The goal of the project

In our project, A continuous data flow was provided via Apache Kafka with historical and current data received from financial markets. Market trends were analyzed using Apache Spark and predictions were made for future movements. MongoDB was used to store market data and forecast results.

Method and Technologies Used

The project was developed using the Python programming language. Python's powerful libraries and wide community support met various data processing and web development needs in this project. The main technologies used are:

- **Apache Kafka** It is an open source distributed messaging system used to process and manage high volumes of data streams in real time. Thanks to its high scalability and durability features, it is preferred in various usage scenarios such as data integration, event streaming and real-time analysis. We received stock market data in the project. api'den communication and real-time analysis of incoming data with other services via Kafka used.

- **MongoDB** It was used as a database, where financial data was kept in an organized form, and was positioned between Kafka and Spark to create a data warehouse for real-time analysis. It was also used to store the analysis result of the data in dataframe format returned as a result of the analysis. Additionally, realtime database was monitored with Mongo watch collection watch.

- **Apache Spark** It is a fast and general-purpose open source distributed computing engine developed for big data processing. It was used in the project to analyze real-time financial data and performs market trend analysis and price prediction.

- **Confluent** Confluent offers a variety of tools, services and extensions to make using Kafka easier and more effective. Confluent's The features it provides include advanced management and monitoring tools, security features, data integration tools and cloud-based solutions. This makes it possible to manage and analyze large-scale data flows more efficiently.

Application Steps

1. STEP ONE

- **Connecting to API and Pulling Data**

```
conn = http.client.HTTPSConnection("api.collectapi.com")

headers = {
    'content-type': "application/json",
    'authorization': "apikey 0AHKzfMWRiI5Oat0f07Ky5:0TlKtB0ADuKnD2qhrg5vIN"
    }

conn.request("GET", "/economy/currencyToAll?int=10&base=USD", headers=headers)


res = conn.getresponse()
status = res.status
data = res.read()

print("HTTP Status:", status)
print("Raw data:", data)
```

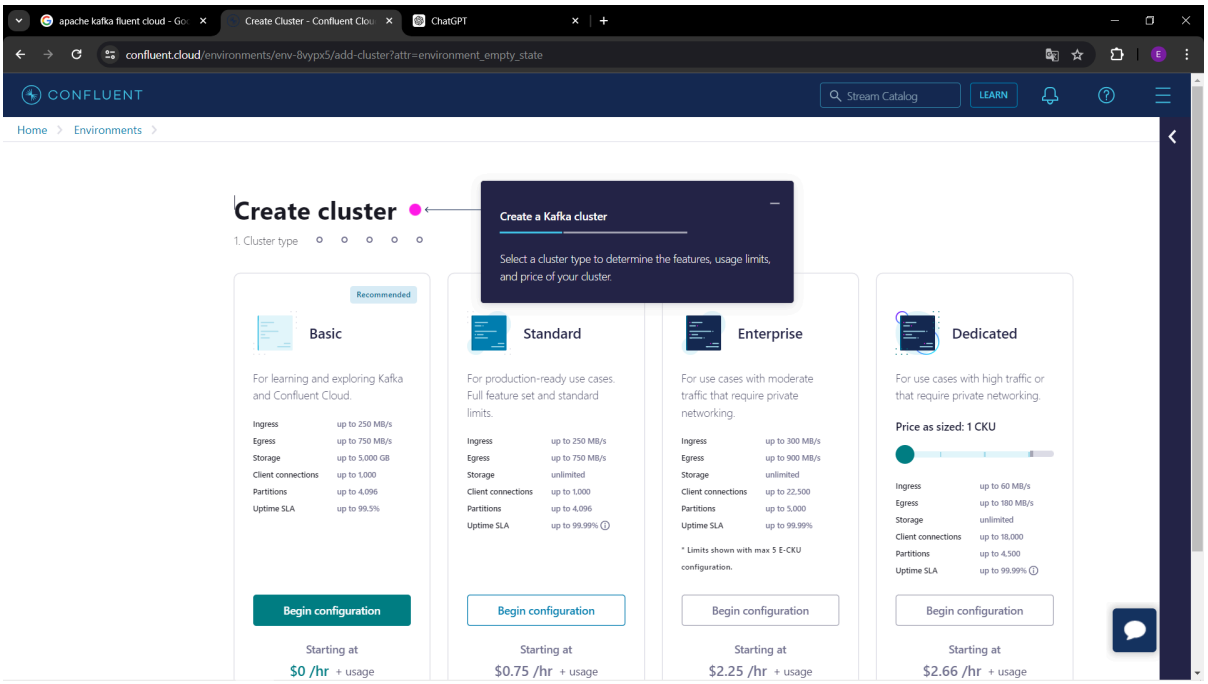
  - This code is used to retrieve data about exchange rates from the Collect API. As a result of the GET request made to the API, the HTTP status code and data of the response received from the server are printed. This data is in JSON format and contains information about different exchange rates as a result of the request. This piece of code can be used to get exchange rates in real time, especially in financial applications.

