포팅메뉴얼

1. 사용 도구

- **이슈 관리**: Notion, Jira
- **형상 관리**: GitLab
- **커뮤니케이션**: MatterMost, Discord
- **디자인**: Figma
- **CI/CD**: Jenkins, argo, Docker, Kubernetes
- **모니터링**: ELK(Elastic Search, Logstash, Kibana), Grafana

2. 개발 도구

2.1. Frontend

주요 개발 도구

1. **Next.js** (14.2.7) - React 기반 프레임워크로 SSR 및 SSG 지원.
2. **TypeScript** (5.6.2) - JavaScript에 타입을 추가하여 코드의 안정성 향상.
3. **Babel** - 최신 JavaScript 문법을 구형 브라우저에서도 사용 가능하게 변환.
    - `@babel/core`
    - `@babel/preset-env` - ECMAScript 문법 변환.
    - `@babel/preset-react` - React 문법 변환.
    - `@babel/preset-typescript` - TypeScript 문법 변환.
    - `@babel/runtime` - Babel의 런타임 헬퍼 기능.
4. **pnpm** (9.9.0) - 빠르고 효율적인 패키지 매니저.
5. **ESLint** - 코드의 일관성 및 품질을 유지하는 린팅 도구.
    - `eslint-config-next` - Next.js의 기본 ESLint 설정.

- `eslint` - 기본 ESLint 패키지.

주요 프론트엔드 라이브러리

1. **React** (18.3.1) - 사용자 인터페이스를 구성하기 위한 라이브러리.
2. **Styled-components** (6.1.13) - CSS-in-JS로 스타일링을 지원하는 라이브러리.
3. **Emotion** - CSS-in-JS 스타일링 도구.
   - `@emotion/react`
   - `@emotion/styled`
4. **Three.js** (0.142.0) - 3D 그래픽 및 애니메이션을 웹에서 구현하는 라이브러리.
5. **Chart.js** - 차트를 그리기 위한 라이브러리.
   - `chart.js`
   - `react-chartjs-2` - Chart.js와 React를 통합한 라이브러리.
   - `chartjs-adapter-date-fns`, `chartjs-chart-wordcloud`, `chartjs-plugin-zoom` - 차트 플러그인.
6. **D3.js** (7.9.0) - 데이터 기반의 인터랙티브 시각화 도구.
   - `d3-cloud` - 워드 클라우드 구현을 위한 라이브러리.
   - `d3-scale`, `d3-scale-chromatic` - D3.js의 스케일 및 색상 관련 기능.
7. **Framer Motion** (11.8.0) - 애니메이션 구현을 위한 React 기반 라이브러리.
8. **Axios** (1.7.7) - HTTP 클라이언트로 API 요청에 사용.
9. **Recoil** (0.7.7) - React 상태 관리 라이브러리.
10. **Klinecharts** (9.8.10) - 주식 차트 구현 라이브러리.
11. **Lodash** (4.17.21) - 유틸리티 함수 제공 라이브러리.
12. **Moment.js** (2.30.1) - 날짜와 시간 처리 라이브러리.
13. **JWT Decode** (4.0.0) - JSON Web Token을 디코드하는 라이브러리.
14. **React Icons** (5.3.0) - 다양한 아이콘을 제공하는 라이브러리.

기타 유틸리티 및 도구

1. **React Apexcharts** (1.4.1) - ApexCharts와 React를 결합한 차트 라이브러리.
2. **React Calendar** (5.0.0) - 달력 컴포넌트.

3. **React DatePicker** (7.4.0) - 날짜 선택 컴포넌트.
4. **React Error Boundary** (4.0.13) - React에서 에러 경계를 설정하여 오류 처리.
5. **React Lodash** (0.1.2) - Lodash와 React 통합 라이브러리.
6. **Anime.js** (3.2.2) - 애니메이션 라이브러리.
7. **Sharp** (0.33.5) - 이미지 처리 라이브러리.
8. **Null-loader** (4.0.1) - Webpack에서 사용되지 않는 모듈을 무시하기 위한 로더.
9. **Embla Carousel React** (8.3.0) - React용 슬라이드 컴포넌트 라이브러리.

타입 정의

- **@types** 패키지들:
  - `@types/lodash`, `@types/node`, `@types/react`, `@types/react-dom`, `@types/styled-components`, `@types/three`

빌드 및 최적화

- **Terser-webpack-plugin** (5.3.10) - Webpack을 위한 JavaScript 압축 도구.

2.2. Backend

- **개발 언어**: Java(17)
- **프레임워크:** Spring Boot (3.3.3)
  - **Dependencies:**
    - Spring Boot Dev Tools
    - Spring Data JPA
    - Lombok
    - MySQLDB Driver
    - Spring Web
    - Spring Data Redis
    - Spring for Apache Kafka
    - Spring Security

- WebSocket
- Spring Reactive Web

2.3. BigData
▼ Docker로 Hadoop 설치하기(Test) —> 2트 도전 ~ 컨테이너에 올리는거 성공!

https://velog.io/@woojoo121/도커-컨테이너에-하둡-설치-통신#오늘-이-포스트에서-다루는-영역

이 분과 함께 여행을 떠날거다

```
HOST main
        HostName {외부 ip 주소}
        User jwjinn
        IdentityFile /home/joo/.ssh/google.pem
```

- /home/joo/.ssh에 있는 config 파일을 위와 같이 수정했다.
- 메타데이터의 ssh에 public key를 등록하면, 내가 아무리 많은 인스턴스를 생성, 삭제를 해도 해당 인스턴스의 authorized_keys에 public key를 다 등록을 자동으로 해준다.

접속

```
(base) joo@joo-IdeaPad-5-Pro-14ACN6:~$ ssh main
```

- 접속 가능

초반 해당 부분은 나는 local로 할거라서 따라하지 않음

업데이트

```
$ sudo apt-get -y update

$ sudo apt-get -y upgrade

$ sudo apt-get -y dist-upgrade
```

```
$ sudo apt-get install -y vim wget unzip ssh openssh-* net-
```

ubuntu image pull

```
20.04: Pulling from library/ubuntu
846c0b181fff: Pull complete
Digest: sha256:0e0402cd13f68137edb0266e1d2c682f217814420f2c
Status: Downloaded newer image for ubuntu:20.04
docker.io/library/ubuntu:20.04

$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
ubuntu       20.04     d5447fc01ae6   2 weeks ago   72.8MB
```

도커 실행 후 접속

```
$ docker run -d --name nameNode -p 23:22 -p 9871:9870 -p 80
$ docker attach nameNode
```

sudo 설치 및 기초 업데이트(여기부터는 nameNode 내부)

```
apt-get update && apt-get install -y sudo

sudo apt-get -y update

sudo apt-get -y upgrade

sudo apt-get -y dist-upgrade

sudo apt-get install -y vim wget unzip ssh openssh-* net-to
```

자바 설치

```
sudo apt-get install -y openjdk-11-jdk

sudo vim ~/.bashrc #환경등록

export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
export PATH=${PATH}:${JAVA_HOME}/bin
```

파이썬 설치

```
sudo apt install python3-pip -y

sudo vim ~/.bashrc

alias python=python3

# 설정 이후 source해서 bash 변경 내용 적용해주기
source ~/.bashrc
```

파이스파크 설치

```
sudo pip3 install pyspark
```

하둡다운(※버전 엄수 필요, 이후 spark, mongodb 연동 작업 시 버전 호환 이슈 발생 가능성 있음)

```

### 3.3.4 -> 3.3.5 변경(3.3.4로 설치 시 DataNotFound 발생)

wget https://dlcdn.apache.org/hadoop/common/hadoop-3.3.5/ha

sudo tar -xvzf hadoop-3.3.5.tar.gz -C /usr/local/

sudo chown root:root -R /usr/local/hadoop-3.3.5
```

스파크 다운

```
### 3.2.3 -> 3.4.3 버전

wget https://dlcdn.apache.org/spark/spark-3.4.3/spark-3.4.3

sudo tar xvzf spark-3.4.3-bin-without-hadoop.tgz -C /usr/lo
```

압축파일명 변경

```
mv /usr/local/hadoop-3.3.5 /usr/local/hadoop
mv /usr/local/spark-3.4.3-bin-without-hadoop /usr/local/spa

# ls로 바뀐거 확인
```

다시, 환경등록

```
# JAVA
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
export PATH=${PATH}:${JAVA_HOME}/bin

#python
alias python=python3


#Hadoop
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin


#Spark
```

```
export SPARK_HOME=/usr/local/spark
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
export SPARK_DIST_CLASSPATH=$($HADOOP_HOME/bin/hadoop class
```

하둡 환경설정

sudo vim core-site.xml

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://nameNode:9000</value>
    </property>
</configuration>
```

- 모든 데이터노드들이 nameNode에 9000번 포트로 통신을 한다.

sudo vim hdfs-site.xml

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>2</value>
    </property>

    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:///hdfs_dir/namenode</value>
    </property>

    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:///hdfs_dir/datanode</value>
    </property>
```

```
    <property>
        <name>dfs.namenode.secondary.http-address</name>
        <value>dataNode:50090</value>
    </property>
</configuration>
```

sudo vim yarn-site.xml

```
<configuration>
    <property>
        <name>yarn.nodemanager.local-dirs</name>
        <value>file:///hdfs_dir/yarn/local</value>
    </property>

    <property>
        <name>yarn.nodemanager.log-dirs</name>
        <value>file:///hdfs_dir/yarn/logs</value>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>nameNode</value>
    </property>
</configuration>
```

sudo vim mapred-site.xml

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

sudo vim hadoop-env.sh

```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64

#hadoop user
export HDFS_NAMENODE_USER="root"
export HDFS_DATANODE_USER="root"
export HDFS_SECONDARYNAMENODE_USER="root"
export YARN_RESOURCEMANAGER_USER="root"
export YARN_NODEMANAGER_USER="root"
```

스파크 환경설정

spark-defaults.conf

```
# 해당 폴더로 이동
root@1d3a91c4f6a8:/usr/local/spark/conf

$ sudo cp spark-defaults.conf.template spark-defaults.conf
$ sudo vim spark-defaults.conf

# 다음 내용 추가
spark.master                            yarn
```

spark-env.sh

```
# 위와 같이 spark/conf 경로에서 진행
$ sudo cp spark-env.sh.template spark-env.sh
$ sudo vim spark-env.sh

# 다음 내용 추가
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64

export HADOOP_HOME=/usr/local/hadoop

export SPARK_MASTER_HOST=master
```

```
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
export SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoop

export PYSPARK_PYTHON=/usr/bin/python3
export PYSPARK_DRIVER_PYTHON=/usr/bin/python3
```

도커 이미지 커밋

- ctrl + p, ctrl + q로 정지하지 않고 나간다

```
$ docker commit nameNode sparkHadoop

$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
sparkhadoop   latest    44531a2148d7   8 seconds ago    4.61
ubuntu        20.04     d5447fc01ae6   2 weeks ago      72.8
```

- 지금까지 nameNode 컨테이너에서 작업한 것들을 sparkHadoop이라는 이름의 이미지로 저장

데이터노드 컨테이너 생성

```
$ docker run -d --name dataNode -it sparkhadoop
```

- 저장한 이미지로 만드는 dataNode
- nameNode에서 작업한 환경설정이 복사가 되어있다.

도커 네트워크

네트워크 생성, 등록

```
docker network create hadoop

docker network connect hadoop nameNode
```

```
docker network connect hadoop dataNode

# 되었는지 확인
$ docker network inspect hadoop

# Ping을 보내서 통신이 되는지 확인
sudo apt-get install iputils-ping
ping dataNode
```

- 추후 다른것들을 연동할때에도(필자는 NoSQL) 해당 네트워크에 추가해주어야 한다.

두 컨테이너에 호스트 등록

```
sudo vim /etc/hosts

172.18.0.3      dataNode
172.18.0.2      nameNode
```

컨테이너끼리 루트로 통신가능하게 세팅

```
sudo vim /etc/ssh/sshd_config

# 주석되어있는 아래 코드들 찾아서 주석 해제 후 코드 변경

PermitRootLogin yes

PasswordAuthentication yes

passwd # root 비밀번호를 미리 설정해두자.
```

필요 설정 파일 설정 및 명령어들 모음

```
sudo apt-get install openssh-server openssh-client

sudo /etc/init.d/ssh restart
sudo /etc/init.d/ssh stop
sudo /etc/init.d/ssh start
```

- ssh 통신을 위한 설정들임

ssh key 생성

```
ssh-keygen # 키를 만든다.

cd ~/.ssh # 키가 저장되어 있는 위치로 이동

ssh-copy-id root@nameNode # public key를 root계정의 nameNode(
ssh-copy-id root@dataNode

# 테스트
ssh root@dataNode
```

nameNode 컨테이너에서 실행

```
/usr/local/hadoop/bin/hdfs namenode -format /hdfs_dir
```

dataNode 컨테이너에서 실행

```
/usr/local/hadoop/bin/hdfs namenode -format /hdfs_dir

/usr/local/hadoop/bin/hdfs datanode -format /hdfs_dir
```

nameNode 컨테이너에서 워커들 등록

스파크 세팅

```
pwd # /usr/local/spark/conf

cp workers.template workers

vi /usr/local/spark/conf/workers

nameNode
dataNode
```

하둡

```
pwd # /usr/local/hadoop/etc/hadoop

vi workers

dataNode
nameNode
```

Hadoop 테스트 해보기

```
# 1. Hadoop 시작
$ start-all.sh

# 2. jps로 nameNode랑 dataNode 잘 뜨는지 확인
# 이건 예시
$ jps
837 ResourceManager
968 NodeManager
409 DataNode
1724 Jps
252 NameNode
621 SecondaryNameNode

# 3. Live datanodes(2) 확인하기 : 2개의 데이터노드가 구동되고 있음을
$ hdfs dfsadmin -report
```

```
# 4. 폴더 하나 만들고 반영되나 확인
$ hadoop fs -mkdir -p /user/dave
$ hadoop fs -ls / # 이걸로 확인해도 되고, WEB UI로 확인해도 됨
```

Spark 테스트 해보기(Scala)

```
# Spark Shell 실행하여 구동
$ spark-shell # 프롬프트에 scala>가 표시되면 Spark가 정상적으로 작
# $ pyspark # 위에는 scala, 이건 pyspark 실행 명령어

# HDFS에 데이터 삽입하는 작업 간단히 진행

# 1. Spark Session 생성
import org.apache.spark.sql.SparkSession

# 2. 간단한 데이터 생성(에러는 뜰건데, 잘 생성됨)
val spark = SparkSession.builder
  .appName("HDFS Test")
  .master("yarn")
  .getOrCreate()

val data = Seq(
  ("Seoul", 200),
  ("Busan", 300),
  ("Incheon", 150)
)

import spark.implicits._
val df = data.toDF("City", "Population")

# 3. HDFS 경로 설정 및 저장
df.coalesce(1)
    .write
  .format("csv")
  .save("hdfs://nameNode:9000/user/baejun/nosqltest")

# 4. nameNode로 가서 잘 들어갔는지 조회
```

```
hdfs dfs -ls /user/baejun/test1File

# Found 3 items 하고 아이템 목록 뜨면 성공 !
```

Spark에서 HDFS data load(Scala)

```
# 1. Spark Session 생성
val spark = SparkSession.builder
  .appName("HDFS Load Example")
  .master("yarn")
  .getOrCreate()

# 2. Data Load
val df = spark.read
    .format("csv")
    .option("header", "true")  // 헤더가 있는 경우
    .load("hdfs://nameNode:9000/user/baejun/test1File/part-

# 3. 확인
df.show()
```

```
scala> df.show()
2024-09-15 14:34:22,727 INFO codegen.CodeGenerator: Code generated in 8.512161 ms
2024-09-15 14:34:22,744 INFO codegen.CodeGenerator: Code generated in 10.103488 ms
+-------+----------+
|   City|Population|
+-------+----------+
|  Seoul|       200|
|  Busan|       300|
|Incheon|       150|
+-------+----------+
```

▼ NoSQL(MongoDB) 컨테이너에 올리고 Spark 컨테이너와 연동하기

▼ NoSQL(MongoDB) 컨테이너 위에 올리기

도커는 이미 깔려있다는 전제로 진행

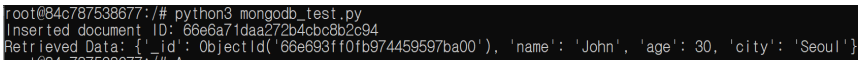
1. 기본세팅

```
# 버전 명시 안하면 가장 최근 버전 pull함(7.0.14) 버전
# 1. mongo image pull
$ docker pull mongo
$ docker pull mongo:7.0.14 # 프로젝트 적용 버전

# 2. 잘 가져왔는지 확인
$ docker imgaes
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
mongo        latest    545fe6e3d65b   4 weeks ago   721M

# 3. 컨테이너 실행
# 필자는 로컬에 따로 27017사용중이라 27018으로 매핑
$ docker run --name mongo-container -d -p 27018:27017 mo

# 4. 접속
$ docker exec -it mongo-container bash

# 5. mongo shell에 접속
$ mongosh
```

2. 테스트

```
# 1. pip, python3 설치(mongo shell 환경이 아닌 컨테이너 내부 환
$ apt update
$ apt install python3 python3-pip -y
$ pip3 install pymongo

# 2. 만약 python3 설치 안되어있으면 진행
$ python3 --version  # Python 설치 여부 확인

$ apt update
$ apt install python3

# 3. vi 설치
$ apt update
$ apt install vim -y
```

```
# 4. 테스트
$ vi mongodb_test.py # test py 생성

# 5. 테스트 코드 삽입
from pymongo import MongoClient

# MongoDB에 연결
client = MongoClient('localhost', 27018)  # 호스트의 local
db = client['test_db']  # 데이터베이스 이름 설정
collection = db['test_collection']  # 컬렉션 이름 설정

# 데이터 삽입
data = {"name": "John", "age": 30, "city": "Seoul"}
result = collection.insert_one(data)
print(f"Inserted document ID: {result.inserted_id}")

# 데이터 조회
retrieved_data = collection.find_one({"name": "John"})
print("Retrieved Data:", retrieved_data)

# 데이터베이스 연결 종료
client.close()

# 7. 실행
$ python3 mongodb_test.py
```



mongo 컨테이너 환경 구축 완료!

