Part 1:

To tackle this technical challenge effectively, I need to break it down into smaller tasks and implement each part carefully. Below is a comprehensive outline to guide you through the process:

1- Mock Data Generation:

Generate mock data that resembles realistic sales transactions.

Ensure variation in data to test different scenarios.

2-Reading Data from CSV File:

Use Python's csv module to read data from the CSV file.

Handle any exceptions that may occur during file reading.

3-Data Transformation:

Map the CSV columns to the database table columns.

Perform any necessary data cleaning and transformation.

Implement data quality checks for missing values, outliers, and inconsistencies.

Optimize for large datasets by using efficient data manipulation techniques (e.g., pandas DataFrame).

4-4- Connection to database, Parameterization, Creating table, and Loading data:

Establish a connection to the SQL database.

Use configuration files (e.g., YAML, JSON) store database connection details in order to avoid hardcoding any sensitive information in the script.

Create sales table with mentioned columns.

Implement incremental loading to process only new records since the last ETL run.

Implement robust error handling mechanisms using try-except blocks.

5-Error Handling and Logging:

Utilize Python's logging module to log errors, warnings, and other relevant information.

Ensure comprehensive monitoring of the ETL process by logging essential metrics and statuses.

6-Data Encryption:

Implement encryption techniques (e.g., AES encryption) to secure sensitive information during transit and storage.

Utilize libraries like cryptography for encryption and decryption.

7-Schema Evolution Handling:

Use libraries like pandas to handle changes to the schema of the CSV file or the destination database table.

Ensure backward and forward compatibility without data loss during schema changes.

8-Version Control with GitHub:

Create a GitHub repository to manage the development of the ETL script.

Use branches for feature development and bug fixes.

Implement issue tracking and pull requests for collaborative code review.

Maintain a clear and organized directory structure for the project.

9-Optimization and Scalability:

Optimize the script for performance by using efficient data structures and algorithms.

Consider parallel processing techniques for handling large datasets.

Test the script's scalability by benchmarking against varying dataset sizes.

10-Adherence to Best Practices:

Follow best practices for data manipulation, concurrency, error handling, logging, security, and version control.

Document the code thoroughly, including comments and docstrings.

Ensure code readability and maintainability by following PEP 8 guidelines.

11-Testing and Validation:

Develop test cases to validate the correctness and completeness of the ETL process.

Perform unit tests, integration tests, and end-to-end tests to ensure all functionalities work as expected.

Lets go through them one by one:

1-Mock Data Generation:

We'll create a Python script to generate random sales data using the faker library. This library provides a convenient way to generate various types of fake data.

This script will generate 1000 random sales records with transaction IDs, customer IDs, product IDs, quantities, and timestamps within the last year. You can adjust the number of records as needed by modifying the num\_records variable.

I've adjusted the range of quantities to vary between 1 and 20 to introduce more variability in the data.

I've introduced variability in the timestamps to simulate different sales patterns:

There's a higher probability of recent transactions (within the last week) to reflect the common scenario of recent sales data being more relevant.

Some transactions may have larger quantities (between 5 and 30) to simulate bulk orders or high-demand periods.

These changes aim to make the generated data more reflective of realistic sales transaction scenarios.

I've defined lists for product categories, brands, and types to simulate diversity in products.

During data generation, I randomly select a product category, brand, and type for each product to create a unique product ID that reflects a variety of products in different categories and from different brands.

The product ID format includes the chosen category, brand, and type, separated by underscores. This format ensures that each product ID is unique and representative of diverse products.

With these changes, the script now generates mock sales data with diverse product IDs, reflecting a variety of products that might be sold in a real-world scenario.

I've introduced randomness in customer IDs to simulate different customer behaviors. Some customers are assigned a new random customer ID for each transaction, while others reuse the same customer ID for a percentage of transactions to simulate frequent purchases by returning customers.

I've also introduced randomness in quantities to simulate different purchase behaviors. Some customers make larger orders less frequently, so a percentage of transactions have larger quantities.

I've defined a dictionary regions that maps each region or country to corresponding purchasing behaviors, such as frequency of purchases and likelihood of larger orders.