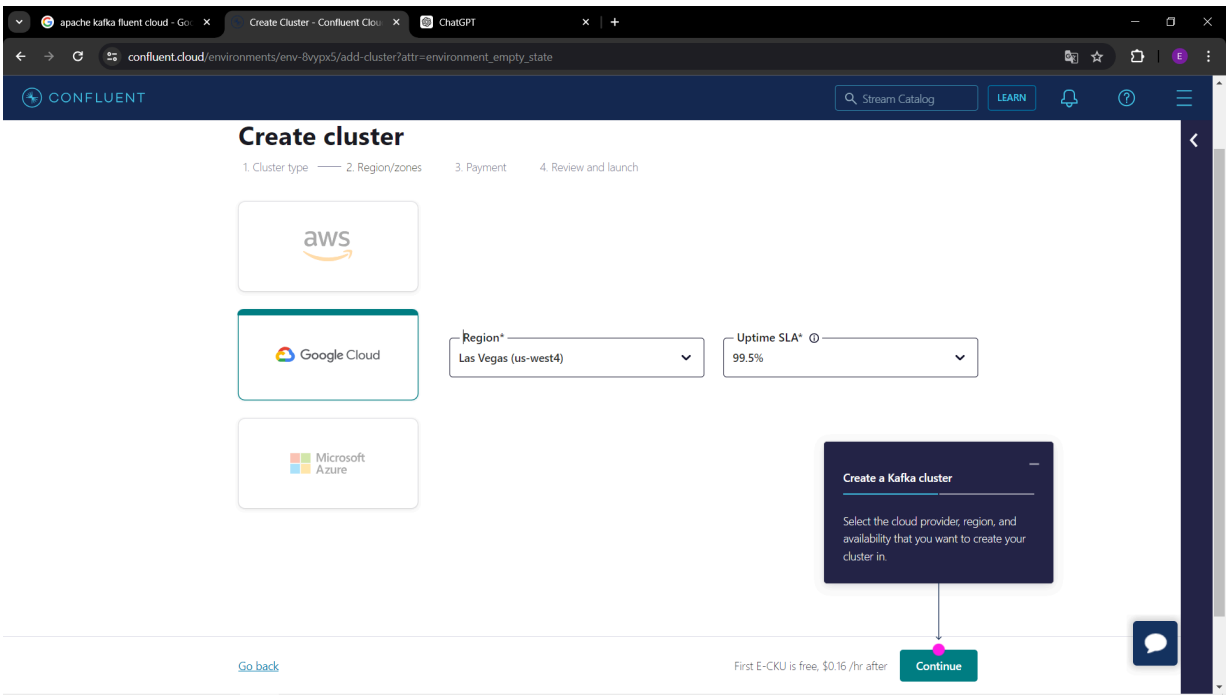
- **Converting API Data to Json Format**

```
try:
    json_data = json.loads(data.decode("utf-8"))
    print("JSON data:", json_data)
except json.JSONDecodeError as e:
    print("JSON decode error:", e)
    json_data = None
```