▼ NoSQL(MongoDB)와 Spark 연동

1. 네트워크 연동

- 필자같은 경우는, 이 과정을 빼먹고 진행을 해서 계속 안됐다..
- 이 이유라곤 생각도 못하고, mongo와 spark의 문제에 포커스만 맞춰서.. 거의 반나절을 날려먹었다.

```
# 1. spark가 속해있는 network 환경에 mongocontainer 추가
# 필자의 경우엔 hadoop network에, nameNode(HDFS), dataNode(
$ docker network connect hadoop mongo-container

# 2. 연결확인
$ docker network inspect hadoop # 1. mongo-container 잘 들

$ sudo apt-get install iputils-ping
$ ping mongo-container # 컨테이너 간 통신 잘 되는지 확인하기
```

2. pyspark 실행

```
# ※주의 : spark버전이 3.4라서 mongo-spark-connector 버전이 1
# mongo-container는 mongo-shell 내부가 아닌 컨테이너 내부단계에
pyspark --conf "spark.mongodb.read.connection.uri=mongod
--conf "spark.mongodb.write.connection.uri=mongodb://mon
--packages org.mongodb.spark:mongo-spark-connector_2.12:


# MongoDB에서 데이터 읽기
df = spark.read.format("mongodb").load()

# 데이터 출력
df.show()
```

전처리 까지의 과정

▼ nosql → spark → hadoop(데이터 전처리 : 유사도 분석으로 중복기사 제거 + 날짜별 파티셔닝)

▼ 초안

```
# nosql의 news_db의 news_collection1에 데이터가 있다고 가정

# pyspark 실행 : 해당 db의 collection으로 접속
pyspark --conf "spark.mongodb.read.connection.uri=mongod
--conf "spark.mongodb.write.connection.uri=mongodb://mor
--packages org.mongodb.spark:mongo-spark-connector_2.12:

# nosql -> spark -> (날짜별 파티셔닝)HDFS

# import
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Spark 세션 생성 : nosql 경로 지정
spark = SparkSession.builder \
    .appName("nosql-to-hdfs") \
    .config("spark.mongodb.read.connection.uri", "mongod
    .getOrCreate()

# NoSQL 읽어오기
df = spark.read.format("mongodb").load()

# 'date' 컬럼에서 날짜 부분만 추출 (예: '2022.01.20')
df_with_date = df.withColumn("date_only", split(col("dat

# 날짜 범위 필터링
filtered_df = df_with_date.filter((col("date_only") >= "

# 날짜 형식이 'yyyy.MM.dd' 이므로 해당 형식으로 필터 적용
# 예를 들어, 2022년 1월 11일 ~ 2022년 1월 20일 사이의 데이터 필
filtered_df = df.filter((col("date") >= "2022.01.11") &

# 날짜만 기준으로 파티셔닝하여 HDFS에 저장
filtered_df.write \
    .mode("overwrite") \
    .partitionBy("date_only") \
    .format("parquet") \
```

```
    .save("hdfs://nameNode:9000/user/baejun/news_data_pa
f
# spark로 데이터 저장 확인
# HDFS에서 저장된 parquet 파일을 다시 불러오기
df_from_hdfs = spark.read.parquet("hdfs://nameNode:9000/

# 데이터 확인
df_from_hdfs.show()
```

```
# 컨테이너 내부에서 다운로드
pip install scikit-learn
pip install numpy


from pyspark.sql import functions as F
from pyspark.sql import Window
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import Normalizer
from pyspark.sql import Row
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    return df.filter(F.col('date').startswith(date))

# 코사인 유사도를 계산하고, 유사도가 0.8 이상인 기사를 제거하는 함수
def remove_similar_articles(df):
    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol=
    wordsData = tokenizer.transform(df)

    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="r
    featurizedData = hashingTF.transform(wordsData)

    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변
```

```
    idf = IDF(inputCol="rawFeatures", outputCol="feature
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)

    # 데이터프레임을 리스트로 변환 후 코사인 유사도 계산
    rows = rescaledData.select('features').rdd.map(lambd
    similarity_matrix = cosine_similarity(rows)

    # 유사도 행렬을 바탕으로 유사한 기사 필터링
    remove_indices = set()
    for i in range(len(similarity_matrix)):
        for j in range(i+1, len(similarity_matrix)):
            if cosine_similarity([rows[i]], [rows[j]])[0
                remove_indices.add(j)

    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.co

    # 필터링된 데이터를 데이터프레임으로 변환
    filtered_df = spark.createDataFrame(filtered_rows, d

    return filtered_df

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
    df.write \
      .partitionBy("date_only") \
      .mode("overwrite") \
      .parquet(output_path)

# 전체 로직
def process_and_save(df, date_list, output_path):
    for date in date_list:
        # 날짜별로 필터링
        df_filtered = filter_by_date(df, date)

        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique = remove_similar_articles(df_filtered)
```

```
        # 날짜별로 파티셔닝하여 HDFS에 저장
        save_partitioned_by_date(df_unique, f"{output_pa

# 날짜 리스트 예시
date_list = ['2022.01.11', '2022.01.12', '2022.01.13']

# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.ne
    .load()

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_dat
process_and_save(df, date_list, output_path)
```

```
from pyspark.sql import functions as F
from pyspark.sql import Window
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import Normalizer
from pyspark.sql import Row
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    return df.filter(F.col('date').startswith(date))

# 코사인 유사도를 계산하고, 유사도가 0.8 이상인 기사를 제거하는 함수
def remove_similar_articles(df):
    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol=
    wordsData = tokenizer.transform(df)
    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="r
    featurizedData = hashingTF.transform(wordsData)
```

```
    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변
    idf = IDF(inputCol="rawFeatures", outputCol="feature
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)
    # 데이터프레임을 리스트로 변환 후 코사인 유사도 계산
    rows = rescaledData.select('features').rdd.map(lambd
    similarity_matrix = cosine_similarity(rows)
    # 유사도 행렬을 바탕으로 유사한 기사 필터링
    remove_indices = set()
    for i in range(len(similarity_matrix)):
        for j in range(i+1, len(similarity_matrix)):
            if cosine_similarity([rows[i]], [rows[j]])[0
                remove_indices.add(j)
    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.co
    # 필터링된 데이터를 데이터프레임으로 변환
    filtered_df = spark.createDataFrame(filtered_rows, d
    return filtered_df

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
    # 'date_only' 컬럼을 추가하여 날짜별 파티셔닝을 가능하게 함
    df_with_date_only = df.withColumn('date_only', F.col
    df_with_date_only.write \
      .partitionBy("date_only") \
      .mode("overwrite") \
      .parquet(output_path)

# 전체 로직
def process_and_save(df, date_list, output_path):
    for date in date_list:
        # 날짜별로 필터링
        df_filtered = filter_by_date(df, date)

        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique = remove_similar_articles(df_filtered)

        # 날짜별로 파티셔닝하여 HDFS에 저장
```

```
        save_partitioned_by_date(df_unique, f"{output_pa

# 날짜 리스트 예시
date_list = ['2022.01.11']

# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.ne
    .load()

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_dat
process_and_save(df, date_list, output_path)
```

▼ 미완들

```
from pyspark.sql import functions as F
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from pyspark.sql.types import FloatType
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import time

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    return df.filter(F.col('date').startswith(date))

# 코사인 유사도를 계산하는 UDF (벡터 간 유사도 계산)
def cosine_similarity_udf(v1, v2):
    return float(np.dot(v1, v2) / (np.linalg.norm(v1) *

# 분산 환경에서 코사인 유사도를 계산하고, 유사도가 0.9 이상인 기사를
def remove_similar_articles(df):
    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol=
    wordsData = tokenizer.transform(df)
```

```
    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="r
    featurizedData = hashingTF.transform(wordsData)
    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변
    idf = IDF(inputCol="rawFeatures", outputCol="feature
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)
    # 각 벡터를 broadcast 변수로 전달
    vectors = rescaledData.select("features").rdd.map(la
    broadcast_vectors = spark.sparkContext.broadcast(vec
    # UDF를 등록하여 코사인 유사도를 분산 환경에서 계산
    cosine_udf = F.udf(lambda v1, v2: cosine_similarity_
    # 유사도 계산을 위한 Self-Join
    window_spec = Window.orderBy(F.lit(1))
    df_with_index = rescaledData.withColumn("row_idx", F
    # 데이터프레임을 자기 자신과 Join하여 유사도 계산
    joined_df = df_with_index.alias("df1").join(
        df_with_index.alias("df2"),
        F.col("df1.row_idx") < F.col("df2.row_idx")
    ).select(
        F.col("df1.row_idx").alias("idx1"),
        F.col("df2.row_idx").alias("idx2"),
        cosine_udf(F.col("df1.features"), F.col("df2.fea
    )
    # 유사도가 0.9 이상인 행을 제거
    to_remove = joined_df.filter(F.col("cosine_sim") >=
    remove_indices = to_remove.rdd.flatMap(lambda x: x).
    # 중복되지 않은 데이터만 남기기
    filtered_df = rescaledData.filter(~F.col("row_idx").
    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    return filtered_df, removed_count

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
    df_with_date_only = df.withColumn('date_only', F.col
    df_with_date_only.write \
      .partitionBy("date_only") \
```

```
      .mode("overwrite") \
      .parquet(output_path)

# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
    total_start_time = time.time()  # 전체 실행 시간 시작

    for date in date_list:
        print(f"Processing data for date: {date}")

        # 날짜별로 필터링
        start_time = time.time()  # 날짜별 처리 시간 시작
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이터

        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique, removed_count = remove_similar_articl
        unique_count = df_unique.count()  # 중복 제거된 후

        # 날짜별로 파티셔닝하여 HDFS에 저장
        save_partitioned_by_date(df_unique, f"{output_pa

        # 처리 시간과 데이터 개수 출력
        elapsed_time = time.time() - start_time
        print(f"Date: {date} - Original: {original_count

    total_elapsed_time = time.time() - total_start_time
    print(f"Total processing time: {total_elapsed_time:.

# 날짜 리스트 예시
date_list = ['2022.01.11']

# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.ne
    .load()
```

```
# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_dat
process_and_save(df, date_list, output_path)
```

```


from pyspark.sql import functions as F
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from sklearn.feature_extraction.text import TfidfVectori
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import time

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    return df.filter(F.col('date').startswith(date))

# 개선된 코사인 유사도 분석 함수
def remove_similar_articles(df, chunk_size=500):
    # 실행 시간 측정을 위한 시작 시간
    start_time = time.time()
    # 1. content를 TF-IDF 벡터로 변환 (sklearn 사용)
    content_list = df.select("content").rdd.flatMap(lamb
    tfidf_vectorizer = TfidfVectorizer()
    tfidf_matrix = tfidf_vectorizer.fit_transform(conter
    # 2. TF-IDF 행렬을 브로드캐스트로 Spark 클러스터에 전송
    broadcast_tfidf_matrix = spark.sparkContext.broadcas
    # 3. TF-IDF 행렬을 청크로 나눠서 분산 처리
    tfidf_chunks = np.array_split(tfidf_matrix.toarray()
    rdd = spark.sparkContext.parallelize(tfidf_chunks)
    # 4. 각 청크에 대해 코사인 유사도 계산
    def calculate_cosine_similarity(chunk, tfidf_matrix_
        chunk_matrix = np.array(chunk)
        similarity_scores = cosine_similarity(chunk_matr
        return similarity_scores
    similarity_results = rdd.map(lambda chunk: calculate
    # 5. 유사도가 0.9 이상인 문서 필터링
    remove_indices = set()
    for i in range(len(similarity_results)):
```

```
        for j in range(i+1, len(similarity_results[i])):
            if similarity_results[i][j] >= 0.9:
                remove_indices.add(j)
    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.co
    filtered_df = spark.createDataFrame(filtered_rows, d
    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    # 유사도 분석 실행 시간 출력
    elapsed_time = time.time() - start_time
    print(f"유사도 분석 시간: {elapsed_time:.2f} 초")
    return filtered_df, removed_count

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
    df_with_date_only = df.withColumn('date_only', F.col
    df_with_date_only.write \
      .partitionBy("date_only") \
      .mode("overwrite") \
      .parquet(output_path)

# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
    total_start_time = time.time()  # 전체 실행 시간 시작
    for date in date_list:
        print(f"Processing data for date: {date}")
        # 날짜별로 필터링
        start_time = time.time()  # 날짜별 처리 시간 시작
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이터
        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique, removed_count = remove_similar_articl
        unique_count = df_unique.count()  # 중복 제거된 후
        # 날짜별로 파티셔닝하여 HDFS에 저장
        save_partitioned_by_date(df_unique, f"{output_pa
        # 처리 시간과 데이터 개수 출력
        elapsed_time = time.time() - start_time
        print(f"Date: {date} - Original: {original_count
```

```
    total_elapsed_time = time.time() - total_start_time
    print(f"Total processing time: {total_elapsed_time:.

# 날짜 리스트 예시
date_list = ['2022.01.11']

# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.ne
    .load()

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_dat
process_and_save(df, date_list, output_path)
```

```
from pyspark.sql import functions as F
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from sklearn.feature_extraction.text import TfidfVectori
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import time

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    start_time = time.time()  # 필터링 시작 시간
    result_df = df.filter(F.col('date').startswith(date)
    elapsed_time = time.time() - start_time  # 필터링 소요
    print(f"filter_by_date 실행 시간: {elapsed_time:.2f} s
    return result_df

# 개선된 코사인 유사도 분석 함수
def remove_similar_articles(df, chunk_size=100):  # chun
    start_time = time.time()  # 유사도 분석 시작 시간
    # 1. content를 TF-IDF 벡터로 변환 (sklearn 사용)
    content_list = df.select("content").rdd.flatMap(lamb
    tfidf_vectorizer = TfidfVectorizer()
    tfidf_matrix = tfidf_vectorizer.fit_transform(conter
```

```
    # 2. TF-IDF 행렬을 청크로 나눠서 분산 처리 (broadcast 사용
    tfidf_chunks = np.array_split(tfidf_matrix.toarray()
    rdd = spark.sparkContext.parallelize(tfidf_chunks)
    # 3. 각 청크에 대해 코사인 유사도 계산 후 청크별로 결과 처리
    def calculate_cosine_similarity(chunk, full_matrix):
        chunk_matrix = np.array(chunk)
        similarity_scores = cosine_similarity(chunk_matr
        return similarity_scores
    similarity_results = []
    for chunk in tfidf_chunks:
        sim_result = calculate_cosine_similarity(chunk,
        similarity_results.append(sim_result)
    # 4. 유사도가 0.9 이상인 문서 필터링
    remove_indices = set()
    for sim_chunk in similarity_results:
        for i in range(len(sim_chunk)):
            for j in range(i+1, len(sim_chunk[i])):
                if sim_chunk[i][j] >= 0.9:
                    remove_indices.add(j)
    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.co
    filtered_df = spark.createDataFrame(filtered_rows, d
    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    elapsed_time = time.time() - start_time  # 유사도 분석
    print(f"remove_similar_articles 실행 시간: {elapsed_ti
    return filtered_df, removed_count

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
    start_time = time.time()  # 저장 시작 시간
    df_with_date_only = df.withColumn('date_only', F.col
    df_with_date_only.write \
      .partitionBy("date_only") \
      .mode("overwrite") \
      .parquet(output_path)
    elapsed_time = time.time() - start_time  # 저장 소요 시
    print(f"save_partitioned_by_date 실행 시간: {elapsed_t
```

```
# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
    total_start_time = time.time()  # 전체 실행 시간 시작
    for date in date_list:
        print(f"Processing data for date: {date}")
        # 날짜별로 필터링
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이
        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique, removed_count = remove_similar_articl
        unique_count = df_unique.count()  # 중복 제거된 후
        # 날짜별로 파티셔닝하여 HDFS에 저장
        save_partitioned_by_date(df_unique, f"{output_pa
        # 처리 시간과 데이터 개수 출력
        print(f"Date: {date} - Original: {original_count
    total_elapsed_time = time.time() - total_start_time
    print(f"Total processing time: {total_elapsed_time:.

# 날짜 리스트 예시
date_list = ['2022.01.11']

# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.ne
    .load()

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_dat
process_and_save(df, date_list, output_path)
```

```
from pyspark.sql import functions as F
from pyspark.sql import Window
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import Normalizer
from pyspark.sql import Row
```

```
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import time

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    return df.filter(F.col('date').startswith(date))

# 코사인 유사도를 계산하고, 유사도가 0.9 이상인 기사를 제거하는 함수
def remove_similar_articles(df):
    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol="wo
    wordsData = tokenizer.transform(df)
    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="rawF
    featurizedData = hashingTF.transform(wordsData)
    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변환
    idf = IDF(inputCol="rawFeatures", outputCol="features")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)
    # 데이터프레임을 리스트로 변환 후 코사인 유사도 계산
    rows = rescaledData.select('features').rdd.map(lambda r
    similarity_matrix = cosine_similarity(rows)
    # 유사도 행렬을 바탕으로 유사한 기사 필터링
    remove_indices = set()
    for i in range(len(similarity_matrix)):
        for j in range(i+1, len(similarity_matrix)):
            if cosine_similarity([rows[i]], [rows[j]])[0][0
                remove_indices.add(j)
    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.colle
    # 필터링된 데이터를 데이터프레임으로 변환
    filtered_df = spark.createDataFrame(filtered_rows, df.s
    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    return filtered_df, removed_count

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
```

```
def save_partitioned_by_date(df, output_path):
    # 'date_only' 컬럼을 추가하여 날짜별 파티셔닝을 가능하게 함
    df_with_date_only = df.withColumn('date_only', F.col('d
    df_with_date_only.write \
      .partitionBy("date_only") \
      .mode("overwrite") \
      .parquet(output_path)

# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
    total_start_time = time.time()  # 전체 실행 시간 시작

    for date in date_list:
        print(f"Processing data for date: {date}")

        # 날짜별로 필터링
        start_time = time.time()  # 날짜별 처리 시간 시작
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이터 개

        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique, removed_count = remove_similar_articles(
        unique_count = df_unique.count()  # 중복 제거된 후 데이

        # 날짜별로 파티셔닝하여 HDFS에 저장
        save_partitioned_by_date(df_unique, f"{output_path}

        # 처리 시간과 데이터 개수 출력
        elapsed_time = time.time() - start_time
        print(f"Date: {date} - Original: {original_count},

    total_elapsed_time = time.time() - total_start_time
    print(f"Total processing time: {total_elapsed_time:.2f}

# 날짜 리스트 예시
date_list = ['2022.01.11']

# MongoDB로부터 데이터를 로드
```

```
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.news_
    .load()

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_date_t
process_and_save(df, date_list, output_path)
```

- 바로 위에 코드로 코사인 유사도 돌린 결과
- 약 1600개의 유사도를 측정하는데 5분 30초 가량 소요

```
Date: 2022.01.11 - Original: 1593, Unique: 1380, Removed: 213, Time: 329.80 seconds
Total processing time: 329.80 seconds
```

성능 저하의 원인?

- 아래 코드의 함수별 실행 시간을 분석해봤을때
    - filter_by_date 실행 시간: 0.06 seconds
    - remove_similar_articles 실행 시간: 305.69 seconds
    - save_partitioned_by_date 실행 시간: 18.78 seconds
    - save_single_file : 0.28seconds

```
from pyspark.sql import functions as F
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import time

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    start_time = time.time()  # 필터링 시작 시간
    result_df = df.filter(F.col('date').startswith(date))
    elapsed_time = time.time() - start_time  # 필터링 소요 시
    print(f"filter_by_date 실행 시간: {elapsed_time:.2f} seco
    return result_df

# 코사인 유사도를 계산하고, 유사도가 0.9 이상인 기사를 제거하는 함수
```

```
def remove_similar_articles(df):
    start_time = time.time()  # 유사도 분석 시작 시간

    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol="wo
    wordsData = tokenizer.transform(df)

    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="rawF
    featurizedData = hashingTF.transform(wordsData)

    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변환
    idf = IDF(inputCol="rawFeatures", outputCol="features")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)

    # 데이터프레임을 리스트로 변환 후 코사인 유사도 계산
    rows = rescaledData.select('features').rdd.map(lambda r
    similarity_matrix = cosine_similarity(rows)

    # 유사도 행렬을 바탕으로 유사한 기사 필터링
    remove_indices = set()
    for i in range(len(similarity_matrix)):
        for j in range(i+1, len(similarity_matrix)):
            if cosine_similarity([rows[i]], [rows[j]])[0][0
                remove_indices.add(j)

    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.colle
    filtered_df = spark.createDataFrame(filtered_rows, df.s
    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    elapsed_time = time.time() - start_time  # 유사도 분석 소
    print(f"remove_similar_articles 실행 시간: {elapsed_time
    return filtered_df, removed_count

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
```

```
    start_time = time.time()  # 저장 시작 시간
    df_with_date_only = df.withColumn('date_only', F.col('d
    df_with_date_only.write \
      .partitionBy("date_only") \
      .mode("overwrite") \
      .parquet(output_path)
    elapsed_time = time.time() - start_time  # 저장 소요 시간
    print(f"save_partitioned_by_date 실행 시간: {elapsed_time

# HDFS에 데이터를 단일 파일로 저장하는 함수
def save_to_single_file(df, output_path):
    start_time = time.time()  # 저장 시작 시간
    df.coalesce(1).write \
      .mode("overwrite") \
      .parquet(output_path)
    elapsed_time = time.time() - start_time  # 저장 소요 시간
    print(f"save_to_single_file 실행 시간: {elapsed_time:.2f}

# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
    total_start_time = time.time()  # 전체 실행 시간 시작
    for date in date_list:
        print(f"Processing data for date: {date}")
        # 날짜별로 필터링
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이터 개
        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique, removed_count = remove_similar_articles(
        unique_count = df_unique.count()  #3중복 제거된 후 데이
        # 날짜별로 파티셔닝하여 HDFS에 저장
        save_to_single_file(df_unique, f"{output_path}/{dat
        # 처리 시간과 데이터 개수 출력
        print(f"Date: {date} - Original: {original_count},
    total_elapsed_time = time.time() - total_start_time
    print(f"Total processing time: {total_elapsed_time:.2f}

# 날짜 리스트 예시
date_list = ['2022.01.11']
```

```
# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.news_
    .load()

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_date_t
process_and_save(df, date_list, output_path)
```

