Data Modeling

An Overview of Effective Database Design

By Courts Carter & David Rolston

Information Engineering is the disciplined approach to dynamic requirements gathering and software engineering. Utilizing Joint Application Development and RAD techniques it pinpoints business process flow: Its end product is the Data Model. the working blue print of what will be in the database.

A Data Model is usually referred to an Entity-Relationship Diagram - a graphic diagram depicting interrelationships between data elements.

Data Modeling

Database Design requires a rigorous design methodology. To design a RDRMS you create an Entity-Relationship (ER) diagram. In relational database parlance this discipline involves the rules of normalization (Nixon was going to Normalize relations with China remember?) So when Codd formulated his rules for relational database design he called them Normal forms. A proper database design is “normalized”.

Forget about all the lower level details and think about things at a more object oriented level. This begins with 3 key ideas:

1. Identify things or objects of importance real or imagined. Entities resolve to a single table or set of tables. Entitles: People, Computers, Offices, Countries, Books

2. For each entity identify its properties. These become its attributes: “Book” has an “Author”. Book has a “Title”. It has a “Publisher”, “Contents”, etc. Attributes are columns.

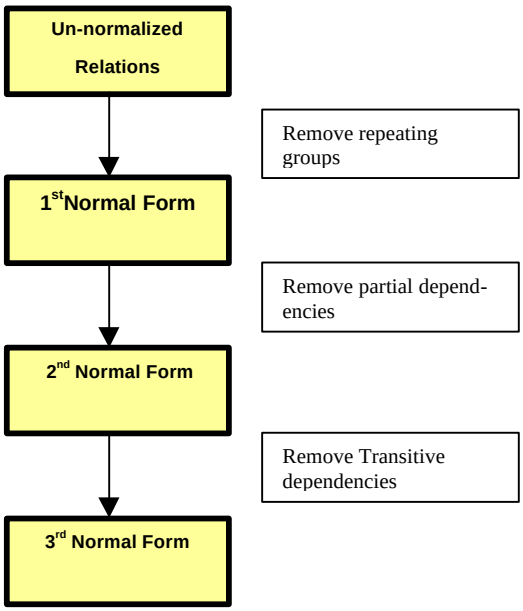
3. Entities can be related to each other. Authors write Books. Books are written by Authors. Join Authors to Books where the Author = “Stephen King” and you get a table which includes rows for “Carrie”, “The Shining” and “Pet Cemetery”.

Normalization

Normal forms are an attempt to make sure that you don’t destroy true data or create false data in your database. One of the ways of avoiding errors is to represent a fact only once in the database, since if a fact appears more than once one of the instances of it is likely to be an error. A man with two watches can never be sure what the correct time is. Or, succinctly:

One Fact In One Place

This process of table design is called normalization. the various steps of which are described by the following rules.



Rather than trying to fully explain the theoretical steps involved in normalizing a database I will first state the rules and then quickly apply them to a real-world example. First, however, there is time for a soapbox and an explanation couple of conventions.

Soapbox: Primary Keys

In having done several database systems it has become an unquestioned rule that I (nearly) always use surrogate keys. It is often the case that there are one or two truly beguiling candidate keys which have occasionally tempted me, nonetheless, I choose to create an internal surrogate key.

The classic example is Employee where two really good candidate keys nearly always exist: SSN and Employee ID. While I am second to none in my trust in the government and HR it is my belief that it is too easy to make a typo when entering data and for this reason alone am of the mind that the user should never get to directly enter a key.

If a credit agency enters an incorrect SSN one's credit history will be thrashed for months before it can all be sorted out.

Even on simple tables like a table States (Alaska, Arizona, etc) I would not use the StateCode or abbreviation as the key, because, again, data entry is too prone to errors and who really knows whether MI is the official abbreviation for Michigan or Minnesota anyway?

1st Normal Form (1 NF)

1st Normal Form (1NF) - Must be a table and each column contains atomic (single) values. Consider phone table - You could store city and state in a field called citystate (Memphis, TN). This would be bad… how do you find all records for TN? An index can't be applied effectively because the statecode is a string at the end of this field. With millions of records, you are going to tablescan (top to bottom)

Also remove repeating groups. For example: one of our systems has a Customer table with columns for day_phone, night_phone. These are both "phone numbers". Also what do you do to add Fax? How do you query for the phone# 555-9999?

