Applied A.I. Solutions
Foundations of Data Management

Lab Exercises 2

Group-10 members

1. Goyal, Vinayak
2. Bhasgauri, Harshal Shashikant
3. Sebastian, Arun
4. ., Himani
5. Singh, Satyajeet
6. Trongkitroongruang, Kajhonprom
7. Cheng, Qianfan

Introduction

In the rapidly evolving world of data, understanding the intricacies of data management and database design is paramount. This lab exercise is a testament to that exploration. The purpose of this exercise is to delve deep into the world of data analysis, design, and management using the "Sample Superstore" dataset as our primary data source. Through this endeavor, we aim to:

Understand and visually represent the flow of data within a system using a Data Flow Diagram.

Design and implement a database schema that accurately captures the relationships, entities, and attributes of our dataset, as depicted in the Entity Relationship Diagram.

Utilize programming and database tools, specifically Python and SQLalchemy, to automate data extraction, cleaning, and loading processes, ensuring data integrity and consistency.

Highlight the methodologies and codes employed in this intricate process, providing a clear roadmap for similar future projects.

Our journey begins with loading the raw data from the "Sample Superstore" dataset into a dedicated table, ‘superstore_orders'. This table serves as our base, holding the raw data in its original form. From there, using a combination of Python and SQL, we dissect, refine, and channel this data into specific entities, ensuring it's primed for analysis and reporting. The subsequent sections detail our approach, methodologies, and the results of this exercise, with visual representations and code snippets provided in the appendices.

Methodology

The methodology employed in this lab exercise was meticulously crafted to ensure a systematic and efficient approach to handling, transforming, and analyzing the data from the "Sample Superstore" dataset. The primary stages of our methodology are detailed below.

1. **Data Exploration**

Before any transformations or cleaning, it's crucial to understand the data's structure, attributes, and potential inconsistencies. Tools like Python, with its extensive data analysis libraries, provided a quick and comprehensive overview of the dataset, highlighting areas that required attention.

2. **Data Visualization with Lucidchart**

Visual representations often simplify complex data structures, making them more comprehensible. Lucidchart was chosen due to its intuitive interface and the ease with which it can create both Data Flow Diagrams and Entity Relationship Diagrams. These visual aids were instrumental in mapping out the data's flow and relationships, setting a clear path for subsequent stages.

3. **Data Cleaning & Transformation with Python and SQL**

Data rarely comes in a perfect format. Using Python, we were able to automate several cleaning tasks such as filling in missing values, removing duplicates, and splitting attributes (like names). SQL, on the other hand, provided a robust platform for data extraction, transformation, and loading (ETL). Its inherent capabilities in handling relational data made it an ideal choice for tasks like aggregations, joins, and filtering.

4. **Database Design with SQLalchemy**

The design of a database is paramount to its efficiency and ease of use. SQLalchemy, a versatile toolkit for SQL in Python, was used to streamline the creation of entities in the database, ensuring a smooth transition from raw data to a structured database format. Its Object Relational Mapping (ORM) capabilities allowed for a more intuitive interaction with the database using Python.

5. Iterative Refinement

Given the complex nature of data transformation and design, our approach was inherently iterative. After each major step, a series of checks and validations were performed to ensure data integrity and consistency. If inconsistencies were detected, the data was looped back for refinement, ensuring the highest quality in the final output.

Data analysis:

1. Metadata table:
    a. metadata

2. Reference data tables:
    1. countries
    2. states
    3. cities
    4. categories
    5. segments
    6. order statuses
    7. regions

3. Master data tables:
    1. customers
    2. addresses
    3. employees
    4. address customers
    5. products

4. Transactional data tables:
    1. orders
    2. product orders
    3. shipments

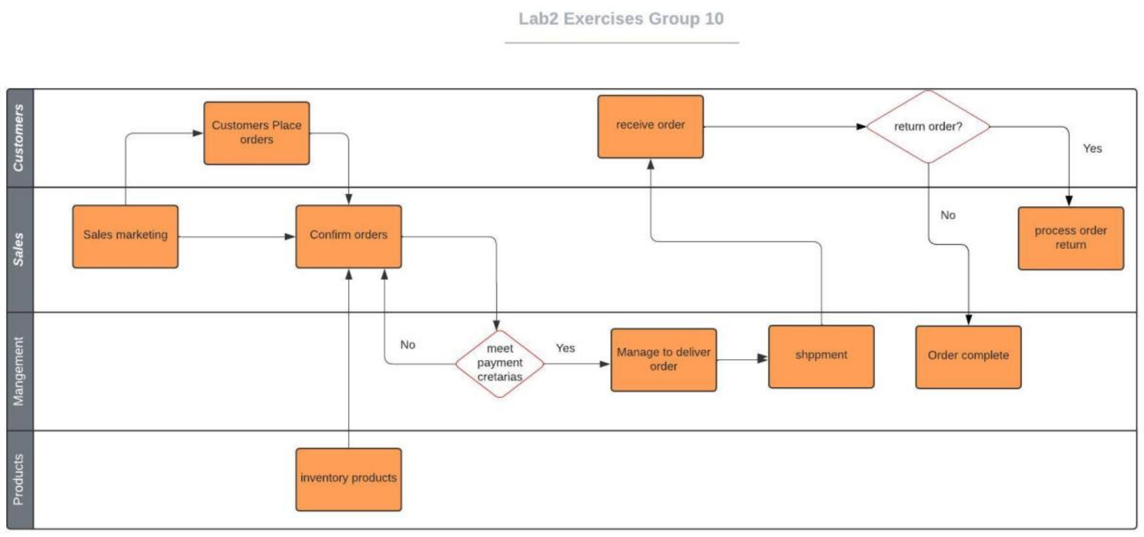
5. Reporting data tables:
    1. Operational reports
    2. Executives' reports

Data Modification:

