assignment2

October 23, 2025

Link to Streamlit application: https://ind320-tereseivesdal.streamlit.app/

Link to Github repository: https://github.com/teresemyhre/IND320-tereseivesdal

1 AI Usage

AI tools were used throughout the assignment to improve efficiency and code quality. GitHub Copilot was used for code completion and smaller syntax suggestions while working in VS Code, mainly to speed up repetitive coding tasks and fill in simple functions. ChatGPT was used for explanations, debugging, and guidance on how to connect the different components, including Cassandra, Spark, and MongoDB. It was also used to clarify error messages, improve code readability, and help with documentation and plotting adjustments in Streamlit.

[115]:
```python
# import required libraries
from pymongo.mongo_client import MongoClient
from pymongo.server_api import ServerApi
import requests
import calendar
import pandas as pd
from datetime import datetime
import toml
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from cassandra.cluster import Cluster
import plotly.express as px
import plotly.io as pio
pio.defaults.default_format = "png"
```

[107]:
```python
# Load secrets
secrets = toml.load(".streamlit/secrets.toml")
uri = secrets["MONGO"]["uri"]

# Create a new client and connect to the server
client = MongoClient(uri, server_api=ServerApi('1'))
# Send a ping to confirm a successful connection
try:
    client.admin.command('ping')
    print("Pinged your deployment. You successfully connected to MongoDB!")
```

```
except Exception as e:
    print(e)
```

```
Pinged your deployment. You successfully connected to MongoDB!
```

[3]:
```
# Manipulate data from Python
db = client.test_database
collection = db.test_collection
collection.insert_one({"name": "test", "value": 42})
result = collection.find_one({"name": "test"})
print("Retrieved document:", result)
```

```
Retrieved document: {'_id': ObjectId('68f9359030af0db0206c1daa'), 'name':
'test', 'value': 42}
```

This code successfully connected to MongoDB, inserted a test document, and retrieved it to confirm that data manipulation from Python works correctly.

2 Retrieving Data from the Elhub API

The Elhub API provides hourly production data per price area and production group.
Since each API request can only cover one month, I loop through all months of 2021 to collect the complete dataset.

[4]:
```
# Basic configuration
BASE_URL = "https://api.elhub.no/energy-data/v0/price-areas"  # Elhub endpoint␣
 ↪for production data
DATASET  = "PRODUCTION_PER_GROUP_MBA_HOUR"                    # Dataset name as␣
 ↪defined in the API docs
YEAR     = 2021                                               # Year to collect␣
 ↪data for

# Helper function: generate monthly start/end timestamps
def month_ranges(year):
    """
    Generator function that yields start and end datetimes for each month in a␣
 ↪given year.
    Each month is formatted as a full timestamp with hours, minutes, and␣
 ↪seconds.
    """
    for m in range(1, 13):
        start = datetime(year, m, 1, 0, 0, 0)                 # First day of␣
 ↪month, 00:00:00
        end_day = calendar.monthrange(year, m)[1]             # Last day of the␣
 ↪month (28-31)
        end = datetime(year, m, end_day, 23, 59, 59)          # End of last day,␣
 ↪23:59:59
```

```
        yield start, end, m                                    # Return start,␣
 ↪end, and month number

# Initialize lists for results and failed requests
all_rows = []    # To store all production records across months
failures = []    # To log months that fail (e.g., API 400 or timeout)

# Loop through each month and retrieve data
for start, end, m in month_ranges(YEAR):
    # Manually format timestamps in the correct ISO format with encoded␣
 ↪timezone (+01:00)
    start_str = start.strftime("%Y-%m-%dT%H:%M:%S") + "%2B01:00"
    end_str   = end.strftime("%Y-%m-%dT%H:%M:%S") + "%2B01:00"

    # Build the full URL manually to avoid 'requests' double-encoding the␣
 ↪timezone
    url = f"{BASE_URL}?
 ↪dataset={DATASET}&startDate={start_str}&endDate={end_str}"

    # Make the GET request to Elhub
    r = requests.get(url, timeout=60)

    # If the response was successful (status code 200)
    if r.ok:
        j = r.json()  # Parse the JSON response

        # Extract the nested list in "productionPerGroupMbaHour" for each price␣
 ↪area
        month_rows = [
            rec
            for item in j.get("data", [])                          # Loop through␣
 ↪all price areas
            for rec in item.get("attributes", {}).
 ↪get("productionPerGroupMbaHour", [])
        ]

        # Add all the records for this month to our full list
        all_rows.extend(month_rows)

        # Print a short success message for tracking
        print(f"{YEAR}-{m:02d}: {len(month_rows)} rows added")

    # If the request failed (400, 500, etc.)
    else:
        failures.append((m, r.status_code))  # Record month and error code
        print(f"{YEAR}-{m:02d}: HTTP {r.status_code}")
```

```python
# Print summary of all months
print(f"\nTotal rows: {len(all_rows)} | Failed months: {len(failures)}")

# Convert all results into a Pandas DataFrame
df_production = pd.DataFrame(all_rows)

if not df_production.empty:
    # Standardize column names to lowercase for consistency
    df_production.columns = [c.lower() for c in df_production.columns]

    # Display the first few rows of the dataset
    display(df_production.head())
else:
    print("No data returned.")
```

```
2021-01: 17856 rows added
2021-02: 16128 rows added
2021-03: 17832 rows added
2021-04: 17280 rows added
2021-05: 17856 rows added
2021-06: 17976 rows added
2021-07: 18600 rows added
2021-08: 18600 rows added
2021-09: 18000 rows added
2021-10: 18625 rows added
2021-11: 18000 rows added
2021-12: 18600 rows added

Total rows: 215353 | Failed months: 0
```

