

Course Project: Big Data Concepts and Implementations

Name: Abhishek Prasanna Walavalkar

Email: awalaval@iu.edu

Introduction

Step into the world of my data pipeline implementation project, where MongoDB serves as the backbone of my distributed database, complemented by a Spark cluster for distributed processing. My goal is to choreograph a smooth flow of data, streamlining extraction, transformation, and loading (ETL) processes. Through the periodic submission of Spark jobs, I delve into exploratory data analysis (EDA) and data transformation, unearthing insights and fine-tuning my data for deeper analysis. Come along as I navigate the realms of distributed computing and database management, crafting a resilient and flexible pipeline to handle my data with accuracy and agility.

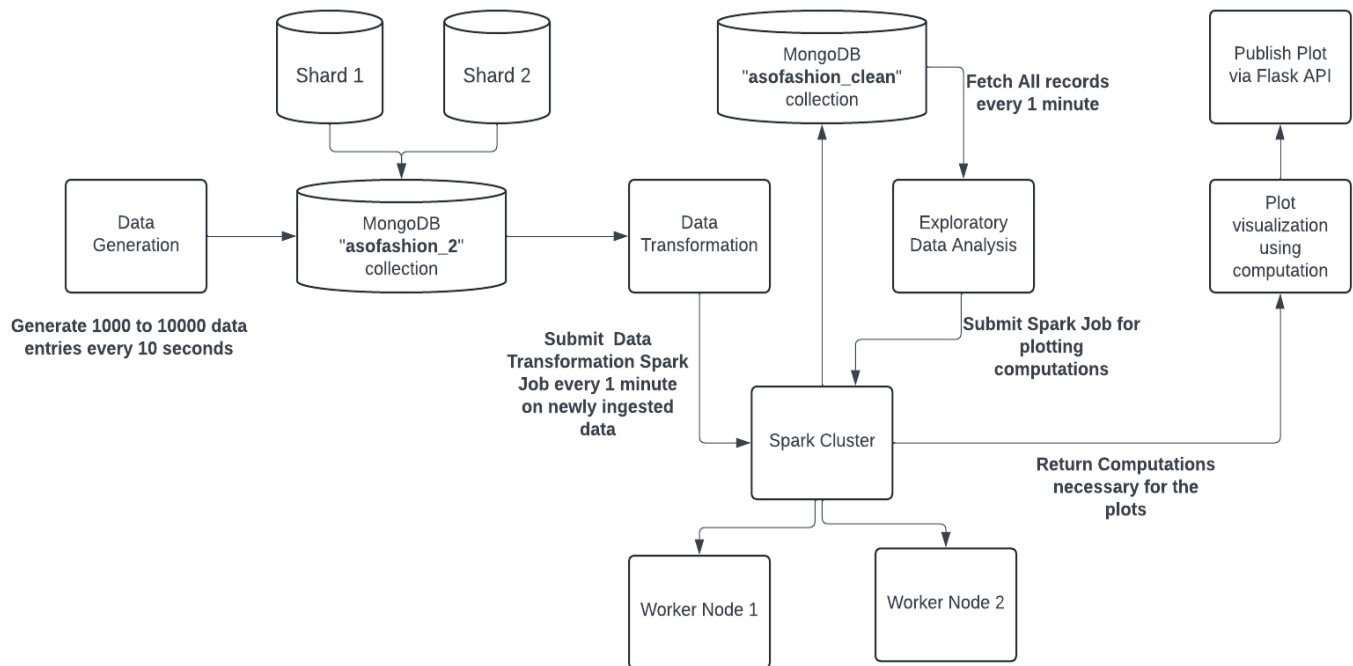
Background

In the dynamic landscape of fashion e-commerce, where trends shift rapidly and customer preferences evolve swiftly, the effective management and analysis of data play a critical role in staying ahead of the curve. From the intricate details of product catalogs to the nuanced insights derived from customer behavior, the breadth and depth of data generated in this domain are immense. To navigate this data deluge, a robust data pipeline tailored specifically for fashion products is indispensable.

Distributed storage solutions like MongoDB offer a scalable and adaptable framework for accommodating the diverse and voluminous datasets inherent to fashion e-commerce. By distributing data across multiple nodes, MongoDB ensures not only the seamless storage and retrieval of information but also guarantees high availability and fault tolerance, essential for maintaining uninterrupted operations in a dynamic online environment. This distributed approach empowers fashion businesses to store and manage vast amounts of data efficiently, providing a solid foundation for subsequent analysis and decision-making.

Complementing distributed storage, distributed processing platforms such as Apache Spark come to the fore, offering unparalleled capabilities in handling the computational demands of fashion e-commerce big data. With its parallel processing paradigm, Spark enables organizations to dissect and analyze massive datasets in parallel, significantly reducing processing times and unlocking insights at unprecedented speeds. Whether it's conducting real-time analysis of customer interactions or performing predictive analytics on market trends, Spark's distributed processing capabilities empower fashion e-commerce businesses to extract actionable insights swiftly, enabling them to adapt and respond to market dynamics with agility and precision.

Data Pipeline Flowchart



<https://github.iu.edu/awalaval/SP24-I535-awalaval-asofashion-data>

Methodology

The project methodology unfolded through well-defined stages, leveraging spark and distributed database cluster implemented using 3 of jet2stream instances named (abhi_1, abhi_2 and abhi_3)

Instance Specification:

OS: Ubuntu 22.04 (Jammy)

CPU: 16 cores

RAM: 60GB

Root Disk: 60GB

Phase 1: Data Generation

The dataset known as "asofashion data" comprises approximately 64,000 entries. Each entry encompasses attributes of products featured on the fashion category section of the ASOS website. Data is randomly selected in bulk and pushed to an "asofashion_2" sharded collection every 10 seconds. This emulates the actual traffic that occurs in real-world scenarios on a database, reflecting the volume and frequency of data accesses akin to those experienced in operational environments.

