DO NOT consult this solution set until after you have completed all work on your assignment. It is a violation of the Honor Code to view the solution set before your work is completed and turned in.

Assignment 5: The Entity-Relationship Model – SOLUTIONS

It is an honor code violation to look at these solutions before completing and submitting the CS121 assignment. This is also true if you are not presently taking CS121, but may do so in the future.

If you have already completed and submitted your assignment, feel free to read on!

CS121 – Introduction to Relational Databases Fall 2014

DO NOT consult this solution set until after you have completed all work on your assignment. It is a violation of the Honor Code to view the solution set before your work is completed and turned in.

Airline Database *(100 points)*

For this problem you will be creating a database schema to record some basic information for an airline, such as what flights are available, what airplanes are used on each flight, and what tickets have been purchased. We will approach this problem in stages to keep it from becoming unwieldy; at the end, you will assemble the various parts into a complete database system.

This problem will give you some design guidance, but there are many places where it will not. You must figure out what mapping cardinalities and participation constraints should be enforced by the database, based on the descriptions below, as well as your intuitions of what would "make the most sense" given the problem domain.

Problem 1: Flights and Airplanes *(15 points)*

For this problem you must create an E-R diagram that can record what flights are available, what kind of airplane is used for each flight, and what seats are available on each kind of plane. Your diagram should represent flights, kinds of aircraft used for flights, and the seats available on those flights. It is not necessary to create other kinds of entities besides these. Additional details are given below.

Flight Information

Each flight has a flight number (a short string, e.g. "QF11" or "QF108"), a date (e.g. "2007-05-21"), and a time (e.g. "14:10:00") associated with it. A given flight number will be reused on different days, but the combination of flight number and date will be unique. You can see how it would definitely be best to keep the date and time values separate in this schema.

Flights also have a source and destination airport, which should be represented by their 3-letter International Air Transport Association (IATA) airport code[1] (e.g. "LAX" for Los Angeles International Airport, or "SYD" for Sydney Airport).

> *(One could imagine creating another entity-set to represent airports, but you definitely do not need to do this for the assignment.)*

There should also be a way to mark a flight as domestic (within the country) or international (between two countries). The reason for this is that travelers must provide additional information when on an international flight.

Airplane Information

Each flight also has a certain kind of aircraft associated with it. This allows customers to reserve specific seats for the flight. Each kind of aircraft is represented by several pieces of information:

- The manufacturer's company (e.g. "Airbus" or "Boeing")
- The aircraft's model (e.g. "A380" or "747-400")

[1] http://en.wikipedia.org/wiki/International_Air_Transport_Association_airport_code

- The IATA aircraft type code,[2] a 3-character value that the International Air Transport Association uses to specify the kind of airplane that a flight uses. The value is unique for every kind of aircraft. For example, the Airbus A380 is designated by the code "380", and the Boeing 747-400 is designated by the code "744". This value can include letters and numbers.

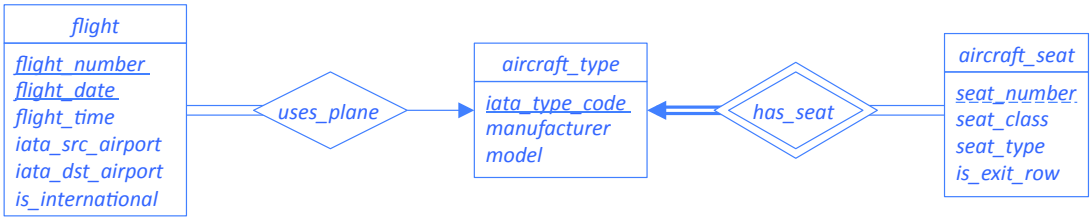
Along with each type of aircraft the company has, information about the seats available on that aircraft must be available. Individual seats have these details associated with them:

- A seat number such as "34A" or "15E" – the numeric component specifying the row of the seat, and the letter specifying the position within the row. (You can represent seat numbers as a single field; there's no reason to break the values apart.)
- A "seat class" such as "first class", "business class", or "coach". (Different classes can be represented as 1-character codes, or as strings, or whatever you prefer.)
- A "seat type" specifying whether the seat is an aisle, middle, or window seat
- A flag specifying whether the seat is in an exit row

Obviously, each seat number will be unique on a specific kind of aircraft (i.e. on the 747-400, "17B" will specify exactly one seat), but different kinds of aircraft will definitely have seats with overlapping seat numbers. This suggests that it would make sense to represent seats as a weak entity-set.

