Phase 1: Extract

```
In [1]: import pandas as pd
        import pymysql
```

```
In [ ]: # Connect to the MySQL database
        conn = pymysql.connect(
            host='localhost',
            user='your_mysql_username',
            password='your_mysql_password',
            database='book_store'
        )

        cursor = conn.cursor()
```

```
In [4]: # Extract data from CSV files
        authors_df = pd.read_csv('.csv adatforrások/authors.csv')
        book_genres_df = pd.read_csv('.csv adatforrások/book_genres.csv')
        books_df = pd.read_csv('.csv adatforrások/books.csv', usecols=['id','title','author_id','release_date',
                                                                       'description','list_price'], encoding="utf-8-sig")
        customers_df = pd.read_csv('.csv adatforrások/customers.csv', encoding="utf-8-sig")
        genres_df = pd.read_csv('.csv adatforrások/genres.csv')
        inventory_df = pd.read_csv('.csv adatforrások/inventory.csv')
        order_items_df = pd.read_csv('.csv adatforrások/order_items.csv')
        orders_df = pd.read_csv('.csv adatforrások/orders.csv')
        payments_df = pd.read_csv('.csv adatforrások/payments.csv')
        warehouses_df = pd.read_csv('.csv adatforrások/warehouses.csv', encoding="utf-8-sig")
```

What’s Happening Here?

- We start by importing the necessary libraries: pandas for data manipulation and pymysql to connect to the MySQL database.
- We then establish a connection to the MySQL database using the pymysql.connect method.
- The pd.read_csv function is used to extract data from CSV files, respectively. This data is loaded into DataFrames, which are versatile and powerful data structures in Python that allow us to easily manipulate and analyze the data.

Phase 2: Transform

In [ ]:
```
# Update list prices in order_items table
for i in range(len(books_df)):
    mask = order_items_df['book_id'] == books_df.loc[i, 'id']
    order_items_df.loc[mask, 'unit_price'] = books_df.loc[i, 'list_price']

# Calculate total amount in orders table
for i in range(len(orders_df)):
    total = 0
    mask = order_items_df['order_id'] == orders_df.loc[i, 'id']
    total = order_items_df.loc[mask, 'line_total'].sum()
    orders_df.loc[i, 'total_amount'] = total

# Calculate total stock in orders table
for i in range(len(orders_df)):
    total = 0
    mask = order_items_df['order_id'] == orders_df.loc[i, 'id']
    total = order_items_df.loc[mask, 'order_quantity'].sum()
    orders_df.loc[i, 'total_stock'] = total
```

In [ ]:
```
# Handling invalid emails in the Users DataFrame
def generate_unique_email(index):
    return f'unknown_{index}@example.com'

customers_df['emailaddress'] = customers_df.apply(lambda row: row['emailaddress'] if '@' in row['emailaddress']and '.' in
                                                  row['emailaddress'] else generate_unique_email(row.name), axis=1)

# Ensure all prices are numeric
books_df['list_price'] = pd.to_numeric(books_df['list_price'], errors='coerce').fillna(0.0).round(2)
orders_df['total_amount'] = pd.to_numeric(orders_df['total_amount'], errors='coerce').fillna(0.0).round(2)
order_items_df['unit_price'] = pd.to_numeric(order_items_df['unit_price'], errors='coerce').fillna(0.0).round(2)
order_items_df['line_total'] = pd.to_numeric(order_items_df['line_total'], errors='coerce').fillna(0.0).round(2)

# Ensure all stock quantities are numeric
orders_df['total_stock'] = pd.to_numeric(orders_df['total_stock'], errors='coerce').fillna(0).astype(int)
order_items_df['order_quantity'] = pd.to_numeric(order_items_df['order_quantity'], errors='coerce').fillna(0).astype(int)
```

```
inventory_df['quantity'] = pd.to_numeric(inventory_df['quantity'], errors='coerce').fillna(0).astype(int)

# Standardize and convert date formats to string
books_df['release_date'] = pd.to_datetime(books_df['release_date'], errors='coerce', format='%Y-%m-%d').fillna
(pd.Timestamp('1971-01-01'))
books_df['release_date'] = books_df['release_date'].dt.strftime('%Y-%m-%d')

orders_df['order_date'] = pd.to_datetime(orders_df['order_date'], errors='coerce').fillna(pd.Timestamp('1971-01-01'))
orders_df['order_date'] = orders_df['order_date'].dt.strftime('%Y-%m-%d %H:%M:%S')

orders_df['ship_date'] = pd.to_datetime(orders_df['ship_date'], errors='coerce').fillna(pd.Timestamp('1971-01-01'))
orders_df['ship_date'] = orders_df['ship_date'].dt.strftime('%Y-%m-%d %H:%M:%S')

customers_df['dateofbirth'] = pd.to_datetime(customers_df['dateofbirth'], errors='coerce', format='%Y-%m-%d').fillna
(pd.Timestamp('1971-01-01'))
customers_df['dateofbirth'] = customers_df['dateofbirth'].dt.strftime('%Y-%m-%d')

# Replace negative values with 0 in inventory table
inventory_df['quantity'] = inventory_df['quantity'].clip(lower=0)
```

In [7]:
```
# Fill missing values in customers dataframe as needed
customers_df = customers_df.fillna({
    'gender': 'Missing',
    'dateofbirth': '1971-01-01',
    'postalcode': 'Missing',
    'country': 'Missing',
    'region': 'Missing',
    'city': 'Missing',
    'street': 'Missing',
    'houseno': 'Missing',
    'registrationdate': '1971-01-01'
})
```

What’s Happening Here?

