Phase 1 Report

CSCE 4350.021 LEGO Database Design Project

Author: Matthew Fredericksen

The purpose of this report is to briefly elucidate the design rationale used in my team's database project. This will mostly entail justifying the placement of foreign keys, which determine the nature of data relationships in the database. To do this, I will first show the diagrams produced during the designing process and explain what they mean. I will then provide the actual SQL code needed to instantiate the database schema.

Figure 1: Entity-Relationship Diagram (ERD)

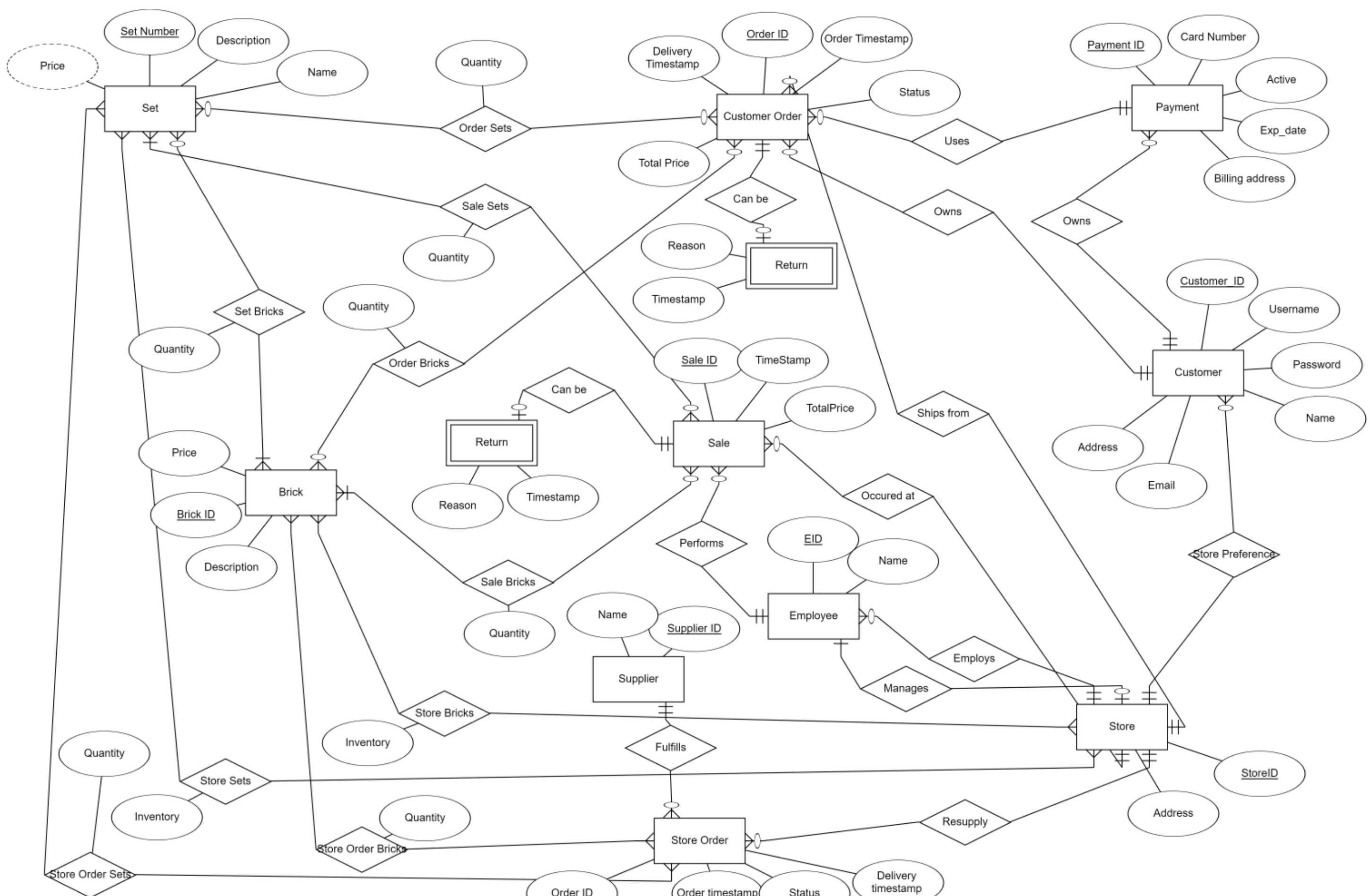
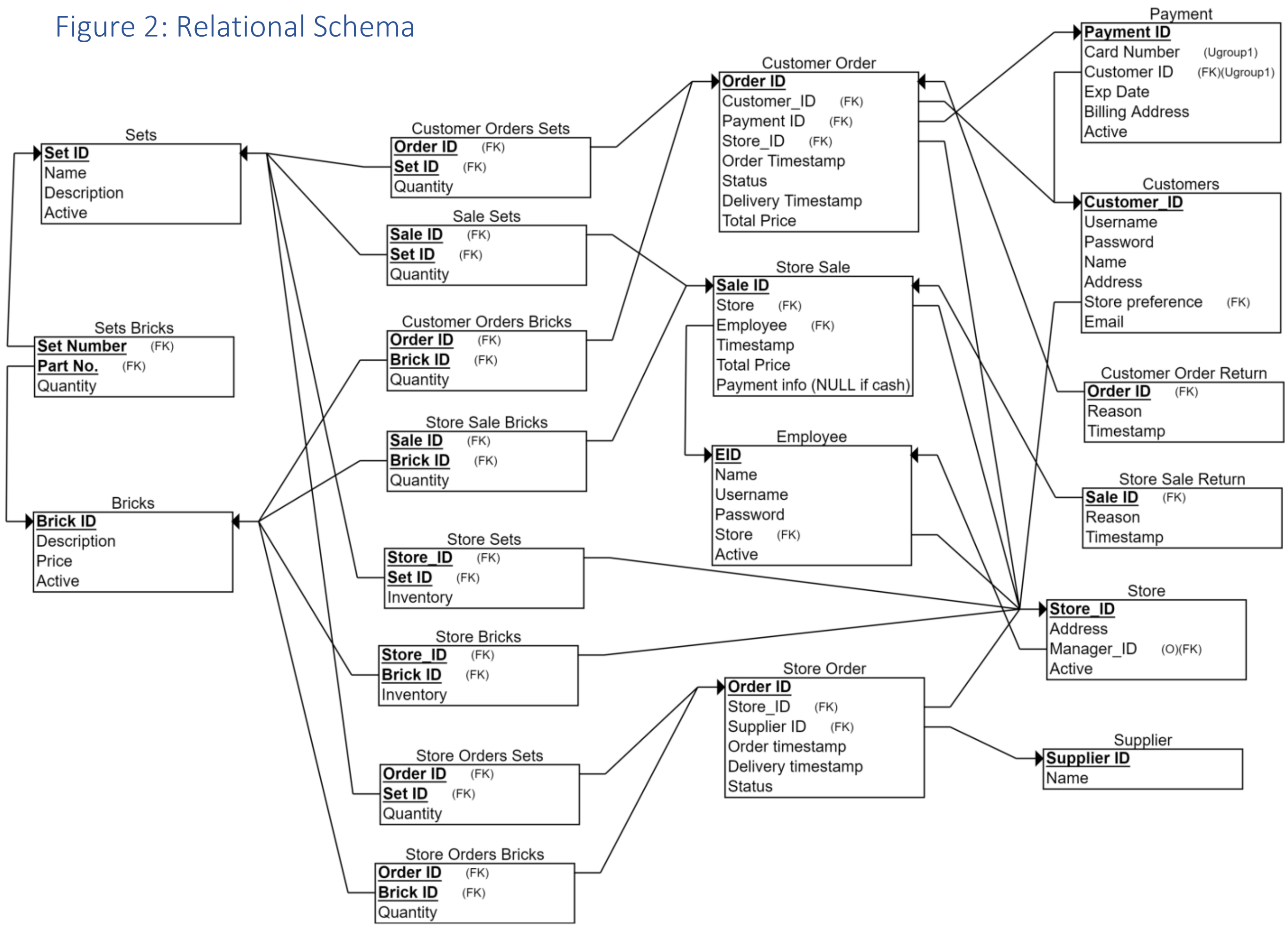


Figure 2: Relational Schema



The core challenge my team faced while designing this schema was figuring out how to store as much information as possible with as little redundancy as possible. Our goal was for our schema to have a small footprint, but still be able to express the information needed to be completely functional.