Solution:

Here is an E-R diagram that represents the details outlined in this section:



There isn't a lot of nuance to this part of the design, except that we would like to constrain that every kind of aircraft has at least one seat; otherwise, it doesn't make sense to use that kind of aircraft in a commercial airline.

It is probably good to also enforce a candidate-key constraint on (*manufacturer*, *model*) in the *aircraft_type* entity-set.

Problem 2: Airline Customers *(15 points)*

Next you must create an E-R diagram to represent customers of the airline. Customers fall into two categories – "purchasers," who are responsible for buying tickets, and "travelers," who actually do the traveling. These sets of customers can certainly overlap, but they are not required to: people may buy tickets for their own use, and/or they may also buy tickets

[2] http://www.airlinecodes.co.uk/arctypes.asp

for other people. You can see that some kind of generalization hierarchy would likely be a useful approach for this schema.

Note that for this problem, you are not specifying how to represent tickets. Tickets turn out to be a bit tricky, so you will do this in Problem 3, when you assemble the various parts of the database schema together.

Purchasers and Purchases

As stated before, "purchasers" are the people who purchase tickets for specific flights. Note that purchasers might buy tickets for people other than themselves, such as a department administrator buying flights for other people in the department. Purchasers have the following information:

- A first and last name (should be represented as separate attributes in the database)
- A contact email address
- One or more contact phone numbers
- Optional payment information, including a 16-digit credit card number, expiration date (MM/YY), and 3-digit card verification code. This data may all be *null* if the purchaser doesn't trust the airline to properly secure this data… ☺

A "purchase" is a collection of one or more tickets bought by a particular purchaser in a single transaction. (As stated earlier, don't worry about incorporating tickets at this point; we will get there.) A purchaser can make multiple purchases. A purchase includes the following information:

- An integer ID that uniquely identifies the purchase
- A timestamp specifying when the purchase occurred
- A six-character "confirmation number" that the purchaser can use to access the purchase. In the real world, confirmation numbers are not guaranteed to be unique, but we will simplify our database design by enforcing that they are unique across all purchases in the database. (If we take our confirmation numbers from the capital letters A-Z and the digits 0-9, we can represent about 2.1 billion purchases before we start having trouble.)

Travelers

Travelers are the people who are actually going on a particular flight. Travelers have some similar attributes as purchasers:

- A first and last name (should be represented as separate attributes in the database)
- A contact email address
- One or more contact phone numbers

Travelers do not have payment information.

As mentioned before, some flights are domestic, and other flights are international. For international flights, travelers must provide these additional details:

DO NOT consult this solution set until after you have completed all work on your assignment. It is a violation of the Honor Code to view the solution set before your work is completed and turned in.

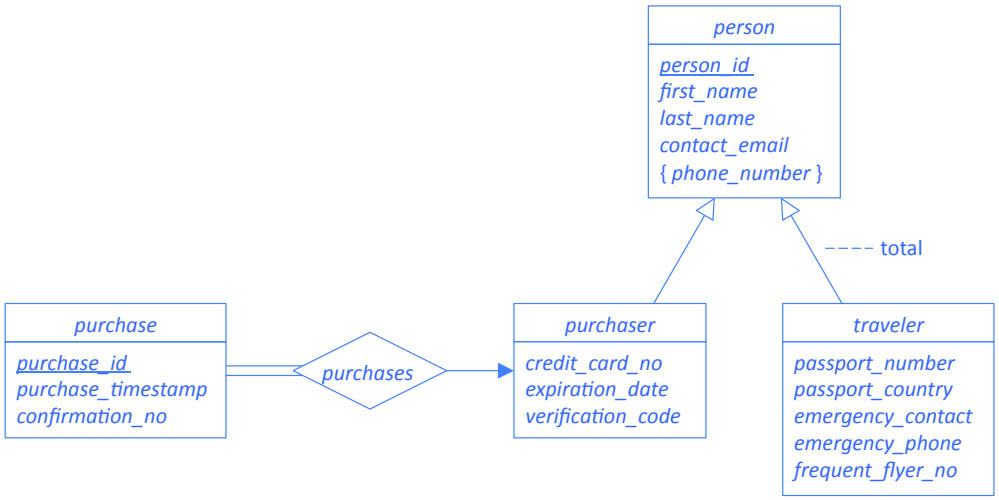
- A passport number. The length of the passport number varies from country to country, but it is usually pretty long, so provide up to 40 characters for this value.
- The country of citizenship for the passport.
- The name of an emergency contact (the first and last name can be together in a single field, if you wish).
- A single phone number for the emergency contact.