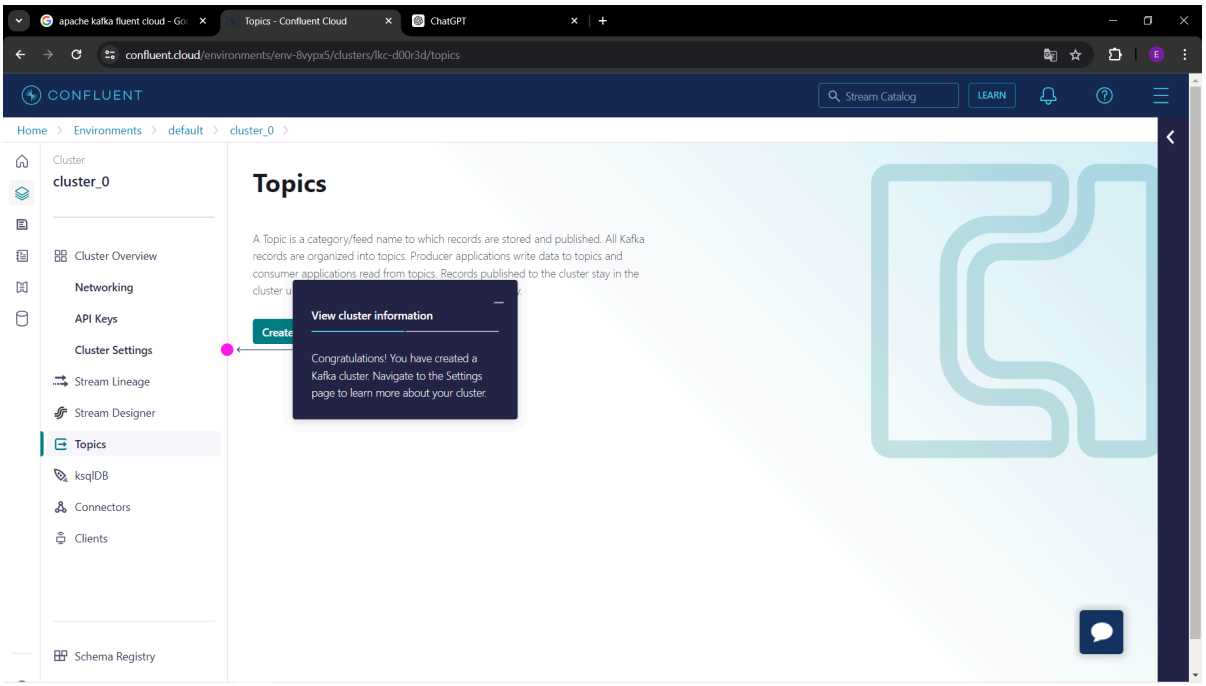
- This code aims to convert the raw data received from the server in the previous steps into JSON format and catch any possible errors that may occur in the process. In case of successful conversion, JSON data is printed. If an error occurs, the error message is printed and the JSON data is set to None. This type of error checking is especially critical to verify whether the data received from the server is in the expected format.

