User Manual for ETL Pipeline

1. Overview

This ETL pipeline is designed to automate the process of **extracting, transforming, and loading (ETL)** social media analytics data into a **PostgreSQL data warehouse**. It enables efficient analysis and reporting by organizing raw data from various sources into well-structured data marts, following the principles of a **star schema**. The pipeline leverages **PySpark** for large-scale data processing, **Apache Airflow** for orchestrating ETL workflows, and **dbt** for managing transformations in the data warehouse.

The purpose of this pipeline is to:

- **Extract** raw social media interaction data from JSON files.
- **Transform** the data into dimensions and fact tables that provide insights into user engagement, content performance, tag analysis, and location analysis.
- **Load** the cleaned and transformed data into a **PostgreSQL data warehouse** to power analytics dashboards and reports.

Scope of the Pipeline

This pipeline addresses the data processing needs for the **social media analytics platform** by automating the entire ETL process. The pipeline supports:

- Handling and transforming high-volume social media data.
- Creating data marts to analyze user activity, content trends, geographical distribution, and tag usage.
- Facilitating downstream analysis for business intelligence applications.

Key components and technologies:

- **PySpark**: Used for the heavy lifting of data extraction and transformation.
- **Apache Airflow**: Orchestrates the entire ETL workflow by managing task dependencies and scheduling.
- **dbt (Data Build Tool)**: Executes SQL transformations and manages the schema within the PostgreSQL data warehouse.
- **PostgreSQL**: Serves as the data warehouse where transformed data is stored and queried for reporting.

2. System Requirements

Hardware: Minimum Requirements

To run the ETL pipeline, the following hardware specifications are recommended:

- **CPU**: 6 cores
- **RAM**: 16 GB
- **Storage**: 10 GB free disk space for the pipeline's working files, in addition to any data storage required (e.g., your data size of ~300 MB).

Software:

The ETL pipeline leverages a Dockerized environment containing multiple components. Below are the software requirements:

1. **Python**: Version 3.11.7 (as specified in your environment) for the PySpark jobs and dbt.
2. **PySpark**: Used for large-scale data processing, integrated with PostgreSQL.
   - Installed within the PySpark container. The Jupyter notebook interface is available on port 8888, and the PySpark job UI can be accessed on port 4040.
3. **Docker**: To manage containerized environments for all services.
   - Minimum version: 20.10+ (or latest stable version)
   - Compose version: 2.4+ (or compatible with your Docker installation)
   - Instructions to install Docker can be found here.
4. **PostgreSQL**: A PostgreSQL instance runs inside a Docker container as the data warehouse.
   - Container: `postgres_warehouse`
   - Credentials are available in the Docker Compose file, and the warehouse stores all transformed data.
5. **Apache Airflow**: Orchestrates the pipeline tasks.
   - UI available on port 8088.
   - Credentials: username: `airflow`, password: `airflow`.
6. **dbt (Data Build Tool)**: For managing SQL transformations within the data warehouse.
   - The dbt container is pre-configured to connect to PostgreSQL, and you can write transformations under the `dbt_project` directory.
   - Version: 1.8.6.
7. **Git**: Version control for tracking changes to code and configurations.
   - Instructions to install Git can be found here.

3. Installation and Setup

This section provides detailed instructions for setting up Docker containers, configuring the Docker Compose environment, and configuring PySpark, dbt, and Airflow, along with any network configurations and environment variables.

1. Setting up Docker Containers

Pre-requisites:

- Install **Docker** and **Docker Compose** if not already installed.
  - Docker installation: Docker Get Started Guide.
  - Docker Compose installation: Compose Installation.

Steps:

1. **Clone the repository**: If your code is in a repository, clone it:

   ```
   bash
   Copy code
   git clone <repository-url>
   cd <repository-directory>
   ```

2. **Start Docker Containers**: After navigating to the project directory containing the `docker-compose.yaml` file, run the following commands:

   ```
   bash
   Copy code
   chmod u+x ./init.sh
   ./init.sh
   ```

   This script will pull and set up the Docker environment, creating all necessary containers, including:

   - **Airflow** (Port 8088)
   - **PySpark** (Notebook on Port 8888, UI for jobs 4040)
   - **dbt** container for managing SQL transformations
   - **PostgreSQL** data warehouse

   The total environment size is approximately **5-6 GB**. The first run will require an internet connection.

3. **Access PySpark**: To access the Jupyter notebook interface for PySpark, run the following command:

   ```
   bash
   Copy code
   docker logs --tail 20 pyspark
   ```

   This will display a URL to access the Jupyter server, which will look something like:

   ```
   arduino
   Copy code
   http://127.0.0.1:8888/lab?token=<token>
   ```

4. **Access Airflow**: Navigate to `http://localhost:8088` in your browser. The default credentials are:
   - **Username**: airflow
   - **Password**: airflow
5. **Access dbt**: You can write dbt transformations under the directory `dbt_project` on your host machine, which is mounted into the dbt container. You can also access the dbt container by running:

   ```
   bash
   Copy code
   docker exec -it dbt bash
   ```

2. Configuring the Docker Compose Environment

The `docker-compose.yaml` file outlines the configuration for all containers, including network and volume mounts. Key components:

- **PySpark**:
  - Volumes: The `spark_code` directory is mapped to the container where you write your PySpark code.
  - Networking: Ensure that all containers are on the same network as defined in the Compose file for smooth communication.
- **PostgreSQL**: Credentials are defined in the Compose file, and the volume mounts ensure data persistence.
- **dbt**: The dbt container uses mounted volumes for the `dbt_project` directory, allowing you to manage transformations easily.

