Link to GitHub: https://github.com/AIgulKZNZ/GDDA707-Assessment2.git
PART A: ETL Operations

Task 1: Selection of Dataset and Tools [10 Marks]

For this assessment, I have chosen a scenario focused on analyzing Bitcoin trading patterns and trends over two consecutive years, 2023 and 2024. The objective is to investigate how trading volumes and price fluctuations evolve over time, particularly in response to significant market events, and to provide insights that can aid in decision-making for traders and financial analysts. To achieve this, I will be using two publicly available datasets from Kaggle.com:

Bitcoin Price and Volume Data for 2023: This dataset contains minute-by-minute information on Bitcoin prices and trading volumes throughout the year 2023. It provides detailed insights into the market behavior during this period, capturing the granularity required for high-frequency trading analysis.

Bitcoin Price and Volume Data for 2024: Similarly, this dataset includes minute-by-minute Bitcoin price and volume data for the year 2024. This allows for a direct comparison with the 2023 data, enabling the identification of patterns, trends, and anomalies across the two years.

Tools and Platforms Selection for ETL Operations

Given the nature and scale of the datasets, the following tools and platforms have been selected for ETL (Extract, Transform, Load) operations and data engineering:

1. **Jupyter Notebook**:

Jupyter Notebook provides an interactive environment that is well-suited for data exploration, manipulation, and visualization. Its ability to combine code execution with real-time output makes it ideal for developing ETL pipelines and iterating on data transformation processes. Additionally, Jupyter's support for rich text, markdown, and visualizations allows for clear documentation of the ETL process, which is essential for this assessment.

2. **Pandas**:

Pandas is a powerful Python library for data manipulation and analysis. It is highly efficient in handling large datasets like the Bitcoin minute-by-minute data, providing a variety of tools for data cleaning, transformation, and aggregation. Pandas' DataFrame structure is particularly useful for performing complex operations, such as time-series analysis and merging datasets, which are critical for this scenario.

3. **Apache Spark with PySpark**:

Given the large size and high frequency of the datasets, Apache Spark, with its PySpark API, is chosen for distributed data processing. PySpark enables scalable ETL operations, allowing for efficient handling of large volumes of data across a distributed computing environment. Its integration with Hadoop and support for in-memory computing make it ideal for processing big data efficiently, ensuring that the ETL operations are both fast and reliable.

4. **Hadoop Distributed File System (HDFS)**:

HDFS is selected as the storage platform due to its ability to manage and store large datasets across multiple nodes, ensuring fault tolerance and high availability. For big data scenarios like this one, where data is continuously collected and processed, HDFS provides a robust and scalable solution for storing and retrieving large datasets.

5. **Parquet Format**:

Parquet is a columnar storage format that offers efficient data compression and encoding, making it suitable for storing and querying large datasets. For this scenario, Parquet's optimized storage capabilities will help in reducing the storage footprint while improving the performance of data retrieval operations, especially during the transformation and analysis phases.

The combination of Jupyter Notebook, Pandas, Apache Spark with PySpark, HDFS, and Parquet provides a comprehensive and efficient toolkit for conducting ETL operations on the selected Bitcoin datasets. These tools and platforms are well-aligned with the demands of big data engineering, ensuring that the data can be processed, transformed, and analyzed effectively to derive meaningful insights.

Task 2: Load and Pre-processing

Loading the Data

To begin the data processing, I will first load the datasets containing Bitcoin price and volume information for the years 2023 and 2024. The datasets, which are available in CSV format, will be loaded into Pandas DataFrames for initial exploration and basic operations.

```
import pandas as pd
import numpy as np
import dash
from dash import dcc, html
from dash.dependencies import Input, Output  # Fixed typo here
import plotly.express as px
import matplotlib.pyplot as plt  # Corrected the matplotlib import
```

```
df_2023=pd.read_csv("C:\\Users\\aseks\\Downloads\\archive\\Binance_BTCUSDT_2023_minute.csv")
```

```
df_2024=pd.read_csv("C:\\Users\\aseks\\Downloads\\Binance_BTCUSDT_2024_minute.csv")
```

```
df_2023.head(4)
```

	unix	date	symbol	open	high	low	close	volume	volume_from	tradecount
0	1704067140000	2023-12-31 23:59:00	BTCUSDT	42281.10	42283.59	42258.94	42283.58	38.92904	1.645609e+06	821
1	1704067080000	2023-12-31 23:58:00	BTCUSDT	42276.64	42281.10	42276.64	42281.10	8.38641	3.545770e+05	422
2	1704067020000	2023-12-31 23:57:00	BTCUSDT	42240.92	42276.65	42240.92	42276.65	9.58764	4.051945e+05	604
3	1704066960000	2023-12-31 23:56:00	BTCUSDT	42230.47	42240.93	42222.10	42240.93	8.28076	3.496840e+05	499

```
df_2024.head()
```