- **Creating a Confluent Cloud Cluster**

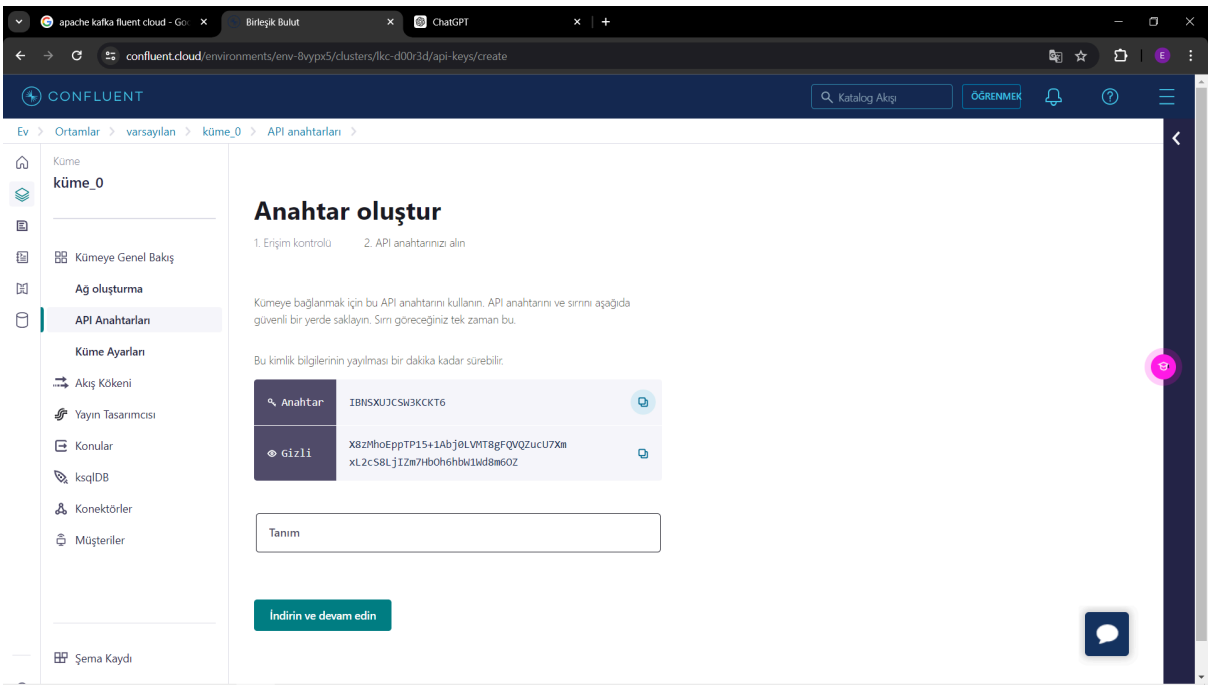




- **Creating Confluent Cloud Topics**



- **Creating a Confluent Cloud Topic Key**



- **Topic connecting**

```
conf = {
    'bootstrap.servers': 'pkc-6ojv2.us-west4.gcp.confluent.cloud:9092',
    'sasl.mechanism': 'PLAIN',
    'security.protocol': 'SASL_SSL',
    'sasl.username': 'IBNSXUJCSW3KCKT6',
    'sasl.password': 'X8zMhoEppTP15+1Abj0LVMT8gFQVQZucU7XmxL2cS8LjIZm7HbOh6hbW1Wd8m6OZ'
}
```

  - This configuration dictionary defines the settings required for an Apache Kafka client to securely connect to a Kafka cluster on Confluent Cloud. These settings include server address, authentication mechanism, security protocol, username, and password. This type of configuration is critical to ensuring secure and authenticated communication with Kafka.

- **Creating Colab Producer**

```
# Producer oluşturma
producer = Producer(**conf)
```

- This code creates a Producer object that is used to send messages to Kafka. The Producer object is configured using the configuration settings specified in the conf dictionary. These settings are required for the Kafka producer to connect to the server and communicate securely. The **conf syntax unpacks the contents of the dictionary and passes each key-value pair as an argument to the constructor of the Producer class. This way, the Kafka producer is configured correctly.

- **Kafka Api'den Sending Data**

```
# Kafka'ya gönderme
producer.produce('topic_0', key='currency_data', value=json.dumps(json_data), callback=delivery_report)
producer.flush()
```

- This code generates and sends messages on a Kafka topic. The content of the message is data in JSON format and is sent with a specific key. The produce method ensures that the message is sent, and the delivery status of the message is checked with the callback function. The flush method completes the sending of all messages in the producer queue and clears the queue. This is important to ensure that all messages reach Kafka before the application shuts down.

2. STEP TWO

- **Creating a Consumer**

```
# Consumer oluşturma
conf.update({
    'group.id': 'my_group',
    'auto.offset.reset': 'earliest'
})


consumer = Consumer(**conf)
consumer.subscribe(['topic_0'])
```

- This code adds new settings by updating the Kafka consumer configuration settings. The group.id setting determines the group the consumer belongs to, and consumers within this group share and consume messages. The auto.offset.reset setting allows the consumer

to consume messages starting from the oldest message when an existing offset cannot be found. These settings control and optimize the message consumption behavior of the Kafka consumer.