One of the ways in which we accomplished this is by expressing the contents of transactions (customer orders, store sales, and store orders) through *relationships* with the products they contain, rather than storing the information directly with the order. In other words, rather than storing an order with names, descriptions, and contents of each product contained in the order, we create a many-to-many relationship between orders and products, so the order can be efficiently linked to the information describing its contents. These many-to-many relationships are represented by the "-order sets/bricks" and "sale sets/bricks" objects in both of the above diagrams.

Another challenge was managing the *persistence* of certain information that may be capable of changing in unwanted ways. For example, let's say that we decide not to store the total cost of any transactions, since all the information we need to compute them (quantities and prices) are already stored in the database. This may seem viable on its surface, however, problems arise if there is ever a possibility that product prices will change. For example, if a customer purchases an item immediately before the price of that item is increased, and then they decide to return the item, we want to refund them the original amount spend, not the new amount.

Another aspect of managing data persistence is the application of “active” attributes for sets, bricks, stores, employees, and customers’ payment methods. Since we have already decided that order contents and history are to be defined by relationships, rather than directly stored data, we must guarantee that these relationships are never broken. So, when the “LEGO” company releases a limited-time product, or decides to phase out and old product, we do not want the records of these items to be erased, damaging the integrity of orders and sales in the process. Instead, the products can be switched to “inactive,” and they will be hidden from all user interactions. Likewise, if a customer makes an online purchase, then removes that payment method from their account, and then cancels their purchase, we must retain a way to refund the money they spent. The simple solution is to retain the record, but hide it from all user interactions, which is accomplished through the use of the “active” attribute. Once more, if a store is to be closed down, we want to have a way to prevent interactions through the application without losing all of the information stored in its relationships, so we just set it as “inactive.”

The last notable challenge we faced was choosing which additional attributes would store the most useful information to our hypothetical commercial entity. The first we chose was “timestamps,” as this allows for sorting and analyzing trends in data over time, which can be very useful for enhancing business goals or practices. In our models, major points in transaction processes are marked with a timestamp, such as when an order is first placed, if and when it is delivered, and if and when it is returned. This information could be used to gain valuable insights. For example, if you look at transaction history and notice a significant climb in product return rates after switching to a new supplier, then perhaps that supplier is providing lower-quality goods.

As a final note, the “payment” table was separated from the rest of customers’ information because (1) a customer may want to have multiple payment options simultaneously and (2) we want to retain “old” payment information in case transactions need to be reversed. Additionally, the combination of “customer id” and “card number” is given a unique constraint, rather than just the card number, because it is imaginable that two individuals with separate accounts, perhaps living in the same household, would want to share a payment method.