	endtime	lastupdatedtime	pricearea	productiongroup	quantitykwh	starttime
0	2021-01-01T01:00:00+01:00	2024-12-20T10:35:40+01:00	NO1	hydro	2507716.8	2021-01-01T00:00:00+01:00
1	2021-01-01T02:00:00+01:00	2024-12-20T10:35:40+01:00	NO1	hydro	2494728.0	2021-01-01T01:00:00+01:00
2	2021-01-01T03:00:00+01:00	2024-12-20T10:35:40+01:00	NO1	hydro	2486777.5	2021-01-01T02:00:00+01:00
3	2021-01-01T04:00:00+01:00	2024-12-20T10:35:40+01:00	NO1	hydro	2461176.0	2021-01-01T03:00:00+01:00
4	2021-01-01T05:00:00+01:00	2024-12-20T10:35:40+01:00	NO1	hydro	2466969.2	2021-01-01T04:00:00+01:00

3 Start Spark session with Cassandra connector

[ ]:
```
# Create or get Spark session with Cassandra connector enabled
spark = (
    SparkSession.builder
    .appName("ElhubToCassandra")
    .config("spark.jars.packages", "com.datastax.spark:
 ↪spark-cassandra-connector_2.12:3.5.1")
    .config("spark.cassandra.connection.host", "127.0.0.1")  # Cassandra in␣
 ↪Docker
    .getOrCreate()
)

print("Spark session connected to Cassandra.")
```

```
25/10/23 13:38:59 WARN Utils: Your hostname, Tereses-MacBook-Air.local resolves
to a loopback address: 127.0.0.1; using 10.58.80.209 instead (on interface en0)
25/10/23 13:39:00 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
Ivy Default Cache set to: /Users/teresemyhre/.ivy2/cache
The jars for the packages stored in: /Users/teresemyhre/.ivy2/jars
com.datastax.spark#spark-cassandra-connector_2.12 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-
parent-7db634d3-bf23-416d-ab07-a307527f204b;1.0
        confs: [default]

:: loading settings :: url =
jar:file:/Users/teresemyhre/opt/anaconda3/envs/D2D_env/lib/python3.12/site-packa
ges/pyspark/jars/ivy-2.5.1.jar!/org/apache/ivy/core/settings/ivysettings.xml

        found com.datastax.spark#spark-cassandra-connector_2.12;3.5.1 in central
        found com.datastax.spark#spark-cassandra-connector-driver_2.12;3.5.1 in
central
        found org.scala-lang.modules#scala-collection-compat_2.12;2.11.0 in
central
        found org.apache.cassandra#java-driver-core-shaded;4.18.1 in central
        found com.datastax.oss#native-protocol;1.5.1 in central
        found com.datastax.oss#java-driver-shaded-guava;25.1-jre-graal-sub-1 in
central
        found com.typesafe#config;1.4.1 in central
        found org.slf4j#slf4j-api;1.7.26 in central
        found io.dropwizard.metrics#metrics-core;4.1.18 in central
        found org.hdrhistogram#HdrHistogram;2.1.12 in central
        found org.reactivestreams#reactive-streams;1.0.3 in central
        found org.apache.cassandra#java-driver-mapper-runtime;4.18.1 in central
        found org.apache.cassandra#java-driver-query-builder;4.18.1 in central
        found org.apache.commons#commons-lang3;3.10 in central
        found com.thoughtworks.paranamer#paranamer;2.8 in central
        found org.scala-lang#scala-reflect;2.12.19 in central
```

```
:: resolution report :: resolve 229ms :: artifacts dl 7ms
        :: modules in use:
        com.datastax.oss#java-driver-shaded-guava;25.1-jre-graal-sub-1 from
central in [default]
        com.datastax.oss#native-protocol;1.5.1 from central in [default]
        com.datastax.spark#spark-cassandra-connector-driver_2.12;3.5.1 from
central in [default]
        com.datastax.spark#spark-cassandra-connector_2.12;3.5.1 from central in
[default]
        com.thoughtworks.paranamer#paranamer;2.8 from central in [default]
        com.typesafe#config;1.4.1 from central in [default]
