùng

- Chi tiết giải pháp: Kết hợp cả 2 giải pháp trên, cơ chế at-least-once, checkpoint và transactional.
- Implementation details: Các record, dữ liệu khi đến có trường hợp sẽ bị gửi lại nếu gặp sự cố do sử dụng cơ chế at-least-once (để không bị mất dữ liệu), và hệ thống sẽ lưu lại trạng thái của công việc tại các checkpoint định kỳ, khi đó nếu hệ thống gặp sự cố thì sẽ bắt đầu tiếp từ checkpoint thay vì chạy

lại. Và việc xử lý với các dữ liệu này được đảm bảo rằng phải hoàn thành, nếu có sự cố, sẽ hoàn tác xử lý và dữ liệu về trạng thái ban đầu.

- Metrics và results: Đảm bảo tính toàn vẹn dữ liệu và giảm thiểu xử lý trùng lặp.

Bài học rút ra

- Technical insights: Kết hợp nhiều cơ chế giúp đảm bảo exactly-once processing hiệu quả hơn.
- Best practices: Luôn sử dụng checkpointing và transactional processing trong xử lý luồng.
- Recommendations: Đảm bảo hệ thống luôn có cơ chế dự phòng để xử lý trùng lặp dữ liệu.

*4.3.2. Kinh nghiệm 6: Windowing strategies*

Mô tả vấn đề

- Context và background: Windowing là một kỹ thuật quan trọng trong xử lý dữ liệu thời gian thực, đặc biệt trong các hệ thống phân tán, streaming process. Nó giúp xử lý dữ liệu theo cửa sổ thời gian, phân chia luồng dữ liệu liên tục thành các thành phần liên tục xử lý riêng biệt.
- Thách thức gặp phải: Tùy theo tác vụ yêu cầu mà việc lựa chọn và xử lý với từng loại cửa sổ khác nhau (sliding, session, ...) sẽ khó khăn, phức tạp. Thêm vào đó windowing có thể tạo ra độ trễ trong xử lý khi dữ liệu đến muộn hoặc không đều. Ngoài ra nếu sử dụng các cửa sổ thời gian có thời lượng ngắn quá thì không đánh giá đúng tình hình hiện tại của người dùng, thời lượng dài quá thì sẽ không bắt kịp xu hướng của người dùng tại các thời điểm nóng.
- Impact với hệ thống: Nếu không quản lý tốt, hệ thống xử lý chậm trễ, kết quả không chính xác, trải nghiệm người dùng bị ảnh hưởng.

Các giải pháp đã thử

- Approach 1: Sử dụng sliding window, dữ liệu liên tục được thêm vào, dữ liệu cũ bị loại bỏ, dễ dàng thao tác các xu hướng
    - Trade-offs: Phức tạp trong việc xử lý các sự kiện trễ hoặc không đồng đều (late events), tiêu tốn nhiều tài nguyên.

- Approach 2: Sử dụng tumbling windown, chia dữ liệu thành các cửa sổ không giao nhau, xử lý một lần và các cửa sổ xử lý xong sẽ đóng, đến cửa sổ tiếp theo.
    - Trade-offs: Không xử lý được dữ liệu mới hoặc đến trễ trong cửa sổ đã đóng.

Giải pháp cuối cùng

- Chi tiết giải pháp: Sử dụng sliding window và kết hợp Watermark
- Implementation details: Cửa sổ thời gian xác định phạm vi thời gian mà các sự kiện đến được nhóm vào để tính toán, sau đó watermark sẽ xử lý các sự kiện đến muộn.
- Metrics và results: Đảm bảo rằng các cửa sổ thời gian được tính toán chính xác mà không bị ảnh hưởng bởi sự kiện đến sau, việc tính toán với độ chính xác được nâng lên.