Figure 3: SQL Code

This section introduced more constraints than could be shown in the previous diagrams. These are basic steps to make sure data is consistent, such as ensuring that transactions cannot be negative, that an online shopping cart cannot have a delivery timestamp, that a set cannot be comprise of zero bricks, etc.

```
CREATE TABLE Stores (
    store_id INT NOT NULL AUTO_INCREMENT,
    address VARCHAR(256) NOT NULL,
    manager_id INT,
    active BOOLEAN NOT NULL DEFAULT TRUE,
      /* because we don't want to lose a
```

```sql
        store's history if the store is closed */
    PRIMARY KEY (store_id)
);

CREATE TABLE Employees (
    employee_id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(64) NOT NULL,
    username VARCHAR(64) NOT NULL UNIQUE,
    password VARCHAR(64) NOT NULL,
    store_id INT NOT NULL,
    active BOOLEAN NOT NULL DEFAULT TRUE,
    PRIMARY KEY (employee_id),
    FOREIGN KEY (store_id) REFERENCES Stores (store_id)
);

ALTER TABLE Stores
ADD FOREIGN KEY (manager_id) REFERENCES Employees (employee_id);

CREATE TABLE Bricks (
    brick_id INT NOT NULL AUTO_INCREMENT,
    description VARCHAR(128) NOT NULL,
    price FLOAT NOT NULL,
    active BOOLEAN NOT NULL DEFAULT TRUE,
    PRIMARY KEY (brick_id),
    CHECK (price > 0.0)
);

-- Attempt to prevent Bricks & Sets IDs from overlapping
ALTER TABLE Bricks AUTO_INCREMENT=10000;

CREATE TABLE Sets (
    set_id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(128) NOT NULL,
    description VARCHAR(512) NOT NULL,
    active BOOLEAN NOT NULL DEFAULT TRUE,
    PRIMARY KEY (set_id)
);

ALTER TABLE Sets AUTO_INCREMENT=1000;

CREATE TABLE Sets_Bricks (
    set_id INT NOT NULL,
    brick_id INT NOT NULL,
    quantity INT NOT NULL,
    FOREIGN KEY (set_id) REFERENCES Sets (set_id),
    FOREIGN KEY (brick_id) REFERENCES Bricks (brick_id),
    PRIMARY KEY (set_id, brick_id),
    CHECK (quantity > 0)
);

CREATE TABLE Stores_Bricks (
    store_id INT NOT NULL,
    brick_id INT NOT NULL,
```

```sql
    inventory INT NOT NULL,
    FOREIGN KEY (store_id) REFERENCES Stores (store_id),
    FOREIGN KEY (brick_id) REFERENCES Bricks (brick_id)
    /* no checks here because we don't want to
       cause issues if inventory goes negative */
);

CREATE TABLE Stores_Sets (
    store_id INT NOT NULL,
    set_id INT NOT NULL,
    inventory INT NOT NULL,
    FOREIGN KEY (store_id) REFERENCES Stores (store_id),
    FOREIGN KEY (set_id) REFERENCES Sets (set_id)
    /* no checks here because we don't want to
       cause issues if inventory goes negative */
);

CREATE TABLE Customers (
    customer_id INT NOT NULL AUTO_INCREMENT,
    email VARCHAR(64) NOT NULL UNIQUE,
    username VARCHAR(64) NOT NULL UNIQUE,
    password VARCHAR(128) NOT NULL,
    name VARCHAR(64) NOT NULL,
    address VARCHAR(256) NOT NULL,
    store_preference INT NOT NULL,
    PRIMARY KEY (customer_id),
    FOREIGN KEY (store_preference) REFERENCES Stores (store_id)
);

CREATE TABLE Payments (
    payment_id INT NOT NULL AUTO_INCREMENT,
    customer_id INT NOT NULL,
    card_number CHAR(16) NOT NULL,
    exp_date DATE NOT NULL,
    billing_address VARCHAR(256) NOT NULL,
    active BOOLEAN NOT NULL DEFAULT TRUE,
        /* because this data should be retained in case a customer
           removes this payment method and then cancels an order,
           so we can still perform the refund */
    PRIMARY KEY (payment_id),
    FOREIGN KEY (customer_id) REFERENCES Customers (customer_id),
    UNIQUE (customer_id, card_number),
    CHECK (LENGTH(card_number) = 16)
);

CREATE TABLE Customer_Orders (
    order_id INT NOT NULL AUTO_INCREMENT,
    customer_id INT NOT NULL,
    payment_id INT,
    store_id INT,
    order_timestamp TIMESTAMP,
    delivery_timestamp TIMESTAMP,
    status VARCHAR(16) NOT NULL,
```

```
    total_price FLOAT,
    PRIMARY KEY (order_id),
    FOREIGN KEY (customer_id) REFERENCES Customers (customer_id),
    FOREIGN KEY (payment_id) REFERENCES Payments (payment_id),