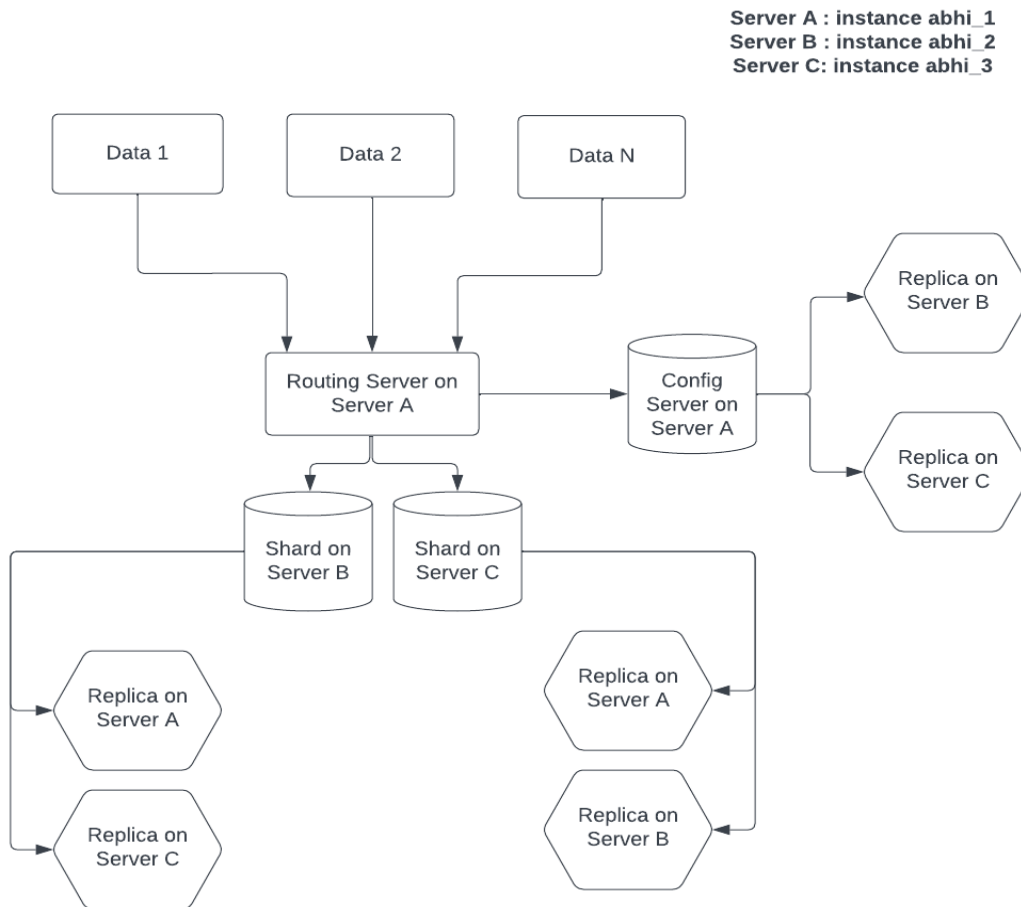
Phase 2: Cluster setup

The cluster goes beyond merely connecting instances physically; it involves establishing a network wherein they can exchange data and collaborate seamlessly, essentially functioning as a unified entity.

All three instances need to be linked and capable of communication without requiring passwords. Eliminating passwords simplifies communication in the cluster, making data exchange smoother and more efficient. This also improves security by incorporating alternative secure authentication methods.

Phase 3: Distributed MongoDB database setup

The MongoDB database is configured with sharding and replication. Each shard represents a portion of the database and is duplicated and distributed across the nodes.

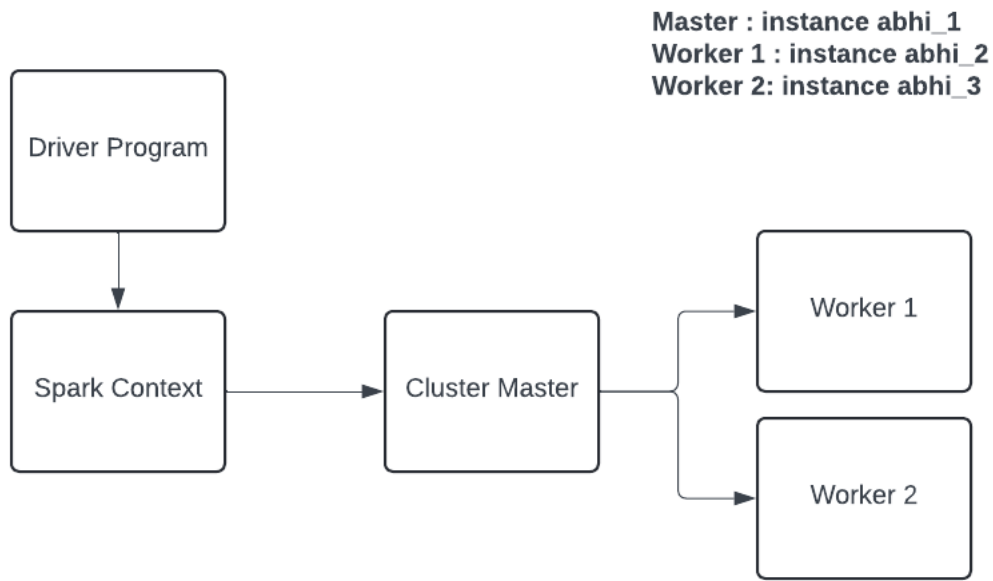


In MongoDB's sharding and replica architecture, the configuration server stores metadata about the cluster, while the routing server directs client queries to the appropriate shard. They work together to manage data distribution and query routing efficiently.

Sharding enables MongoDB to scale horizontally by distributing data across multiple servers, while replication ensures fault tolerance and high availability by maintaining multiple copies of data. Together, they enhance scalability, reliability, and performance in MongoDB deployments.

Phase 4: Spark Cluster Setup

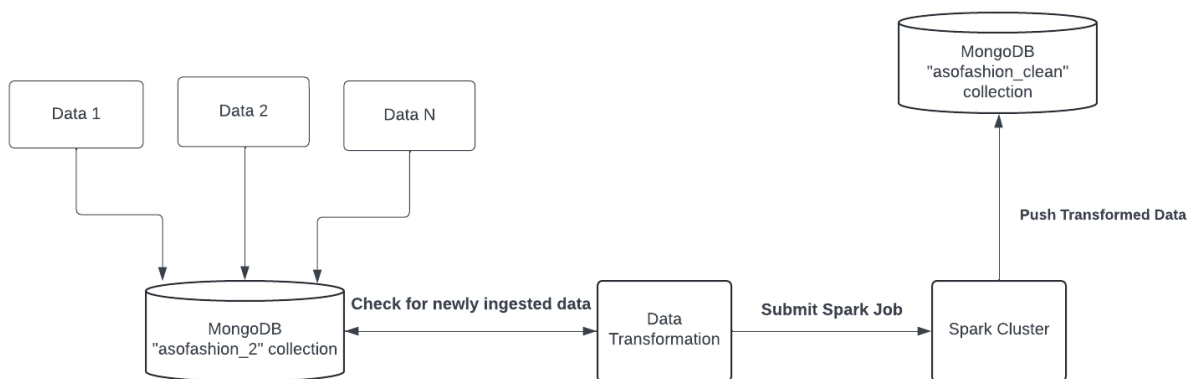
Spark clusters are established by utilizing three instances. Data cleaning and plotting computations are performed using PySpark on a deployed cluster. This process involves distributing the computational workload across the instances within the Spark cluster, leveraging the parallel processing capabilities of Spark to enhance efficiency and scalability in data processing tasks.



