Objective

To design and implement a new relational database for a small corner shop the sells a range of groceries and domestic products,

Proposed Solution

Understanding the business requirements

The data that needs to be stored by the business is data regarding the products, sales and customer information. Data about the products should include current stock of the product, a product description (name and brand), pricing, as well as the category of the product. For example, fruit and veg or meats. The sales data should include the number of times a product has been sold to enable the tracking of stock levels, this would also allow calculations to work out the total sales from the number of stock sold times their price. Data for customers should include customer information such as Name, contact details such as a number or email to keep in contact and ID for the loyalty program that the business wants to set up.

The user of the database will most likely be the owner of the business who oversees most things such as stock management as well as managing pricing of products. Their job would also be to update the sales table and any relevant information to keep track of stock sold and see performance numbers for their profit.

Designing the Database Schema

For structure I would create 5 tables

1. Customers Table

We need to store customer details like personal information and loyalty program data.

- customer_id (Primary Key): A unique identifier for each customer.
- first_name
- last_name
- email: Used for communication or promotional purposes.
- phone_number
- loyalty_points: Tracks how many loyalty points the customer has earned.
- date_joined: When they joined the store's loyalty program.

This table is focused on customer demographics and loyalty data.

2. Products Table

This is where we track the inventory of products, including stock quantities, price, and product details.

- product_id (Primary Key): Unique identifier for each product.
- product_name
- category: Categorizes the product (e.g., groceries, beverages, domestic products).
- description: A description of the product.
- price: The unit price of the product.
- stock_quantity: Tracks how many units are available.
- supplier_id (Foreign Key, optional): If the store buys products from suppliers, this would link to a Supplier table.

3. Sales Table

This table captures overall sales transactions. Each sale will link to a customer (if applicable), and track the total amount.

- sale_id (Primary Key): A unique identifier for each sale.
- customer_id (Foreign Key to Customers): This links each sale to a customer.
- sale_date: The date and time when the sale took place.
- total_amount: The total price for this sale transaction.

4. Sale Details Table

This table records each individual product sold as part of a sale. This allows us to track the line items in a sale.

- sale_detail_id (Primary Key): Unique identifier for each line item.
- sale_id (Foreign Key to Sales table): Links to the specific sale.
- product_id (Foreign Key to Products): Links to the product sold.
- quantity: How many units of the product were sold.
- price_per_unit: The price of the product at the time of sale.

5. Inventory Transactions Table

This table tracks inventory adjustments, such as restocking, returns, or sales-related stock changes.

- transaction_id (Primary Key): Unique identifier for each inventory adjustment.
- product_id (Foreign Key to Products table): Links to the product whose stock is adjusted.
- transaction_type: Type of transaction (e.g., "restock," "sale," "return").
- quantity: The number of units affected.
- transaction_date: When the transaction occurred.

To clearly display the relationships we need to establish foreign key relationships between the tables. Here's how the relationships should be structured:

1. Customer ↔ Sales

- One-to-many relationship: A customer can make multiple purchases (sales), but each sale can only belong to one customer (unless it's a walk-in, in which case we can leave the customer field null).
- How to implement: In the Sales table, the customer_id field will be a foreign key that references the Customers table.

2. Sales ↔ Sale Details

- One-to-many relationship: A sale can consist of multiple products (items), and each sale detail represents one product in the sale. This captures the individual products bought within one sale transaction.
- How to implement: The Sale Details table will have a sale_id field that links to the Sales table and a product_id field that references the Products table.

3. Sale Details ↔ Products

- Many-to-one relationship: Each sale detail line item corresponds to one specific product, but a product can appear in multiple sale transactions.
- How to implement: In the Sale Details table, the product_id field is a foreign key referencing the Products table.

4. Products ↔ Inventory Transactions

- One-to-many relationship: Each product can have multiple inventory adjustments (e.g., restocks, sales, returns).

- How to implement: In the Inventory Transactions table, the product_id field is a foreign key that links to the Products table.

5. Customers ↔ Loyalty Program

- One-to-one or one-to-many relationship: If the loyalty program is linked directly to customers (each customer has one loyalty record), we would use a one-to-one relationship. If loyalty points can accumulate across different programs or promotions, it could be one-to-many.

- How to implement: The Loyalty Program table (if used) will have a customer_id foreign key linking it to the Customers table.

6. Products ↔ Suppliers (Optional)

- One-to-many relationship: A supplier can provide multiple products, but each product typically comes from only one supplier.

- How to implement: The Products table can have a supplier_id field that links to the Suppliers table, if suppliers are part of your business model.

Visualizing the Relationships

Here's a simple breakdown of the relationships:

- Customers ↔ Sales → A customer can have multiple sales (1:N).
- Sales ↔ Sale Details → Each sale has multiple items (1:N).
- Sale Details ↔ Products → Each line item refers to one product (M:1).
- Products ↔ Inventory Transactions → Each product has many inventory adjustments (1:N).
- Customers ↔ Loyalty Program → Each customer can have a loyalty record (1:1 or 1:N).
- Products ↔ Suppliers → A product is linked to one supplier (M:1, optional).