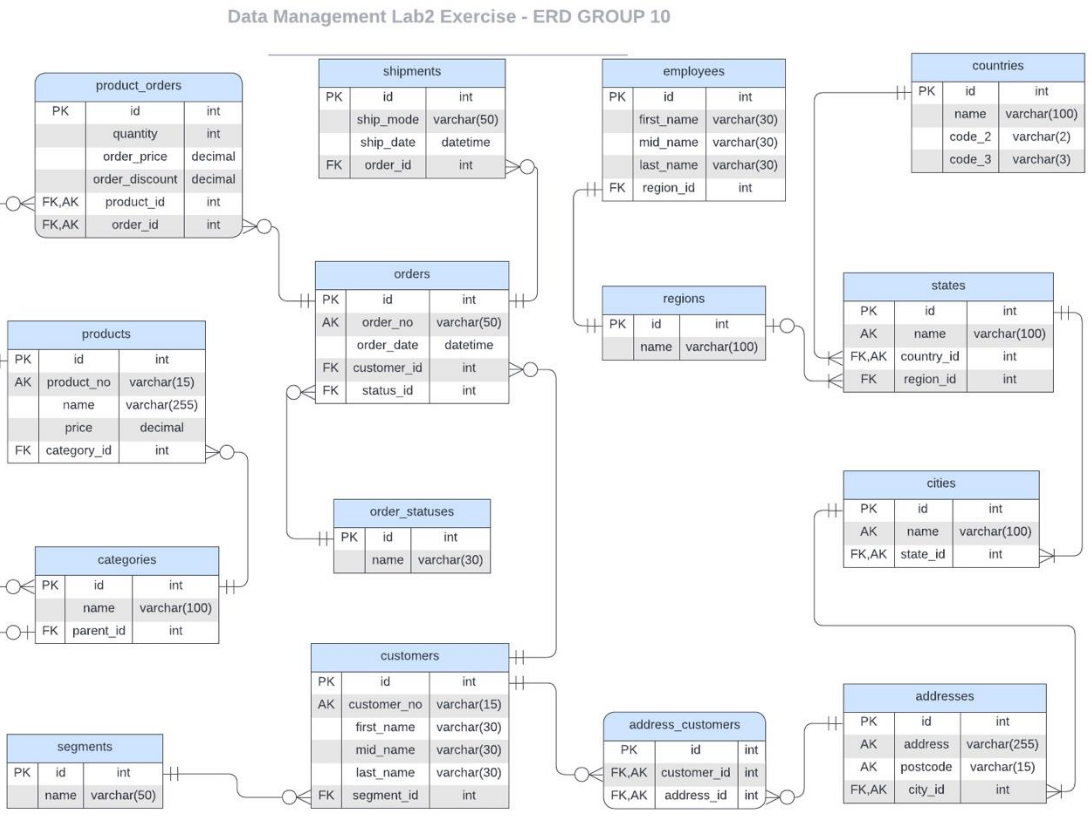
1. Missing 'postal codes' are filled in with random values picked from the postal codes belonging to that city.
2. Some products have different names with the same product number, in this case, different names are updated with a unique name.

3. Order statuses are filled in with 'returned' and 'completed'. To accomplish this, the 'return_status_id' attribute is added to the superstore_orders table, values are filled if orders are returned, and kept 'null' if not.
4. The 'address' attribute is added to the address’s entity, and no data will be filled in.
5. The 'price' attribute is added to the product entity, it means the unit price for a product.
6. Names are divided into 'first_name', 'mid_name', and 'last_name' attributes in all name-related attributes. Names will be split based on whitespace and saved in different attributes.
7. 'code_2' and 'code_3' attributes are added to the countries entity, for future possible usage.
8. The same product appears in the same order multiple times with different quantities and sales, in this scenario, these product's quantity will be summed to remove duplicates.