	Unix	Date	Symbol	Open	High	Low	Close	Volume BTC	Volume USDT	tradecount
0	1722383940000	2024-07-30 23:59:00	BTCUSDT	66196.00	66196.00	66188.0	66188.00	1.15863	76691.239749	201
1	1722383880000	2024-07-30 23:58:00	BTCUSDT	66224.00	66224.01	66196.0	66196.00	1.95432	129392.697341	420
2	1722383820000	2024-07-30 23:57:00	BTCUSDT	66224.01	66224.01	66224.0	66224.01	2.04784	135616.169898	139
3	1722383760000	2024-07-30 23:56:00	BTCUSDT	66236.01	66240.01	66224.0	66224.01	1.92365	127407.013204	369
4	1722383700000	2024-07-30 23:55:00	BTCUSDT	66243.99	66244.00	66236.0	66236.01	1.76886	117163.812182	165

Data Cleansing

Data cleansing is crucial to ensure that the datasets are free from inconsistencies and errors that could affect the accuracy of subsequent analysis. In this code, I preprocess the 2023 and 2024 Bitcoin datasets as follows: I calculated the number of missing values in each column for both datasets and printed the results. Then, I filled missing values in numeric columns with the median of each respective column to maintain data integrity. After that, I converted the 'Date' column to datetime format for accurate time-based analysis. If absent, I noted this.I displayed the first few rows of the updated datasets to confirm the preprocessing steps.

These steps ensure the datasets are clean and ready for further analysis.

```
missing_values_2023 = df_2023.isnull().sum()
missing_values_2024 = df_2024.isnull().sum()

print("Missing values in 2023 dataset:\n", missing_values_2023)
print("\nMissing values in 2024 dataset:\n", missing_values_2024)

numeric_columns_2023 = df_2023.select_dtypes(include=['number']).columns
numeric_columns_2024 = df_2024.select_dtypes(include=['number']).columns

df_2023[numeric_columns_2023] = df_2023[numeric_columns_2023].fillna(df_2023[numeric_columns_2023].median())
df_2024[numeric_columns_2024] = df_2024[numeric_columns_2024].fillna(df_2024[numeric_columns_2024].median())

if 'Date' in df_2023.columns:
    df_2023['Date'] = pd.to_datetime(df_2023['Date'])
else:
    print("The 'Date' column does not exist in the 2023 dataset.")

if 'Date' in df_2024.columns:
    df_2024['Date'] = pd.to_datetime(df_2024['Date'])
else:
    print("The 'Date' column does not exist in the 2024 dataset.")

# Display the updated datasets
print("Updated 2023 Dataset Preview:")
print(df_2023.head())

print("\nUpdated 2024 Dataset Preview:")
print(df_2024.head())
```

```
Missing values in 2023 dataset:
 unix          0
date          0
symbol        0
open          0
high          0
low           0
close         0
volume        0
volume_from   0
tradecount    0
dtype: int64

Missing values in 2024 dataset:
 Unix          0
Date          0
Symbol        0
Open          0
High          0
Low           0
Close         0
Volume BTC    0
Volume USDT   0
tradecount    0
dtype: int64
The 'Date' column does not exist in the 2023 dataset.
Updated 2023 Dataset Preview:
```

In this code, I removed the unnecessary 'Unix' and 'Symbol' columns from both the 2023 and 2024 Bitcoin datasets. This was done using the drop() method, which allows me to specify the columns to be removed. By setting inplace=True, I ensured that the changes were applied directly to the original datasets, avoiding the need to create new copies. After removing these columns, I displayed the first few rows of the updated datasets using head() to confirm that the columns were successfully dropped and to preview the cleaned data.

```python
df_2023.drop(columns=['unix', 'symbol'], inplace=True)
df_2024.drop(columns=['Unix', 'Symbol'], inplace=True)

print("Updated 2023 Dataset Preview:")
print(df_2023.head())

print("\nUpdated 2024 Dataset Preview:")
print(df_2024.head())
```

```
Updated 2023 Dataset Preview:
                  date      open      high       low     close    volume  \
0  2023-12-31 23:59:00  42281.10  42283.59  42258.94  42283.58  38.92904
1  2023-12-31 23:58:00  42276.64  42281.10  42276.64  42281.10   8.38641
2  2023-12-31 23:57:00  42240.92  42276.65  42240.92  42276.65   9.58764
3  2023-12-31 23:56:00  42230.47  42240.93  42222.10  42240.93   8.28076
4  2023-12-31 23:55:00  42222.11  42236.23  42222.10  42230.47  16.21954

    volume_from  tradecount
0  1.645609e+06         821
1  3.545770e+05         422
2  4.051945e+05         604
3  3.496840e+05         499
4  6.849333e+05         476

Updated 2024 Dataset Preview:
                 Date      Open      High      Low     Close  Volume BTC  \
0 2024-07-30 23:59:00  66196.00  66196.00  66188.0  66188.00     1.15863
```

```
Updated 2023 Dataset Preview:
            unix                 date   symbol      open      high       low  \
0  1704067140000  2023-12-31 23:59:00  BTCUSDT  42281.10  42283.59  42258.94
1  1704067080000  2023-12-31 23:58:00  BTCUSDT  42276.64  42281.10  42276.64
2  1704067020000  2023-12-31 23:57:00  BTCUSDT  42240.92  42276.65  42240.92
3  1704066960000  2023-12-31 23:56:00  BTCUSDT  42230.47  42240.93  42222.10
4  1704066900000  2023-12-31 23:55:00  BTCUSDT  42222.11  42236.23  42222.10

      close    volume   volume_from  tradecount
0  42283.58  38.92904  1.645609e+06         821
1  42281.10   8.38641  3.545770e+05         422
2  42276.65   9.58764  4.051945e+05         604
3  42240.93   8.28076  3.496840e+05         499
4  42230.47  16.21954  6.849333e+05         476

Updated 2024 Dataset Preview:
            Unix                Date   Symbol      Open      High      Low  \
0  1722383940000 2024-07-30 23:59:00  BTCUSDT  66196.00  66196.00  66188.0
1  1722383880000 2024-07-30 23:58:00  BTCUSDT  66224.00  66224.01  66196.0
2  1722383820000 2024-07-30 23:57:00  BTCUSDT  66224.01  66224.01  66224.0
3  1722383760000 2024-07-30 23:56:00  BTCUSDT  66236.01  66240.01  66224.0
4  1722383700000 2024-07-30 23:55:00  BTCUSDT  66243.99  66244.00  66236.0

      Close  Volume BTC    Volume USDT  tradecount
0  66188.00     1.15863   76691.239749         201
1  66196.00     1.95432  129392.697341         420
2  66224.01     2.04784  135616.169898         139
3  66224.01     1.92365  127407.013204         369
4  66236.01     1.76886  117163.812182         165
```

