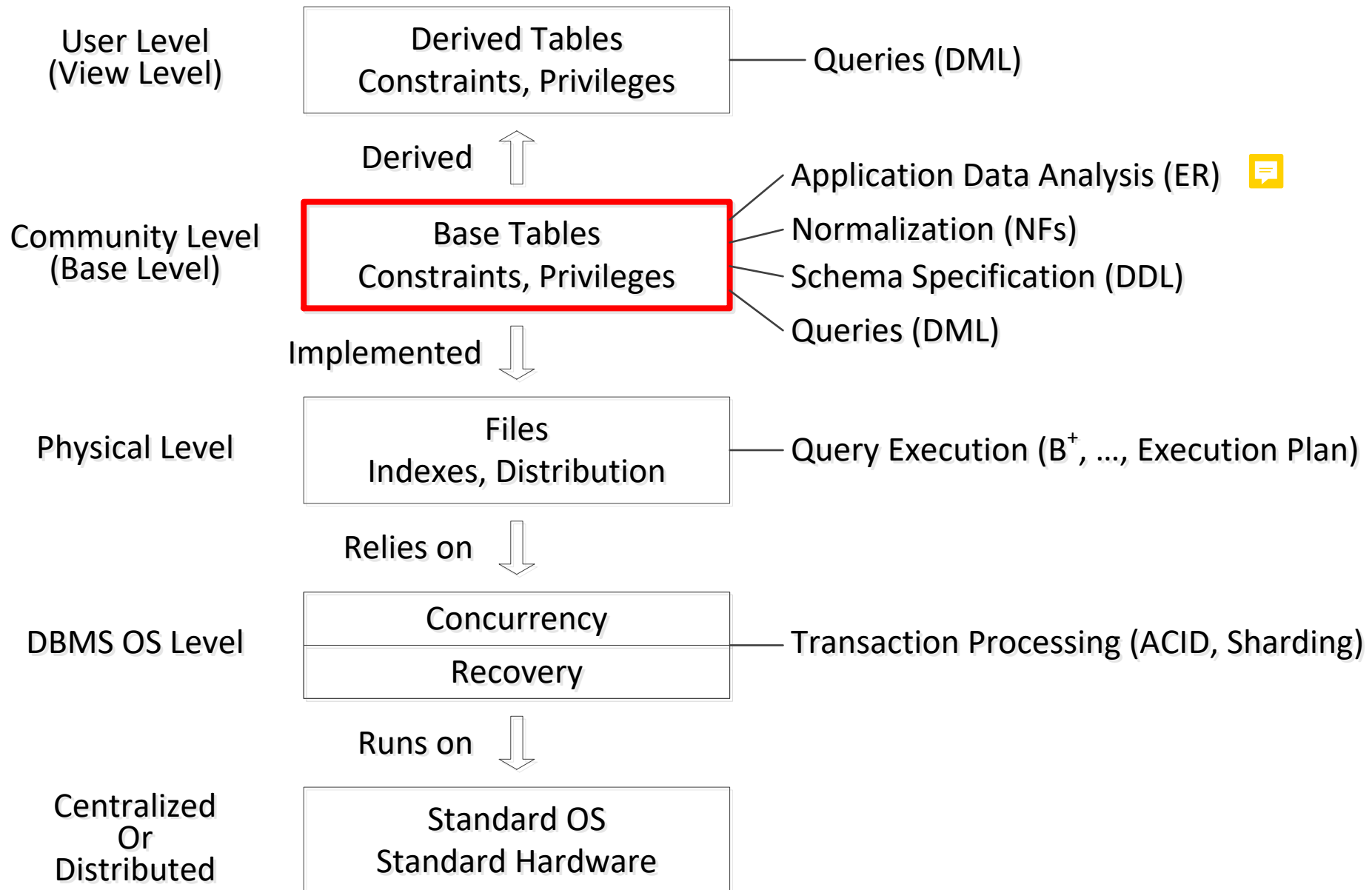


***Unit 3***  
***The Relational Model***  
***And From ER Diagrams to Relational Databases***

# Relational Implementation in Context



# ***Introduction***

# ***MultiSets, Relations, and Tables***

- The basic “data type”, or a “variable” of a relational database, is a ***relation***
- These are ***multisets*** because duplicates are allowed
- In SQL such a variable is called a ***table***
- We may use the term table for a relation in this unit too

