CSCE 608

Project 1

Database System Design

Fall 2017

Divyesh M Tekale

UIN: 923004428

NetID: tekale2

Project Description

I was really glad that the choice of the database design was left for us to decide and hence I chose to design a miniature version of Amazon like shopping store. Amazon is used by people all over the world every day for buying things online. Retailers add new products and customers buy those products. Also Amazon stores multiple addresses and Credit card information for customers and those addresses can be selected for each order. Also history of current and previous orders are stored in the database even though the product may no longer be available or the credit card information may have been deleted. I think it is really challenging to maintain such a huge system which is also distributed to scale. Hence I planned and created a miniature version of Amazon: **Evergreen Groceries Store**. The whole application has been developed using **Ruby on Rails for** and **Postgres SQL** database has been used for storing the relations. The app was designed with the use case of **customer support**, since they have administrative access to add, edit, modify and delete the information in the database.

There were many challenges in the initial design phase of the project. Customer can change information like addresses, payment information (credit/debit cards), email and contact. Many people can reside at same address to and have different accounts in the database. Similarly, suppliers can remove products from the shopping store, but the old orders should still contain the right information about the amount tendered for the order. Hence I began my design with three entities: Customers, Orders and Products and then I normalized to 5 relations and a total of 7 tables in the database.

For customer support usage, the application offers searching the database for customers by their name, email, or contact number. Addresses can be searched by street name, city, state and zipcode. If two or more people share the same name, then a particular customer can be found out by entering email id or contact. Similarly products in the store can be searched by name or price range. Once the customer is searched, their associated orders, payment information, addresses can be found and modified.

Certain Assumptions were made designing this database:

- Each customer has many credit cards, and those cards cannot be shared by any other customer.
- Addresses are independent of customer as many people can live in the same place. Deleting a customer should not delete the address.
- Orders contain information about the payment, shipping addresses products and their quantities. If a product is deleted then, amount and quantity information is still retained. On the other hand if associated payment or address is removed then Orders are not affected, but they no more contain reference to deleted payments and addresses
- Orders and payments wholly belong to a customer. Hence if a customer is deleted, then all the associated payments, and orders are deleted. Linkages to addresses are deleted but actual address is not deleted.
- Similarly if an order is deleted, then the linkage to products is deleted but actual products are not deleted, because they might belong to other orders.

- For now just like amazon, my application supports 1 type of payment storage: credit cards. Other forms of payment like echeck, or wire transfer would have made the database design more complex.

Based on the above assumptions, my application provides CRUD operations on 5 entities, with no data duplication and maintaining the database integrity.

Data Collection

Raw data for this application was collected using https://www.mockaroo.com. This website provides free mock data upto 1000 rows. Cutomer information like name, email, contact, US based addresses (city, state, street, zipcode) and credit card payments information (card number, cvv, exipry, type). Once this data was obtained this was cleaned using the dataGenerator.py file which is included with this project.

After the normalization was done data was generated in the following way:

All tables **used incremental integer as primary key**. This is a default option in rails, since it automatically manages the primary key value for every table in the database.