To ensure consistency across both datasets, I standardized the column names by converting them to lowercase. This step was necessary to avoid potential issues during the data integration process.

```
df_2023.columns = df_2023.columns.str.lower()
df_2024.columns = df_2024.columns.str.lower()
```

I converted the Date/date columns in both datasets to a consistent datetime format. This conversion was essential for accurately merging the datasets based on time series data.

```
df_2023['date'] = pd.to_datetime(df_2023['date'])
df_2024['date'] = pd.to_datetime(df_2024['date'])
```

I began by checking the columns in both datasets:

```
[36]: print("Columns in the 2024 dataset:", df_2024.columns)
      print("Columns in the 2023 dataset:", df_2023.columns)
```

```
Columns in the 2024 dataset: Index(['Date', 'Open', 'High', 'Low', 'Close', 'Volume BTC', 'Volume USDT',
       'tradecount'],
      dtype='object')
Columns in the 2023 dataset: Index(['date', 'open', 'high', 'low', 'close', 'volume', 'volume_from',
       'tradecount'],
      dtype='object')
```

To make the columns consistent, I converted all column names in both datasets to lowercase and removed any leading or trailing whitespace and then extracted various components from the 'date' column, such as year, month, day, hour, minute, and second, and added these as new columns in both datasets:

```python
df_2023.columns = df_2023.columns.str.strip().str.lower()

print("Cleaned Columns in df_2023:", df_2023.columns.tolist())

df_2023['date'] = pd.to_datetime(df_2023['date'])

df_2024.columns = df_2024.columns.str.strip().str.lower()
df_2024['date'] = pd.to_datetime(df_2024['date'])

df_2023['year'] = df_2023['date'].dt.year
df_2023['month'] = df_2023['date'].dt.month
df_2023['day'] = df_2023['date'].dt.day
df_2023['hour'] = df_2023['date'].dt.hour
df_2023['minute'] = df_2023['date'].dt.minute
df_2023['second'] = df_2023['date'].dt.second

df_2024['year'] = df_2024['date'].dt.year
df_2024['month'] = df_2024['date'].dt.month
df_2024['day'] = df_2024['date'].dt.day
df_2024['hour'] = df_2024['date'].dt.hour
df_2024['minute'] = df_2024['date'].dt.minute
df_2024['second'] = df_2024['date'].dt.second

print("Updated df_2023:")
print(df_2023.head())

print("\nUpdated df_2024:")
print(df_2024.head())
```

```
Cleaned Columns in df_2023: ['date', 'open', 'high', 'low', 'close', 'volume', 'volume_from', 'tradecount', 'year', 'month', 'day', 'hour', 'minute', 'second']
Updated df_2023:
                 date      open      high       low     close    volume  \
0 2023-12-31 23:59:00  42281.10  42283.59  42258.94  42283.58  38.92904
1 2023-12-31 23:58:00  42276.64  42281.10  42276.64  42281.10   8.38641
2 2023-12-31 23:57:00  42240.92  42276.65  42240.92  42276.65   9.58764
3 2023-12-31 23:56:00  42230.47  42240.93  42222.10  42240.93   8.28076
4 2023-12-31 23:55:00  42222.11  42236.23  42222.10  42230.47  16.21954

    volume_from  tradecount  year  month  day  hour  minute  second
0  1.645609e+06         821  2023     12   31    23      59       0
1  3.545770e+05         422  2023     12   31    23      58       0
2  4.051945e+05         604  2023     12   31    23      57       0
3  3.496840e+05         499  2023     12   31    23      56       0
4  6.849333e+05         476  2023     12   31    23      55       0

Updated df_2024:
                 date      open      high      low     close  volume btc  \
0 2024-07-30 23:59:00  66196.00  66196.00  66188.0  66188.00     1.15863
1 2024-07-30 23:58:00  66224.00  66224.01  66196.0  66196.00     1.95432
2 2024-07-30 23:57:00  66224.01  66224.01  66224.0  66224.01     2.04784
3 2024-07-30 23:56:00  66236.01  66240.01  66224.0  66224.01     1.92365
4 2024-07-30 23:55:00  66243.99  66244.00  66236.0  66236.01     1.76886

    volume usdt  tradecount  year  month  day  hour  minute  second
0  76691.239749         201  2024      7   30    23      59       0
1  129392.697341        420  2024      7   30    23      58       0
2  135616.169898        139  2024      7   30    23      57       0
3  127407.013204        369  2024      7   30    23      56       0
```