- **Creating a MongoDb Connection**

```
# MongoDB bağlantısı
mongodb_host = "mongodb+srv://mrfrkkan234:fvadYMhdJbqHoYvT@cluster0.qwbicey.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"
mongo_client = MongoClient(mongodb_host)
db = mongo_client['databse']  # Veritabanı adı
collection = db['jsonVeri']  # Koleksiyon adı
```

  - This code is used to connect to the MongoDB database and perform operations on a particular database and collection. Step by step explanation of the code:
  - MongoDB Connection String Definition: The mongodb_host variable contains the connection string required to connect to the MongoDB server.
  - Creating a MongoDB Client: Using the MongoClient class, you connect to the MongoDB server with the mongodb_host connection string and create a client object named mongo_client.
  - Database Selection: The database named databse (the properly written name should be database) is selected through the mongo_client object.
  - Collection Selection: The collection named jsonData is accessed through the db database object.

With these settings, you can perform operations such as adding, updating, deleting and querying data with a specific database and collection on MongoDB.

- **Saving the Message from Kafka to MongoDb**

```
# Kafka'dan gelen mesajı MongoDB'ye kaydetme
json_data = msg.value().decode('utf-8')
document = json.loads(json_data)
collection.insert_one(document)
print('Message saved to MongoDB:', document)
```

  - This code takes a message from Kafka and saves it to MongoDB. Step by step explanation:

- Decoding the Message Coming from Kafka: Using msg.value().decode('utf-8'), the byte array value of the message coming from Kafka is converted into a string in UTF-8 format.
- Converting JSON String to Dictionary: Using json.loads(json_data), the string in JSON format is converted to Python dictionary.
- Saving Data to MongoDB: JSON data is added to the MongoDB collection as a document with the collection.insert_one(document) method.
- Printing the Success Message: With print('Message saved to MongoDB:', document), it is printed to the console that the message has been successfully saved to MongoDB.

This snippet is a basic example of processing data from Kafka and saving that data to MongoDB. This is a common way to use Kafka and MongoDB together to manage and store the flow of data.

3. STEP THREE

Apache in MongoDB in previous steps Kafka'and save the data we receive we arranged and a data warehouse for analysis we had created. In this step in Mongo We will retrieve data from the data warehouse in real time and make it ready for analysis. We will also set up a realtime monitoring system on MongoDB.

- **in Mongo pulling data from data warehouse**

  - First, we need to get the data to analyze it from the data warehouse we created by saving it to Mongo. For this process, we first need to make mongo connections.

```
# MongoDB bağlantı bilgileri
mongodb_host = "mongodb+srv://mrfrkkan234:[redacted]@cluster0.qwbicey.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"

# MongoDB Atlas bağlantısı (sizin veritabanınıza göre değiştirin)
client = MongoClient(mongo_uri)
db = client["databse"]
collection = db["jsonVeri"]
```

  - First, we define the url given by Mongo Atlas and we make the connection through this url. Then, we define our database and collections fields. In MongoDB, collections correspond to tables in relational databases.

- **Apache Spark Login**
  - In order to analyze with Spark, we need to create a spark session.

```
# Spark oturumunu başlat
spark = SparkSession.builder \
    .appName("MongoDBtoSpark") \
    .getOrCreate()
```

- **Creating a Mongo Listener**
    - This process was done to instantly process every new data in MongoDB. In other words, for every instant data coming to Kafka, Mango listens and prepares for the analysis process by pulling the data from the warehouse.

```
# MongoDB Change Stream dinleyici
change_stream = collection.watch()

print("Dinlemeye başlandı...")
for change in change_stream:
    process_change(change)
```

4. STEP FOUR

- **Analyzing Data**
    - First, we wrote the code to group the data by currency code and collect the rate values.

```
# Döviz koduna göre gruplama ve rate değerlerini toplama
codes = df.select("code").distinct().collect()
columns = ["code", "name"]
data = []

for code in codes:
    code_value = code["code"]
    df_filtered = df.filter(df.code == code_value).orderBy("rate")
    rates = df_filtered.select("rate").collect()
    row = [code_value, df_filtered.select("name").first()["name"]]
    for rate in rates:
        row.append(rate["rate"])
    data.append(row)
```

- Then, we created a column and a new dataframe to display the latest rate value.

```
# Yeni sütun adları oluştur
for i in range(len(data[0]) - 2):
    columns.append(f"rate_{i+1}")

# Yeni DataFrame'i oluştur
new_pdf = pd.DataFrame(data, columns=columns)
new_df = spark.createDataFrame(new_pdf)
```