        io.dropwizard.metrics#metrics-core;4.1.18 from central in [default]
        org.apache.cassandra#java-driver-core-shaded;4.18.1 from central in
[default]
        org.apache.cassandra#java-driver-mapper-runtime;4.18.1 from central in
[default]
        org.apache.cassandra#java-driver-query-builder;4.18.1 from central in
[default]
        org.apache.commons#commons-lang3;3.10 from central in [default]
        org.hdrhistogram#HdrHistogram;2.1.12 from central in [default]
        org.reactivestreams#reactive-streams;1.0.3 from central in [default]
        org.scala-lang#scala-reflect;2.12.19 from central in [default]
        org.scala-lang.modules#scala-collection-compat_2.12;2.11.0 from central
in [default]
        org.slf4j#slf4j-api;1.7.26 from central in [default]
        ---------------------------------------------------------------------
        |                  |            modules            ||   artifacts   |
        |       conf       | number| search|dwnlded|evicted|| number|dwnlded|
        ---------------------------------------------------------------------
        |      default     |   16  |   0   |   0   |   0   ||   16  |   0   |
        ---------------------------------------------------------------------
:: retrieving :: org.apache.spark#spark-submit-
parent-7db634d3-bf23-416d-ab07-a307527f204b
        confs: [default]
        0 artifacts copied, 16 already retrieved (0kB/10ms)
25/10/23 13:39:00 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform… using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).

Spark session connected to Cassandra.
```

4 Convert Pandas to Spark DataFrame

[6]:
```
# Convert Pandas DataFrame to Spark DataFrame
df_spark = spark.createDataFrame(df_production)

# Show the schema and a few rows
df_spark.printSchema()
df_spark.show(5)
```

```
root
 |-- endtime: string (nullable = true)
 |-- lastupdatedtime: string (nullable = true)
 |-- pricearea: string (nullable = true)
 |-- productiongroup: string (nullable = true)
 |-- quantitykwh: double (nullable = true)
 |-- starttime: string (nullable = true)


25/10/23 13:39:11 WARN TaskSetManager: Stage 0 contains a task of very large
size (2877 KiB). The maximum recommended task size is 1000 KiB.
/Users/teresemyhre/opt/anaconda3/envs/D2D_env/lib/python3.12/site-
packages/pyspark/python/lib/pyspark.zip/pyspark/daemon.py:154:
DeprecationWarning: This process (pid=81616) is multi-threaded, use of fork()
may lead to deadlocks in the child.
25/10/23 13:39:16 WARN PythonRunner: Detected deadlock while completing task 0.0
in stage 0 (TID 0): Attempting to kill Python Worker
+--------------------+--------------------+---------+---------------+-----------
+--------------------+
|             endtime|
lastupdatedtime|pricearea|productiongroup|quantitykwh|           starttime|
+--------------------+--------------------+---------+---------------+-----------
+--------------------+
|2021-01-01T01:00:…|2024-12-20T10:35:…|      NO1|          hydro|
2507716.8|2021-01-01T00:00:…|
|2021-01-01T02:00:…|2024-12-20T10:35:…|      NO1|          hydro|
2494728.0|2021-01-01T01:00:…|
|2021-01-01T03:00:…|2024-12-20T10:35:…|      NO1|          hydro|
2486777.5|2021-01-01T02:00:…|
|2021-01-01T04:00:…|2024-12-20T10:35:…|      NO1|          hydro|
2461176.0|2021-01-01T03:00:…|
|2021-01-01T05:00:…|2024-12-20T10:35:…|      NO1|          hydro|
2466969.2|2021-01-01T04:00:…|
+--------------------+--------------------+---------+---------------+-----------
+--------------------+
only showing top 5 rows
```

