c.moevus

Corentin's Hackathon results

Organization of the data

Trip data is organized as such:

- A trip is composed of segments
- Segments are composed of legs
- Legs are non-stop flights

For example, consider the following trip:

1. NYC -> LAX on 11/20
2. LAX -> SFO on 11/21
3. SFO -> NYC on 11/28

This is a multicity trip with 3 segments. Each segment can have one or several legs. For example, for the segment NYC -> LAX, there could be a direct flight NYC -> LAX, which would be one leg, or two flights NYC -> ORD, ORD -> LAX, which would be a non-direct flight of 2 legs, with a layover.

Note that this may not be the official terms, as segments seem to require the same aircraft is used, but this seems to be the terms Google uses, and I'll just use that.

Hence, to organize the trips:

- **trips**
  - id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  - segment_number INT UNSIGNED NOT NULL,
  - segment_id BIGINT UNSIGNED NOT NULL,
  - FOREIGN KEY (segment_id) REFERENCES segments (id) ON DELETE CASCADE ON UPDATE CASCADE,

    - UNIQUE `unique_segment_numbers` (id, segment_number),
- **segments**
    - id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    - leg_number INT UNSIGNED NOT NULL,
    - leg_id BIGINT UNSIGNED NOT NULL,
    - FOREIGN KEY (leg_id) REFERENCES legs (id) ON DELETE CASCADE ON UPDATE CASCADE,
    - UNIQUE `unique_legs` (id, leg_number),
- **legs**
    - id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    - origin INT UNSIGNED NOT NULL,
    - FOREIGN KEY (origin) REFERENCES airports (id) ON DELETE CASCADE ON UPDATE CASCADE,
    - destination INT UNSIGNED NOT NULL,
    - FOREIGN KEY (destination) REFERENCES airports (id) ON DELETE CASCADE ON UPDATE CASCADE
    - departure_time DATETIME(0) NOT NULL,
    - class CHAR(1) NOT NULL,
    - UNIQUE `unique_rows` (origin, destination, departure_time, class)
    - fields BLOB, *for storing permanent properties (airline, flight number, requested_class, duration, mileage, meal, aircraft, etc.) using dynamic columns*
- **airports**
    - id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    - name VARCHAR(200) CHARACTER SET utf8 NOT NULL,
    - code VARCHAR(3) CHARACTER SET ascii NOT NULL

Each row in *trips*, *segments* and *legs* is unique. The basic unit is the leg, a single non-stop flight.

Next, we want to organize the price information. Booking information relates to segments, while price information relates to trips. There is not booking information for legs, as far as I know, as legs are grouped into segments for booking. Hence, the price/booking part can be organized as such:

- **requests**
  - id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  - request_time DATETIME(0) NOT NULL,
- **trip_prices**
  - request_id BIGINT UNSIGNED NOT NULL,
  - FOREIGN KEY (request_id) REFERENCES requests (id) ON DELETE CASCADE ON UPDATE CASCADE,
  - trip_id BIGINT UNSIGNED NOT NULL,
  - FOREIGN KEY (trip_id) REFERENCES trips (id) ON DELETE CASCADE ON UPDATE CASCADE,
  - price float UNSIGNED NOT NULL,
  - fare_info BLOB, *for values that are related to the fare (refundable, taxes, etc.) using dynamic columns*
  - UNIQUE `unique_price` (request_id, trip_id)
- **booking_info**
  - segment_id BIGINT UNSIGNED NOT NULL,
  - FOREIGN KEY (segment_id) REFERENCES segments (id) ON DELETE CASCADE ON UPDATE CASCADE,
  - request_id BIGINT UNSIGNED NOT NULL,
  - FOREIGN KEY (request_id) REFERENCES requests (id) ON DELETE CASCADE ON UPDATE CASCADE,
  - fields BLOB, *for values that are impermanent (booking code, number of seats left, etc.) using dynamic columns*
  - UNIQUE `unique_fields` (segment_id, request_id)

The booking code reflects both the class (economy, first, business, economy premium, etc.) and the price tag within that class, which means that is both a property of the trip (what the user requested) and of the request (how much it will cost). To solve this issue, we'll split it in two: the class requested by the user (requested_class) and the booking code returned by the GDS (booking_code).

Choice of database

For simplicity reasons, I'll use MariaDB. Ultimately, I may switch to PostgreSQL. In addition, MariaDB supports GIS data, which may be useful for locating airports and cities and such.

Populating the database

I have 3 different sources of JSON files:

1. Amadeus
2. QPX
3. Golbibo

I need to convert the JSON file into a data.frame in R, and then reshape this dataframe into a structure similar to the database, and then write it into the database.