Linh Nguyen
COM303
Professor Peitzsch
May 10, 2024

Final Project

I work with Uyen Tran on this final project for the design and implementation of the MUJI USA database.

About MUJI

MUJI, a Japanese company founded in 1980, has made a niche for itself by offering a diverse range of high-quality yet affordable products, spanning household goods, apparel, and food items. The name "Mujirushi Ryohin" translates to "no-brand quality goods," reflecting the brand's core philosophy of providing simple, functional, and well-designed products without unnecessary frills or branding. MUJI's approach is built on three unwavering principles: meticulous selection of materials, streamlining of processes, and simplification of packaging. Despite their minimalist aesthetic, MUJI's products are not merely minimalistic but rather prioritize quality, practicality, and affordability, resonating with consumers who value these attributes.

Business Rules That Influenced The Database Design

In designing the database for MUJI's operations, we implement some key business rules to align with their business. Firstly, as MUJI manufactures their products in-house and manages their own stores directly, we won't need to accommodate vendors or supply chain in the database schema. Secondly, MUJI's philosophy of "Simplicity and emptiness yield the ultimate universality" requires a flexible database structure that can accommodate a wide range of product categories while maintaining a consistent simplicity. This involves a hierarchical approach to product classification, allowing for expansion into new categories.

Our database design also incorporates several technical business rules. Each product is assigned a Universal Product Code (UPC) for external identification and an internal ID for internal control. Additionally, we utilize an attribute to track whether a product is currently in storage or not. When a product needs to be discontinued, instead of deleting it entirely, we simply update the attribute to reflect its discontinued status. This approach allows us to preserve the sales records associated with previous products, for future analysis and business decisions.

Challenges

One of the main challenges we faced was organizing different products into categories while maintaining a general **Product** entity for relationships with other entities such as **Store** or **Transaction**. This will be discussed in more details along with other designing issues. During implementation of this database, we also encountered some minor challenges that required us to redesign the original schema.

Another issue that arises was data generation. We utilized ChatGPT for this task and while it is efficient, there are two problems that come with it. Initially, the generated data contained inconsistencies, such as different prices for the same product and store within the **Transaction** table. Additionally, the data lacked variation, with each customer purchasing the same quantity of items in ascending order. Our solution was to manually go through each entry to check for inconsistencies and used some prompt engineering techniques to generate data with intentional variations. This included introducing randomization in pricing, quantities, and order sequences, ensuring a more realistic representation of actual transactions while maintaining data integrity across tables.