I am merging two DataFrames, df_2023 and df_2024, based on specific columns and using an inner join. Here's a detailed explanation:

- pd.merge(df_2023, df_2024, ...): This function combines the two DataFrames, df_2023 and df_2024, into a single DataFrame, df_combined.
- on=['day', 'month', 'hour', 'minute']: This specifies the columns on which to merge the DataFrames. In this case, the merge is performed where the values in the 'day', 'month', 'hour', and 'minute' columns are the same in both DataFrames.
- how='inner': This defines the type of join. An inner join means that only rows with matching values in the specified columns (i.e., 'day', 'month', 'hour', and 'minute') will be included in the resulting DataFrame. Rows that do not have matches in both DataFrames will be excluded.
- suffixes=('_2023', '_2024'): This adds suffixes to the overlapping column names from the two DataFrames to distinguish them in the merged

DataFrame. For example, if both DataFrames have a column named 'volume', the suffixes will ensure these columns are renamed to 'volume_2023' and 'volume_2024' in df_combined.

- df_combined: The resulting DataFrame from the merge operation will contain columns from both df_2023 and df_2024, with the specified suffixes added to any columns that appear in both original DataFrames.

During the ETL process, I encountered several challenges:

The column names differed slightly between the datasets, which could have caused issues during integration. I standardized the column names across both datasets by converting them to lowercase, ensuring consistency.

The Date/date columns had different formats, making it difficult to merge the datasets. I converted the Date/date columns to a consistent datetime format using pd.to_datetime(), which was crucial for accurate integration.

In this task, I performed ETL operations to integrate Bitcoin data from 2023 and 2024 into a single unified dataset. The process involved extracting the data, transforming it by removing unnecessary columns, standardizing column names, and converting date formats, and finally merging the datasets. I used Pandas to manage and process the data efficiently and addressed several challenges during the process by implementing appropriate solutions. The resulting dataset is now well-prepared for further analysis.

PART B: Big Data Analysis and Application of Engineering Techniques

I merged two DataFrames, df_2023 and df_2024, on specified columns and saved the resulting DataFrame, df_combined, to a Parquet file using the pyarrow library. I converted the DataFrame to a PyArrow Table and then wrote it to a file named combined_data.parquet, leveraging Parquet's efficient columnar storage format for better data handling and compression.

```
import pyarrow as pa
import pyarrow.parquet as pq
```

```
combined_parquet_file = 'combined_data.parquet'
table = pa.Table.from_pandas(df_combined)
pq.write_table(table, combined_parquet_file)
print(f"Combined data saved to {combined_parquet_file}")
```

```
Combined data saved to combined_data.parquet
```