3. Detailed Setup for PySpark, dbt, and Airflow

PySpark:

- The `spark_code` directory on your host machine is mapped to `/app/spark_code` inside the container. All your PySpark scripts should reside in this directory.
- A **PostgreSQL JDBC driver** is included to enable PySpark to communicate with the PostgreSQL data warehouse.

dbt:

- Installed dbt core and PostgreSQL connector versions (from `installed-packages.txt`):

  ```
  bash
  Copy code
  dbt-core==1.8.6
  dbt-postgres==1.8.2
  ```

  This ensures compatibility with PostgreSQL as the target database.

- The dbt project directory is mounted to `/app/dbt_project` in the container. Run any dbt commands from inside this container or through your Docker terminal.

Airflow:

- Airflow is responsible for orchestrating your ETL tasks. To run it:

  ```
  bash
  Copy code
  docker exec -it airflow bash
  airflow scheduler
  ```

  You can configure tasks in the `dags/` folder, and monitor the workflow in the Airflow UI at `http://localhost:8088`.

4. Network Configuration

- All services are on the same network as defined in the `docker-compose.yaml` file. Ensure this network is accessible between containers.
- Airflow tasks can communicate with PySpark and dbt containers through the network, ensuring that tasks execute without issues.

5. Environment Variables

- The Docker Compose file contains the necessary environment variables for PostgreSQL, including `POSTGRES_USER`, `POSTGRES_PASSWORD`, and `POSTGRES_DB`. Ensure that these are consistent with your database setup.
- For PySpark and dbt, any additional configuration can be managed via environment variables in the `Dockerfile` or `docker-compose.yaml` files.

4. Operating the Pipeline

This section provides step-by-step instructions for operating the ETL pipeline using **Docker**, **Airflow**, **PySpark**, and **dbt**.

1. Starting the Pipeline Using Docker and Airflow

To start the ETL pipeline, follow these steps:

1. **Start Docker Containers**: Ensure that your Docker containers are up and running by navigating to the directory containing the `docker-compose.yaml` file and running:

   ```
   bash
   Copy code
   docker-compose up -d
   ```

   This command starts all necessary services: **Airflow**, **PySpark**, **PostgreSQL**, and **dbt**.

2. **Access Airflow**: After the containers are up, open Airflow by navigating to `http://localhost:8088` in your browser. Use the following credentials:
   - **Username**: `airflow`
   - **Password**: `airflow`
3. **Trigger the ETL Pipeline DAG**: Once inside the Airflow UI, you can manually trigger the ETL pipeline DAG:
   - Locate the DAG named `etl_pipeline_docker`.
   - Click the "Trigger DAG" button to start the ETL process.

2. Detailed Usage Instructions for Each Component

A. PySpark Scripts

The ETL process is divided into several tasks managed by Airflow. Each task corresponds to a different stage of the pipeline, which runs a PySpark script inside the **PySpark** container.

1. **Extract Data**:
   - This task extracts data using the `Write_CSV.py` PySpark script.
   - Airflow executes the script inside the PySpark container using the following command:

     ```
     bash
     Copy code
     docker exec pyspark spark-submit /home/jovyan/work/Write_CSV.py
     ```

2. **Clean Data**:
    - The `Cleaning.py` PySpark script is responsible for cleaning and transforming the extracted data.
    - This is executed by Airflow using the following command:

    ```
    bash
    Copy code
    docker exec pyspark spark-submit /home/jovyan/work/Cleaning.py
    ```

3. **Logging**:
    - The `Logging.py` script logs important steps during the ETL process.
    - This is executed via:

    ```
    bash
    Copy code
    docker exec pyspark spark-submit /home/jovyan/work/Logging.py
    ```

4. **Connect and Write to DWH**:
    - This task connects to the **PostgreSQL** data warehouse and loads the cleaned data using the `Connect_Dwh.py` script.
    - Airflow executes this with the following command:

    ```
    bash
    Copy code
    docker exec pyspark spark-submit \
    --master local[*] \
    --name "Load Data to PostgreSQL" \
    --driver-class-path /home/jovyan/drivers/postgresql-42.7.3.jar \
    --jars /home/jovyan/drivers/postgresql-42.7.3.jar \
    --driver-memory 8g \
    --executor-memory 8g \
    --conf "spark.sql.shuffle.partitions=100" \
    /home/jovyan/work/Connect_Dwh.py
    ```

B. dbt Models

Once the data is loaded into the PostgreSQL data warehouse, **dbt** is used to manage and run SQL transformations to create data marts.

1. **Run dbt Models**:
    - The `dbt_run` task executes all dbt models defined in your dbt project to transform the data.
    - This is run using:

    ```
    bash
    Copy code
    docker exec dbt dbt run --profiles-dir /root/.dbt --project-dir
    /app
    ```

2. **Run dbt Tests**:
   - After running the dbt models, the `dbt_test` task is used to validate the data.
   - This is executed via:

     ```
     bash
     Copy code
     docker exec dbt dbt test --profiles-dir /root/.dbt --project-dir
     /app
     ```

3. Task Dependencies

In your Airflow DAG (as shown in `ETL_Pipeline.py`), the tasks are structured with dependencies as follows:

- **Extract Data → Clean Data → Logging → Connect and Write to DWH → Run dbt Models → Run dbt Tests**

5. Troubleshooting

This section outlines common issues and troubleshooting steps for each component of the ETL pipeline, including **Docker**, **Airflow**, **PySpark**, **PostgreSQL**, and **dbt**.

A. Docker Issues

1. **Docker Containers Won't Start**:
   - **Error**: Containers fail to start or exit immediately.
   - **Solution**:
     - Check if Docker is running with `docker ps`.
     - Run `docker-compose logs` to view logs and identify the root cause of container failure.
     - Ensure no conflicting services are running on the same ports (Airflow uses 8088, PySpark uses 8888 and 4040).
2. **Cannot Access Jupyter Notebook or Airflow UI**:
   - **Error**: Unable to access Airflow at `localhost:8088` or Jupyter Notebook at `localhost:8888`.
   - **Solution**:
     - Ensure the containers are running with `docker-compose ps`.
     - Check for firewall or security settings that may block access.
     - Run `docker logs <container-name>` (e.g., `docker logs pyspark`) to check for specific errors.