E-R Diagram

Link to Canva design: https://www.canva.com/design/DAF-f9OUkOo

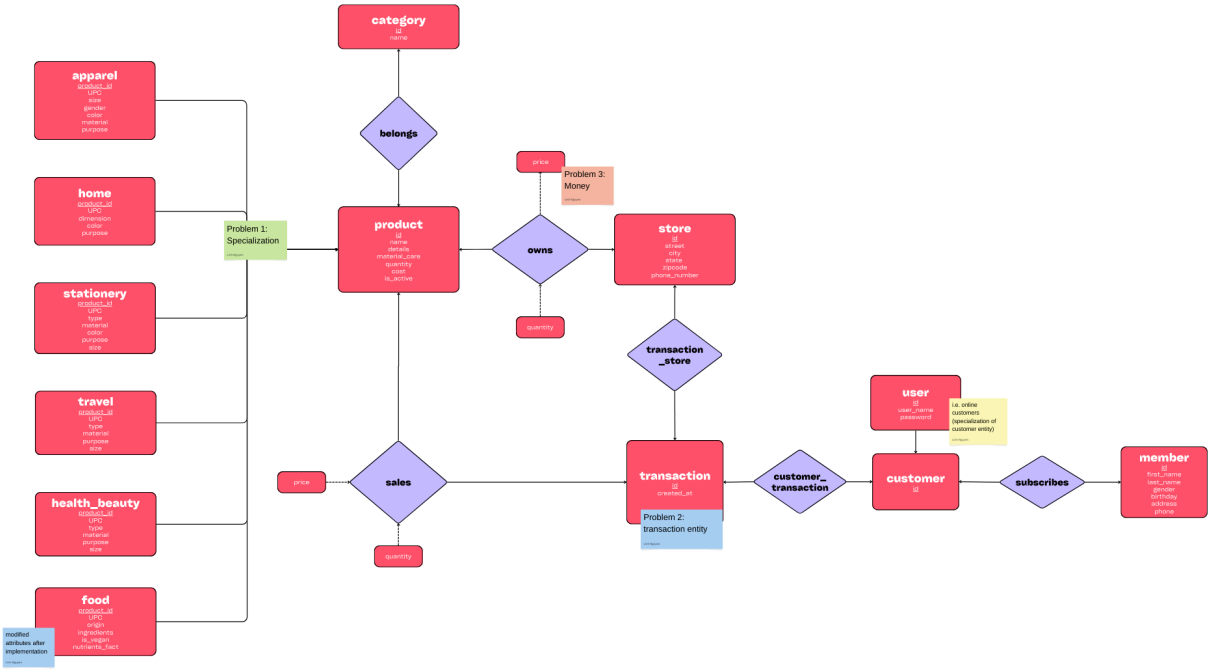

Figure 1. Complete E-R Diagram

Product Entity

Each product has following attributes: *id, name, details, material_care, quantity, cost, is_active*

- *id*: must be unique
- *details*: includes style code and country/region of origin, sometimes product description if necessary
- *material_care*: describes material of product and how to care for it. This is important as most MUJI products are made from natural materials
- *cost*: manufacturing cost
- *is_active*: to discontinue/reshelf product but still keep all relevant information, i.e. transactions

Specialization

There are 6 specializations of a Product entity, representing most general MUJI product categories.

