E-COMMERCE ORDER MANAGER

A product of Group 5

Bui Tuan Anh

Pham Quang Vu

Nguyen Dinh Thang

Instructor: Dr. Hung Tran

OUR LEARNING GOAL

- **Data modeling:** Transform the unnormalized (UNF) order table into 1NF → 2NF → 3NF using the given Functional Dependencies (FDs); identify and justify Primary Keys (PK) and Foreign Keys (FK).

- **Database implementation**: Create the schema and seed data in MySQL with schema.sql and seed.sql.

- **Application development**: Build a complete Python application with a Graphical User Interface (GUI) that connects to MySQL for CRUD and reporting/visualization.

- **Engineering practice**: Collaborate via GitHub (README, Issues/PRs, release tagging).

- **Communication**: Deliver slides, a LaTeX-based PDF report (no source code in the PDF), and a YouTube demo (unlisted) showing each member's face with name & student ID on screen.

PROBLEM DESCRIPTION & SCOPE

The shop currently stores orders in a single, unnormalized table. This leads to redundancy, update anomalies, and inconsistent reports. You will normalize to 3NF and build an application used by staff to manage customers, products, orders, and order items, run operational queries, and summarize revenue.

MENU

01 Database Design

02 Overview

03 Live demo

04 Conclusion

I. DATABSE DESIGN

INPUT GIVEN

UNF columns

- OrderID
- CustomerID
- CustomerName
- ProductID
- ProductName
- Quantity,
- Price
- OrderDate
- Status

FDs:

- CustomerID → CustomerName
- ProductID → {ProductName, Price}
- OrderID → {CustomerID, OrderDate, Status}
- {OrderID, ProductID} → Quantity

UNF TO 1NF

- 1NF ensures that all attributes contain atomic values (no repeating groups) and each field contains only one value.

- Since the attributes contain multiple values (e.g., multiple products per order), we break it down so that there is only one product per row in the OrderItems table.

OrderItems Table

- OrderID
- CustomerID
- CustomerName
- ProductID
- ProductName
- Quantity,
- Price
- OrderDate
- Status

1NF TO 2NF

- 2NF removes partial dependencies, i.e., attributes that depend only on part of a composite primary key. In this case, we have a composite primary key consisting of OrderID and ProductID.
- To achieve 2NF, we need to break the table into multiple smaller tables to ensure that non-key attributes fully depend on the entire primary key.
- We add a new column UnitPrice to preserve historical price at the time of sale (still make the database 2NF)

2NF TO 3NF

- 3NF removes transitive dependencies, which occur when non-key attributes depend on other non-key attributes

- We examine the existing dependencies and notice:
  - CustomerName depends on CustomerID.
  - ProductName and Price depend on ProductID.

- In this case, there is no transitive dependency within the existing tables, as attributes like CustomerName and ProductName are already placed in separate tables based on their respective keys (CustomerID and ProductID).