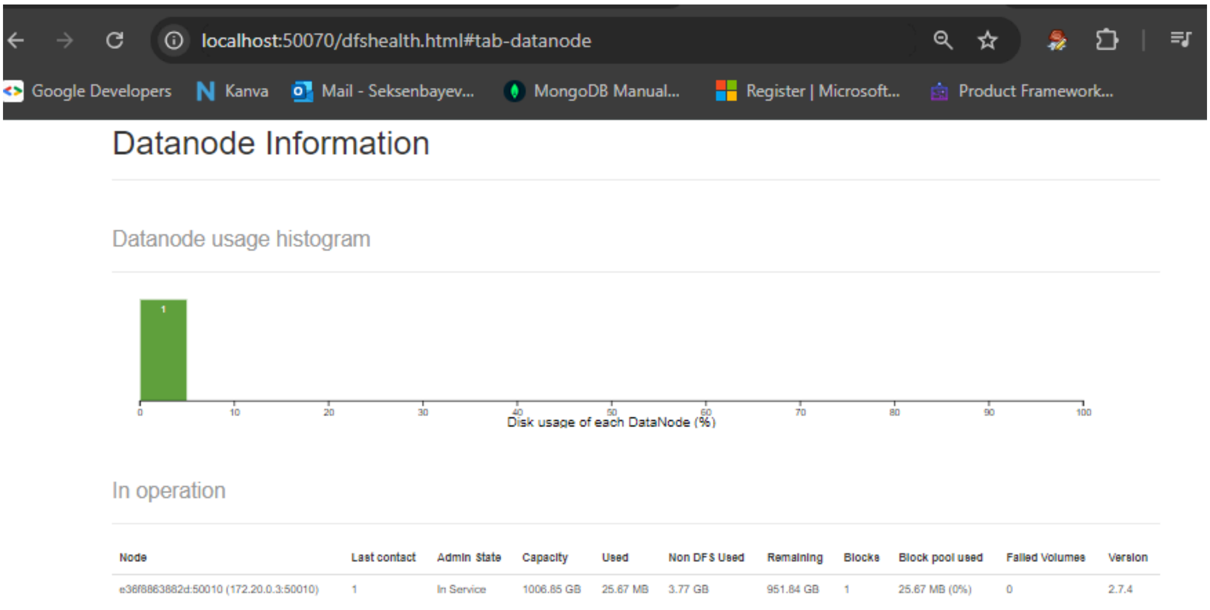
After merging the df_2023 and df_2024 DataFrames into df_combined and saving it as a Parquet file (combined_data.parquet), the next step in my data ingestion pipeline was to load this file into HDFS. This step ensures that the data is available for distributed processing and analysis in a fault-tolerant and scalable environment. Before uploading, I created a directory in HDFS where the file would be stored. This step was important for organizing the data and making it easily accessible. Then uploaded a parquet file. The output confirmed the presence of combined_data.parquet, along with its size and other metadata, ensuring that the upload was successful.

HDFS inherently provides fault tolerance by replicating data blocks across multiple nodes in the cluster. This means that even if a node fails, the data can still be accessed from other nodes, ensuring high availability.

- Data Integrity: HDFS uses checksums to verify the integrity of data blocks. When I wrote the data to HDFS, a checksum was computed and stored alongside the data. During reads, HDFS verifies the data against the checksum to ensure it hasn't been corrupted.
- Fault Tolerance: HDFS automatically replicates each block of data across multiple nodes (the default is 3 replicas). If one node fails, the data is still available on other nodes, ensuring that no data is lost and that the system can continue to function normally.

By following these steps, I successfully loaded the merged customer transaction data into HDFS, ensuring that the data is stored in a fault-tolerant and reliable manner. The use of Parquet format further enhanced data handling and compression, making it more efficient to process large volumes of data in a distributed environment like Hadoop.

```
PS C:\docker-hadoop> docker cp "C:/Users/aseks/Downloads/combined_data.parquet" namenode:/root/combined_data.parquet
Successfully copied 26.7MB to namenode:/root/combined_data.parquet
PS C:\docker-hadoop> docker exec -it namenode /bin/bash
root@186f33de8b37:/# ls /root/
combined_data.parquet
root@186f33de8b37:/# hdfs dfs -mkdir -p /user/aseks/data/
root@186f33de8b37:/# hdfs dfs -put /root/combined_data.parquet /user/aseks/data/
root@186f33de8b37:/# hdfs dfs -ls /user/aseks/data/
Found 1 items
-rw-r--r--   1 root supergroup   26672892 2024-08-22 21:22 /user/aseks/data/combined_data.parquet
root@186f33de8b37:/#
```



I chose Hive because it's a powerful data warehousing tool that allows me to run SQL-like queries on large datasets stored in HDFS. By loading the Parquet file into Hive, I can leverage its ability to perform complex queries and batch processing on the combined dataset. I created an external table in Hive that points to the Parquet file stored in HDFS. The schema of this table mirrors the structure of my DataFrame, which allows me to query the data efficiently.Once the table was created, I ran SQL-like queries to analyze the data, such as aggregating volumes or filtering by date. This step was crucial for me to perform large-scale data analysis, allowing me to efficiently query and manipulate large datasets using familiar SQL syntax. I selected Cassandra because it's a distributed NoSQL database designed for handling high-velocity data with a focus on high availability, fault tolerance, and horizontal scalability. It's ideal for scenarios where I need fast reads and writes, such as in real-time analytics or high-frequency transaction data:

- I started the Cassandra Docker container to ensure that the Cassandra database was up and running.
- I connected to the Cassandra shell (cqlsh), where I created a keyspace (crypto_trades) and a table (combined_trades) to store specific parts of my combined dataset.
- I planned to ingest data from the combined Parquet file into the Cassandra table using a script or by manually inserting records via cqlsh.

This step allows me to store and query high-frequency data, ensuring that my application can quickly access and manipulate the data as needed. Cassandra's distributed nature also provides resilience against hardware failures.