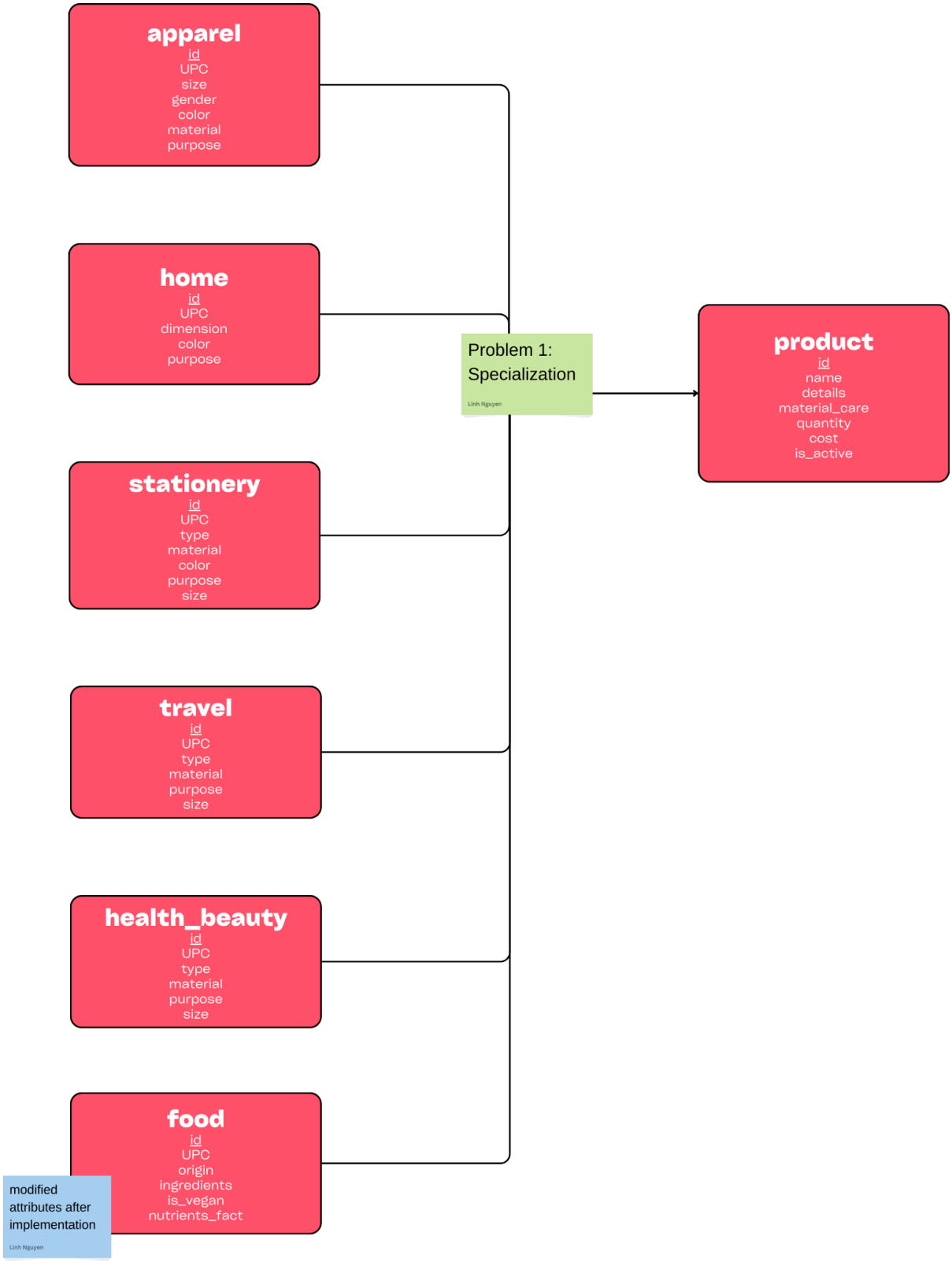

Figure 2. Product entity and its specializations

Initially, we designed the Entity-Relationship (E-R) diagram before learning about specialization, which led to difficulties in organizing the product structure to align with the business requirements. We recognized that products from different categories might have varying attributes. For instance, an apparel product requires information like size, gender, color, and material, while a stationery product may not need all of those attributes. Consequently, each category should be treated as a separate entity. Adopting a single **Product** entity or table that includes attributes for all products is not an ideal approach. It would result in many empty fields within most product entries, potentially leading to issues such as wasted data storage space, more complex queries, slower performance, and ambiguous data interpretation.

To address the challenge of organizing products based on their categories, we decided to implement specialization tables. Each product would be placed in a specific specialization table corresponding to its category. The main **Product** entity would store the most general information common to all products, such as price, quantity, and manufacturing cost. Separate specialization tables would be created for each category, containing attributes specific to products within that category. For example, the apparel specialization table might include details like size, gender, color, and material, while the food specialization table could have attributes like ingredients and nutrition facts.

However, this raises two questions:

- How do we know a product is from which category?
- How do we know about product details?

The first question is solved by a different entity called **Category** and its relationship called **belongs**, which will be discussed in the next section.

To solve the second question, we utilize how specialization tables are set up. Each product entry in the main **Product** table will have an identical unique identifier (*id*) corresponding to its entry in the relevant specialization table for its category. This *id* acts as a bridge, allowing us to connect the general product information in the **Product** table with the category-specific details stored in the appropriate specialization table.

Note that certain attributes may seem similar across multiple specialization tables, but their actual values may differ significantly. For instance, the "size" attribute for travel products (e.g., luggage) may represent three-dimensional measurements, while for stationery products (e.g., notebooks), it might refer to a fixed size designation (e.g., A4, Letter). By separating the products into specialized tables or entities, the database can capture these differences in attribute interpretation or value representation. This ensures that the data model accurately reflects the business and avoids misinterpretation of attribute values across different product categories.