FINAL SCHEMA

CUSTOMERS

- CustomerID (PK)
- CustomerName

PRODUCTS

- ProductID (PK)
- ProductName
- Price

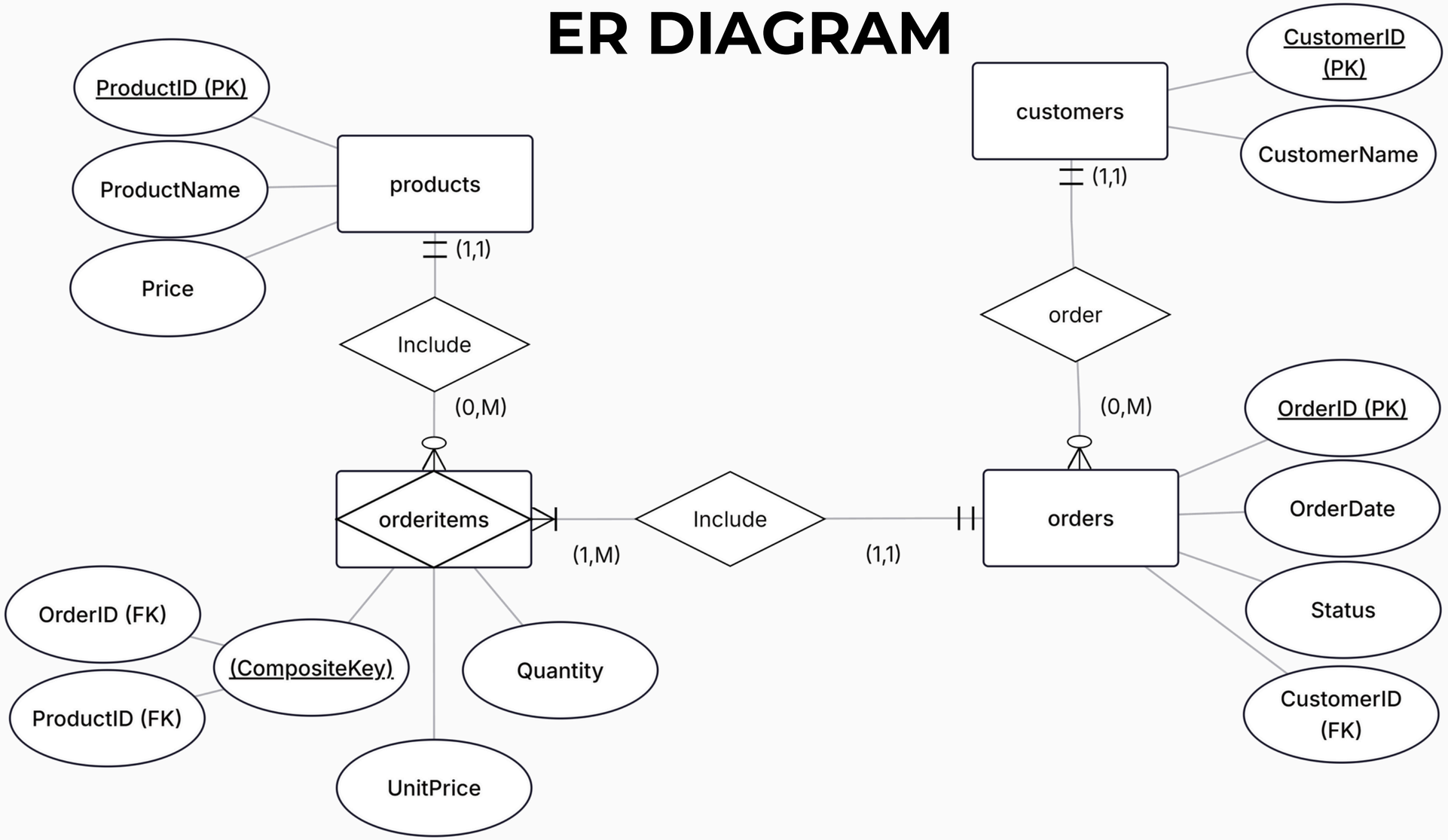
ORDERS

- OrderID (PK)
- OrderDate
- Status
- CustomerID (FK)