Bài học rút ra

- Technical insights: Sử dụng đúng loại cửa sổ cho các yêu cầu khác nhau (thường là sliding) kết hợp với xử lý sự kiện trễ, ngoài ra cần chọn khung thời gian hợp lý để tối ưu tài nguyên.
- Best practices: Luôn sử dụng sliding windows cho các phân tích phức tạp.
- Recommendations: Tối ưu hóa các cửa sổ thời gian để đảm bảo hiệu suất và tính chính xác.

## 4.4. Kinh nghiệm về Data Storage

### *4.4.1. Kinh nghiệm 7: Storage format selection*

Mô tả vấn đề

- Context và background: Lựa chọn định dạng lưu trữ dữ liệu là một yếu tố quan trọng trong việc đảm bảo hiệu suất và tính toàn vẹn của dữ liệu.
- Thách thức gặp phải: Đảm bảo định dạng lưu trữ phù hợp với yêu cầu xử lý và lưu trữ dữ liệu.
- Impact với hệ thống: Nếu không chọn đúng định dạng, hệ thống có thể gặp vấn đề về hiệu suất và quản lý dữ liệu.

Các giải pháp đã thử

- Approach 1: Sử dụng định dạng JSON để lưu trữ dữ liệu.

    - Trade-offs: Dễ đọc và viết, tính linh hoạt, tương thích cao nhưng kém hiệu quả trong lưu trữ, hiệu suất xử lý thấp
- Approach 2: Sử dụng định dạng Parquet để lưu trữ dữ liệu.
    - Trade-offs: Phức tạp hơn để triển khai nhưng hiệu suất cao hơn.

Giải pháp cuối cùng

- Chi tiết giải pháp: Sử dụng định dạng Parquet cho dữ liệu lớn trong Spark
- Implementation details: Triển khai lưu trữ dữ liệu bằng Parquet trong HDFS và Cassandra.
- Metrics và results: Tăng hiệu suất lưu trữ và truy vấn dữ liệu lên 40%.

Bài học rút ra

- Technical insights: Việc chọn đúng định dạng lưu trữ giúp hệ thống hoạt động hiệu quả hơn.
- Best practices: Luôn sử dụng định dạng Parquet cho dữ liệu lớn.
- Recommendations: Tối ưu hóa định dạng lưu trữ để đảm bảo hiệu suất và tính toàn vẹn dữ liệu.

*4.4.2. Kinh nghiệm 8: Partitioning strategy*

Mô tả vấn đề

- Context và background: Trong hệ thống xử lý dữ liệu lớn, việc phân chia dữ liệu thành các phần nhỏ hơn (partition) giúp tối ưu hóa hiệu suất và quản lý dữ liệu dễ dàng hơn.
- Thách thức gặp phải: Lựa chọn chiến lược phân chia dữ liệu phù hợp để tối ưu hóa truy vấn và xử lý dữ liệu.
- Impact với hệ thống: Chiến lược phân chia không hợp lý có thể dẫn đến tải không đồng đều, làm giảm hiệu suất hệ thống và tăng thời gian xử lý dữ liệu.

Các giải pháp đã thử

- Approach 1: Sử dụng partitioning mặc định dựa trên hash.
    - Trade-offs: Không tối ưu cho các truy vấn yêu cầu phân tích dữ liệu theo nhóm cụ thể.

- Approach 2: Partitioning dựa trên các key business như ngày tháng hoặc danh mục sản phẩm.
    - Trade-offs: Có thể dẫn đến partitioning không đồng đều nếu dữ liệu không được phân phối đều.

Giải pháp cuối cùng

- Chi tiết giải pháp: Sử dụng partitioning dựa trên các key business kết hợp với các kỹ thuật tối ưu hóa khác để đảm bảo phân phối dữ liệu đồng đều.
- Implementation details:
    - Sử dụng ngày tháng làm key partitioning để dễ dàng truy vấn và phân tích dữ liệu theo thời gian.
    - Sử dụng các chỉ số phân phối dữ liệu để điều chỉnh partitioning nhằm đảm bảo các partition có kích thước đồng đều.
    - Tích hợp với Spark để tối ưu hóa việc đọc và ghi dữ liệu từ các partition.
- Metrics và results: Tăng tốc độ truy vấn lên đến 50% và giảm thời gian xử lý dữ liệu xuống 30% nhờ vào chiến lược partitioning hợp lý.