The process begins with the driver program, which initiates the Spark context and submits a Spark job to the cluster manager. Once the job is received, the cluster manager takes charge, utilizing worker executors to distribute the workload across the available workers for processing. This orchestration ensures that tasks are efficiently allocated and executed across the cluster, optimizing performance and resource utilization.

Phase 5: Data Transformation Module

The module is tasked with cleaning various attributes of newly ingested data. Specifically, it focuses on the string attribute labeled "title," as well as handling null values within the "current price" and "previous_price" attributes. The text cleaning aspect of this process encompasses several steps, including the removal of punctuation marks, trimming trailing white spaces, and converting text to lowercase for consistency.

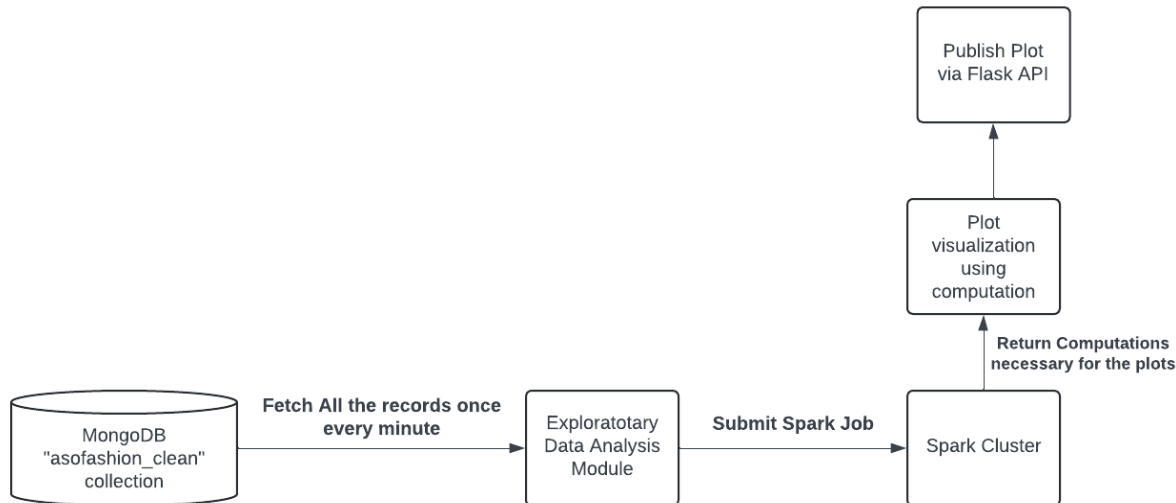


As this initial cleaning function lays the groundwork, it has the potential to undergo further evolution to incorporate more advanced preprocessing techniques. These could include generating embeddings to represent text data in a numerical format, tokenization for breaking text into smaller units like words or phrases, or even text summarization techniques to condense lengthy text passages into concise summaries. By incorporating these advanced methods, the cleaning module can enhance the quality and utility of the processed data for subsequent analysis or modeling tasks.

Phase 6: Exploratory Data Analysis Module:

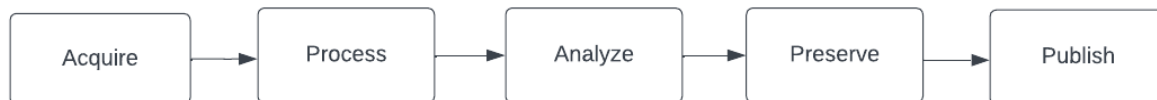
This module generates various plots and visual representations to illuminate patterns, outliers, and relationships within the data.

The visual representation includes a histogram illustrating the distribution of products based on their current price range. Additionally, there is a bar plot showcasing the frequency of products associated with the top 20 brands, offering insights into the popularity or prevalence of each brand within the dataset. Furthermore, another bar plot displays the distribution of products across different genders, highlighting the gender-specific composition of the dataset. Finally, a line plot depicts the trend of average current prices across various brands, providing a comprehensive overview of the pricing dynamics within the dataset.



Python's matplotlib library is used for plotting. It requires the entire dataset to be loaded into memory before generating plots, which is a limitation considering we are handling big data with millions of records. One strategy to address this challenge is to perform data pre-calculation within the Spark cluster and subsequently utilize the processed data for visualization with matplotlib.

Data Lifecycle:



Acquire Data: Fetch data from various sources and ensure it's structured for processing.

Store Data: Use a suitable database like MongoDB to store the acquired data.

Process and Analyze: Utilize Spark for distributed data processing, performing transformations and analysis.

Preserve Transformed Data: Save the processed data back into the database.

Publish Visualization: Create visualizations using tools like Matplotlib presenting insights from the transformed data for stakeholders.

Steps Performed:

Step 1] Configuring a Data Generation setup to mimic real-world data interactions with a database.

a) The following module selects 1,000 rows randomly and push them into the "asofashion_2" MongoDB distributed database.

```
jupyter data_load_random_100.py Last Checkpoint: 6 days ago
File Edit View Settings Help

1  #!/usr/bin/env python
2  # coding: utf-8
3
4
5
6  from pymongo import MongoClient
7  import pandas as pd
8
9  client = MongoClient("149.165.172.75", 60000)
10 db = client.shardDB
11
12 collection = db["asofashion_2"]
13 collection2 = db["asofashion_clean"]
14
15 data = pd.read_csv("/home/exouser/spark_pyjobs/data_generator/Asofashion.csv")
16 print(data.shape)
17 print(data.columns)
18 final_data = data[['product_id', 'brand_name', 'title', 'current_price',
19                   'previous_price', 'colour', 'currency', 'rrp', 'productCode',
20                   'productType', 'gender']].copy()
21
22 print(final_data.shape)
23 collection.insert_many(final_data.sample(1000).to_dict('records'))
```

b) Type crontab -e on a instance terminal. It opens file "/tmp/crontab.RTmaHh/crontab"
Add following lines in the file and save the file

```
* * * * * /home/exouser/jupEnv/bin/python /home/exouser/spark_pyjobs/data_generator/data_load_random_100.py
* * * * * sleep 10; /home/exouser/jupEnv/bin/python
/home/exouser/spark_pyjobs/data_generator/data_load_random_100.py
* * * * * sleep 20; /home/exouser/jupEnv/bin/python
/home/exouser/spark_pyjobs/data_generator/data_load_random_100.py
* * * * * sleep 30; /home/exouser/jupEnv/bin/python
/home/exouser/spark_pyjobs/data_generator/data_load_random_100.py
* * * * * sleep 40; /home/exouser/jupEnv/bin/python
/home/exouser/spark_pyjobs/data_generator/data_load_random_100.py
* * * * * sleep 50; /home/exouser/jupEnv/bin/python
/home/exouser/spark_pyjobs/data_generator/data_load_random_100.py
```