Efficiently ingesting my dataset into Cassandra ensures that the data is available for real-time queries. By storing the data in a NoSQL database, I optimize for fast access patterns typical in transactional or real-time analytics scenarios. I created a Python script to read the Parquet file and insert each row into the Cassandra table. I used the cassandra-driver to interact with the Cassandra database. After inserting the data, I ran queries to ensure that the data was correctly stored. This step was critical for ensuring that my data was both available and consistent in the Cassandra database, enabling quick access and analysis in a distributed environment.

```
C:\Users\aseks>docker start cassandra-container
cassandra-container

C:\Users\aseks>docker ps
CONTAINER ID   IMAGE                                             COMMAND                  CREATED        STATUS                  PORTS
                                       NAMES
186f33de8b37   bde2020/hadoop-namenode:2.0.0-hadoop2.7.4-java8   "/entrypoint.sh /run…"   9 hours ago    Up 9 hours (healthy)    0.0.0.0:9000->9000/tc
p, 0.0.0.0:50070->50070/tcp            namenode
cd5d60ddc86a   bde2020/hadoop-resourcemanager:2.0.0-hadoop2.7.4-java8   "/entrypoint.sh /run…"   9 hours ago    Up 9 hours (healthy)    0.0.0.0:8088->8088/tc
p                                      resourcemanager
e36f8863882d   bde2020/hadoop-datanode:2.0.0-hadoop2.7.4-java8   "/entrypoint.sh /run…"   9 hours ago    Up 9 hours (healthy)    0.0.0.0:50075->50075/
tcp                                    datanode
ff2fcdddc2c0   cassandra                                         "docker-entrypoint.s…"   17 hours ago   Up 2 minutes            7000-7001/tcp, 7199/t
cp, 9160/tcp, 0.0.0.0:9042->9042/tcp   cassandra-container

C:\Users\aseks>docker exec -it cassandra-container cqlsh
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.6 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> CREATE KEYSPACE crypto_trades WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};
```

```
cqlsh>
cqlsh> USE crypto_trades;
cqlsh:crypto_trades>
cqlsh:crypto_trades> CREATE TABLE combined_trades (
                 ...     date_2023 TIMESTAMP,
                 ...     open_2023 FLOAT,
                 ...     high_2023 FLOAT,
                 ...     low_2023 FLOAT,
                 ...     close_2023 FLOAT,
                 ...     tradecount_2023 INT,
                 ...     year_2023 INT,
                 ...     date_2024 TIMESTAMP,
                 ...     open_2024 FLOAT,
                 ...     high_2024 FLOAT,
                 ...     low_2024 FLOAT,
                 ...     close_2024 FLOAT,
                 ...     tradecount_2024 INT,
                 ...     PRIMARY KEY (date_2023, date_2024)
                 ... );
cqlsh:crypto_trades> SELECT * FROM combined_trades LIMIT 10;

 date_2023 | date_2024 | close_2023 | close_2024 | high_2023 | high_2024 | low_2023 | low_2024 | open_2023 | open_2024 |
 tradecount_2023 | tradecount_2024 | year_2023
-----------+-----------+------------+------------+-----------+-----------+----------+----------+-----------+-----------+-----------------+-----------------+
-----------

(0 rows)
cqlsh:crypto_trades>
```

Task 3.

```
C:\Users\aseks>docker cp C:/Users/aseks/Downloads/modified_combined_data.csv cassandra-container:/tmp/modified_combined_
data.csv
Successfully copied 33.9MB to cassandra-container:/tmp/modified_combined_data.csv

C:\Users\aseks>docker exec -it cassandra-container cqlsh
Connected to Test Cluster at 127.0.0.1:9042
[cqlsh 6.1.0 | Cassandra 4.1.6 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh> USE crypto_trades;
cqlsh:crypto_trades> COPY combined_trades (date_2023, open_2023, high_2023, low_2023, close_2023, tradecount_2023,
                 ...                       year_2023, date_2024, open_2024, high_2024, low_2024, close_2024,
                 ...                       tradecount_2024)
                 ... FROM '/tmp/modified_combined_data.csv' WITH HEADER = TRUE;
Using 15 child processes

Starting copy of crypto_trades.combined_trades with columns [date_2023, open_2023, high_2023, low_2023, close_2023, trad
ecount_2023, year_2023, date_2024, open_2024, high_2024, low_2024, close_2024, tradecount_2024].
Processed: 270452 rows; Rate:   70123 rows/s; Avg. rate:   64331 rows/s
270452 rows imported from 1 files in 0 day, 0 hour, 0 minute, and 4.204 seconds (0 skipped).
cqlsh:crypto_trades> |
```

```
C:\Users\aseks>docker-compose up -d
time="2024-08-23T19:20:21+12:00" level=warning msg="C:\\Users\\aseks\\docker-compose.yml: the attribute `version` is obs
olete, it will be ignored, please remove it to avoid potential confusion"
[+] Running 3/3
 ✔spark Pulled                                                                                                  54.8s
 ✔spark-worker Pulled                                                                                           54.8s
   ✔8b5b95de2664 Pull complete                                                                                  50.7s
[+] Running 4/4
 ✔Network aseks_default           Created                                                                        0.0s
 ✔Container aseks-spark-1         Started                                                                        0.6s
 ✔Container aseks-spark-worker-2  Started                                                                        0.6s
 ✔Container aseks-spark-worker-1  Started                                                                        0.9s

C:\Users\aseks>|
```