# ***Relations***

# Sets

- We will **not** use axiomatic set theory
- A **set** is a “bag” of elements, some/all of which could be sets themselves and a binary relationship “is element of” denoted by  $\in$ , such as  $2 \in \{2, 5, 3, 7\}$ ,  $\{2,8\} \in \{2, \{2, 8\}, 5, 3, 7\}$ ,
- You **cannot** specify
  - How many times an element appears in a set (if you could, this would be a **multiset**)
  - In which position an element appears (if you could, this would be a **sequence**)
- Therefore, as sets:  $\{2, 5, 3, 7\} = \{2, 7, 5, 3, 5, 3, 3\}$
- Note: in many places you will read: “an element can appear in a set at most once”

**This is not quite right.** And it is important not to assume this, as we saw in our discussion of SQL.

# MultiSets

- Two multisets  $A$  and  $B$  are equal iff (that is, if and only if) they have the same elements with the same number of duplicates of each element
- In other words, for every  $x$ :  $x$  is an element of  $A$  iff  $x$  is an element of  $B$  with the same cardinality.
- Note that position doesn't matter

# Relation

- Consider a table, with a fixed number of columns where the elements of each column are drawn from some specific domain
- The columns are labeled and the labels are distinct
- We will consider such a table to be **a multiset of rows** (another word for “row” in our context: **tuple**)
- An example of a table S of two columns A and B

S	A	B
	a	2
	a	2
	b	3
	c	4
	d	3



- A **relation** is such a table
- We will also write  $S(A,B)$  for table S with columns A and B



# ***Relational Schema***

- What we saw was an ***instance*** (current value for a relation with the defined columns and domains)
- To specify this relation in general (not the specific instance) we need to talk about a ***relational schema***
- A relational schema defines a set of relations
- In databases everything is finite, so a relational schema defines a finite set of finite relations

# ***Relational Schema***

- Here is an informal, but complete, description what a relational schema of one relation is
- We want to define a structure for some table
  1. We give it a name (we had *S*)
  2. We chose the number of columns (we had 2) and give them distinct names (we had *A* and *B*)
  3. We decide on the domains of elements in the columns (we had letters for *A* and integers for *B*)
  4. We decide on constraints, if any, on the permitted values (for example, we can assume as it was true for our example that any two rows that are equal on *A* must be equal on *B*)

This part must not be omitted if such constraints exist!

# Relational Schema

- Let's verify that the following hold in our example if it is supposed to satisfy
  - $A$ : all lower case letters in English
  - $B$ : all positive integers less than 100
  - $S(A,B)$  satisfies the condition that any two tuples that are equal on  $A$  must also be equal on  $B$
- Our example was an instance of this relational schema

S	A	B
	A	2
	a	2
	b	3
	c	4
	d	3

# Tables vs. Relations

- Since relational tables are *multisets* of tuples, *the following two tables are unequal* (though they are equal as mathematical “relations”)



S	A	B
	a	2
	a	56
	b	2

S	A	B
	a	56
	a	2
	b	2
	a	56
	a	2
	a	56

# Tables

- To specify tables, it is enough to do what we have done above
- As long as we understand what are the domains for the columns, the following are formally fully specified relations (more formally relational schemas)
  - (schema) P(Name, SSN, DOB, Grade) with some (not specified, but we should have done it) domains for attributes
  - (schema) Q(Grade, Salary) with some (not specified, but we should have done it) domains for attributes

P	Name	SSN	DOB	Grade
	A	121	2367	2
	B	101	3498	4
	C	106	2987	2

Q	Grade	Salary
	1	90
	2	80
	3	70
	4	70

# ***Keys***

# Keys

- We will specify generally, as suitable for the schema:
  - **Primary keys**
  - **Keys** (beyond primary)
  - **Foreign keys** and what they reference (we will see soon what this means)
- The above are most important **structurally**
- Later, especially when we talk about SQL DDL, we will specify additional properties and constraints
  - For example, the height of a person has to be positive, if it is known
- Some of the constraints may involve more than one relation

# Keys (and Superkeys)

- Consider relation (schema) Person(FN, LN, Grade, YOB)
- Instance:

