PHASE 2 : DATABASE PROTOTYPE

SWAPBOX

Sean Park / Devansh Kukreja / Chih-Sen Chang

RELATIONAL MODEL:

Relational model is formatted as follows: *Table - (**Primary Key**, Foreign Key)*

Subscriptions - (**user_assigned**, **start_date**, length, cost, name, monthly_quota, deliveries_used)
Users - (**id**, name, email, phone, gender, address, join_time)
Packages - (**id**, size, weight, to_user, from_user, from_swapbox, to_swapbox, to_address, from_address)
Assignments - (**package_id**, **carrier_id**, tracking_id)
Carriers - (**id**, name)
Swapboxes - (**id**, capacity, count, address)
Package_Statuses - (**package_id**, **time**, last_swapbox, next_swapbox)
Locations - (**address**, long, lat)

When converting our conceptual model to the final relational model, most table conversions were one-to-one, but we had to resolve the association entities and generalizations. We placed the `trackingID` association entity inside a new entity table called *Assignments* that links *Carriers* with *Packages*. For the association entity between *Packages* and *Statuses*, we dropped the `usersNotified` property (we decided it wasn't necessary for our user stories) and merged `timeUpdated` into the `Statuses` entity. We resolved our two generalizations by merging them into their parent entities. In order to avoid having potentially dangerous `NULL` values in our table, we added dummy values that clearly indicate them as not applicable.

FUNCTIONAL DEPENDENCIES:

Subscriptions - ***user_assigned, start_date*** *→ length, cost, monthly_quota, deliveries_used*

Users - ***Id*** *→ name, email, phone, gender, address, join_time*

Packages - ***Id*** *→ size, weight, to_user, from_swapbox, from_user, to_swapbox, to_address, from_address*

Assignments - ***package_id, carrier_id*** *→ tracking_id*

Carriers - ***Id*** *→ name*

Swapboxes - ***Id*** *→ capacity, count, address*

Package_Status - ***package_id, time*** *→ last_swapbox, next_swapbox*

Location - ***Address*** *→ long, lat*

NORMALIZATION:

Although most of our functional dependencies were already in BCNF form, the *Subscriptions* table still had a transitive dependency; we resolved it by splitting it into two tables. Below is the initial structure of *Subscriptions*:

Subscriptions - (**user_assigned**, **start_date** → length, cost, name, monthly_quota, deliveries_used)

Subscriptions was separated into *Subscriptions* and *Plans* as follows:

Subscriptions - (**user_assigned**, **start_date** → deliveries_used, plan_name)

Plans - (**name** → length, cost, monthly_quota)

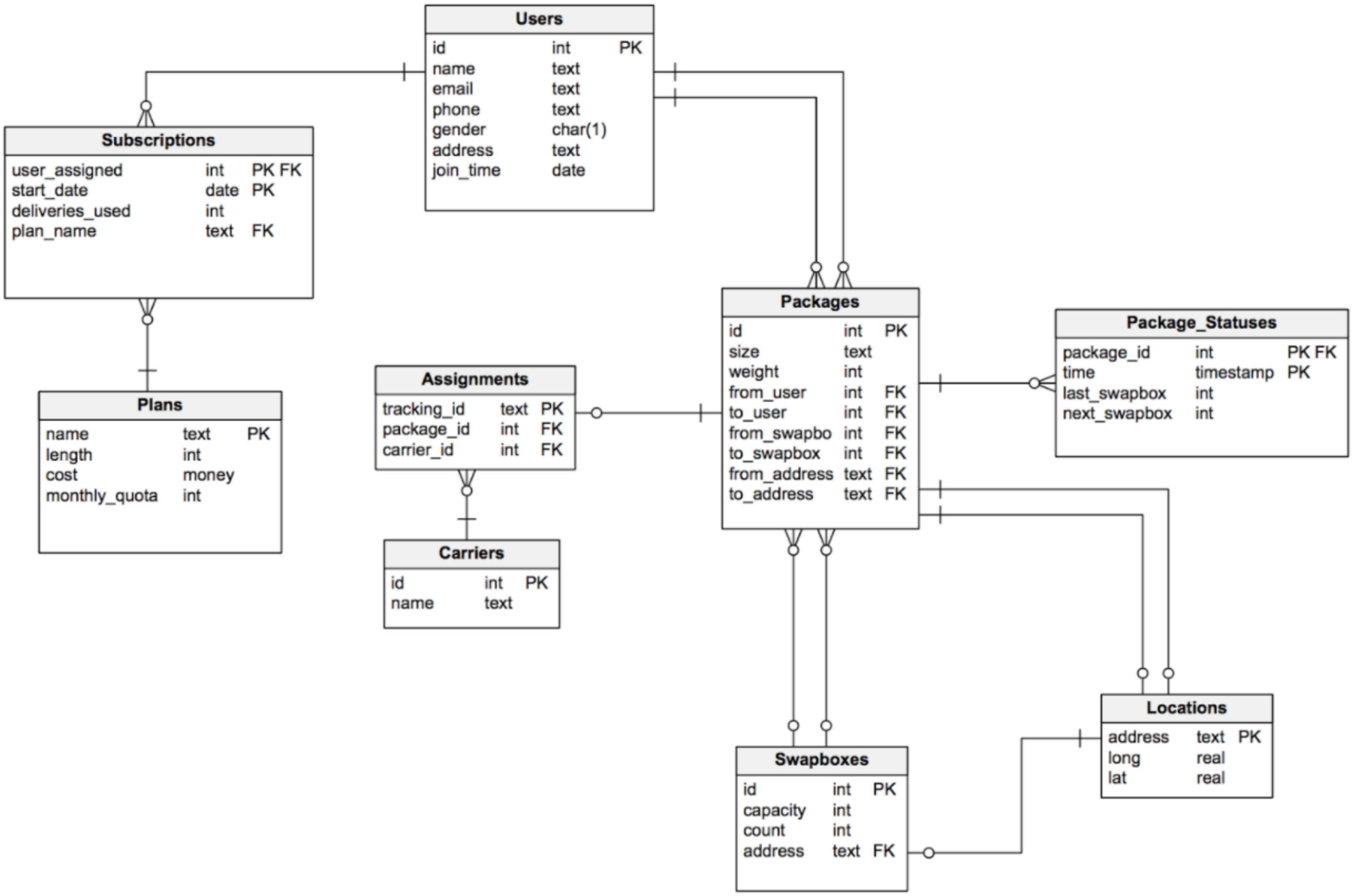
This was the only required change to normalize our database.

