Batch: A1 Roll No.: 16010123012

Experiment No. 7

Title: To perform Extract, Transform, Load (ETL) using Python

Aim: Load a dataset from external sources (web scraping), apply transformations such as scaling, normalization, and feature encoding, save the transformed data into a structured format.

Course Outcome:

CO**3:** Learn data cleaning, transformation, and feature engineering techniques.

Books/ Journals/ Websites referred:

1. The Comprehensive R Archive Network
2. Posit

Resources used:

(Students should write the data sources used)

Theory:

What is ETL?

ETL stands for "Extract, Transform, and Load" and describes the set of processes to extract data from one system, transform it, and load it into a target repository. An ETL pipeline is a traditional type of data pipeline for cleaning, enriching, and transforming data from a variety of sources before integrating it for use in data analytics, business intelligence and data science.

Key Benefits

Using an ETL pipeline to transform raw data to match the target system, allows for systematic and accurate data analysis to take place in the target repository. Specifically, the key benefits are:

1. More stable and faster data analysis on a single, pre-defined use case. This is because the data set has already been structured and transformed.
2. Easier compliance with GDPR, HIPAA, and CCPA standards. This is because users can omit any sensitive data prior to loading in the target system.
3. Identify and capture changes made to a database via the change data capture (CDC) process or technology. These changes can then be applied to another data repository or made available in a format consumable by ETL, EAI, or other types of data integration tools.

Extract > Transform > Load (ETL)

In the ETL process, transformation is performed in a staging area outside of the data warehouse and before loading it into the data warehouse. The entire data set must be transformed before loading, so transforming large data sets can take a lot of time up front. The benefit is that analysis can take place immediately once the data is loaded. This is why this process is appropriate for small data sets which require complex transformations.

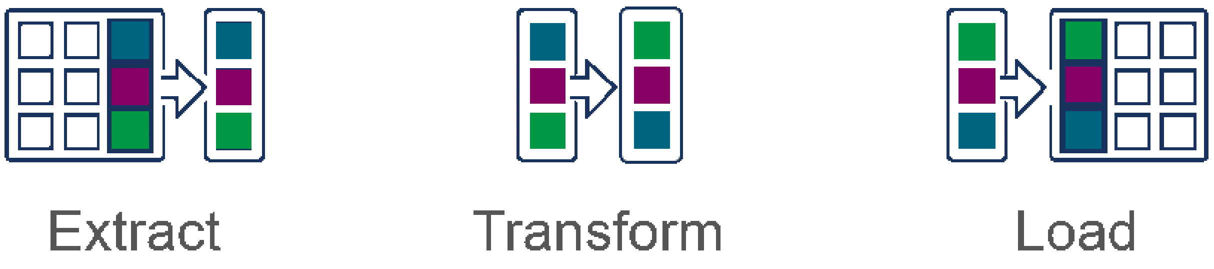


Fig. 1: Illustration showing the 3 steps of the ETL process which are extract, transform and load.

Extract > Load > Transform (ELT)

In the ELT process, data transformation is performed on an as-needed basis within the target system. This means that the ELT process takes less time. But if there is not sufficient processing power in the cloud solution, transformation can slow down the querying and analysis processes. This is why the ELT process is more appropriate for larger, structured and unstructured data sets and when timeliness is important.

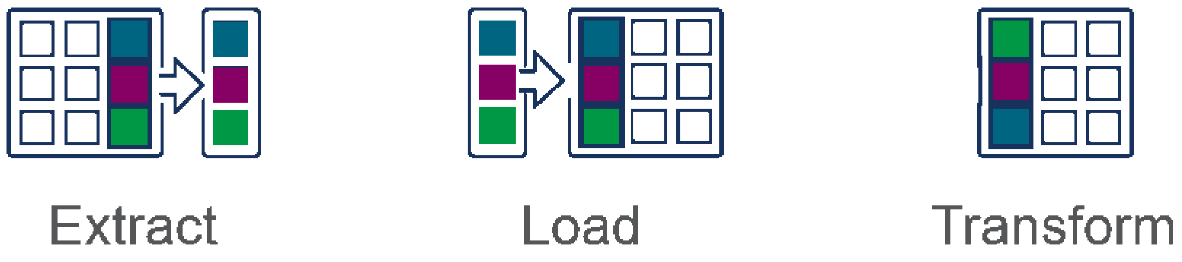


Fig. 2: Illustration showing the 3 steps of the ELT process which are extract, load and transform.