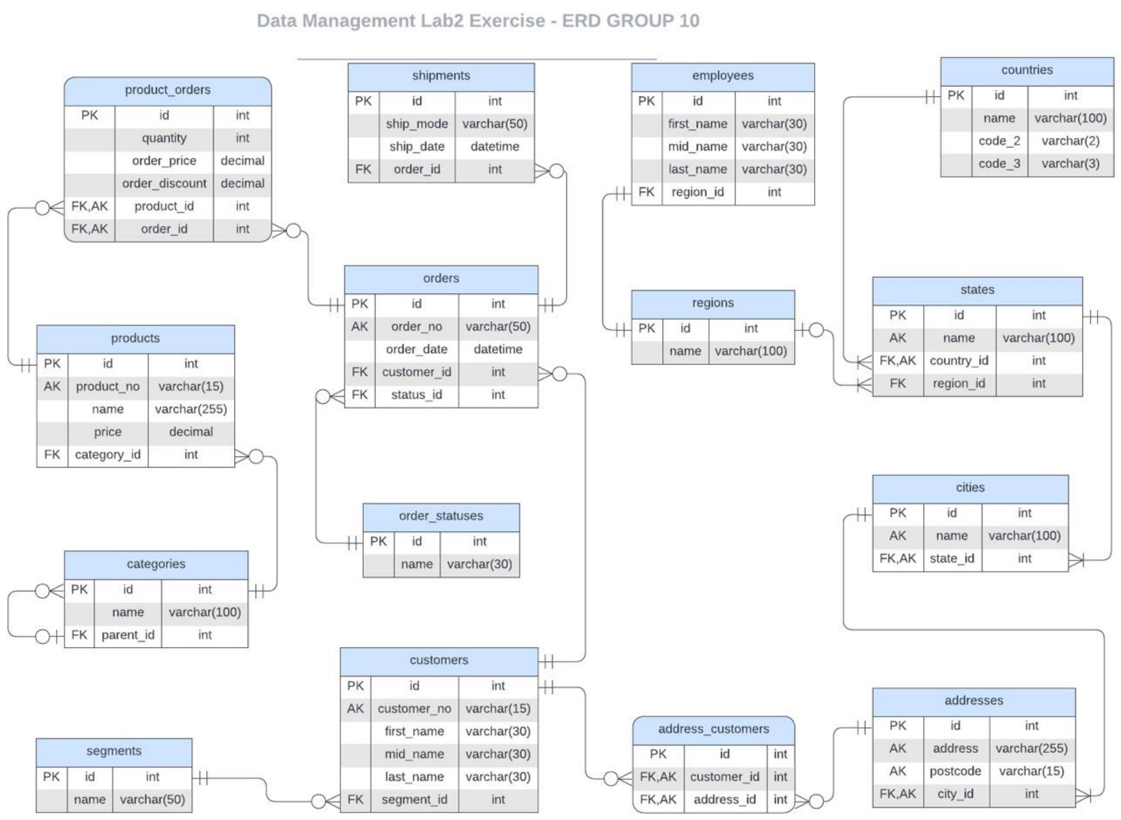
Appendix I – Data Flow Diagram

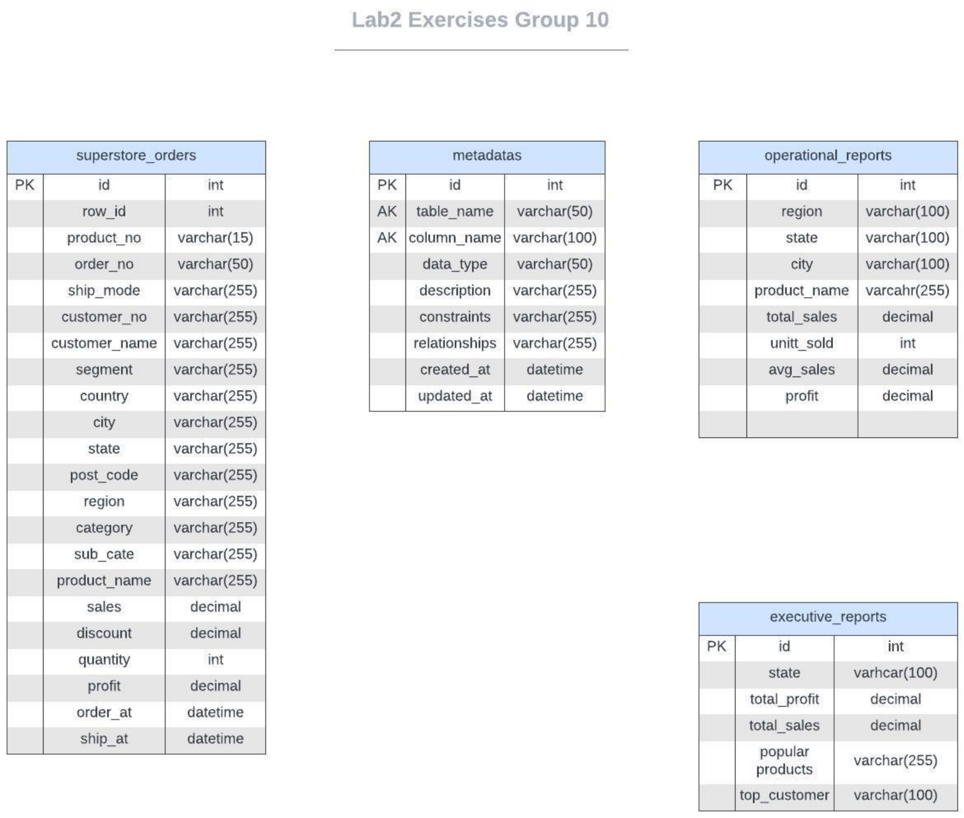


Appendix II – ERD



Data Schema


Data Management Lab2 Exercise - ERD GROUP 10
product_orders
PK id int
quantity int
order_price decimal
order_discount decimal
FK,AK product_id int
FK,AK order_id int
shipments
PK id int
ship_mode varchar(50)
ship_date datetime
FK order_id int
employees
PK id int
first_name varchar(30)
mid_name varchar(30)
last_name varchar(30)
FK region_id int
countries
PK id int
name varchar(100)
code_2 varchar(2)
code_3 varchar(3)
orders
PK id int
AK order_no varchar(50)
order_date datetime
FK customer_id int
FK status_id int
regions
PK id int
name varchar(100)
states
PK id int
AK name varchar(100)
FK,AK country_id int
FK region_id int
products
PK id int
AK product_no varchar(15)
name varchar(255)
price decimal
FK category_id int
order_statuses
PK id int
name varchar(30)
cities
PK id int
AK name varchar(100)
FK,AK state_id int
categories
PK id int
name varchar(100)
FK parent_id int
customers
PK id int
AK customer_no varchar(15)
first_name varchar(30)
mid_name varchar(30)
last_name varchar(30)
FK segment_id int
address_customers
PK id int
FK,AK customer_id int
FK,AK address_id int
addresses
PK id int
AK address varchar(255)
AK postcode varchar(15)
FK,AK city_id int
segments
PK id int
name varchar(50)



Lab2 Exercises Group 10

superstore_orders		
PK	id	int
	row_id	int
	product_no	varchar(15)
	order_no	varchar(50)
	ship_mode	varchar(255)
	customer_no	varchar(255)
	customer_name	varchar(255)
	segment	varchar(255)
	country	varchar(255)
	city	varchar(255)
	state	varchar(255)
	post_code	varchar(255)
	region	varchar(255)
	category	varchar(255)
	sub_cate	varchar(255)
	product_name	varchar(255)
	sales	decimal
	discount	decimal
	quantity	int
	profit	decimal
	order_at	datetime
	ship_at	datetime

metadatas		
PK	id	int
AK	table_name	varchar(50)
AK	column_name	varchar(100)
	data_type	varchar(50)
	description	varchar(255)
	constraints	varchar(255)
	relationships	varchar(255)
	created_at	datetime
	updated_at	datetime

operational_reports		
PK	id	int
	region	varchar(100)
	state	varchar(100)
	city	varchar(100)
	product_name	varcahr(255)
	total_sales	decimal
	unitt_sold	int
	avg_sales	decimal
	profit	decimal

executive_reports		
PK	id	int
	state	varhcar(100)
	total_profit	decimal
	total_sales	decimal
	popular products	varchar(255)
	top_customer	varchar(100)