Travelers are not required to provide these details immediately; they only need to be entered at least 72 hours before the flight. Therefore, it is perfectly acceptable to allow *null* values in the above fields.

Finally, this airline has a frequent-flyer program for travelers to participate in. Therefore, you should also include an attribute to hold the traveler's frequent flyer number, if they have one. Although we call it a "frequent flyer number," the value is actually comprised of both numbers and letters, and is always 7 characters long.

Solution:

Given the overlap between purchasers and travelers, it makes a lot of sense to use some kind of specialization to represent them. Since a person could be both a purchaser and a traveler then it makes sense to use overlapping specialization in the design. Here is an E-R diagram that captures the above description:



The *confirmation_no* attribute on *purchase* is a candidate key.

It could be argued that a purchaser should have at least one purchase (and therefore the arrow on the *purchases* relationship-set should be a double-lined arrow to indicate total participation), but we will allow customers to have records in the *purchaser* entity-set before they have made any purchases.

We could also enforce a candidate key on the pair of attributes (*passport_number*, *passport_country*) in the *traveler* entity-set, as well as on (*frequent_flyer_no*), but it

isn't critical to do so, as other external checks would ensure that these values are indeed valid.

Problem 3: Bringing Everything Together – Tickets! *(15 points)*

At this point you should have an E-R diagram representing kinds of aircraft, flights and seats, and another E-R diagram representing travelers, purchasers and purchases. The thing that ties these two parts together is the ticket. For this problem you will create a complete E-R diagram representing the entire database schema, with tickets tying together the concepts of purchases, travelers, flights and seats. You will need to figure out the appropriate relationships, mapping cardinalities and participation constraints to satisfy the description below, as well as whatever other constraints would simply "make sense" to include in the system.[3]

Tickets themselves are very simple:

- Each ticket should have a unique ID (unique across all tickets ever issued by the airline, not just unique to a specific flight).
- Each ticket should also store the sale price of the ticket. Airlines frequently vary the prices of seats on a particular flight, both by where the seat is located (e.g. first class or coach), as well as how close the purchase-date is to the flight-date. The schema must be able to represent this variation in pricing. You should assume that tickets will always cost less than $10,000 each on this airline.

What is much more complicated is the relationship between tickets and the other entities in our database. Here are the relevant details:

- A purchase may include one or more tickets.
- A purchase may include multiple tickets for the same traveler, e.g. for a multi-leg flight. Of course, a purchase may include tickets for multiple travelers as well. In other words, we really don't want to constrain what tickets appear in the purchase, just that there must be at least one ticket in each purchase.

- Each traveler will have one or more tickets associated with them.
- As with purchasers, we really don't want to impose any other constraints between tickets and travelers. For example, a traveler can even hold multiple tickets on the same flight (sometimes necessary for very large/fat people, or people traveling with pets or small children). So again, we don't want to impose too many constraints between travelers and tickets, just that each traveler will have at least one ticket.

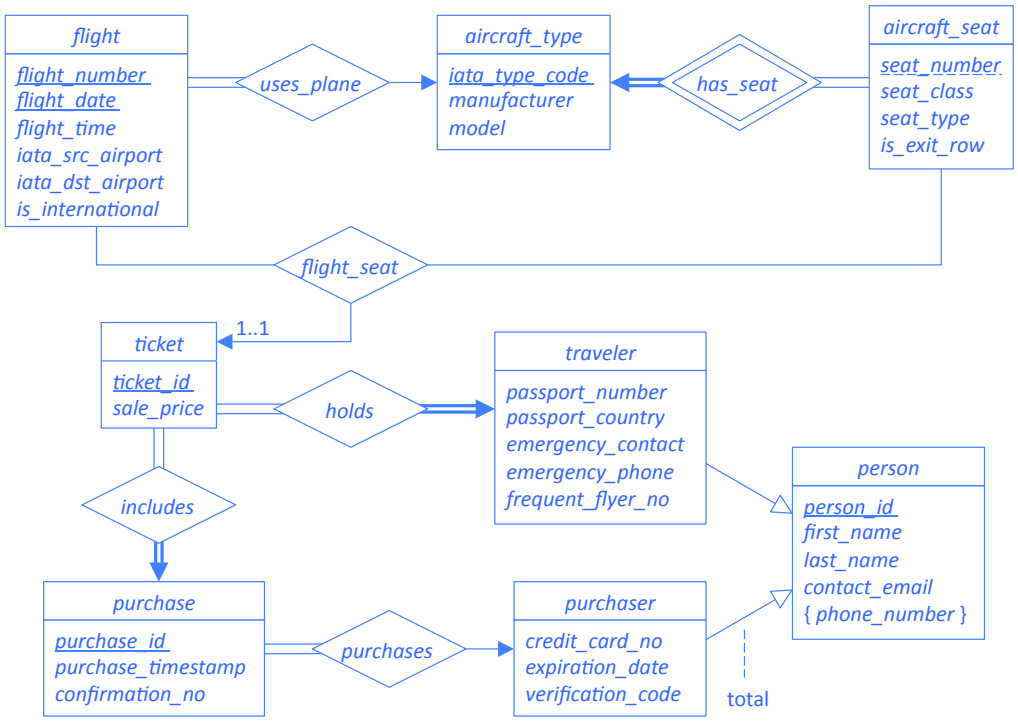
- Each ticket represents a particular *seat* on a particular *flight*. It should not be possible to associate a specific ticket with multiple seats, or with multiple flights, in the database.
- Similarly, it should only be possible to give each ticket to one traveler.