During data generation, I randomly select a region and adjust customer behavior (frequency of purchases and likelihood of larger orders) based on the chosen region.

This approach allows the script to simulate different purchasing behaviors observed in various regions or countries, ensuring that the generated mock sales data reflects geographic factors.

I've introduced randomness to simulate missing values in quantities with a 20% chance of introducing missing values.

Outliers are simulated by randomly generating larger quantities for a percentage of transactions based on customer behavior and region.

This approach ensures that the generated mock sales data includes missing values and outliers, allowing for testing of how the ETL pipeline handles such scenarios.

missing\_values containing various representations of missing values, such as None, 'N/A', an empty string '', and 'NA'.

During data generation, I randomly choose from this list to introduce different types of missing values in quantities, with a 20% chance of introducing a missing value.

This approach introduces more variability in missing value representations, allowing for better testing of how the ETL pipeline handles different types of missing values.

Run this script, and it will create a CSV file named mock\_sales\_data.csv containing the generated mock sales data.

2-Reading Data from CSV File:

The read\_csv\_file function takes the filename as input and reads the sales data from the CSV file using Python's csv.reader.

I handle exceptions using a try-except block to catch any errors that may occur during file reading. If an exception occurs, an error message is printed.

The function returns the sales data if reading is successful, otherwise, it returns None.

Finally, I call this function to read the mock sales data from the mock\_sales\_data.csv file.

This code will effectively read the data from the CSV file while handling any exceptions gracefully.

3-Data Transformation:

**The transform\_sales\_data function** takes the sales data as input and performs data cleaning and transformation tasks. It converts the sales data into a pandas DataFrame, handles missing values, converts data types, and performs data quality checks.

Convert to DataFrame: The function first converts the sales data, which is typically in a list or array format, into a pandas DataFrame. It specifies the column names as 'transaction\_id', 'customer\_id', 'product\_id', 'quantity', and 'timestamp'.

Data Cleaning and Transformation:

Convert Quantity to Numeric: It converts the 'quantity' column to numeric data type, handling errors by coercing invalid values into NaN (Not a Number).

Handle Missing Values: It fills missing values in the 'quantity' column with 0, assuming that missing quantity indicates no sales for that transaction.

Handle Missing Customer IDs: Missing customer IDs are filled with the string 'Unknown', assuming that the customer information is not available.

Handle Missing or Inconsistent Product IDs: It replaces missing or inconsistent product IDs (e.g., None, 'N/A', '', 'NA') with the string 'Unknown'.

Handle Missing Timestamps: Missing timestamps are converted to pandas Timestamp objects, and if any are still missing, they're filled with the current timestamp.

Data Quality Checks:

It checks for missing values in the DataFrame and prints the count of missing values for each column.

**The detect\_outliers function** identifies outliers in a specified column of the DataFrame using the Interquartile Range (IQR) method. It calculates the lower and upper bounds for outliers and returns a DataFrame containing the outliers.

Calculate Quartiles and IQR:

It calculates the first quartile (Q1), third quartile (Q3), and the Interquartile Range (IQR) for the specified column.

Define Outlier Boundaries:

Lower and upper boundaries for outliers are defined based on the quartiles and IQR. Outliers fall below Q1 - 1.5 \* IQR or above Q3 + 1.5 \* IQR.

Identify Outliers:

It creates a boolean mask to identify rows where the column value is outside the outlier boundaries. These rows represent outliers in the data.

Return Outliers:

The function returns a DataFrame containing the rows identified as outliers based on the specified column.

**The handle\_outliers function** replaces outliers that we detected in detect\_outliers function in a specified column of the DataFrame with a suitable value, such as the median of that column.

Replace Outliers:

Outliers below the lower boundary are replaced with the median value of the column, and outliers above the upper boundary are also replaced with the median value.

Return DataFrame:

The function returns the DataFrame with outliers replaced by the median value.

Main Function: The main function is the entry point of the script. It reads the sales data from a CSV file, transforms the data, detects and handles outliers in the 'quantity' column, and prints the original and transformed data.

4- Connection to database, Creating table, and Loading data to the created table:

Making Connection with Database:

We establish a connection with a PostgreSQL database. PostgreSQL is a powerful, open-source relational database management system known for its reliability, robustness, and extensibility.