Appendix III - Codes

Create models for entities:

```
from datetime import datetime
from decimal import Decimal
from typing import List
from typing import Optional

from sqlalchemy import ForeignKey
from sqlalchemy import func
from sqlalchemy import Numeric
# from sqlalchemy import Integer
from sqlalchemy.dialects.mysql import INTEGER
from sqlalchemy import String
from sqlalchemy import Index
from sqlalchemy.orm import DeclarativeBase
from sqlalchemy.orm import Mapped
from sqlalchemy.orm import mapped_column
from sqlalchemy.orm import registry
```

```python
from sqlalchemy.orm import relationship

mapper_registry = registry()

class Base(DeclarativeBase):
    pass

# metadata table
class Metadata(Base):
    __tablename__ = 'metadatas'

    id = mapped_column(INTEGER(unsigned=True),
                       primary_key=True,
                       autoincrement=True)
    table_name: Mapped[str] = mapped_column(String(50),
                                            nullable=False,
                                            comment='table name')
    column_name: Mapped[Optional[str]] = mapped_column(String(100),
                                                       comment='column name')
    data_type: Mapped[Optional[str]] = mapped_column(String(50),
                                                     comment='data type for the
column, eg: int')
    description: Mapped[Optional[str]] = mapped_column(String(255),
                                                       comment='description')
    constraints: Mapped[Optional[str]] = mapped_column(String(255),
                                                       comment='constrains, eg:
foreign key, index, unique')
    relationships: Mapped[Optional[str]] = mapped_column(String(255),
                                                         comment='relationships, eg:
has many... belongs to...')

    created_at: Mapped[datetime] = mapped_column(insert_default=func.now())
    updated_at: Mapped[datetime] = mapped_column(insert_default=func.now())

    __table_args__ = (
        Index('table_column_name_index', 'table_name', 'column_name', unique=True),
    )

    def __repr__(self):
        return f'<Metadata {self.id} ({self.table_name} - {self.column_name})>'

# reporting data
class ExecutiveReport(Base):
    __tablename__ = 'executive_reports'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    region: Mapped[str] = mapped_column(String(100), nullable=True,
                                      comment='region name, such as "Ontario"')
    state: Mapped[str] = mapped_column(String(100), nullable=True,
                                  comment='state name, such as "Ontario"')
    city: Mapped[str] = mapped_column(String(100), nullable=True,
                                  comment='city name, such as "Ontario"')
    product_name: Mapped[str] = mapped_column(String(255), nullable=True,
                                  comment='city name, such as "Ontario"')
    total_sales: Mapped[int] = mapped_column(Numeric(12, 2), nullable=True,
                                    comment='product discount in order, such as
"0.23"')
    unit_sold: Mapped[Optional[int]]
    avg_sales: Mapped[int] = mapped_column(Numeric(12, 2), nullable=True,
                                    comment='product discount in order, such as
"0.23"')
    profit: Mapped[int] = mapped_column(Numeric(12, 2), nullable=True,
                                    comment='product discount in order, such as
"0.23"')


# reporting data
class OperationalReport(Base):
    __tablename__ = 'operational_reports'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    state: Mapped[str] = mapped_column(String(100), nullable=True,
                                  comment='state name, such as "Ontario"')
    total_profit: Mapped[int] = mapped_column(Numeric(12, 2), nullable=True,
                                    comment='product discount in order, such as
"0.23"')
    total_sales: Mapped[int] = mapped_column(Numeric(12, 2), nullable=True,
                                    comment='product discount in order, such as
"0.23"')

# reference data
class Segment(Base):
    __tablename__ = 'segments'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(50), nullable=False,
                                  comment='segment name, such as "Home Office"')

    # one segment can be assignment to multiple customers
    customers:Mapped[List['Customer']] = relationship(back_populates='segment')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.name})>'

# master data
class Customer(Base):
    __tablename__ = 'customers'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    customer_no: Mapped[str] = mapped_column(String(15), unique=True,
                                             comment='generated customer number, such
as "ABC-SDF-121123"')
    first_name: Mapped[str] = mapped_column(String(30), nullable=False,
                                            comment='first name')
    mid_name: Mapped[Optional[str]] = mapped_column(String(30),
                                                    comment='middle name')
    last_name: Mapped[str] = mapped_column(String(30), nullable=False,
                                           comment='last name')

    segment_id: Mapped[INTEGER(unsigned=True)] =
mapped_column(ForeignKey('segments.id',
                                                                          ondelete='NO
ACTION',

onupdate='CASCADE'),
                                                               nullable=False,
                                                               comment='fk: references
to segments table')
    # one customer has one segment
    segment:Mapped['Segment'] = relationship(back_populates='customers')
    # one customer have many orders
    orders:Mapped[List['Order']] = relationship(back_populates='customer')
    # customer have mupltiple addresses
    address_customers:Mapped[List['AddressCustomer']] =
relationship(back_populates='customer')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.customer_no} references to segment:
{self.segment_id})>'

# reference data
class OrderStatus(Base):
    __tablename__ = 'order_statuses'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(30), nullable=False,
                                      comment='order status label, such as
"returned"')

    # one order status label can be assigned to multiple orders
    orders:Mapped[List['Order']] = relationship(back_populates='order_status')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.name})>'
```

```
# transactional data
class Order(Base):
    __tablename__ = 'orders'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    order_no: Mapped[str] = mapped_column(String(50), unique=True, comment='generated
order number, such as "ABC-SDF-121123"')
    order_date: Mapped[datetime] = mapped_column(insert_default=func.now())
    customer_id: Mapped[INTEGER(unsigned=True)] =
mapped_column(ForeignKey('customers.id', ondelete='NO ACTION', onupdate='CASCADE'),
                                                               nullable=False,
                                                               comment='fk: references
to customers table')

    status_id: Mapped[INTEGER(unsigned=True)] =
mapped_column(ForeignKey('order_statuses.id', ondelete='NO ACTION',
onupdate='CASCADE'),
                                                               nullable=False,
                                                               comment='fk: references
to order_statuses table')
    # one order belongs to one customer
    customer:Mapped['Customer'] = relationship(back_populates='orders')
    # one order has one order status label
    order_status:Mapped['OrderStatus'] = relationship(back_populates='orders')
    # one order has one shipment info
    shipment:Mapped['Shipment'] = relationship(back_populates='order')
    # one order has many product orders
    product_orders:Mapped[List['ProductOrder']] = relationship(back_populates='order')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.order_no}, {self.customer_id})>'

# transactional data
class Shipment(Base):
    __tablename__ = 'shipments'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    ship_mode: Mapped[str] = mapped_column(String(50), nullable=False,
                                           comment='ship mode, such as ECONOMIC
')
    ship_date: Mapped[datetime] = mapped_column(insert_default=func.now())
    order_id: Mapped[INTEGER(unsigned=True)] = mapped_column(ForeignKey('orders.id',
                                                                        ondelete='NO
ACTION',

onupdate='CASCADE'),
                                                               nullable=False,
                                                               comment='fk: references 
to orders table')
    # one shippment info belongs to one order
    order:Mapped['Order'] = relationship(back_populates='shipment')
    # order_status:Mapped['OrderStatus'] = relationship(back_populates='orders')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.ship_date} {self.order_id})>'

# master data
class Employee(Base):
    __tablename__ = 'employees'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    first_name: Mapped[str] = mapped_column(String(30), nullable=False,
                                            comment='first name')
    mid_name: Mapped[Optional[str]] = mapped_column(String(30),
                                                    comment='middle name')
    last_name: Mapped[str] = mapped_column(String(30), nullable=False,
                                           comment='last name')

    region_id: Mapped[INTEGER(unsigned=True)] = mapped_column(ForeignKey('regions.id',
                                                                      ondelete='NO 
ACTION',

onupdate='CASCADE'),
                                                               nullable=False,
                                                               comment='fk: references 
to regions table')
    # one employee has one region
    region:Mapped['Region'] = relationship(back_populates='employee')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.first_name} references to segment: 
{self.region_id})>'

# reference data
class Region(Base):
    __tablename__ = 'regions'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(100), nullable=False,
                                      comment='region name, such as "West"')

    # one region belongs to one employee
    employee:Mapped['Employee'] = relationship(back_populates='region')
    # one region has many states
    states:Mapped[List['State']] = relationship(back_populates='region')
    def __repr__(self):
        return f'<Metadata {self.id} - {self.name})>'

# reference data
class Country(Base):
    __tablename__ = 'countries'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(100), nullable=False,
                                      comment='country name, such as "Canada"')
    code_2: Mapped[Optional[str]] = mapped_column(String(2),
                                                  comment='country 2-alpha code, such
as "CA"')
    code_3: Mapped[Optional[str]] = mapped_column(String(3),
                                                  comment='country 3-alpha code, such
as "CAN"')

    # one country has many states
    states:Mapped[List['State']] = relationship(back_populates='country')


    def __repr__(self):