[3] An example of this that we have discussed all term, is that every loan must have at least one customer associated with the loan; otherwise, our bank won't be in business for very long. You must think of similar considerations for our airline.

- These two constraints together should guarantee that when we give a ticket to a traveler, they won't have to share that seat on the flight with anybody else… an uncomfortable prospect at best!

Solution:

Tickets are a challenging aspect of the design because of all the constraints we want to enforce with them. Here is an E-R diagram of one design that comes pretty close to capturing everything in the above description:



The relationship between tickets and travelers, at least, is easy to understand: Each traveler must have at least one ticket, and each ticket may be held by exactly one traveler. This is captured by the *holds* relationship-set, which is a many-to-one mapping from tickets to travelers, with total participation enforced on both sides of the relationship-set.

Similarly, the relationship between tickets and purchases is also easy to understand. Each ticket must be part of exactly one purchase, and a purchase must include one or more tickets. (It is conceivable that a ticket might be granted to a traveler without an associated purchase, but we will ignore that possibility in our design.)

Finally we come to flights, seats and tickets. Each combination of flight and seat should correspond to at most one ticket (there may be no ticket for a flight and seat if that seat hasn't been sold). This can be modeled via a ternary relationship involving *flight*, *aircraft_seat* and *ticket*, with an arrow towards *ticket*. This will ensure that we cannot have two (flight, seat, ticket) relationships (QF108, 15C,

1234) and (QF108, 15C, 1256). However, it does not prevent us from having the pair of relationships (QF108, 15C, 1234) and (QF11, 25E, 1234). Thus, we introduce an additional numeric constraint on the *ticket* side of the ternary relationship, specifying that each ticket may participate in exactly one relationship instance. This will disallow the second situation from occurring. (In the mapping to the relational model, we would already have a primary key on *flight_seat* of primary_key(*flight*) ∪ primary_key(*aircraft_seat*); the numeric constraint would enforce a second candidate key of primary_key(*ticket*) on this relationship-set, producing the desired set of constraints.)

There is one final issue in the above design, one that is not easy to resolve. The ternary relationship-set associates a flight and a seat with a ticket, but there is no restriction guaranteeing that the flight and the seat must use the same aircraft. Resolving this issue would likely require more advanced E-R modeling techniques such as aggregation, but that is beyond the scope of this assignment. It may be possible to resolve this issue in the implementation schema, however.

Problem 4: Translation to Relational Model *(15 points)*

Once you have completed your entire E-R diagram in Problem 3, translate it into a relational model schema. Write up your schema in the file **`airline-schema.txt`**. For each relation schema, make sure to identify any candidate keys besides the primary key, and also any foreign keys and what they reference.

If you perform any schema combinations then you must explain why it is a good idea to combine the schemas.

Solution:

The E-R diagram has a whole bunch of many-to-one relationship-sets with total participation on the "many" side of the mapping, so the design is rich with opportunities to combine schemas. This will help in simplifying the schema, as well as enforcing some of the total-participation constraints much more easily.