1NF Rule: Remove repeated attributes or groups of attributes and place them in new entities.

2nd Normal Form (2NF)

2nd Normal Form (2NF) - Must be 1NF and all attributes depend on primary key. Simple primary key tables are already in 2NF. Composite keys may need to be separated into one or more tables.

Rule: Remove attributes dependent on only part of the primary key.

3rd Normal Form (3NF)

3rd Normal Form (3NF) - If 2NF and has no "transitive dependencies". All non-key attributes must be mutually independent.

From example above, SERVERNAME needs to move out of login into its on table. Also HOMESERVER needs to move out into its own table.

After 3NF things get obscure:

Boyce-Codd Normal Form (BCNF)

Boyce-Codd Normal Form (BCNF) - Enhances 3NF. Every determinant (attribute which determines values of other attributes) is a candidate (unique) key.

4th Normal Form (4NF)

4th Normal Form (4NF) - 3NF or BCNF and there are no multi-valued dependencies. (dependent key returns multiple records that have 2 or more attributes with the same value)

5th Normal Form (5NF)

5th Normal Form (5NF) - Too obscure to even describe in this amount of time!

6th Normal Form (6NF)

6th Normal Form (6NF) - Even worse.

Kennel Case Study

Requirements

You are hired to design a database for a kennel that offers veterinary services. After interviewing the owners and employees of the kennel you are able to arrive at a short list of requirements. Knowing that the budget is tight you create two lists: one for features which must be incorporated into the design ("can't live withouts") and another for those things which everyone agrees "would be nice" but are not essential to be included in the first version of the product.

Needs:

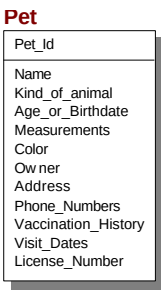
- want to track pet visits;
- send reminders and bills to pet owners;
- report on the average number of visits and the average length of a Pet's stay;
- track any kind of animal appearing on the doorstep;

Would be nice features:

- maintain a complete history of measurements taken each time an animal is brought in for an examination;

So we do some brainstorming with the client and come up with our first entity, "Pet". Here is what we know we want to capture and a simple Entity-Relationship Diagram:

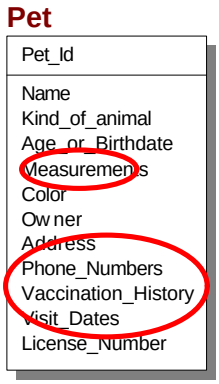
name
kind of animal
measurements (height, weight, etc.)
color
age or birthdate
owner
address
phone numbers
shots (i.e. vaccination history)
visit dates



Take a moment to understand what this ERD is showing us: the table's name appears over the box, the Primary Key appears as the first column in the box, but set off from the other columns via a horizontal line.

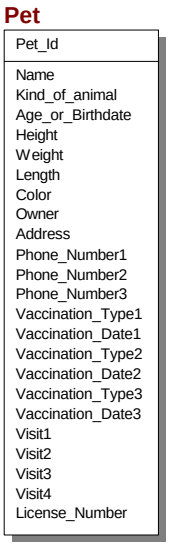
Right now our model is said to be in Zero-Normal Form.

1st Normal Form: Remove Repeating Groups



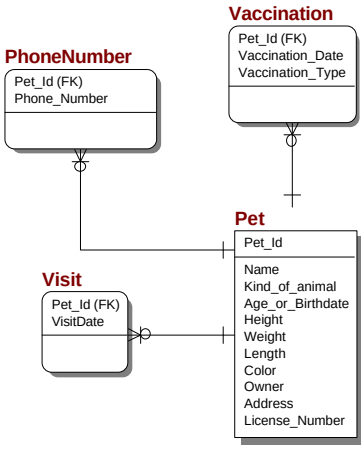
Need to remove those columns which contain non-scalar data: i.e. those that contain lists. A good clue indicating that a column contains non-scalar data is if the column name ends in an "s" or is otherwise plural. We have several: Measurements, Phone_Numbers, and Visit_Dates. You might miss it but the History column is also array data. We must replace this columns with columns which may only hold one fact ("one fact in one place," remember?).