- Now we calculated the average rate value of the data and added it to the required column.

```
# Ortalama rate değerini hesapla ve ekle
rate_columns = [col(c) for c in new_df.columns if c.startswith("rate_")]
new_df = new_df.withColumn("average_rate", sum(rate_columns) / len(rate_columns))
```

- Now it's time to compare the last rate value from the database with the average we created, and if the last rate value is higher, it is higher than the average; If less, we wrote the analysis as below average in the "comparison" column.

```
# Son gelen rate değerini ortalama ile karşılaştır ve yeni sütunu ekle
last_rate_col = new_df.columns[-2]  # En son eklenen rate sütunu
new_df = new_df.withColumn(
    "comparison",
    when(col(last_rate_col) > col("average_rate"), "ortalamanın üstünde").otherwise("ortalamanın altında")
)
```

- Finally, we wrote the code that lists the columns we determined.

```
# Sadece gerekli sütunları seç ve göster
result_df = new_df.select("code", "name", last_rate_col, "average_rate", "comparison")
result_df.show(n=result_df.count(), truncate=False)
```

- Received financial forecast analysis of dataframe For example, a particular row is in the form of a table like this.

code	name	rate_15	average_rate	comparison
BRL	Brazilian Real	5.17257	5.165762333333333	ortalamanın üstünde
CZK	Czech Koruna	22.713363	22.707732066666665	ortalamanın üstünde
VES	Venezuelan Bolivar	3648341.999044	3644716.2442305335	ortalamanın üstünde
MUR	Mauritian Rupee	46.051333	45.97267199999999	ortalamanın üstünde
COP	Colombian Peso	3876.829869	3873.895543666666	ortalamanın üstünde
IDR	Indonesian Rupiah	16035.018937	16008.680704999999	ortalamanın üstünde
IRR	Iranian Rial	42002.634246	41972.831170599995	ortalamanın üstünde
ARS	Argentine Peso	890.503156	890.3432140666666	ortalamanın üstünde
TRY	Turkish Lira	32.246923	32.17979433333333	ortalamanın üstünde
CAD	Canadian Dollar	1.368077	1.3667543333333332	ortalamanın üstünde
AED	Emirati Dirham	3.6725	3.6725	ortalamanın altında
CNY	Chinese Yuan Renminbi	7.242648	7.242613733333335	ortalamanın üstünde
SAR	Saudi Arabian Riyal	3.75	3.75	ortalamanın altında
EUR	Euro	0.922194	0.9219627999999999	ortalamanın üstünde
TWD	Taiwan New Dollar	32.278428	32.269895600000005	ortalamanın üstünde
LKR	Sri Lankan Rupee	299.339094	299.33691086666664	ortalamanın üstünde
TTD	Trinidadian Dollar	6.793559	6.791175	ortalamanın üstünde
THB	Thai Baht	36.670482	36.65579486666666	ortalamanın üstünde
ZAR	South African Rand	18.441761	18.427734466666664	ortalamanın üstünde
PKR	Pakistani Rupee	278.515386	278.2020552000001	ortalamanın üstünde
BHD	Bahraini Dinar	0.376	0.37600000000000006	ortalamanın altında
ISK	Icelandic Krona	138.276132	138.25383933333333	ortalamanın üstünde
ILS	Israeli Shekel	3.664125	3.661720466666667	ortalamanın üstünde
BWP	Botswana Pula	13.571798	13.525372733333333	ortalamanın üstünde
NOK	Norwegian Krone	10.602922	10.592629000000004	ortalamanın üstünde
QAR	Qatari Riyal	3.64	3.64	ortalamanın altında
AUD	Australian Dollar	1.511004	1.508969933333333	ortalamanın üstünde

5. STEP FİVE

- **Spark DataFrame'i Pandas DataFrame'e Conversion**

```
result_pdf = result_df.toPandas()
```

result_pdf: This is Spark between DataFrame. Spark DataFrames, big data with clusters Apache, a distributed data processing framework, designed to run Spark's is a part of it.