PHYSICAL MODEL:

Since our database is named `project_swapbox`, the command to execute the SQL statements was slightly different. In our `initialize.sql` file, we created our own database as `project_swapbox` so we can freely drop it to reset without losing data from other databases. Execute the following commands to initialize and display the database:

```
% psql -d postgres -U isdb16 -f initialize.sql
% psql -d project_swapbox -U isdb16 -f show_all.sql
```

Below is our physical model from Vertabelo:



Users		
id	int	PK
name	text	
email	text	
phone	text	
gender	char(1)	
address	text	
join_time	date	

Subscriptions		
user_assigned	int	PK FK
start_date	date	PK
deliveries_used	int	
plan_name	text	FK

Plans		
name	text	PK
length	int	
cost	money	
monthly_quota	int	

Assignments		
tracking_id	text	PK
package_id	int	FK
carrier_id	int	FK

Carriers		
id	int	PK
name	text	

Packages		
id	int	PK
size	text	
weight	int	
from_user	int	FK
to_user	int	FK
from_swapbo	int	FK
to_swapbox	int	FK
from_address	text	FK
to_address	text	FK

Package_Statuses		
package_id	int	PK FK
time	timestamp	PK
last_swapbox	int	
next_swapbox	int	

Swapboxes		
id	int	PK
capacity	int	
count	int	
address	text	FK

Locations		
address	text	PK
long	real	
lat	real	

QUERIES

Since our database is named `project_swapbox`, the command to execute the sql statements are slightly different:

```
% psql -d project_swapbox -U isdb16 -f simple_queries.sql
```

We have 6 simple queries and 4 complex queries. The 6 simple queries are as follows:

- Track the current number of packages each carrier is currently transporting
- Determine a user's most common Swapbox to receive at
- Determine a user's most common Swapbox to send out from
- Display all of a specific user's packages that arrived on a certain day
- Find the last known status/location of a given package
- Find the most active months

Following are the 4 complex queries. They should be executed in the following format:

```
% python complex_query_1.py
```

1. `add_package (size, weight, from_user, to_user, from_swapbox, to_swapbox, from_address, to_address)` takes in all the details for a package Before adding it to the *Packages* table, it must first cross-check the user's monthly quota against the deliveries they've used on their most recent subscription. If the user's allowed to ship it, it will add the package; otherwise it will throw an error.

2. `track_packages(userID)` takes in a userID, and prints all information about all the user's packages. It categorizes them into Undelivered, InTransit, and Arrived packages–counting them up and printing it out a list of the packages in each category.
3. `assign(packageID, carrierName)` takes in a `packageID` and `carrierName`, and assigns the package to the carrier, but before creating an entry in the *Assignments* table, it checks to see if a carrier by that name exists in our database. If they don't, then we create an entry for that carrier before creating an *Assignment* entity that links the `packageID` with the corresponding `carrierID`.
4. `renew_subscription(userID, today)` takes in a tuple of today's date in the form of (month, day, year) and inserts a new subscription into the database if renewal criteria have been met. The renewal criteria checks if the user has either used up his/her quota or if the subscription has expired.