B. Airflow Issues

1. **Airflow DAG Not Running**:
   - **Error**: The DAG does not start when triggered manually.
   - **Solution**:
     - Ensure the Airflow scheduler is running by checking `docker-compose logs airflow-scheduler`.
     - Check Airflow logs in the UI for error messages related to task dependencies or task execution failures.
2. **Tasks Failing in Airflow**:
   - **Error**: One or more tasks fail, halting the DAG execution.
   - **Solution**:
     - Check task-specific logs in Airflow UI to identify the issue.
     - Ensure the PySpark scripts and dbt configurations are correct and accessible by the corresponding containers.
     - Use the "Retry" option for failed tasks after resolving the issue.

C. PySpark Issues

1. **PySpark Scripts Failing**:
   - **Error**: PySpark tasks (e.g., `Write_CSV.py`, `Cleaning.py`) fail to execute.
   - **Solution**:
     - Check logs for specific errors using `docker logs pyspark`.
     - Ensure the JDBC driver (`postgresql-42.7.3.jar`) is correctly mounted in the PySpark container.
     - Verify that the scripts in the `spark_code` directory are correct and executable.
2. **Memory or Performance Issues**:
   - **Error**: Out of memory or slow execution during data processing.
   - **Solution**:
     - Adjust the memory and executor settings in the Airflow DAG (for example, increase driver and executor memory in `connect_write_dwh` task).
     - Reduce the number of shuffle partitions by adjusting the `spark.sql.shuffle.partitions` setting.

D. PostgreSQL Issues

1. **Connection Issues**:
   - **Error**: PySpark cannot connect to PostgreSQL, or dbt throws connection errors.
   - **Solution**:
     - Ensure the PostgreSQL container is running: `docker-compose ps postgres_warehouse`.
     - Verify the credentials in `docker-compose.yaml` and Airflow environment variables.
     - Confirm that the PostgreSQL JDBC driver is correctly configured in the PySpark scripts.
2. **Data Load Issues**:
   - **Error**: Data does not load into the PostgreSQL database.
   - **Solution**:
     - Check the PySpark logs for connection or query execution errors.
     - Verify that the `Connect_Dwh.py` script has the correct SQL logic and is properly handling data insertion.

E. dbt Issues

1. **dbt Run Failing**:
   - **Error**: The `dbt_run` task fails in Airflow.
   - **Solution**:
     - Check the logs in Airflow for specific errors during the `dbt run` command.
     - Ensure that the `profiles.yml` and `dbt_project.yml` files are correctly configured.

    - Validate that the dbt container has access to the PostgreSQL database.

2. **dbt Tests Failing**:
    - **Error**: One or more dbt tests fail, indicating data quality issues.
    - **Solution**:
        - Review the test logs in the Airflow UI or dbt container.
        - Investigate whether the source data conforms to the expected schema and formats.
        - Fix any data quality issues identified by the tests and rerun the pipeline.

General Troubleshooting Tips:

- **Check Logs**: The first step in troubleshooting is checking logs for the specific container or service using:

  ```
  bash
  Copy code
  docker logs <container-name>
  ```

- **Verify Network Settings**: Ensure that all containers are running on the same network as configured in `docker-compose.yaml`.
- **Restart Containers**: If you encounter persistent issues, restart the containers:

  ```
  bash
  Copy code
  docker-compose down && docker-compose up -d
  ```

- **Update Docker Images**: Ensure you are using the latest images by running:

  ```
  bash
  Copy code
  docker-compose pull
  ```

Support Contact Details

For further assistance with the ETL pipeline, you can reach out to the following contacts:

- **Support Email**: [a.mahmoud1803@gmail.com]
- **GitHub Repository**: For detailed code-related issues, please create an issue in the private

Technical Specifications

1. Architecture

High-Level Architecture Overview

The ETL pipeline is built on a **containerized microservices architecture** using Docker, where each component runs in its own container. The following components work together to form the pipeline:

- **PySpark**: Handles data extraction, cleaning, transformation, and loading (ETL) tasks. It is responsible for ingesting raw data from the source (e.g., JSON files), processing the data, and loading it into the PostgreSQL data warehouse.
- **PostgreSQL**: Acts as the data warehouse. This is where cleaned and transformed data is stored in relational tables following a star schema, supporting analytical queries.
- **dbt (Data Build Tool)**: Manages data transformations inside PostgreSQL. Once data is loaded into the warehouse, dbt is responsible for running SQL transformations that prepare the data for analysis, such as creating fact and dimension tables.
- **Apache Airflow**: Orchestrates the entire ETL pipeline, automating the workflow by scheduling and managing dependencies between different stages of the pipeline. Airflow DAGs execute each stage, from data extraction to dbt transformations.

Component Interactions

Each component is containerized using Docker, and they interact as follows:

1. **PySpark Container**:
   - Extracts raw data (e.g., from JSON files) and applies transformation logic.
   - It then connects to the PostgreSQL container using the JDBC driver to load the transformed data into specific tables.
2. **PostgreSQL Container**:
   - Stores raw and transformed data in a relational format.
   - Receives data from PySpark and provides it to dbt for further processing.
3. **dbt Container**:
   - Connects to the PostgreSQL data warehouse to run SQL transformations that prepare the data for analytics.
   - Executes models defined in dbt, which materializes data marts (e.g., fact and dimension tables) in PostgreSQL.
4. **Airflow Container**:
   - Schedules and orchestrates the pipeline by triggering tasks in PySpark and dbt containers.
   - Manages dependencies between tasks, ensuring they run in the correct order.