        return f'<Metadata {self.id} - {self.name})>'

# reference data
class State(Base):
    __tablename__ = 'states'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(100), nullable=False,
                                      comment='state name, such as "Ontario"')

    country_id: Mapped[INTEGER(unsigned=True)] =
mapped_column(ForeignKey('countries.id',
                                                                          ondelete='NO
ACTION',

onupdate='CASCADE'),
                                                               nullable=False,
                                                               comment='fk: references
to countries table')
    region_id: Mapped[Optional[INTEGER(unsigned=True)]] =
mapped_column(ForeignKey('regions.id',

ondelete='NO ACTION',

onupdate='CASCADE'),
                                                              comment='fk: references
to regions table')
    __table_args__ = (
        Index('countryid_name_idx', 'country_id', 'name', unique=True),
    )
    # one state belongs to one region
    region:Mapped['Region'] = relationship(back_populates='states')
    # one state belongs to one country
    country:Mapped['Country'] = relationship(back_populates='states')
    # one state has many cities
    cities:Mapped[List['City']] = relationship(back_populates='state')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.name})>'

# reference data
class City(Base):
    __tablename__ = 'cities'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(100), nullable=False,
                                     comment='city name, such as "Toronto"')

    state_id: Mapped[INTEGER(unsigned=True)] = mapped_column(ForeignKey('states.id',
                                                                       ondelete='NO
ACTION',

onupdate='CASCADE'),
                                                              nullable=False,
                                                              comment='fk: references
to states table')
    __table_args__ = (
        Index('stateid_name_idx', 'state_id', 'name', unique=True),
    )
    # one city belongs to one state
    state:Mapped['State'] = relationship(back_populates='cities')
    # one city has many addresses
    addresses:Mapped[List['Address']] = relationship(back_populates='city')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.name})>'

# master data
class Address(Base):
    __tablename__ = 'addresses'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    address: Mapped[Optional[str]] = mapped_column(String(255),
                                                    comment='address name, such as "52
fairglen ave"')
    postcode: Mapped[str] = mapped_column(String(15), nullable=False,
                                         comment='post code, such as "123123"')

    city_id: Mapped[INTEGER(unsigned=True)] = mapped_column(ForeignKey('cities.id',
                                                                    ondelete='NO
ACTION',

onupdate='CASCADE'),
                                                              nullable=False,
                                                              comment='fk: references
to cities table')
    __table_args__ = (
        Index('cityid_address_postcode_idx', 'city_id', 'address', 'postcode',
unique=True),
    )
    # one address belongs one city
    city:Mapped['City'] = relationship(back_populates='addresses')
    # one address has many customer addresses
    address_customers:Mapped[List['AddressCustomer']] =
relationship(back_populates='address')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.post_code})>'

# master data
class AddressCustomer(Base):
    __tablename__ = 'address_customers'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    customer_id: Mapped[INTEGER(unsigned=True)] =
mapped_column(ForeignKey('customers.id',

ondelete='NO ACTION',

onupdate='CASCADE'),
                                                              nullable=False,
                                                              comment='fk: references
to customers table')
    address_id: Mapped[INTEGER(unsigned=True)] =
mapped_column(ForeignKey('addresses.id',
                                                                       ondelete='NO
ACTION',

onupdate='CASCADE'),
                                                              nullable=False,
                                                              comment='fk: references
to addresses table')
    __table_args__ = (
        Index('ck_customerid_addressid', 'customer_id', 'address_id', unique=True),
    )

    # one address customer reflects one address info
    address:Mapped['Address'] = relationship(back_populates='address_customers')
    # one address customer reflects one customer info
    customer:Mapped['Customer'] = relationship(back_populates='address_customers')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.customer_id})>'

# reference data
class Category(Base):
    __tablename__ = 'categories'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(100),
                                      nullable=False,
                                      comment='category name, such as "Fruit"')

    parent_id: Mapped[Optional[INTEGER(unsigned=True)]] =
mapped_column(ForeignKey('categories.id',

ondelete='NO ACTION',

onupdate='CASCADE'),
                                                              comment='fk: self-
references to categories table')

    # one sub-category belongs to one category
    parent:Mapped[Optional['Category']] = relationship(back_populates='children')
    # one category has many sub-category
    children:Mapped[Optional[List['Category']]] =
relationship(back_populates='parent',

remote_side='Category.id')
    # category can be assignment to many products
    products:Mapped[List['Product']] = relationship(back_populates='category')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.name})>'

# master data
class Product(Base):
    __tablename__ = 'products'
    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    product_no: Mapped[str] = mapped_column(String(15), unique=True,
                                            comment='generated product number, such as
"ABC-SDF-121123"')
    name: Mapped[str] = mapped_column(String(255), nullable=False,
                                      comment='product name, such as "Computer"')
    price: Mapped[int] = mapped_column(Numeric(12, 2), nullable=False,
                                       comment='product price, such as "1233.23"')
    # discount: Mapped[int] = mapped_column(Numeric(4, 2), nullable=False,
    #                                       comment='product discount, such as
"0.23"')
    category_id: Mapped[INTEGER(unsigned=True)] =
mapped_column(ForeignKey('categories.id',

ondelete='NO ACTION',

onupdate='CASCADE'),
                                                               nullable=False,
                                                               comment='fk: references
to categories table')

    # one product belongs to one category
    category:Mapped['Category'] = relationship(back_populates='products')
    # one product belongs to many orders
    # orders:Mapped[List['Order']] = relationship(back_populates='product')
    # one product has many product orders
    product_orders:Mapped[List['ProductOrder']] =
relationship(back_populates='product')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.name}, {self.category_id})>'

# transactional data
class ProductOrder(Base):
    __tablename__ = 'product_orders'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    quantity: Mapped[int] = mapped_column(INTEGER(unsigned=True))
    order_price: Mapped[int] = mapped_column(Numeric(12, 2),
                                             nullable=False,
                                             comment='product price in order, such as
"1233.23"')
    order_discount: Mapped[int] = mapped_column(Numeric(4, 2),
                                                nullable=False,
                                                comment='product discount in order,
such as "0.23"')
```

```
    product_id: Mapped[INTEGER(unsigned=True)] =
mapped_column(ForeignKey('products.id',
                                                                          ondelete='NO
ACTION',

onupdate='CASCADE'),
                                                               nullable=False,
                                                               comment='fk: references
to products table')
    order_id: Mapped[INTEGER(unsigned=True)] = mapped_column(ForeignKey('orders.id',
                                                                        ondelete='NO
ACTION',

onupdate='CASCADE'),
                                                               nullable=False,
                                                               comment='fk: references
to orders table')

    __table_args__ = (
        Index('ck_productid_orderid', 'product_id', 'order_id', unique=True),
    )

    # one product order reflects many products
    product:Mapped['Product'] = relationship(back_populates='product_orders')
    # one product order reflects many orders
    order:Mapped['Order'] = relationship(back_populates='product_orders')

    def __repr__(self):
        return f'<Metadata {self.id} - {self.order_id}, {self.product_id})>'


# this table contains data from sample-superstore.xsl orders sheet
# this table is used to ETL data into different tables
# technically, it's not a part of the project database
class SupserstoreOrder(Base):
    __tablename__ = 'superstore_orders'

    id = mapped_column(INTEGER(unsigned=True), primary_key=True, autoincrement=True)
    row_id: Mapped[int] = mapped_column(INTEGER(unsigned=True))
    product_no: Mapped[str] = mapped_column(String(15), nullable=False,
                                            comment='generated product number, such as
"ABC-SDF-121123"')
    order_no: Mapped[str] = mapped_column(String(50), nullable=False,
                                          comment='generated order number, such as
"ABC-SDF-121123"')
    ship_mode: Mapped[Optional[str]] = mapped_column(String(255))
    customer_no: Mapped[Optional[str]] = mapped_column(String(255))
    customer_name: Mapped[Optional[str]] = mapped_column(String(255))
    segment: Mapped[Optional[str]] = mapped_column(String(255))
    country: Mapped[Optional[str]] = mapped_column(String(255))
    city: Mapped[Optional[str]] = mapped_column(String(255))
    state: Mapped[Optional[str]] = mapped_column(String(255))
    post_code: Mapped[Optional[str]] = mapped_column(String(255))
    region: Mapped[Optional[str]] = mapped_column(String(255))
    category: Mapped[Optional[str]] = mapped_column(String(255))
    sub_cate: Mapped[Optional[str]] = mapped_column(String(255))
    product_name: Mapped[Optional[str]] = mapped_column(String(255))
    sales: Mapped[Optional[int]] = mapped_column(Numeric(12, 2))
    discount: Mapped[Optional[int]] = mapped_column(Numeric(4, 2))
    quantity: Mapped[int] = mapped_column(INTEGER(unsigned=True))
    profit: Mapped[int] = mapped_column(Numeric(12, 2), nullable=False,
                                        comment='product discount in order, such as
"0.23"')
    return_status_id: Mapped[Optional[int]]
    order_at: Mapped[Optional[datetime]]
    ship_at: Mapped[Optional[datetime]]

    def __repr__(self):
        return f'<Metadata {self.id} - {self.order_no}, {self.product_name})>'
```

