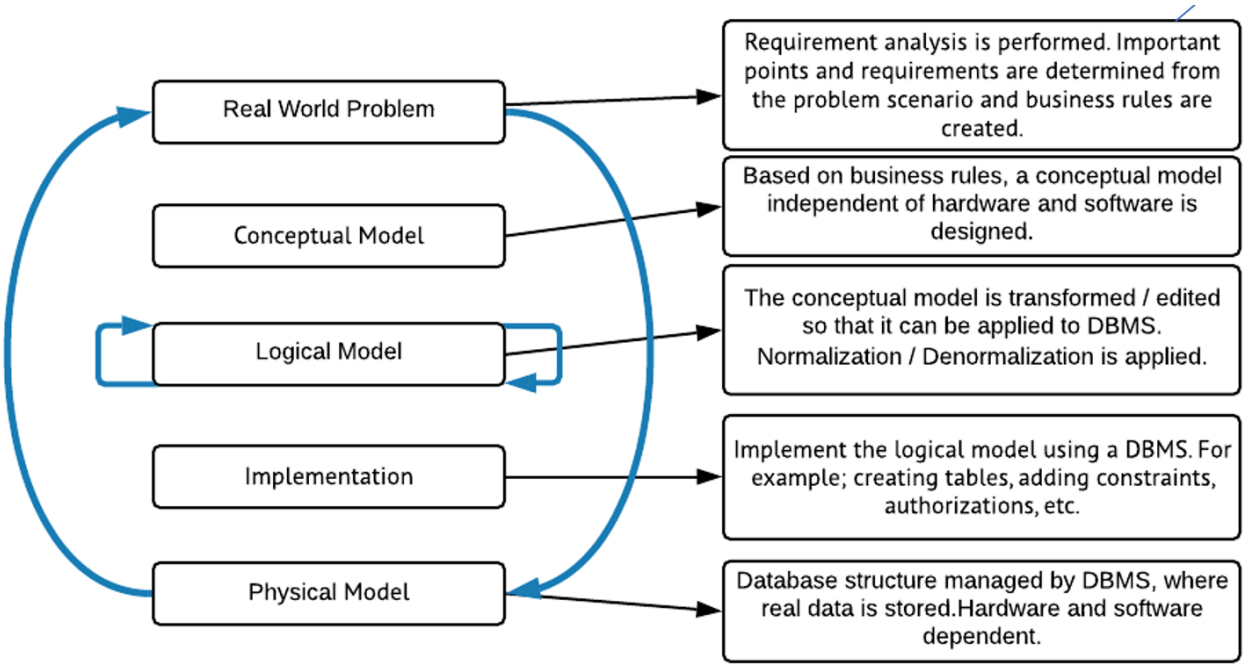
Module 3: Relational Database Model

Database Development Lifecycle

When developing databases, we need to adhere to the **Database Development Lifecycle.** It consists of several stages to build a well-structured database:



- **Requirements Analysis**: Understanding business needs and defining **business rules**.
- **Conceptual Design**: Designing high-level Entity-Relationship (ER) models.
- **Logical Design**: Transforming ER models into relational schemas.
- **Implementation**: Building and configuring the database using SQL

1. Fundamentals of the Relational Model

The relational model organizes data into tables (relations) consisting of rows (tuples) and columns (attributes).

- **Relation**: A table with rows and columns.
- **Tuple**: A single row in a table representing a record.
- **Attribute**: A column in a table representing a specific data field.
- **Domain**: The set of valid values for an attribute.

- **Relational Schema**: Describes the structure of a table, including the names and data types of attributes, PK, and FK .

  Product Table Relational Schema:

  Product
  ProductID: INT, **PRIMARY KEY**, Unique identifier for each product.
  ProductName: VARCHAR(255), **NOT NULL**, The name of the product.
  Price: DECIMAL(10, 2), **NOT NULL**, The cost of the product.
  StockQuantity: INT, **NOT NULL**, The number of products available in stock.

Why Relational Model?