Person	FN	LN	Grade	YOB
	John	Smith	8	1976
	Lakshmi	Smith	9	1981
	John	Smith	8	1976
	John	Yao	9	1992

- We are **told** that any two tuples that are equal on both FN and LN are (completely) equal
  - We have some tuples appearing multiple times: this is just for clarifying that this is permitted in the definition, we do not discuss here why we would have the same tuple more than once (we will talk about this later)
- This is a property of **every possible instance** of Person in our application: we are told that
- Then (FN, LN) is a **superkey** (is a superset of a key) of Person, and in fact a **key**, because neither FN nor LN by themselves are sufficient (we are told that too)



# Keys (and Superkeys)

- Consider relation (schema) Pay(Grade, Salary)

- Example:

Pay	Grade	Salary
	8	128
	9	139
	7	147

- We are told that for any instance of Pay, any two tuples that are equal on Grade are (completely) equal
  - Of course, if each Grade appears in only one tuple, this is automatically true
- Then, similarly to before, Grade is a key
- What about Salary, is this a key also?
- No, because we **are not told (that is, we are not guaranteed)** that any two tuples that are equal on Salary are equal on Grade in every possible instance of Pay

# ***Keys (and Superkeys)***

- A set of columns in a relation is a **superkey** (superset of a key:  $\supseteq$ ) if and only any two tuples that are equal on the elements of these columns are (completely equal)
- A minimal superkey, is a **key**
- There may be more than one key
- Exactly one key is chosen as **primary key** (there is no formal way to decide which one)
- Other keys are just keys and sometimes they are called **candidate keys** (as they are candidates for the primary key, though not chosen)

# Keys (and Superkeys)

- To summarize
- **Superkey**: a set of columns whose values determine the values of all the columns in a row for every instance of the relation
- **Key**: a set of columns whose values determine the values of all the columns in a row for every instance of the relation, **but any proper subset of that set of columns does not do that**
- **Primary Key**: a chosen key. **Important: no attribute of a primary key can be NULL (the value of every attribute must be always known);** why will be discussed later

