Part 1. Relational Algebra In this problem, you will use a few relations representing some fictitious

Southwest Airlines flights and airplanes. Use only this instance. `flights` contains information about flight routes on a particular day, departure time and the exact aircraft (`tail`) used to fly that particular flight. aircraft contains information about all airplanes that Southwest owns and operates. A smaller relation, `airtran_aircraft` contains information about airplanes that Southwest acquired in its purchase of Airtran in 2011. Tuples in `airtran_aircraft` also appear in `aircraft` if they were acquired by Southwest. [If curious, the aircraft types are described as B73G (Boeing 737-700), B738 (Boeing 737-800) and B38M (Boeing 737 MAX 8, or 737-8).]

from	to	flightnum	departure	tail
LAX	SFO	181	8am	N8751R
LAX	SJC	185	9am	N705SW
SJC	LAX	186	10am	N404WN
BUR	SJC	191	11am	N957WN
LAX	ATL	993	12pm	N7851A
MCO	CUN	991	1pm	N7827A
SJC	BUR	192	2pm	N709SW
SFO	LAX	182	3pm	N8751R
SJC	DAL	94	4pm	N705SW
SJC	PHX	99	5pm	N957WN

tail	type
N404WN	B73G
N705SW	B73G
N709SW	B73G
N8751R	B73G
N7851A	B38M
N7827A	B73G
N7854B	B73G
N7826B	B73G
N957WN	B738

tail	type
N7851A	B73G
N7827A	B73G
N7854B	B73G
N7826B	B73G

Exercises.

(a) Write a relational algebra expression that returns the number of flights flown by each `type` of aircraft. A flight is uniquely identified by its flight number (denoted `flightnum`). Each flight number is used for one take off and one landing. Your result should provide insight like "4 flights were flown by an airplane that is of type B738." `flightnum` is a flight number (i.e. Southwest flight 181) and not the number of flights flown.

Response:

$$_{\texttt{type}}\gamma_{\texttt{COUNT(flightnum)}\rightarrow\texttt{count}}(\texttt{flights} \bowtie \texttt{aircraft})$$

(b) In 2011, Southwest Airlines acquired Airtran. The relation `aircraft` contains Southwest owns, including those acquired from Airtran. The relation `airtran_aircraft` includes information about *only* Airtran's aircraft. Write a relational algebra expression that returns all flight numbers (`flightnum`) operated by aircraft that were *not* operated by Airtran.

Response:

$$\Pi_{\texttt{flightnum}}(\texttt{flights} \bowtie [\texttt{aircraft - airtran\_aircraft}])$$

(c) Most aircraft fly multiple routes in one day. For example, tail N705SW flies from LAX (Los Angeles) to SJC (San Jose, CA) and then flies from SJC (San Jose) to DAL (Dallas). Such schedules form a graph. Write the relational algebra expression that return the tail and where each plane starts and ends up after two flights: tail, origin (`from`), and `final_destination` (to after 2 flights). In the example earlier, the query would return N705SW, LAX and DAL since it started at LAX and ended up at DAL after two flights. A couple of notes and hints: (1) if a tail only flew one flight, it would not appear in the output, (2) you are essentially traversing a graph, and this is an example of a *self join*, (3) you need to somehow use the departure time and this is an example of a *non-equi-join*. **Be very careful with aliasing and renaming in this problem.**

Response:

$$\Pi_{\texttt{dest.tail}\to\texttt{tail},\texttt{src.from}\to\texttt{origin},\texttt{dest.to}\to\texttt{final\_destination}}$$
$$(\rho_{\texttt{src}}(\texttt{flights}) \bowtie_{(\texttt{src.tail = dest.tail} \wedge \texttt{dest.from = src.to} \wedge \texttt{src.departure < dest.departure})} \rho_{\texttt{dest}}(\texttt{flights}))$$

Part 2. SQL Schemas Use the Starship food delivery scenario from Homework 1 and look at the schema diagram in the solutions. Note that we modify it here slightly.

In Homework 1, we created a relational schema and diagram for the Starship example. In this problem, we will create a SQL schema using the `CREATE TABLE` syntax. This means we also need to pick the proper data types for each column. For a description of how Starship works, see Homework 1.