Extract phase

The Extract phase is the first step of the ETL process. It involves gathering raw data from a source. In this case, we collect data from IMDb's Top Rated Movies list using web scraping in Python. Below is a detailed step-by-step explanation of how this is done.

Step 1: Sending an HTTP Request

To access the IMDb page, we send an HTTP request using the requests library.

An HTTP request is like knocking on IMDb's door and asking for a copy of the webpage.

IMDb responds by sending back the HTML code of the page, which contains all the movie details.

Step 2: Parsing the HTML Content

The HTML code received from IMDb is not easy to read directly.

We use the BeautifulSoup library to convert the messy HTML into a structured format that we can work with.

This makes it easier to locate and extract specific movie details like titles, release years, and ratings.

Step 3: Extracting Movie Details

Once the HTML is structured, we search for specific HTML tags that contain the movie information.

We extract the following details for each movie:

Title – The name of the movie

Year – The year the movie was released

Duration – The length of the movie in hours and minutes

IMDb Rating – The rating given by IMDb users

Vote Count – The number of users who rated the movie

We also clean the extracted data by removing unwanted characters.

Step 4: Storing the Extracted Data

After extracting the data, we store it in a Pandas DataFrame (a table format in Python).

This structured table makes it easy to analyze and manipulate the data.

Finally, we save the extracted data as a CSV file so that it can be used later in the next steps of the ETL process.

Transform Phase

The Transform phase is the second step in the ETL process. After extracting raw movie data from IMDb, we need to clean, organize, and modify it so that it becomes useful for analysis. Below is a detailed step-by-step explanation of how this is done.

Step 1: Extracting the Correct Year

Sometimes, the year column may contain extra symbols or unwanted text.

We use Python to extract only the four-digit year (e.g., 1994, 2001) from the raw data.

Step 2: Converting Duration to Minutes

The duration of movies is usually written in the format "2h 30m" (2 hours 30 minutes).

We convert this into just minutes (e.g., "2h 30m" → 150 minutes) so it is easier to analyze.

Step 3: Cleaning Vote Counts

IMDb sometimes shows votes in short forms like "2M" (2 million) or "500K" (500,000).

We convert these values into actual numbers (e.g., "2M" → 2,000,000).

Step 4: Categorizing Movies by Decade

We group movies based on their release decade (e.g., movies from 1990-1999 are in the “1990s” group).

This helps in analyzing trends over different time periods.

Step 5: Classifying Movies by Duration

We divide movies into categories based on their length:

Short (<90 min)

Medium (90-120 min)

Long (120-150 min)

Very Long (150-180 min)

Epic (>180 min)

Step 6: Categorizing Movies by Rating

We group movies based on their IMDb rating:

Low (<7.5)

Average (7.5-8.0)

Good (8.0-8.5)

Excellent (8.5-9.0)

Masterpiece (9.0+)

Step 7: Handling Missing Values

If any movie is missing important details like title, year, or rating, we remove it from the dataset.

Step 8: Removing Duplicates

If the same movie appears multiple times, we remove duplicates to keep the dataset clean.

Step 9: Creating a Popularity Score

A custom popularity score is calculated based on IMDb rating and number of votes.

This helps in ranking movies based on both their quality and popularity.

Load Phase

Once the transformed movie data is ready, it needs to be stored in a structured format for further analysis. This step is called the Load Phase, where we save the processed data into a SQLite database and export it into useful formats.

1. Establish a Connection to the SQLite Database

To store the processed data, we first need to create a SQLite database engine. SQLite is a lightweight database that stores data in a single file, making it perfect for small to medium-sized datasets.

Using the SQLAlchemy create_engine function, we create a database connection:

```
engine = create_engine('sqlite:///data/processed/movies_database.sqlite')
```

This creates (or opens, if it already exists) a database file named movies_database.sqlite inside the data/processed folder.

2. Load the Transformed Data into the Main Table (movies)

The main dataset, stored in a Pandas DataFrame (df), contains individual movie details like title, rating, duration, votes, and popularity. We load this data into a table named movies inside the SQLite database:

```
df.to_sql('movies', engine, if_exists='replace', index=False)
```

This creates the movies table. If it already exists, it replaces it to ensure we always have the latest data.

Excludes the default Pandas index from being stored.

