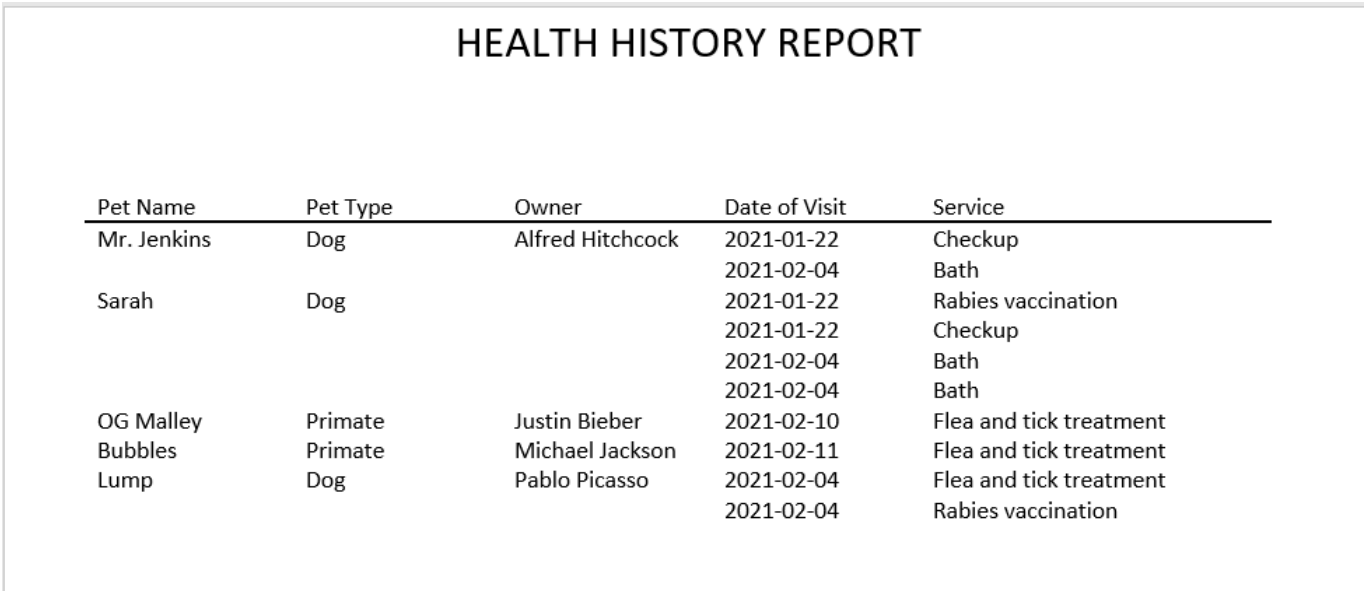
Tutorial for database design

In this tutorial, you'll build a database for Star-Vet, the veterinarian clinic to the stars. The clinic needs to track services provided to pets so that they can:

- Create invoices to customers for services provided to their pets.
- Produce a pet medical history, showing all services provided to any pet over time.

Here is an example of these documents:



HEALTH HISTORY REPORT

Pet Name	Pet Type	Owner	Date of Visit	Service
Mr. Jenkins	Dog	Alfred Hitchcock	2021-01-22	Checkup
			2021-02-04	Bath
Sarah	Dog		2021-01-22	Rabies vaccination
			2021-01-22	Checkup
			2021-02-04	Bath
			2021-02-04	Bath
OG Malley	Primate	Justin Bieber	2021-02-10	Flea and tick treatment
Bubbles	Primate	Michael Jackson	2021-02-11	Flea and tick treatment
Lump	Dog	Pablo Picasso	2021-02-04	Flea and tick treatment
			2021-02-04	Rabies vaccination