clean some inconsistent data:

```
import sqlalchemy as sa
import pandas as pd
import os
from sqlalchemy.orm import Session
from src.models.mapped_models import SupserstoreOrder
from src.database import engine as db_engine
from src.constants import ROOT_DIR

data_file = os.path.join(ROOT_DIR, 'data', 'Sample - Superstore.xls')

df_orders = pd.read_excel(data_file, sheet_name='Orders')
df_people = pd.read_excel(data_file, sheet_name='People')
df_returns = pd.read_excel(data_file, sheet_name='Returns')

engine = db_engine.sql_engine()

# update the product name so that it's aligned with product no
# doing this by updating every row of product name with the latest name used
def clean_products_v1():
    stmt_sets = (
        sa.select(SupserstoreOrder.product_no, SupserstoreOrder.product_name,
sa.func.min(SupserstoreOrder.id))
        .group_by(SupserstoreOrder.product_no, SupserstoreOrder.product_name)
        .order_by(SupserstoreOrder.product_no)
    )
    with Session(bind=engine) as session:
        for p_no, p_name, _ in session.execute(stmt_sets):
            update_stmt = (
                sa.update(SupserstoreOrder)
                .where(SupserstoreOrder.product_no == p_no)
                .values(
                    {'product_name': p_name}
                )
            )
            session.execute(update_stmt)
            session.commit()

# add return status id to superstore orders table
def fillin_order_status():
    df_status = df_returns.drop_duplicates()

    with Session(bind=engine) as session:
        for i in range(len(df_status)):
            update_stmt = (
                sa.update(SupserstoreOrder)
                .where(SupserstoreOrder.order_no == df_status.iloc[i, 1])
                .values(
                    {'return_status_id': 1 if df_status.iloc[i, 0].capitalize() ==
'Yes' else 2}
                )
            )
            session.execute(update_stmt)
            session.commit()
```