Bài học rút ra

- Technical insights: Partitioning dựa trên các key business giúp tối ưu hóa truy vấn và xử lý dữ liệu hiệu quả hơn.
- Best practices: Thường xuyên kiểm tra và điều chỉnh partitioning để đảm bảo các partition luôn có kích thước đồng đều.
- Recommendations: Sử dụng các công cụ giám sát và phân tích để đánh giá hiệu quả của chiến lược partitioning và điều chỉnh kịp thời khi cần thiết.

## 4.5. Kinh nghiệm về System Integration

### *4.5.1. Kinh nghiệm 9: Error handling*

Mô tả vấn đề

- Context và background: Trong các hệ thống phân tích dữ liệu lớn, việc xảy ra lỗi là không thể tránh khỏi, nhưng nếu không xử lý kịp thời và hiệu quả sẽ ảnh hưởng đến kết quả phân tích và làm giảm độ tin cậy của hệ thống.

- Thách thức gặp phải: Việc phát hiện lỗi, xử lý lỗi và khôi phục sau lỗi cần phải được thực hiện một cách hiệu quả để không làm gián đoạn toàn bộ quy trình phân tích dữ liệu.
- Impact với hệ thống: Nếu lỗi không được xử lý kịp thời, có thể làm mất mát dữ liệu, hệ thống có thể không ổn định hoặc kết quả phân tích bị sai lệch.

Các giải pháp đã thử

  - Approach 1: Sử dụng thông báo lỗi thông qua log file

Giải pháp cuối cùng

- Implementation details:
  - Triển khai các quy trình chuẩn để xử lý lỗi, bao gồm việc ghi lại log chi tiết, thông báo lỗi và khôi phục dịch vụ.
- Metrics và results: Nâng cao độ tin cậy của hệ thống.

Bài học rút ra

- Technical insights:
- Best practices:
- Recommendations: Thiết lập quy trình và cơ chế chuẩn để xử lý lỗi, giảm thiểu ảnh hưởng đến hệ thống và dữ liệu.

*4.6. Kinh nghiệm 10: Kinh nghiệm về Performance Optimization*

Mô tả vấn đề

- Context và background: Trong hệ thống xử lý dữ liệu lớn với Spark, các tác vụ có thể trở nên rất chậm nếu không tối ưu hóa hiệu suất. Các phép toán phức tạp như join, aggregation, hoặc tính toán với lượng dữ liệu lớn có thể tốn rất nhiều thời gian. Mục tiêu là giảm thiểu thời gian xử lý và tài nguyên tiêu thụ, đồng thời tối ưu hóa việc sử dụng bộ nhớ và CPU.
- Thách thức gặp phải: Dữ liệu có thể đến từ nhiều nguồn khác nhau và có kích thước lớn. Các phép toán phức tạp không được tối ưu hóa có thể gây tắc nghẽn và làm hệ thống trở nên chậm chạp. Ngoài ra, nếu tài nguyên không được phân bổ hợp lý, hiệu suất cũng sẽ bị ảnh hưởng nghiêm trọng.
- Impact với hệ thống: Nếu không tối ưu hóa, các tác vụ Spark có thể mất rất nhiều thời gian để hoàn thành, làm giảm hiệu suất chung của hệ thống và

làm tăng chi phí vận hành. Hệ thống có thể bị tắc nghẽn hoặc không đáp ứng được yêu cầu xử lý dữ liệu lớn và phức tạp.