Relevant Entities with Product

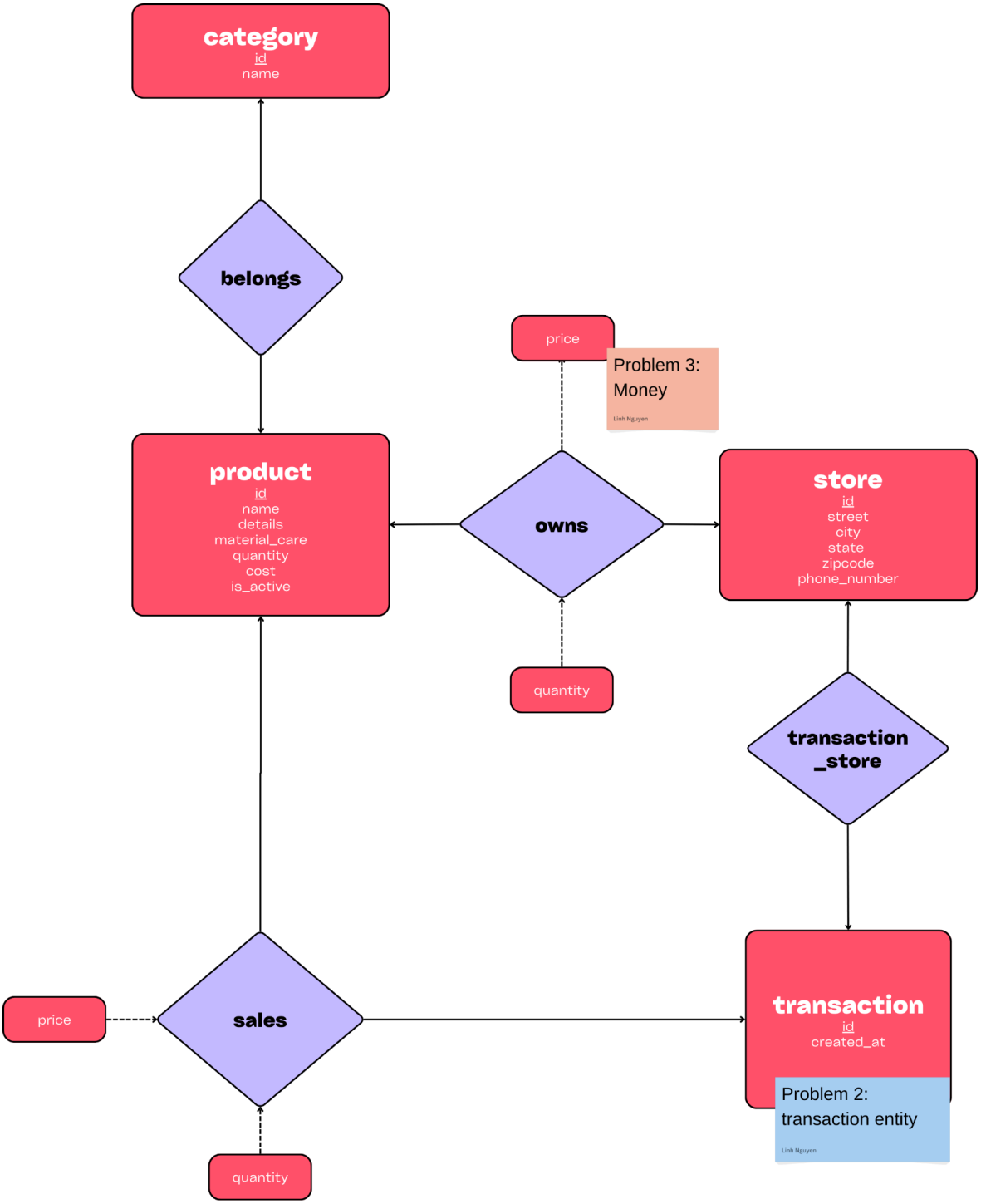

Figure 3. Product and relevant entities

Category Entity

The **Category** entity stores all categories of MUJI products. We use the **belongs** relationship to determine the category a product falls under. Since the relationship between **Product** and **Category** is Many-to-Many, the table **belongs** is necessary. This table maps each product to its corresponding category, allowing for a product to go under more than one category.

Transaction Entity

Our initial design of **Transaction** entity was significantly different from the current version. At first, we did not include a relationship between **Transaction** and **Product**, and store *product_id* directly under **Transaction**. However, as we progressed with the implementation, we realized that the relationship between **Transaction** and **Product** is a Many-to-Many relationship, meaning a single transaction can involve multiple products, and a single product can be part of multiple transactions. To accurately represent this relationship, we added a relationship and denoted it as **sales**. Originally, the **sales** relationship was intended to link **Transaction** and **Store** entities, as our initial plan was to store prices specific to each store location.