load data into entities respectively:

```
import os
import pandas as pd
from sqlalchemy.orm import Session
import sqlalchemy as sa

from src.constants import ROOT_DIR
from src.database import engine as db_engine
from src.models import mapped_models as mm
from src.helpers import parse_name, unit_price

data_file = os.path.join(ROOT_DIR, 'data', 'Sample - Superstore.xls')
engine = db_engine.sql_engine()

df_orders = pd.read_excel(data_file, sheet_name='Orders')
```

```
df_people = pd.read_excel(data_file, sheet_name='People')
df_returns = pd.read_excel(data_file, sheet_name='Returns')

# dump orders data into superstore_orders table
def dump_orders_db():
    table_name = 'superstore_orders'
    # pick random post code from city 'Burlington'
    df_orders['Postal Code'] = df_orders['Postal Code'].fillna('52601')
    # print(df_orders.head(2))
    df_orders_insert = df_orders.rename(columns={
            'Row ID': 'row_id', 'Order ID':'order_no', 'Order Date':'order_at', \
            'Ship Date':'ship_at', 'Ship Mode':'ship_mode', \
            'Customer ID':'customer_no', 'Customer Name':'customer_name',\
            'Segment':'segment', 'Country/Region':'country', 'City':'city', \
            'State':'state', 'Postal Code':'post_code', 'Region':'region', \
            'Product ID':'product_no', 'Category':'category', \
            'Sub-Category':'sub_cate', 'Product Name':'product_name', \
            'Sales':'sales', 'Quantity':'quantity', 'Discount':'discount', \
            'Profit':'profit'
    })
    df_order_insert = df_orders_insert.to_dict(orient='records')
    stmt = sa.text(f'INSERT INTO {table_name} (row_id, order_no, order_at, ship_at,\
                ship_mode, customer_no, customer_name, segment, country, city,\
                state, post_code, region, product_no, category, sub_cate, \
                product_name, sales, quantity, discount, profit) \
            values
(:row_id, :order_no, :order_at, :ship_at, :ship_mode, :customer_no,\
                :customer_name, :segment, :country, :city, :state, :post_code, :region
,\
                :product_no, :category, :sub_cate, :product_name, :sales, :quantity, :
discount, :profit)')
    with Session(bind=engine) as session:
        session.execute(stmt, df_order_insert)
        session.commit()

# dump metadata table with descriptions of tables
def insert_metadatas():
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Metadata), [
                {'table_name': 'address_customers', 'column_name': '', 'data_type':
'',
                    'description': 'this table stores customer_id and address_id',
                    'constraints': 'combination of customer_id and address_id is
unqiue key in this table',
                    'relationships': 'references to custoemrs table and addresses
table'},
                {'table_name': 'address_customers', 'column_name': 'id', 'data_type':
'unsigned int',
                    'description': 'primary key of the table',
                    'constraints': 'nullable=false, autoincrement',
                    'relationships': ''},
                {'table_name': 'address_customers', 'column_name': 'customer_id',
'data_type': 'unsigned int',
                    'description': 'foreign key of the table',
                    'constraints': 'nullable=false',
                    'relationships': 'references to customers table'},
                {'table_name': 'address_customers', 'column_name': 'address_id',
'data_type': 'unsigned int',
                    'description': 'foreign key of the table',
                    'constraints': 'nullable=false',
                    'relationships': 'references to addresses table'},
            ],
        )
        session.commit()

# Extract Transforn and Load  country into countries table
def etl_country():
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Country), [
                {'name': df_orders['Country/Region'].unique()[0]}
            ]
        )
        session.commit()

# Extract Transforn and Load  people into employees and regions table
def etl_people():
    df_people['id'] = range(1, 1 + len(df_people))
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Region), [
                {'id': id, 'name': region}
                    for id, region in zip(df_people['id'].tolist(),
df_people['Region'].tolist())
            ]
        )
        session.execute(
            sa.insert(mm.Employee), [
                {'first_name': name.split()[1], 'last_name': name.split()[0],
'region_id': id}
                    for id, name in zip(df_people['id'].tolist(), df_people['Regional
Manager'].tolist())
            ]
        )
        session.commit()

# Extract Transforn and Load state data into states table
def etl_state():
    subq = (
        sa.select(mm.SupserstoreOrder.country, mm.SupserstoreOrder.state,
mm.SupserstoreOrder.region)
        .group_by(mm.SupserstoreOrder.country, mm.SupserstoreOrder.state,
mm.SupserstoreOrder.region)
        .subquery()
    )

    stmt = sa.select(mm.Country.id, subq.c.state,
mm.Region.id.label('region_id')).join_from(
        subq, mm.Country, mm.Country.name == subq.c.country
    ).join(
        mm.Region, subq.c.region == mm.Region.name
    )
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.State), [
                {'name': state, 'country_id': country_id, 'region_id': region_id}
                for country_id, state, region_id in session.execute(stmt)
            ],
        )
        session.commit()

# ETL city data to cities table
def etl_city():
    subq = (sa.select(mm.SupserstoreOrder.state, mm.SupserstoreOrder.city)
        .group_by(mm.SupserstoreOrder.state, mm.SupserstoreOrder.city)
        .subquery()
    )
    stmt = sa.select(mm.State.id.label('state_id'), subq.c.city).join_from(
        subq, mm.State, mm.State.name == subq.c.state
    )

    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.City), [
                {'name': city, 'state_id': state_id}
                for state_id, city in session.execute(stmt)
            ],
        )
        session.commit()

# ETL address data to addresses table
```

```python
def etl_address():
    subq_state = (sa.select(mm.SupserstoreOrder.state,
                            mm.SupserstoreOrder.city,
                            mm.SupserstoreOrder.post_code)
        .group_by(mm.SupserstoreOrder.state,
                  mm.SupserstoreOrder.city,
                  mm.SupserstoreOrder.post_code)
        .subquery()
    )
    subq_city = (sa.select(mm.State.id.label('state_id'),
                           subq_state.c.post_code,
                           subq_state.c.city).join_from(
        subq_state, mm.State, mm.State.name == subq_state.c.state
    ).subquery())
    stmt = sa.select(mm.City.id.label('city_id'), subq_city.c.post_code).join_from(
        subq_city, mm.City, sa.and_(mm.City.name == subq_city.c.city,
                                    mm.City.state_id == subq_city.c.state_id)
    )

    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Address), [
                {'postcode': post_code, 'city_id': city_id}
                for city_id, post_code in session.execute(stmt)
            ],
        )
        session.commit()

# ETL categories into categories table
def etl_category():
    cate_stmt = sa.select(sa.distinct(mm.SupserstoreOrder.category))
    subq = (sa.select(mm.SupserstoreOrder.category, mm.SupserstoreOrder.sub_cate)
        .group_by(mm.SupserstoreOrder.category, mm.SupserstoreOrder.sub_cate)