INVOICE

Star-Vet Hollywood, CA

Invoice Date: 02-12-2021

Paul McCartney

41 West 54th Street
New York, NY 10019-5404

Pet Name	Service	Amount
Martha	Heart worm test	$30.00
Martha	Flea and tick treatment	$12.00
Martha	Clip nails	$14.00
Martha	Bath	$25.00
Thisbe	Rabies vaccination	$45.00
Thisbe	Heart worm test	$30.00
Thisbe	Flea and tick treatment	$12.00
Thisbe	Clip nails	$14.00
Thisbe	Bath	$25.00
Thisbe	Parvo vaccination	$45.00
Subtotal		$252.00
8% Tax		$20.16
Total due		$272.16

Logical design

Look at the sample reports. Notice the individual facts (pieces of data), and think about what "things" those facts apply to. For example, on the invoice there's a list of services performed on a pet. On the history report, there's a list of pets and services performed.

From these reports, you identify attributes you need to track:

- pet name
- pet type (dog, cat, etc.)
- pet owner name
- address (street, city, state, postal code)
- service performed
- service date
- service cost

These attributes seem to belong to several different entities, so you organize the attributes into entities:

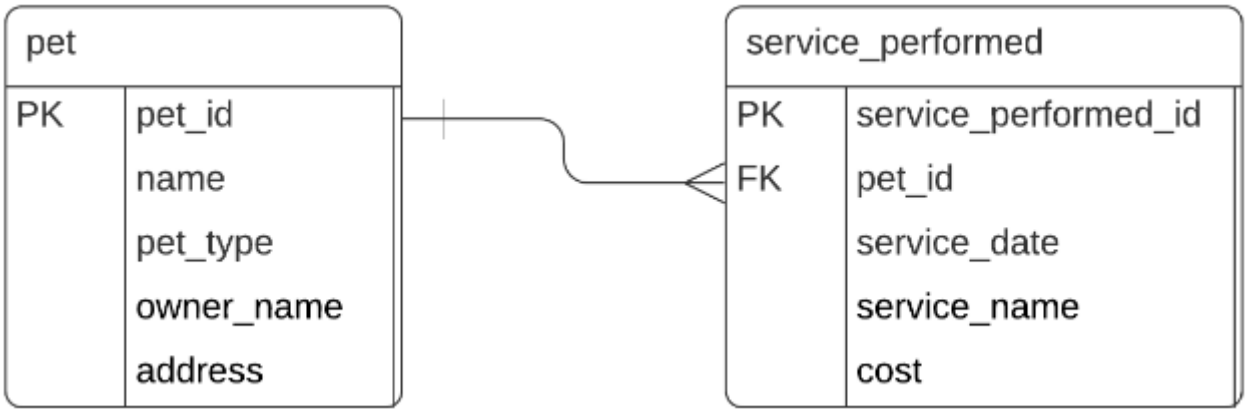
- pet
    - name

    - type (dog, cat, etc.)
    - owner_name
    - address (street, city, state, postal code)
- service_performed
    - pet_name
    - service_date
    - service_name
    - cost

You also must define a primary key for each entity. A primary key is an identifier. Given the primary key, you must be able to locate exactly one item that matches the primary key. You might think `name` would be a good identifier. But you probably know several dogs named "Buddy", so a pet's name isn't a good identifier.

Instead, it's common to create a new attribute, usually called `(entityname)_id`, to use as a primary key for an entity. In most database systems, the database automatically assigns the next id each time you insert a row.

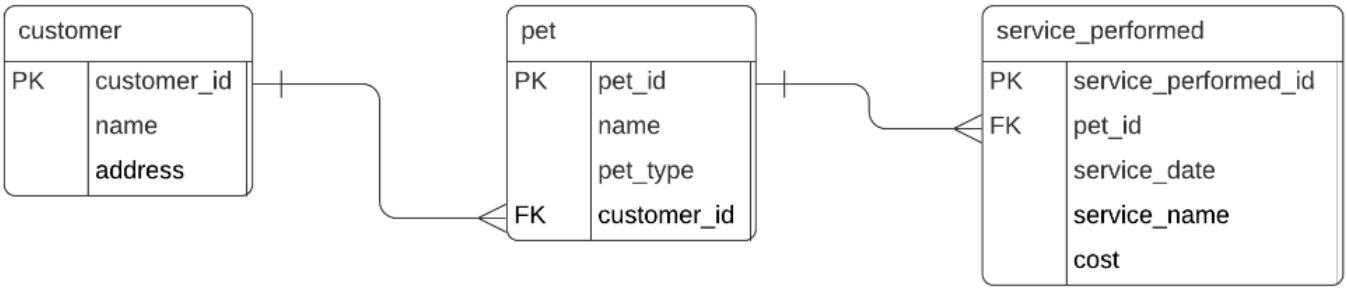
You can update your model by adding primary keys:



Notice that when you added the primary keys, you also replaced `pet_name` in the `service_performed` entity with the foreign key `pet_id`. From that relationship, you can get to the pet's name, owner, and address.

One question you must ask yourself throughout this process is "for any fact (attribute), is that fact dependent upon the primary key of its entity?" `name`, `type`, and `owner` are all determined by the `pet_id` of the pet. However, `address` isn't dependent directly on the pet. It's dependent on the owner. If the owner changes address, the pet, and any other pet owned by the same person, moves with them.

Your solution is to factor out another entity of owners. Pet owners are the customers of the vet, so you'll call that table `customer`. Then *reference* the customer from the `pet` entity through the foreign key:



Now ask the same question about the attributes of `service_performed`: do all of these facts depend directly on the key? The `cost` of the service is dependent on the type of service (the `service_name`). For example, the cost of a bath is the same for one dog versus another dog.

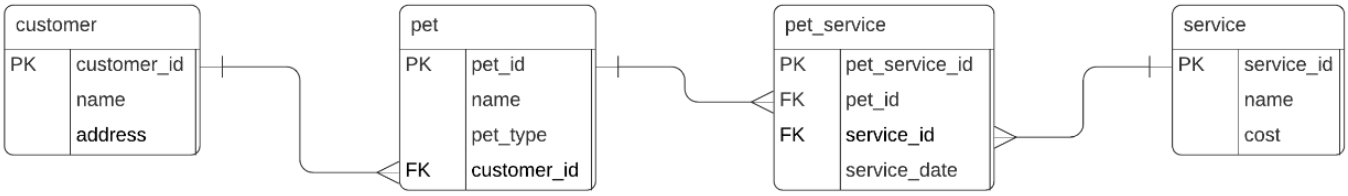
Once again, you factor `service` into its own entity:

- service
  - service_id (PK)
  - name
  - cost

This entity represents the list of services that the clinic performs.

Any one pet can have multiple services performed upon it, and you can deliver one service to multiple pets. This means there's a *many-to-many* relationship between `pet` and `service`. The association between these two entities—one pet and one service—is the delivery of a service to a pet on a certain date. You'll call this *associative entity* `pet_service`.

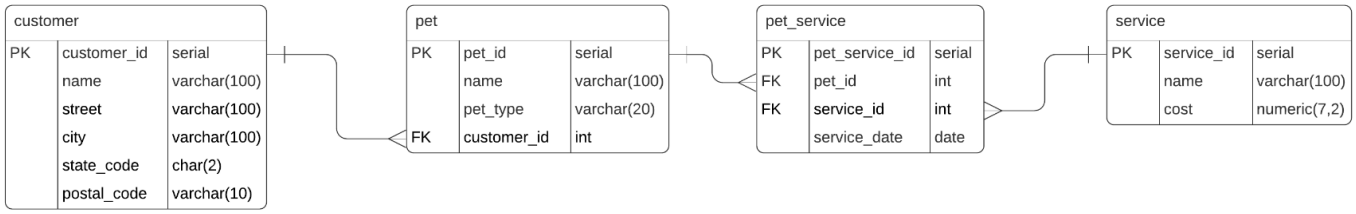
Your re-factored model now looks like this:



> Note that in `pet_service`, you *could* have selected `pet_id` plus `service_id` as the *composite primary key*. However, since a primary key must be unique, doing so limits the performance of a particular service to particular pet only once. For example, Bruce the dog could only get one bath in his lifetime. This isn't what you want. Since the combination of `pet`+`service` may *not* be unique, you introduced another serial column for the primary key, `pet_service_id`.