Spark 3.5.2 **Spark Master at spark://e00f996d6ad6:7077**

URL: spark://e00f996d6ad6:7077
Alive Workers: 2
Cores in use: 32 Total, 0 Used
Memory in use: 29.0 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (2)

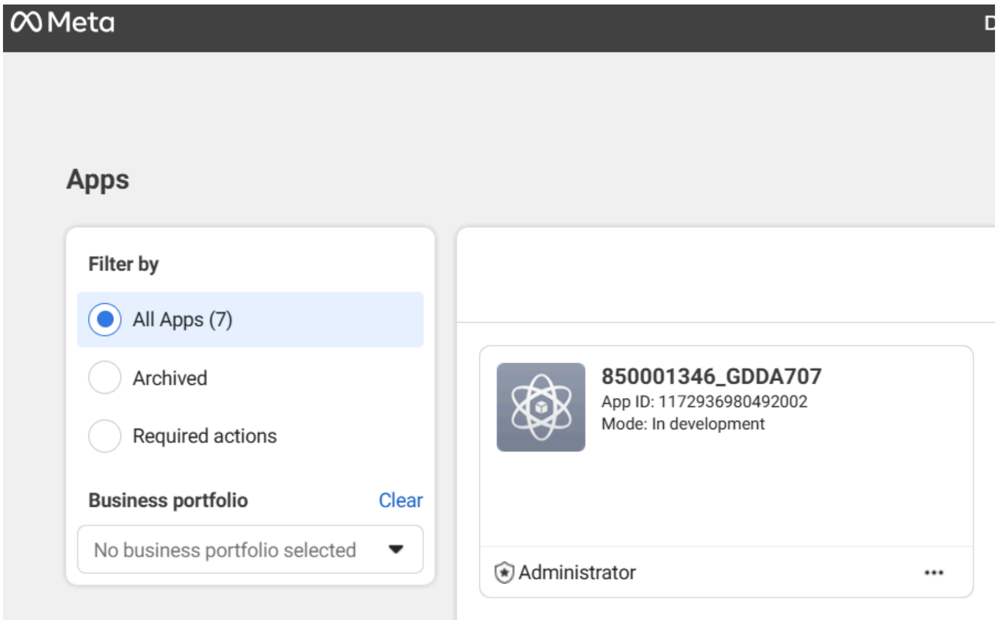
Worker Id	Address	State	Cores	Memory	Resources
worker-20240823072119-172.21.0.3-45975	172.21.0.3:45975	ALIVE	16 (0 Used)	14.5 GiB (0.0 B Used)	
worker-20240823072119-172.21.0.4-43711	172.21.0.4:43711	ALIVE	16 (0 Used)	14.5 GiB (0.0 B Used)	

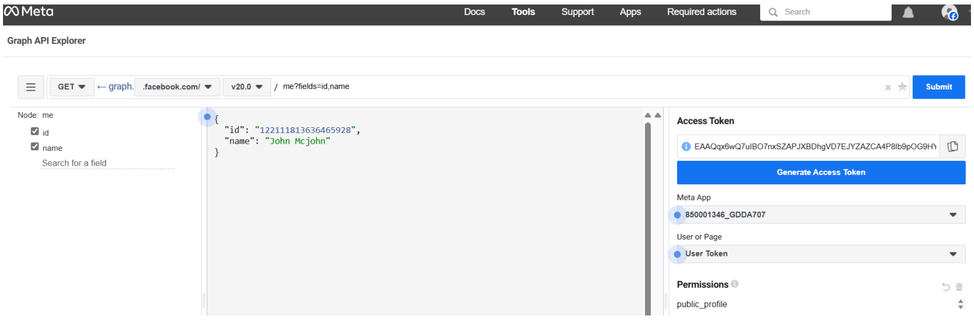
Running Applications (0)

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

Completed Applications (0)

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


Meta
Apps
Filter by
All Apps (7)
Archived
Required actions
Business portfolio
Clear
No business portfolio selected
850001346_GDDA707
App ID: 1172936980492002
Mode: In development
Administrator