To connect to the PostgreSQL database from Python, we use the psycopg2 library, which is a PostgreSQL adapter for Python.

Choosing PostgreSQL:

PostgreSQL was chosen as the database system due to its features, including support for ACID transactions, data integrity, extensibility, and scalability.

Additionally, PostgreSQL is widely used in enterprise environments and has strong community support.

Using YAML File and the Reason We Use It:

We use a YAML (YAML Ain't Markup Language) configuration file to store database connection details, such as server address, port, database name, username, and password.

YAML is a human-readable data serialization standard that is easy to understand and edit.

Storing configuration details in a separate YAML file makes it easier to manage and update the configuration without modifying the source code.

This approach follows best practices for separating configuration from code, making the code more maintainable and secure.

Creation Table Python Script:

The Python script is responsible for creating a table named "sales" in the PostgreSQL database.

We use the psycopg2 library to establish a connection to the PostgreSQL database.

The SQL query for creating the table is defined as a multi-line string in the Python script.

The script executes the SQL query using a cursor object obtained from the database connection.

If the table does not exist, it is created with the specified columns: transaction\_id, customer\_id, product\_id, quantity, and sale\_date.

This script ensures that the necessary table structure is in place before we start loading data into the database.

By following this process, we ensure that we have a robust and scalable database setup with PostgreSQL, manage database connection details using a YAML configuration file, and create the required table structure using a Python script. This approach allows for flexibility, maintainability, and ease of use in managing the database and its schema.

Files under database folder in the project are:

create\_table.py:

Purpose: This script is responsible for creating the 'sales' table in a PostgreSQL database based on the provided configuration.

Usage: It reads the database configuration from the config.yaml file, establishes a connection to the database using database\_utils.py, and executes a SQL query to create the 'sales' table.

Dependencies:

config.yaml: Contains the database connection details such as host, port, dbname, user, and password.

database\_utils.py: Provides utility functions for loading configuration and connecting to the database.

Flow:

Load database configuration from config.yaml.

Connect to the PostgreSQL database using database\_utils.connect\_to\_database.

Execute a SQL query to create the 'sales' table if it does not exist.

Close the database connection using database\_utils.close\_connection.

Error Handling: Handles errors that may occur during database connection or table creation.

Output: Prints status messages indicating the success or failure of the table creation process.

delete\_table.py:

Purpose: This script is responsible for deleting the 'sales' table from a PostgreSQL database if it exists.

Usage: It reads the database configuration from the config.yaml file, establishes a connection to the database using database\_utils.py, checks if the 'sales' table exists, and deletes it if it does.

Dependencies:

config.yaml: Contains the database connection details.

database\_utils.py: Provides utility functions for loading configuration, connecting to the database, and closing connections.

Flow:

Load database configuration from config.yaml.

Connect to the PostgreSQL database using database\_utils.connect\_to\_database.

Check if the 'sales' table exists by querying the information schema.

If the table exists, execute a SQL query to delete it.

Close the database connection using database\_utils.close\_connection.

Error Handling: Handles errors that may occur during database connection, table existence check, or table deletion.

Output: Prints status messages indicating the success or failure of the table deletion process.

database\_utils.py:

Purpose: This script contains utility functions for database operations such as loading configuration, connecting to the database, and closing connections.

Functions:

load\_config(file\_path): Loads the database configuration from a YAML file.

connect\_to\_database(config): Connects to the PostgreSQL database using the provided configuration.

close\_connection(conn, cursor=None): Closes the database connection and cursor if they are open.

Dependencies: None directly, but indirectly depends on yaml and psycopg2 modules.

Flow: Each function performs a specific database-related task and provides error handling where necessary.

Output: Does not produce any output directly but may print status messages or errors during execution.

config.yaml:

Purpose: This YAML file stores the configuration details required for connecting to the PostgreSQL database.

Contents: Contains the following fields:

host: Hostname or IP address of the PostgreSQL server.

port: Port number on which the PostgreSQL server is running.

dbname: Name of the database to connect to.

user: Username for authenticating to the database.

password: Password for authenticating to the database.