Physical design

Your physical design enhances the logical model by stating such things as data types, column lengths (of string data) and the presence of auto-incrementing keys:



Physical implementation

Now that you've designed your vet clinic data model, the next step is to create the physical implementation of the database. You do this by writing SQL DDL (data definition language) statements.

The primary DDL statements for adding, changing, or removing tables are `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE`. This tutorial gives you an opportunity to practice each.

Getting started

To get started, open a database query tool, like the **pgAdmin** Query Tool, with a connection to the `PetDB` database. If your PostgreSQL server doesn't have a `PetDB` database, you need to create one. Refer to the *Intro to Tools* unit for instructions on creating a new database.

Next, open the file `database-design-tutorial.sql` in the query tool window.

Step One: Create the `customer` table

When creating DDL for a database, it's best to create the *independent* tables first, followed by the *dependent* ones. An independent table has no foreign keys. For example, `customer` doesn't have any foreign key columns, so it's *not dependent* on the existence of another table. On the other hand, `pet_service` *depends* on the existence of both the `pet` and `service` tables, so you'll create that last.

Under the **Step 1:** comment, add this DDL:

```
/*****************************************************************************
-- Step 1: Create the customer table
*****************************************************************************/
-- Table customer
CREATE TABLE customer(
    customer_id serial NOT NULL,
    name varchar(100) NOT NULL,
    street varchar(100) NULL,
    city varchar(100) NULL,
    state_code char(2) NULL,
    postal_code varchar(10) NULL,
    CONSTRAINT PK_customer PRIMARY KEY (customer_id)
);
```

Some notes on this DDL:

- `CREATE TABLE` is the DDL command to add a new table to the schema.
- The data type `serial` indicates to PostgreSQL that this is an auto-numbered column. When you insert a row into the table, PostgreSQL determines the next serial-number and places that value into the column, guaranteeing a unique value. All relational DBMSs have this feature. However, there isn't an ISO standard for how to accomplish this. If you work in another DBMS, you might use a different technique.
- For most string columns, you use the `varchar` data type because the string may vary in size. You must specify a maximum length. For `state_code`, you use `char`, because state codes are always two characters.
- You may define constraints, such as primary key or foreign key, inside the `CREATE` statement, as the code shows. You'll learn later how to define a constraint on an existing table by using an `ALTER` statement.

Insert test data for the `customer` table

The tutorial includes some test data so you can check your work. Find the comment labeled **Step 1a:**. Uncomment the `INSERT` and `SELECT` statements that follow the comment.

> Tip: You can use the shortcut Ctrl + . to quickly uncomment the code in pgAdmin.

Now execute the script. You'll see nine rows returned, from *Alfred Hitchcock* to *Paul McCartney*.

Step Two: Make the script idempotent

Execute the script again. This time, you see an error similar to this:

```
ERROR:  relation "customer" already exists
```

You can't run the script more than once because after the tables exist, you can't create them. When you write a script like this, you want to make the script *idempotent*. This means that no matter how many times you execute the script, you end up with the same results.

To do this, at the start of your script, you'll remove all the tables from the database. In DDL, you do this with `DROP TABLE`. Then you proceed to re-create the tables.

Under the **Step 2:** comment (note this is *before* the Step 1 code), add this DDL to drop tables:

```
/*****************************************************************************
-- Step 2: Drop all database objects to start with an empty database
*****************************************************************************/
DROP TABLE IF EXISTS pet_service;
DROP TABLE IF EXISTS pet;
DROP TABLE IF EXISTS service;
DROP TABLE IF EXISTS customer;
```

Notes on this DDL:

- `DROP TABLE` removes a table completely from the database.
- The `IF EXISTS` clause indicates to the DBMS to only drop the table if it exists. Without this clause, you receive an error when you try to drop a non-existent table.
- To successfully drop a table, *no other tables can depend upon it*. For example, you can't drop `customer` if the `pet` table references it. So, you drop them in the order from *most dependent* to *independent*—in other words, the opposite of the order you create them.