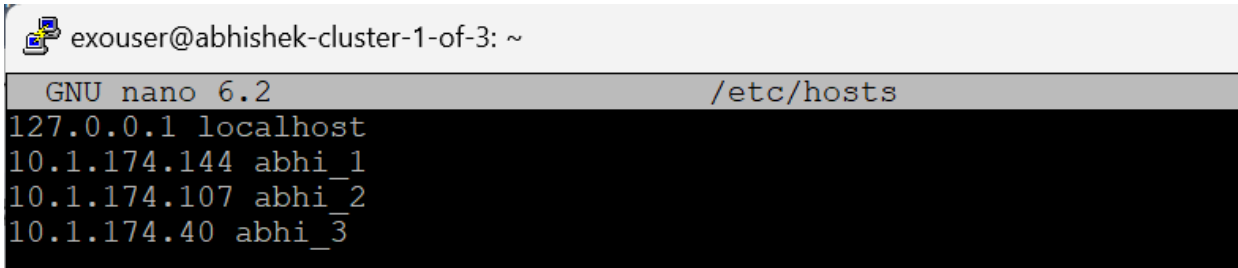
The above crontab will execute data generation file every 10 senconds

Step 2] Configuring Cluster setup

a) All Instances of the cluster should exist in the same network.

sudo vi /etc/hosts/

Write private IP address following instance alias in /etc/hosts of all instances.

A screenshot of a terminal window. The prompt is 'exouser@abhishek-cluster-1-of-3: ~'. The terminal shows the 'GNU nano 6.2' editor editing the file '/etc/hosts'. The content of the file is as follows:

```
127.0.0.1 localhost
10.1.174.144 abhi_1
10.1.174.107 abhi_2
10.1.174.40 abhi_3
```

b) Establishing a passwordless communication between instances of a cluster.

Generate Key Pair: *ssh-keygen -t rsa*

Create SSH directory on a server: *mkdir -p .ssh*

Upload Public Key to Remote Server:

ssh-copy-id exouser@abhi_2

ssh-copy-id exouser@abhi_3

Perform similar operations on all the instances of a cluster

Step 3] Configuring MongoDB distributed database setup

a) Install mongoDB on every instance

sudo apt-get install gnupg curl

*curl -fsSL https://www.mongodb.org/static/pgp/server-7.0.asc | *
*sudo gpg -o /usr/share/keyrings/mongodb-server-7.0.gpg *
--dearmor

echo "deb [arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-7.0.gpg]
https://repo.mongodb.org/apt/ubuntu jammy/mongodb-org/7.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-
7.0.list

sudo apt-get update

sudo apt-get install -y mongodb-org

b) Setup config servers

sudo mongod --configsvr --replSet cfgrs --port 40001 --dbpath dbdata/config_data/data --bind_ip abhi_1

sudo mongod --configsvr --replSet cfgrs --port 40002 --dbpath dbdata/config_data/data --bind_ip abhi_2

```
sudo mongod --configsvr --replSet cfgrs --port 40003 --dbpath dbdata/config_data/data --bind_ip abhi_3
```

c) open mongodb shell with “*mongosh --host abhi_1 --port 40001*”

Type the following to start the config servers

```
rs.initiate(  
  {  
    _id: "cfgrs",  
    configsvr: true,  
    members: [  
      { _id: 0, host: "abhi_1:40001" },  
      { _id: 1, host: "abhi_2:40002" },  
      { _id: 2, host: "abhi_3:40003" }  
    ]  
  }  
)
```

d) create a shard and its replica sets over instances in a cluster

```
sudo mongod --shardsvr --replSet shard1rs --port 50001 --dbpath dbdata/shard1/data --bind_ip abhi_1  
sudo mongod --shardsvr --replSet shard1rs --port 50002 --dbpath dbdata/shard1/data --bind_ip abhi_2  
sudo mongod --shardsvr --replSet shard1rs --port 50003 --dbpath dbdata/shard1/data --bind_ip abhi_3
```

e) Initiate shard1rs

```
mongosh --host abhi_1 --port 40001
```

```
rs.initiate(  
  {  
    _id: "shard1rs",  
    members: [  
      { _id: 0, host: "abhi_1:50001" },  
      { _id: 1, host: "abhi_2:50002" },  
      { _id: 2, host: "abhi_3:50003" }  
    ]  
  }  
)
```

f) Add shard 2

```
sudo mongod --shardsvr --replSet shard2rs --port 50004 --dbpath dbdata/shard2/data --bind_ip abhi_1  
sudo mongod --shardsvr --replSet shard2rs --port 50005 --dbpath dbdata/shard2/data --bind_ip abhi_2  
sudo mongod --shardsvr --replSet shard2rs --port 50006 --dbpath dbdata/shard2/data --bind_ip abhi_3
```

g) Initiate shard2rs

```
mongosh --host abhi_1 --port 50004
```



```
rs.initiate(
{
  _id: "shard2rs",
  members: [
    { _id: 0, host: "abhi_1:50004" },
    { _id: 1, host: "abhi_2:50005" },
    { _id: 2, host: "abhi_3:50006" }
  ]
}
)
```

h) Create a query Router

```
mongos --configdb cfgrs/abhi_1:40001,abhi_2:40002,abhi_3:40003 --bind_ip 0.0.0.0 --port 60000
```

i) Connect to a query router and add shards to a cluster

```
mongosh --host abhi_1 --port 60000
sh.addShard("shard1rs/abhi_1:50001,abhi_2:50002,abhi_3:50003")
sh.addShard("shard2rs/abhi_1:50004,abhi_2:50005,abhi_3:50006")
```

j) Shard a database

```
sh.enableSharding(shardDB)
sh.shardCollection("shardDB.asofashion_2",{"product_id":"hashed"})
```

```
[direct: mongos] shardDB> db.asofashion_2.getShardDistribution()
Shard shard2rs at shard2rs/abhi_1:50004,abhi_2:50005,abhi_3:50006
{
  data: '0B',
  docs: 0,
  chunks: 1,
  'estimated data per chunk': '0B',
  'estimated docs per chunk': 0
}
---
Shard shard1rs at shard1rs/abhi_1:50001,abhi_2:50002,abhi_3:50003
{
  data: '0B',
  docs: 0,
  chunks: 1,
  'estimated data per chunk': '0B',
  'estimated docs per chunk': 0
}
---
Totals
{
  data: '0B',
  docs: 0,
  chunks: 2,
  'Shard shard2rs': [ '0 % data', '0 % docs in cluster', '0B avg obj size on shard' ],
  'Shard shard1rs': [ '0 % data', '0 % docs in cluster', '0B avg obj size on shard' ]
}
```

Now the mongodb database asofashion_2 sharded and each shard has a replica set in a instances of a cluster.