5 Convert string timestamps to Spark timestamp type

[ ]:
```
df_spark = (
    df_spark
    .withColumn("starttime", col("starttime").cast("timestamp"))
    .withColumn("endtime", col("endtime").cast("timestamp"))
    .withColumn("lastupdatedtime", col("lastupdatedtime").cast("timestamp"))
)

df_spark.printSchema()
```

```
root
 |-- endtime: timestamp (nullable = true)
 |-- lastupdatedtime: timestamp (nullable = true)
 |-- pricearea: string (nullable = true)
 |-- productiongroup: string (nullable = true)
 |-- quantitykwh: double (nullable = true)
 |-- starttime: timestamp (nullable = true)
```

[ ]:
```
# Connect to Cassandra (same container as before)
cluster = Cluster(['127.0.0.1'])
session = cluster.connect()

# Create keyspace if it doesn't exist
session.execute("""
CREATE KEYSPACE IF NOT EXISTS elhub
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
""")

# Connect to that keyspace
session.set_keyspace('elhub')

# Create the production table
session.execute("""
CREATE TABLE IF NOT EXISTS production (
    pricearea text,
    productiongroup text,
    starttime timestamp,
    endtime timestamp,
    quantitykwh double,
    lastupdatedtime timestamp,
    PRIMARY KEY (pricearea, starttime, productiongroup)
);
""")

print("Cassandra keyspace and table created successfully!")
```

```
Cassandra keyspace and table created successfully!
```

6 Write data to Cassandra

[ ]:
```python
# Insert the Spark DataFrame into Cassandra
(
    df_spark.write
    .format("org.apache.spark.sql.cassandra")
    .mode("append")
    .options(table="production", keyspace="elhub")
    .save()
)

print("Data successfully inserted into Cassandra!")
```

```
25/10/23 13:47:43 WARN TaskSetManager: Stage 1 contains a task of very large
size (2877 KiB). The maximum recommended task size is 1000 KiB.
[Stage 1:>                                                          (0 + 8) / 8]
```

```
Data successfully inserted into Cassandra!
```

7 Extract selected columns and create visualization

[14]:
```python
# Read the necessary columns from Cassandra
df_spark_read = (
    spark.read
    .format("org.apache.spark.sql.cassandra")
    .options(table="production", keyspace="elhub")
    .load()
    .select("pricearea", "productiongroup", "starttime", "quantitykwh")
)

df_pd = df_spark_read.toPandas()
print(f"Retrieved {len(df_pd)} rows from Cassandra.")
```

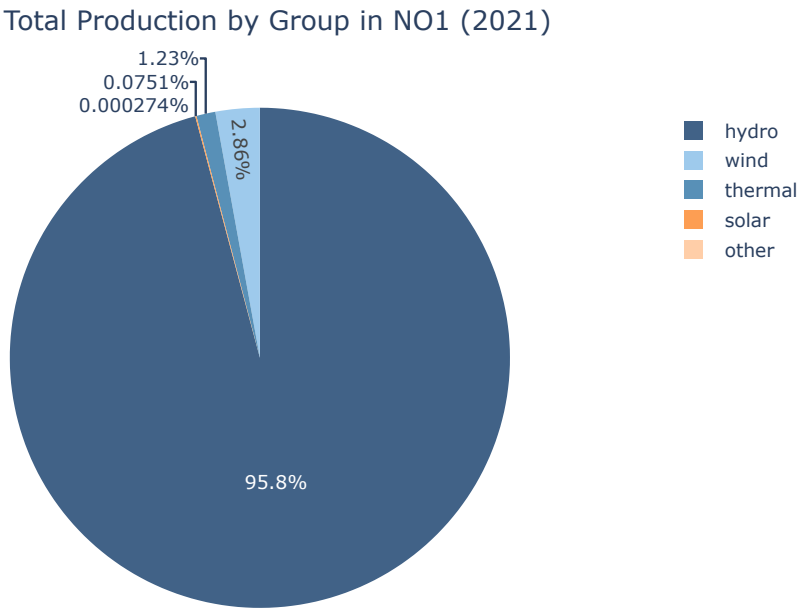
```
Retrieved 215353 rows from Cassandra.
```