Before getting to that, we will start with the part of the design focusing on customers:

person(*person_id*, *first_name*, *last_name*, *contact_email*)

person_phone_nums(*person_id*, *phone_number*)

- *person_id* is a foreign key to *person.person_id*
- (*person_id*, *phone_number*) is the primary key

traveler(*person_id*, *passport_number*, *passport_country*, *emergency_contact*, *emergency_phone*, *frequent_flyer_no*)

- *person_id* is a foreign key to *person.person_id*

purchaser(*person_id*, *credit_card_no*, *expiration_date*, *verification_code*)

- *person_id* is a foreign key to *person.person_id*

Since each purchase must be associated with exactly one purchaser, it makes sense to combine the *purchase* entity-set and the *purchases* relationship-set:

purchase(*purchase_id*, *purchase_timestamp*, *confirmation_no*, *person_id*)

- *person_id* is a foreign key to *purchaser.person_id*. (Note that having a foreign key to *person* or *traveler* would be incorrect.) A NOT NULL constraint must also be placed on this attribute to enforce total participation of *purchase* entities in the *purchases* relationship-set.

Aircraft types and seats can be translated into these relational model schemas:

aircraft_type(*iata_type_code*, *manufacturer*, *model*)

- (*manufacturer*, *model*) can be enforced as a candidate key as well

aircraft_seat(*iata_type_code*, *seat_number*, *seat_class*, *seat_type*, *is_exit_row*)

- *iata_type_code* is a foreign key to *aircraft_type.iata_type_code*

Each flight uses some aircraft, so again we can combine the schemas for *flight* and *uses_plane* together:

flight(*flight_number*, *flight_date*, *flight_time*, *iata_src_airport*, *iata_dst_airport*, *is_international*, *iata_type_code*)

- *iata_type_code* is a foreign key to *aircraft_type.iata_type_code*
- A NOT NULL constraint must also be placed on *iata_type_code* to enforce the total participation constraint.

Tickets have many-to-one relationships with both travelers and purchases, with total participation on the "many" side, so we can combine both the *holds* and *includes* relationship-sets into the relational-model representation of the *ticket* entity-set:

ticket(*ticket_id*, *sale_price*, *purchase_id*, *person_id*)

- *purchase_id* is a foreign key to *purchase.purchase_id*
- *person_id* is a foreign key to *traveler.person_id*. Again, note that it would be incorrect to have this foreign key to *purchaser* or *person*.

This leaves the flight_seat relationship-set that must be represented:

flight_seat(*flight_number*, *flight_date*, *iata_type_code*, *seat_number*, *ticket_id*)

- (*flight_number*, *flight_date*) is a foreign key to *flight*
- (*iata_type_code*, *seat_number*) is a foreign key to *aircraft_seat*
- *ticket_id* is a foreign key to *ticket*

- Because of our 1..1 numeric constraint, *ticket_id* is also a candidate key of this schema

Recall that we had an earlier issue of not being able to constrain that the flight and the seat used the same aircraft. However, in this flight_seat schema we have all details necessary to enforce this additional constraint: we can enforce that the set of attributes (*flight_number*, *flight_date*, *iata_type_code*) is a foreign key to *flight*, as long as we enforce an additional superkey containing the same attributes. This is allowed since it contains the primary key of the *flight* relation, so the database will be able to ensure that our flights and seats indeed use the same kind of aircraft.

Alternative Schema for *ticket* Entity-Set and *flight_seat* Relationship-Set

Note that both the *ticket* relation and the *flight_seat* relation have a candidate key of *ticket_id*. Further, every ticket will participate in exactly one relationship instance in *flight_seat*. This means we can even combine these two schemas into a single schema without introducing nulls or causing other problems. The new ticket schema would be as follows:

ticket(*ticket_id*, *sale_price*, *purchase_id*, *person_id*, *flight_number*, *flight_date*, *iata_type_code*, *seat_number*)

- (*flight_number*, *flight_date*, *iata_type_code*, *seat_number*) is also a candidate key on this schema
- (*flight_number*, *flight_date*) is a foreign key to *flight*
- (*iata_type_code*, *seat_number*) is a foreign key to *aircraft_seat*
- Again, we can add a foreign key constraint on the attributes (*flight_number*, *flight_date*, *iata_type_code*), to *flight*. (This would replace the other foreign key constraint specified two points earlier.)

It is not surprising that our attributes on the *ticket* relation schema match what appears on an actual boarding pass rather closely!

Problem 5: Translation to SQL DDL *(25 points)*

Next, translate your relational model schema into SQL DDL. Your DDL file needs to load successfully into MySQL. Write your DDL in a file called **`make-airline.sql`**. At the top of your file you should also include **`DROP TABLE`** commands so your schema can be reloaded easily. (Your **`DROP TABLE`** commands must respect referential integrity.)