ORDERITEMS

- OrderID (PK, FK)
- ProductID (PK, FK)
- Quantity
- UnitPrice

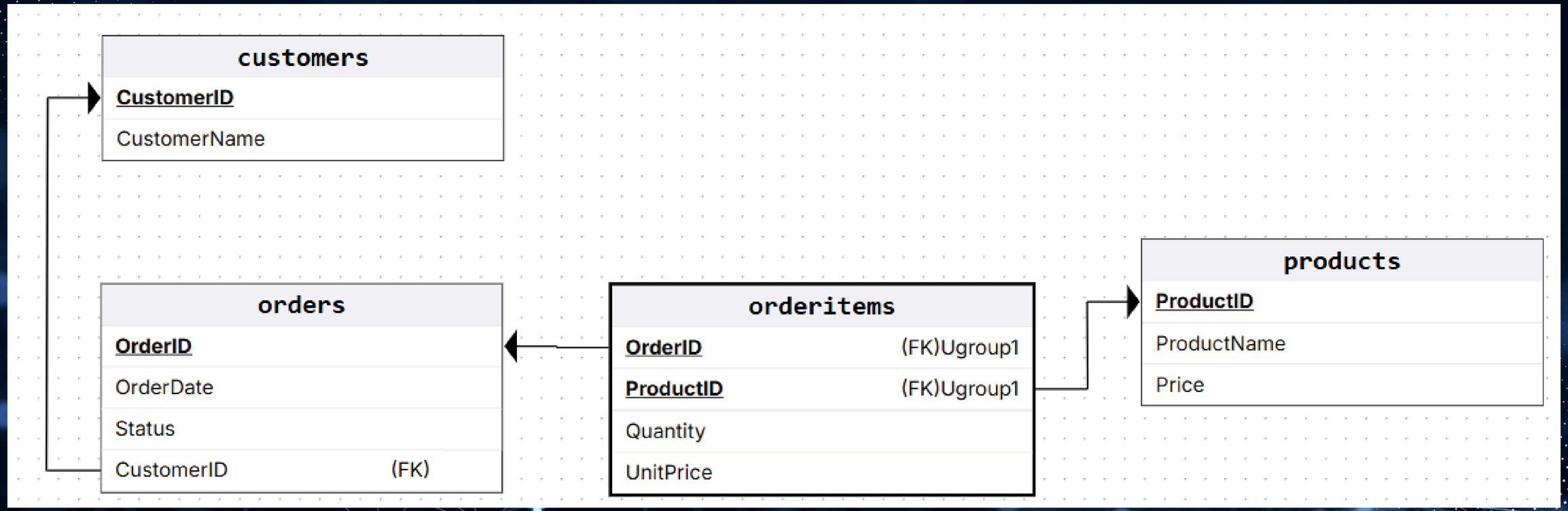
ER DIAGRAM


ProductID (PK)
ProductName
Price
products
(1,1)
Include
(0,M)
orderitems
(1,M)
Include
(1,1)
orders
(0,M)
order
(1,1)
customers
CustomerID (PK)
CustomerName
OrderID (PK)
OrderDate
Status
CustomerID (FK)
OrderID (FK)
(CompositeKey)
ProductID (FK)
Quantity
UnitPrice

ER DIAGRAM

- Note: “orderitems” an is associative entity because it’s keys are composite keys
- Relationships Explained
  - customers → orders (1 to many)
  - orders → orderitems (1 to many)
  - products → orderitems (1 to many)

RELATIONAL SCHEMA



II. OVERVIEW

ARCHITECTURE

App on Desktop: Using Tkinter + PyMySQL

- Tkinter: For UI design, easier to modify on available Python packages
- PyMySQL: to connect to database from MySQL using Python language

ARCHITECTURE

- Consistency: using 'CONSTRAINT' to standardize the format and data type of keys and other fields. It provides the base layer of handling exceptions in SQL.
- Error handling: an additional layer in Python that provides advanced protection and notifications in message box.
- Seed data: simulation of a realistic context, containing 50+ customers, 20 products, 50+ orders, and 100+ order items accross multiple statuses and most recent dates.

