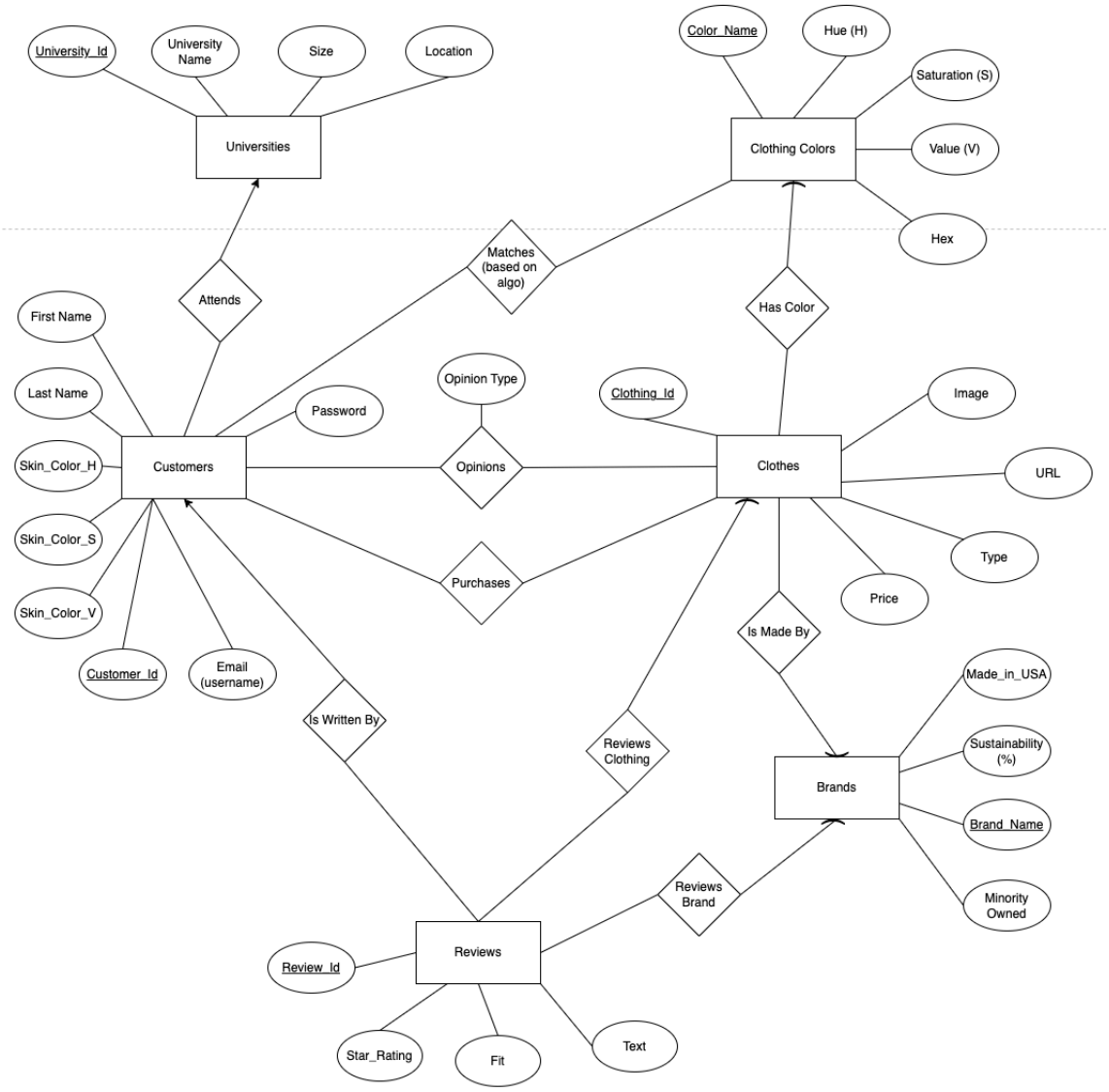
Swipr Stage 2: Database Design

Group Members:
Raahul Rajah (rrajah2)
Ritvik Manda (rsmanda2)
Pranav Nagarajan (pranavn6)
Trivikram Battalapalli (tb17)

ER Diagram



Entities

In our database for our project, we will have **six** main entities.

Entity 1: Users

This table will hold all the information about all users in the app. We chose this table to be one of the entities in our database as there is a lot of personal information associated with each user. In addition, each user's information is connected to multiple other entities, enabling us to give clothing recommendations. User authentication information will be stored in this table. When first using the Swipr App the user will be prompted to create an account with an email and password of their choice. Any information unique to the user will be stored, such as their skin color, which will be extracted using a picture of their face, and their name. Finally, each user will have a unique Customer_Id, which will be the primary key for the table. Since other tables relate to Customer_Id uniquely, it is imperative to characterize the Users table as an entity.