- Simple and intuitive structure.
- Ensures data integrity and reduces redundancy.
- Supports powerful querying using SQL.

2. Table (Relation) Structure

- **Table Name**: Unique identifier for the table.
- **Columns**: Define the attributes (e.g., `StudentID`, `Name`, `Age`).
- **Rows**: Represent individual records.

- **Product** Table

ProductID (PK)	ProductName	Price	StockQuantity
1	SSD 512GB	60	120
2	16GB DDR4 RAM	75	150
3	Intel i7 CPU	300	80
4	1TB HDD	50	200
5	750W PSU	85	100

- **ProductID (Primary Key)**: A unique identifier for each product.
- **ProductName**: The name of the product.
- **Price**: The cost of the product.
- **StockQuantity**: The number of products available in stock.

3. Integrity Rules (Constraints)

Integrity rules help maintain data accuracy and consistency.

- **Primary Key (PK)**:
  - Uniquely identifies each row in a table.
  - Cannot be **NULL**.
  - Examples: ProductId in the Product table, StudentID in the Students table.
- **Foreign Key (FK)**:
  - Establishes a relationship between two tables.
  - Ensures referential integrity.
- **Unique Constraint**:
  - Ensures all values in a column are unique.
  - Example: Email in the Users table.
- **Not Null Constraint**:
  - Ensures a column cannot have NULL values.
  - Example: Name in the Course table.

Referential Integrity and FK

Category Table

CategoryID (PK)	CategoryName
1	Storage Devices
2	Memory
3	Processors
4	Power Supplies

Products Table

ProductID (PK)	ProductName	Price	StockQuantity	CategoryID (FK)
1	SSD 512GB	60	120	1
2	16GB DDR4 RAM	75	150	2
3	Intel i7 CPU	300	80	3
4	1TB HDD	50	200	1
5	750W PSU	85	100	4

- **Referential Integrity** ensures that the relationship between tables remains consistent.
- **CategoryID** in the **Products** table is a **Foreign Key** that refers to **CategoryID** in the **Category** table.
    - This means every value in the **Products.CategoryID** column must exist in the **Category.CategoryID** column. It can also be **NULL**.
    - For example, the product **SSD 512GB** has a CategoryID **=** 1, which corresponds to **Storage Devices** in the **Category** table.

What Happens Without Referential Integrity?

1. **Inserting Invalid Data**: If you try to add a product with `CategoryID = 5`, it will fail because `CategoryID = 5` doesn't exist in the **Category** table.
2. **Deleting a Category**: If you delete the **Storage Devices** category, the database will prevent it because products like **SSD 512GB** and **1TB HDD** depend on that category.

Best Practices:

When designing databases, always use foreign keys to enforce referential integrity.

This prevents orphaned records and ensures the data remains **reliable** and **consistent**.

4. Best Practices for Primary Keys

- Keep primary keys **compact** and **simple** to increase database performance(search and constraint enforcement). Prefer **integer** types.
    - For example; Instead of using a string-based primary key like `ProductCode = "SSD512GB"`, use a compact integer key: `ProductID = 1`. This makes operations more efficient and scalable.
- Avoid using meaningful data (e.g., names) as primary keys.
- Ensure primary keys are **immutable** (do not change over time).
- Avoid using **business data** (e.g., SSNs) as primary keys to ensure data privacy.
- Use **surrogate keys** (e.g., auto-incremented integers like **id**) instead of natural keys (e.g., SSN, StudentNumber,etc.).

5. Indexes and Their Importance

Index in a table is a data structure that **improves the speed of data retrieval operations** on a database table. Indexes can speed up SELECT queries and improve performance for complex queries involving joins or WHERE clauses, but this is not always the case.

Indexes can result in **slower INSERT, UPDATE, and DELETE operations** due to the need for index maintenance..