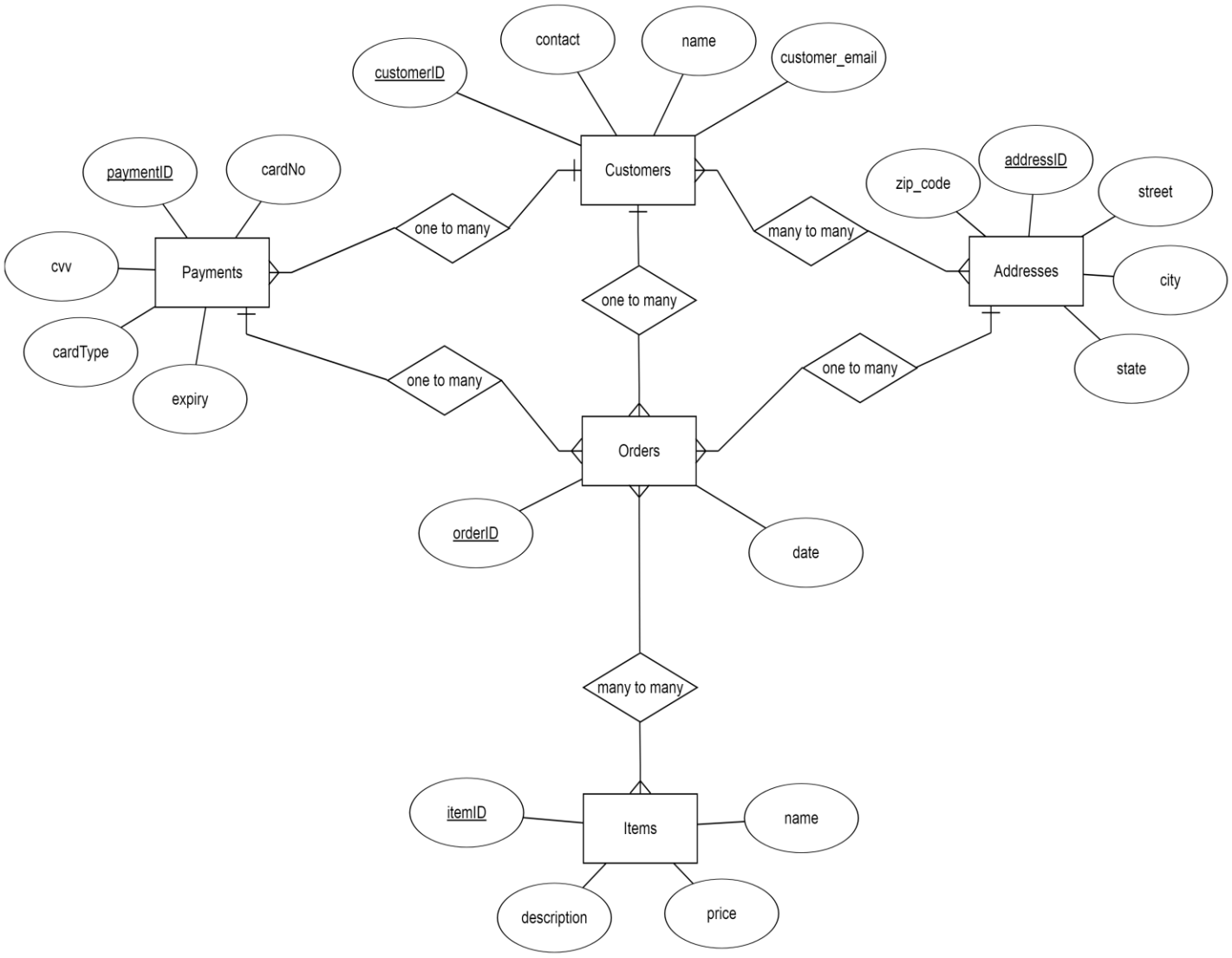
For every customer, payments in the range of one to ten were associated with them. Addresses were similarly associated using customer_addresses table as customers and addresses had many to many relationship.

The most challenging part was generating orders. I made sure that data generation was as much random as possible. I decided to generate about 100 orders in total. A customer was randomly chosen and then for that customer a random address and payment is associated with the order. Finally for each order, 1 to 10 items were randomly added, and the quantity was chosen from 1 to 5.

Data generation part was challenging since it required careful planning of phases as each generation phase was dependent on the output of the previous phases. All the data was stored as CSV file. Using ruby a script was written that loaded all items to their corresponding tables. These scripts can be invoked using rake taskname:tasktorun. Primary keys were excluded from the tables as rails automatically adds them and maintains it for various purposes like forign key, validations and rendering templates.