We need a table to represent a `robot`. Each of the following statements is designed to give you a hint as to the proper data type.

1. Each robot has an identifier `robot_id`, a number. Since Starship is a startup, we assume that there are no more than 10,000 robots.
2. Each robot has a flag `status` that marks it as online, offline (broken etc.), and lost/stolen. Each robot can have only one of these states at a time, and must have a state.

We need a table to represent a **customer** (user is a system keyword so I will not use it):

1. Each user has an identifier `user_id`, a number, and we assume that Starship has at most 500,000 users for now.
2. A user is just someone that installed the app, not necessarily someone that will use a robot. Thus, they may, or may not have a credit card number `ccnum` (exactly 16 digits) and expiration date `expdate`. Expiration dates usually look like MM/YY, but to make this simpler so you can use a more apparent data type, it is safe to assume that the card expires at midnight (00:00) on the 1st of the month. A credit card can only be associated with one user.
3. Each user must have an `email` address. Assume an email address length is at most 100. An email can only be associated with one user.
4. Each user must have a `name`. Assume a name cannot exceed 255 letters.

We need a table to represent an **order**. Only once food is placed into the robot do we enter an order in this table, and its journey begins. Each order is associated with:

1. a unique identifier `order_id`, a number. We expect there to be more than 50,000 orders.
2. exactly one user `user_id`, exactly one robot `robot_id` and exactly one restaurant `restaurant_id`.
3. a `start_time` and `end_time`, which includes the date. In the event that the order is canceled after being dispatched, or the robot fails, there may not be an end time.
4. a `delivery_location` as a GPS coordinate (a latitude/longitude pair). *Hint:* See the documentation here. Note that latitude and longitude together form a point on a Cartesian plane (actually a sphere, but we will assume Cartesian plane for this problem). If the order is canceled or an error occurs with the robot, there may not be a delivery location stored.
5. a required order status called `status` that can take on the values `delivered`, `canceled`, `error`.

Finally, we need a table to represent a **restaurant** which contains:

1. a unique identifier for the restaurant `restaurant_id`. Assume there are currently 10,000 restaurants.

2. the name of the restaurant which contains at most 255 characters.

3. a `region_id` which is a 4 digit alphanumeric code representing the geofenced region. For UCLA this could be UCLA or a code like 1919.

Exercise. Write the SQL schema for the tables discussed above using `CREATE TABLE`. Specify a primary key, or composite primary key using the correct syntax. Specify the proper foreign key relationship on each table (if they exist) using the proper syntax. Try to minimize storage space because we can always promote later.

```
CREATE TYPE robot_status AS ENUM (
    'online',
    'offline',
    'broken',
    'default'
);

CREATE TABLE robot (
    robot_id smallint NOT NULL UNIQUE,
    status robot_status NOT NULL DEFAULT 'default',
    PRIMARY KEY (robot_id)
);

CREATE TABLE customer (
    user_id int NOT NULL UNIQUE,
    ccnum numeric(16, 0) UNIQUE,
    expdate date,
    email varchar(100) UNIQUE,
    name varchar(255)
    PRIMARY KEY (user_id)
);

CREATE TYPE order_status AS ENUM (
    'in-progress'
    'delivered',
    'canceled',
    'error'
);

CREATE TABLE order (
    order_id int NOT NULL UNIQUE,
    user_id int,
    robot_id smallint,
    restaurant_id smallint,
    start_time timestamptz DEFAULT CURRENT_TIMESTAMP,
    end_time timestamptz,
    delivery_location point,
    status order_status NOT NULL,
    PRIMARY KEY (order_id),
    FOREIGN KEY (user_id) REFERENCES customer (user_id),
    FOREIGN KEY (robot_id) REFERENCES robot (robot_id),
    FOREIGN KEY (restaurant_id) REFERENCES restaurant (restaurant_id)
);

CREATE TABLE restaurant (
    restaurant_id smallint NOT NULL UNIQUE,
    name varchar(255),
    region_id numeric(4, 0) NOT NULL,
    PRIMARY KEY (restaurant_id)
```

```
);
```