Student Table

RowNumber	StudentID (PK)	FirstName	LastName	Age	Major
1	1	John	Doe	20	Computer Sci
2	2	Jane	Smith	21	Business
3	3	John	Doe	22	Engineering
4	4	Alice	Brown	19	Mathematics
5	5	John	Doe	23	Physics
6	6	Jane	Smith	24	Economics

Index Table for FirstName (with Unique First Names and Row Numbers)

FirstName	Location
John	1, 3, 5
Jane	2, 6
Alice	4

An **index scan** is a process where the **DBMS** uses an **index** to quickly locate data in a table without having to scan the entire table. It is especially useful for speeding up query performance, especially when searching for specific values or ranges in indexed columns.

A **non-index scan**, often referred to as a **sequential scan**, is a process where the **DBMS** scans **all rows in a table** to satisfy a query, rather than using an index to directly locate specific rows. In a sequential scan, the **DBMS** scans the entire table row by row and checks each row to determine if it satisfies the query condition.

Types of Indexes:

1. Primary Index

- Constructed automatically for the **Primary Key**.
- Ensures **unique and non-null** values for the primary key column.
- Example: **`ProductID`** in a **Products** table.

2. Unique Index

- Ensures that all values in the indexed column are **unique**.
- Unlike the primary index, it allows **one NULL value** (depending on the database).
- It can be applied to any column to enforce uniqueness without making it a primary key.
- **Use Case:** Email addresses in a **Users** table, where each email must be unique.

3. Secondary Index (Non-unique Index)

- An additional index defined on non-primary key columns to improve query performance.
- It allows **duplicate values**.
- **Use Case:** Defining an index on the ProductName column in the Products table to speed up searches by product name.

4. Full-text Index (full text search)

- Specialized index for searching text data efficiently (supported in databases like PostgreSQL, MySQL, etc.).
- **Use Case:** Searching keywords in a **blog posts** table.
    -
- **Benefits**:
    - Faster **SELECT** queries.
    - Improved performance for complex queries involving joins or WHERE clauses.

6. System Catalog and Metadata

- **System Catalog**: A collection of tables that store metadata (data about data). It provides information about database objects like tables, views, indexes, users, etc.
- Managed by the Database Management System.
- Metadata (tables, table fields, field types, value ranges, keys, indexes, relationships, constraints, etc.) for all databases are stored here.
- Users are allowed to perform limited read-only queries on the system catalog and metadata tables, depending on their access privileges.

Hands-on Exercise 1: E-Commerce System Design - From Business Rules to Relational Model

Objective

In this exercise, you will practice database modeling by converting business rules into an Entity-Relationship (ER) model using Crow's Foot notation, and then constructing a relational model.

Task 1: Analyze Business Rules

Below are several business rules describing the structure of an e-commerce database. Carefully analyze them and identify the necessary entities, attributes, and relationships:

1. A **customer** can place **many orders**, but each order is placed by **one customer**.
2. Each **customer** has a **unique customer ID**, **name**, **email**, **phone number**, and **shipping address**.
3. An **order** has a **unique order ID**, an **order date**, a **total amount**, and a **status** (e.g., pending, shipped, delivered).
4. An **order** can contain **multiple products**, and a **product** can be part of multiple orders. The quantity of each product in an order must be recorded.
5. Each **product** has a **unique product ID**, **name**, **description**, **price**, and **stock quantity**.
6. A **product** belongs to **one or more categories**, and each **category** can have multiple products.
7. The system tracks **payments** made by customers. A **payment** is associated with **one order**, and an order can have **only one payment**.
8. Each **payment** has a **unique payment ID**, **payment date**, **amount paid**, and **payment method** (e.g., credit card, PayPal).
9. An **order** may have one or more **shipments**, but each **shipment** must be associated with exactly one order.