- remove_similar_articles 실행 시간이 305.69 seconds로 압도적인 부분을 차지 중인데..
- 그럼 remove_similar_articles 함수 내에서도 어떠한 작업이 그렇게 오래 걸리는지 알아보자
- ↓↓↓↓ remove_similar_articles 함수 내 각 작업들의 실행시간 책정

```
from pyspark.sql import functions as F
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import time

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    start_time = time.time()  # 필터링 시작 시간
    result_df = df.filter(F.col('date').startswith(date))
    elapsed_time = time.time() - start_time  # 필터링 소요 시
    print(f"filter_by_date 실행 시간: {elapsed_time:.2f} seco
    return result_df

# 코사인 유사도를 계산하고, 유사도가 0.9 이상인 기사를 제거하는 함수
def remove_similar_articles(df):
    start_time = time.time()  # 유사도 분석 시작 시간

    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer_start_time = time.time()
```

```
    tokenizer = Tokenizer(inputCol="content", outputCol="wo
    wordsData = tokenizer.transform(df)
    tokenizer_elapsed_time = time.time() - tokenizer_start_
    print(f"Tokenizer 실행 시간: {tokenizer_elapsed_time:.2f}

    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF_start_time = time.time()
    hashingTF = HashingTF(inputCol="words", outputCol="rawF
    featurizedData = hashingTF.transform(wordsData)
    hashingTF_elapsed_time = time.time() - hashingTF_start_
    print(f"HashingTF 실행 시간: {hashingTF_elapsed_time:.2f}

    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변환
    idf_start_time = time.time()
    idf = IDF(inputCol="rawFeatures", outputCol="features")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)
    idf_elapsed_time = time.time() - idf_start_time
    print(f"IDF 실행 시간: {idf_elapsed_time:.2f} seconds")

    # 데이터프레임을 리스트로 변환 후 코사인 유사도 계산
    cosine_start_time = time.time()
    rows = rescaledData.select('features').rdd.map(lambda r
    similarity_matrix = cosine_similarity(rows)
    cosine_elapsed_time = time.time() - cosine_start_time
    print(f"코사인 유사도 계산 실행 시간: {cosine_elapsed_time:.

    # 유사도 행렬을 바탕으로 유사한 기사 필터링
    remove_indices = set()
    filtering_start_time = time.time()
    for i in range(len(similarity_matrix)):
        for j in range(i + 1, len(similarity_matrix)):
            if cosine_similarity([rows[i]], [rows[j]])[0][0
                remove_indices.add(j)
    filtering_elapsed_time = time.time() - filtering_start_
    print(f"유사한 기사 필터링 실행 시간: {filtering_elapsed_tim

    # 중복되지 않은 데이터만 남기기
```

```
    filtered_rows = [row for idx, row in enumerate(df.colle
    filtered_df = spark.createDataFrame(filtered_rows, df.s

    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    elapsed_time = time.time() - start_time  # 유사도 분석 소
    print(f"remove_similar_articles 전체 실행 시간: {elapsed_
    return filtered_df, removed_count


# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
    start_time = time.time()  # 저장 시작 시간
    df_with_date_only = df.withColumn('date_only', F.col('d
    df_with_date_only.write \
      .partitionBy("date_only") \
      .mode("overwrite") \
      .parquet(output_path)
    elapsed_time = time.time() - start_time  # 저장 소요 시간
    print(f"save_partitioned_by_date 실행 시간: {elapsed_time

# HDFS에 데이터를 단일 파일로 저장하는 함수
def save_to_single_file(df, output_path):
    start_time = time.time()  # 저장 시작 시간
    df.coalesce(1).write \
      .mode("overwrite") \
      .parquet(output_path)
    elapsed_time = time.time() - start_time  # 저장 소요 시간
    print(f"save_to_single_file 실행 시간: {elapsed_time:.2f}

# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
    total_start_time = time.time()  # 전체 실행 시간 시작
    for date in date_list:
        print(f"Processing data for date: {date}")
        # 날짜별로 필터링
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이터 개
```

```
            # 코사인 유사도를 기반으로 중복 기사 제거
            df_unique, removed_count = remove_similar_articles(
            unique_count = df_unique.count()  #3중복 제거된 후 데이
            # 날짜별로 파티셔닝하여 HDFS에 저장
            save_to_single_file(df_unique, f"{output_path}/{dat
            # 처리 시간과 데이터 개수 출력
            print(f"Date: {date} - Original: {original_count},
        total_elapsed_time = time.time() - total_start_time
        print(f"Total processing time: {total_elapsed_time:.2f}

# 날짜 리스트 예시
date_list = ['2022.01.11']

# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.news_
    .load()

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_date_t
process_and_save(df, date_list, output_path)
```

실행시간을 보면

- Tokenizer 실행 시간: 0.04 seconds
- HashingTF 실행 시간: 0.02 seconds
- IDF 실행 시간: 0.51 seconds
- 코사인 유사도 계산 실행 시간: 1.38 seconds
- 유사한 기사 필터링 실행 시간: 313.40 seconds

개선방안 1 : for loop break

- 유사도 임계점을 넘었음에도 모든 기사에 대해 검사를 진행하고 있었음
    - continue, break문 추가
    - 동일 데이터 동일 임계점에 대해 313.40 seconds → 266.75 seconds 개선 성공

```
# 유사도 행렬을 바탕으로 유사한 기사 필터링
for i in range(len(similarity_matrix)):
  if i in remove_indices:
    continue
    for j in range(i + 1, len(similarity_matrix)):
        if cosine_similarity([rows[i]], [rows[j]])[0][0] >=
            remove_indices.add(j)
            break
```

▼ 특정 날짜를 설정하여 데이터 전처리를 진행하는 과정

```
from pyspark.sql import functions as F
from datetime import datetime, timedelta
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import time

# "본문 없음" 또는 "Error" 데이터를 필터링하는 함수
def filter_invalid_content(df):
    start_time = time.time()  # 시작 시간 측정

    # "content" 컬럼에서 '본문 없음' 또는 'Error'가 포함된 행을 제
    original_count = df.count()  # 필터링 전 데이터 개수
    filtered_df = df.filter(~(F.col("content").contains("본
    filtered_count = filtered_df.count()  # 필터링 후 데이터 개

    removed_count = original_count - filtered_count  # 제거된
    elapsed_time = time.time() - start_time  # 필터링 소요 시

    # 결과 출력
    print(f"filter_invalid_content 실행 시간: {elapsed_time:
    print(f"삭제된 데이터 개수: {removed_count}")

    return filtered_df

# 날짜별로 데이터를 필터링하는 함수
```

```
def filter_by_date(df, date):
    start_time = time.time()  # 필터링 시작 시간
    result_df = df.filter(F.col('date').startswith(date))
    elapsed_time = time.time() - start_time  # 필터링 소요 시
    print(f"filter_by_date 실행 시간: {elapsed_time:.2f} seco
    return result_df

# 코사인 유사도를 계산하고, 유사도가 0.7 이상인 기사를 제거하는 함수
def remove_similar_articles(df):
    start_time = time.time()  # 유사도 분석 시작 시간

    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer_start_time = time.time()
    tokenizer = Tokenizer(inputCol="content", outputCol="wo
    wordsData = tokenizer.transform(df)
    tokenizer_elapsed_time = time.time() - tokenizer_start_
    print(f"Tokenizer 실행 시간: {tokenizer_elapsed_time:.2f}

    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF_start_time = time.time()
    hashingTF = HashingTF(inputCol="words", outputCol="rawF
    featurizedData = hashingTF.transform(wordsData)
    hashingTF_elapsed_time = time.time() - hashingTF_start_
    print(f"HashingTF 실행 시간: {hashingTF_elapsed_time:.2f}

    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변환
    idf_start_time = time.time()
    idf = IDF(inputCol="rawFeatures", outputCol="features")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)
    idf_elapsed_time = time.time() - idf_start_time
    print(f"IDF 실행 시간: {idf_elapsed_time:.2f} seconds")

    # 데이터프레임을 리스트로 변환 후 코사인 유사도 계산
    cosine_start_time = time.time()
    rows = rescaledData.select('features').rdd.map(lambda r
    similarity_matrix = cosine_similarity(rows)
    cosine_elapsed_time = time.time() - cosine_start_time
```

```
    print(f"코사인 유사도 계산 실행 시간: {cosine_elapsed_time:.

    # 유사도 행렬을 바탕으로 유사한 기사 필터링
    remove_indices = set()
    filtering_start_time = time.time()
    for i in range(len(similarity_matrix)):
        if i in remove_indices:
            continue
        for j in range(i + 1, len(similarity_matrix)):
            if cosine_similarity([rows[i]], [rows[j]])[0][0
                remove_indices.add(j)
                break
    filtering_elapsed_time = time.time() - filtering_start_
    print(f"유사한 기사 필터링 실행 시간: {filtering_elapsed_tim

    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.colle
    filtered_df = spark.createDataFrame(filtered_rows, df.s

    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    elapsed_time = time.time() - start_time  # 유사도 분석 소
    print(f"remove_similar_articles 전체 실행 시간: {elapsed_

    return filtered_df, removed_count

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
    start_time = time.time()  # 저장 시작 시간
    df_with_date_only = df.withColumn('date_only', F.col('d
    df_with_date_only.write \
        .partitionBy("date_only") \
        .mode("overwrite") \
        .parquet(output_path)
    elapsed_time = time.time() - start_time  # 저장 소요 시간
    print(f"save_partitioned_by_date 실행 시간: {elapsed_time

# HDFS에 데이터를 단일 파일로 저장하는 함수
```

```
def save_to_single_file(df, output_path):
    start_time = time.time()  # 저장 시작 시간
    df.coalesce(1).write \
        .mode("overwrite") \
        .parquet(output_path)
    elapsed_time = time.time() - start_time  # 저장 소요 시간
    print(f"save_to_single_file 실행 시간: {elapsed_time:.2f}

# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
    total_start_time = time.time()  # 전체 실행 시간 시작
    for date in date_list:
        print(f"Processing data for date: {date}")
        # 날짜별로 필터링
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이터 개

        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique, removed_count = remove_similar_articles(
        unique_count = df_unique.count()  # 중복 제거된 후 데이

        # 날짜별로 파티셔닝하여 HDFS에 저장
        save_to_single_file(df_unique, f"{output_path}/{dat

        # 처리 시간과 데이터 개수 출력
        print(f"Date: {date} - Original: {original_count},

    total_elapsed_time = time.time() - total_start_time
    print(f"Total processing time: {total_elapsed_time:.2f}

# 날짜 리스트 생성 함수
def generate_date_range(start_date, end_date):
    date_list = []
    current_date = start_date
    while current_date <= end_date:
        date_list.append(current_date.strftime('%Y.%m.%d'))
        current_date += timedelta(days=1)  # 하루씩 증가
    return date_list
```

```
# 시작 날짜 및 종료 날짜 설정
start_date = datetime(2022, 1, 11)
end_date = datetime(2022, 1, 12)

# 날짜 리스트 예시
date_list = generate_date_range(start_date, end_date)

# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.news_
    .load()

# 필터링 된 데이터
df_filtered = filter_invalid_content(df)

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_date_t
process_and_save(df_filtered, date_list, output_path)
```

```
from pyspark.sql import functions as F
from datetime import datetime, timedelta
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from sklearn.metrics.pairwise import cosine_similarity
from pyspark.sql import DataFrame
import numpy as np
import time

# "본문 없음" 또는 "Error" 데이터를 필터링하는 함수
def filter_invalid_content(df):
    start_time = time.time()  # 시작 시간 측정

    # "content" 컬럼에서 '본문 없음' 또는 'Error'가 포함된 행을 제
```

```
    original_count = df.count()  # 필터링 전 데이터 개수
    filtered_df = df.filter(~(F.col("content").contains("본
    filtered_count = filtered_df.count()  # 필터링 후 데이터 개

    removed_count = original_count - filtered_count  # 제거된
    elapsed_time = time.time() - start_time  # 필터링 소요 시

    # 결과 출력
    print(f"filter_invalid_content 실행 시간: {elapsed_time:
    print(f"삭제된 데이터 개수: {removed_count}")

    return filtered_df

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    start_time = time.time()  # 필터링 시작 시간
    result_df = df.filter(F.col('date').startswith(date))
    elapsed_time = time.time() - start_time  # 필터링 소요 시
    print(f"filter_by_date 실행 시간: {elapsed_time:.2f} seco
    return result_df

# 코사인 유사도를 계산하고, 유사도가 0.7 이상인 기사를 제거하는 함수
def remove_similar_articles(df: DataFrame) -> (DataFrame, i
    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol="wo
    wordsData = tokenizer.transform(df)
    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="rawF
    featurizedData = hashingTF.transform(wordsData)
    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변환
    idf = IDF(inputCol="rawFeatures", outputCol="features")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)

    # 벡터를 RDD로 변환
    feature_vectors = rescaledData.select('features').rdd.m
    # 전체 유사도 매트릭스 계산
    similarity_matrix = cosine_similarity(feature_vectors)
```

```
    # 유사한 기사 필터링
    remove_indices = set()
    for i in range(len(similarity_matrix)):
        if i in remove_indices:
            continue
        for j in range(i + 1, len(similarity_matrix)):
            if similarity_matrix[i][j] >= 0.7:
                remove_indices.add(j)
                break
    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.colle
    filtered_df = spark.createDataFrame(filtered_rows, df.s
    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    return filtered_df, removed_count

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
    start_time = time.time()  # 저장 시작 시간
    df_with_date_only = df.withColumn('date_only', F.col('d
    df_with_date_only.write \
        .partitionBy("date_only") \
        .mode("overwrite") \
        .parquet(output_path)
    elapsed_time = time.time() - start_time  # 저장 소요 시간
    print(f"save_partitioned_by_date 실행 시간: {elapsed_time

# HDFS에 데이터를 단일 파일로 저장하는 함수
def save_to_single_file(df, output_path):
    start_time = time.time()  # 저장 시작 시간
    df.coalesce(1).write \
        .mode("overwrite") \
        .parquet(output_path)
    elapsed_time = time.time() - start_time  # 저장 소요 시간
    print(f"save_to_single_file 실행 시간: {elapsed_time:.2f}

# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
```

```
    total_start_time = time.time()  # 전체 실행 시간 시작
    for date in date_list:
        print(f"Processing data for date: {date}")
        # 날짜별로 필터링
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이터 개

        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique, removed_count = remove_similar_articles(
        unique_count = df_unique.count()  # 중복 제거된 후 데이

        # 날짜별로 파티셔닝하여 HDFS에 저장
        save_to_single_file(df_unique, f"{output_path}/{dat

        # 처리 시간과 데이터 개수 출력
        print(f"Date: {date} - Original: {original_count},

    total_elapsed_time = time.time() - total_start_time
    print(f"Total processing time: {total_elapsed_time:.2f}

# 날짜 리스트 생성 함수
def generate_date_range(start_date, end_date):
    date_list = []
    current_date = start_date
    while current_date <= end_date:
        date_list.append(current_date.strftime('%Y.%m.%d'))
        current_date += timedelta(days=1)  # 하루씩 증가
    return date_list

# 시작 날짜 및 종료 날짜 설정
start_date = datetime(2022, 1, 11)
end_date = datetime(2022, 1, 12)

# 날짜 리스트 예시
date_list = generate_date_range(start_date, end_date)

# MongoDB로부터 데이터를 로드
df = spark.read \
```

```
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.news_
    .load()

# 필터링 된 데이터
df_filtered = filter_invalid_content(df)

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_date_t
process_and_save(df_filtered, date_list, output_path)
```

```
# 코사인 유사도를 계산하고, 유사도가 0.7 이상인 기사를 제거하는 함수
def remove_similar_articles(df: DataFrame) -> (DataFrame, i
    start_time = time.time()  # 유사도 분석 시작 시간
    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol="wo
    wordsData = tokenizer.transform(df)
    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="rawF
    featurizedData = hashingTF.transform(wordsData)
    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변환
    idf = IDF(inputCol="rawFeatures", outputCol="features")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)
    # 데이터프레임을 RDD로 변환 후 유사도 계산
    def compute_similarity(rows):
        from sklearn.metrics.pairwise import cosine_similar
        # 코사인 유사도 계산
        return cosine_similarity(rows)
    # 벡터를 RDD로 변환
    feature_vectors = rescaledData.select('features').rdd.m
    # 전체 유사도 매트릭스 계산
    similarity_matrix = compute_similarity(feature_vectors)
    # 유사한 기사 필터링
    remove_indices = set()
    for i in range(len(similarity_matrix)):
```

```
            if i in remove_indices:
                continue
            for j in range(i + 1, len(similarity_matrix)):
                if similarity_matrix[i][j] >= 0.7:
                    remove_indices.add(j)
                    break
        # 중복되지 않은 데이터만 남기기
        filtered_rows = [row for idx, row in enumerate(df.colle
        filtered_df = spark.createDataFrame(filtered_rows, df.s
        # 중복 처리된 데이터 개수
        removed_count = len(remove_indices)
        elapsed_time = time.time() - start_time  # 유사도 분석 소
        print(f"remove_similar_articles 전체 실행 시간: {elapsed_
        return filtered_df, removed_count

# 코사인 유사도를 계산하고, 유사도가 0.7 이상인 기사를 제거하는 함수
def remove_similar_articles(df):
    start_time = time.time()  # 유사도 분석 시작 시간

    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer_start_time = time.time()
    tokenizer = Tokenizer(inputCol="content", outputCol="wo
    wordsData = tokenizer.transform(df)
    tokenizer_elapsed_time = time.time() - tokenizer_start_
    print(f"Tokenizer 실행 시간: {tokenizer_elapsed_time:.2f

    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF_start_time = time.time()
    hashingTF = HashingTF(inputCol="words", outputCol="rawF
    featurizedData = hashingTF.transform(wordsData)
    hashingTF_elapsed_time = time.time() - hashingTF_start_
    print(f"HashingTF 실행 시간: {hashingTF_elapsed_time:.2f

    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변환
    idf_start_time = time.time()
    idf = IDF(inputCol="rawFeatures", outputCol="features")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)
```

```
    idf_elapsed_time = time.time() - idf_start_time
    print(f"IDF 실행 시간: {idf_elapsed_time:.2f} seconds")

    # 데이터프레임을 리스트로 변환 후 코사인 유사도 계산
    cosine_start_time = time.time()
    rows = rescaledData.select('features').rdd.map(lambda r
    similarity_matrix = cosine_similarity(rows)
    cosine_elapsed_time = time.time() - cosine_start_time
    print(f"코사인 유사도 계산 실행 시간: {cosine_elapsed_time:.

    # 유사도 행렬을 바탕으로 유사한 기사 필터링
    remove_indices = set()
    filtering_start_time = time.time()
    for i in range(len(similarity_matrix)):
        if i in remove_indices:
            continue
        for j in range(i + 1, len(similarity_matrix)):
            if cosine_similarity([rows[i]], [rows[j]])[0][0
                remove_indices.add(j)
                break
    filtering_elapsed_time = time.time() - filtering_start_
    print(f"유사한 기사 필터링 실행 시간: {filtering_elapsed_tim

    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.colle
    filtered_df = spark.createDataFrame(filtered_rows, df.s

    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    elapsed_time = time.time() - start_time  # 유사도 분석 소
    print(f"remove_similar_articles 전체 실행 시간: {elapsed_

    return filtered_df, removed_count
```