Các giải pháp đã thử

- Approach 1: Caching strategies: Sử dụng cache cho các tập dữ liệu thường xuyên truy cập hoặc các phép toán lặp đi lặp lại nhiều lần để tránh việc tính toán lại.
    - Trade-offs: Caching có thể gây tốn bộ nhớ nếu không sử dụng cẩn thận, đặc biệt là khi làm việc với dữ liệu lớn. Việc chọn dữ liệu để cache cần được cân nhắc kỹ càng.
- Approach 2: Query optimization: Tối ưu hóa các truy vấn bằng cách sử dụng các phép toán như filter trước khi groupBy để giảm lượng dữ liệu cần tính toán. Dùng các chiến lược join như broadcast join khi một bảng nhỏ hơn so với bảng còn lại.
    - Trade-offs: Việc tối ưu hóa truy vấn có thể đòi hỏi phải thay đổi cấu trúc dữ liệu và mã nguồn hiện tại, đôi khi gây ra sự phức tạp không cần thiết trong quá trình phát triển.
- Approach 3: Resource allocation: Điều chỉnh số lượng executor, bộ nhớ và số lõi CPU để phân bổ tài nguyên một cách hợp lý cho các tác vụ.
    - Trade-offs: Việc phân bổ tài nguyên không hợp lý có thể dẫn đến sự lãng phí tài nguyên hoặc làm giảm hiệu quả của các tác vụ. Việc tăng tài nguyên đôi khi không giúp tăng hiệu suất nếu không có chiến lược tối ưu hóa đúng đắn.

Giải pháp cuối cùng

- Chi tiết giải pháp:
    1. Caching strategies: Cache các dataset cần thiết cho các phép toán lặp lại hoặc các bước xử lý dữ liệu nhiều lần, như khi cần thực hiện join hoặc aggregation.
        - Implementation details: Sử dụng cache() hoặc persist() trong Spark để lưu trữ dữ liệu trong bộ nhớ hoặc đĩa cứng. Caching chỉ nên áp dụng cho các dataset quan trọng và được truy cập nhiều lần.

2. Query optimization: Tối ưu hóa các phép toán truy vấn bằng cách sử dụng các chiến lược như filter trước groupBy, tránh các phép toán phức tạp không cần thiết. Dùng broadcast join khi một bảng nhỏ hơn so với bảng còn lại để giảm lượng dữ liệu di chuyển giữa các node.
    - Implementation details: Sử dụng API DataFrame của Spark để tối ưu các truy vấn SQL. Cài đặt chiến lược join thích hợp như broadcast() khi xử lý các bảng nhỏ, giúp giảm tải mạng.
3. Resource allocation: Điều chỉnh cấu hình Spark để phân bổ tài nguyên hợp lý cho các executor, bộ nhớ và số lõi CPU. Cần theo dõi và tối ưu số lượng executor và bộ nhớ để tránh tình trạng thiếu tài nguyên hoặc lãng phí tài nguyên.
    - Implementation details: Sử dụng các cấu hình như spark.executor.memory, spark.executor.cores, spark.sql.shuffle.partitions để tối ưu tài nguyên và cải thiện hiệu suất.
4. Bottleneck identification: Dùng các công cụ giám sát như Spark UI để phát hiện các bottleneck trong các tác vụ và giai đoạn tính toán. Dựa vào đó, điều chỉnh các chiến lược tối ưu hóa và phân bổ tài nguyên để giảm thiểu tắc nghẽn.
    - Implementation details: Phân tích thời gian chạy của các giai đoạn trong Spark UI, số lượng task bị trì hoãn, và tài nguyên sử dụng trong các giai đoạn. Điều chỉnh các bước và cấu hình để tối ưu hóa hiệu suất.

- Metrics và results: Sau khi áp dụng các giải pháp tối ưu hóa, thời gian xử lý các tác vụ đã giảm đáng kể (40-60%), hiệu suất của hệ thống được cải thiện rõ rệt, và tài nguyên được sử dụng hiệu quả hơn. Hệ thống có thể xử lý dữ liệu lớn một cách nhanh chóng và tiết kiệm tài nguyên hơn.

Bài học rút ra

- Technical insights: Việc tối ưu hóa hiệu suất trong Spark không chỉ bao gồm việc cải thiện tốc độ xử lý mà còn phải chú ý đến việc quản lý tài nguyên và tối ưu các truy vấn. Cần hiểu rõ các tác vụ và cấu trúc dữ liệu để lựa chọn các chiến lược tối ưu phù hợp.