Store Entity

This entity stores information about all stores of MUJI. At the beginning, we had a weak entity **opening_hours**, but decided to remove it for a more simple implementation.

An important note here is attributes *price* and *quantity* in the relationship between **Store** and **Product**. These attributes represent the prices of products that are specific to each store location. However, there arises a question regarding how to save store price versus sale price, i.e. the actual price at which a product is sold. This distinction allows for keeping records of discounted item sales while still maintaining the original retail price. In our final version of the diagram, the store price (the price specific to a store location) is represented under the **owns** relationship, while the sale price (the actual selling price) is denoted in the relationship between **Transaction** and **Product**.

Customer and Member Entity

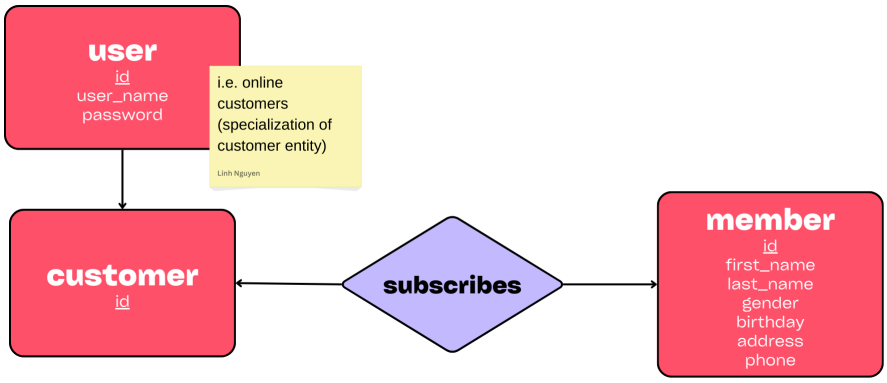

Figure 4. Customer and Member Entity

The last but not least component of the E-R diagram is the **Customer** entity. Any anonymous customer would have an entry in **Customer**, and those who signed up for membership would have an entry in **Member** with all necessary information. To implement an online customer interface, we created a specialization table **User** under **Customer** entity.

Relational Schema

Our relational schema can be translated directly from the E-R diagram. With relationships between entities, we use binary relationships to create appropriate relational tables. For Many-to-Many relationships, we create a relationship table with primary key as union of both participating entities.

- Product - Category: Many-to-Many ⟶ create a relationship table **belongs**
- Product - Transaction: Many-to-Many ⟶ create a relationship table **sales**
- Product - Store: Many-to-Many ⟶ create a relationship table **owns**
- Transaction - Store: One-to-Many ⟶ create an attribute *store_id* in **Transaction**
- Transaction - Customer: One-to-One ⟶ create an attribute *customer_id* in **Transaction**
- Customer - Member: One-to-One ⟶ create an attribute membership_id in **Customer**