Database Implementation

Implementing the Database by the following SQL commands:

```
CREATE TABLE Customers ( Customer_Id INT PRIMARY KEY AUTO_INCREMENT,

First_Name VARCHAR(50),

Last_Name VARCHAR(50),
```

email VARCHAR(100) UNIQUE,

Phone_Number VARCHAR(15),

Address VARCHAR(20) ,

Loyalty Points INT,

Date_Joined DATETIME DEFAULT CURRENT_TIMESTAMP);

Inerting values into Customers Table:

INSERT INTO Customers (First_Name, Last_Name, email, Phone_Number, Loyalty_Points, Date_Joined) VALUES ('John', 'Doe', 'john.doe@example.com', '555-1234', 150 ,”22-01-2025”), ('Daniel', 'Smith', 'jane.smith@example.com', '555-5678', 230, ,”23-01-2025”), ('George', 'Jone', 'alice.johnson@example.com', '555-9876', 320, ,”02-02-2025”);

CREATE TABLE Products(Product_Id PRIMARY KEY AUTO_INCREMENT,

P_Name VARCHAR(20) NOT NULL,

P_Desc TEXT,

Price DOUBLE NOT NULL,

Quantity_in_stock INT NOT NULL,

Category VARCHAR(100),

Supplier_Id INT FOREIGN KEY (supplier_id) REFERENCES Suppliers(supplier_id

);

Inserting into Products Table:

INSERT INTO Products (Name, Brand, Category, Description, Price, Stock) VALUES ('Apple', 'GoGreen', 'Fruit', 'Fresh British Apples', 1.00, 100, 001), ('Chicken Breast', 'UKForm', 'Meat', 'Boneless chicken', 5.00, 50,002) (‘Carrots’, ‘FreshVeg’, ‘Vegetable’ , ‘Organic Carrots’, 2.00, 30,003) ;

Creating Sales Table:

CREATE TABLE Sales (Sale_Id INT PRIMARY KEY AUTO_INCREMENT,

Customer_Id INT,

Sale_Date DATETIME DEFAULT CURRENT_TIMESTAMP,

Total_Amout DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (Customer_Id) REFERENCES Customers(Customer_Id)

INSERT INTO Sales (customer_id, total_amount)VALUES (1, 25.50),(2, 15.00), (3, 50.75);

Sale_ID field is auto-incremented, so you don't need to manually insert values for it

Customer_ID refers from the Customers table. here assuming id's are 1,2,3

Sale_Date - automatically set to the current timestamp

CREATE TABLE SaleDetails (Sale_Detail_Id INT PRIMARY KEY AUTO_INCREMENT,

Sale_Id INT,

Product_Id INT, quantity INT NOT NULL,

Price_Per_Unit DECIMAL(10, 2) NOT NULL,

FOREIGN KEY (Sale_Id) REFERENCES Sales(Sale_Id),

FOREIGN KEY (Product_Id) REFERENCES Products(Product_Id));

INSERT INTO SaleDetails (sale_id, product_id, quantity, price_per_unit) VALUES (1, 1, 5, 1.50), (1, 2, 2, 0.99) (2,2,3,.89);

Populating the Database

To input initial data into the database, we use the INSERT INTO statement as show in the former section, We also recommend to use transaction to ensure all updates succeed or none at all. For example, the Loyalty Points and Sales record could be updated as shown below:

START TRANSACTION;

UPDATE Customers

SET loyalty_points = loyalty_points + 5

WHERE customer_id = 2;

```
UPDATE SalesTransactions
SET discount_applied = 5.00
WHERE transaction_id = 101;

COMMIT;
```

Database Maintenance

a. Ensuring Database Accuracy and Up-to-Date Information

To maintain accuracy and keep the database updated, I would implement the following measures:

1. Data Validation and Constraints

Use NOT NULL, CHECK, and UNIQUE constraints to prevent invalid entries.

For Example: Ensuring stock quantities are always non-negative:

2. Foreign Key Constraints

- Establish FOREIGN KEY constraints with ON DELETE CASCADE or SET NULL to maintain data integrity when related records are deleted.

For Example: If a supplier is removed, its products are set to NULL instead of being orphaned.

3. Role-Based Access Control (RBAC)

- Restrict access based on user roles:
  - Cashiers: Can only add sales transactions.
  - Managers: Can update stock, generate reports.
  - Owners/Admins: Have full database control.
- Implement user authentication using hashed passwords.

b. Handling Backups and Data Security

1. Regular Database Backups

- Schedule daily incremental backups and weekly full backups using Database tools:

- Arrange those backups being stored locally and remotely.

2. Implement User Access Controls

- Create separate database users with limited privileges:
- Use two-factor authentication (2FA) for managers and owners.

3. Encryption for Sensitive Data

- Hash passwords before storing them