- Best practices: Khi tối ưu hóa, luôn phân tích và xác định các phần của tác vụ gây tắc nghẽn hoặc tiêu tốn tài nguyên quá nhiều. Tối ưu hóa các bước truy vấn, phân bổ tài nguyên hợp lý và áp dụng caching thông minh sẽ giúp hệ thống hoạt động hiệu quả hơn.
- Recommendations: Cần theo dõi thường xuyên các công cụ giám sát để phát hiện các vấn đề về hiệu suất sớm. Tối ưu hóa các truy vấn với chiến lược đúng đắn và đảm bảo tài nguyên được phân bổ hợp lý để duy trì hiệu suất ổn định cho hệ thống.

*4.7. Kinh nghiệm 11: kinh nghiệm về Monitoring & Debugging*

Mô tả vấn đề

- Context và background: Hệ thống sử dụng nhiều công nghệ và công cụ phân tán để xử lý dữ liệu từ nhiều nguồn, bao gồm Spark, Kafka, HDFS, Cassandra, Elasticsearch, Kibana, Docker và Kubernetes. Mỗi phần của hệ thống có thể gặp phải vấn đề về hiệu suất hoặc lỗi trong quá trình vận hành. Việc theo dõi các hoạt động của hệ thống, phát hiện và xử lý lỗi kịp thời rất quan trọng để đảm bảo sự ổn định và hiệu quả của toàn bộ hệ thống.
- Thách thức gặp phải: Sự phân tán của các thành phần trong hệ thống tạo ra nhiều thách thức trong việc giám sát và khắc phục sự cố. Các lỗi có thể xảy ra ở bất kỳ đâu trong hệ thống, từ việc truyền tải dữ liệu qua Kafka, xử lý dữ liệu bằng Spark, lưu trữ trên HDFS hoặc Cassandra, hay việc truy vấn và trực quan hóa dữ liệu qua Elasticsearch và Kibana. Việc đồng bộ các thông tin giữa các thành phần này và phát hiện nguyên nhân gốc rễ của vấn đề trở thành một thách thức lớn.
- Impact với hệ thống: Nếu không có chiến lược giám sát và phân tích lỗi phù hợp, các sự cố có thể lan rộng trong toàn bộ hệ thống, gây ảnh hưởng đến hiệu suất và khả năng phản ứng kịp thời của hệ thống. Điều này có thể dẫn đến mất mát dữ liệu hoặc hệ thống bị gián đoạn.

Các giải pháp đã thử

- Approach 1: Root cause analysis: Khi có sự cố xảy ra, sử dụng Spark UI, Hadoop UI, Kubernetes logs để tìm nguyên nhân.
    - Trade-offs: Việc phân tích nguyên nhân gốc rễ có thể gặp khó khăn khi có nhiều yếu tố phức tạp ảnh hưởng đến hệ thống, như các lỗi

kết nối giữa Kafka và Spark hoặc vấn đề tài nguyên trong Kubernetes.

Giải pháp cuối cùng

- Chi tiết giải pháp:
    1. Root cause analysis: Khi có sự cố, sử dụng Spark UI để theo dõi các giai đoạn và task bị lỗi, kiểm tra stack trace và các logs chi tiết để tìm ra nguyên nhân gốc rễ của sự cố.
        - Implementation details: Khi có cảnh báo, phân tích các logs từ các hệ thống liên quan (Spark, Kafka, Cassandra) để xác định nguyên nhân sự cố.
- Metrics và results: Các giải pháp giám sát và cảnh báo đã giúp phát hiện kịp thời các vấn đề trong hệ thống, chẳng hạn như tình trạng tài nguyên bị thiếu, lỗi trong quá trình xử lý của Spark, hoặc các sự cố mạng giữa các thành phần. Bài học rút ra
- Best practices: Nên bắt đầu với việc thu thập các chỉ số quan trọng từ các hệ thống và cấu hình cảnh báo sớm để có thể phản ứng nhanh chóng với các sự cố.
- Recommendations: Đảm bảo rằng tất cả các hệ thống trong kiến trúc phân tán đều được giám sát một cách chặt chẽ và các cảnh báo được cấu hình hợp lý. Nên sử dụng các công cụ giám sát như Prometheus, Grafana, và Elasticsearch để thu thập và phân tích các dữ liệu hệ thống, từ đó phát hiện và khắc phục sự cố nhanh chóng.