Here are additional guidelines:

- Specify all relevant not-*null* constraints, default values where appropriate, and make sure to include all primary / foreign / candidate key constraints.
- Specify cascade operations where there are existence-dependencies between records in the database schema.
- Your DDL must be clearly documented. Every table should have at least *some* documentation explaining its contents, particularly tables that might have critical

meaning within the database schema. Any fields whose meaning isn't completely obvious should also include some explanatory comments. If you have cascade operations or other unusual constraints (e.g. multi-column primary or foreign keys, **CHECK** constraints, etc.) then briefly comment these as well.

- Make sure to choose reasonable column types and size/precision limits.

Solution:

Here is SQL DDL for the schema design given earlier. Note that cascade constraints are required "where there are existence-dependencies between records." One could argue that this would include any relationship-set, since relationship instances can only exist if their participating entities exist. However, we will take this definition narrowly, and only say there are existence dependencies between weak entities and their owning entities, and between multivalues and the entities they are a part of.

```
-- Get rid of existing tables, if any.
-- Respect referential integrity so we don't run into errors.

DROP TABLE IF EXISTS flight_seat;
DROP TABLE IF EXISTS ticket;
DROP TABLE IF EXISTS flight;
DROP TABLE IF EXISTS aircraft_seat;
DROP TABLE IF EXISTS aircraft_type;
DROP TABLE IF EXISTS purchase;
DROP TABLE IF EXISTS traveler;
DROP TABLE IF EXISTS purchaser;
DROP TABLE IF EXISTS person_phone_nums;
DROP TABLE IF EXISTS person;


-- This table holds details common to both travelers and purchasers,
-- including first name, last name, contact emails, and phone numbers
-- (in the person_phone_nums table).
--
-- People can be either a traveler, or a purchaser, or both.  It should not
-- be allowed to have a record in this table, without a corresponding record
-- in at least purchaser or traveler.  However, this total specialization
-- constraint is expensive to enforce, so we just don't enforce it.
CREATE TABLE person (
    person_id INTEGER AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    contact_email VARCHAR(200) NOT NULL
);


-- This table holds zero or more contact phone numbers for each person.
-- Since it represents a multivalued attribute of people, we have cascade
-- constraints for both deletion and updates on person.
CREATE TABLE person_phone_nums (
    person_id INTEGER REFERENCES person (person_id)
        ON DELETE CASCADE ON UPDATE CASCADE,

    -- I don't really specify any fixed format for phone numbers, in case we
    -- need to store international phone numbers, etc.
    phone_num VARCHAR(20),
```

```
    PRIMARY KEY (person_id, phone_num)
);


-- Travelers are one kind of person, with relevant travel information.  Since
-- this table represents a subclass of people, we have cascade constraints
-- for both deletion and updates on person.
CREATE TABLE traveler (
    person_id INTEGER PRIMARY KEY REFERENCES person (person_id)
        ON DELETE CASCADE ON UPDATE CASCADE,

    -- It is possible that different countries have different numbers of
    -- digits for passport numbers.
    passport_number VARCHAR(40),     -- Allowed to be NULL

    -- Countries are denoted via 2-digit country codes following the ISO-3166-1
    -- alpha-2 specification.
    passport_country CHAR(2),        -- Allowed to be NULL

    -- The full name (first and last) of an emergency contact
    emergency_contact VARCHAR(100),  -- Allowed to be NULL

    -- The phone number of an emergency contact
    emergency_phone VARCHAR(20),     -- Allowed to be NULL

    --
    frequent_flyer_no CHAR(7)        -- Allowed to be NULL
);


-- Purchasers are another kind of person, with relevant purchase information.
-- Since this table represents a subclass of people, we have cascade
-- constraints for both deletion and updates on person.
CREATE TABLE purchaser (
    person_id INTEGER PRIMARY KEY REFERENCES person (person_id)
        ON DELETE CASCADE ON UPDATE CASCADE,

    -- We only support 16-digit credit cards.  Sorry, American Express.
    credit_card_no CHAR(16),      -- Allowed to be NULL

    -- Stored as 4 digits, in the form "MMYY"
    expiration_date CHAR(4),         -- MM/YY, allowed to be NULL

    -- Card verification code, typically 3 digits on credit cards.  (Again,
    -- sorry, American Express.)
    verification_code CHAR(3)    -- Allowed to be NULL
);


-- A purchaser can make one or more purchases, which are represented in this
-- table.  A purchase is comprised of one or more tickets.  Tickets have
-- prices associated with them, so we don't actually have any monetary details
-- in this table.
CREATE TABLE purchase (
    purchase_id INTEGER AUTO_INCREMENT PRIMARY KEY,

    -- The timestamp of when the purchase was made
    purchase_timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
    -- "Confirmation numbers" are 6-digit values comprised of the charcters
    -- A-Z and the digits 0-9, allowing a customer to look up their purchase.
    -- Note that these confirmation numbers are enforced to be unique.
    confirmation_no CHAR(6) NOT NULL UNIQUE,

    -- The purchaser who made this purchase.  Note that we must reference