▼ 최종 데이터 전처리 코드(이걸로 전처리 진행하면 됨)

```
Date: 2022.01.11 - Original: 1629, Unique: 1386, Removed: 243
Total processing time: 3.66 seconds
```

```
# at nameNode
$ docker restart nameNode
$ docker attach nameNode

# at dataNode
$ docker restart dataNode
$ docker attach dataNode

# at mongo-container
$ docker restart mongo-container
$ docker exec -it mongo-container bash

# at nameNode, at dataNode
$ service ssh restart
$ service ssh status # sshd is running

# at nameNode
$ start-all.sh

# at nameNode, at dataNode
$ jps

# at dataNode
# ※db명과 collection명은 서버 nosql환경에 맞춰 변경 필요
$ pyspark --conf "spark.mongodb.read.connection.uri=mongodb

# 이후 pyspark에서 아래 코드 실행
```

```
from pyspark.sql import functions as F
from datetime import datetime, timedelta
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from sklearn.metrics.pairwise import cosine_similarity
from pyspark.sql import DataFrame
import numpy as np
import time

# "본문 없음" 또는 "Error" 데이터를 필터링하는 함수
```

```
def filter_invalid_content(df):
    # "content" 컬럼에서 '본문 없음' 또는 'Error'가 포함된 행을 제
    original_count = df.count()  # 필터링 전 데이터 개수
    filtered_df = df.filter(~(F.col("content").contains("본
    filtered_count = filtered_df.count()  # 필터링 후 데이터 개

    removed_count = original_count - filtered_count  # 제거된

    # 결과 출력
    print(f"삭제된 데이터 개수: {removed_count}")

    return filtered_df

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    result_df = df.filter(F.col('date').startswith(date))
    return result_df

# 코사인 유사도를 계산하고, 유사도가 0.7 이상인 기사를 제거하는 함수
def remove_similar_articles(df: DataFrame) -> (DataFrame, i
    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol="wo
    wordsData = tokenizer.transform(df)
    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="rawF
    featurizedData = hashingTF.transform(wordsData)
    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변환
    idf = IDF(inputCol="rawFeatures", outputCol="features")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)
    # 데이터프레임을 RDD로 변환 후 유사도 계산
    def compute_similarity(rows):
        from sklearn.metrics.pairwise import cosine_similar
        # 코사인 유사도 계산
        return cosine_similarity(rows)
    # 벡터를 RDD로 변환
    feature_vectors = rescaledData.select('features').rdd.m
    # 전체 유사도 매트릭스 계산
```

```
    similarity_matrix = compute_similarity(feature_vectors)
    # 유사한 기사 필터링
    remove_indices = set()
    for i in range(len(similarity_matrix)):
        if i in remove_indices:
            continue
        for j in range(i + 1, len(similarity_matrix)):
            if similarity_matrix[i][j] >= 0.9:
                remove_indices.add(j)
                break
    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.colle
    filtered_df = spark.createDataFrame(filtered_rows, df.s
    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    return filtered_df, removed_count

# HDFS에 날짜별로 파티셔닝하여 저장하는 함수
def save_partitioned_by_date(df, output_path):
    df_with_date_only = df.withColumn('date_only', F.col('d
    df_with_date_only.write \
        .partitionBy("date_only") \
        .mode("overwrite") \
        .parquet(output_path)

# HDFS에 데이터를 단일 파일로 저장하는 함수
def save_to_single_file(df, output_path, date):
    year = date.split(".")[0]  # 연도 추출
    full_path = f"{output_path}/{year}/{date}"  # 연도와 날짜
    df.coalesce(1).write \
        .mode("overwrite") \
        .parquet(full_path)

# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
    for date in date_list:
        print(f"Processing data for date: {date}")
        # 날짜별로 필터링
```

```
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이터 개

        # df_filtered가 비어있지 않은 경우에만 처리
        if original_count > 0:
            # 코사인 유사도를 기반으로 중복 기사 제거
            df_unique, removed_count = remove_similar_artic
            unique_count = df_unique.count()  # 중복 제거된 후

            # 날짜별로 단일 파일로 HDFS에 저장
            save_to_single_file(df_unique, output_path, dat


# 날짜 리스트 생성 함수
def generate_date_range(start_date, end_date):
    date_list = []
    current_date = start_date
    while current_date <= end_date:
        date_list.append(current_date.strftime('%Y.%m.%d'))
        current_date += timedelta(days=1)  # 하루씩 증가
    return date_list

# 시작 날짜 및 종료 날짜 설정
start_date = datetime(2024, 10, 11)
end_date = datetime(2024, 10, 11)

# 날짜 리스트 예시
date_list = generate_date_range(start_date, end_date)

# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.news_
    .load()

# 필터링 된 데이터
df_filtered = filter_invalid_content(df)
```

```
# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/news_preprocessing_result"
process_and_save(df_filtered, date_list, output_path)
```

▼ 트러블 슈팅 : 비정상적으로 오래 걸리던 코사인 유사도검사 과정 해결

↓↓↓ 포스팅 작성

https://velog.io/@sonbaejun/pyspark에서-비정상적으로-오래-걸리던-코사인-유사도검사-과정-해결

문제 개요

- 뉴스 데이터의 본문 내용에 대해서 코사인 유사도를 계산하여 특정 임계값을 넘는 기사를 "중복 기사"로 간주하고 제거하는 전처리 과정 진행
- 하루 기준 약 1600개의 뉴스 데이터에 대해 비정상적으로 긴 시간이 걸리는 문제가 발생
- 약 350만개에서 많게는 1000만개의 데이터를 전처리하여야 했기때문에, 해당 실행 시간으로는 데이터 전처리에 차질이 발생

```
Date: 2022.01.11 - Original: 1593, Unique: 1380, Removed: 213, Time: 329.80 seconds
Total processing time: 329.80 seconds
```

문제 원인

- 코사인 유사도 검사를 반복해서 호출하는 것이 문제
- 기존 코드

```
# 코사인 유사도를 계산하고, 유사도가 0.7 이상인 기사를 제거하는 함수
def remove_similar_articles(df):
    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol="words")
    wordsData = tokenizer.transform(df)

    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=10000)
    featurizedData = hashingTF.transform(wordsData)
```

```
    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로
변환
    idf = IDF(inputCol="rawFeatures", outputCol="feat
ures")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)

    # 데이터프레임을 리스트로 변환 후 코사인 유사도 계산
    rows = rescaledData.select('features').rdd.map(la
mbda row: row['features'].toArray()).collect()

    # 유사도 행렬을 바탕으로 유사한 기사 필터링
    remove_indices = set()
    for i in range(len(similarity_matrix)):
        if i in remove_indices:
            continue
        for j in range(i + 1, len(similarity_matri
x)):
            if cosine_similarity([rows[i]], [rows
[j]])[0][0] >= 0.7:
                remove_indices.add(j)
                break

    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(d
f.collect()) if idx not in remove_indices]
    filtered_df = spark.createDataFrame(filtered_row
s, df.schema)

    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)

    return filtered_df, removed_count
```

```
# 데이터프레임을 리스트로 변환 후 코사인 유사도 계산
rows = rescaledData.select('features').rdd.map(lambda
```

```
row: row['features'].toArray()).collect()

# 유사도 행렬을 바탕으로 유사한 기사 필터링
remove_indices = set()
for i in range(len(similarity_matrix)):
    if i in remove_indices:
        continue
    for j in range(i + 1, len(similarity_matrix)):
        if cosine_similarity([rows[i]], [rows[j]])[0]
[0] >= 0.7:
            remove_indices.add(j)
            break
```

- 코드를 보면 해당 부분에서 반복적으로 cosine_similarity를 호출
- 1600개의 뉴스 기사에 대해 **1600×1600=2,560,000 번의 유사도 계산**이 필요.
- 심지어 벡터 하나하나가 10,000차원의 TF-IDF 벡터라면 계산이 매우 부담스러워진다.

우선 명확히 어디서 오버헤드가 발생하는지 알기위하여 각 함수에 대해 실행시간을 측정하였다.

- 측정 결과, remove_similar_articles 실행 시간이 305.69 seconds로 압도적인 부분을 차지중인데..
- 그럼 remove_similar_articles 함수 내에서도 어떠한 작업이 그렇게 오래 걸리는지 알아보자

역시 remove_similar_articles 함수 내 각 과정의 실행시간을 측정하였다.

- Tokenizer 실행 시간: 0.04 seconds
- HashingTF 실행 시간: 0.02 seconds
- IDF 실행 시간: 0.51 seconds
- 코사인 유사도 계산 실행 시간: 1.38 seconds
- 유사한 기사 필터링 실행 시간: 301.40 seconds

보다시피 이중 for loop에서 거의 모든 시간을 독식하고 있었다.
아마, 모든 데이터에 대해서 코사인 유사도 검사를 진행하고 있는데, 그 작업이 O(n제곱)의 작업이라 이러한 시간이 나왔던거 같다.

1차적인 간단한 성능개선 방안으로는

```
# 유사도 행렬을 바탕으로 유사한 기사 필터링
for i in range(len(similarity_matrix)):
  if i in remove_indices:
    continue
    for j in range(i + 1, len(similarity_matrix)):
        if cosine_similarity([rows[i]], [rows[j]])[0][0]
>= 0.7:
            remove_indices.add(j)
            break
```

이렇게 이미 유사도검사 상 "중복"으로 판단되었다면 loop를 탈출하도록 하였다.

```
Date: 2022.01.11 - Original: 1629, Unique: 1322, Remove
d: 307
Total processing time: 266.75 seconds
```

그 결과, 다음과 같이 어느정도의 시간 단축에는 성공하였다. 하지만 비율을 보면 알수있다싶이, 중복 기사의 비율이 그리 많지 않기때문에 드라마틱하게 소요시간이 개선되진 않았다.

근본적인 원인인, 많은 데이터 수에 대해 일일히 코사인 유사도 검사를 진행하는 로직 자체를 해결하여야 했다.

해결방안

역시, 해결방안은 for loop 자체의 코사인 유사도를 반복해서 진행하는 과정을 없애는 것이었다.

```
# 벡터를 RDD로 변환
feature_vectors = rescaledData.select('features').rdd.ma
p(lambda row: row.features.toArray()).collect()

# 전체 유사도 매트릭스 계산
similarity_matrix = cosine_similarity(feature_vectors)

# 유사한 기사 필터링
remove_indices = set()
for i in range(len(similarity_matrix)):
```

```
    if i in remove_indices:
        continue
    for j in range(i + 1, len(similarity_matrix)):
        if similarity_matrix[i][j] >= 0.7:
            remove_indices.add(j)
            break
```

- 코사인 유사도 검사를 한번에 진행
- RDD로 변환된 데이터를 통해 코사인 유사도 검사를 한번에 진행함으로써 코사인 유사도 처리에 대해 병렬처리 진행.
- spark의 특성인 RDD를 이용한 병렬처리로 대량의 데이터에 대해 빠른 코사인 유사도 검사 처리 완료
- 결론적으로 329.80초가 걸리던 작업을 3.66초로 98.89%의 성능 향상

```
Date: 2022.01.11 - Original: 1629, Unique: 1386, Removed: 243
Total processing time: 3.66 seconds
```

배운점

Spark의 동작원리에 대해서 공부할 수 있었다.
Spark의 핵심은
분산과 **병렬처리**에 있다고 생각하는데, 그 부분을 구현해내는 RDD, 또는 그 이외에도 성능적으로 도움을 주는 Lazy Evaluation 등의 개념에 대해 공부할 수 있었다.

▼ Spark를 이용한 병렬처리로 데이터 전처리 과정을 빠르게 해결하기

↓↓↓ 포스팅 작성

https://velog.io/@sonbaejun/Spark를-이용한-병렬처리로-데이터-전처리-과정을-빠르게-해결하기

개요

프로젝트에서 뉴스 데이터 처리를 담당하게 되었다.

우리 팀의 크롤링 데이터 목표는 **1200만개이고, 지금도 400만개 정도의 뉴스 데이터를 크롤링하며 계속 데이터를 열심히 모으고 있다.**

데이터 숫자가 꽤나 커졌기 때문에, 빅데이터 기술을 적용할 필요가 있다고 느꼈다.

따라서 데이터를 Hadoop과 Spark를 이용하여 처리하기로 결정하였다. 이 포스팅에서는 해당 과정의 전처리 과정에 대해 서술하겠다.

Hadoop과 Spark 환경구축

컨테이너 기반 분산 환경 구축을 통한 협업 효율성 극대화

결론적으로, Docker를 이용해 Hadoop과 Spark, NoSQL를 각각의 컨테이너로 구성하고 각 컨테이너의 환경구축을 완료했다.

https://velog.io/@sonbaejun/Hadoop과-Spark의-테스트-환경을-로컬이-아닌-Docker로-구축한-이유

이러한 방식을 결정한 이유는 위에서 자세히 서술하였다.

성능개선을 위한 고민

```
from pyspark.sql import functions as F
from datetime import datetime, timedelta
from pyspark.ml.feature import Tokenizer, HashingTF, IDF
from sklearn.metrics.pairwise import cosine_similarity
from pyspark.sql import DataFrame
import numpy as np
import time

# "본문 없음" 또는 "Error" 데이터를 필터링하는 함수
def filter_invalid_content(df):
    # "content" 컬럼에서 '본문 없음' 또는 'Error'가 포함된 행을 제
    filtered_df = df.filter(~(F.col("content").contains("본
    filtered_count = filtered_df.count()  # 필터링 후 데이터 개
```

```
    return filtered_df

# 날짜별로 데이터를 필터링하는 함수
def filter_by_date(df, date):
    result_df = df.filter(F.col('date').startswith(date))
    return result_df

# 코사인 유사도를 계산하고, 유사도가 0.7 이상인 기사를 제거하는 함수
def remove_similar_articles(df: DataFrame) -> (DataFrame, i
    # Tokenizer를 이용해 content 컬럼을 토큰화
    tokenizer = Tokenizer(inputCol="content", outputCol="wo
    wordsData = tokenizer.transform(df)

    # HashingTF로 각 단어를 TF(Term Frequency)로 변환
    hashingTF = HashingTF(inputCol="words", outputCol="rawF
    featurizedData = hashingTF.transform(wordsData)

    # IDF(Inverse Document Frequency)를 이용해 TF-IDF로 변환
    idf = IDF(inputCol="rawFeatures", outputCol="features")
    idfModel = idf.fit(featurizedData)
    rescaledData = idfModel.transform(featurizedData)

    # 벡터를 RDD로 변환
    feature_vectors = rescaledData.select('features').rdd.m

    # 전체 유사도 매트릭스 계산
    similarity_matrix = cosine_similarity(feature_vectors)

    # 유사한 기사 필터링
    remove_indices = set()
    for i in range(len(similarity_matrix)):
        if i in remove_indices:
            continue
        for j in range(i + 1, len(similarity_matrix)):
            if similarity_matrix[i][j] >= 0.7:
                remove_indices.add(j)
                break
```

```
    # 중복되지 않은 데이터만 남기기
    filtered_rows = [row for idx, row in enumerate(df.colle
    filtered_df = spark.createDataFrame(filtered_rows, df.s

    # 중복 처리된 데이터 개수
    removed_count = len(remove_indices)
    return filtered_df, removed_count

# HDFS에 날짜별로 파티셔닝하여 개별 파일로 저장하는 함수
def save_partitioned_by_date(df, output_path):
    df_with_date_only = df.withColumn('date_only', F.col('d
    df_with_date_only.write \
        .partitionBy("date_only") \
        .mode("overwrite") \
        .parquet(output_path)

# HDFS에 날짜별로 파티셔닝하여 단일 파일로 저장하는 함수
def save_to_single_file(df, output_path):
    df.coalesce(1).write \
        .mode("overwrite") \
        .parquet(output_path)

# 전체 로직 - 실행 시간과 데이터 개수를 출력
def process_and_save(df, date_list, output_path):
    for date in date_list:
        print(f"Processing data for date: {date}")
        # 날짜별로 필터링
        df_filtered = filter_by_date(df, date)
        original_count = df_filtered.count()  # 원본 데이터 개

        # 코사인 유사도를 기반으로 중복 기사 제거
        df_unique, removed_count = remove_similar_articles(
        unique_count = df_unique.count()  # 중복 제거된 후 데이

        # 날짜별로 파티셔닝하여 HDFS에 저장
        save_to_single_file(df_unique, f"{output_path}/{dat

# 날짜 리스트 생성 함수
```

```
def generate_date_range(start_date, end_date):
    date_list = []
    current_date = start_date
    while current_date <= end_date:
        date_list.append(current_date.strftime('%Y.%m.%d'))
        current_date += timedelta(days=1)
    return date_list

# 시작 날짜 및 종료 날짜 설정
start_date = datetime(2022, 1, 11)
end_date = datetime(2022, 1, 11)

# 날짜 리스트 생성
date_list = generate_date_range(start_date, end_date)

# MongoDB로부터 데이터를 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.news_
    .load()

# "본문 없음"과 "Error" Content가 필터링 된 데이터
df_filtered = filter_invalid_content(df)

# 날짜별로 데이터를 처리한 후 HDFS에 저장
output_path = "/user/baejun/news_data_partitioned_by_date"
process_and_save(df_filtered, date_list, output_path)
```

전처리 최종 코드이다.

당연한거지만, 크롤링된 4~500만개의 데이터를 그냥 전처리 작업을 한다면, 어마어마한 시간과 공간이 소모될 것이다.

특히, 전처리 과정에서 코사인 유사도 분석을 통해 중복 기사를 제거해야되기 되는데, 이 연산이 그리 가볍지만은 않기 때문에 성능적으로 효율적인 방안을 찾아야만 했다. 그리고 지금부터는 그 고민의 결과들을 서술하려한다.

1. 날짜별로 필터링하여 전처리 작업 후 HDFS에 날짜별로 파티셔닝하여 저장하기.

당연하지만, 데이터의 크기가 클 수록 **분산하여 처리하여야 한다.**

우리 서비스는 **특정 시점에 대한 주식 정보와 뉴스 정보를 제공하는 서비스이다.**

따라서 날짜별로 뉴스 데이터를 조회 할 일이 많기 때문에

데이터 전처리 및 HDFS에 적재하는 것 또한 날짜별로 파티셔닝 하여 적용하기로 결정하였다.

Spark의 **파티셔닝** 기능을 활용하여 HDFS에 데이터를 저장하여 추후 **데이터 분석 및 처리에 있어 빠른 접근**을 가능하게 만들었다. 특히 날짜별로 데이터를 나누어 저장하였기 때문에 특정 날짜에 해당하는 뉴스 데이터를 빠르게 불러올 수 있도록 하였다.

이 코드는 `df_with_date_only.write.partitionBy("date_only")`를 통해 데이터를 파티셔닝하여 HDFS에 저장하는 구조로 되어 있어, **데이터 저장과 관리의 효율성을 극대화하였다**.

뿐만 아니라, Spark를 통한 전처리 작업에 있어서도 날짜별 파티셔닝은 효과적이다. 날짜 당 대략 1650개의 뉴스 데이터이기 때문에, 코사인 유사도 검사와 같은 무거운 작업의 데이터 단위를 1650개로 축소하여 작업할 수 있기 때문에, 작업의 시간 효율성도 크게 증가할 수 있었다.

2. Spark를 이용한 분산 처리와 병렬 처리

Spark는 **디스크가 아닌 인메모리 방식으로 동작하여, 메모리에서 동작하기 때문에, 속도적으로 더 빠르다는 장점이 있다.** 그리고 Spark는 데이터를 분산 처리하기 때문에 Spark의 병렬 처리 로직을 잘 사용한다면 큰 성능 향상을 얻을 수 있다.

먼저 PySpark의 **DataFrame API**를 통해 로드하고, 필터링 및 코사인 유사도 검사를 진행하였다. Spark의 **분산 처리 엔진**은 데이터를 여러 노드에 분산하여 병렬로 처리할 수 있기 때문에, 대규모 데이터를 처리하는 데 최적화되어 있다.

특히 `RDD` 변환을 통해 벡터 데이터를 병렬로 처리하는 부분이 그 예시이다. 여기서 `rescaledData.select('features').rdd.map(lambda row: row.features.toArray()).collect()`를 통해 데이터의 특성을 병렬로 처리하여 유사도 계산을 빠르게 수행하였다. 이는 **대용량 데이터를 처리할 때 단일 머신에서 처리하는 것보다 훨씬 빠른 성능**을 보장할 수 있다.

또한, Spark의 **지연 평가**를 이용하여 모든 변환이 **필요한 시점에만 실행되도록 하였다.** 이 코드는 `filter`, `map`, `select` 등의 변환 작업을 다수 포함하고 있지만, Spark는 마지막 `collect()`나 `count()`와 같은 액션이 호출되기 전까지는 실제로 연산을 수행하지 않는단 특성이 있다. 이를 통해 **데이터 처리 과정을 최적화**할 수 있었다.

예를 들어, `df_filtered.count()`를 호출할 때에만 필터링이 적용된 데이터에 대한 실제 연산이 실행된다. 이를 통해 전체 데이터셋이 아닌, 필요한 데이터만을 처리함으로써 **메모리 사용량을 줄이고 성능을 최적화**할 수 있었다.

3. 리팩토링

날짜별 파티셔닝과 병렬처리 과정을 적용하였다 생각했는데, 실행 시간이 비정상적으로 아주 길었다.

```
Date: 2022.01.11 - Original: 1593, Unique: 1380, Removed: 213, Time: 329.80 seconds
Total processing time: 329.80 seconds
```

그래서 코드의 모든 동작에 대해 시간측정을 진행하여, 어디서 시간이 오래 걸리는지를 찾고, 코드 수정을 진행하였다.

이 부분은 해당 포스팅에 자세히 정리하였다.

https://velog.io/@sonbaejun/pyspark에서-비정상적으로-오래-걸리던-코사인-유사도검사-과정-해결

결론

```
Date: 2022.01.11 - Original: 1629, Unique: 1386, Removed: 243
Total processing time: 3.66 seconds
```

결론적으로 하루의 뉴스 데이터를 전처리 하는 소요시간을 3.5초까지 단축할 수 있었다.

Spark의 분산 처리와 병렬 처리 로직, 그리고 날짜별로 필터링하여 데이터를 처리하는 아이디어 등을 통해 소요시간을 단축하였고, 해당 로직에 대해 공부할 수 있었던 것이 큰 수확이었다.