Entity 2: Clothes

A massive part of the Swipr App will be the clothes that will be recommended, so we thought it deserved to be an entity in our database schema. The table will hold lots of information about each clothing article that will be used to provide recommendations. Each of the clothes will have a unique Clothing_Id, which is the primary key for the table. Other information specific to the clothing article will also be stored such as the color, type of clothing (tops, bottoms, outerwear), price, and the URL to buy the specific item. Many relationships with other entities are also required. For example, each piece of clothing will be associated with a color in the Colors table through a matching algorithm based on hsv values. Additionally, each user will be related to many pieces of clothing through a likes/dislikes table where each clothing article that the user likes will be stored.

Entity 3: Colors

This entity table holds all information about the specific colors of all the clothing articles in the clothes table. We need this table to help choose what colors can be recommended to the user based on skin color. This will be calculated through an algorithm based on the HSV values of both the skin color and the actual colors of the clothes themselves. Simply organizing the colors by name and applying them as an attribute to the clothes would not give us the information necessary to match clothing to skin color according to color theory. Instead, we broke the colors down into their individual hue, saturation, and value elements to enable our algorithm to match effectively, motivating us to make them their own entity. The color name would be the primary key in this table. Additionally, since both the user and clothing tables are connected to the colors, we needed to distinguish these as their own entity.

Entity 4: Brands

The Brands table holds all the information specific to each brand we are using. The primary key for this table would be the brand name. In our app, the user is also allowed to choose filters on brands that would make it so only clothes from certain brands would be used to give recommendations. Rather than just filtering by brand name, the user would have the option to filter based on brand attributes such as if the brand is minority-owned, if the brand manufactures clothes with sustainable materials, or if the brand primarily manufactures its products in the USA. Each clothing item would be associated with a brand so there is a relation between this table and the clothes table. We will also implement reviews, which helps customers see which clothes, and in turn, which brands have the best reviews. As a result, it is important to connect the reviews table, which will be discussed below, to brands such that their attributes are also taken into account.

Entity 5: Reviews

The reviews table holds all of the customer reviews of each clothing item. Each review has its own "fit" attribute which describes how well the clothing fits true to its given size, a rating attribute where a star rating out of 5 is given, and a "text" attribute where the user has the option to provide their review in text format. The table has a Review_Id which is the primary key of the table. This is related to the customer, clothing, and brands tables. The customers write the reviews, and these reviews are connected to each clothing item and in turn each brand.

Entity 6: Universities

The Universities table in the Swipr App enriches the platform by linking users with academic institutions, allowing for clothing recommendations based on university-specific trends. Each university is uniquely identified by a University_Id and detailed with attributes like University_Name, Size, and Location. This setup not only facilitates personalized style suggestions but also fosters a sense of community among users from the same institution. The table's integration enables features such as filtering clothing preferences by university, promoting campus spirit, and tailoring content to regional styles. By associating the Universities table with the Users table through a foreign key relationship, the app ensures relevant recommendations, enhancing user engagement with a focus on academic connections.

Relationship Between Entities

In our database, we have three relationships that use tables, and five relationships without tables.

Relationship 1: Opinions

We first needed a way to represent exactly how a customer will choose the clothing that matches them. If they see a piece of clothing, they can categorize it as part of their likes, dislikes, or favorites. However, this is not necessarily an entity since it composes information from two of our main entities, which is the customer and the clothing. Hence, we realized that instead of an entity, it would be a relationship between the customer and clothing tables. We looked to see the multiplicity of relationships, and realized that it was a many-many relationship because many users can provide their opinions on many clothing items. This brought the need for a many-many relationship. We can represent this relationship more in detail through a table, and hence we have a table that relates both customers and clothing, while also characterizing it by their opinions (like, dislikes, favorites).

Relationship 2: Purchases

A single user can purchase many articles of clothing. Articles of clothing can also be purchased by many users. This means that the cardinality of this relationship is many-many. There is a table to represent this relationship where each entry to the table uses the Customer_Id and Clothing_Id foreign keys to connect the tables.

Relationship 3: Color_Match

Our app will use the skin color of each user to help match them to the appropriate colors they would look best in. After the algorithm to choose the colors is run, we need a way to store which colors were matched with which user. This is where the Color_Match table would be created and used. Each user would be matched with a set of color names through a table that uses the Customer_Id and color names as foreign keys. Each user could be matched with multiple colors and each color could be associated with multiple users, meaning the cardinality of this relationship is also many-to-many.