    -- purchaser, not traveler or person (which would be incorrect).
    person_id INTEGER NOT NULL REFERENCES purchaser (person_id)
);


-- This table represents a kind of aircraft, like a Boeing 747-400 or an
-- Airbus A380.
CREATE TABLE aircraft_type (
    -- The IATA aircraft type code for the kind of aircraft.  For example,
    -- the A380's code is "380", and the 747-400's code is "744".
    iata_type_code CHAR(3) NOT NULL PRIMARY KEY,

    -- Company that manufactures the aircraft.
    manufacturer VARCHAR(40) NOT NULL,

    -- The company's model name for the aircraft.
    model VARCHAR(40) NOT NULL,

    -- It is conceivable that two companies would use the same model name for
    -- an aircraft, so we will include the manufacturer name as well to
    -- ensure uniqueness.
    UNIQUE (manufacturer, model)
);


-- This table represents individual seats on a certain kind of aircraft.
-- Seat numbers are not sufficient to uniquely identify seats, so we must
-- also reference the aircraft type (i.e. this is a weak entity set).  Since
-- this is the case, there is an existence-dependence between seats and their
-- corresponding aircraft type, so we have a cascade constraint for deletion
-- or updates on aircraft_type.
CREATE TABLE aircraft_seat (
    -- The kind of aircraft is necessary to uniquely identify the seat.
    iata_type_code CHAR(3) REFERENCES aircraft_type (iata_type_code)
        ON DELETE CASCADE ON UPDATE CASCADE,

    -- The "seat number", which usually includes both a row number and a seat
    -- position.  For example, "15F" or "47A"
    seat_number VARCHAR(5),

    -- Indicates the class of the seat, taken from the set of values "first",
    -- "business", "economy".
    seat_class VARCHAR(10) NOT NULL,

    -- Indicates the general position of the seat, taken from the set of
    -- values "aisle", "middle", "window".
    seat_type VARCHAR(10) NOT NULL,

    -- True if the seat is in an exit row, false otherwise.
    is_exit_row BOOLEAN NOT NULL,

    PRIMARY KEY (iata_type_code, seat_number)
);
```

```
-- Represents a flight on a given date, between two airports, using a
-- specific type of aircraft.
CREATE TABLE flight (
    -- Flight numbers are not unique of themselves; they are reused.  But, on
    --  a given date, they are unique.
    flight_number VARCHAR(10),

    -- The date and time of the flight, represented separately since the
    -- flight number and date are part of the primary key for the table.
    flight_date DATE,
    flight_time TIME NOT NULL,

    -- Source and destination airports for the flight, denoted as IATA
    -- 3-character airport codes (e.g. "LAX" or "BUR").
    iata_src_airport CHAR(3) NOT NULL,
    iata_dst_airport CHAR(3) NOT NULL,

    -- Indicates if the flight is international or not.
    is_international BOOLEAN NOT NULL,

    -- The kind of aircraft used for the flight.
    iata_type_code CHAR(3) REFERENCES aircraft_type (iata_type_code),

    PRIMARY KEY (flight_number, flight_date),

    -- This allows us to ensure that the flight and seat on a ticket actually
    -- use the same kind of aircraft.  Note that we still need the primary
    -- key enforced separately.
    UNIQUE (flight_number, flight_date, iata_type_code)
);


-- An individual ticket for a seat on a given flight.  The sale price of
-- the ticket is included, since we vary the prices of tickets based on
-- their class, type, position, and the date of the ticket's purchase.
CREATE TABLE ticket (
    ticket_id INTEGER AUTO_INCREMENT PRIMARY KEY,

    -- Tickets should cost less than $10000 (i.e. $9999.99 is the max price)
    sale_price NUMERIC(6, 2) NOT NULL,

    -- ID of the purchase that the ticket is part of.  Not null, so that
    -- every ticket must be part of some purchase.
    purchase_id INTEGER NOT NULL REFERENCES purchase (purchase_id),

    -- ID of the traveler.  Not null, so that every ticket must be associated
    -- with some traveler.
    person_id INTEGER NOT NULL REFERENCES traveler (purchase_id)
);


-- An association between a flight, a seat on that flight, and a ticket for
-- that flight and seat.  We impose some extra constraints so that we can
-- ensure that the seat is for the same type of aircraft that the flight
-- uses; we don't have another place to enforce this constraint.
CREATE TABLE flight_seat (
    -- The primary key of the flight we are on.
    flight_number VARCHAR(10),
    flight_date DATE,
```

```
    -- The primary key of the seat we will occupy.
    iata_type_code CHAR(4),
    seat_number VARCHAR(5),

    -- The primary key of the ticket associated with the flight and seat.
    ticket_id INTEGER NOT NULL,

    -- Ensures that each (flight, seat) combination only appears once - don't
    -- hand out the same (flight, seat) combination twice!
    PRIMARY KEY (flight_number, flight_date, iata_type_code, seat_number),

    -- Ensures that each ticket can only be associated with one
    -- (flight, seat) combination!
    UNIQUE (ticket_id),

    -- Ensures that the flight is valid, and (in combination with the next
    -- foreign key constraint) also that the flight and seat use the same
    -- type of aircraft.
    FOREIGN KEY (flight_number, flight_date, iata_type_code)
        REFERENCES flight (flight_number, flight_date, iata_type_code),

    -- Ensures that the seat is valid.
    FOREIGN KEY (iata_type_code, seat_number)
        REFERENCES aircraft_seat (iata_type_code, seat_number),

    FOREIGN KEY (ticket_id) REFERENCES ticket (ticket_id)
);
```