Usage: Provides the necessary parameters for establishing a connection to the PostgreSQL database.

Security: Contains sensitive information such as passwords, so it should be kept secure and not shared publicly.

Format: Follows the YAML syntax for key-value pairs and nested structures.

Overall Flow:

The create\_table.py and delete\_table.py scripts utilize the functions defined in database\_utils.py to interact with the database.

They both rely on the database configuration provided in config.yaml to establish connections and perform operations.

Error handling is implemented throughout the scripts to handle potential issues during database operations.

These scripts are modular and can be reused or extended for similar tasks in other projects.

Load Data into Database:

Function: load\_data(conn, df)

This function is responsible for loading the transformed data from a DataFrame into a PostgreSQL database.

Parameters:

conn: PostgreSQL database connection object.

df: Pandas DataFrame containing the transformed sales data.

Steps:

Open Cursor:

Open a cursor to execute SQL queries.

Iterate Over DataFrame Rows:

Loop through each row in the DataFrame.

Extract transaction details such as transaction\_id, customer\_id, product\_id, quantity, and sale\_date.

Construct SQL Query:

Construct an SQL query to insert the row data into the sales table in the database.

Execute SQL Query:

Execute the SQL query with the row data using the cursor.

Commit Transaction:

Commit the transaction to save changes to the database.

Exception Handling:

Handle exceptions that may occur during data loading, such as database connection errors or SQL execution errors.

Roll back the transaction if an error occurs to ensure data consistency.

5-Error Handling and Logging:

Error Handling: Logging allows capturing errors and exceptions that occur during the execution of the script. These error messages are recorded along with timestamps, severity levels, and descriptions of the error. This helps in diagnosing and troubleshooting issues that may arise during script execution.

Informational Logging: Besides errors, the logging system also captures informational messages. For example, when the script successfully reads data from a CSV file, transforms it, handles outliers, loads data into the database, or closes the database connection, it logs corresponding informational messages. These messages provide insights into the flow of execution and the successful completion of various tasks.

Level-Based Logging: The logging system is configured to log messages at different severity levels, such as INFO, ERROR, and potentially others. This allows distinguishing between different types of messages based on their importance. For example, informational messages may provide general progress updates, while error messages indicate critical issues that need attention.

Log Formatting: The logging system is set up with a specific format for log messages, including timestamps, severity levels, and message descriptions. This standardized format ensures consistency and readability of log messages, making it easier for developers and administrators to interpret them.

Logging to File: The logging system is configured to write log messages to a file (etl.log in this case). This file serves as a persistent record of script execution, errors encountered, and other relevant information. Storing log messages in a file allows for later analysis, auditing, and debugging.

Encryption of Customer ID:

To enhance the security of sensitive information, particularly the customer\_id field, encryption techniques are employed during data transit and storage. The encryption process involves converting the customer\_id into a ciphertext using the Advanced Encryption Standard (AES) encryption algorithm.

Encryption Process:

Key Generation: A secret key is generated using the Fernet symmetric encryption algorithm provided by the cryptography library.

Encryption Function:

Before storing the customer\_id in the database, it is encrypted using the generated secret key.

The encrypt\_customer\_id function accepts the customer\_id as input, converts it into ciphertext using the Fernet algorithm, and returns the encrypted value.

Decryption Function:

When retrieving the customer\_id from the database, it needs to be decrypted back to its original form.

The decrypt\_customer\_id function takes the encrypted customer\_id as input, decrypts it using the Fernet algorithm and the secret key, and returns the original plaintext value.

Integration:

During the data transformation process, the customer\_id is encrypted using the encrypt\_customer\_id function before insertion into the database.

When accessing the customer\_id from the database, it is decrypted using the decrypt\_customer\_id function to obtain the original value.

Key Management:

The secret key used for encryption and decryption is generated dynamically within the script using the Fernet algorithm.

It's crucial to securely manage and protect the encryption key to prevent unauthorized access to sensitive information.

Consider storing the key in a secure environment or utilizing a key management system for enhanced security.

By implementing encryption techniques like AES encryption, the confidentiality and integrity of sensitive data, such as the customer\_id, are safeguarded, reducing the risk of unauthorized access or data breaches.