We have a couple of options for correcting this problem and thus getting the data into Second Normal Form: we can replace these "mega columns" with discrete columns which will hold exactly the number of facts we need, or we can create separate tables to store the data. As we shall see we have hit upon another of my religious issues, but first I'll appear objective.



We could do the following: replace the single Phone_Number column with three Phone_Number columns named, aptly, Phone_ Number1, Phone_ Number_2, and Phone_Number3. We then do the same with Visits and Vaccination History. For Measurements we can be more specific. If we did this our ERD looks like the one at the top of the next column.

Now, this would work, but it violates some logical beliefs I hold dear. First, what is so magical about three phone numbers? Why not four, one, or six?



The same question may be applied to Vaccination and Visit. The number of columns we have created is arbitrary, so when the inevitable happens, some Pet arrives for their fifth visit, what do we do? Call the guy that designed our database and have them correct it? This is great if you are the data architect and happen to be a contractor paid by the hour, but if I were the users I would be annoyed – even if I was the one who suggested that four Visits would be "more than enough".

What we can instead do is remove the offending columns and place them in separate tables as shown here:

Okay, this one requires quite a bit of explanation. Let's look at the Visit table. In it we see that the Primary Key (which is shown above the line) is a Compound Key (recall the definition of a compound key as merely a key made up of more than one column) in this case made up of the Pet_Id column and the VisitDate. The Pet_Id column also happens to be a Foreign Key (a Foreign Key is merely a column holds the Primary Key which is foreign to the table). Before this becomes too confusing lets put some sample data into the tables:

Pet

Pet_Id	Name	Animal	Age	Owner
201	Spot	Dog	6	Samantha
202	Snowflake	Cat	4	Chris
203	Tibbles	Horse	12	Lou
206	Sir Drool Stain	Dog	7	Tim

Visit

Pet_Id	Visit Date
201	23-May-91
201	5-Apr-88
203	23-Nov-96
206	16-Oct-91
203	12-Oct-86
201	12-Mar-97
206	1-Sep-76
201	16-Mar-97

As you can see the Pet with Pet_Id of 201 has visited on four occasions, whereas Pet_Id has only visited twice.

By not trying to anticipate the maximum number of Visits (thus locking the system into a number that may need to be changed later) we have a more flexible design.

The same may be said of Phones and Vaccinations.

This type of relationship is called a one-to-many. In this case it's fairly clear from whence the name comes, each Pet may have many Visits.

Reading Relationship Verb Phrases:

Relationships show how one Entity affects another. We usually create a verb phrase for these to make the nature of the relationship clear.

Each <entity 1> {may be/must be} <relationship name> {one and only one/one or more} <entity 2>

For example:

Each Pet may be examined during zero, one, or more Visits.

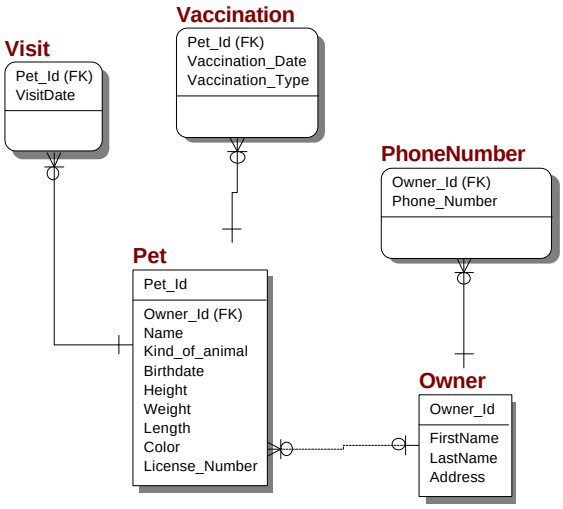
Each Pet may be immunized with zero, one, or more Vaccinations.

We still have one problem in the ERD: Age or Birthdate. There is no way to know which is stored in this column: the age of the Pet or the Birthday. Columns like these are real problems for database developers because they have to add special code to decrypt what is being stored. These things are often introduced by well-intentioned programmers coming from an environment where every byte of storage had to be accounted for and used efficiently. In the world of RDBMS, though, this counting-bits mentality can really cause some headaches. So either store the data in two separate columns of choose one or the other as the official columns and drop the other. In this case Birthdate is a better choice.