Data Flow

- **Data Extraction**:
  - PySpark reads raw data from a specified source (e.g., JSON files) located in the `spark_code` directory.
- **Data Cleaning and Transformation**:
  - PySpark applies cleaning and transformation logic to the raw data. This may include deduplication, handling missing values, and transforming data formats.
- **Data Loading**:
  - After cleaning, PySpark loads the transformed data into the PostgreSQL data warehouse using JDBC.
- **Data Modeling and Aggregation**:
  - dbt is triggered by Airflow to run transformations. It takes the data from PostgreSQL and models it according to the desired structure, generating fact and dimension tables.
- **Orchestration**:
  - Airflow manages the entire ETL workflow, ensuring that each task (from extraction to transformation and loading) happens sequentially and without failure.

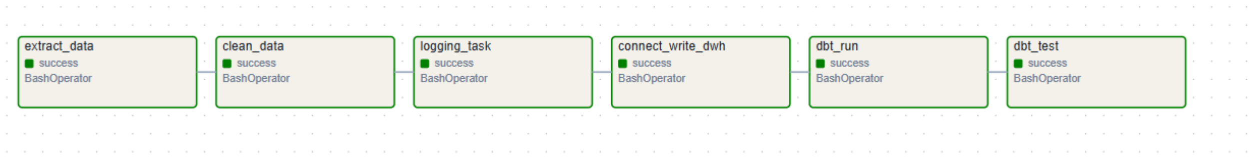
---

2. Data Flow Diagrams

A. Data Flow Overview

Below is a simplified **data flow diagram** showing how data moves through the pipeline:

1. **Raw Data Source** → (PySpark Extracts) →
2. **Raw Data in PySpark** → (PySpark Transforms) →
3. **Transformed Data in PySpark** → (PySpark Loads) →
4. **PostgreSQL Data Warehouse** → (dbt Runs Transformations) →
5. **Data Marts in PostgreSQL** (Fact/Dimension Tables).



This flow is managed and automated by **Apache Airflow**, ensuring that the stages happen in a predetermined sequence, with each task dependent on the success of the previous one.

B. Detailed Component Interaction Diagram

1. **Airflow** DAG (Managed Workflow) → Triggers the following steps:
    - **PySpark Task 1 (Extract)** → Reads raw data from source.
    - **PySpark Task 2 (Transform)** → Cleans and transforms the data.
    - **PySpark Task 3 (Load)** → Writes the cleaned data into the PostgreSQL database.
    - **dbt Task 1 (Run Models)** → Transforms raw tables into analytics-ready tables (fact/dimension).
    - **dbt Task 2 (Test)** → Validates the data after the transformations.

C. Data Transformation Steps

- **Raw Data Extraction (PySpark)**:
    - Extracts data from JSON files or other external sources, processes it (e.g., cleaning, aggregation).
- **Data Transformation and Loading (PySpark)**:
    - Cleans the data (e.g., removing duplicates, filling missing values), applies business logic, and loads the data into PostgreSQL.
- **Data Modeling (dbt)**:
    - Creates **fact tables** (e.g., interactions, posts) and **dimension tables** (e.g., users, tags) based on business rules and the star schema.
- **Data Aggregation (dbt)**:
    - Further aggregates data to prepare data marts for analytics, such as summarizing user engagement or content performance.

2. Detailed Component Operation

A. PySpark: Data Extraction, Transformation, and Loading

PySpark is the engine responsible for handling large-scale data processing tasks in the ETL pipeline. Each script is assigned a specific task: extracting, transforming, and loading data into the PostgreSQL data warehouse.

1. Data Extraction: `Write_CSV.py`

- **Function**: This script extracts raw data from an external source (e.g., JSON files or CSV files) and prepares it for the transformation step.
- **Key Operations**:
  - Reads data from the local file system or cloud storage using PySpark's `spark.read()` method (e.g., `spark.read.json()` or `spark.read.csv()`).
  - Loads the raw data into a DataFrame for further processing.
- **Example**:

```
python
Copy code
df = spark.read.json("/path/to/json/file")
df.show()
```

2. Data Cleaning and Transformation: `Cleaning.py`

- **Function**: This script is responsible for cleaning and transforming the extracted data. It performs tasks like filtering, handling missing values, deduplication, and applying business rules.
- **Key Operations**:
  - **Filtering**: Removes invalid records or data that doesn't meet certain criteria.
  - **Handling Missing Values**: Fills or drops rows with null values depending on business requirements.
  - **Transformations**: Applies necessary transformations like converting data types or generating calculated columns.
- **Example**:

```
python
Copy code
clean_df = df.filter(df['column'].isNotNull())
clean_df = clean_df.withColumn('new_column',
clean_df['existing_column'] * 2)
```

3. Data Loading to PostgreSQL: `Connect_Dwh.py`

- **Function**: This script connects to the PostgreSQL data warehouse and loads the cleaned and transformed data from PySpark into relational tables.
- **Key Operations**:
  - **Connection Setup**: Uses the PostgreSQL JDBC driver to establish a connection with the database.
  - **Data Write**: Writes the DataFrame into PostgreSQL tables using `df.write()` with JDBC options.
- **Example**:

```
python
Copy code
df.write \
  .format("jdbc") \
  .option("url", "jdbc:postgresql://postgres_warehouse:5432/warehouse")
\
  .option("dbtable", "public.fact_table") \
  .option("user", "warehouse") \
  .option("password", "warehouse") \
  .save()
```

4. Logging: `Logging.py`

- **Function**: This script logs the data processing steps and their outcomes (e.g., success or failure, number of rows processed, etc.). It ensures that the pipeline execution can be monitored.
- **Key Operations**:
  - Logging messages to keep track of data processing activities.
- **Example**:

```
python
Copy code
print(f"Processed {df.count()} records in DataFrame.")
```

B. Airflow: DAG Configuration and Scheduling

Apache Airflow is used to orchestrate the ETL pipeline, managing the execution and sequencing of tasks like data extraction, transformation, and loading. The **Directed Acyclic Graph (DAG)** defines the workflow and manages the dependencies between tasks.