3. Aggregate and Store Data by Decades (decade_stats Table)

To analyze movie trends over time, we group movies by decade and compute:

- Total number of movies per decade
- Average rating
- Average duration
- Average popularity score

```
decade_stats = df.groupby('decade').agg({

    'title': 'count',

    'rating': 'mean',

    'duration_minutes': 'mean',

    'popularity_score': 'mean'

}).reset_index()

decade_stats.to_sql('decade_stats', engine, if_exists='replace', index=False)
```

This creates a new table decade_stats in the database.

4. Aggregate and Store Data by Rating Categories (rating_stats Table)

Another useful breakdown is by rating category (e.g., “Excellent”, “Good”, “Average”). We compute:

- Total movies per category
- Average release year
- Average duration
- Average votes count

```
rating_stats = df.groupby('rating_category').agg({

    'title': 'count',

    'year': 'mean',

    'duration_minutes': 'mean',

    'votes_count': 'mean'

}).reset_index()

rating_stats.to_sql('rating_stats', engine, if_exists='replace', index=False)
```

This creates a new table rating_stats in the database.

5. Export Aggregated Data to CSV Files

To make the aggregated tables available for external use, we save them as CSV files:

```
decade_stats.to_csv('data/processed/decade_stats.csv', index=False)

rating_stats.to_csv('data/processed/rating_stats.csv', index=False)
```

These CSV files allow us to analyze the data in Excel, Pandas, or other tools without requiring database access.

6. Verify Data Integrity with a Record Count

To ensure the data was successfully stored, we run a query to count the number of movies in the database:

```
conn = sqlite3.connect('data/processed/movies_database.sqlite')

movie_count = pd.read_sql("SELECT COUNT(*) FROM movies", conn).iloc[0, 0]

conn.close()
```

This queries the movies table and retrieves the total record count.

Finally, we print the count to confirm the data was successfully loaded.

Visualization and ETL Pipeline

Once data is loaded into the database, visualizing it helps uncover patterns and trends. We create four key visualizations:

1. **Ratings Distribution:**
   - A histogram is used to show how movie ratings are distributed.
   - Helps identify whether most movies have high or low ratings, and if the distribution is skewed.
2. **Movies by Decade:**
   - A bar chart shows how many movies were produced in each decade.
   - Helps analyze how the film industry has grown or declined over time.

3. **Rating vs. Duration:**
    - A scatter plot is used to observe the relationship between movie ratings and duration.
    - Helps determine whether longer movies tend to have higher ratings.
4. **Average Movie Duration by Decade:**
    - A line chart tracks how the average movie length has changed over decades.
    - Useful for analyzing trends in storytelling preferences over time.

These visualizations help summarize large amounts of data into **easy-to-understand insights**, aiding in decision-making and further analysis.

An **ETL (Extract, Transform, Load) pipeline** automates data collection, processing, and storage. Instead of manually handling raw data, we use a structured pipeline to ensure consistency, accuracy, and efficiency.

1. **Extract (E)**
    - Fetches movie data from an external source (e.g., web scraping).
    - Automates data collection, reducing manual effort.
2. **Transform (T)**
    - Cleans and processes the data (e.g., removing duplicates, categorizing movies by decade).
    - Ensures data is structured and meaningful for analysis.
3. **Load (L)**
    - Stores the transformed data in a **SQLite database**.
    - Creates summary tables for easy access and saves results in CSV files for external use.