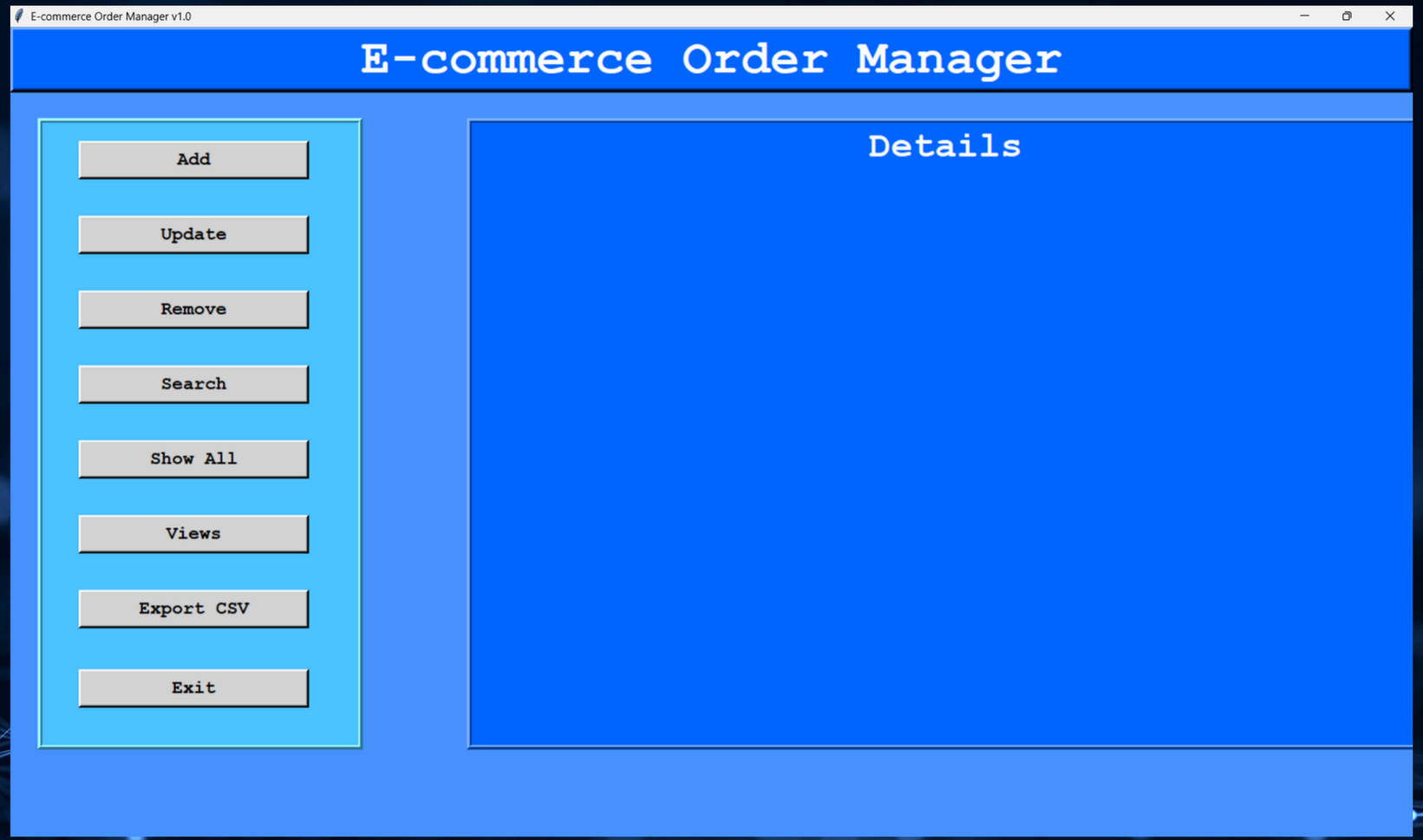
FOLDER LAYOUT OF THE APP

app/

- csv/ # To store exported .csv file
- db/
  - .env # Store connection data
  - connection.py # Reads .env; returns MySQL connection/pool
  - procedures.sql
  - schema.sql
  - seed.sql
- models/ # data access
- queries/ # business logic
- services/ # reusable SQL for reports
- ui/
  - gui.py # GUI source
- main.py # entry point

BASIC GUI

Due to the limited time, we can only design a simple GUI. Still, we ensure to handle all possible outcomes and circumstances to make sure that the package works smoothly without having bugs or errors.



WHAT CAN OUR APP DO?

- Our app is written in Python; therefore, it is easier for inexperienced programmers to understand, use, test, modify, and update for their own project.
- Our data can be stored, created, modified, deleted manually outside the app by using MySQL. This gives everyone free range access of data (as a manager) when the app is malfunction
- Our app has all functions that a manager in our problem needed to work with. They still can modify in the future if new columns are added

III. LIVE DEMO

IV. CONCLUSION

THANK YOU FOR WATCHING