1. DAG Overview: `etl_pipeline_docker`

- The DAG controls the order in which the PySpark scripts and dbt commands are executed. It ensures that data extraction happens first, followed by cleaning, logging, loading into the warehouse, and finally running dbt models.

2. DAG Configuration

- **Default Arguments**: These define the behavior of the DAG, such as the start date, the owner of the tasks, and the retry policy.

  ```
  python
  Copy code
  default_args = {
      'owner': 'airflow',
      'start_date': days_ago(1),
      'retries': 1,
      'retry_delay': timedelta(minutes=5),
  }
  ```

- **DAG Definition**: The DAG is created with a specific `dag_id` and scheduled manually (using `schedule_interval=None`). Tasks are executed sequentially.

  ```
  python
  Copy code
  with DAG(
      dag_id='etl_pipeline_docker',
      default_args=default_args,
      schedule_interval=None,
      catchup=False,
  ) as dag:
  ```

3. Task Configuration

Each task in the DAG is defined using the `BashOperator` to run PySpark scripts or dbt commands. Dependencies are established to enforce task execution order.

- **Extract Data Task**:

  ```
  python
  Copy code
  extract_data = BashOperator(
      task_id='extract_data',
      bash_command='docker exec pyspark spark-submit
  /home/jovyan/work/Write_CSV.py',
  )
  ```

- **Transform Data Task**:

  ```
  python
  Copy code
  clean_data = BashOperator(
      task_id='clean_data',
      bash_command='docker exec pyspark spark-submit
  /home/jovyan/work/Cleaning.py',
  )
  ```

- **Load Data to DWH Task**:

  ```
  python
  Copy code
  connect_write_dwh = BashOperator(
      task_id='connect_write_dwh',
      bash_command='docker exec pyspark spark-submit
  /home/jovyan/work/Connect_Dwh.py',
  )
  ```

4. Task Dependencies

Tasks are linked sequentially to ensure that the entire pipeline executes in the correct order.

```
python
Copy code
extract_data >> clean_data >> logging_task >> connect_write_dwh >> dbt_run >>
dbt_test
```

C. dbt: Data Marts and Transformations

dbt (Data Build Tool) is used to handle the SQL transformations and materialize the final data marts within the PostgreSQL data warehouse. dbt transforms raw data into well-structured fact and dimension tables based on the star schema.

1. Data Marts Overview

In your dbt project, several data marts are created to support analytics for the social media platform:

- **User Engagement Mart**: Aggregates and analyzes user interactions (likes, shares, comments) across posts.
- **Content Performance Mart**: Measures the performance of different content by tracking engagements like reactions and shares.
- **Tag Analysis Mart**: Analyzes the use and impact of different tags on post engagement.
- **Location Analysis Mart**: Analyzes engagement and performance by geographical location.

2. dbt Transformations

dbt transformations are SQL-based and occur within the PostgreSQL warehouse. Each transformation models the data into fact and dimension tables.

- **Example of a dbt Model (`user_engagement_mart.sql`)**: This model aggregates user interactions to create a fact table that summarizes likes, shares, and comments by user.

  ```
  sql
  Copy code
  select
      user_id,
      count(distinct post_id) as post_count,
      sum(like_count) as total_likes,
      sum(share_count) as total_shares,
      sum(comment_count) as total_comments
  from
      raw_data.social_interactions
  group by
      user_id
  ```

3. dbt Materialization

dbt allows models to be materialized as either **tables** or **views**. The default materialization is set to `table`, which means the transformations result in persistent tables within the PostgreSQL database.

- **Model Materialization**:

  ```
  yaml
  Copy code
  models:
    dbt_project:
      marts:
        materialized: table
  ```

4. dbt Testing

After the dbt models are run, tests are executed to ensure data quality and integrity. dbt tests check for common issues like null values, unique constraints, or foreign key integrity.

- **Example of a dbt Test**:

  ```
  yaml
  Copy code
  tests:
    - unique:
        column_name: user_id
    - not_null:
        column_name: post_id
  ```

3. Security and Compliance

A. Security Measures

The ETL pipeline is designed with multiple layers of security to ensure that data is handled and processed securely, from extraction to transformation and loading into the PostgreSQL data warehouse. Below are the key security measures implemented across different components of the pipeline:

1. Docker Security

- **Isolated Containers**: Each component of the ETL pipeline (e.g., PySpark, dbt, PostgreSQL) runs in its own isolated Docker container. This isolation minimizes the risk of cross-container vulnerabilities and ensures that any breach in one service does not affect others.
- **Network Segmentation**: Docker Compose defines a private network for all containers. Only the services within the network (e.g., PySpark, PostgreSQL) can communicate with each other, ensuring that the database and other services are not exposed to the public internet.
- **Environment Variable Management**: Sensitive credentials (e.g., PostgreSQL user, password) are stored in environment variables within the `docker-compose.yaml` file. These credentials are kept out of the codebase, minimizing the risk of accidental exposure.

2. Database Security

- **PostgreSQL Authentication**: PostgreSQL is configured with strict username and password authentication. The credentials are provided through environment variables and managed securely within the Docker Compose configuration. Access is restricted to the containers within the same Docker network.
- **JDBC Connection Encryption**: The connection between PySpark and PostgreSQL uses the JDBC driver, which can be configured to use SSL encryption for securing the data in transit between the ETL process and the data warehouse.
- **Role-Based Access Control (RBAC)**: PostgreSQL can be further secured by defining role-based access controls, allowing different users and services only the necessary permissions for the data they need to access.