Step 4] Establishing a Spark cluster by executing the following steps on each instance

a) Install spark-3.2.4.gz from <https://archive.apache.org/dist/spark/spark-3.2.4/>

```
tar -zxvf spark-3.2.4.gz
```

b) Install scala from <https://github.com/scala/scala/releases/tag/v2.13.13>

```
tar -zxvf scala-2.13.13.gz
```

c) Install Java https://download.oracle.com/java/21/archive/jdk-21.0.2_linux-x64_bin.tar.gz

```
tar -zxvf jdk-21.0.2_linux-x64_bin.tar.gz
```

d) add spark, java and scala paths in .bashrc

```
vi .bashrc
```

```
export PYSPARK_SUBMIT_ARGS="--master local[2] pyspark-shell"
export SPARK_HOME=/home/exouser/spark-3.2.4
export SCALA_HOME=/home/exouser/scala-2.13.13
export SPARK_LOCAL_HOSTNAME=localhost

export PATH=$PATH:$SPARK_HOME/bin:$SCALA_HOME/bin
echo "All Good server 1!"
```

e) Build Spark (<https://spark.apache.org/docs/latest/building-spark.html#apache-maven>)

```
export MAVEN_OPTS="-Xss64m -Xmx2g -XX:ReservedCodeCacheSize=1g"
```

```
cd spark-3.2.4
```

```
./build/mvn -DskipTests clean package
```

```
./dev/make-distribution.sh --name custom-spark --pip --r --tgz -Psparkr -Phive -Phive-thriftserver -Pmesos -Pyarn -Pkubernetes
```

f) Edit spark config files on master and worker nodes

```
cd spark-3.2.4/conf/
```

```
exouser@abhishek-cluster-1-of-3:~/spark-3.2.4/conf$ ls -l
total 52
-rw-r--r-- 1 exouser exouser 1105 Apr  9  2023 fairscheduler.xml.template
-rw-r--r-- 1 exouser exouser 2472 Apr 20 20:54 log4j.properties
-rw-r--r-- 1 exouser exouser 2471 Apr  9  2023 log4j.properties.template
-rw-r--r-- 1 exouser exouser 9141 Apr  9  2023 metrics.properties.template
-rw-r--r-- 1 exouser exouser 1292 Apr  9  2023 spark-defaults.conf.template
-rwxr-xr-x 1 exouser exouser 4820 Apr  7 15:31 spark-env.sh
-rwxr-xr-x 1 exouser exouser 4428 Apr  9  2023 spark-env.sh.template
-rw-r--r-- 1 exouser exouser  887 Apr  7 15:31 workers
-rw-r--r-- 1 exouser exouser  865 Apr  9  2023 workers.template
```

```
vi workers
```

Write public ip of workers instances on workers file

```
# A Spark Worker will be started
149.165.169.17
149.165.169.243
~
~
~
"workers" 20L, 887B
```

vi spark-env.sh

Add following configurations in a file on every instance

```
export SPARK_MASTER_HOST=10.1.174.144
export SPARK_LOCAL_IP=10.1.174.144
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
```

g) Start Spark Cluster

cd spark-3.2.4
sbin/start-all.sh

← → ↻

Not secure 149.165.172.75:8080

☆

🔍

📁

⋮

Dynamic Programm...

Careers

Label Studio Docum...

GitHub

Indiana University ...


Lobby | Top Hat

Instance Selection

Kaggle: Your Machi...

»

📁 All



3.2.4

Spark Master at spark://10.1.174.144:7077

URL: spark://10.1.174.144:7077

Alive Workers: 2

Cores in use: 32 Total, 0 Used

Memory in use: 115.7 GiB Total, 0.0 B Used

Resources in use:

Applications: 0 Running, 0 Completed

Drivers: 0 Running, 0 Completed

Status: ALIVE

▼ Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20240428015716-10.1.174.40-45333	10.1.174.40:45333	ALIVE	16 (0 Used)	57.8 GiB (0.0 B Used)	
worker-20240428015717-10.1.174.107-38441	10.1.174.107:38441	ALIVE	16 (0 Used)	57.8 GiB (0.0 B Used)	

▼ Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

▼ Completed Applications (0)

Step 5] Spark Job for Data transformation

The following code examines recently ingested data. Upon detecting new data, it triggers a Spark job to cleanse and migrate the data to the "asofashion_clean" collection.

The scheduler runs “*execute_spark_job()*” function after every 37 seconds.

```
from pymongo import MongoClient
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.context import SparkContext
import string
import schedule
import time
import os

def cleaning_func(x):

    # lower case
    # replace & with and
    # remove punctuation

    x["title"] = x["title"].lower().replace("&","and").translate(str.maketrans('', '', string.punctuation))
    if str(x["current_price"]).lower() == 'nan':
        x["current_price"] = 0

    if str(x["previous_price"]).lower() == 'nan':
        x["previous_price"] = 0

    return x

def execute_spark_job():
    # print("----- Job")
    client = MongoClient("149.165.172.75", 60000)
    db = client.shardDB

    c1 = db["asofashion_2"]
    c2 = db["asofashion_clean"]

    k = c1.aggregate([
        {"$match":{"_id":{"$exists":1}}},
        {"$lookup":{"from": "asofashion_clean",
                    "localField":"_id",
                    'foreignField' : '_id',
                    "as":"missedWallet"
                }},
        {"$match":{"missedWallet.0":{"$exists":0}}}
    ]);
```

```

input_data = list(k)
if(len(input_data) != 0 ):

    spark = SparkSession \
        .builder \
        .appName("mongodbttest1") \
        .master("spark://10.1.174.144:7077")\
        .getOrCreate()

    clean_doc = spark.sparkContext.parallelize(input_data).map(cleaning_func).collect()
    # print(len(clean_doc))
    c2.insert_many(clean_doc)
    spark.stop()

else:
    pass
    # print("data is upto date")

schedule.every(37).seconds.do(execute_spark_job)

while 1:
    schedule.run_pending()
    time.sleep(1)

```

Step 6] Exploratory Data Analysis

The code retrieves all records from the "asofashion_clean" dataset and conducts exploratory analysis. Spark cluster handles the computations needed for plotting due to the large dataset. This function is triggered upon a GET request to <https://large-bee-oddly.ngrok-free.app/dashboard>

```

from flask import Flask, request, send_file, make_response
from flask_restful import Resource, Api
import base64
from pyngrok import ngrok

import pyspark
from pyspark.sql import SparkSession
from pyspark.sql import Row

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import io

app = Flask(__name__)
api = Api(app)

class Picture(Resource):
    def get(self):
        spark = SparkSession \
            .builder \
            .appName("mongodbttest1") \
            .master("spark://10.1.174.144:7077")\
            .config("spark.mongodb.read.uri", "mongodb://abhi_1:60000/shardDB.asofashion_2") \
            .config("spark.mongodb.write.uri", "mongodb://abhi_1:60000/shardDB.asofashion_clean") \
            .config('spark.jars.packages', 'org.mongodb.spark:mongo-spark-connector_2.13:10.2.2') \
            .getOrCreate()