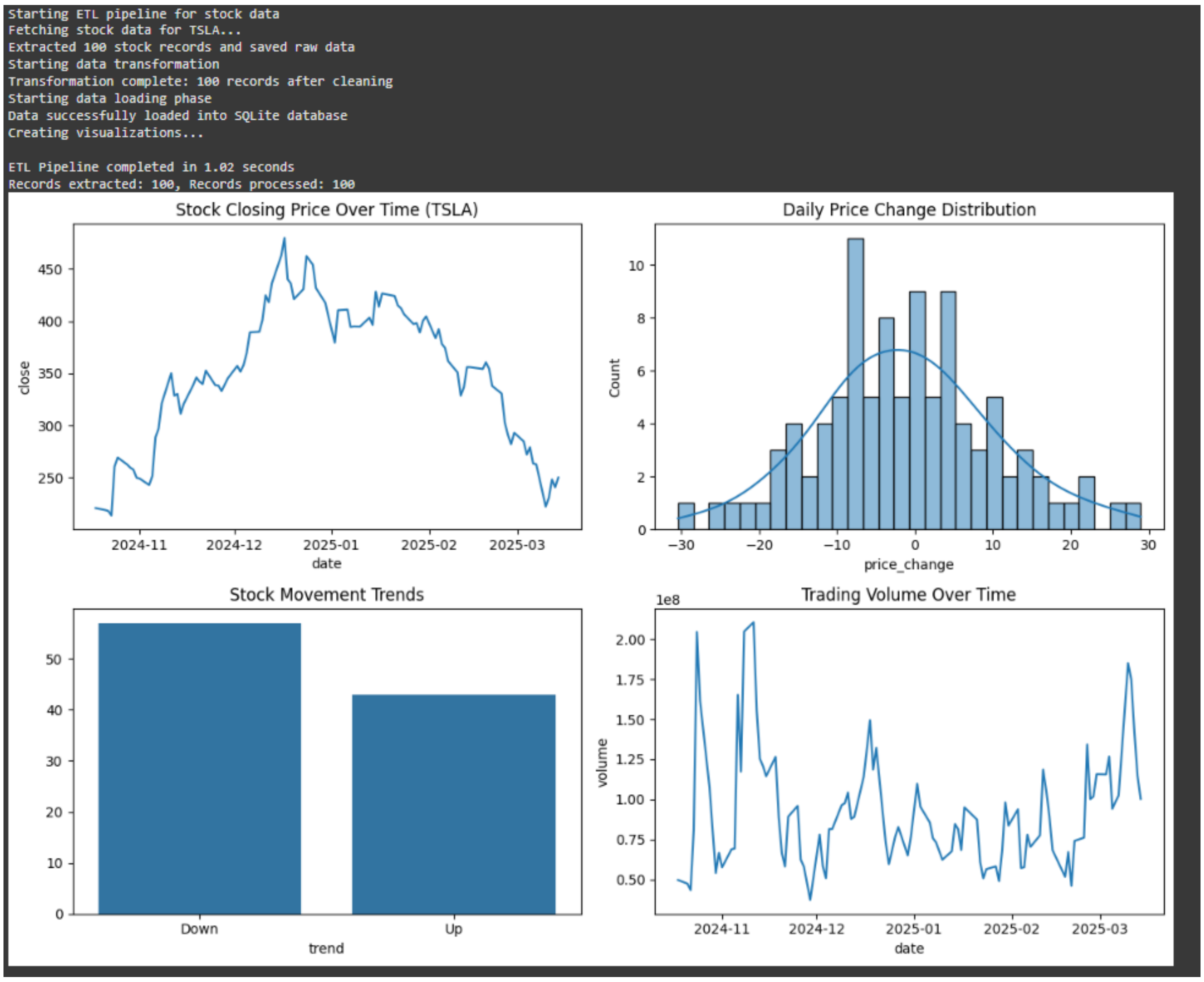
This pipeline ensures that **data is collected, structured, and stored automatically**, making it easier to analyze trends over time without manual intervention.

Students' task:

API - https://www.alphavantage.co/

```python
import requests
import pandas as pd
import sqlite3
from sqlalchemy import create_engine
import matplotlib.pyplot as plt
import seaborn as sns
import time
from datetime import datetime
import os

# Create directories for storing data
os.makedirs('data/raw', exist_ok=True)
os.makedirs('data/processed', exist_ok=True)

# Alpha Vantage API Key (Replace with your own key)
API_KEY = "8UVZD5SZ22L2QE3G"
BASE_URL = "https://www.alphavantage.co/query"

# EXTRACT PHASE
def extract_stock_data(symbol="TSLA"):
    """
    Extracts stock price data from Alpha Vantage API.
    """
    print(f"Fetching stock data for {symbol}...")

    url =
f"{BASE_URL}?function=TIME_SERIES_DAILY&symbol={symbol}&apikey={API_KEY
}"
    response = requests.get(url)

    if response.status_code == 200:
        data = response.json()

        if "Time Series (Daily)" not in data:
            print("Error fetching stock data:", data)
            return pd.DataFrame()

        stock_list = []
        for date, values in data["Time Series (Daily)"].items():
            stock_list.append({
                'date': date,
                'open': float(values['1. open']),
                'high': float(values['2. high']),
                'low': float(values['3. low']),
                'close': float(values['4. close']),
                'volume': int(values['5. volume']),
                'symbol': symbol,
                'timestamp': datetime.now().strftime('%Y-%m-%d
%H:%M:%S')
            })

        raw_df = pd.DataFrame(stock_list)
        raw_df.to_csv(f'data/raw/{symbol}_stock_raw.csv', index=False)
        print(f"Extracted {len(raw_df)} stock records and saved raw
data")

        return raw_df
    else:
        print(f"Failed to fetch stock data: {response.status_code}")
        return pd.DataFrame()

# TRANSFORM PHASE
def transform_stock_data(raw_df):
    """
    Clean and transform the raw stock data.
    """
    print("Starting data transformation")

    df = raw_df.copy()

    # Convert date to datetime format
    df['date'] = pd.to_datetime(df['date'])

    # Create a 'year' column
    df['year'] = df['date'].dt.year

    # Calculate daily price change
    df['price_change'] = df['close'] - df['open']

    # Calculate percentage change
    df['percent_change'] = (df['price_change'] / df['open']) * 100

    # Categorize stock movements
    df['trend'] = df['price_change'].apply(lambda x: 'Up' if x > 0 else
('Down' if x < 0 else 'No Change'))
```

```
    # Save transformed data
    df.to_csv(f'data/processed/{df["symbol"].iloc[0]}_stock_transformed
.csv', index=False)
    print(f"Transformation complete: {len(df)} records after cleaning")

    return df

# LOAD PHASE
def load_stock_data(df):
    """
    Load the transformed data into a SQLite database.
    """
    print("Starting data loading phase")

    engine =
create_engine('sqlite:///data/processed/stock_database.sqlite')

    # Store main stock data
    df.to_sql('stocks', engine, if_exists='replace', index=False)

    # Aggregate data by year
    yearly_stats = df.groupby('year').agg({
        'symbol': 'count',
        'open': 'mean',
        'close': 'mean',
        'volume': 'sum'
    }).reset_index()
    yearly_stats.columns = ['year', 'record_count', 'avg_open',
'avg_close', 'total_volume']
    yearly_stats.to_sql('yearly_stats', engine, if_exists='replace',
index=False)

    # Export to CSV
    yearly_stats.to_csv('data/processed/yearly_stats.csv', index=False)

    print("Data successfully loaded into SQLite database")
    return True

# VISUALIZATION
def visualize_stock_data(df):
    """
    Create visualizations of the stock data.
    """
    print("Creating visualizations...")

    conn = sqlite3.connect('data/processed/stock_database.sqlite')

    plt.figure(figsize=(12, 8))

    # Price Trend Over Time
    plt.subplot(2, 2, 1)
    sns.lineplot(x='date', y='close', data=df)
    plt.title(f'Stock Closing Price Over Time
({df["symbol"].iloc[0]})')

    # Daily Price Change Distribution
    plt.subplot(2, 2, 2)
    sns.histplot(df['price_change'], bins=30, kde=True)
    plt.title('Daily Price Change Distribution')

    # Stock Trend Breakdown (Up vs. Down Days)
    plt.subplot(2, 2, 3)
    trend_counts = df['trend'].value_counts()
    sns.barplot(x=trend_counts.index, y=trend_counts.values)
    plt.title('Stock Movement Trends')

    # Volume Traded Over Time
    plt.subplot(2, 2, 4)
    sns.lineplot(x='date', y='volume', data=df)
    plt.title('Trading Volume Over Time')

    plt.tight_layout()
    plt.savefig(f'data/processed/{df["symbol"].iloc[0]}_stock_visualiza
tion.png')

    conn.close()

# Run the full ETL pipeline
def run_stock_etl_pipeline(symbol="TSLA"):
    """
    Execute the full ETL pipeline for stock data.
    """
    start_time = time.time()
    print("Starting ETL pipeline for stock data")

    # Extract
    raw_data = extract_stock_data(symbol)

    # Transform
    transformed_data = transform_stock_data(raw_data)

    # Load
    load_success = load_stock_data(transformed_data)

    # Visualize
    visualize_stock_data(transformed_data)

    total_time = time.time() - start_time
    print(f"\nETL Pipeline completed in {total_time:.2f} seconds")
    print(f"Records extracted: {len(raw_data)}, Records processed:
{len(transformed_data)}")

    return transformed_data

# Execute for a specific stock symbol
stock_data = run_stock_etl_pipeline(symbol="TSLA")
```



```
Starting ETL pipeline for stock data
Fetching stock data for TSLA...
Extracted 100 stock records and saved raw data
Starting data transformation
Transformation complete: 100 records after cleaning
Starting data loading phase
Data successfully loaded into SQLite database
Creating visualizations...

ETL Pipeline completed in 1.02 seconds
Records extracted: 100, Records processed: 100
```