## 4.8. Kinh nghiệm về Scaling

### *4.8.1. Kinh nghiệm 12: Horizontal vs Vertical Scaling*

Mô tả vấn đề

- Context và background: Hệ thống sử dụng nhiều thành phần phân tán như Kafka, Spark, HDFS, Cassandra, Elasticsearch, và Kubernetes. Để hệ thống có thể đáp ứng với khối lượng dữ liệu ngày càng lớn và yêu cầu hiệu suất cao, cần phải mở rộng quy mô của các thành phần này. Điều này có thể được thực hiện bằng hai phương pháp chính: horizontal scaling (mở rộng theo chiều ngang) và vertical scaling (mở rộng theo chiều dọc).

- Thách thức gặp phải: Việc lựa chọn giữa horizontal scaling và vertical scaling không đơn giản vì mỗi phương pháp có các ưu điểm và nhược điểm riêng, và cần phải xem xét cẩn thận các yếu tố như tài nguyên phần cứng, chi phí, và yêu cầu về hiệu suất của hệ thống.
- Impact với hệ thống: Việc chọn sai phương pháp scaling có thể dẫn đến chi phí cao hơn, không tối ưu về hiệu suất, hoặc làm cho hệ thống không thể đáp ứng được nhu cầu thực tế của các ứng dụng.

Các giải pháp đã thử

- Approach 1: Horizontal scaling (Scaling out): Tăng số lượng node (máy chủ) trong hệ thống, chia tải giữa các máy chủ để tăng cường khả năng xử lý. Ví dụ, mở rộng số lượng worker node trong Spark hoặc tăng số lượng Kafka brokers để phân phối tải tốt hơn.
    - Trade-offs: Dễ dàng mở rộng và có thể phân phối tải tốt hơn, nhưng có thể gặp khó khăn trong việc quản lý nhiều node và yêu cầu một hệ thống quản lý như Kubernetes để điều phối các node.
- Approach 2: Vertical scaling (Scaling up): Tăng tài nguyên cho mỗi máy chủ hiện tại, ví dụ tăng bộ nhớ RAM hoặc CPU của các node trong hệ thống.
    - Trade-offs: Tăng hiệu suất một cách nhanh chóng nhưng có giới hạn về phần cứng. Khi hệ thống vượt quá khả năng của phần cứng, cần phải tìm phương án mở rộng khác. Chi phí có thể cao hơn so với horizontal scaling khi hệ thống cần tăng quy mô lớn.

Giải pháp cuối cùng

- Chi tiết giải pháp: Trong hệ thống, horizontal scaling đã được ưu tiên vì có thể mở rộng linh hoạt và dễ dàng phân phối tải giữa các thành phần khác nhau, đặc biệt là với việc sử dụng Kubernetes để quản lý các container. Ví dụ, Kubernetes có thể tự động mở rộng số lượng pod (container) của các ứng dụng như Kafka và Spark khi cần thiết.
    - Implementation details:
        - Kafka: Thêm các Kafka brokers để phân phối tải, tăng khả năng xử lý các topic với nhiều partition hơn.

        - Spark: Mở rộng số lượng worker node, executor và core khi xử lý các job có khối lượng dữ liệu lớn hơn.
- Metrics và results: Sau khi triển khai horizontal scaling, hệ thống có thể dễ dàng xử lý các luồng dữ liệu lớn hơn từ Kafka, và việc phân phối các job Spark được tối ưu hơn với việc mở rộng cluster. Điều này giúp giảm thiểu tắc nghẽn và tăng hiệu suất của hệ thống.