▼ Hadoop과 Spark의 테스트 환경을 로컬이 아닌 Docker로 구축한 이유

↓↓↓ 포스팅 작성

https://velog.io/@sonbaejun/Hadoop과-Spark의-테스트-환경을-로컬이-아닌-Docker로-구축한-이유

개요

프로젝트에서 뉴스 데이터 처리를 담당하게 되었다.

우리 팀의 크롤링 데이터 목표는 **1200만개이고, 지금도 400만개 정도의 뉴스 데이터를 크롤링하며 계속 데이터를 열심히 모으고 있다.**

데이터 숫자가 꽤나 커졌기 때문에, 빅데이터 기술을 적용할 필요가 있다고 느꼈다.

따라서 데이터를 Hadoop과 Spark를 이용하여 처리하기로 결정하였다. 이 포스팅에서는 해당 과정의 환경구축 과정에 대해 서술하고자 한다.

Hadoop과 Spark 환경구축

컨테이너 기반 분산 환경 구축을 통한 협업 효율성 극대화

결론적으로, Docker를 이용해 Hadoop과 Spark, NoSQL를 각각의 컨테이너로 구성하고 각 컨테이너의 환경구축을 완료했다.

사실 로컬에서 전처리가 잘 되기만 하면 됐기 때문에, 굳이 Docker Container로 구축할 필요는 없었다(그 과정이 굉장히 번거롭기도 했고 말이다,,)

그럼에도 Docker Container로 환경설정을 한 이유는 명확했다.

1. **여러명의 팀원이 Spark를 이용하는 작업에 참여되기 때문이다.**

   Hadoop과 Spark의 환경을 구축하고 연동하는게 생각보다 복잡한 환경 구축이 필요하다고 느꼈다. 그렇기 때문에 각자의 팀원들이 Spark를 사용해 작업을 할때, 모두가 각자의 버전과 환경에서 테스트를 한다면, 추후 병합 시 문제가 발생할 가능성이 불 보듯 뻔했다.

   따라서 **Docker Container를 이용해 환경 구축을 함으로써, 모든 팀원들이 동일한 환경에서 각자의 Spark 작업을 처리할 수 있도록 만들었다.**

2. **추후 서버 배포에 용이하게 하기위해**

   팀의 인프라 담당자 분이 Hadoop과 Spark를 서버에 배포하여야 되는데, 인프라를 담당하는 분은 데이터 처리 관련 작업을 담당하지 않으셨다. 물론 버전같은 부분만 따로 알려드려서 통일을 할 순 있었지만, 데이터 처리 작업을 주도하여 담당하는 내가 해당 환경에 대해 완벽히 이해하고 있어야 한다고 생각했다. 따라서 실제 배포때 사용할 Docker Container를 사용했고, 그 결과 인프라 담당자가 서버에 Hadoop과 Spark를 올릴 때를 대비한 매뉴얼을 제시할 수 있었다.

   https://velog.io/@sonbaejun/Docker를-이용하여-Hadoop과-Spark-환경-구축

   https://velog.io/@sonbaejun/NoSQLMongoDB-컨테이너-구축-및-Spark-연동

결론

위와 같은 이유로, 많은 cost가 들었음에도 docker container로 환경 구축을 했다.

도커를 본격적으로 써본건 처음이라서, 많은 시행착오를 겪고 초반 세팅에 꽤나 많은 시간을 소요한 것은 아쉽지만, 그로 인해서 추후 서버 배포, Spark 협업 진행 시에 초반의 cost보다 훨씬 더 큰 return들을 받을 수 있기 때문에 해당 방식으로 구축한 것에 굉장히 만족한다.

▼ Kafka 컨테이너에 올리고 Spark 컨테이너와 연동

▼ Kafka 컨테이너에 올리기

```
# Kafka로 전송하기 위해 JSON 형식으로 변환
samsung_news_df.selectExpr("CAST(_id AS STRING) AS key",
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:300
    .option("topic", "test-topic") \
    .save()
```

```
pyspark --conf "spark.mongodb.read.connection.uri=mongod
--conf "spark.mongodb.write.connection.uri=mongodb://mon
--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.
```

```
from pyspark.sql import SparkSession

# Spark 세션 생성
spark = SparkSession.builder \
    .appName("KafkaProducer") \
    .config("spark.mongodb.read.connection.uri", "mongod
    .getOrCreate()

# MongoDB에서 데이터 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.ne
    .load()

# Kafka로 전송하기 위해 JSON 형식으로 변환
df.selectExpr("CAST(_id AS STRING) AS key", "to_json(str
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:300
    .option("topic", "test-topic") \
    .save()
```

토픽 2 : DailyStockFrequency

▼ 최종

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import StructType, StructField, Str
import os
from datetime import datetime, timedelta
from functools import reduce
from pyspark.sql import functions as F
from pyspark.sql.functions import col, current_timestamp,

# Spark 세션 생성
spark = SparkSession.builder \
    .appName("StockMentionCount") \
    .getOrCreate()

# HDFS 경로 설정
base_path = "/user/news_preprocessing_result"
start_year = 2024
end_year = 2024
date_format_str = "%Y.%m.%d"  # 날짜 형식

# 날짜 리스트 생성 함수
def generate_date_list(start_date, end_date):
    date_list = []
    current_date = start_date
    while current_date <= end_date:
        date_list.append(current_date.strftime(date_format
        current_date += timedelta(days=1)
    return date_list

# 시작 및 종료 날짜 설정
start_date = datetime(2024, 10, 6)  # 시작 날짜
end_date = datetime(2024, 10, 6)    # 종료 날짜

# 날짜 리스트 생성
```

```
date_list = generate_date_list(start_date, end_date)

# 키워드 그룹 정의
keyword_groups = {
    "삼성전자": ["삼전", "삼성", "갤럭시", "비스포크"],
    "SK하이닉스": ["SK하이닉스", "하이닉스", "반도체", "메모리"],
    "현대차": ["현대차", "아이오닉", "아반떼"],
    "셀트리온": ["셀트리온", "바이오" ],
    "KB금융": ["KB금융", "국민은행"],
    "POSCO홀딩스": ["POSCO", "포스코", "철강", "제철"],
    "기아": ["기아", "기아차", "카니발", "k3","k7"],
    "신한지주": ["신한지주", "신한", "SOL"],
    "NAVER": ["NAVER", "네이버", "CLOVA"],
    "삼성SDI": ["삼성SDI", "SDI", "배터리"],
    "삼성바이오로직스": ["삼성바이오로직스", "바이오"],
    "LG화학": ["LG화학","석유"],
    "하나금융지주": ["하나금융", "하나은행"],
    "LG에너지솔루션": ["LG에너지솔루션", "LG에너지"],
    "현대모비스": ["현대모비스", "모비스"],
    "삼성물산": ["삼성물산", "물산"],
    "LG전자": ["LG전자", "LG", "가전"],
    "카카오": ["카카오", "Kakao"],
    "삼성화재": ["삼성화재"],
    "우리금융지주": ["우리금융", "우리은행"],
    "크래프톤": ["크래프톤", "배틀그라운드"],
    "KT": ["KT"],
    "SK텔레콤": ["SK텔레콤", "SKT"],
    "삼성생명": ["삼성생명"],
    "LG": ["LG", "LG그룹"],
    "삼성전기": ["삼성전기"],
    "카카오뱅크": ["카카오뱅크", "카뱅"],
    "삼성중공업": ["삼성중공업"],
    "SK이노베이션": ["SK이노베이션", "SK이노", "화학"],
    "삼성에스디에스": ["삼성에스디에스", "SDS"],
    "대한항공": ["대한항공"],
    "SK": ["SK", "SK그룹"],
    "DB손해보험": ["DB손해보험", "손해보험"],
    "HD현대중공업": ["HD현대중공업", "현대중공업"],
```

```
    "기업은행": ["기업은행"],
    "한국항공우주": ["한국항공우주"],
    "엔씨소프트": ["엔씨소프트", "NC", "리니지"],
    "하이브": ["하이브", "BTS", "방탄소년단","르세라핌","뉴진스"],
    "LG디스플레이": ["LG디스플레이", "디스플레이", "패널"],
    "SK바이오팜": ["SK바이오팜", "바이오", "제약"]
}

# 키워드 빈도수 계산 함수
def calculate_keyword_frequencies(df, keyword_groups, date
    frequencies = []

    for stock, keywords in keyword_groups.items():
        # 각 키워드에 대해 contains 조건을 생성
        conditions = [F.col("title").contains(keyword) for
        # 모든 키워드 조건을 OR로 결합
        filter_condition = reduce(lambda a, b: a | b, cond

        count = df.filter(filter_condition).count()

        if count > 0:
            frequencies.append((stock, count, date))

    return frequencies

# 전체 로직 - 날짜별로 파티셔닝된 데이터를 읽고 키워드 빈도수를 계산
all_keyword_frequencies = []

for date_str in date_list:
    # 날짜별로 각 Parquet 파일 경로 설정
    hdfs_path = os.path.join(base_path, str(start_year), d
    try:
        df_temp = spark.read.parquet(hdfs_path + "/*.parqu
        # 날짜별 키워드 빈도수 계산
        keyword_frequencies = calculate_keyword_frequencie
        all_keyword_frequencies.extend(keyword_frequencies
    except Exception as e:
        print(f"파일 로드 실패: {hdfs_path}, 에러: {e}")
```

```
# 결과를 DataFrame으로 변환
schema = StructType([
    StructField("stockName", StringType(), True),
    StructField("count", IntegerType(), True),
    StructField("newsPublishedDate", StringType(), True)
])

# DataFrame 생성
keyword_freq_df = spark.createDataFrame(all_keyword_freque

# newsPublishedDate 형식 변경 및 dailyStockFrequencyCreatedAt
final_df = keyword_freq_df.select(
    regexp_replace(col("newsPublishedDate"), r'\.', '-').a
    col("stockName"),
    col("count"),
    date_format(current_timestamp(), "yyyy-MM-dd'T'HH:mm:s
)

# 결과 Kafka로 발행
final_df.selectExpr("to_json(struct(*)) AS value") \
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:30092
    .option("topic", "DailyStockFrequency-Topic") \
    .save()
```

토픽 3 : DailyKeywordFrequency

▼ 최종

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, collect_list, explod
from pyspark.sql.types import MapType, StringType, IntegerT
from datetime import datetime, timedelta
import os

# Spark 세션 생성
```

```
spark = SparkSession.builder \
    .appName("DailyKeywordAggregation") \
    .getOrCreate()

# HDFS 경로 설정
base_path = "/user/news_preprocessing_testing"
start_year = 2021
end_year = 2021
date_format_str = "%Y.%m.%d"  # 날짜 형식

# 날짜 리스트 생성 함수
def generate_date_list(start_date, end_date):
    date_list = []
    current_date = start_date
    while current_date <= end_date:
        date_list.append(current_date.strftime(date_format_
        current_date += timedelta(days=1)
    return date_list

# 시작 및 종료 날짜 설정
start_date = datetime(2021, 1, 10)  # 시작 날짜
end_date = datetime(2021, 1, 11)    # 종료 날짜

# 날짜 리스트 생성
date_list = generate_date_list(start_date, end_date)

# HDFS에서 Parquet 파일 로드
dfs = []  # 모든 DataFrame을 저장할 리스트

for date_str in date_list:
    hdfs_path = os.path.join(base_path, str(start_year), da
    # Parquet 파일 로드
    try:
        df_temp = spark.read.parquet(hdfs_path + "/*.parque
        dfs.append(df_temp)
    except Exception as e:
        print(f"파일 로드 실패: {hdfs_path}, 에러: {e}")
```

```
# 모든 DataFrame을 하나로 합치기
if dfs:
    df = dfs[0]  # 첫 번째 DataFrame
    for temp_df in dfs[1:]:
        df = df.union(temp_df)

spark_df = df

# 형태소 분석 함수
def analyze_content(content):
    from bareunpy import Tagger
    API_KEY = "koba-TLIK2BA-WPNUG3I-UV4COFQ-I6J62JI"
    tagger = Tagger(API_KEY, 'bareun', 5757)
    try:
        res = list(tagger.tags([content]).pos())
        dict_ = {}
        for keyword in res:
            if keyword[1] in ['NNG', 'NNP', 'NNB']:  # 관심
                dict_[keyword[0]] = dict_.get(keyword[0], 0
        return dict_
    except Exception as e:
        return {}

# UDF(User Defined Function)로 변환하여 스파크에서 사용 가능하게 만
analyze_content_udf = udf(analyze_content, MapType(StringTy

# DataFrame에 형태소 분석 적용
spark_df = spark_df.withColumn("keyword_map", analyze_conte

# 날짜만 추출하여 그룹핑
spark_df = spark_df.withColumn("date_only", col("date").sub

# 날짜별로 키워드 집계
aggregated_df = spark_df.groupBy("date_only").agg(
    collect_list("keyword_map").alias("keyword_maps")
)

# 키워드 및 빈도수 계산
```

```
final_result = aggregated_df.select(
    "date_only",
    explode(col("keyword_maps")).alias("keyword_map")
).groupBy("date_only") \
    .agg(
        collect_list("keyword_map").alias("keywords")
    )

# 키워드 딕셔너리로 변환
def merge_keywords(keywords):
    merged = {}
    for keyword in keywords:
        for k, v in keyword.items():
            if v > 25:  # 빈도수가 25 이하인 키워드는 제외
                merged[k] = merged.get(k, 0) + v
    return merged

# UDF 등록
merge_keywords_udf = udf(merge_keywords, MapType(StringType

# merged_keywords 생성
final_result = final_result.withColumn("merged_keywords", m

# 최종 결과를 DataFrame으로 변환
final_df = final_result.select(
    regexp_replace(col("date_only"), r'\.', '-').alias("new
    col("merged_keywords").alias("keyword"),
    date_format(current_timestamp(), "yyyy-MM-dd'T'HH:mm:ss
)

# 결과 Kafka로 발행
final_df.selectExpr("to_json(struct(*)) AS value") \
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:30092"
    .option("topic", "DailyStockFrequency-Topic") \
    .save()
```

```
print("데이터가 Kafka로 발행되었습니다.")
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, collect_list, explod
from pyspark.sql.types import MapType, StringType, IntegerT
from datetime import datetime, timedelta

# Spark 세션 생성
spark = SparkSession.builder \
    .appName("DailyKeywordAggregation") \
    .getOrCreate()

# HDFS 경로 설정
base_path = "/user/news_preprocessing_result"
start_year = 2024
end_year = 2024
date_format_str = "%Y.%m.%d"  # 날짜 형식

# 날짜 리스트 생성 함수
def generate_date_list(start_date, end_date):
    date_list = []
    current_date = start_date
    while current_date <= end_date:
        date_list.append(current_date.strftime(date_format_
        current_date += timedelta(days=1)
    return date_list

# 시작 및 종료 날짜 설정
start_date = datetime(2024, 1, 6)  # 시작 날짜
end_date = datetime(2024, 2, 28)    # 종료 날짜

# 날짜 리스트 생성
date_list = generate_date_list(start_date, end_date)

# HDFS에서 Parquet 파일 로드
dfs = []  # 모든 DataFrame을 저장할 리스트
```

```
for year in range(start_year, end_year + 1):
    for date_str in date_list:
        hdfs_path = os.path.join(base_path, str(year), date
        # Parquet 파일 로드
        try:
            df_temp = spark.read.parquet(hdfs_path + "/*.pa
            dfs.append(df_temp)
        except Exception as e:
            print(f"파일 로드 실패: {hdfs_path}, 에러: {e}")

# 모든 DataFrame을 하나로 합치기
if dfs:
    df = dfs[0]  # 첫 번째 DataFrame
    for temp_df in dfs[1:]:
        df = df.union(temp_df)

spark_df = df

# 형태소 분석 함수
def analyze_content(content):
    from bareunpy import Tagger
    API_KEY = "koba-TLIK2BA-WPNUG3I-UV4COFQ-I6J62JI"
    tagger = Tagger(API_KEY, 'bareun', 5757)
    try:
        res = list(tagger.tags([content]).pos())
        dict_ = {}
        for keyword in res:
            if keyword[1] in ['NNG', 'NNP', 'NNB']:  # 관심
                dict_[keyword[0]] = dict_.get(keyword[0], 0
        return dict_
    except Exception as e:
        return {}

# UDF(User Defined Function)로 변환하여 스파크에서 사용 가능하게 만
analyze_content_udf = udf(analyze_content, MapType(StringTy

# DataFrame에 형태소 분석 적용
spark_df = spark_df.withColumn("keyword_map", analyze_conte
```

```
# 날짜만 추출하여 그룹핑
spark_df = spark_df.withColumn("date_only", col("date").sub

# 날짜별로 키워드 집계
aggregated_df = spark_df.groupBy("date_only").agg(
    collect_list("keyword_map").alias("keyword_maps")
)

# 키워드 및 빈도수 계산
final_result = aggregated_df.select(
    "date_only",
    explode(col("keyword_maps")).alias("keyword_map")
).groupBy("date_only") \
    .agg(
        collect_list("keyword_map").alias("keywords")
    )

# 키워드 딕셔너리로 변환
def merge_keywords(keywords):
    merged = {}
    for keyword in keywords:
        for k, v in keyword.items():
            if v > 25:  # 빈도수가 50이하인 키워드는 제외
                merged[k] = merged.get(k, 0) + v
    return merged

# UDF 등록
merge_keywords_udf = udf(merge_keywords, MapType(StringType

# merged_keywords 생성
final_result = final_result.withColumn("merged_keywords", m

# 최종 결과를 DataFrame으로 변환
final_df = final_result.select(
    regexp_replace(col("date_only"), r'\.', '-').alias("new
    col("merged_keywords").alias("keyword"),
    date_format(current_timestamp(), "yyyy-MM-dd'T'HH:mm:ss
```

```
)

# 결과 Kafka로 발행
final_df.selectExpr("to_json(struct(*)) AS value") \
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:30092"
    .option("topic", "DailyStockFrequency-Topic") \
    .save()

print("데이터가 Kafka로 발행되었습니다.")
```

토픽 4, 5 : 뉴스 메타데이터 + 뉴스와 관련된 종목이름

▼ NEWS 토픽 전송 코드

▼ 초안

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_json, struct, lit,
from pyspark.sql.types import ArrayType, StringType
import os
from datetime import datetime, timedelta

# Spark 세션 생성
spark = SparkSession.builder \
    .appName("KafkaProducer") \
    .getOrCreate()

# HDFS 경로 설정
base_path = "/user/news_preprocessing_testing"
start_year = 2021
end_year = 2021
date_format_str = "%Y.%m.%d"  # 날짜 형식

# 날짜 리스트 생성 함수
def generate_date_list(start_date, end_date):
```

```
    date_list = []
    current_date = start_date
    while current_date <= end_date:
        date_list.append(current_date.strftime(date_form
        current_date += timedelta(days=1)
    return date_list

# 시작 및 종료 날짜 설정
start_date = datetime(2021, 1, 1)  # 시작 날짜
end_date = datetime(2021, 1, 2)    # 종료 날짜

# 날짜 리스트 생성
date_list = generate_date_list(start_date, end_date)

# HDFS에서 Parquet 파일 로드
dfs = []  # 모든 DataFrame을 저장할 리스트

for year in range(start_year, end_year + 1):
    for date_str in date_list:
        hdfs_path = os.path.join(base_path, str(year), d
        # Parquet 파일 로드
        try:
            df_temp = spark.read.parquet(hdfs_path + "/*
            dfs.append(df_temp)
        except Exception as e:
            print(f"파일 로드 실패: {hdfs_path}, 에러: {e}"

# 모든 DataFrame을 하나로 합치기
if dfs:
    df = dfs[0]  # 첫 번째 DataFrame
    for temp_df in dfs[1:]:
        df = df.union(temp_df)

# 키워드 그룹 정의
keyword_groups = {
    "삼성전자": ["삼전", "삼성", "반도체"],
    "SK하이닉스": ["SK하이닉스", "하이닉스", "반도체"],
    "현대차": ["현대차", "차량", "자동차"],
```