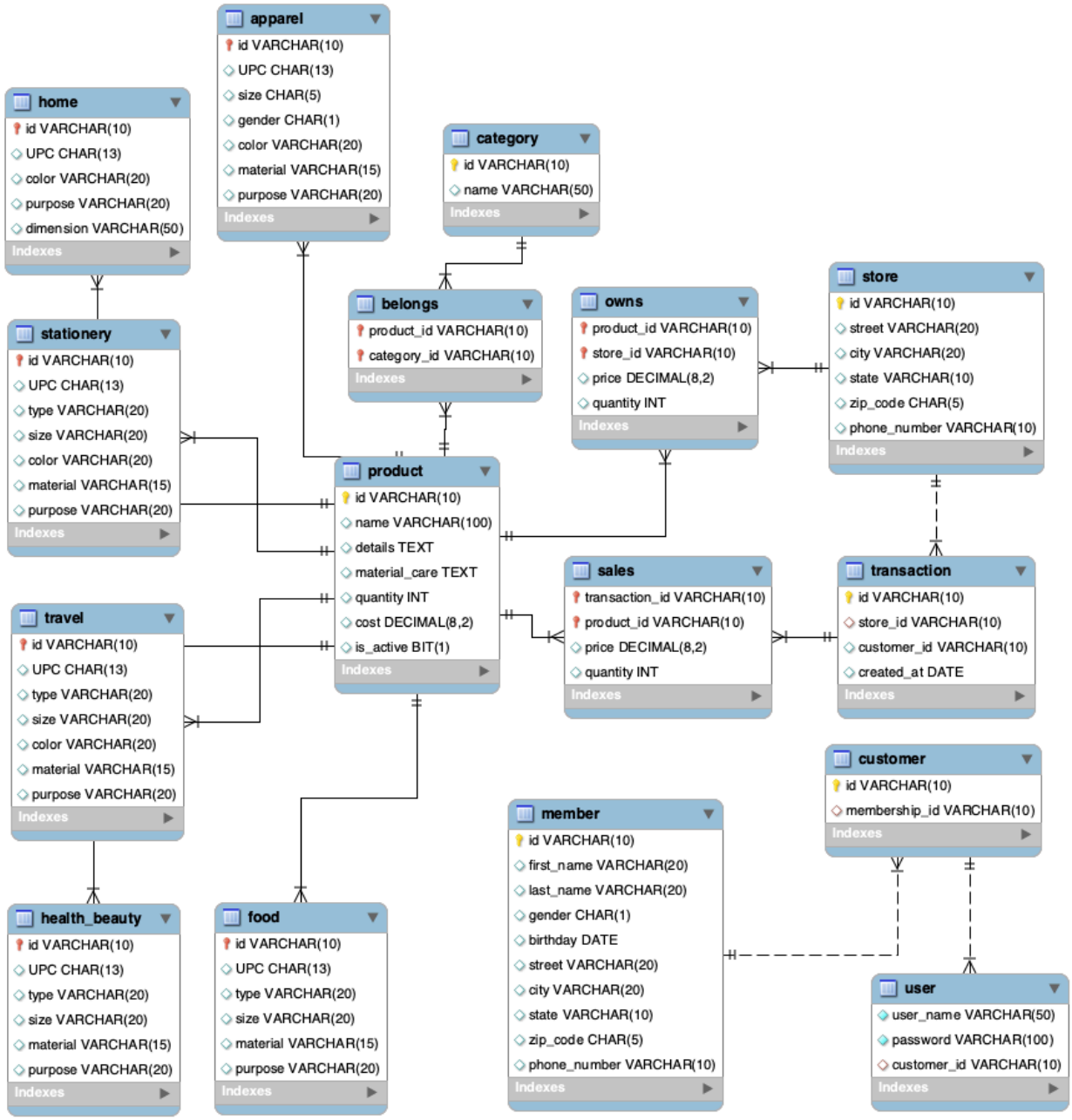

Figure 5. Relational Schema

Sample Queries

To gain insights into the performance and operations of our MUJI stores, we create a set of sample queries to explore our database. These queries cover various aspects, including store information, sales and revenue data, customer and membership details, and product-related queries. Specifically, we have

- 2 queries focused on stores,
- 4 queries relevant to sales and revenue,
- 2 queries for customer and member information,
- 2 queries relevant to products.

We want to provide valuable information to database administrators and business stakeholders, enabling them to understand the overall performance of the brand, individual store locations, sales trends, customer behavior, and product performance. By executing these queries, we can extract meaningful data that can inform strategic decisions, identify areas for improvement, and drive business growth while enhancing customer satisfaction.

Additionally, we implemented some functions to facilitate inventory management, such as adding new inventory to warehouse and to stores, removing inventory (discontinue product) and shifting inventory from store to store.