8 Create a pie chart (total production by group for a chosen area)

[118]:
```python
chosen_area = "NO1"

# Aggregate yearly production by group
df_area = (
    df_pd[df_pd["pricearea"] == chosen_area]
```

```
    .groupby("productiongroup", as_index=False)["quantitykwh"]
    .sum()
)

# Custom color palette
colors = ["#416287", "#9ecaec", "#5890b7", "#fd9e53", "#ffcea8"]

# Clean pie chart
fig = px.pie(
    df_area,
    values="quantitykwh",
    names="productiongroup",
    color_discrete_sequence=colors,
    title=f"Total Production by Group in {chosen_area} (2021)",
    template="plotly_white",
)

# Center the title and fix sizing/whitespace
fig.update_layout(
    title_x=0.5,               # center title
    width=600,                 # control total figure width
    height=600,                # control total figure height
)

fig.show()
```

```
WARNING Thread(Thread-33 (run)) Task(Task-161)
choreographer.browser_async:browser_async.py:_close()- Resorting to unclean kill
browser.
```

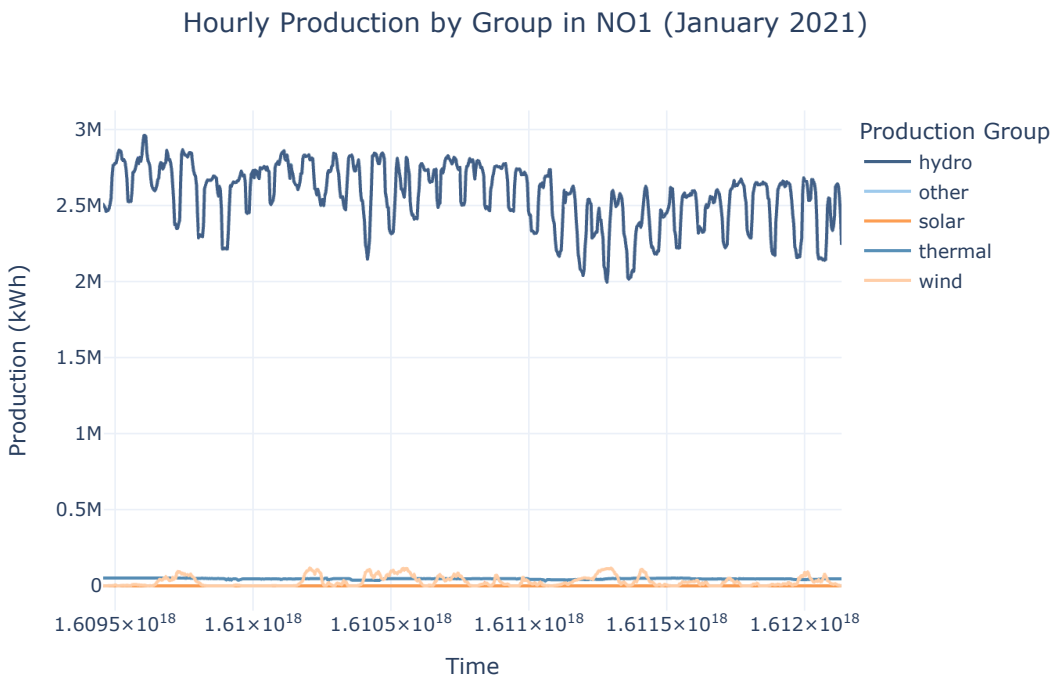


The pie chart shows the total electricity production for a selected price area in 2021.
Hydropower dominates most areas, while other sources such as wind, solar, and thermal make up smaller shares.