- Filling Missing Values: We use the fillna method to replace missing values with default values. For example, if the country or region fields are missing in the customers_df DataFrame, we fill them with 'Missing', respectively.

- Handling Invalid Emails: We define a function generate_unique_email to create a placeholder email for records with invalid or missing email addresses.
- Ensuring Numeric Data: We convert the total_stock fields in orders_df, order_quantity fields in order_items_df and quantity fields in inventory_df to numeric values. Any non-numeric data is replaced with a default 0 value.
- Standardizing Dates: Dates in books_df, orders_df and customers_df are standardized to a consistent format (YYYY-MM-DD) and any invalid dates are replaced with a default placeholder date (1971-01-01).

Phase 3: Load

In [13]:
```
# Insert data into Authors table
author_id_mapping = {}
for index, row in authors_df.iterrows():
    cursor.execute("""
        INSERT INTO authors (author_name, date_of_birth)
        VALUES (%s, %s)
    """, (row['author_name'], row['date_of_birth']))

    author_id_mapping[index] = cursor.lastrowid
conn.commit()
```

In [45]:
```
# Insert data into Books table
book_id_mapping = {}
for index, row in books_df.iterrows():
    cursor.execute("""
        INSERT INTO books (title, author_id, release_date, list_price, description)
        VALUES (%s, %s, %s, %s, %s)
    """, (row['title'], row['author_id'], row['release_date'], row['list_price'], row['description']))

    book_id_mapping[index] = cursor.lastrowid
conn.commit()
```

In [15]:
```
# Insert data into Genres table
genre_id_mapping = {}
for index, row in genres_df.iterrows():
    cursor.execute("""
        INSERT INTO genres (genre)
        VALUES (%s)
    """, (row['genre']))

    genre_id_mapping[index] = cursor.lastrowid
conn.commit()
```

In [16]:
```
# Insert data into Books_Genres table
book_genre_id_mapping = {}
for index, row in book_genres_df.iterrows():
    cursor.execute("""
        INSERT INTO books_genres (book_id, genre_id)
        VALUES (%s, %s)
    """, (row['book_id'], row['genre_id']))

    book_genre_id_mapping[index] = cursor.lastrowid
conn.commit()
```

In [17]:
```
# Insert data into Customers table
customer_id_mapping = {}
for index, row in customers_df.iterrows():
    cursor.execute("""
        INSERT INTO customers (name, gender, date_of_birth, postal_code, country, region, city, street, house_no, email_addres
        VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
    """, (row['name'], row['gender'], row['dateofbirth'], row['postalcode'], row['country'], row['region'], row['city'], row['
          row['houseno'], row['emailaddress'], row['registrationdate']))

    customer_id_mapping[index] = cursor.lastrowid
conn.commit()
```

In [18]:
```
# Insert data into Warehouses table
warehouse_id_mapping = {}
for index, row in warehouses_df.iterrows():
    cursor.execute("""
        INSERT INTO warehouses (name, location, country)
        VALUES (%s, %s, %s)
    """, (row['name'], row['location'], row['country']))

    warehouse_id_mapping[index] = cursor.lastrowid
conn.commit()
```

In [19]:
```python
# Insert data into Payments table
payment_id_mapping = {}
for index, row in payments_df.iterrows():
    cursor.execute("""
        INSERT INTO payments (method)
        VALUES (%s)
    """, (row['payment_type']))

    payment_id_mapping[index] = cursor.lastrowid
conn.commit()
```

In [20]:
```python
# Insert data into Inventory table
inventory_id_mapping = {}
for index, row in inventory_df.iterrows():
    cursor.execute("""
        INSERT INTO inventory (book_id, warehouse_id, quantity)
        VALUES (%s, %s, %s)
    """, (row['book_id'], row['warehouse_id'], row['quantity']))

    inventory_id_mapping[index] = cursor.lastrowid
conn.commit()
```

In [47]:
```python
# Insert data into Orders table
order_id_mapping = {}
for index, row in orders_df.iterrows():
    cursor.execute("""
        INSERT INTO orders (order_date, ship_date, customer_id, payment_id, total_items, total_amount)
        VALUES (%s, %s, %s, %s, %s, %s)
    """, (row['order_date'], row['ship_date'], row['customer_id'], row['payment_id'], row['total_stock'], row['total_amount'])

    order_id_mapping[index] = cursor.lastrowid
conn.commit()
```

In [22]:
```python
# Insert data into Order items table
cursor.execute(f"TRUNCATE TABLE order_items")
order_item_id_mapping = {}
for index, row in order_items_df.iterrows():
    cursor.execute("""
        INSERT INTO order_items (order_id, book_id, warehouse_id, order_quantity, unit_price, unit_price_discount)
        VALUES (%s, %s, %s, %s, %s, %s)
    """, (row['order_id'], row['book_id'], row['warehouse_id'], row['order_quantity'], row['unit_price'], row['unit_price_disc

    order_item_id_mapping[index] = cursor.lastrowid
conn.commit()
```

What’s Happening Here?

- Inserting Data: We iterate through each DataFrame, row by row, and insert the data into the corresponding table in the MySQL database. The cursor.execute method is used to run SQL INSERT statements with the data from each row.
- Mapping IDs: Saves the newly inserted row’s auto-generated primary key (ID) from the database into a dictionary, using the DataFrame row’s index as the key.