    CHECK (delivery_timestamp IS NULL OR
           delivery_timestamp > order_timestamp),
    CHECK (status IN ('Processing', 'Shipping',
                      'Delivered', 'Cancelled',
                      'Returned', 'Cart')),
    CHECK ((payment_id IS NULL OR
            store_id IS NULL OR
            order_timestamp IS NULL OR
            total_price IS NULL)
           XOR status != 'Cart'),
    CHECK (total_price IS NULL OR
           total_price > 0.0)
);

-- so that customers will always have a cart
CREATE TRIGGER auto_cart_insert AFTER INSERT ON Customers
FOR EACH ROW
    INSERT INTO customer_orders (customer_id, status)
    VALUES (NEW.customer_id, 'Cart');

CREATE TABLE Customer_Orders_Returns (
    order_id INT NOT NULL,
    reason VARCHAR(256),
    return_timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP(),
    FOREIGN KEY (order_id) REFERENCES Customer_Orders (order_id),
    PRIMARY KEY (order_id)
);

CREATE TABLE Customer_Orders_Bricks (
    order_id INT NOT NULL,
    brick_id INT NOT NULL,
    quantity INT NOT NULL,
    FOREIGN KEY (order_id) REFERENCES Customer_Orders (order_id),
    FOREIGN KEY (brick_id) REFERENCES Bricks (brick_id),
    PRIMARY KEY (order_id, brick_id),
    CHECK (quantity > 0)
);

CREATE TABLE Customer_Orders_Sets (
    order_id INT NOT NULL,
    set_id INT NOT NULL,
    quantity INT NOT NULL,
    FOREIGN KEY (order_id) REFERENCES Customer_Orders (order_id),
    FOREIGN KEY (set_id) REFERENCES Sets (set_id),
    PRIMARY KEY (order_id, set_id),
    CHECK (quantity > 0)
);

CREATE TABLE Store_Sales (
```

```sql
    sale_id INT NOT NULL AUTO_INCREMENT,
    store_id INT NOT NULL,
    employee_id INT NOT NULL,
    sale_timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP(),
    total_price FLOAT NOT NULL,
    credit_card CHAR(16),
    PRIMARY KEY (sale_id),
    FOREIGN KEY (store_id) REFERENCES Stores (store_id),
    FOREIGN KEY (employee_id) REFERENCES Employees (employee_id),
    CHECK (total_price > 0.0),
    CHECK (LENGTH(credit_card) = 16)
);

CREATE TABLE Store_Sales_Returns (
    sale_id INT NOT NULL,
    reason VARCHAR(256),
    return_timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP(),
    FOREIGN KEY (sale_id) REFERENCES Store_Sales (sale_id),
    PRIMARY KEY (sale_id)
);

CREATE TABLE Store_Sales_Bricks (
    sale_id INT NOT NULL,
    brick_id INT NOT NULL,
    quantity INT NOT NULL,
    FOREIGN KEY (sale_id) REFERENCES Store_Sales (sale_id),
    FOREIGN KEY (brick_id) REFERENCES Bricks (brick_id),
    PRIMARY KEY (sale_id, brick_id),
    CHECK (quantity > 0)
);

CREATE TABLE Store_Sales_Sets (
    sale_id INT NOT NULL,
    set_id INT NOT NULL,
    quantity INT NOT NULL,
    FOREIGN KEY (sale_id) REFERENCES Store_Sales (sale_id),
    FOREIGN KEY (set_id) REFERENCES Sets (set_id),
    PRIMARY KEY (sale_id, set_id),
    CHECK (quantity > 0)
);

CREATE TABLE Suppliers (
    supplier_id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(128) NOT NULL,
    PRIMARY KEY (supplier_id)
);

CREATE TABLE Store_Orders (
    order_id INT NOT NULL AUTO_INCREMENT,
    store_id INT NOT NULL,
    supplier_id INT NOT NULL,
    order_timestamp TIMESTAMP,
    delivery_timestamp TIMESTAMP,
```

```sql
    status VARCHAR(16) NOT NULL,
    PRIMARY KEY (order_id),
    FOREIGN KEY (store_id) REFERENCES Stores (store_id),
    FOREIGN KEY (supplier_id) REFERENCES Suppliers (supplier_id),
    CHECK (status IN ('Ordered', 'Delivered', 'Cancelled'))
);

CREATE TABLE Store_Orders_Bricks (
    order_id INT NOT NULL,
    brick_id INT NOT NULL,
    quantity INT NOT NULL,
    FOREIGN KEY (order_id) REFERENCES Store_Orders (order_id),
    FOREIGN KEY (brick_id) REFERENCES Bricks (brick_id),
    PRIMARY KEY (order_id, brick_id),
    CHECK (quantity > 0)
);

CREATE TABLE Store_Orders_Sets (
    order_id INT NOT NULL,
    set_id INT NOT NULL,
    quantity INT NOT NULL,
    FOREIGN KEY (order_id) REFERENCES Store_Orders (order_id),
    FOREIGN KEY (set_id) REFERENCES Sets (set_id),
    CHECK (quantity > 0)
);
```