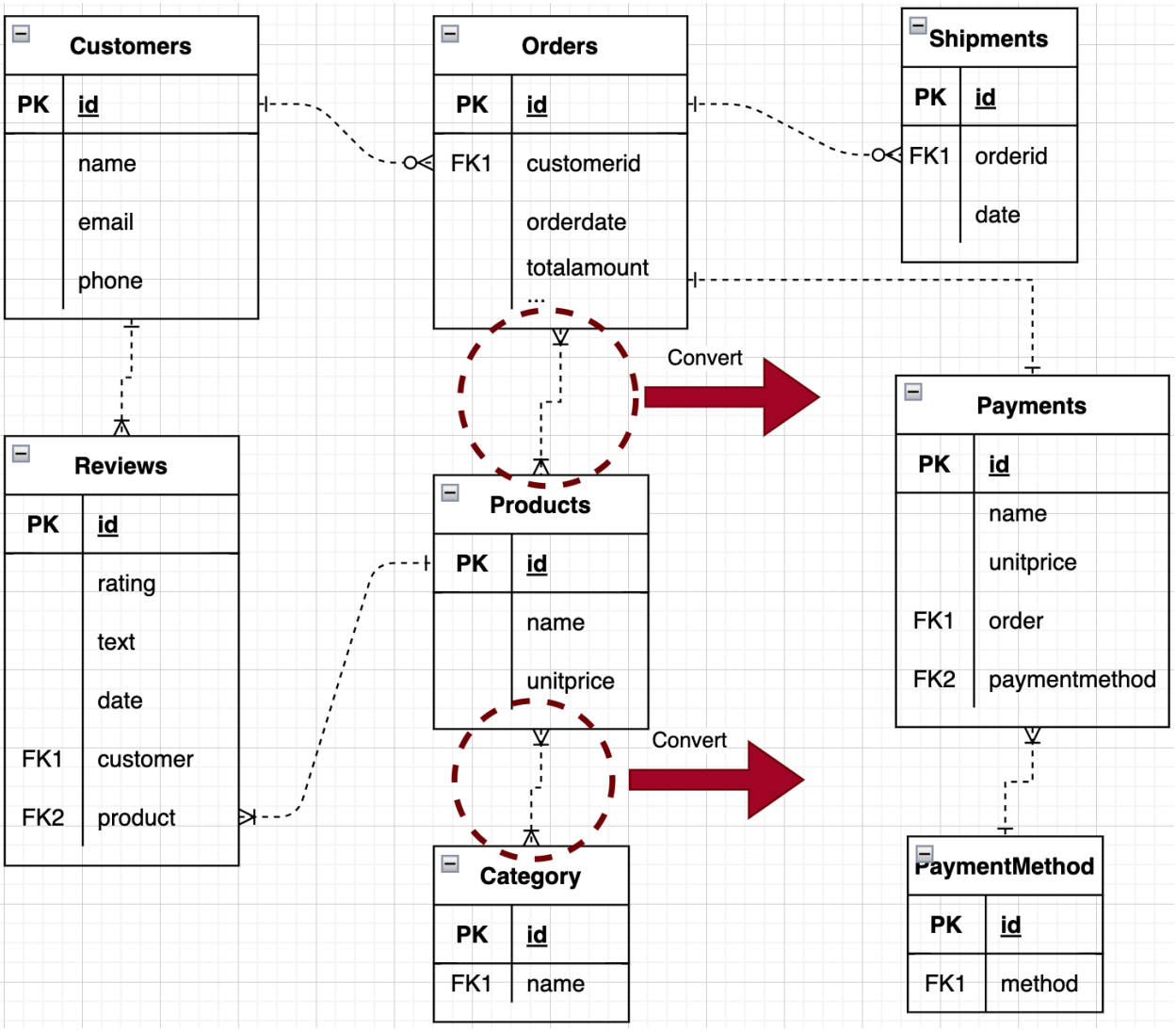
10. Customers can leave **reviews** for products. A **review** belongs to **one customer** and **one product**.
11. Each **review** has a **review ID**, a **rating** (1 to 5), **review text**, and a **date**.