```
    "셀트리온": ["셀트리온", "바이오", "의약품"],
    "KB금융": ["KB금융", "금융"],
    "POSCO홀딩스": ["POSCO홀딩스", "철강", "제철"],
    "기아": ["기아", "차", "자동차"],
    "신한지주": ["신한지주", "금융"],
    "NAVER": ["NAVER", "포털"],
    "삼성SDI": ["삼성SDI", "배터리"],
    "삼성바이오로직스": ["삼성바이오로직스", "바이오"],
    "LG화학": ["LG화학", "화학"],
    "하나금융지주": ["하나금융지주", "금융"],
    "LG에너지솔루션": ["LG에너지솔루션", "에너지", "배터리"],
    "현대모비스": ["현대모비스", "부품"],
    "삼성물산": ["삼성물산", "건설"],
    "LG전자": ["LG전자", "가전"],
    "카카오": ["카카오", "IT"],
    "삼성화재": ["삼성화재", "보험"],
    "우리금융지주": ["우리금융지주", "금융"],
    "크래프톤": ["크래프톤", "게임"],
    "KT": ["KT", "통신"],
    "SK텔레콤": ["SK텔레콤", "통신"],
    "삼성생명": ["삼성생명", "보험"],
    "LG": ["LG", "기업"],
    "삼성전기": ["삼성전기", "전자"],
    "카카오뱅크": ["카카오뱅크", "은행"],
    "삼성중공업": ["삼성중공업", "조선"],
    "SK이노베이션": ["SK이노베이션", "화학"],
    "삼성에스디에스": ["삼성에스디에스", "IT"],
    "대한항공": ["대한항공", "항공"],
    "SK": ["SK", "기업"],
    "DB손해보험": ["DB손해보험", "보험"],
    "HD현대중공업": ["HD현대중공업", "조선"],
    "기업은행": ["기업은행", "은행"],
    "한국항공우주": ["한국항공우주", "항공"],
    "엔씨소프트": ["엔씨소프트", "게임"],
    "하이브": ["하이브", "엔터"],
    "LG디스플레이": ["LG디스플레이", "디스플레이"],
    "SK바이오팜": ["SK바이오팜", "바이오"]
}
```

```
# 키워드 빈도수 계산 함수
def extract_keywords(title, content):
    keywords = []
    for stock, keywords_group in keyword_groups.items():
        if any(keyword in title or keyword in content fo
            keywords.append(stock)
    return keywords

# UDF 등록
extract_keywords_udf = udf(extract_keywords, ArrayType(S

# 뉴스 데이터에 키워드 열 추가
transformed_df = df.withColumn("keywords", extract_keywo

# newsId를 제외하고 Kafka로 전송하기 위해 JSON 형식으로 변환
transformed_df.drop("newsId").selectExpr("to_json(struct
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:300
    .option("topic", "NEWS") \
    .save()
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_json, struct, lit,
from pyspark.sql.types import ArrayType, StringType

# Spark 세션 생성
spark = SparkSession.builder \
    .appName("KafkaProducer") \
    .config("spark.mongodb.read.connection.uri", "mongod
    .getOrCreate()

# MongoDB에서 데이터 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.ne
    .load()
```

```
# 필드명 변환 및 추가 필드 적용
transformed_df = df.withColumnRenamed("_id", "newsId") \
    .withColumnRenamed("title", "title") \
    .withColumnRenamed("content", "content") \
    .withColumnRenamed("img_src", "thumbnailImg") \
    .withColumnRenamed("link", "newsLink") \
    .withColumnRenamed("date", "publishedDate") \
    .withColumnRenamed("category1", "category") \
    .withColumn("sentimentIndex", lit(0.0)) \
    .withColumn("newsCreatedAt", current_timestamp())

# 카테고리를 ENUM으로 변환
transformed_df = transformed_df.withColumn(
    "category",
    when(col("category") == "정치", "정치")
    .when(col("category") == "경제", "경제")
    .when(col("category") == "사회", "사회")
    .when(col("category") == "기술", "기술")
    .when(col("category") == "스포츠", "스포츠")
    .when(col("category") == "연예", "연예")
    .when(col("category") == "세계", "세계")
    .when(col("category") == "날씨", "날씨")
    .when(col("category") == "건강", "건강")
    .when(col("category") == "생활", "생활")
    .otherwise(None)
)

# Spark 세션에서 시간 파싱 정책을 LEGACY로 설정
spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEG

# publishedDate를 "yyyy-MM-dd'T'HH:mm:ss" 형식으로 변환
transformed_df = transformed_df.withColumn(
    "publishedDate",
    to_timestamp(
        regexp_replace(
            regexp_replace(col("publishedDate"), "오후",
        ),
```

```
        "yyyy.MM.dd. a hh:mm"
    )
)

# publishedDate를 "yyyy-MM-dd'T'HH:mm:ss" 형식으로 변환
transformed_df = transformed_df.withColumn(
    "publishedDate",
    date_format(col("publishedDate"), "yyyy-MM-dd'T'HH:m
)

# newsCreatedAt 컬럼 추가 (현재 시간, 형식: yyyy-MM-dd'T'HH:
transformed_df = transformed_df.withColumn(
    "newsCreatedAt",
    date_format(current_timestamp(), "yyyy-MM-dd'T'HH:mm
)

# 주식명 목록
stock_names = [
    "삼성전자", "SK하이닉스", "현대차", "셀트리온", "KB금융", "
    "기아", "신한지주", "NAVER", "삼성SDI", "삼성바이오로직스",
    "하나금융지주", "LG에너지솔루션", "현대모비스", "삼성물산", "
    "카카오", "삼성화재", "우리금융지주", "크래프톤", "KT", "SK
    "삼성생명", "LG", "삼성전기", "카카오뱅크", "삼성중공업", "S
    "삼성에스디에스", "대한항공", "SK", "DB손해보험", "HD현대중공
    "기업은행", "한국항공우주", "엔씨소프트", "하이브", "LG디스플
]

# 주식명이 포함된 뉴스 키워드 추출
def extract_keywords(title, content):
    keywords = []
    for stock in stock_names:
        if stock in title or stock in content:
            keywords.append(stock)
    return keywords

# UDF 등록
extract_keywords_udf = udf(extract_keywords, ArrayType(S
```

```
# 뉴스 데이터에 키워드 열 추가
transformed_df = transformed_df.withColumn("keywords", e

# newsId를 제외하고 Kafka로 전송하기 위해 JSON 형식으로 변환
transformed_df.drop("newsId").selectExpr("to_json(struct
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:300
    .option("topic", "NEWS-TEST-LOCAL14") \
    .save()
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_json, struct, lit,

# Spark 세션 생성
spark = SparkSession.builder \
    .appName("KafkaProducer") \
    .config("spark.mongodb.read.connection.uri", "mongod
    .getOrCreate()

# MongoDB에서 데이터 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.ne
    .load() \
    .limit(5)

# 필드명 변환 및 추가 필드 적용
transformed_df = df.withColumnRenamed("_id", "news_id")
    .withColumnRenamed("category1", "category") \
    .withColumnRenamed("content", "content") \
    .withColumnRenamed("img_src", "thumbnail_img") \
    .withColumnRenamed("link", "news_link") \
    .withColumnRenamed("title", "title") \
    .withColumnRenamed("date", "published_date") \
    .withColumn("sentiment_index", lit(0.0)) \
    .withColumn("news_created_at", current_timestamp())
```

```
# Spark 세션에서 시간 파싱 정책을 LEGACY로 설정
spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEG

# published_date를 "yyyy-MM-dd HH:mm:ss" 형식의 DATETIME으
transformed_df = transformed_df.withColumn(
    "published_date",
    to_timestamp(
        regexp_replace(
            regexp_replace(col("published_date"), "오후",
        ),
        "yyyy.MM.dd. a hh:mm"
    )
)

# published_date를 "yyyy-MM-dd HH:mm:ss" 형식으로 변환
transformed_df = transformed_df.withColumn(
    "published_date",
    date_format(col("published_date"), "yyyy-MM-dd HH:mm
)

# news_created_at 컬럼 추가 (현재 시간, 형식: yyyy-MM-dd HH:
transformed_df = transformed_df.withColumn(
    "news_created_at",
    date_format(current_timestamp(), "yyyy-MM-dd HH:mm:s
)

# Kafka로 전송하기 위해 JSON 형식으로 변환
transformed_df.selectExpr("CAST(news_id AS STRING) AS ke
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:300
    .option("topic", "test-news2") \
    .save()
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_json, struct, lit,

# Spark 세션 생성
```

```
spark = SparkSession.builder \
    .appName("KafkaProducer") \
    .config("spark.mongodb.read.connection.uri", "mongod
    .getOrCreate()

# MongoDB에서 데이터 로드
df = spark.read \
    .format("mongodb") \
    .option("uri", "mongodb://localhost:27018/news_db.ne
    .load() \

# 필드명 변환 및 추가 필드 적용
transformed_df = df.withColumnRenamed("_id", "newsId") \
    .withColumnRenamed("title", "title") \
    .withColumnRenamed("content", "content") \
    .withColumnRenamed("img_src", "thumbnailImg") \
    .withColumnRenamed("link", "newsLink") \
    .withColumnRenamed("date", "publishedDate") \
    .withColumnRenamed("category1", "category") \
    .withColumn("sentimentIndex", lit(0.0)) \
    .withColumn("newsCreatedAt", current_timestamp())

# 카테고리를 ENUM으로 변환
transformed_df = transformed_df.withColumn(
    "category",
    when(col("category") == "정치", "정치")
    .when(col("category") == "경제", "경제")
    .when(col("category") == "사회", "사회")
    .when(col("category") == "기술", "기술")
    .when(col("category") == "스포츠", "스포츠")
    .when(col("category") == "연예", "연예")
    .when(col("category") == "세계", "세계")
    .when(col("category") == "날씨", "날씨")
    .when(col("category") == "건강", "건강")
    .when(col("category") == "생활", "생활")
    .otherwise(None)  # 혹시 모를 예외 상황을 처리하기 위해 No
)
```

```
# Spark 세션에서 시간 파싱 정책을 LEGACY로 설정
spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEG

# publishedDate를 "yyyy-MM-dd'T'HH:mm:ss" 형식의 DATETIME
transformed_df = transformed_df.withColumn(
    "publishedDate",
    to_timestamp(
        regexp_replace(
            regexp_replace(col("publishedDate"), "오후",
        ),
        "yyyy.MM.dd. a hh:mm"
    )
)

# publishedDate를 "yyyy-MM-dd'T'HH:mm:ss" 형식으로 변환
transformed_df = transformed_df.withColumn(
    "publishedDate",
    date_format(col("publishedDate"), "yyyy-MM-dd'T'HH:m
)

# newsCreatedAt 컬럼 추가 (현재 시간, 형식: yyyy-MM-dd'T'HH:
transformed_df = transformed_df.withColumn(
    "newsCreatedAt",
    date_format(current_timestamp(), "yyyy-MM-dd'T'HH:mm
)

# newsId를 제외하고 Kafka로 전송하기 위해 JSON 형식으로 변환
transformed_df.drop("newsId").selectExpr("to_json(struct
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:300
    .option("topic", "NEWS-TEST-LOCAL5") \
    .save()
```

잘 들어갔나 확인하려면

```
# Kafka에서 데이터 읽어오기
kafka_df = spark \
```

```
    .read \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:300
    .option("subscribe", "DailyStockFrequencyTest") \
    .load()

# value 컬럼의 데이터를 문자열로 변환
kafka_df = kafka_df.selectExpr("CAST(key AS STRING)", "C

# 데이터 출력
kafka_df.show(truncate=False)
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_json, struct, lit,
import random
from datetime import datetime, timedelta

# Spark 세션 생성
spark = SparkSession.builder \
    .appName("KafkaProducer") \
    .config("spark.mongodb.read.connection.uri", "mongod
    .getOrCreate()

# 국내 종목명 리스트
stock_names = ["삼성전자", "SK하이닉스", "LG화학", "네이버", "

# 데이터 생성
data = []
for i in range(12):
    stock_name = random.choice(stock_names)  # 랜덤으로 종
    news_published_date = datetime.now() - timedelta(day
    frequency = random.randint(1, 100)  # 랜덤 빈도수 생성
    created_at = datetime.now()  # 현재 생성 일시

    data.append((stock_name, news_published_date.strftim

# Spark DataFrame 생성
columns = ["stock_name", "news_published_date", "frequer
```

```
df = spark.createDataFrame(data, columns)

# Kafka로 전송하기 위해 JSON 형식으로 변환
df.selectExpr("CAST(stock_name AS STRING) AS key", "to_j
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:300
    .option("topic", "DailyStockFrequencyTest") \
    .save()

# Spark 세션 종료
spark.stop()
```

▼ 최종

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_json, struct, lit,
from pyspark.sql.types import ArrayType, StringType
import os
from datetime import datetime, timedelta

# Spark 세션 생성
spark = SparkSession.builder \
    .appName("KafkaProducer") \
    .getOrCreate()

# HDFS 경로 설정
base_path = "/user/news_preprocessing_result"
start_year = 2024
end_year = 2024
date_format_str = "%Y.%m.%d"  # 날짜 형식

# 날짜 리스트 생성 함수
def generate_date_list(start_date, end_date):
    date_list = []
    current_date = start_date
    while current_date <= end_date:
        date_list.append(current_date.strftime(date_form
```

```
        current_date += timedelta(days=1)
    return date_list

# 시작 및 종료 날짜 설정
start_date = datetime(2024, 10, 8)  # 시작 날짜
end_date = datetime(2024, 10, 8)    # 종료 날짜

# 날짜 리스트 생성
date_list = generate_date_list(start_date, end_date)

# HDFS에서 Parquet 파일 로드
dfs = []  # 모든 DataFrame을 저장할 리스트

for year in range(start_year, end_year + 1):
    for date_str in date_list:
        hdfs_path = os.path.join(base_path, str(year), d
        # Parquet 파일 로드
        try:
            df_temp = spark.read.parquet(hdfs_path + "/*
            dfs.append(df_temp)
        except Exception as e:
            print(f"파일 로드 실패: {hdfs_path}, 에러: {e}"

# 모든 DataFrame을 하나로 합치기
if dfs:
    df = dfs[0]  # 첫 번째 DataFrame
    for temp_df in dfs[1:]:
        df = df.union(temp_df)

# 필드명 변환 및 추가 필드 적용
transformed_df = df.withColumnRenamed("_id", "newsId") \
    .withColumnRenamed("title", "title") \
    .withColumnRenamed("content", "content") \
    .withColumnRenamed("img_src", "thumbnailImg") \
    .withColumnRenamed("link", "newsLink") \
    .withColumnRenamed("date", "publishedDate") \
    .withColumnRenamed("category1", "category") \
    .withColumn("sentimentIndex", lit(0.0)) \
```

```
    .withColumn("newsCreatedAt", current_timestamp())

# 카테고리를 ENUM으로 변환
transformed_df = transformed_df.withColumn(
    "category",
    when(col("category") == "정치", "정치")
    .when(col("category") == "경제", "경제")
    .when(col("category") == "사회", "사회")
    .when(col("category") == "기술", "기술")
    .when(col("category") == "스포츠", "스포츠")
    .when(col("category") == "연예", "연예")
    .when(col("category") == "세계", "세계")
    .when(col("category") == "날씨", "날씨")
    .when(col("category") == "건강", "건강")
    .when(col("category") == "생활", "생활")
    .otherwise(None)
)

# Spark 세션에서 시간 파싱 정책을 LEGACY로 설정
spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEG

# publishedDate를 "yyyy-MM-dd'T'HH:mm:ss" 형식으로 변환
transformed_df = transformed_df.withColumn(
    "publishedDate",
    to_timestamp(
        regexp_replace(
            regexp_replace(col("publishedDate"), "오후",
        ),
        "yyyy.MM.dd. a hh:mm"
    )
)

# publishedDate를 "yyyy-MM-dd'T'HH:mm:ss" 형식으로 변환
transformed_df = transformed_df.withColumn(
    "publishedDate",
    date_format(col("publishedDate"), "yyyy-MM-dd'T'HH:m
)
```

```
# newsCreatedAt 컬럼 추가 (현재 시간, 형식: yyyy-MM-dd'T'HH:
transformed_df = transformed_df.withColumn(
    "newsCreatedAt",
    date_format(current_timestamp(), "yyyy-MM-dd'T'HH:mm
)

# 키워드 그룹 정의
keyword_groups = {
    "삼성전자": ["삼전", "삼성", "갤럭시", "비스포크"],
    "SK하이닉스": ["SK하이닉스", "하이닉스", "반도체", "메모리"]
    "현대차": ["현대차", "아이오닉", "아반떼"],
    "셀트리온": ["셀트리온", "바이오" ],
    "KB금융": ["KB금융", "국민은행"],
    "POSCO홀딩스": ["POSCO", "포스코", "철강", "제철"],
    "기아": ["기아", "기아차", "카니발", "k3","k7"],
    "신한지주": ["신한지주", "신한", "SOL"],
    "NAVER": ["NAVER", "네이버", "CLOVA"],
    "삼성SDI": ["삼성SDI", "SDI", "배터리"],
    "삼성바이오로직스": ["삼성바이오로직스", "바이오"],
    "LG화학": ["LG화학","석유"],
    "하나금융지주": ["하나금융", "하나은행"],
    "LG에너지솔루션": ["LG에너지솔루션", "LG에너지"],
    "현대모비스": ["현대모비스", "모비스"],
    "삼성물산": ["삼성물산", "물산"],
    "LG전자": ["LG전자", "LG", "가전"],
    "카카오": ["카카오", "Kakao"],
    "삼성화재": ["삼성화재"],
    "우리금융지주": ["우리금융", "우리은행"],
    "크래프톤": ["크래프톤", "배틀그라운드"],
    "KT": ["KT"],
    "SK텔레콤": ["SK텔레콤", "SKT"],
    "삼성생명": ["삼성생명"],
    "LG": ["LG", "LG그룹"],
    "삼성전기": ["삼성전기"],
    "카카오뱅크": ["카카오뱅크", "카뱅"],
    "삼성중공업": ["삼성중공업"],
    "SK이노베이션": ["SK이노베이션", "SK이노", "화학"],
    "삼성에스디에스": ["삼성에스디에스", "SDS"],
```

```
    "대한항공": ["대한항공"],
    "SK": ["SK", "SK그룹"],
    "DB손해보험": ["DB손해보험", "손해보험"],
    "HD현대중공업": ["HD현대중공업", "현대중공업"],
    "기업은행": ["기업은행"],
    "한국항공우주": ["한국항공우주"],
    "엔씨소프트": ["엔씨소프트", "NC", "리니지"],
    "하이브": ["하이브", "BTS", "방탄소년단","르세라핌","뉴진스"
    "LG디스플레이": ["LG디스플레이", "디스플레이", "패널"],
    "SK바이오팜": ["SK바이오팜", "바이오", "제약"]
}

# 키워드 빈도수 계산 함수
def extract_keywords(title):
    keywords = []
    for stock, keywords_group in keyword_groups.items():
        if any(keyword in title for keyword in keywords_
            keywords.append(stock)
    return keywords

# UDF 등록
extract_keywords_udf = udf(extract_keywords, ArrayType(S

# 뉴스 데이터에 키워드 열 추가
transformed_df = transformed_df.withColumn("keywords", e

# newsId를 제외하고 Kafka로 전송하기 위해 JSON 형식으로 변환
transformed_df.drop("newsId").selectExpr("to_json(struct
    .write \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "3.38.183.146:300
    .option("topic", "NEWS-Topic") \
    .save()
```

3. properties 환경 및 DockerFile

3-1. news서버 properties

```
spring:
  application:
    name: sog-news
  datasource:
    url: jdbc:mysql://localhost:3306/sog_db
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: ${SPRING_DATASOURCE_USERNAME}
    password: ${SPRING_DATASOURCE_PASSWORD}
  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect
        jdbc:
          time_zone: Asia/Seoul
    show-sql: true
  data:
    redis:
      host: redis
      port: 6379

logging:
  level:
    org:
      hibernate:
        SQL: debug

server:
  port: 8080  # 포트 번호를 8081로 변경
  error:
    include-message: always
    include-binding-errors: always
  tomcat:
    uri-encoding: UTF-8
  servlet:
```

```
    encoding:
      charset: UTF-8
      enabled: true
      force: true

management:
  endpoints:
    web:
      exposure:
        include: health, prometheus
  metrics:
    export:
      prometheus:
        enabled: true
  endpoint:
    prometheus:
      enabled: true

springdoc:
  api-docs:
    path: /v3/api-docs
  swagger-ui:
    path: /swagger-ui/index.html
```