Execute the script multiple times. The script drops and re-creates the `customer` table each time.

Step Three: Create the `service` table

The `service` table is also independent—meaning it has no foreign keys—so create that next.

Under the **Step 3:** comment, add this DDL:

```
/*********************************************************************
-- Step 3: Create the service table
*********************************************************************/
-- Table service
CREATE TABLE service(
    service_id serial NOT NULL,
    name varchar(100) NOT NULL,
    cost numeric(7,2) NOT NULL,
    CONSTRAINT PK_service PRIMARY KEY (service_id)
);
```

Notes on this DDL:

- You chose the data type `numeric` for the `cost` column, which allows 5 digits to the left of the decimal point, and 2 digits to the right (total of 7 digits).

Insert test data for the `service` table

Uncomment the `INSERT` and `SELECT` statements that follow the **Step 3a:** comment.

Now execute the script. You'll see seven rows returned, from *Bath* to *Rabies vaccination*.

Step Four: Create the `pet` table

You have two tables left to create: `pet` and `pet_service`. Since `pet_service` depends upon—or has a foreign key to—`pet`, you create `pet` next.

Under the **Step 4:** comment, add this DDL:

```
/*********************************************************************
-- Step 4: Create the pet table
*********************************************************************/
-- Table pet
CREATE TABLE pet(
    pet_id serial NOT NULL,
    name varchar(100) NOT NULL,
    type varchar(20) NOT NULL,
    customer_id int NOT NULL,
    CONSTRAINT PK_pet PRIMARY KEY (pet_id),
    CONSTRAINT FK_pet_customer FOREIGN KEY(customer_id) REFERENCES customer
(customer_id)
);
```

Notes on this DDL:

- You included a foreign key constraint and a primary key constraint on this table.
- `pet` has a foreign key to the `customer_id` column of the `customer` table. It's standard practice to name that column with the name of the foreign attribute, so the column's name is `customer_id`.

Insert test data for the `pet` table

Uncomment the `INSERT` and `SELECT` statements that follow the **Step 4a:** comment.

Now execute the script. You'll see 14 rows returned, from *Alfred Hitchcock's dog Sarah* to *Michael Jackson's chimp Bubbles*.

Step Five: Add another constraint to the `pet` table

The `pet-type` column allows any string 20 characters or less. That means you can add a pet "Lemon" or "Tractor." However, your vet clinic only services certain types of pets.

You can add a **check constraint** to the column to specify a condition to check before PostgreSQL allows the data. At the time of `INSERT` (as well as `UPDATE`), PostgreSQL tests the condition. If it's `true`, PostgreSQL inserts (updates) the data. If not, you receive an error.

Under the **Step 5:** comment, add this DDL:

```
/*****************************************************************************
-- Step 5: Add a check constraint to the pet table
*****************************************************************************/
ALTER TABLE pet
    ADD CONSTRAINT CHK_type CHECK (type IN ('Dog', 'Bird', 'Cat', 'Reptile',
'Fish', 'Primate'));
```

Notes on this DDL:

- This is an `ALTER TABLE` statement. You use it to modify the schema of an existing table.
- You *could have* written this constraint in the `CREATE TABLE` statement along with the primary and foreign key constraints. Doing it this way provides you an opportunity to use an `ALTER TABLE` statement.
- The condition in the parentheses is the same type of condition you might expect to see in a `WHERE` clause of a `SELECT` statement.