Task 2: Convert Business Rules into an ER Model

Using Crow's Foot notation, design an ER model that represents the above business rules. Your model should include:

- Entities and their attributes
- Relationships between entities
- Relationship types (cardinalities) (one-to-one, one-to-many, many-to-many)

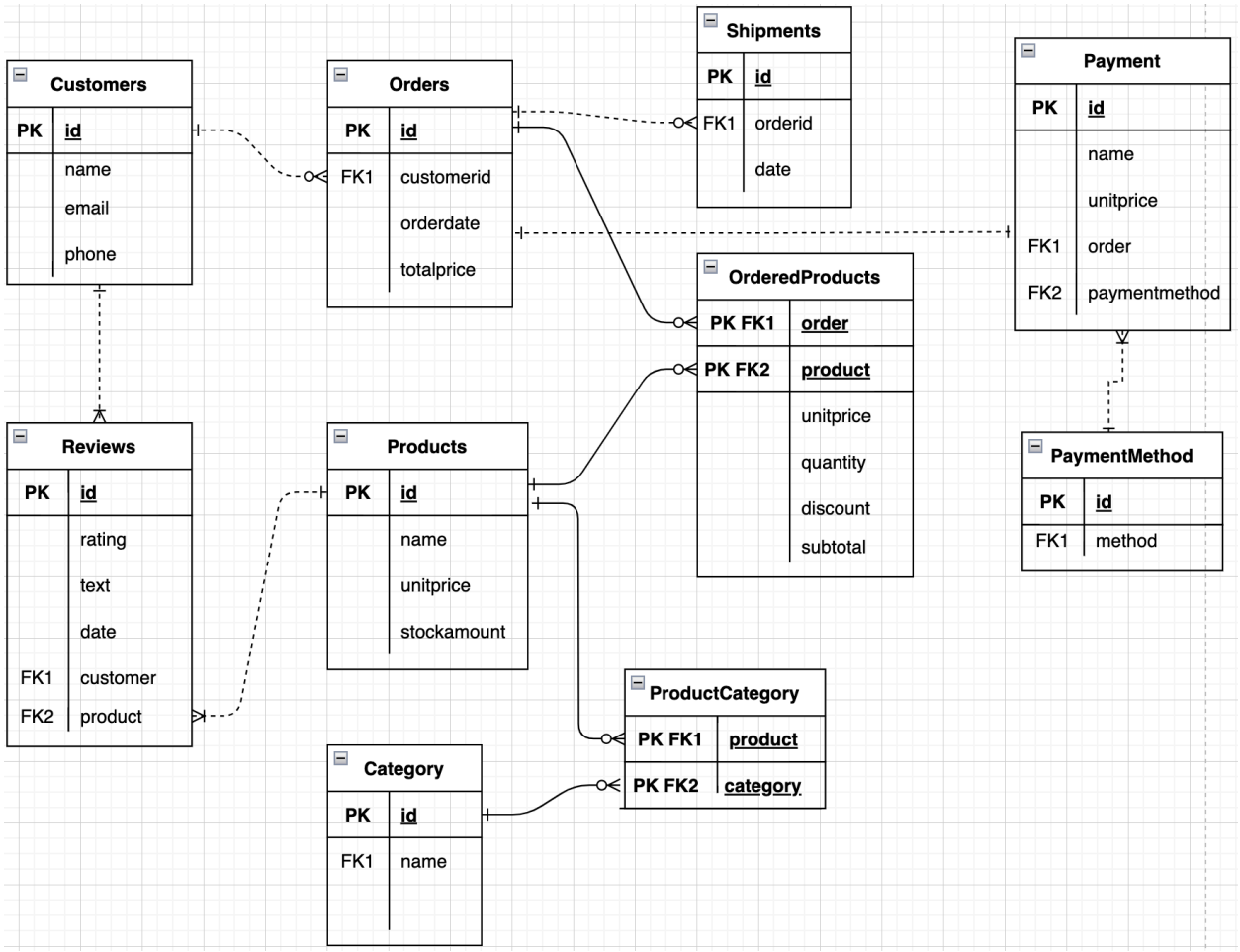
ER Diagram-V1



ER Diagram-V2

The **OrderedProducts** table stores all information related to the association between **Products** and **Orders**, such as product IDs, order IDs, quantities, unit prices, etc.

The **ProductCategory** table stores all information related to the association between **Products** and **Category**.



ER Diagram-V3

We need a data model that tracks individual products ordered by customers. For this we can use a table similar to the following- **ProductSale**.