```

```

df = spark.read\
    .format("mongodb")\
    .option("connection.uri", "mongodb://abhi_1:60000/")\
    .option("database", "shardDB")\
    .option("collection", "asofashion_clean")\
    .load()

vals = df.select("current_price").rdd.flatMap(lambda x: x).histogram(20)
classes = df.groupBy("brand_name").count().orderBy('count', ascending=False)
classes2 = df.groupBy("gender").count().orderBy('count', ascending=False)
line = df.groupBy("brand_name").mean("current_price").toPandas().sort_values(by='avg({})'.format("current_price"))

```

```

# gridspec = dict(hspace=1, height_ratios=[0, 1, 0.4, 3])
fig, ax = plt.subplots(nrows=4, ncols=1, figsize=(15,18), constrained_layout=True)

```

Histogram in Spark

```

width = vals[0][1] - vals[0][0]
loc = [vals[0][0] + (i+1) * width for i in range(len(vals[1]))]
ax[0].bar(loc, vals[1], width=width)
ax[0].set_xlabel("Current Price")
ax[0].set_ylabel("Number fo Products")

```

#Barplot in Spark

```

pd_df = classes.limit(20).toPandas()
pd_df.plot(kind='bar', x="brand_name", legend=True, ax=ax[1])
ax[1].set_ylabel("Number of products per Brand")
ax[1].set_xlabel("Top 20 Brand Names")

```

```

pd_df = classes2.limit(20).toPandas()
pd_df.plot(kind='bar', x="gender", legend=True, ax=ax[2])
ax[2].set_ylabel("Number of products per Gender")
ax[2].set_xlabel("Gender")

```

#Lineplot in Spark

```

pd_df = line
pd_df.plot.line("brand_name", 'avg({})'.format("current_price"), legend=True, ax=ax[3])
ax[3].set_ylabel("Avg Current Price per Brand")
ax[3].set_xticks(np.arange(pd_df.shape[0]), list(pd_df["brand_name"]), rotation=90, ha='right')

```

```

spark.stop()

```

```

bytes_image = io.BytesIO()
plt.savefig(bytes_image, format='png')
bytes_image.seek(0)

```

```

return send_file(bytes_image,
                  download_name='plot.png',
                  mimetype='image/png')

```

```

api.add_resource(Picture, "/dashboard")

```

```

public_url = ngrok.connect(name='flask').public_url
print(" * ngrok URL: " + public_url + " *")
app.run()

```

Results

Below, you'll find the outcome of the deployed pipeline, complete with accompanying screenshots explanations.

a) Data generation result

As depicted in the following screenshots, data is being regularly inserted into the "asofashion_2" collection in bulk every 10 seconds. The frequency and quantity of entries pushed during each iteration can be customized. Presently, the ingestion rate is set at 100 rows every 10 seconds.

```
[direct: mongos] shardDB> db.asofashion_2.count()  
0
```

```
[direct: mongos] shardDB> db.asofashion_2.count()  
300
```

```
[direct: mongos] shardDB> db.asofashion_2.count()  
1300
```

b) MongoDB Distributed Database result

The data generated by the data generation module and stored in the "asofashion_2" collection is distributed across multiple server nodes.

```
[direct: mongos] shardDB> db.asofashion_2.getShardDistribution()  
Shard shard2rs at shard2rs/abhi_1:50004,abhi_2:50005,abhi_3:50006  
{  
  data: '180KiB',  
  docs: 653,  
  chunks: 1,  
  'estimated data per chunk': '180KiB',  
  'estimated docs per chunk': 653  
}  
---  
Shard shard1rs at shard1rs/abhi_1:50001,abhi_2:50002,abhi_3:50003  
{  
  data: '178KiB',  
  docs: 647,  
  chunks: 1,  
  'estimated data per chunk': '178KiB',  
  'estimated docs per chunk': 647  
}  
---  
Totals  
{  
  data: '358KiB',  
  docs: 1300,  
  chunks: 2,  
  'Shard shard2rs': [  
    '50.24 % data',  
    '50.23 % docs in cluster',  
    '282B avg obj size on shard'  
  ],  
  'Shard shard1rs': [  
    '49.75 % data',  
    '49.76 % docs in cluster',  
    '282B avg obj size on shard'  
  ]  
}
```

c) Data transformation module result

Original vs Transformed Data

```
[direct: mongos] shardDB> db.asofashion_2.find({_id:ObjectId("6632767a1ce7b8e87d9e973e")})
[
  {
    _id: ObjectId('6632767a1ce7b8e87d9e973e'),
    product_id: 203326185,
    brand_name: 'Puma',
    title: 'Puma city graphic t-shirt in white',
    current_price: 26.06,
    previous_price: 42.64,
    colour: NaN,
    currency: 'USD',
    rrp: NaN,
    productCode: 120615658,
    productType: 'Product',
    gender: 'male'
  }
]
[direct: mongos] shardDB> db.asofashion_clean.find({_id:ObjectId("6632767a1ce7b8e87d9e973e")})
[
  {
    _id: ObjectId('6632767a1ce7b8e87d9e973e'),
    product_id: 203326185,
    brand_name: 'Puma',
    title: 'puma city graphic tshirt in white',
    current_price: 26.06,
    previous_price: 42.64,
    colour: NaN,
    currency: 'USD',
    rrp: NaN,
    productCode: 120615658,
    productType: 'Product',
    gender: 'male',
    missedWallet: []
  }
]
```

Spark Job execution to perform data cleaning:

▼ Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
----------------	------	-------	---------------------	------------------------	----------------	------	-------	----------

▼ Completed Applications (9)