ER Diagram

The following ER diagram was constructed using the assumptions stated and normalization. This ER diagram contains 5 entities in total: Customers, Addresses, Payments, Orders and Items. After normalization and to describe many to many relationship total of 7 tables were eventually created.



The following relationship holds among the entities:

1. Many to Many between Customers and Addresses
2. Many to Many between Orders and Items
3. One to Many between Customer and Payments
4. One to Many between Customer and Orders
5. One to Many between Orders and Payments
6. One to Many between Orders and Addresses

To hold many to many relationship between Customers and Addresses, a link table Customer_Addresses was used to link them together using address and customer as foreign keys.

Similarly, to establish relation between Items and Orders, order_items table was constructed which contained orderId, and itemID as foreign keys, and relation attributes: total price and

quantity are stored in that table. Rest of the relations use regular tables and reference each other using foreign keys.

Normalization

Seven tables were created as follows in rails:

```
create_table "addresses", force: :cascade do |t|
  t.string "street"
  t.string "city"
  t.string "state"
  t.string "zipcode", null: false
end

create_table "customer_addresses", force: :cascade do |t|
  t.integer "customer_id"
  t.integer "address_id"
end

create_table "customers", force: :cascade do |t|
  t.string "name",    null: false
  t.string "email",   null: false
  t.string "contact", null: false
end

create_table "items", force: :cascade do |t|
  t.string  "name",        null: false
  t.text    "description"
  t.integer "price",       null: false
end

create_table "order_items", force: :cascade do |t|
  t.integer "order_id"
  t.integer "item_id"
  t.integer "quantity",       null: false
  t.integer "totalItemPrice"
end

create_table "orders", force: :cascade do |t|
  t.date    "placedOn",    null: false
  t.integer "customer_id", null:false
  t.integer "address_id"
  t.integer "payment_id"
end
```

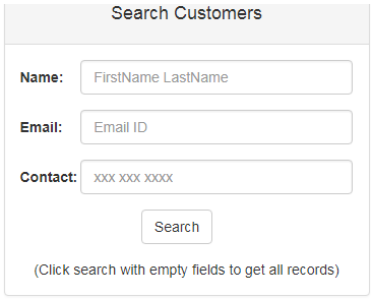
```
create_table "payments", force: :cascade do |t|
  t.string  "cvv",         null: false
  t.string  "cardNo",      null: false
  t.string  "cardType",    null: false
  t.date    "expiry",      null: false
  t.integer "customer_id"
end
```

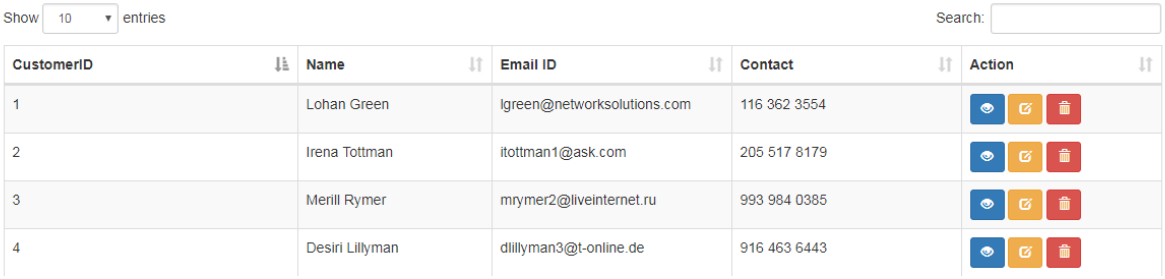
For a relation to be in BCNF the following condition must be met: There should not be any functional dependency from prime or non-prime attribute to a prime attribute. In simple terms a prime attribute cannot be derived. We see that in all our relations that they are in BCNF because the normalized tables contains mostly non-prime attributes. Example the Customer (email,contact,id) →(name). Address(id)→(street, city,state,zip), Payment(id)→(cvv, cardNo, expiry, cardType). All of relations follow the same derivation and hence the **schema is normalized in BCNF.**