Many-to-1 Relationships

Between our tables, we have five many-to-one relationships which are Customer, Color, Clothing, Brand #1, and Brand #2. We needed to relate some of the attributes between the tables, and this can only be done through relationships. For reviews, many of them can be related to one customer, as a customer can create many reviews. For color, many clothes can have the same color but lots of colors cannot correlate to a single piece (sure, there are multicolored clothes, but not in our implementation). For clothing, we needed to relate a review to what type of clothing is actually being reviewed, and there can be many reviews for one particular piece of clothing. For brand #1, we needed to attribute the clothes based on the brand, and if the user wants to toggle based on the brand, then they have that option as well. For brand #2, we wanted to implement a feature where users can select a brand based on reviews, so we made sure to relate the review with the brand. In

this case, multiple reviews can be directed to one brand, but multiple brands cannot be directed to a single review. Lastly, the University-to-Users relationship is a many-to-one connection where multiple users can be associated with a single university. This relationship allows for the aggregation of user preferences and trends at a university level, letting us give targeted recommendations and insights based on the collective tastes of a university's student body.

Database Normalization

Functional Dependencies

Customer_Id→Email, Password, First_Name, Last_Name, Skin_Color_H, Skin_Color_S, Skin_Color_V

Clothing_Id → Type, Price, Image, URL

Brand_Name → Minority_Owned, Made_In_USA, Sustainability

Color_Name → Hue, Saturation, Value, Hex

Review_Id → Star_Rating, Fit, Text

University_Id → University_Name, Size, Location

3NF Conversion

Customers(Customer_Id, Email, Password, First_Name, Last_Name, Skin_Color_H, Skin_Color_S, Skin_Color_V)

Clothes(Clothing_Id, Type, Price, Image, URL)

Brands(Brand_Name, Minority_Owned, Made_In_USA, Sustainability)

Colors(Color_Name, Hue, Saturation, Value, Hex)

Reviews(Review_Id, Star_Rating, Fit, Text)

Universities(University_Id, University_Name, Size, Location

Motivation for 3NF

After carefully deciding entities, and creating the UML diagram, ensuring that the multiplicity relationships are accurate, and the relationships themselves are also accurate, we took a look at the database as a whole. We realized that the form we created was already in the form of 3NF, since for every table, every element inside it is dependent on that specific table's primary key, and no element is dependent on attributes from other tables. As a result, we also did not need to add an additional key since every element is taken into account already. Hence, we chose the 3NF method rather than the BCNF method.

Relational Schema

Customers(
Customer_Id: INT [PK],
Email:VARCHAR(225),
Password:VARCHAR(225),
First_Name:VARCHAR(225),
Last_Name:VARCHAR(225),
Skin_Color_H:INT,
Skin_Color_S:INT,
Skin_Color_V:INT,
University_Id:INT [FK to University.University_Id]
)

University(
University_Id:INT [PK],
University_Name:VARCHAR(225),
Size:INT,
Location:VARCHAR(225)
)

Opinions(
Customer_Id:INT [FK to Customers.Customer_Id],
Clothing_Id:INT [FK to Clothes.Clothing_Id],
Opinion_Type:VARCHAR(225),
(Customer_Id:INT, Clothing_Id:INT) [PK]
)

Purchases(
Customer_Id:INT [FK to Customers.Customer_Id],

Clothing_Id:INT [FK to Clothes.Clothing_Id],
(Customer_Id:INT, Clothing_Id:INT) [PK]
)

Clothes(
Clothing_Id:INT [PK],
Clothing_Color:VARCHAR(100) [FK to Clothing_Color.Color_Name],
Brand:VARCHAR(100) [FK to Brands.Brand_Name],
Type:VARCHAR(20),
Price:REAL,
Image:VARCHAR(225),
URL:VARCHAR(225),
)

Brands(
Brand_Name: VARCHAR(100) [PK],
Minority_Owned:BOOL,
Made_In_USA:BOOL,
Sustainability:INT
)

Matches(
Customer_Id:INT [FK to Customers.Customer_Id],
Color_Name: VARCHAR(100) [FK to Clothing_Colors.Color_Name],
(Customer_Id:INT, Color_Name:VARCHAR(100)) [PK]
)

Clothing_Colors(
Color_Name:VARCHAR(100) [PK],
Hue (H):INT,
Saturation (S):INT,
Value (V):INT,
Hex:VARCHAR(10)
)

Reviews(
Review_Id: INT [PK],
Customer_Id:INT [FK to Customers.Customer_Id],
Clothing_Id:INT [FK to Clothing.Clothing_Id],
Brand: VARCHAR(100) [FK to Brands.Brand_Name],
Star_Rating: REAL,

Fit: VARCHAR(100),
Text: VARCHAR(1000)
)