3. Data Encryption

- **Data at Rest**: PostgreSQL can be configured to encrypt data at rest using tools like Transparent Data Encryption (TDE). This ensures that even if the database storage is compromised, the data remains protected.
- **Data in Transit**: PySpark's JDBC connection to PostgreSQL can be encrypted using SSL/TLS to protect data during transmission. This ensures that sensitive data is not exposed to man-in-the-middle (MITM) attacks.

4. Airflow Security

- **User Authentication**: Airflow UI is secured using username and password authentication. Only authorized users with valid credentials can access the Airflow UI to trigger, monitor, or alter the pipeline.
- **Role-Based Access in Airflow**: Airflow supports role-based access controls (RBAC), which can be configured to grant or restrict access to specific Airflow components or DAGs based on the user's role (e.g., admin, operator, viewer).

5. Data Masking and Anonymization

- **Data Masking**: Sensitive information, such as personally identifiable information (PII), can be masked during the data transformation process to protect privacy. Masking replaces sensitive data with obfuscated versions while retaining the original structure for analysis.
- **Anonymization**: For use cases involving sensitive data, anonymization techniques can be applied during data processing. This ensures that sensitive attributes (e.g., names, addresses) are either removed or anonymized to protect individual privacy while still allowing for meaningful analysis.

B. Compliance with Data Protection Regulations

The ETL pipeline design takes into account compliance with major data protection regulations, ensuring that the data handling and processing meet industry-standard requirements.

1. General Data Protection Regulation (GDPR) Compliance

If your system handles personal data of individuals in the European Union (EU), it must comply with GDPR requirements.

- **Data Minimization**: The pipeline ensures that only the required data is extracted, processed, and stored. Unnecessary data is either not collected or removed during the transformation process.
- **Right to Access**: The system can be configured to respond to data access requests by querying the PostgreSQL database for specific user records. PySpark and dbt transformations ensure that data is traceable and can be easily retrieved if requested.
- **Right to Erasure (Right to Be Forgotten)**: In compliance with GDPR's right to erasure, the pipeline can be configured to delete user data from both raw sources and processed datasets, ensuring complete removal of PII.
- **Data Retention Policies**: Data retention policies can be enforced through dbt models or PostgreSQL, automatically purging data after a specific retention period to ensure compliance with GDPR's data minimization requirements.

2. California Consumer Privacy Act (CCPA) Compliance

For handling data of California residents, compliance with CCPA is essential.

- **Right to Opt-Out**: If users choose to opt-out of having their data processed or shared, the pipeline can be adjusted to exclude such data during the extraction or transformation phase.
- **Data Access and Portability**: The pipeline ensures that all data processed through PySpark and stored in PostgreSQL can be accessed and exported in a machine-readable format upon request.

3. Health Insurance Portability and Accountability Act (HIPAA) Compliance

For healthcare data, HIPAA compliance requires special attention to the handling of sensitive health information (PHI).

- **Access Control**: PostgreSQL can be configured to restrict access to PHI based on user roles, ensuring that only authorized personnel can access or process sensitive health data.
- **Data Encryption**: Both at-rest and in-transit encryption measures are critical for securing PHI and ensuring HIPAA compliance. SSL/TLS is used to secure data transmission between PySpark and PostgreSQL, while encryption of data at rest can be enabled in the PostgreSQL configuration.
- **Audit Logs**: Airflow can be configured to keep detailed logs of who accesses the pipeline and performs tasks, ensuring traceability and accountability in case of audits.

4. ISO 27001 Compliance

ISO 27001 is an international standard for information security management. Ensuring compliance with this standard involves:

- **Risk Management**: Regular risk assessments should be performed to identify potential vulnerabilities in the pipeline, particularly with regard to data handling and processing.
- **Security Policies**: The system can implement access control and data handling policies that align with ISO 27001’s guidelines for protecting information assets.
- **Monitoring and Logging**: The entire ETL pipeline can be equipped with logging and monitoring mechanisms (via Airflow) to track security incidents and maintain an audit trail.

4. Performance and Scalability

Optimizing the performance and scalability of an ETL pipeline is crucial to ensure it can handle increasing data volumes while maintaining speed and efficiency. Below are the best practices followed to optimize the ETL pipeline, along with maintenance guidelines to keep it efficient.

A. Best Practices for Performance and Scalability

1. PySpark Optimizations

- **In-Memory Data Processing**: PySpark uses distributed in-memory data processing, which significantly speeds up transformations by reducing the need for I/O operations.
- **Efficient Shuffling**: In the `Connect_Dwh.py` script, the shuffle partitions are set to 100 (`spark.sql.shuffle.partitions=100`). This optimizes the shuffle process by balancing the number of partitions and preventing excessive shuffling, which can slow down transformations.
- **Partitioning**: Data is partitioned during transformations, allowing PySpark to distribute the workload across multiple cores. Given that you have **6 cores** and the total data size is about **300MB**, partitioning ensures that each core handles a reasonable amount of data in parallel, speeding up processing.
- **Memory Management**: By allocating **8 GB of memory** to both the driver and executor (`--driver-memory 8g --executor-memory 8g`), the ETL pipeline can handle more data in memory, reducing the overhead of disk I/O.
- **Caching DataFrames**: For iterative operations, caching intermediate DataFrames in memory ensures that they do not need to be recomputed, improving the overall performance of the pipeline.

