**Team BRAMS**

**Members:** Bryan Vu, Rachel Pai, Autumn Nguyen,   
Marjorie Balaoro and Saurav Chhapawala  
CECS 323 Term Project

**Term Project: Phase 3 - Denormalization**

**Note: All bolded words are classes and underlined + bolded words are Primary Keys**

**CustomerAccounts**

In 3rd normal form, we distinguished different types of customers. We identified which account is either a private or corporation by splitting the customer table into two categories (complete, disjoint): **PrivateCustomer** and **CorporationCustomer** which are the children of the Customer, all uniquely represented with a **CustomerID**.

However, if we denormalize this part of the model, we would be combining all of the customer accounts to be represented in a table of **Customers**. By doing this, it would cause redundancy of information about both types of customers. However, denormalizing would make it easier to query information from all of the **Customers** made without joining the child classes.

Denormalizing this section of the model would create the following areas of redundancy:

* **PrivateCustomer** contains the following additional attribute: Email
* **CorporationCustomer** contains the following additional attributes: corporationName, departmentName and is associated through a many to one relationship with a **CorporationContactInfo**.

If we were to combine all types of customers into one, it would create a subkey in the resulting table in order to distinguish which type of customer account it is. Additionally, the additional attributes of a **PrivateCustomer** and **CorporationCustomers** would be redundantly stored in the one table. For example, for a private customer account, the columns that would be populated for a corporation customer would be null and vice versa if the customer was a private customer. This would cause inconsistent data in the database because these attributes could be filled in with contradictory information. For example, a private customer account could have the attributes of corporation contact information populated or with many different accounts under different departments of a corporation, the corporation contact information would be inconsistent.

In order to prevent ending up with inconsistent data, we would implement an on insert and an on update trigger into the combined table: **Customers** to ensure that the customer information agrees with the type of customer account it is. The trigger would act logically like an if else statement that would only place information in the additional attributes of the customer if the customer account type agrees. For example, the attribute Email would only be filled out only if the customer account type is private. Additionally, for customer accounts that are of type corporation customers, the trigger would need to ensure that all of the attributes of a corporation: corporationName, departmentName, contactInfoType, and contactInfo, would all need to be populated.

Below is another section of the model that we can denormalize:

\*\*Author’s Note: We wrote this one out first but decided to go with Customers instead. Please provide us feedback on which one seems more optimal to implement into the project.

**Orders**

In 3rd normal form, we distinguished different types of **Orders**. We identified the type of order being made by a customer by splitting the Orders table into two categories (complete, disjoint): **To-GoOrder** and **EatInOrder** which are the children of **Orders**, all uniquely represented with a **OrderID**.

However, if we denormalize this part of the model, we would be combining all of the attributes of the different types of orders to be represented in just one table of **Orders**. By doing this, it would cause redundancy but make it easier to query information from all of the orders made without joining the child classes.

Denormalizing this section of the model would create the following areas of redundancy:

* **EatInOrders** contain the following additional attributes: PartySize and GratuityCharge
* **To-GoOrders** contain the following additional attributes: PickUpTime and TimeCooked

If we were to combine all types of orders into one would create a subkey in the resulting table in order to distinguish which type of order it is. Additionally, the additional attributes of an EatInOrder would be null for an order that is to-go and vice versa if the order was an eat-in. This would cause inconsistent data in the database because these attributes could be filled in with contradictory information. For example, a given to-go order could have the attribute PartySize populated.

In order to prevent ending up with inconsistent data, we would implement an on insert and an on update trigger into **Orders** to ensure that the order information agrees with the type of order it is. The trigger would act as an if else statement that would only place information in the additional attributes of the order if the order type agrees. For example, the attributes PartySize and GratuityCharge would only be filled out only if the order type is an eat-in order.