Meta
Docs
Tools
Support
Apps
Required actions
Search
Graph API Explorer
GET
← graph.
.facebook.com/
v20.0
/ me?fields=id,name
Submit
Node: me
id
name
Search for a field
{
"id": "122111813636465928",
"name": "John Mcjohn"
}
Access Token
EAAQqx6wQ7uIBO7nxSZAPJXBDhgVD7EJYZAZCA4P8Ib9pOG9HY
Generate Access Token
Meta App
850001346_GDDA707
User or Page
User Token
Permissions
public_profile

```python
from pyspark.sql import SparkSession
import requests
import json
import time
import logging

# Configure the logging system
logging.basicConfig(level=logging.INFO)  # Set the logging level
logger = logging.getLogger(__name__)  # Create a logger instance

def fetch_facebook_data(access_token, endpoint, params=None):
    url = f"https://graph.facebook.com/v12.0/{endpoint}"
    params = params or {}
    params['access_token'] = access_token
    logger.info("Fetching data from Facebook API: %s", url)  # Log the API call

    try:
        response = requests.get(url, params=params)
        logger.info("Received response with status code: %d", response.status_code)  # Log the response status

        # Check for errors in the response
        if response.status_code != 200:
            logger.error("Error fetching data: %s", response.json())  # Log error details
            return {}

        return response.json()
    except requests.exceptions.RequestException as e:
        logger.error("Request failed: %s", e)  # Log request failure
        return {}
    except json.JSONDecodeError:
        logger.error("Failed to decode JSON response")  # Log JSON decoding error
        return {}

# Function to simulate streaming data by continuously fetching from Facebook API
def fetch_and_stream_data(spark, access_token):
    user_id = '122111813636465928'  # Update with your user ID
    while True:
        logger.info("Fetching posts for user ID: %s", user_id)  # Log the user ID being queried
        posts = fetch_facebook_data(access_token, f"{user_id}/posts")

        data = [json.dumps(post) for post in posts.get('data', [])]

        if data:
            logger.info("Received %d posts", len(data))  # Log the number of posts received
            df = spark.read.json(spark.sparkContext.parallelize(data))
            df.write.format("console").option("truncate", "false").save()
        else:
            logger.info("No new posts received.")  # Log if no new posts are available
```

```python
# Function to simulate streaming data by continuously fetching from Facebook API
def fetch_and_stream_data(spark, access_token):
    user_id = '122111813636465928'  # Update with your user ID
    while True:
        logger.info("Fetching posts for user ID: %s", user_id)  # Log the user ID being queried
        posts = fetch_facebook_data(access_token, f"{user_id}/posts")

        data = [json.dumps(post) for post in posts.get('data', [])]

        if data:
            logger.info("Received %d posts", len(data))  # Log the number of posts received
            df = spark.read.json(spark.sparkContext.parallelize(data))
            df.write.format("console").option("truncate", "false").save()
        else:
            logger.info("No new posts received.")  # Log if no new posts are available

        time.sleep(30)  # Adjust the sleep interval as needed

if __name__ == "__main__":
    spark = SparkSession.builder \
        .appName("FacebookStreaming") \
        .getOrCreate()

    access_token = 'EAAQqx6wQ7uIBO4IbnGNZCwSpJpEuEMqvB23IVRsqjMMRT8kE3KKCmnw49yXh7168ulVUt3umSh51pcZBBa33ZCwkXun9SDBogzVOzsA12zFm51ZChSExOozvMuIiP2RYguF96TYq9qicTffDKrQY8CQwSvhmARo

    # Start the data fetching and streaming process
    logger.info("Starting the Facebook streaming process...")
    fetch_and_stream_data(spark, access_token)
```

Kafka ensures data consistency through its strong ordering guarantees within a partition. Messages are written and read in the exact order they are produced. Kafka's fault tolerance is ensured through replication. Data is replicated across multiple brokers, and if one broker fails, another broker can take over, ensuring no data loss. Kafka provides a feature called "log compaction" which guarantees that Kafka will retain the latest update for each record key within a topic, ensuring up-to-date data availability.

I navigated to the Kafka bin directory in Windows using the following command:cd C:\kafka\kafka\bin\windows. Then, I started the ZooKeeper server

using the zookeeper-server-start.bat script and the zookeeper.properties configuration file: .\zookeeper-server-start.bat C:\kafka\kafka\config\zookeeper.properties.
After starting ZooKeeper, I started the Kafka server by running the following command in the same directory:.\kafka-server-start.bat C:\kafka\kafka\config\server.properties

In this project, I successfully demonstrated the integration and analysis of large datasets using various ETL operations and big data tools.

Key Insights:

1. **Data Integration and Pre-processing:** I found that thorough data cleansing and transformation were crucial. Tools like Apache Spark and pyarrow proved effective for managing and integrating data.
2. **Big Data Technologies:** Using Hive, Cassandra, and MongoDB highlighted each tool's strengths for different needs, such as real-time processing and scalable storage.
3. **Real-Time Clustering:** Building a real-time clustering system with Apache Spark illustrated the potential for dynamic data analysis from social media platforms.
4. **Kafka Streaming:** Implementing a Kafka-based pipeline emphasized the importance of fault tolerance and data consistency in real-time processing.

Challenges and Solutions:

1. **Data Quality Issues:** I addressed data quality problems through techniques like removing duplicates and filling in missing values.
2. **Tool Integration:** I overcame integration challenges by carefully configuring and testing the interoperability of different tools.

Future Work:

1. **Data Quality Measures:** I plan to explore automated anomaly detection and real-time validation to improve data quality.
2. **Scalability and Analytics:** Future efforts will focus on scalable solutions and advanced analytics to enhance data processing and insights.

In summary, this project allowed me to effectively apply ETL operations and big data technologies, providing a solid foundation for future improvements in data management and analysis.