[119]:
```
chosen_area = "NO1"

# Filter for chosen area and first month (January)
df_jan = df_pd[
    (df_pd["pricearea"] == chosen_area)
    & (pd.to_datetime(df_pd["starttime"]).dt.month == 1)
].copy()

# Ensure time is datetime and sort
df_jan["starttime"] = pd.to_datetime(df_jan["starttime"])
df_jan.sort_values("starttime", inplace=True)

# Plot
colors = ["#416287", "#9ecaec", "#fd9e53", "#5890b7", "#ffcea8"]

fig = px.line(
    df_jan,
    x="starttime",
```

```
    y="quantitykwh",
    color="productiongroup",
    color_discrete_sequence=colors,
    title=f"Hourly Production by Group in {chosen_area} (January 2021)",
    template="plotly_white"
)

# Update layout for better readability
fig.update_layout(
    title_x=0.5,
    width=800,
    height=450,
    xaxis_title="Time",
    yaxis_title="Production (kWh)",
    legend_title_text="Production Group",
)

fig.show()
```

```
WARNING Thread(Thread-35 (run)) Task(Task-199)
choreographer.browser_async:browser_async.py:_close()- Resorting to unclean kill
browser.
```



The line plot shows hourly electricity production in price area NO1 during January 2021. Hydropower clearly dominates and remains relatively stable, though small daily fluctuations are visible. Wind production is minor but variable, with short peaks throughout the month, while solar and thermal sources contribute only marginally to total output.

9 Insert Spark-extracted data into MongoDB

[ ]:
```
# Connect to MongoDB, reuse 'uri' from previous code
client = MongoClient(uri, server_api=ServerApi('1'))
db = client["elhub"]
collection = db["production"]

# Convert Spark DataFrame to Pandas
df_mongo = df_spark_read.toPandas()
# look at df_mongo.head()
print(df_mongo.head())

# Convert to list of dictionaries (MongoDB format)
records = df_mongo.to_dict("records")

# Insert only if collection is empty (prevents duplicates)
if collection.count_documents({}) == 0:
    collection.insert_many(records)
    print(f"Inserted {len(records)} new records into 'production'.")
else:
    print(f"Collection already contains data - skipping insert.")
```

```
  pricearea productiongroup  starttime  quantitykwh
0       NO1           hydro 2021-01-01  2507716.800
1       NO1           other 2021-01-01        0.000
2       NO1           solar 2021-01-01        6.106
3       NO1         thermal 2021-01-01    51369.035
4       NO1            wind 2021-01-01      937.072
Inserted 215353 records into MongoDB collection 'production'.
```

10 Quick verification

[105]:
```
print("Total documents in collection:", collection.count_documents({}))
print("Sample document:")
print(collection.find_one())
```

```
Total documents in collection: 215353
Sample document:
{'_id': ObjectId('68fa2e189f7fc889c9d22c1e'), 'pricearea': 'NO1',
'productiongroup': 'hydro', 'starttime': datetime.datetime(2021, 1, 1, 0, 0),
```

```
'quantitykwh': 2507716.8}
```

[112]:
```
spark.stop()
print("Spark session stopped.")
```

```
Spark session stopped.
```

11 Log of compulsory work

At the start of this assignment, I had already set up both Cassandra and Spark from earlier work, which made the first part easier. I began by testing whether the connection between Cassandra and Spark worked correctly. This was done directly from the terminal, where I verified that Spark could read and write data to Cassandra. Once that was confirmed, I could continue to the data processing part in the Jupyter Notebook.

When I started working with the Elhub API, I spent quite a bit of time understanding how to retrieve the correct data. The API required that I download one month at a time, not the full year in a single request. It took several attempts and small adjustments before I managed to get stable responses and gather all twelve months of data for 2021. Once the data collection worked, I followed the steps described in the assignment to extract, transform, and load the data into Cassandra using Spark.

After preparing the data in Cassandra, I tested reading it back into Spark and verified that the schema and data types were correct. I then inserted the dataset into MongoDB. During this process, I switched from using an .env file to a secrets.toml file for storing the database connection string. This change made the setup cleaner and more secure, and it worked both locally and later in the Streamlit Cloud deployment.

In the Streamlit part, I created a new page that was first called "MongoDB." I implemented all the functionality described in the task: connecting to the MongoDB cluster, displaying plots, and making the interface interactive. Later, I renamed and reorganized the pages to improve structure and readability. I used two columns for layout, added radio buttons and selection elements, and created both pie charts and line charts using Plotly Express. I spent some time adjusting the visual design, such as colors, labels, and layout, to make the plots look cleaner and easier to interpret.

Finally, I updated the requirements.txt file so that the app could be deployed without issues. I tested the application locally using localhost and only deployed it once everything looked and worked as intended. The whole process gave a good understanding of how data can move through a pipeline: from extraction with an API, processing in Spark, storage in Cassandra and MongoDB, and finally interactive visualization in Streamlit.