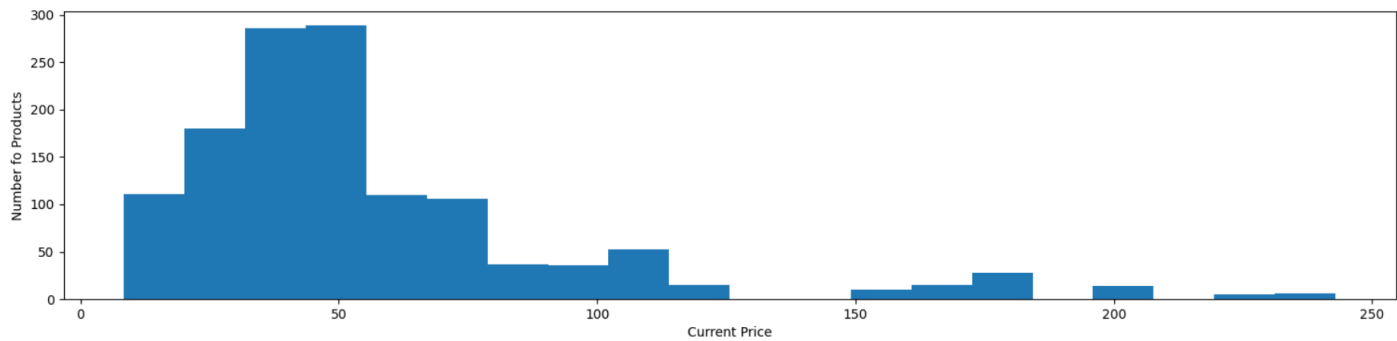
Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20240501170923-0008	mongodbtst1	32	1024.0 MiB		2024/05/01 17:09:23	exouser	FINISHED	7 s
app-20240501170850-0007	mongodbtst1	32	1024.0 MiB		2024/05/01 17:08:50	exouser	FINISHED	3 s
app-20240501170810-0006	mongodbtst1	32	1024.0 MiB		2024/05/01 17:08:10	exouser	FINISHED	3 s
app-20240501170729-0005	mongodbtst1	32	1024.0 MiB		2024/05/01 17:07:29	exouser	FINISHED	3 s
app-20240501170648-0004	mongodbtst1	32	1024.0 MiB		2024/05/01 17:06:48	exouser	FINISHED	3 s
app-20240501170615-0003	mongodbtst1	32	1024.0 MiB		2024/05/01 17:06:15	exouser	FINISHED	8 s
app-20240501170607-0002	mongodbtst1	32	1024.0 MiB		2024/05/01 17:06:07	exouser	FINISHED	3 s
app-20240501170527-0001	mongodbtst1	32	1024.0 MiB		2024/05/01 17:05:27	exouser	FINISHED	3 s
app-20240501170445-0000	mongodbtst1	32	1024.0 MiB		2024/05/01 17:04:45	exouser	FINISHED	3 s

d) Exploratory Data Analysis module result

The visualizations are accessible through a Flask API published at <https://large-bee-oddly.ngrok-free.app/dashboard>.

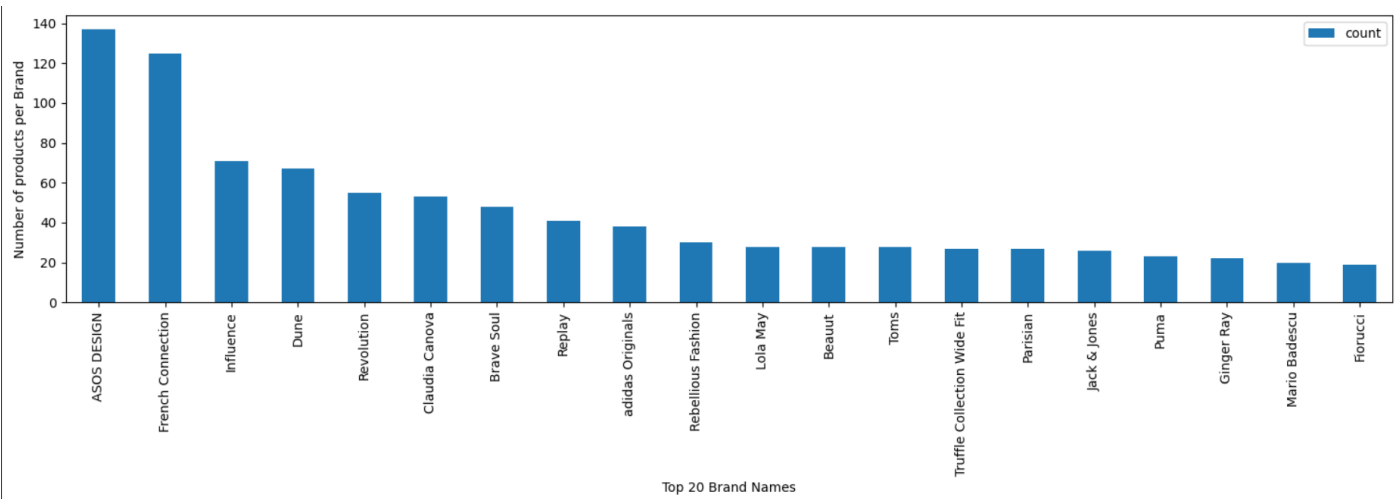
The free domain name offered by ngrok is used to make them accessible on internet globally.

Histogram: current_price



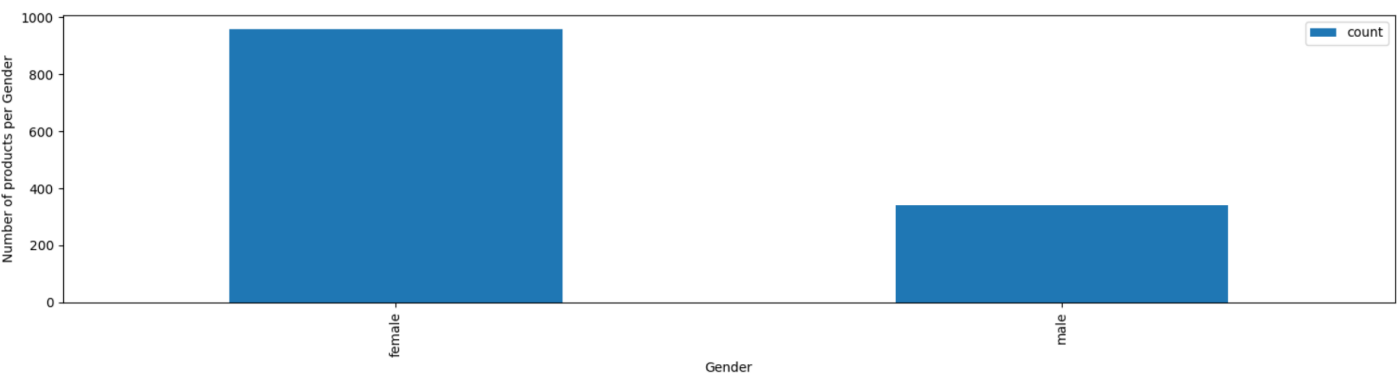
Analyzing the histogram distribution of the number of products across different price ranges provides a comprehensive overview of the product assortment's pricing dynamics within the fashion e-commerce domain. By segmenting the price range into bins and visualizing the count of products within each bin, this analysis offers insights into the breadth and depth of available offerings at various price points