Step Six: Create the `pet_service` table

The last table you must create is `pet_service`. Under the **Step 6:** comment, add this DDL:

```
/*****************************************************************************
-- Step 6: Create the pet_service table
*****************************************************************************/
-- Table pet_service
CREATE TABLE pet_service(
    pet_service_id serial NOT NULL,
    pet_id int NOT NULL,
    service_id int NOT NULL,
    service_date date NOT NULL,
    CONSTRAINT PK_pet_service PRIMARY KEY (pet_service_id),
    CONSTRAINT FK_pet_service_pet FOREIGN KEY(pet_id) REFERENCES pet (pet_id),
    CONSTRAINT FK_pet_service_service FOREIGN KEY(service_id) REFERENCES service
(service_id)
);
```

Notes on this DDL:

- This table defines a primary key plus two foreign keys to associate the `pet` and `service` tables into a many-to-many relationship.
- You use data type `date` to track the date of the service.

Insert test data for the `pet_service` table

Uncomment the `INSERT` and `SELECT` statements that follow the **Step 6a:** comment.

Now execute the script. You'll see 22 rows returned, dating from Jan 22, 2021 to Feb 12, 2021.

Step Seven: Write the health report query

Now that you've created the database and populated some sample data, you can test your data model.

You'll create a query to get the information you need to populate the Health History report shown previously in this tutorial. This lists services performed, ordered by customer, then by pet, then in chronological order.

At the bottom of your script, type the following SQL:

```
-- Get data for the Health History Report: by customer, pet, and date
SELECT p.name AS "Pet Name", p.type AS "Pet Type", c.name AS "Owner",
ps.service_date AS "Date", s.name AS "Service"
    FROM customer c
    JOIN pet p ON p.customer_id = c.customer_id
    JOIN pet_service ps ON ps.pet_id = p.pet_id
    JOIN service s ON ps.service_id = s.service_id
    ORDER BY c.name, p.name, service_date;
```

Execute the script. You'll see results that list each customer, their pets, and the services the clinic delivered to each pet (22 services delivered in total). From this data, you could create the Health History report shown earlier.

Step Eight: Write the invoice query

The last step is to create a query to get the information you need to populate the Invoice as shown previously in this tutorial.

An invoice has customer information, plus all the services performed on any of the customer's pets on a given date. For this example, get the data needed to invoice Paul McCartney for his visit on February 12, 2021.

At the bottom of your script, type the following SQL:

```
-- Get data for the Invoice for McCartney visit of 2/12
SELECT ps.service_date AS "Date", c.name, c.street, c.city, c.state_code,
c.postal_code, p.name AS "Pet Name", s.name AS "Service", s.cost
    FROM customer c
    JOIN pet p ON p.customer_id = c.customer_id
    JOIN pet_service ps ON ps.pet_id = p.pet_id
    JOIN service s ON ps.service_id = s.service_id
    WHERE c.name = 'Paul McCartney' AND service_date = '2021-02-12'
    ORDER BY p.name
```

Execute the script. You'll see data showing the 10 total services delivered to Paul McCartney's two dogs, Martha and Thisbe, on February 12. It also shows the customer's full address. From this data, you could create the Invoice shown earlier.

Next steps

Further data model testing

You built a useful data model to serve the needs of the clinic. Suppose Star-Vet tells you they have thought of some other uses for this data. Can you come up with the SQL queries to satisfy these needs?

- The clinic wants to send follow-up "thank you" emails with an invitation to take a survey to customers who visited the clinic yesterday. So you need a query to find the names of all customers, plus their pets' names, who had any service performed on a given date.
- For a business analysis, the clinic wants a list of the services provided in the past year, along with the number of times the clinic performed each service, in order of the most popular service to least.
- The clinic wants to send out reminder cards to prompt customers to bring their pets in for a checkup. They need a list of all pets who *haven't* had a checkup in the past year.

Changes to the data model

Suppose the clinic came to you and asked for these enhancements. How would your data model change to support these enhancements?

- The clinic found that multiple humans may bring in the same pet over time. In other words, a pet can have multiple owners. How might you support this in your data model?

- The clinic found that their pricing model is too rigid. The cost of services depends not only on the service rendered, but also on the type of pet performed upon. For example, a dog bath may cost $25, a cat bath $20, and primate bath $125. How might you support this in your data model?