3-2. user서버 properties

```
spring:
  application:
    name: sog

  datasource:
    url: jdbc:mysql://localhost:3306/sog_db
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root  # 환경 변수로부터 가져옴
    password: rootpassword  # 환경 변수로부터 가져옴
  jpa:
    hibernate:
      ddl-auto: update
```

```
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect  # 최신 M
        jdbc:
          time_zone: UTC
    show-sql: true  # SQL 쿼리 출력

  data:
    redis:
      host: redis
      port: 6379

jwt:
  access-token-expiry: 600000    # 10분
  refresh-token-expiry: 86400000 # 1일
  secret: "3c84e62c46e5438e9d12e28a9876591bc34e4e92a3a489b749

logging:
  level:
    org:
      hibernate:
        SQL: debug  # Hibernate SQL 로그 레벨 설정
    com:
      sog:
        user:
          infrastructure:
            security:
              LoginFilter: debug # 특정 클래스에 대한 로깅 설정

server:
  error:
    include-message: always          # 에러 메시지 포함
    include-binding-errors: always   # 바인딩 에러 메시지 포함

management:
  endpoints:
    web:
      exposure:
```

```
        include: health, prometheus  # Actuator Prometheus 엔드
  metrics:
    export:
      prometheus:
        enabled: true  # Prometheus 메트릭 활성화
  endpoint:
    prometheus:
      enabled: true  # Prometheus 엔드포인트 활성화
```

3-3. stock서버 properties

```
spring:
  application:
    name: sog

  datasource:
    url: jdbc:mysql://localhost:3306/sog_db
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root
    password: rootpassword
  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQLDialect
        jdbc:
          time_zone: UTC
    show-sql: true
  data:
    redis:
      host: redis
      port: 6379

logging:
  level:
    org:
```

```
      hibernate:
        SQL: debug

kis:
  realtime-stock:
    appkey: "PSji3Tn79KSAKz0msD1dldUIXNre9JhlcJ2M"
    appsecret: "q5OxO241o89X0qoz+6i0Zh7+Xvg4nSHEvFdpU7KYJECpn
  chart:
    appkey: "PSTVy2mFMMrHQ97LAnfgRVcI5zRr7p3JGOxo"
    appsecret: "JmbcXXasdhSxCWqzQDV3/2BG5plUGGL+udWU+uL+coAQ0

server:
  error:
    include-message: always
    include-binding-errors: always
```

3-4. gateway서버 properties

```
spring:
  cloud:
    gateway:
      filter:
        remove-non-proxy-headers:
          headers:
            - Proxy-Authenticate
            - Proxy-Authorization
            - Keep-Alive
            - TE
            - Trailer
            - Transfer-Encoding
      globalcors:
        corsConfigurations:
          '[/**]':
            allowedOrigins:
              - "https://ssafy11s.com"
              - "http://localhost:3000"
            allowedMethods:
              - GET
```

```
            - POST
            - PUT
            - DELETE
            - OPTIONS
          allowedHeaders:
            - "Content-Type"
            - "Authorization"
            - "X-Requested-With"
            - "Accept"
          allowCredentials: true
          maxAge: 3600
      discovery:
        locator:
          enabled: true
      routes:
        - id: authenticated-routes
          uri: http://user-service.backcd.svc.cluster.local:8
          predicates:
            - Path=/api/*/auth/**
          filters:
            - JwtAuthenticationFilter

        - id: gateway-status
          uri: http://localhost:8080
          predicates:
            - Path=/status

        - id: user-service
          uri: http://user-service.backcd.svc.cluster.local:8
          predicates:
            - Path=/api/user/**

        - id: stock-service
          uri: http://stock-service.backcd.svc.cluster.local:
          predicates:
            - Path=/api/stock/**

        - id: stock-rocket-service
```

```
          uri: http://stock-service.backcd.svc.cluster.local:
          predicates:
            - Path=/api/rockets/**

        - id: stock-service-websocket
          uri: ws://stock-service.backcd.svc.cluster.local:80
          predicates:
            - Path=/api/ws/**
          filters:
            - RemoveRequestHeader=Sec-WebSocket-Protocol
            - AddRequestHeader=Connection, Upgrade
            - AddRequestHeader=Upgrade, websocket

        - id: news-service
          uri: http://news-service.backcd.svc.cluster.local:8
          predicates:
            - Path=/api/news/**

        - id: user-service-swagger
          uri: http://user-service.backcd.svc.cluster.local:8
          predicates:
            - Path=/user-service/v3/api-docs
            - Path=/user-service/swagger-ui/**
          filters:
            - RewritePath=/user-service(?<segment>/.*), ${segm

        - id: stock-service-swagger
          uri: http://stock-service.backcd.svc.cluster.local:
          predicates:
            - Path=/stock-service/v3/api-docs
            - Path=/stock-service/swagger-ui/**
          filters:
            - RewritePath=/stock-service(?<segment>/.*), ${se

        - id: news-service-swagger
          uri: http://news-service.backcd.svc.cluster.local:8
          predicates:
            - Path=/news-service/v3/api-docs
```

```
            - Path=/news-service/swagger-ui/**
          filters:
            - RewritePath=/news-service(?<segment>/.*), ${segm

jwt:
  secret: ${JWT_SECRET:default-secret}

springdoc:
  swagger-ui:
    urls:
      - name: user-service
        url: /user-service/v3/api-docs
      - name: stock-service
        url: /stock-service/v3/api-docs
      - name: news-service
        url: /news-service/v3/api-docs

management:
  endpoints:
    web:
      exposure:
        include: gateway, health
  endpoint:
    health:
      show-details: always

logging:
  level:
    org.springframework.cloud.gateway: DEBUG

# Elastic APM 관련 설정
elastic:
  apm:
    service_name: "gateway-service" # APM에 등록될 서비스 이름
    server_urls: "http://3.38.183.146:8200" # APM 서버 URL
    application_packages: "com.sog" # 패키지 이름 설정
    enable_framework_auto_detection: true # WebFlux 자동 감지 활
    distributed_tracing: true # 분산 추적 활성화
```

```
    capture_body: "all" # 모든 요청 본문 캡처
    capture_headers: true # HTTP 헤더 캡처 활성화
```

3-1 . news서버 DockerFile

```
# openjdk 17 버전의 환경을 구성
FROM openjdk:17-jdk-slim

WORKDIR /app

# COPY만 docker-compose 파일의 위치를 기반으로 작동함
COPY . .

# 개행문자 오류 해결 [unix와 window 시스템 차이]
RUN sed -i 's/\r$//' gradlew

# RUN은 현재 파일을 위치를 기반으로 작동함
RUN chmod +x ./gradlew
RUN ./gradlew clean build -x test --stacktrace

RUN mv build/libs/news-0.0.1-SNAPSHOT.jar /app/app.jar

ENTRYPOINT ["java", "-jar", "-Dspring.profiles.active=dev", "
```

3-2 . stock서버 DockerFile

```
# openjdk 17 버전의 환경을 구성
FROM openjdk:17-jdk-slim

WORKDIR /app

# COPY만 docker-compose 파일의 위치를 기반으로 작동함
COPY . .

# 개행문자 오류 해결 [unix와 window 시스템 차이]
RUN sed -i 's/\r$//' gradlew
```

```
# RUN은 현재 파일을 위치를 기반으로 작동함
RUN chmod +x ./gradlew
RUN ./gradlew clean build -x test --stacktrace

RUN mv build/libs/stock-0.0.1-SNAPSHOT.jar /app/app.jar

ENTRYPOINT ["java", "-jar", "-Dspring.profiles.active=dev", "
```

3-3 . user서버 DockerFile

```
# openjdk 21 버전의 환경을 구성
FROM openjdk:17-jdk-slim

WORKDIR /app

# COPY만 docker-compose 파일의 위치를 기반으로 작동함
COPY . .

# 개행문자 오류 해결 [unix와 window 시스템 차이]
RUN sed -i 's/\r$//' gradlew

# RUN은 현재 파일을 위치를 기반으로 작동함
RUN chmod +x ./gradlew
RUN ./gradlew clean build -x test --stacktrace

RUN mv build/libs/user-0.0.1-SNAPSHOT.jar /app/app.jar

ENTRYPOINT ["java", "-jar", "-Dspring.profiles.active=dev", "
```

3-4. gateway서버 DockerFile

```
# openjdk 17 버전의 환경을 구성
FROM openjdk:17-jdk-slim

WORKDIR /app

# COPY만 docker-compose 파일의 위치를 기반으로 작동함
```

```
COPY . .

# 개행문자 오류 해결 [unix와 window 시스템 차이]
RUN sed -i 's/\r$//' gradlew

# RUN은 현재 파일을 위치를 기반으로 작동함
RUN chmod +x ./gradlew
RUN ./gradlew clean build -x test --stacktrace

RUN mv build/libs/gateway-0.0.1-SNAPSHOT.jar /app/app.jar

ENTRYPOINT ["java", "-jar", "-Dspring.profiles.active=dev", "
```

4. yaml

1) frontend k8s

- deployment.yml

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
  namespace: backcd
  annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/proxy-body-size: 50m
    kubernetes.io/ingress.class: nginx
spec:
  tls:
  - hosts:
    - ssafy11s.com
    secretName: argocd-server-tls
  rules:
  - host: ssafy11s.com
```

```
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
  namespace: backcd
  labels:
    app: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: zlxldgus123/frontend:771
        ports:
        - containerPort: 80
        resources:
          requests:
            memory: "512Mi"  # 최소 512Mi 메모리 요청
          limits:
            memory: "1Gi"    # 최대 1Gi 메모리 제한
```

```
        env:
        - name: NEXT_PUBLIC_GPT_API_KEY  # 추가된 부분
          valueFrom:
            secretKeyRef:
              name: gpt-api-secret
              key: gpt-api-key

---
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
  namespace: backcd
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30082  # 필요에 따라 변경 가능
```

2) backend k8s

- argocd-ingress.yaml

```
# ---
# apiVersion: networking.k8s.io/v1
# kind: Ingress
# metadata:
#   name: argocd-ingress
#   namespace: backcd
#   annotations:
#     cert-manager.io/cluster-issuer: "letsencrypt-prod"
#     nginx.ingress.kubernetes.io/ssl-redirect: "true"
#     nginx.ingress.kubernetes.io/proxy-body-size: 50m
```

```
#     kubernetes.io/ingress.class: nginx
# spec:
#   tls:
#   - hosts:
#     - ssafy11s.com
#     secretName: argocd-server-tls
#   rules:
#   - host: ssafy11s.com
#     http:
#       paths:
#       - path: /
#         pathType: Prefix
#         backend:
#           service:
#             name: argocd-server
#             port:
#               number: 80
```

- gateway-service-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gateway-service
  namespace: backcd
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gateway-service
  template:
    metadata:
      labels:
        app: gateway-service
    spec:
      securityContext:
        runAsUser: 1000  # 컨테이너에서 사용하는 사용자 ID
        runAsGroup: 1000  # 그룹 ID
```

```
        fsGroup: 1000     # 볼륨에 접근할 때 사용할 그룹 ID
      initContainers:
        - name: init-apm-agent
          image: curlimages/curl:latest  # Curl 이미지를 사용
          command:
            - "sh"
            - "-c"
            - >
              curl -o /elastic/elastic-apm-agent.jar https
          volumeMounts:
            - name: elastic-apm-volume
              mountPath: /elastic  # APM 에이전트 파일을 저장할
      containers:
        - name: gateway-service
          image: zlxldgus123/gateway-service:287
          ports:
            - containerPort: 8080
          env:
            - name: JWT_SECRET
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: secret
            - name: JAVA_TOOL_OPTIONS
              value: "-javaagent:/elastic/elastic-apm-agen
          volumeMounts:
            - name: elastic-apm-volume
              mountPath: /elastic  # APM 에이전트가 위치할 경로
          resources:
            requests:
              memory: "250Mi"  # 최소 메모리 요청
            limits:
              memory: "500Mi"  # 최대 메모리 제한
      volumes:
        - name: elastic-apm-volume
          emptyDir: {}  # 파드 내 임시 디렉터리 사용
```

- gateway-service-hpa.yaml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: gateway-service-hpa
  namespace: backcd
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: gateway
  minReplicas: 1  # 최소 2개의 파드를 유지
  maxReplicas: 1 # 최대 10개의 파드로 확장
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50  # CPU 사용량이 50%를 초과할
```

- gateway-service-ingress.yaml

```
#apiVersion: networking.k8s.io/v1
#kind: Ingress
#metadata:
#  name: gateway-service-ingress
#  namespace: backcd
#  annotations:
#    nginx.ingress.kubernetes.io/rewrite-target: /
#spec:
#  tls:
#    - hosts:
#        - ssafy11s.com
#      secretName: ssafy11s-tls  # TLS 인증서가 저장된 Secret
#  rules:
#    - host: ssafy11s.com
#      http:
```

```
#        paths:
#          - path: /api/user
#            pathType: Prefix
#            backend:
#              service:
#                name: user-service  # 올바른 서비스 이름으로 수
#                port:
#                  number: 80
#          - path: /api/stock
#            pathType: Prefix
#            backend:
#              service:
#                name: stock-service  # 올바른 서비스 이름으로
#                port:
#                  number: 80
#          - path: /api/news
#            pathType: Prefix
#            backend:
#              service:
#                name: news-service  # 올바른 서비스 이름으로 수
#                port:
#                  number: 80
```

- gateway-service-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: gateway-service
  namespace: backcd
spec:
  type: ClusterIP  # NodePort에서 ClusterIP로 변경
  selector:
    app: gateway-service
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080  # 클러스터 내부에서 사용할 포트 매핑
```

- news-service-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: news-service
  namespace: backcd
spec:
  replicas: 2  # 기본 레플리카 수, HPA 사용 시 동적으로 조정 가능
  selector:
    matchLabels:
      app: news-service
  template:
    metadata:
      labels:
        app: news-service
    spec:
      securityContext:
        runAsUser: 1000  # 컨테이너에서 사용하는 사용자 ID
        runAsGroup: 1000  # 그룹 ID
        fsGroup: 1000     # 볼륨에 접근할 때 사용할 그룹 ID
      initContainers:
        - name: init-apm-agent
          image: curlimages/curl:latest  # Curl 이미지를 사용
          command:
            - "sh"
            - "-c"
            - >
              curl -o /elastic/elastic-apm-agent.jar https
          volumeMounts:
            - name: elastic-apm-volume
              mountPath: /elastic  # APM 에이전트 파일을 저장할
      containers:
        - name: news-service
          image: zlxldgus123/news-service:287
          ports:
            - containerPort: 8080
          env:
```

```
            - name: SPRING_DATASOURCE_URL
              value: jdbc:mysql://mysql:3306/sog_db
            - name: SPRING_DATASOURCE_USERNAME
              valueFrom:
                secretKeyRef:
                  name: db-secrets
                  key: username
            - name: SPRING_DATASOURCE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: db-secrets
                  key: password
            - name: JWT_SECRET
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: secret
            - name: JAVA_TOOL_OPTIONS
              value: "-javaagent:/elastic/elastic-apm-agen
          volumeMounts:
            - name: elastic-apm-volume
              mountPath: /elastic  # APM 에이전트가 위치할 경로
          resources:  # 리소스 요청 및 제한 설정
            requests:
              memory: "512Mi"
              cpu: "250m"
            limits:
              memory: "512Mi"
              cpu: "500m"
      volumes:
        - name: elastic-apm-volume
          emptyDir: {}  # 파드 내 임시 디렉터리 사용
```

- news-service-hpa.yaml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
```

```
  name: news-service-hpa
  namespace: backcd
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: news-service
  minReplicas: 2  # 최소 2개의 파드를 유지
  maxReplicas: 4 # 최대 10개의 파드로 확장
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50  # CPU 사용량이 50%를 초과할
```

- news-service-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: news-service
  namespace: backcd
spec:
  type: ClusterIP  # NodePort에서 ClusterIP로 변경
  selector:
    app: news-service
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080  # 클러스터 내부에서 사용할 포트 매핑
```

- secrets.yaml

```
---
apiVersion: v1
```

```
kind: Secret
metadata:
  name: db-secrets
  namespace: backcd
type: Opaque
stringData:
  username: root  # 실제 DB 사용자로 변경
  password: rootpassword  # 실제 DB 비밀번호로 변경

---
apiVersion: v1
kind: Secret
metadata:
  name: jwt-secret
  namespace: backcd
type: Opaque
stringData:
  secret: "3c84e62c46e5438e9d12e28a9876591bc34e4e92a3a489b
```

- stock-service-deployment.yaml

```
---
apiVersion: v1
kind: Secret
metadata:
  name: db-secrets
  namespace: backcd
type: Opaque
stringData:
  username: root  # 실제 DB 사용자로 변경
  password: rootpassword  # 실제 DB 비밀번호로 변경

---
apiVersion: v1
kind: Secret
metadata:
```

```
  name: jwt-secret
  namespace: backcd
type: Opaque
stringData:
  secret: "3c84e62c46e5438e9d12e28a9876591bc34e4e92a3a489b
```

- stock-service-hpa.yaml

```
# apiVersion: autoscaling/v2
# kind: HorizontalPodAutoscaler
# metadata:
#   name: stock-service-hpa
#   namespace: backcd
# spec:
#   scaleTargetRef:
#     apiVersion: apps/v1
#     kind: Deployment
#     name: stock-service
#   minReplicas: 1  # 최소 2개의 파드를 유지
#   maxReplicas: 1 # 최대 10개의 파드로 확장
```

- stock-service-service.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: stock-service
  namespace: backcd
spec:
  type: NodePort
  selector:
    app: stock-service
  ports:
    - protocol: TCP
      port: 80
```

```
      targetPort: 8080
      nodePort: 30085  # NodePort는 기존 30081을 사용
```

- user-service-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
  namespace: backcd
spec:
  replicas: 1  # 기본 레플리카 수, HPA 사용 시 동적으로 조정 가능
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      securityContext:
        runAsUser: 1000  # 컨테이너에서 사용하는 사용자 ID
        runAsGroup: 1000  # 그룹 ID
        fsGroup: 1000     # 볼륨에 접근할 때 사용할 그룹 ID
      initContainers:
        - name: init-apm-agent
          image: curlimages/curl:latest  # Curl 이미지를 사용
          command:
            - "sh"
            - "-c"
            - >
              curl -o /elastic/elastic-apm-agent.jar https
          volumeMounts:
            - name: elastic-apm-volume
              mountPath: /elastic  # APM 에이전트 파일을 저장할
      containers:
        - name: user-service
          image: zlxldgus123/user-service:287
```

```
          ports:
            - containerPort: 8080
          env:
            - name: SPRING_DATASOURCE_URL
              value: jdbc:mysql://mysql:3306/sog_db
            - name: SPRING_DATASOURCE_USERNAME
              valueFrom:
                secretKeyRef:
                  name: db-secrets
                  key: username
            - name: SPRING_DATASOURCE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: db-secrets
                  key: password
            - name: JWT_SECRET
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: secret
            - name: JAVA_TOOL_OPTIONS
              value: "-javaagent:/elastic/elastic-apm-agen
          volumeMounts:
            - name: elastic-apm-volume
              mountPath: /elastic  # APM 에이전트가 위치할 경로
          resources:  # 리소스 요청 및 제한 설정
            requests:
              memory: "256Mi"
              cpu: "125m"
            limits:
              memory: "500Mi"
              cpu: "250m"
      volumes:
        - name: elastic-apm-volume
          emptyDir: {}  # 파드 내 임시 디렉터리 사용
```

- user-service-hpa.yaml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: user-service-hpa
  namespace: backcd
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: user-service
  minReplicas: 1  # 최소 2개의 파드를 유지
  maxReplicas: 2 # 최대 10개의 파드로 확장
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50  # CPU 사용량이 50%를 초과할
```

- user-service-ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: gateway-ingress
  namespace: backcd
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/proxy-read-timeout: "300"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "300"
    nginx.ingress.kubernetes.io/proxy-connect-timeout: "30
    nginx.ingress.kubernetes.io/proxy-http-version: "1.1"
    nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
spec:
  ingressClassName: nginx  # Ingress 클래스 명시적으로 추가
  rules:
```

```
    - host: ssafy11s.com
      http:
        paths:
          - path: /api/ws
            pathType: Prefix
            backend:
              service:
                name: gateway-service
                port:
                  number: 80
          - path: /api/user
            pathType: Prefix
            backend:
              service:
                name: gateway-service
                port:
                  number: 80
          - path: /api/stock
            pathType: Prefix
            backend:
              service:
                name: gateway-service
                port:
                  number: 80
          - path: /api/news
            pathType: Prefix
            backend:
              service:
                name: gateway-service
                port:
                  number: 80
          - path: /user-service/swagger-ui
            pathType: Prefix
            backend:
              service:
                name: gateway-service
                port:
                  number: 80
```