Queries

1. What is the current inventory of a particular store?

```
SELECT o.store_id, o.product_id, p.name, o.quantity

FROM product p, owns o

WHERE p.is_active = 1

        AND o.product_id = p.id

        AND o.store_id = 'S009';
```

2. What are the top-selling products at a particular store?

```
SELECT product.name, SUM(sales.quantity) AS quantity_sold

FROM sales, transaction, product

WHERE sales.transaction_id = transaction.id

AND sales.product_id = product.id

AND store_id = ‘S000’
```

```
GROUP BY product.name

ORDER BY quantity_sold DESC;
```

3. Which store has the highest total sales revenue?

```
SELECT st.id,  SUM(sl.quantity * sl.price) AS total_revenue

FROM transaction t, sales sl, store st

WHERE t.store_id = st.id AND t.id = sl.transaction_id

GROUP BY st.id

ORDER BY total_revenue DESC;
```

4. What are the 5 stores with the most sales so far this month?

```
SELECT t.store_id, SUM(s.price * s.quantity) AS total_sales

FROM sales s, transaction t

WHERE t.id = s.transaction_id

        AND YEAR(t.created_at) = '2024'

        AND MONTH(t.created_at) = '5'

GROUP BY t.store_id

ORDER BY total_sales DESC

LIMIT 5;
```

5. How many customers are currently enrolled in the frequent-shopper program?

```
SELECT COUNT(id) AS customer, COUNT(membership_id) AS member
```

```
FROM customer;
```

6. What is the average order value for online orders compared to in-store purchases?

```
-- Calculate average order value for online orders (store_id = 'S000')
SELECT AVG(total_order_value) AS avg_online_order_value
FROM (
   SELECT transaction_id, SUM(price * quantity) AS total_order_value
   FROM sales
   WHERE transaction_id IN (SELECT id FROM transaction WHERE store_id = 'S000')
   GROUP BY transaction_id
) AS online_orders;


-- Calculate average order value for in-store purchases (store_id != 'S000')
SELECT AVG(total_order_value) AS avg_instore_order_value
FROM (
   SELECT transaction_id, SUM(price * quantity) AS total_order_value
   FROM sales
   WHERE transaction_id IN (SELECT id FROM transaction WHERE store_id != 'S000')
   GROUP BY transaction_id
) AS instore_orders;
```

7. Which products have the highest profit margin across all stores?

```
SELECT p.id, p.name, (SUM((s.price - p.cost) * s.quantity) / SUM(s.price * s.quantity)) * 100 AS
profit_margin
FROM product p, sales s
```

```
WHERE p.id = s.product_id

GROUP BY p.id, p.name

ORDER BY profit_margin DESC;
```

8. How does the sales performance of a particular product compare between different store locations?

```
SELECT t.store_id , s.price as sale_price, SUM(s.quantity) as sale_quantity

FROM sales AS s, transaction AS t

WHERE s.transaction_id = t.id

AND s.product_id = 'P015'

GROUP BY t.store_id , sale_price;
```

9. Which store locations have the highest percentage of repeat customers?

```
SELECT t.store_id,

        COUNT(DISTINCT c.membership_id) AS total_member,

   COUNT(DISTINCT t.customer_id) AS total_customer,

   (COUNT(DISTINCT c.membership_id) / COUNT(DISTINCT t.customer_id) * 100) AS
membership_percentage

FROM customer c, transaction t

WHERE t.customer_id = c.id

GROUP BY t.store_id

ORDER BY membership_percentage DESC;
```

10. What are the most popular product combinations purchased together by customers?

```
WITH P001_transaction AS (
        SELECT transaction_id
        FROM sales
        WHERE product_id = 'P001'
)
SELECT product_id, COUNT(*) AS count
FROM sales
WHERE transaction_id IN (SELECT transaction_id FROM P001_transaction)
        AND NOT product_id = 'P001'
GROUP BY product_id
ORDER BY count DESC;
```

Functions

This part is only a reference to the actual implementation. More details are in the codes.

Add inventory to a warehouse	`add_inventory_to_product()` `add_inventory_to_specialization(product_id)`
Add inventory to a store from a warehouse	`add_inventory_to_store()`
Remove inventory	`remove_inventory(cnx, product_id)`
Remove inventory from a store	`remove_inventory_from_store(cnx, store_id, product_id, new_quantity)`
Shift inventory	`shift_inventory(cnx, product_id, quantity_moved, source_store_id, target_store_id)`