2nd Normal Form: Partial Dependencies

Here the objective is to move columns which do not logically depend upon the entity in question into a separate table. If we look at our ERD we find that the Owner's name and address is in the Pet table. Even stranger, Phone Numbers are being associated with the Pets and not the Owners. Also, if an Owner has two Pets we would be forced to double enter the Name and Address for the Owner. Bad, bad, bad!

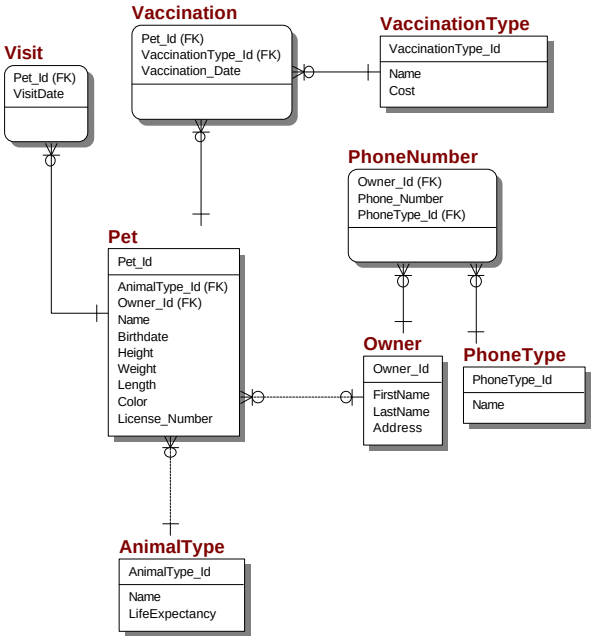
The solution is moving the Owner data into its own table, and of course, adding an Owner_Id.



3rd Normal Form: Transitive Dependencies

What happens if we delete a Pet? Let us say that we delete "Tibbles", Pet_Id of 203. There would no longer be any "Horses" in the database. What's more, there would be no way of telling that Horse was a valid "Kind of Animal". Likewise, if we delete a Vaccination for "Rabies" from the database how can we be sure that the Vaccination Type of Rabies would be anywhere in the database? The answer may be that we simply do not care, but if we do we have experienced a transitive dependency. A Transitive Dependency arises when one piece of data depends upon another for its inclusion in the database (it's getting late and I am aware that that made no sense, but I'll revise these notes later).

What we do to eradicate this is store the dependent data in a separate table, so for Vaccination we might do the following:



Some sample data for Vaccination Types and Phone Types should help where the words are failing me.

Vaccination Type

Vaccination Type_Id	Name
1	Rabies
2	Hoof-n-Mouth
3	Rocky Mountain Fever

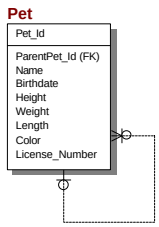
Phone Type

Phone Type_Id	Name
1	Home
2	Business
3	FAX
4	Pager
5	Modem

Making it cool

If I were making a really robust database with the complete history of Pets I might decide that the DB should stored the children, parents, siblings, etc. To track the parent of a litter of puppies all I need to do is store the parent Dog's Pet_Id in the Pet table. I'd call this column ParentPaet_Id because I

couldn't call it Pet_Id, that column name has already been taken by the Primary Key column.



What I am describing here is a self-referencing one-to-many relationship, and it looks like the model shown.

Oracle actually provides a really cool, though non-standard, mechanism for traversing these type of tables because it is so common and such a pain to do otherwise. In most RDBMS one would need to write a stored procedure, but Oracle has this construct. Let's say we want all of the descendents of "Snowflake", Pet_Id of 202. Well, we can write one SQL statement to get this (I know this doesn't belong here, but so what?):

```
SELECT Pet_Id, Name, Level
  FROM Pet
  START WITH Pet_Id = 201
  CONNECT BY ParentPet_Id = PRIOR Pet_Id;
```

"Level" is a pseudo column Oracle provides to tell you how many generations (in this case) deep you have gone. You would get back a result set beginning with Snowflake (Level of zero) and ending whenever there are no more children. If you only want a fixed number of descendents you may specify the Level to stop with in the WHERE clause. This same mechanism also works going the other way, in this case it would give you all of the ancestors of Snowflake.