        .subquery()
    )
    sub_cate_stmt = sa.select(mm.Category.id.label('parent_id'),
subq.c.sub_cate).join_from(
        subq, mm.Category, mm.Category.name == subq.c.category
    )
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Category), [
                {'name': name} for name in session.scalars(cate_stmt)
            ],
        )
        session.commit()
        session.execute(
            sa.insert(mm.Category), [
                {'name': sub_cate, 'parent_id': parent_id}
                    for parent_id, sub_cate in session.execute(sub_cate_stmt)
            ],
        )
        session.commit()

    # print(cate_stmt)

# etl order statuses into table
def etl_order_status():
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.OrderStatus), [
                {'name': 'completed'},
                {'name': 'returned'}
            ],
        )
        session.commit()

# etl segment into table
def etl_segment():
    stmt = sa.select(sa.distinct(mm.SupserstoreOrder.segment))
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Segment), [
                {'name': name} for name in session.scalars(stmt)
            ],
        )
        session.commit()

# etl customers into customers table
def etl_customer():
    subq = (sa.select(mm.SupserstoreOrder.customer_no,
                      mm.SupserstoreOrder.customer_name,
                      mm.SupserstoreOrder.segment)
        .group_by(mm.SupserstoreOrder.customer_no,
                  mm.SupserstoreOrder.customer_name,
                  mm.SupserstoreOrder.segment)
        .subquery()
    )
    stmt = sa.select(mm.Segment.id.label('segment_id'),
                     subq.c.customer_no,
                     subq.c.customer_name).join_from(
        subq, mm.Segment, mm.Segment.name == subq.c.segment
    )
    # print(stmt)
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Customer), [
                {'customer_no': customer_no, 'segment_id': segment_id,
                'first_name': parse_name(customer_name)[0],
                'mid_name': parse_name(customer_name)[1],
                'last_name': parse_name(customer_name)[2]}
                for segment_id, customer_no, customer_name in session.execute(stmt)
            ],
        )
        session.commit()

# etl customer address table
def etl_address_customer():
    with Session(bind=engine) as session:
        q = (session.query(sa.distinct(mm.SupsersoreOrder.customer_no),
                           mm.Customer.id, mm.Address.id)
                .join(mm.Customer, mm.SupsersoreOrder.customer_no ==
mm.Customer.customer_no)
                .join(mm.Address, mm.SupsersoreOrder.post_code ==
mm.Address.postcode)
                .all())

        session.execute(
            sa.insert(mm.AddressCustomer), [
                {'customer_id': customer_id, 'address_id': address_id}
                    for _, customer_id, address_id in q
            ]
        )
        session.commit()

# etl products table
def etl_product():
    subq_id = (sa.select(sa.func.min(mm.SupsersoreOrder.id).label('min_id'))
        .group_by(mm.SupsersoreOrder.product_no, mm.SupsersoreOrder.product_name)
        # .subquery()
    )
    subq_products = (sa.select(mm.SupsersoreOrder.product_no,
                               mm.SupsersoreOrder.product_name,
                               mm.SupsersoreOrder.sub_cate,
                               mm.SupsersoreOrder.sales,
                               mm.SupsersoreOrder.quantity,
                               mm.SupsersoreOrder.discount)
                    .where(mm.SupsersoreOrder.id.in_(subq_id))
                    .subquery()
    )
    stmt = sa.select(mm.Category.id.label('category_id'),
                     subq_products.c.product_no,
                     subq_products.c.product_name,
                     subq_products.c.sales,
                     subq_products.c.quantity,
                     subq_products.c.discount).join_from(
        subq_products, mm.Category, mm.Category.name == subq_products.c.sub_cate
    )
    # print(stmt)
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Product), [
                {'product_no': product_no,
                 'category_id': category_id,
                'name': product_name,
                'price': unit_price(sales, quantity, discount)}
                for category_id, product_no, product_name, sales, quantity, discount
                    in session.execute(stmt)
            ],
        )
        session.commit()

# etl orders table
def etl_orders():
    subq = (sa.select(mm.SupserstoreOrder.customer_no, mm.SupserstoreOrder.order_no,
                      mm.SupserstoreOrder.order_at,
mm.SupserstoreOrder.return_status_id)
                      .group_by(mm.SupserstoreOrder.customer_no,
mm.SupserstoreOrder.order_no,
                      mm.SupserstoreOrder.order_at,
mm.SupserstoreOrder.return_status_id)
                    .subquery()
    )
    stmt = sa.select(mm.Customer.id.label('customer_id'), subq.c.order_no,
                      subq.c.order_at, subq.c.return_status_id).join_from(
        subq, mm.Customer, subq.c.customer_no == mm.Customer.customer_no
    )
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Order), [
                {'order_no': order_no, 'customer_id': customer_id,
                'status_id': 1 if status_id == 1 else 2,
                'order_date': order_date}
                for customer_id, order_no, order_date, status_id in
session.execute(stmt)
            ],
        )
        session.commit()

# ETL product_order table
def etl_product_order():
    subq = (sa.select(mm.SupserstoreOrder.order_no, mm.SupserstoreOrder.product_no,
                      sa.func.sum(mm.SupserstoreOrder.sales).label('sum_sales'),
                      sa.func.sum(mm.SupserstoreOrder.quantity).label('sum_quantity'))
                    .group_by(mm.SupserstoreOrder.order_no,
mm.SupserstoreOrder.product_no)
                    .subquery()
    )
    stmt = sa.select(subq.c.sum_sales, subq.c.sum_quantity,
                     mm.Product.price, mm.Product.id.label('product_id'),
                     mm.Order.id.label('order_id')).join(
        mm.Product, mm.Product.product_no == subq.c.product_no
    ).join(
        mm.Order, mm.Order.order_no == subq.c.order_no
    )
    # print(stmt)
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.ProductOrder),[
                {'quantity': sum_quantity, 'order_price': price,
                 'order_discount': round(1-sum_sales/(sum_quantity*price), 2),
                 'order_id': order_id,
                 'product_id': product_id}
                for sum_sales, sum_quantity, price, product_id, order_id
                    in session.execute(stmt)
            ]

        )
        session.commit()

# etl shipment data
def etl_shipment():
    subq = (sa.select(mm.SupserstoreOrder.order_no, mm.SupserstoreOrder.ship_mode,
                      mm.SupserstoreOrder.ship_at)
                    .subquery()
    )
    stmt = sa.select(subq.c.ship_mode, subq.c.ship_at,
mm.Order.id.label('order_id')).join_from(
        subq, mm.Order, subq.c.order_no == mm.Order.order_no
    )
    with Session(bind=engine) as session:
        session.execute(
            sa.insert(mm.Shipment), [
                {'order_id':order_id,
                 'ship_mode': ship_mode,
                 'ship_date': ship_at}
                for ship_mode, ship_at, order_id in session.execute(stmt)
            ]
        )
        session.commit()
```

[redacted]