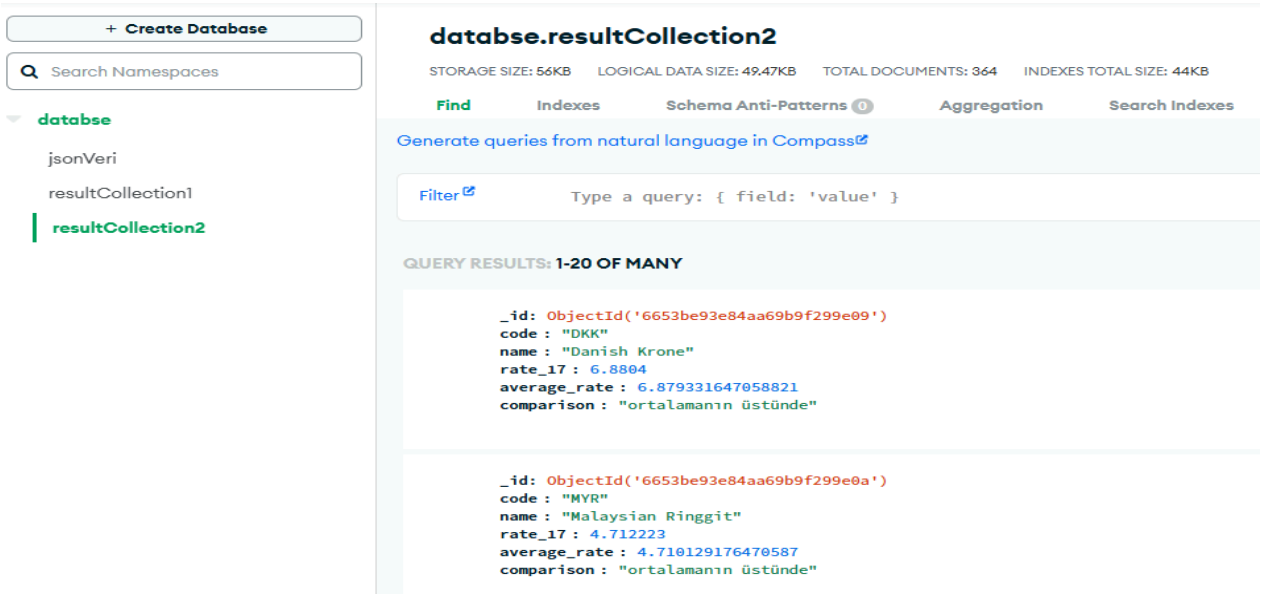
toPanda(): This method is a Spark This DataFrame Pandas DataFrame'ine transforms. Panda, Python'da It is a widely used library for data processing and analysis. `toPanda()` method, Spark DataFrame'in A Pandas file that takes all its data and stores it in a single machine memory DataFrame'i creates.

  - Pandas offers more flexible and easy-to-use data manipulation tools for small and medium-sized datasets. That's why Spark DataFrame'i Pandas DataFrame'e we transformed.

- **Pandas DataFrame'i MongoDB's save**

```
result_collection = db.resultCollection2  # Yeni koleksiyon adı
result_collection.insert_many(result_pdf.to_dict("records"))
```

db: This represents the MongoDB database connection.



`result_collection = db.resultCollection2`: This line, MongoDB in database `resultCollection2` Creates a new collection called . Collections, MongoDB'd They are structures used to store data.

`insert_many(result_pdf.to_dict("records"))`: This method saves the Pandas DataFrame to the MongoDB collection.

`result_pdf.to_dict("records")`: Pandas DataFrame'i Converts Python dictionaries in the form of a list. Each line is represented as a dictionary.

`insert_many`: This pymongo method adds multiple documents to a MongoDB collection using the provided list.

- **Checking MongoDB Collection**

```
for doc in result_collection.find():
    print(doc)
```

`find()`: MongoDB'd `find()` The method returns all documents in the collection. This method returns the entire contents of the collection when called without a specific query.

`print(doc)`: Prints each document. This is used to verify data successfully saved to the MongoDB collection. Each document is printed in Python dictionary format.

- **Signing Out of Spark**

```
spark.stop()
```

`spark.stop()`: This method terminates the current Spark session. Closing a Spark session releases all resources associated with the Spark application.
All data processing and analysis must be completed before closing the Spark session. Log out of Spark, Spark'ta It means the termination of all transactions made.