Conclusion:
I have implemented an ETL (Extract, Transform, Load) pipeline using Python to analyze stock market data from the Alpha Vantage API. We extracted daily stock prices, including open, high, low, close prices, and trading volume, automating data collection through API requests. The transformation phase involved cleaning and structuring the data, converting dates, calculating price changes, and categorizing trends as Up, Down, or No Change for better insights. The processed data was then stored in a SQLite database, enabling efficient storage and retrieval. We also computed yearly stock statistics and visualized trends through price fluctuations, trading volume, and daily changes. This ETL pipeline demonstrated how data engineering enhances financial analysis by automating data extraction, transforming raw stock data into insights, and enabling structured storage. It can be extended to track multiple stocks, integrate machine learning models, or optimize real-time updates for better market predictions.

Post-lab questions:

1. **What were the main challenges you faced while extracting and cleaning data from your chosen source? How did you handle missing or inconsistent data?**

One of the main challenges faced while extracting data from the Alpha Vantage API was handling API limitations such as rate limits and occasional missing data points. Since the free-tier API allows a limited number of requests per minute, we had to implement delays between requests to prevent exceeding the quota. Additionally, some stock records had incomplete data due to market closures or missing trading days. To handle missing or inconsistent data, we used data cleaning techniques such as:

- Dropping empty or null values for essential fields like stock prices.
- Filling missing values using forward fill (copying the last known value) to maintain continuity.
- Ensuring data types were consistent, converting string-based numerical values to float or integer for calculations.
- Sorting data by date to ensure a correct chronological order for trend analysis.

By applying these transformations, we ensured that the dataset remained structured, reliable, and ready for further analysis.

2. **How does the data from your chosen source compare with the movie dataset structure-wise?**

The stock market dataset from Alpha Vantage API differs significantly from the movie dataset in terms of structure and characteristics. Stock data is time-series-based, providing daily price movements with fields like open, high, low, close prices, and trading volume, whereas movie data is more static and categorical, containing details like title, release year, ratings, and vote counts. Unlike movies, which have fixed attributes, stock prices continuously fluctuate, making them highly time-sensitive and suitable for trend analysis. Additionally, stock data is predominantly numerical, requiring transformations like price change calculations, percentage variations, and trend classifications. In contrast, movie data consists of both categorical and numerical values, making it more suited for ranking and classification tasks, such as categorizing movies by decade, rating, and popularity. The processing approaches also differ; stock data demands time-based transformations, while movie data requires categorization and text processing. Overall, stock data is dynamic and predictive, while movie data is more static and analytical, requiring different transformation techniques based on their respective use cases.

3. **If you had to process a much larger dataset (millions of records), what optimizations would you implement in your ETL pipeline? How would you ensure efficient storage and retrieval of data?**

If processing a large-scale dataset with millions of records, several optimizations would be required to ensure efficient ETL (Extract, Transform, Load) pipeline performance and data storage/retrieval. To optimize the extraction phase, we would use batch processing and asynchronous API requests to handle multiple data pulls simultaneously, reducing wait times. Additionally, implementing incremental extraction—only fetching new or updated records instead of the entire dataset—would minimize redundant data retrieval. During the transformation phase, we would use vectorized operations in Pandas (or switch to PySpark for big data processing) instead of loops to improve computation speed. Data cleaning and aggregation tasks would be optimized using parallel processing and chunk-based transformations to handle large volumes efficiently. For the loading phase, we would store data in a high-performance database like PostgreSQL, MySQL, or a columnar storage system like Apache Parquet instead of SQLite, which is more suitable for small datasets. Indexing and partitioning the data by date or categories would enhance retrieval speed, while caching frequently accessed data would further reduce query time. To ensure efficient data retrieval, we would implement optimized SQL queries, indexing strategies, and database normalization to avoid unnecessary duplication. If real-time analysis is needed, we could integrate a distributed processing system like Apache Kafka or Spark Streaming for handling continuous data inflow. By implementing these optimizations, the ETL pipeline would efficiently process large datasets while maintaining speed, accuracy, and scalability.