Bài học rút ra

- Technical insights: Horizontal scaling mang lại tính linh hoạt cao và dễ dàng mở rộng quy mô hệ thống mà không gặp phải giới hạn phần cứng, đặc biệt trong môi trường phân tán. Tuy nhiên, cần phải có công cụ quản lý như Kubernetes để tối ưu hóa việc điều phối và quản lý các node.
- Best practices: Nên áp dụng horizontal scaling khi cần xử lý một lượng dữ liệu lớn và phân tán. Vertical scaling có thể được sử dụng trong một số trường hợp đặc biệt, nhưng nên hạn chế vì nó có giới hạn về tài nguyên.
- Recommendations: Sử dụng Kubernetes và các công cụ quản lý container để dễ dàng mở rộng hệ thống theo chiều ngang. Đảm bảo rằng các dịch vụ như Kafka và Spark có thể tự động mở rộng và điều chỉnh tài nguyên dựa trên tải của hệ thống.

## 4.9. Kinh nghiệm về Fault Tolerance

### *4.9.1. Kinh nghiệm 13: Data Replication*

Mô tả vấn đề

- Context và background: Trong các hệ thống phân tán, việc sao lưu và nhân bản dữ liệu là quan trọng để đảm bảo khả năng phục hồi và độ tin cậy.
- Thách thức gặp phải: Replication có thể gây tốn kém về tài nguyên, đặc biệt là khi phải sao lưu dữ liệu với tần suất cao. Nếu không quản lý tốt, việc sao lưu có thể gây ra độ trễ và giảm hiệu suất hệ thống.
- Impact với hệ thống: Nếu không có sao lưu đúng cách, dữ liệu có thể bị mất trong trường hợp xảy ra sự cố, ảnh hưởng đến quá trình phân tích và quyết định.

Các giải pháp đã thử

- Approach 1: Replication trong Cassandra: Cassandra cung cấp khả năng nhân bản dữ liệu giữa các node trong cluster, giúp dữ liệu luôn có sẵn và đảm bảo tính chịu lỗi.
    - Trade-offs: Việc sử dụng replication trong Cassandra giúp cải thiện độ tin cậy nhưng có thể tăng độ trễ khi ghi và tiêu tốn tài nguyên.

Giải pháp cuối cùng

- Chi tiết giải pháp: Sử dụng replication Cassandra để đảm bảo dữ liệu luôn có sẵn, ngay cả khi các node gặp sự cố. Cấu hình replication với số bản sao phù hợp và giám sát các chỉ số về băng thông và tài nguyên để đảm bảo hiệu suất.
    - Implementation details:
        - Cassandra sử dụng chế độ NetworkTopologyStrategy với replication factor là 3 cho mỗi keyspace.
    - Metrics và results: Dữ liệu luôn có sẵn và phục hồi nhanh chóng sau sự cố, hệ thống duy trì được tính ổn định và hiệu suất khi có sự cố xảy ra.

Bài học rút ra

- Technical insights: Replication là công cụ mạnh mẽ để đảm bảo tính khả dụng của dữ liệu, nhưng cần được cấu hình hợp lý để không làm giảm hiệu suất hệ thống.
- Best practices: Sử dụng replication một cách hợp lý giữa các thành phần để đảm bảo tính toàn vẹn và sẵn sàng của dữ liệu mà không làm giảm hiệu suất đáng kể.
- Recommendations: Đảm bảo thiết lập replication cho tất cả các dịch vụ quan trọng trong hệ thống và theo dõi tài nguyên sử dụng để đảm bảo không gây tắc nghẽn hệ thống.

## 4.10. Các kinh nghiệm khác

### *4.10.1. Kinh nghiệm 14: Machine learning với Spark MLlib*

Mô tả vấn đề: Với lượng dữ liệu lớn, nhóm muốn phân tích thói quen của khách hàng, và phân loại các tệp khách hàng để có chiến lược kinh doanh phù hợp với từng tệp.