ProductSale

product_id	name	description	price	serial_number	customer
2001	Alpha Phone X	128GB, Midnight Gray	750.00	SN100001	Customer 01
2001	Alpha Phone X	128GB, Midnight Gray	750.00	SN100002	Customer 02
2001	Alpha Phone X	128GB, Midnight Gray	750.00	SN100003	Customer 03
2002	VegaBook Pro	512GB SSD, 16GB RAM	1200.00	SN200001	Customer 04
2002	VegaBook Pro	512GB SSD, 16GB RAM	1200.00	SN200002	Customer 05
2003	Orion Pad	10.5" Display, 128GB	600.00	SN300001	Customer 06
2003	Orion Pad	10.5" Display, 128GB	600.00	SN300002	Customer 07
2004	Nova Ultra	256GB, Silver	850.00	SN400001	Customer 08
2004	Nova Ultra	256GB, Silver	850.00	SN400002	Customer 09
2005	Quantum Station	1TB SSD, 32GB RAM	1500.00	SN500001	Customer 10

In the first table (ProductSale), which contains repeating data, there is an embedded table within it. Data repetition leads to wasted space and can result in anomalies during insert, update, and delete operations. These anomalies can compromise data integrity, leading to incorrect or inconsistent data. The solution is to extract the embedded table into a separate table and establish a relationship between these tables.

Normalization is the process of organizing a relational database to reduce redundancy and improve data integrity. This operation is part of that process.

Product Sale

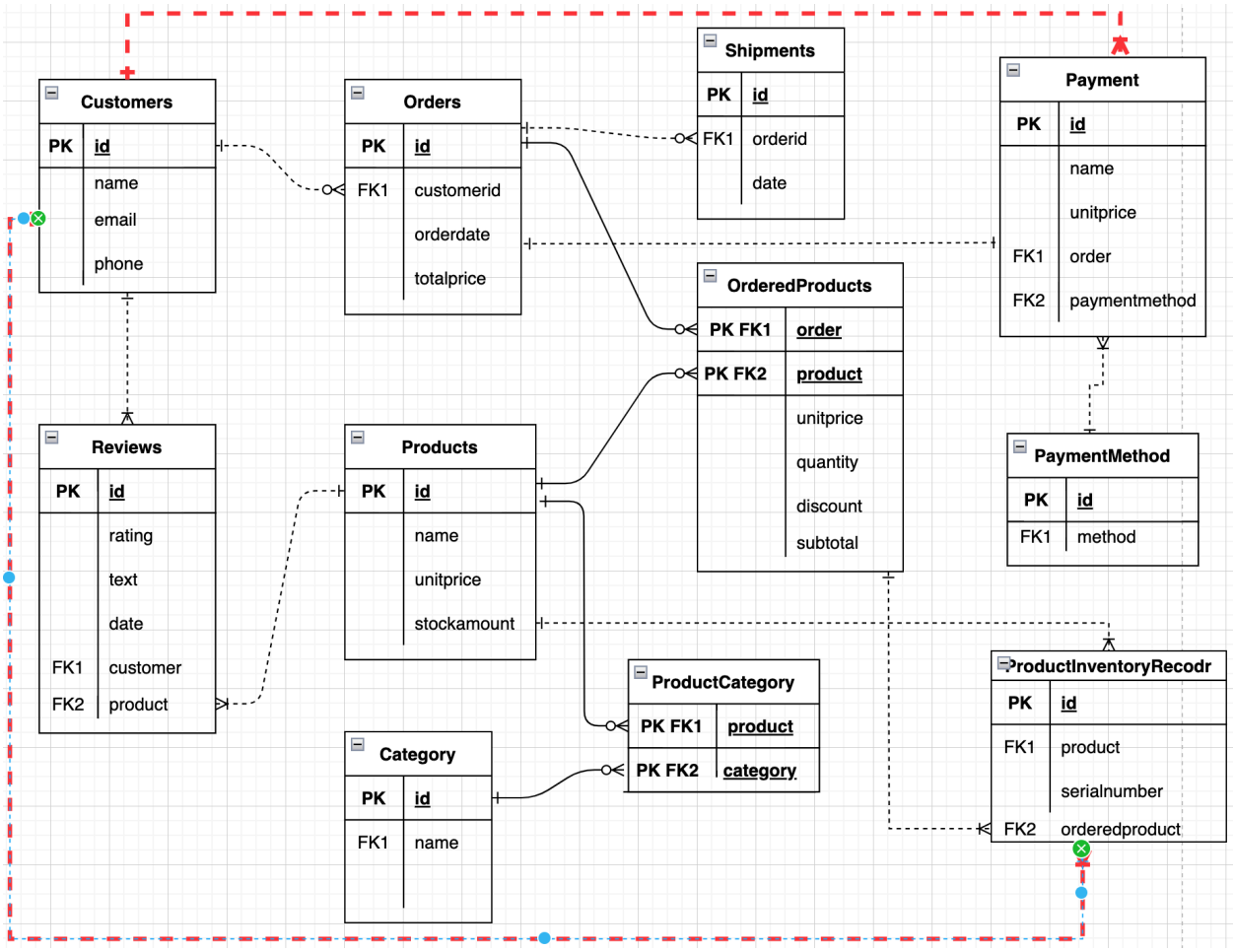
product_id	name	description	price	serial_number	customer
2001	Alpha Phone X	128GB, Midnight Gray	750.00	SN100001	Customer 01
2001	Alpha Phone X	128GB, Midnight Gray	750.00	SN100002	Customer 02
2001	Alpha Phone X	128GB, Midnight Gray	750.00	SN100003	Customer 03
2002	VegaBook Pro	512GB SSD, 16GB RAM	1200.00	SN200001	Customer 04
2002	VegaBook Pro	512GB SSD, 16GB RAM	1200.00	SN200002	Customer 05
2003	Orion Pad	10.5" Display, 128GB	600.00	SN300001	Customer 06
2003	Orion Pad	10.5" Display, 128GB	600.00	SN300002	Customer 07
2004	Nova Ultra	256GB, Silver	850.00	SN400001	Customer 08
2004	Nova Ultra	256GB, Silver	850.00	SN400002	Customer 09
2005	Quantum Station	1TB SSD, 32GB RAM	1500.00	SN500001	Customer 10