Problem 6: Use-Cases and SQL Queries *(15 points; 5 points per query)*

Here are some common use-cases that might need to be performed against the database schema. Write the SQL queries for these operations to see how easy (or hard) they are to do against your schema. Put your answers in the file **`airline-queries.sql`**.

a) We need to provide a way to display all the purchase history for a single customer via the company website. To demonstrate how this will work, write a SQL query that will retrieve all purchases and associated ticket information for the customer (aka "purchaser") with ID 54321. The results should be ordered by these columns: purchase date (descending order), flight date, traveler last name, traveler first name (all other columns ascending order).

Solution:

```
-- The problem is ambiguous about what columns to include in the result, so
-- these are just the ones that I think are most interesting/relevant, plus
-- the ones that are
SELECT purchase_id, purchase_timestamp, sale_price, last_name, first_name
FROM purchase
     JOIN ticket USING (purchase_id)
     JOIN flight_seat USING (ticket_id)
     JOIN person ON (ticket.person_id = person.person_id)
WHERE purchase.person_id = 54321
ORDER BY purchase_timestamp DESC, flight_date, last_name, first_name;
```

b) Write a query that reports that total revenue from ticket sales for each kind of airplane in our flight booking database, generated from flights with a departure time within the last two weeks. Include *all* kinds of airplanes in the database, whether they were used for flights in the last 2 weeks or not.

Solution:

```
-- Again, the problem is somewhat ambiguous on what to include, so these are
-- the columns that are most interesting to me.
SELECT iata_type_code, manufacturer, model, SUM(sale_price) AS total_revenues
FROM aircraft_type NATURAL LEFT JOIN (ticket NATURAL JOIN flight_seat)
WHERE flight_date BETWEEN CURRENT_DATE - INTERVAL 2 WEEK AND CURRENT_DATE
GROUP BY iata_type_code, manufacturer, model;
```

c) Write a query that reports all travelers on international flights that have not yet specified all of their international flight information.

Solution:

```
-- Yet again, the problem is ambiguous on what columns to include, so these
-- are the columns that are most interesting to me.
SELECT person_id, last_name, first_name
FROM flight NATURAL JOIN flight_seat NATURAL JOIN ticket
     NATURAL JOIN traveler NATURAL JOIN person
WHERE is_international AND
      (passport_number IS NULL OR passport_country IS NULL OR
       emergency_contact IS NULL OR emergency_phone IS NULL);
```

The predicate is pretty annoying, but it can be rewritten with a **`CONCAT()`** call since any argument being **`NULL`** will cause the result to be **`NULL`**.

```
SELECT person_id, last_name, first_name
FROM flight NATURAL JOIN flight_seat NATURAL JOIN ticket
     NATURAL JOIN traveler NATURAL JOIN person
WHERE is_international AND
      CONCAT(passport_number, passport_country,
             emergency_contact, emergency_phone) IS NULL;
```