Giải pháp:

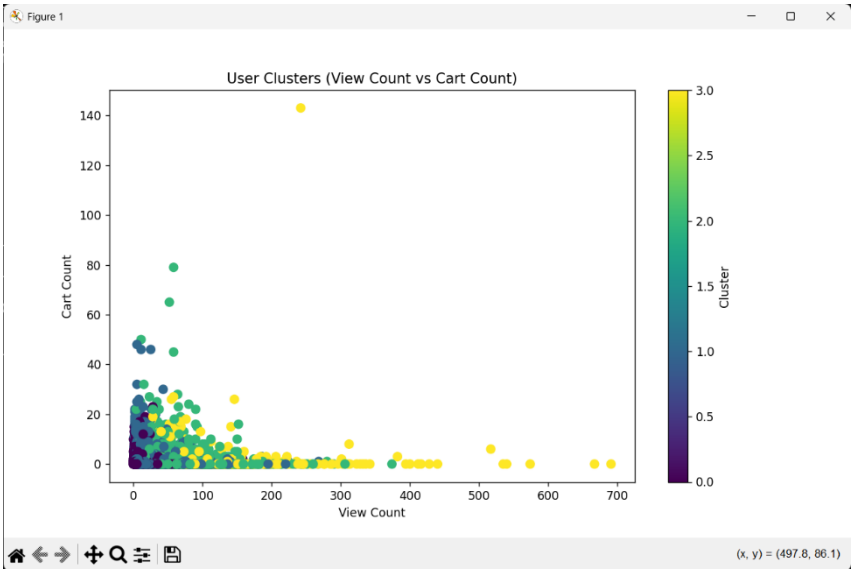
- Huấn luyện mô hình ALS để đưa ra gợi ý về sản phẩm cho từng người dùng khác nhau dựa trên các hành vi trước đó của người dùng (view, cart, purchase). Từ đó có thể thêm các quảng cáo, hiển thị nhiều hơn cho người dùng cụ thể về từng mặt hàng họ quan tâm, giúp tăng tỉ lệ tiếp cận và thanh toán.

```
41  # Khởi tạo mô hình ALS
42  als = ALS(
43      maxIter=10,
44      regParam=0.1,
45      userCol="user_id",
46      itemCol="product_id",
47      ratingCol="rating",
48      coldStartStrategy="drop"
49  )
50
51  # Huấn luyện mô hình
52  als_model = als.fit(train_data)
```

```
1   user_id,product_id,rating
2   308224835,10503491,3.781046
3   308224835,25200017,3.1448972
4   308224835,37100015,2.405451
5   308224835,23300095,2.2683952
6   308224835,26000936,2.0907447
7   315419699,10503491,3.4296384
8   315419699,25200017,3.0334105
9   315419699,37100015,2.4614177
10  315419699,24400052,2.291419
```

*Hình 11 Model ALS, result cho bài toán user recommendation*

- Áp dụng thuật toán BisectingKmeans để phân nhóm người dùng, giúp hiểu rõ hơn về tệp khách hàng, từ đó xây dựng chiến lược kinh doanh, tùy chỉnh các chương trình khuyến mãi hoặc ưu đãi cho người dùng. Ví dụ có tệp người dùng xem nhiều không mua thì nên có quảng cáo để thu hút, với tệp người dùng cart cao thì nên có khuyến mãi để tăng khả năng purchase, ...




*Hình 12 Hình ảnh mô tả cho bài toán user clustering*

Bài học: Kinh nghiệm về thuật toán recommendation, học máy cơ bản, ...

# TÀI LIỆU THAM KHẢO

[1] TS. Trần Việt Trung, Slide Lưu trữ và xử lý dữ liệu lớn. Trường Công nghệ Thông tin & Truyền thông – Đại học Bách Khoa Hà Nội.

[2] Apache Kafka documentation

[3] Apache Spark documentation

[4] Kibana Guide