Inventory Record

product_sale_id	product_id	serial_number	customer
1	2001	SN100001	Customer 01
2	2001	SN100002	Customer 02
3	2001	SN100003	Customer 03
4	2002	SN200001	Customer 04
5	2002	SN200002	Customer 05
6	2003	SN300001	Customer 06
7	2003	SN300002	Customer 07
8	2004	SN400001	Customer 08
9	2004	SN400002	Customer 09
10	2005	SN500001	Customer 10

Products

product_id	name	description	price
2001	Alpha Phone X	128GB, Midnight Gray	750.00
2002	VegaBook Pro	512GB SSD, 16GB RAM	1200.00
2003	Orion Pad	10.5" Display, 128GB	600.00
2004	Nova Ultra	256GB, Silver	850.00
2005	Quantum Station	1TB SSD, 32GB RAM	1500.00



Task 3: Transform the ER Model into a Relational Model and obtain the relational schema.

- **Product Table Relational Schema:**

  Products
  id: INT, **PRIMARY KEY**,
  name: VARCHAR(255), **NOT NULL**, The name of the product.
  unitprice: DECIMAL(10, 2), **NOT NULL**, The cost of the product.
  stockamount: INT, **NOT NULL**, The number of products available in stock.

- **Customer Table Relational Schema:**

  Customers
  id: INT, **PRIMARY KEY**,
  name: VARCHAR(255), **NOT NULL**, The name of the product.
  email: VARCHAR(255), **NOT NULL**, The name of the product.
  phone: VARCHAR(255), **NOT NULL**, The name of the product.

  …

Additional Challenges (Optional)

These tasks are for students to implement at home:

- Investigate and propose potential index definitions for the fields of the table.