User Interface

The user interface supports CRUD on all 7 tables and there are a total of 15 pages to provide this functionality. For the purpose of this report I would like to include the interesting and challenging parts of the interface:

The home page is search for Customers. Executing Search without filling the form would retrieve all the records:





Query is executed based on form parameters, for this view

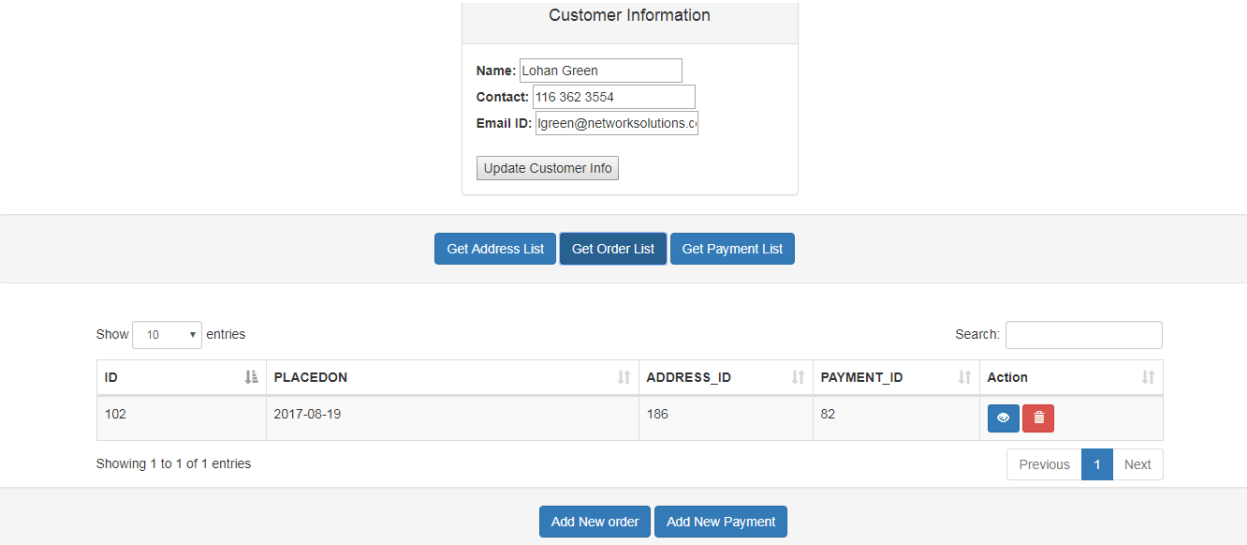
```
SELECT "customers".* FROM "customers"
```

The table search lets you filter even more records. The data can be sorted by any column in ascending or descending order. Each row has supported action key: view edit and delete

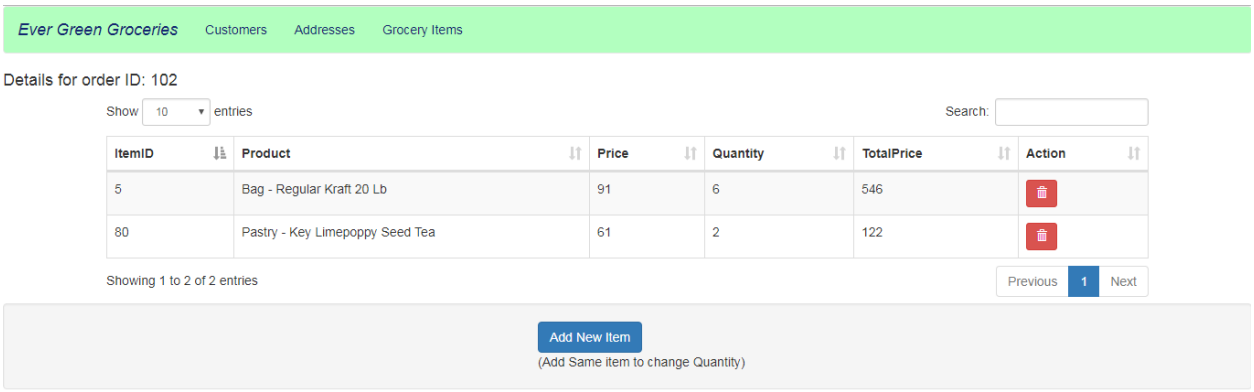
If a customer is deleted from the table following sequence of queries are ran:

SELECT "customers".* FROM "customers" WHERE "customers"."id" = ? LIMIT 1
DELETE FROM "payments" WHERE "payments"."customer_id" = ?
DELETE FROM "orders" WHERE "orders"."customer_id" = ?
DELETE FROM "customer_addresses" WHERE "customer_addresses"."customer_id" = ?
DELETE FROM "customers" WHERE "customers"."id" = ?
commit transaction

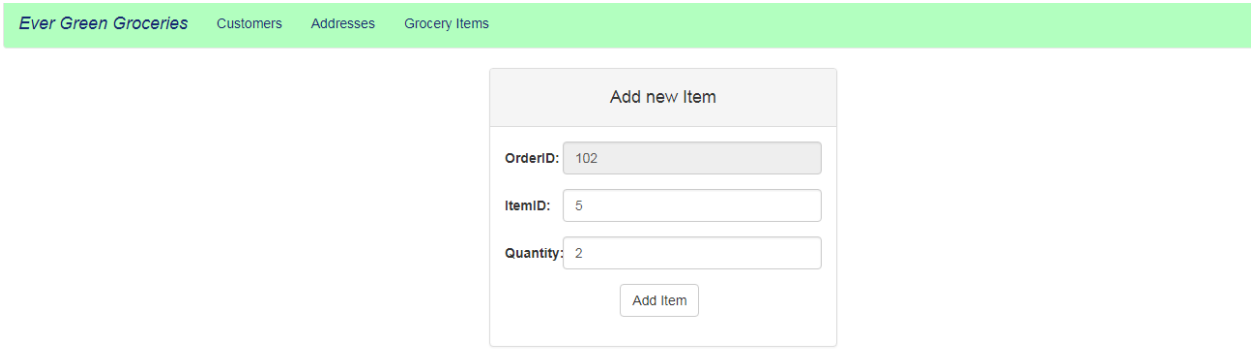
Edit Customer view:
When view/edit button is seleted the user is redirected to this page. The user can view associated payment info, addresses and orders for that customer. Also the customer info can be updated. Also one can add new order/payment to that customer. The tables are dynamically generated based on button clicks and can be filtered using the search on the client side. Again each row of the table has set of supported action. Please note that there is no better way to represent an address or Payment than their primary key because address as a whole is a candidate key and similarly card information as a whole is a candidate key. Hence they are represented by their IDs



Following query is ran for updating customer info based on the changed parameters:
UPDATE "customers" SET "name" = ? WHERE "customers"."id" = ?
Commit transaction
View Order renders into another page which shows the products in the order and other relevant details are presented too:

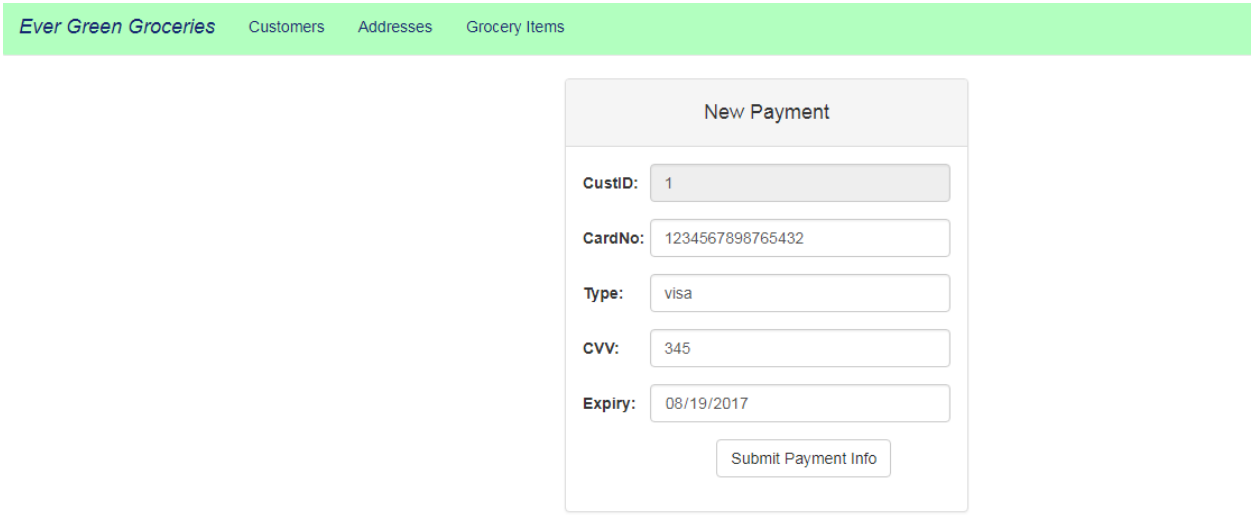


Following queries are ran to get this page:
SELECT "orders".* FROM "orders" WHERE "orders"."id" = ?
SELECT items.id, items.price, items.name, "quantity", "totalItemPrice" FROM "items" INNER JOIN "order_items" ON "items"."id" = "order_items"."item_id" WHERE "order_items"."order_id" = ?
Add new product to existing order can be added using this page. If item already exists, then quantity and price is updated in the order_items table. Validations are made to make sure that quantity is not negative and greater than 0. Also OrderID field is blocked from edit since the product is added to that particular order.



Following queries run when a new item is added:
SELECT "order_items".* FROM "order_items" WHERE "order_items"."order_id" = ? AND "order_items"."item_id" = ? LIMIT 1
begin transaction
INSERT INTO "order_items" ("order_id", "item_id", "quantity", "totalItemPrice") VALUES (?, ?, ?,?)
commit transaction

Similarly a new payment for a customer is added in similar way:



Following Queries run on add new payment:
INSERT INTO "payments" ("cvv", "cardNo", "cardType", "expiry", "customer_id") VALUES (?, ?, ?, ?, ?)
commit transaction

Validations are made to make sure that no duplicates exists before insert and fields like card number are checked for length. The insertion fails if the check fails.

Project Source Code

Application is running on heroku and can be found at: https://stark-crag-91674.herokuapp.com/

Source code for the application can be found on github at github.com/tekale2/Shopping_Cart

Discussion

Designing the database was the most challenging part. I am really glad that prof Chen and the T.A clarified the doubts, which led me to design an improved version of the database. I had faced following challenges:

- I spent many days learning about database normalization, concepts of prime attributes, non-prime attributes, functional dependencies, and candidate keys. I also learnt how to turn a relation in BCNF using the textbook and the internet sources
- I had used ruby on rails for CSCE 606 class and prior to that I had no experience with web development. The project that I did during my 606 class was not database heavy and hence I had to learn a lot about Rails Models and Associations and how they work together to execute queries, Join tables and run validations and callbacks.
- I was a beginner in JavaScript. Since there were so many tables to display, I learnt Jquery, Bootstrap and Datatable libraries. I spent hours debugging issues with AJAX calls and routing.

- I used cloud9 as my development interface. I ran and tested my code on cloud 9, but deploying to heroku was a huge challenge as database versions and ruby versions were different. Also I had to learn to debug my app remotely on Heroku.