Barplot: Top 20 Brand_names vs No of products



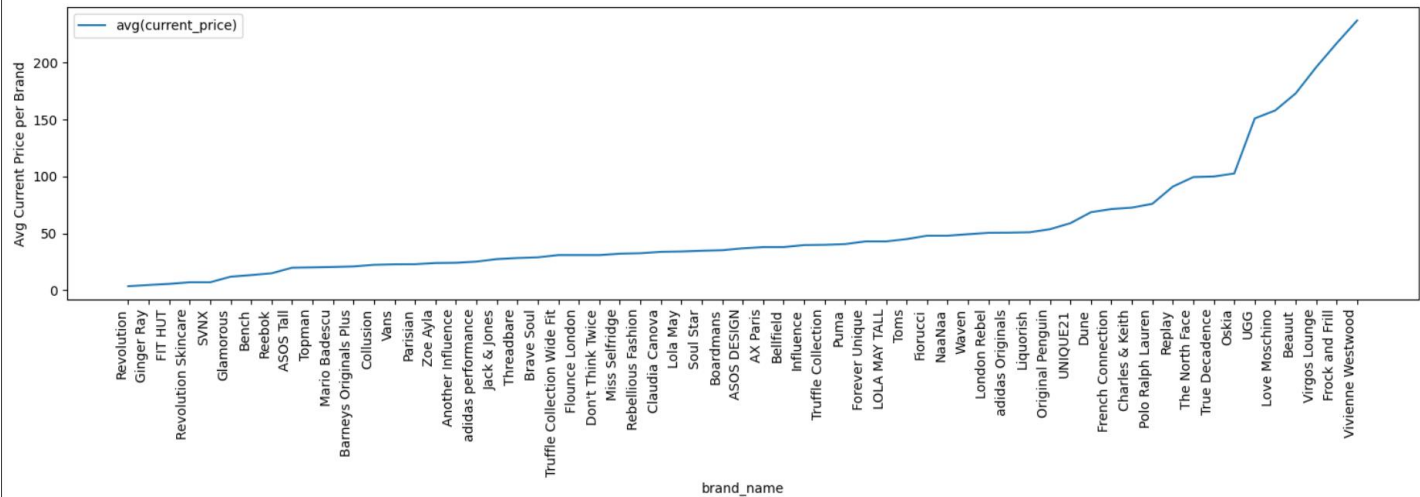
Each bar on the plot represents a distinct brand, with the height of the bar corresponding to the number of products associated with that brand. As such, brands with taller bars signify a larger presence within the dataset, indicating a broader range of offerings and potentially greater market share. Conversely, brands with shorter bars may suggest a more niche presence or a focused product line.

Barplot: Gender vs No of products



By examining the distribution of products across gender categories in real-time, stakeholders gain valuable insights into shifting consumer preferences and emerging market trends. This enables agile decision-making, allowing businesses to tailor their product assortments, marketing strategies, and promotional campaigns to better cater to the needs and desires of their target audience

Lineplot: Brand names vs Avg current price per brand



The line chart depicting brand names versus average price per brand name provides valuable insights into pricing strategies and brand positioning within the fashion e-commerce domain.

Each data point on the line represents a brand, with the x-axis denoting the brand names and the y-axis indicating the average price associated with each brand. The upward or downward trend of the line illustrates the relative pricing levels across different brands.

Analyzing the line chart allows for the identification of brands positioned at various price points within the market. Brands with higher average prices tend to occupy the upper end of the chart, reflecting premium or luxury offerings, while brands with lower average prices are situated towards the bottom, indicating more affordable options. The slope of the line, whether steep or gradual, provides insights into the pricing dynamics and competitiveness of each brand within the market.

Discussion

The results obtained from distributed processing on a Spark cluster and distributed storage on a MongoDB cluster in a data pipeline implementation demonstrate significant improvements in **scalability, performance, reliability and flexibility**, enabling effective management of large-scale data sets in today's data-driven environment

Scalability: Distributed processing on a Spark cluster enables the system to handle large-scale data processing tasks efficiently by distributing the workload across multiple nodes. Similarly, distributed storage on a MongoDB cluster allows for the storage and retrieval of massive datasets across a distributed architecture.

Performance: By harnessing the parallel processing capabilities of a Spark cluster, the data pipeline can execute complex data transformations and analytics tasks in parallel, resulting in significantly reduced processing times compared to traditional single-node processing. Additionally, distributed storage on a MongoDB cluster ensures fast and efficient data access, enabling quick retrieval and manipulation of data regardless of its size.

Reliability: Distributed processing on a Spark cluster and distributed storage on a MongoDB cluster enhance system reliability and fault tolerance. In the event of node failures or network disruptions, both Spark and MongoDB

automatically handle data replication and failover, ensuring continuous operation and data availability without compromising system integrity.

Flexibility: The combination of distributed processing and storage offers flexibility in handling diverse data types and processing requirements. Spark's versatile processing framework supports various data processing tasks, including batch processing, streaming analytics, machine learning, and graph processing. MongoDB's flexible document model accommodates a wide range of data structures and formats, enabling seamless storage and retrieval of structured, semi-structured, and unstructured data.

Technologies and Skills utilized:

The modules on **Distributed Computing and File System, Lifecycle & Pipelines, Ingest & Storage and Processing and Analytics** were extremely helpful and formed the basis of this project. Modules Distributed computing and File system along Processing and Analytics introduced me to Spark implementations which I have extensively used in this project. The module on data lifecycles provided me with a framework on which I could build my project. The Ingest and Storage module introduced me to NoSQL which is the backbone of this project. All in all, the concepts and tools introduced in this course were extremely informative and helped me understand how I could build a project that, if modified and scaled, could have real applications.

Challenges Faced:

From a developer's perspective, implementing a data pipeline with distributed storage and distributed data processing entails challenges such as version compatibility issues, complex installations, and configurations, network barriers between cluster instances, steep learning curves, limited documentation, debugging challenges, and resource management intricacies. Overcoming these hurdles demands meticulous planning, continuous learning, and expertise in distributed systems and big data technologies.

Conclusion

In conclusion, the implementation of a data pipeline equipped with a distributed database and distributed processing capabilities offers numerous benefits for handling big data with ease in various industries, including fashion e-commerce.

The combination of distributed database and processing technologies enhances overall system performance and reliability when dealing with massive datasets. By distributing data and processing tasks across multiple nodes, the system can handle high volumes of concurrent requests and maintain consistent performance levels, ensuring uninterrupted operations and superior user experiences.

Additionally, the scalability of distributed systems allows organizations to scale their infrastructure horizontally as data volumes and processing requirements grow, minimizing infrastructure costs and maximizing resource utilization.

References

<https://www.mongodb.com/products/integrations/spark-connector>

<https://www.mongodb.com/docs/spark-connector/master/batch-mode/batch-read/>

<https://www.mongodb.com/docs/manual/sharding/>

<https://spark.apache.org/docs/3.2.4/>

https://spark.apache.org/docs/3.2.4/api/python/getting_started/index.html

<https://ostecnix.com/a-beginners-guide-to-cron-jobs/>

<https://pymongo.readthedocs.io/en/stable/>