2. dbt Optimizations

- **Model Materialization as Tables**: The default materialization in dbt is set to `table`. Materialized tables in PostgreSQL provide faster query execution than views because the data is stored physically, reducing the need for on-the-fly calculations when queries are run.
- **Incremental Models**: If dealing with larger datasets or real-time data, dbt can be configured to use **incremental models**. This allows only new or changed records to be processed, reducing the workload on the database and speeding up transformations.
- **Selective Refresh**: Instead of running all dbt models every time, Airflow can trigger dbt to only run selective transformations based on the data that has changed. This reduces unnecessary processing and improves runtime.

3. PostgreSQL Optimizations

- **Indexing**: Proper indexing on frequently queried columns, such as foreign keys and time-based fields (e.g., `user_id`, `post_id`), helps speed up query execution, especially in the dbt transformations that create data marts.
- **Vacuum and Analyze**: PostgreSQL's `VACUUM` and `ANALYZE` commands are used to optimize database performance. They help reclaim space and update statistics used by the query planner, which improves query efficiency over time.
- **Connection Pooling**: Use connection pooling to limit the number of open connections and reduce overhead from constantly opening and closing connections.

4. Airflow DAG Optimizations

- **Task Parallelism**: Tasks in the Airflow DAG are sequenced to run one after another, but independent tasks (e.g., dbt transformations) can be parallelized to reduce total execution time.
- **Task Dependencies**: The Airflow DAG is optimized to have clear dependencies between tasks, ensuring that tasks only start when the previous one is completed. This minimizes errors and makes the pipeline more robust.
- **Resource Allocation**: By allocating appropriate resources to Airflow tasks (e.g., CPU and memory), the system can efficiently execute tasks without bottlenecks or resource contention.

B. Maintenance Guidelines

To keep the pipeline efficient and up-to-date, regular maintenance is essential. Here are key maintenance guidelines:

1. Monitoring and Logging

- **Airflow Monitoring**: Regularly check Airflow logs and monitor the DAG execution times (e.g., your pipeline currently runs in about **3 minutes and 10 seconds**). If there is an unexpected increase in runtime, investigate task logs to identify bottlenecks.
- **Resource Usage Monitoring**: Use tools like `docker stats` to monitor the resource usage (CPU, memory) of the PySpark, dbt, and PostgreSQL containers. Adjust resource allocation if necessary.

2. Regular Updates

- **Software Updates**: Keep Docker, PySpark, dbt, Airflow, and PostgreSQL up-to-date with the latest stable versions to benefit from performance improvements and security patches.
- **Dependency Updates**: Periodically update your PySpark dependencies and dbt packages (e.g., `dbt-core` and `dbt-postgres`) to ensure compatibility with new features and improvements.

3. Performance Tuning

- **Partition Sizes**: Regularly check the data sizes being processed by PySpark and adjust partitioning accordingly. If data size grows, increase the number of partitions or optimize partition sizes to distribute the load evenly across cores.
- **PostgreSQL Query Optimization**: Analyze slow-running queries in PostgreSQL using `EXPLAIN` and optimize them by adding indexes or rewriting queries.
- **Garbage Collection**: Clean up any intermediate data stored in PostgreSQL that is no longer needed. Regularly removing old or redundant data helps maintain optimal database performance.

4. Scalability Planning

- **Scale Out**: If the data volume grows significantly, consider adding more resources (CPU, memory) to the PySpark and PostgreSQL containers. Docker Compose allows easy scaling of services.
- **Database Partitioning**: For very large datasets, PostgreSQL can be horizontally partitioned (sharding) to distribute data across multiple nodes, improving query performance.

5. Backup and Recovery

A well-defined backup and recovery process ensures that critical data is protected in the event of a system failure, corruption, or accidental data loss. This section outlines the procedures for data backup and recovery in your ETL pipeline environment.

A. Data Backup Procedures

1. PostgreSQL Database Backup

Since PostgreSQL serves as your data warehouse, ensuring regular backups of the database is essential to protect transformed and processed data.

Backup Strategy:

- **Full Database Backups**: A full backup of the PostgreSQL database should be scheduled periodically (e.g., daily or weekly) depending on the data size and criticality.
- **Incremental Backups**: For larger datasets or high-frequency pipelines, consider performing incremental backups to capture only the data changes between full backups, which reduces the amount of storage required.

Backup Tools:

- **pg_dump**: This tool is commonly used for creating backups in PostgreSQL. It can back up the entire database or selected tables.

Full Database Backup Example:

```
bash
Copy code
docker exec postgres_warehouse pg_dump -U warehouse -F c -b -v -f
/backup/warehouse_backup_$(date +%F).sql warehouse
```

- This command takes a full backup of the `warehouse` database and saves it in the `/backup/` directory with a timestamped filename.

Scheduled Backups:

- Use a **cron job** or configure an **Airflow DAG** to automate the backup process. For example, create an Airflow DAG that runs the `pg_dump` command periodically to back up the database.

2. PySpark Intermediate Data Backup

While PySpark processes the data in memory, intermediate results or cleaned data can also be backed up if necessary.

- **Dataframe Export**: You can export intermediate DataFrames to cloud storage (e.g., AWS S3, Google Cloud Storage) or local storage as CSV, Parquet, or JSON files to provide an additional layer of backup before loading into the PostgreSQL database.

Example:

```
python
Copy code
df.write.parquet("/path/to/backup/intermediate_data_$(date +%F).parquet")
```

3. dbt Model Backups

dbt primarily manages transformations within PostgreSQL. However, backing up the **dbt project** itself is important to ensure the SQL transformations, configuration files, and schema definitions are protected.

- **Backup dbt Project Files**: Periodically back up the `dbt_project` directory, which contains your transformation logic (`.sql` files) and configuration (`dbt_project.yml`, `profiles.yml`).
- **Version Control**: Keep your dbt project in a version control system (e.g., Git) to track changes and provide a recovery point in case of accidental changes or deletions.

