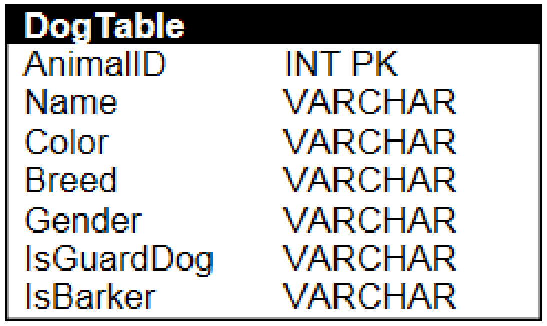
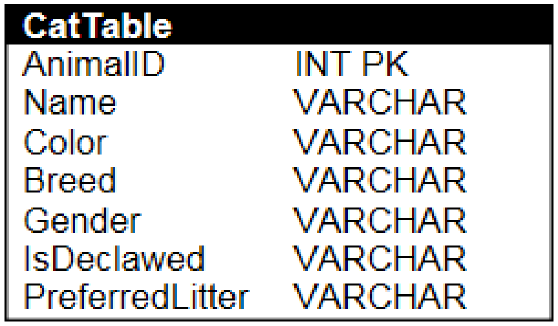
Modeling Object Oriented Data in a Relational Database

Often in business programming, we find ourselves storing data that lends itself to the concept of inheritance that we find in object oriented programming. Let's assume we have an application that works with cats and dogs. They are both animals and in our model they share many attributes. There are several options for how to model this data, each with their own advantages and draw backs.

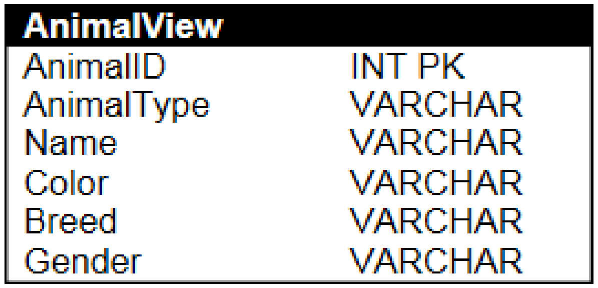
Option 1



DogTable	
AnimalID	INT PK
Name	VARCHAR
Color	VARCHAR
Breed	VARCHAR
Gender	VARCHAR
IsGuardDog	VARCHAR
IsBarker	VARCHAR



CatTable	
AnimalID	INT PK
Name	VARCHAR
Color	VARCHAR
Breed	VARCHAR
Gender	VARCHAR
IsDeclawed	VARCHAR
PreferredLitter	VARCHAR



AnimalView	
AnimalID	INT PK
AnimalType	VARCHAR
Name	VARCHAR
Color	VARCHAR
Breed	VARCHAR
Gender	VARCHAR

In this option we have a separate table for cats and dogs. We also have a view that contains all cats and dogs by union-ing the two tables together. To improve performance, the view may be materialized.

Option 2

AnimalTable	
AnimalID	INT PK
AnimalType	VARCHAR
Name	VARCHAR
Color	VARCHAR
Breed	VARCHAR
Gender	VARCHAR

DogTable	
AnimalID	INT PK
IsGuardDog	VARCHAR
IsBarker	VARCHAR

CatTable	
AnimalID	INT PK
IsDeclawed	VARCHAR
PreferredLitter	VARCHAR

Here our main table is the animal table that contains all the shared attributes. A separate dog and cat table are maintained with only the attributes that are unique to each type.

Option 3

AnimalTable	
AnimalID	INT PK
AnimalType	VARCHAR
Name	VARCHAR
Color	VARCHAR
Breed	VARCHAR
Gender	VARCHAR
IsGuardDog	VARCHAR
IsBarker	VARCHAR
IsDeclawed	VARCHAR
PreferredLitter	VARCHAR

DogView	
AnimalID	INT PK
Name	VARCHAR
Color	VARCHAR
Breed	VARCHAR
Gender	VARCHAR
IsGuardDog	VARCHAR
IsBarker	VARCHAR

CatView	
AnimalID	INT PK
Name	VARCHAR
Color	VARCHAR
Breed	VARCHAR
Gender	VARCHAR
IsDeclawed	VARCHAR
PreferredLitter	VARCHAR

In this option, we have only one table with an AnimalType column. Dogs and cats are separated into their own views using that column. Similar to option 1, materializing the views can improve performance.

Comparison

Option	**Advantages**	**Disadvantages**	**How to add new animal type**
1	Easy to edit a cat or a dog.	An application dealing with generic Animals needs to know which table to query. May suffer from performance problems or complexity in dealing with materialized view refreshes. If you want to edit and Animal, you need to figure out which table to edit (you may create an updatable view to minimize this downside). Generic animals do not exist, each must be a cat or a dog.	New views and new tables
2	One point access for Animal attributes; There is no redundancy of columns, each column is found in only one table. Animals that are neither dogs nor cats can be entered in the animal table.	An application dealing with generic Animals needs to know which table to query. Any read of dog or cat is going to require a join. Any write to a dog or cat is going to require writing to two tables and keeping them in sync. This requires transactions.	Add new tables
3	One point access for all animals and all their individual attributes. One query insert and delete. Query only dogs, only a change to the where clause is required. Autonumber works well with this option, but please use sequences instead of autonumber anyway!	Confusion about which attributes are for which animals should be solved with a prefix. Real databases could eventually run out of columns if you have many different types of animals.	New attribute columns and type columns. For example if you needed to add a turtle, and differentiate mammals and reptiles, you would add a reptile type column and a mammal type column.