```
          - path: /stock-service/swagger-ui
            pathType: Prefix
            backend:
              service:
                name: gateway-service
                port:
                  number: 80
          - path: /news-service/swagger-ui
            pathType: Prefix
            backend:
              service:
                name: gateway-service
                port:
                  number: 80
          # ELK 스택 추가
          - path: /kibana
            pathType: Prefix
            backend:
              service:
                name: kibana
                port:
                  number: 5601
          - path: /elasticsearch
            pathType: Prefix
            backend:
              service:
                name: elasticsearch
                port:
                  number: 9200
```

- user-service-service.yaml

```
# ---
# apiVersion: v1
# kind: Service
# metadata:
#   name: user-service
#   namespace: backcd
```

```
# spec:
#   type: NodePort
#   selector:
#     app: user-service
#   ports:
#     - protocol: TCP
#       port: 80
#       targetPort: 8080
#       nodePort: 30081  # NodePort는 기존 30081을 사용

apiVersion: v1
kind: Service
metadata:
  name: user-service
  namespace: backcd
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: user-service
```

3) manifests

- apm-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: apmserver-config
  namespace: dev
  labels:
    app.kubernetes.io/name: apmserver
data:
  apm-server.yml: |-
    apm-server:
      host: "0.0.0.0:8200"
```

```
      # kibana:
      #   enabled: true
      #   host: "http://10.108.247.129:5601"  # Kibana 호스
      rum:
        enabled: true
    output.elasticsearch:
      hosts: ["http://10.109.249.133:9200"]
      username: "elastic"
      password: "ua8747ll"
    logging:
      level: "info"
    setup.template.enabled: true
```

- apm-deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apm-server
  namespace: dev
  labels:
    app.kubernetes.io/name: apmserver
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: apmserver
  template:
    metadata:
      labels:
        app.kubernetes.io/name: apmserver
    spec:
      containers:
      - name: apmserver
        image: docker.elastic.co/apm/apm-server:7.15.0
        ports:
        - containerPort: 8200
          name: http
```

```
          protocol: TCP
        resources: {}
        volumeMounts:
        - mountPath: /usr/share/apm-server/apm-server.yml
          name: apmserver-config
          readOnly: true
          subPath: apm-server.yml
      volumes:
      - configMap:
          defaultMode: 420
          name: apmserver-config
        name: apmserver-config
---
apiVersion: v1
kind: Service
metadata:
  name: apmserver-nodeport
  namespace: dev
spec:
  type: NodePort
  selector:
    app.kubernetes.io/name: apmserver
  ports:
  - name: apmserver-nodeport
    nodePort: 30190  # 포트 변경
    port: 8200
    protocol: TCP
    targetPort: 8200
```

- apm-server.yaml

```
# apiVersion: apps/v1
# kind: Deployment
# metadata:
#   name: apm-server
#   namespace: dev
#   labels:
#     app: apm-server
```

```
# spec:
#   replicas: 1
#   selector:
#     matchLabels:
#       app: apm-server
#   template:
#     metadata:
#       labels:
#         app: apm-server
#     spec:
#       containers:
#         - name: apm-server
#           image: docker.elastic.co/apm/apm-server:7.15.0
#           ports:
#             - containerPort: 8200
#           env:
#             - name: ELASTICSEARCH_HOSTS
#               value: "http://10.109.249.133:9200"
#             - name: ELASTIC_APM_USERNAME  # 사용자 이름 설정
#               value: "elastic"
#             - name: ELASTIC_APM_PASSWORD  # 비밀번호 설정
#               value: "ua8747ll"
#             - name: APM_SERVER_HOST
#               value: "0.0.0.0:8200"
#             - name: LOGGING_LEVEL
#               value: "info"
#             - name: SETUP_TEMPLATE_ENABLED
#               value: "true"
#           resources:
#             requests:
#               memory: "250Mi"  # 최소 250Mi 메모리 요청
#             limits:
#               memory: "500Mi"  # 최대 메모리 500Mi로 제한
# ---
# apiVersion: v1
# kind: Service
# metadata:
#   name: apm-server
```

```
#   namespace: dev  # 네임스페이스 변경 가능
# spec:
#   selector:
#     app: apm-server
#   type: NodePort
#   ports:
#     - protocol: TCP
#       port: 8200          # APM 서버의 기본 포트
#       targetPort: 8200     # 컨테이너 내부의 포트
#       nodePort: 30880      # NodePort로 노출할 포트 (30000-
```

- elastic-apm-pv-pvc.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: elastic-apm-pv
  namespace: backcd
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  hostPath:
    path: "/mnt/data/elastic-apm"  # 노드의 특정 경로
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: elastic-apm-pvc
  namespace: backcd
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
```

```
      storage: 1Gi
  storageClassName: standard
```

- elasticsearch-pv-pvc.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: elasticsearch-pv
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: "/mnt/data/elasticsearch"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: elasticsearch-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  storageClassName: standard
```

- elasticsearch-statefulset.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: elasticsearch
spec:
```

```
  serviceName: "elasticsearch"
  replicas: 1
  selector:
    matchLabels:
      app: elasticsearch
  template:
    metadata:
      labels:
        app: elasticsearch
    spec:
      securityContext:
        runAsUser: 1001        # elasticsearch 사용자 UID
        fsGroup: 1001          # elasticsearch 그룹 GID
      containers:
        - name: elasticsearch
          image: docker.elastic.co/elasticsearch/elasticse
          ports:
            - containerPort: 9200
              name: http
          volumeMounts:
            - name: elasticsearch-data
              mountPath: /var/lib/elasticsearch/data   # 더
          env:
            - name: discovery.type
              value: single-node
            - name: xpack.security.enabled
              value: "true"  # 보안 활성화
            - name: xpack.security.authc.api_key.enabled
              value: "true"  # API 키 인증 활성화
          resources:
            requests:
              memory: "3Gi"   # 메모리 요청량 (1기가)
              cpu: "1"     # CPU 요청량
            limits:
              memory: "3Gi"   # 메모리 제한량 (1기가)
              cpu: "2"        # CPU 제한량
      volumes:
        - name: elasticsearch-data
```

```
          persistentVolumeClaim:
            claimName: elasticsearch-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
spec:
  type: NodePort
  ports:
    - port: 9200
      targetPort: 9200
      nodePort: 30193
  selector:
    app: elasticsearch
```

- kafka-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kafka-deployment
  namespace: backcd  # 네임스페이스 지정
spec:
  replicas: 1  # 필요 시 레플리카 수 조정
  selector:
    matchLabels:
      app: kafka
  template:
    metadata:
      labels:
        app: kafka
    spec:
      containers:
        - name: kafka
          image: bitnami/kafka:latest  # Bitnami Kafka 이미
          ports:
            - containerPort: 9092  # Kafka 기본 포트
```

```
          env:
            # Zookeeper 연결 설정
            - name: KAFKA_CFG_ZOOKEEPER_CONNECT
              value: "zookeeper-service:2181"
            # 외부에서 Kafka에 접근할 수 있도록 advertised.liste
            - name: KAFKA_CFG_ADVERTISED_LISTENERS
              value: "PLAINTEXT://3.38.183.146:30092"  # I
            # 인증 없이 접근 허용 (로컬 테스트 환경)
            - name: ALLOW_PLAINTEXT_LISTENER
              value: "yes"
          resources:
            requests:
              memory: "512Mi"  # 최소 메모리 요청
            limits:
              memory: "1Gi"    # 최대 메모리 사용량 1Gi로 제한
          volumeMounts:
            - name: kafka-storage
              mountPath: /bitnami/kafka  # Kafka 데이터 저장
      volumes:
        - name: kafka-storage
          persistentVolumeClaim:
            claimName: kafka-pvc  # PVC에 맞게 설정


---
apiVersion: v1
kind: Service
metadata:
  name: kafka-service
  namespace: backcd  # 네임스페이스 지정
spec:
  type: NodePort
  ports:
    - port: 9092  # Kafka 기본 포트
      targetPort: 9092
      nodePort: 30092  # 외부 노출을 위한 포트
  selector:
    app: kafka
```

- kafka-pv-pvc.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: kafka-pv-node1
  namespace: backcd  # 네임스페이스 지정
spec:
  capacity:
    storage: 50Gi  # Kafka의 용량에 맞게 조정
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: "/mnt/data/kafka"  # Kafka의 로컬 디스크 경로
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - ip-172-26-0-238  # 특정 노드에 바인딩
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: kafka-pvc
  namespace: backcd  # 네임스페이스 지정
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  resources:
    requests:
      storage: 50Gi  # 요청 용량
```

- kibana-config.yaml

```
# apiVersion: v1
# kind: ConfigMap
# metadata:
#   name: kibana-config
#   namespace: dev
# data:
#   kibana.yml: |
#     server.host: "0.0.0.0"
#     server.port: 5601
#     elasticsearch.hosts: ["http://3.38.183.146:30193"]
#     elasticsearch.username: "kibana_system"
#     elasticsearch.password: "ua8747ll"  # 실제 환경에서는 S
#     xpack.security.enabled: true
#     xpack.encryptedSavedObjects.encryptionKey: "wEn9kTt4
```

- kibana-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kibana
  namespace: dev  # 필요한 네임스페이스로 수정
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kibana
  template:
    metadata:
      labels:
        app: kibana
    spec:
      containers:
        - name: kibana
          image: docker.elastic.co/kibana/kibana:8.15.2
          ports:
```

```
            - containerPort: 5601
          env:
            - name: ELASTICSEARCH_HOSTS
              value: "http://elasticsearch:9200"
            - name: ELASTICSEARCH_USERNAME
              value: "kibana_system"  # 설정한 사용자 이름
            - name: ELASTICSEARCH_PASSWORD
              value: "ua8747ll"   # 설정한 비밀번호

---
apiVersion: v1
kind: Service
metadata:
  name: kibana
  namespace: dev  # 필요한 네임스페이스로 수정
spec:
  type: NodePort
  ports:
    - port: 5601
      targetPort: 5601
      nodePort: 30056
  selector:
    app: kibana
```

- mongodb-pv-pvc-deployment.yaml

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pv-claim
  namespace: backcd
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

```
  storageClassName: standard
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv-node1
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: "/mnt/data/mongodb"  # MongoDB 데이터를 저장할 로컬
  nodeAffinity:  # 특정 노드에 바인딩
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - ip-172-26-0-238  # 특정 노드 IP 설정
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb
  namespace: backcd
  labels:
    app: mongodb
spec:
  replicas: 1  # 복제본이 1개로 설정되어 있지만 필요시 복제본을 늘릴
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
```

```
      labels:
        app: mongodb
    spec:
      containers:
      - name: mongodb
        image: mongo:5.0  # MongoDB 5.0 이미지 사용
        ports:
        - containerPort: 27017
        volumeMounts:
        - name: mongodb-storage
          mountPath: /data/db  # MongoDB의 데이터 저장 경로
        resources:  # 리소스 요청과 제한을 설정하여 안정적인 자원 할
          requests:
            memory: "512Mi"
            cpu: "0.5"
          limits:
            memory: "1Gi"
            cpu: "0.5"
      volumes:
      - name: mongodb-storage
        persistentVolumeClaim:
          claimName: mongodb-pv-claim
---
apiVersion: v1
kind: Service
metadata:
  name: mongodb-service
  namespace: backcd
spec:
  type: NodePort  # ClusterIP 대신 NodePort로 변경하여 외부 접근
  selector:
    app: mongodb
  ports:
    - protocol: TCP
      port: 27017  # MongoDB 기본 포트
      targetPort: 27017
      nodePort: 30017  # NodePort 번호 30017로 설정
```

- mysql-pod.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-config
  namespace: backcd  # 네임스페이스 추가
data:
  my.cnf: |
    [mysqld]
    ft_min_word_len = 2
    ft_stopword_file = ""
    innodb_buffer_pool_size = 2G  # InnoDB 버퍼 풀 크기 2GB로
    innodb_flush_log_at_trx_commit = 2  # 트랜잭션 커밋 시 로그

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-init-script
  namespace: backcd  # 네임스페이스 추가
data:
  init.sql: |
    CREATE DATABASE IF NOT EXISTS sog_db;

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: backcd  # 네임스페이스 추가
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
```

```
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:8.0
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "rootpassword"  # MySQL root 비밀번호
        - name: MYSQL_DATABASE
          value: "sog_db"  # 자동으로 생성할 데이터베이스 이름
        ports:
        - containerPort: 3306
        volumeMounts:
        - name: mysql-storage
          mountPath: /var/lib/mysql
        - name: mysql-init-script
          mountPath: /docker-entrypoint-initdb.d
        - name: mysql-config-volume
          mountPath: /etc/mysql/my.cnf  # ConfigMap을 MySQL
          subPath: my.cnf
        resources:
          limits:
            memory: "4Gi"  # 메모리 제한 4Gi
          requests:
            memory: "2Gi"  # 최소 요청 메모리 2Gi
      volumes:
      - name: mysql-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
      - name: mysql-init-script
        configMap:
          name: mysql-init-script
      - name: mysql-config-volume
        configMap:
          name: mysql-config
```

```
      nodeSelector:
        kubernetes.io/hostname: ip-172-26-0-238
      tolerations:
      - key: "node-role.kubernetes.io/control-plane"
        operator: "Exists"
        effect: "NoSchedule"
```

- mysql-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv-node1
spec:
  capacity:
    storage: 20Gi  # PVC의 요청에 맞게 설정
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: "/mnt/data/mysql"  # 로컬 디스크 경로
  nodeAffinity:  # 특정 노드에 바인딩
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - ip-172-26-0-238
```

- mysql-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  namespace: backcd
```

```
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  storageClassName: standard
```

- mysql-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: backcd
spec:
  ports:
  - port: 3306
    targetPort: 3306
    nodePort: 32000 # 원하는 포트를 설정 (옵션)
  selector:
    app: mysql
  type: NodePort
```

- redis_config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-config
  namespace: backcd  # 네임스페이스 추가
data:
  redis.conf: |
    bind 0.0.0.0
    protected-mode no

---
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: redis
  namespace: backcd  # 네임스페이스 추가
  labels:
    app: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:6.2
        ports:
        - containerPort: 6379
        volumeMounts:
        - name: redis-storage
          mountPath: /data
        - name: redis-config
          mountPath: /usr/local/etc/redis/redis.conf
          subPath: redis.conf
        args: ["redis-server", "/usr/local/etc/redis/redis
        resources:
          requests:
            memory: "250Mi"  # 최소 메모리 요청
          limits:
            memory: "500Mi"  # 최대 메모리 제한
      volumes:
      - name: redis-storage
        persistentVolumeClaim:
          claimName: redis-pv-claim
      - name: redis-config
```

```
        configMap:
          name: redis-config
      nodeSelector:
        kubernetes.io/hostname: ip-172-26-0-238
      tolerations:
      - key: "node-role.kubernetes.io/control-plane"
        operator: "Exists"
        effect: "NoSchedule"

---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: redis-pv-node1
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: "/mnt/data/redis"  # 로컬 디스크 경로
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - ip-172-26-0-238

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: redis-pv-claim
  namespace: backcd
```

```
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: standard

---
apiVersion: v1
kind: Service
metadata:
  name: redis
  namespace: backcd
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
```

- zookeeper-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zookeeper-deployment
  namespace: backcd  # 네임스페이스 지정
spec:
  replicas: 1  # 필요 시 레플리카 수 조정
  selector:
    matchLabels:
      app: zookeeper
  template:
    metadata:
      labels:
        app: zookeeper
    spec:
```

```
      containers:
        - name: zookeeper
          image: bitnami/zookeeper:latest  # Bitnami Zooke
          ports:
            - containerPort: 2181  # Zookeeper 기본 포트
          env:
            - name: ALLOW_ANONYMOUS_LOGIN
              value: "yes"  # 인증 없이 접근 허용
          volumeMounts:
            - name: zookeeper-storage
              mountPath: /bitnami/zookeeper  # Zookeeper 데
          securityContext:
            runAsUser: 0  # root 사용자로 실행
            runAsGroup: 0  # root 그룹으로 실행
          resources:
            requests:
              memory: "250Mi"  # 최소 메모리 요청
            limits:
              memory: "500Mi"  # 최대 메모리 제한
      volumes:
        - name: zookeeper-storage
          persistentVolumeClaim:
            claimName: zookeeper-pvc  # PVC 설정

---
apiVersion: v1
kind: Service
metadata:
  name: zookeeper-service
  namespace: backcd  # 네임스페이스 지정
spec:
  ports:
    - port: 2181  # Zookeeper 기본 포트
      targetPort: 2181
  selector:
    app: zookeeper
```

- zookeeper-pv-pvc.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: zookeeper-pv
  namespace: backcd  # 네임스페이스 지정
spec:
  capacity:
    storage: 10Gi  # Zookeeper 저장 용량 설정
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: "/mnt/data/zookeeper"  # Zookeeper의 로컬 저장 경로
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - ip-172-26-0-238  # 특정 노드 바인딩
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: zookeeper-pvc
  namespace: backcd  # 네임스페이스 지정
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  resources:
    requests:
      storage: 10Gi  # PVC 요청 용량
```

5. 포트 번호

```
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)           AGE
frontend-service    NodePort    10.111.175.143   <none>        80:30082/TCP      30d
gateway-service     ClusterIP   10.106.91.199    <none>        80/TCP            20d
kafka-service       NodePort    10.106.2.151     <none>        9092:30092/TCP    16d
mongodb-service     NodePort    10.110.123.148   <none>        27017:30017/TCP   8d
mysql               NodePort    10.101.37.173    <none>        3306:32000/TCP    30d
news-service        ClusterIP   10.96.42.53      <none>        80/TCP            21d
redis               ClusterIP   10.107.56.48     <none>        6379/TCP          29d
stock-service       NodePort    10.104.61.231    <none>        80:30085/TCP      20d
user-service        ClusterIP   10.98.52.6       <none>        80/TCP            2d23h
zookeeper-service   ClusterIP   10.98.1.211      <none>        2181/TCP          16d
```

```
apmserver-nodeport   NodePort   10.104.2.1       <none>        8200:30190/TCP   4d1h
elasticsearch        NodePort   10.109.249.133   <none>        9200:30193/TCP   15d
kibana               NodePort   10.108.247.129   <none>        5601:30056/TCP   15d
```

```
ubuntu@ip-172-26-0-238:~$ kubectl get service
NAME                TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
frontend-service    NodePort    10.99.131.8    <none>        80:30080/TCP     31d
kubernetes          ClusterIP   10.96.0.1      <none>        443/TCP          32d
```

6. 빌드 방법

1. 쿠버네티스 설치

2. 기본 패키지 설치

각 EC2 인스턴스에 SSH로 접속한 후 아래 패키지들을 설치합니다.

```
# 업데이트
sudo apt-get update
sudo apt-get upgrade -y

# Docker 설치 (Kubernetes에서 컨테이너 런타임으로 사용)
sudo apt-get install -y docker.io

# Kubernetes 패키지 설치를 위한 설정
sudo apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.g
```

```
pg | sudo apt-key add -
sudo bash -c 'cat <<EOF >/etc/apt/sources.list.d/kubernete
s.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF'
sudo apt-get update

# kubelet, kubeadm, kubectl 설치
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

3. 마스터 노드 초기화

마스터 노드에서 쿠버네티스를 초기화합니다. EC2 인스턴스의 IP 주소를 사용하여 클러스터 네트워크를 설정합니다.

```
sudo kubeadm init --apiserver-advertise-address=<마스터노드의
EC2 사설 IP> --pod-network-cidr=192.168.0.0/16
```

위 명령어가 완료되면 출력되는 `kubeadm join` 명령어는 워커 노드가 클러스터에 참여할 때 사용됩니다.

4. kubectl 설정 (마스터 노드)

마스터 노드에서 `kubectl`을 사용할 수 있도록 설정합니다.

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

5. CNI (Container Network Interface) 플러그인 설치

쿠버네티스 클러스터에 네트워크를 설정하기 위해 플러그인을 설치합니다. 여기서는 **Calico**를 사용합니다.

```
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

6. 워커 노드 연결

워커 노드에서 `kubeadm join` 명령어를 실행하여 클러스터에 참여시킵니다. 이 명령어는 마스터 노드 초기화 후 출력된 명령어입니다.

```
sudo kubeadm join <마스터노드의 EC2 사설 IP>:6443 --token <토큰> --discovery-token-ca-cert-hash sha256:<해시>
```

워커 노드가 클러스터에 성공적으로 참여하면, 마스터 노드에서 다음 명령어로 상태를 확인할 수 있습니다.

```
kubectl get nodes
```

이후 마스터노드에서 각 manifest를 apply하여 서버에 배포합니다.