Example:

```
bash
Copy code
tar -czvf dbt_project_backup_$(date +%F).tar.gz /path/to/dbt_project
```

B. Data Recovery Procedures

In case of system failures, data corruption, or accidental data loss, follow these recovery procedures:

1. PostgreSQL Database Recovery

Recovery Using pg_restore:

- If you have backed up the PostgreSQL database using `pg_dump`, you can restore it using the `pg_restore` command.
- In the event of database failure or corruption, use the latest full backup or incremental backup for recovery.

Full Database Restore Example:

```
bash
Copy code
docker exec postgres_warehouse pg_restore -U warehouse -d warehouse -v
/backup/warehouse_backup_<date>.sql
```

- This command restores the `warehouse` database from a specific backup file.

Partial Table Recovery:

- If only certain tables are affected, you can restore individual tables from the backup file:

```
bash
Copy code
pg_restore -U warehouse -d warehouse -t table_name
/backup/warehouse_backup_<date>.sql
```

2. PySpark Data Recovery

Intermediate Data:

- If intermediate data was backed up (e.g., exported as Parquet or CSV), PySpark can reload it in case of a failure. This allows you to skip earlier stages of the ETL process and restart from the most recent valid checkpoint.

Reload Example:

```
python
Copy code
df = spark.read.parquet("/path/to/backup/intermediate_data_<date>.parquet")
```

3. dbt Project Recovery

If the dbt project or its configuration files are lost or corrupted:

- **Git Restore**: If your dbt project is under version control (e.g., Git), you can easily recover the project by checking out the latest version from the repository.

```
bash
Copy code
git checkout main
```

- **Manual Restore**: If a backup of the dbt project directory exists, you can restore it by extracting the tarball to its original location.

```
bash
Copy code
tar -xzvf dbt_project_backup_<date>.tar.gz -C /path/to/dbt_project/
```

4. Full ETL Pipeline Recovery

In case of a catastrophic failure where the entire ETL pipeline environment needs recovery:

- **Docker Environment Restore**:
  - If the Docker environment (including all containers) is lost, you can use the `docker-compose.yaml` and any backed-up data/configurations to recreate the pipeline.
  - First, restore all database backups, then restart the containers using `docker-compose up`.
- **Re-run the Pipeline**:
  - Once the database and other components are restored, you can re-run the Airflow DAG to restart the ETL process from the appropriate stage.
  - Ensure all dependencies (data, scripts, configurations) are restored before executing the DAG.

C. Maintenance of Backup and Recovery Process

To ensure the backup and recovery process remains reliable:

- **Regular Backup Scheduling**: Ensure that backups (especially for PostgreSQL) are scheduled regularly and are monitored for successful completion.
- **Automated Backup Verification**: Periodically verify backups by restoring them in a separate environment to ensure they are valid and can be used in the event of a failure.
- **Disaster Recovery Drills**: Regularly perform disaster recovery drills to test the backup and recovery procedures. This ensures that the team is familiar with the process and can quickly recover from failures.
- **Backup Retention Policy**: Define a retention policy that specifies how long to keep backups based on storage capacity and business requirements.

Appendices

1. Glossary

Below are definitions of technical terms and acronyms used throughout the documentation.

- **Airflow**: An open-source tool for programmatically authoring, scheduling, and monitoring workflows. It is used to manage and orchestrate ETL pipelines.
- **BashOperator**: A type of Airflow operator that runs a bash command on a specific task in a DAG.
- **CSV (Comma Separated Values)**: A simple file format used to store tabular data, such as a spreadsheet or database.
- **Cron Job**: A time-based job scheduler in Unix-like operating systems. Cron jobs are used to schedule scripts or commands to run periodically.
- **DAG (Directed Acyclic Graph)**: In Airflow, a DAG represents a collection of tasks that need to be run, with their dependencies and execution order defined.
- **dbt (Data Build Tool)**: A command-line tool that helps analysts and engineers transform data within a data warehouse by writing SQL queries and managing transformations.
- **Data Mart**: A subset of a data warehouse focused on a specific aspect or function of the business, often used for reporting and analysis.
- **Data Pipeline**: A sequence of data processing steps where data is extracted from a source, transformed, and loaded into a destination, often for analysis.
- **ETL (Extract, Transform, Load)**: A process in which data is extracted from a source, transformed to fit the needs of the business, and loaded into a target database.
- **Fact Table**: In data warehousing, a central table that contains quantitative data (facts) for analysis, often surrounded by dimension tables.
- **Incremental Backup**: A type of backup that only includes data that has changed since the last backup, reducing the size and time needed for the process.
- **JDBC (Java Database Connectivity)**: An API that allows Java programs to connect and execute SQL statements in a relational database.
- **JSON (JavaScript Object Notation)**: A lightweight data-interchange format that is easy for humans to read and write, and for machines to parse and generate.
- **Parquet**: A columnar storage file format optimized for large-scale data processing systems like Hadoop and Spark.
- **Partitioning**: Dividing a dataset into smaller, manageable chunks (partitions) to allow parallel processing and improve performance.
- **PEP 8**: The official style guide for Python code, ensuring that code is clean, readable, and standardized.
- **PostgreSQL**: A powerful, open-source relational database management system that uses and extends the SQL language.
- **PySpark**: The Python API for Apache Spark, a distributed computing framework used for processing large-scale datasets.
- **RBAC (Role-Based Access Control)**: A security practice where access to resources is restricted based on the roles of individual users within an organization.
- **Schema**: The structure or organization of a database, defining how data is stored, organized, and related within the system.
- **SSL/TLS (Secure Sockets Layer/Transport Layer Security)**: Protocols that provide encrypted communication over the internet, ensuring data privacy and integrity.
- **Star Schema**: A type of data warehouse schema where a central fact table is connected to dimension tables, often used for query optimization in analytical environments.