# ***Keys (and Superkeys)***

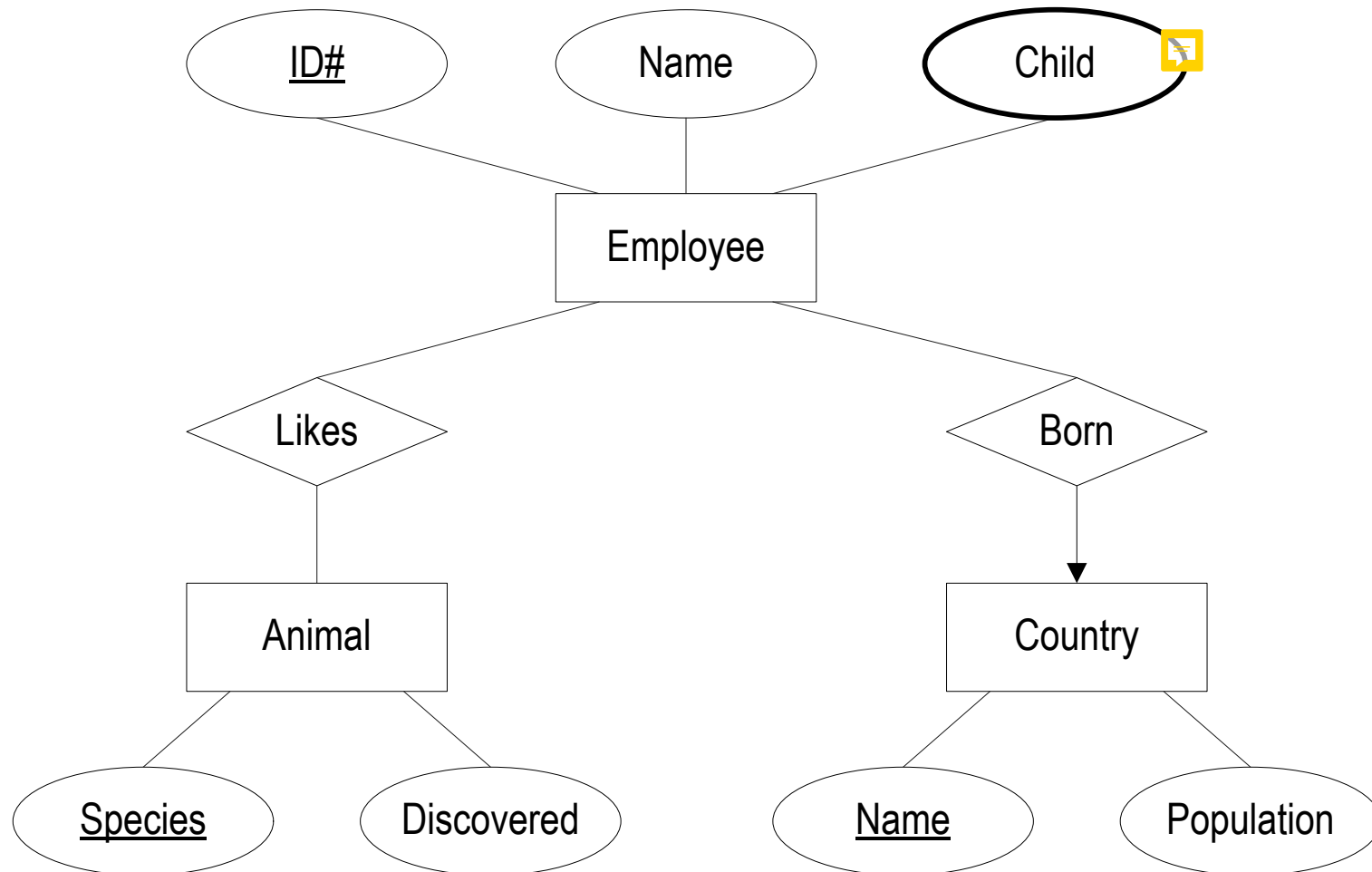
- We will underline the attributes of the chosen primary key
- Returning to Unit 02 and example of City: City(Longitude, Latitude, Country, State, Name, Size)
- We can have
  - City(Longitude, Latitude, Country, State, Name, Size)
  - This implies that (Longitude, Latitude) form the primary key
  - We also have a candidate key: (Country, State, Name)
- We can have
  - City(Longitude, Latitude, Country, State, Name, Size)
  - If the value of State is never NULL This implies that (Country, State, Name) form the primary key
  - We also have a candidate key: (Longitude, Latitude)
- The situation will be described more precisely concerning candidate keys when we talk about SQL DDL

# ***From ER Diagram to Relational Database Key Concepts With Examples***

# ***From ER Diagrams to Relational Databases***

- We are now ready to convert ER diagrams into relational databases
- ***Generally, but not always***
  - ***An entity set is converted into a table***
  - ***A relationship is converted into a table***
- We will first go **very carefully** over a simple example
- Then, we will go through our large example, studied previously
- Then, we look at some additional points of interest
- Finally, we summarize the process
- We will use Visio 2010 in the slides as it is visually attractive and easy to understand
- We will also look at the SQL Power Architect tool

# Small ER Diagram



## ***More About the Example***

- The given ER diagram is clear, other than
  - Discovered, which is the continent in which a particular species was first discovered
- Each child is a “dependent” of only one employee in our database
  - If both parents are employees, the child is “assigned” to one of them (say the one whose ID# is smaller)
- We are given additional information about the application
  - ***Values of attributes in a primary key must not be missing*** (this is a general rule as we know, not only for this example)
  - Other than attributes in a primary key, other attributes, unless stated otherwise, may be missing
  - The value of Name is known (not missing) for every Employee
- To build up our intuition, let's look at some specific instance of our application



## ***Small Example***

### ■ 5 Employees:

- 1 is Alice has Erica and Frank, born in US, likes Horse and Cat
- 2 is Bob has Bob and Frank, born in US, likes Cat
- 4 is Carol
- 5 is David, born in IN
- 6 is Bob, born in CN, likes Yak

### ■ 4 Countries

- US
- IN has 1347
- CN has 1412
- RU

### ■ 5 Animals

- Horse in Asia
- Wolf in Asia
- Cat in Africa
- Yak in Asia
- Zebra in Africa

# Country

- There are four countries, listing for them: Cname, Population (the latter only when known):
  - US
  - IN, 1347
  - CN, 1412
  - RU
- We create a table for Country “in the most obvious way,” by **creating a column for each attribute** (underlining the attributes of the primary key) and that works:

Country	<u>Cname</u>	Population
	US	
	IN	1347
	CN	1412
	RU	

- Note that some “slots” are NULL, indicated by emptiness

# ***Animal***

- There are five animals, listing for them: Species, Discovered (note, that even though not required, Discovered happens to be known for every Species):
  - Horse, Asia
  - Wolf, Asia
  - Cat, Africa
  - Yak, Asia
  - Zebra, Africa
- We create a table for Animal as before, and that works:

Animal	<u>Species</u>	Discovered
	Horse	Asia
	Wolf	Asia
	Cat	Africa
	Yak	Asia
	Zebra	Africa

# Employee

- There are five employees, listing for them: ID#, Name, (name of) Child (note there may be any number of Child values for an Employee, zero or more):
  - 1, Alice, Erica, Frank
  - 2, Bob, Bob, Frank
  - 4, Carol
  - 5, David
  - 6, Bob, Frank
- We create a table for Employee in the most obvious way, and that **does not** work:

Employee	<u>ID#</u>	Name	Child	Child
	1	Alice	Erica	Frank
	2	Bob	Bob	Frank
	4	Carol		
	5	David		
	6	Bob	Frank	

# Employee

- Child is a multivalued attribute so, the number of columns labeled “Child” is unbounded
- A table **must have** a fixed number of columns
  - It must be an instance in/of a relational schema
- If we are ready to store up to 25 children for an employee and create a table with 25 columns for children, perhaps tomorrow we get an employee with 26 children, who will not “fit”
- We **replace our attempted single table** for Employee **by two tables**
  - One for all the attributes of Employee other than the multivalued one (Child)
  - One for pairs of the form (primary key of Employee, Child)
- Note that both tables have a fixed number of columns, no matter how many children an employee has, we have correct structures

# ***Employee and Child***

- Replace (incorrect)

Employee	<u>ID#</u>	Name	Child	Child
	1	Alice	Erica	Frank
	2	Bob	Bob	Frank
	4	Carol		
	5	David		
	6	Bob	Frank	

## By (correct)

Employee	<u>ID#</u>	Name
	1	Alice
	2	Bob
	4	Carol
	5	David
	6	Bob

Child	<u>ID#</u>	<u>Child</u>
	1	Erica
	1	Frank
	2	Bob
	2	Frank
	6	Frank

# *Employee and Child With Better Column Names*

- Replace (incorrect)

Employee	<u>ID#</u>	Name	Child	Child
	1	Alice	Erica	Frank
	2	Bob	Bob	Frank
	4	Carol		
	5	David		
	6	Bob	Frank	

By (correct)

Employee	<u>ID#</u>	Name
	1	Alice
	2	Bob
	4	Carol
	5	David
	6	Bob

Child	<u>Parent</u>	<u>Child</u>
	1	Erica
	1	Frank
	2	Bob
	2	Frank
	6	Frank

## ***Employee and Child***

- The primary key of the table Employee is ID#
  - The primary key of the table Child is the pair: Parent,Child
  - One attribute is not sufficient to get a primary key for Child
  - By “correlating” ID# with Parent, we can answer queries such as: What is the Name of the Child’s Parent
- 
- It is clear from the example how to handle any number of simple multivalued attributes an entity has, basically
    - Create a “main” table with all the attributes other than multivalued ones  
Its primary key is the original primary key of the entity set
    - Create a table for each multivalued attribute consisting of the primary key for the main table and of that multivalued attribute  
Its primary key is the primary key of the entity combined with the multivalued attribute



# Foreign Key

- Let us return to our example
- Note that any value of ID# that appears in Child must also appear in Employee
  - Because a child must be a dependent of an existing employee
- This is an instance of a **foreign key**
- ID# in Child is a **foreign key referencing** Employee
  - This means that ID# appearing in Child must appear in some row “under” the columns (here only one) of primary key in Employee (slightly oversimplifying in this Unit: not necessarily primary key)
  - Note that ID# is not a key of Child (but is a part of a key), **so a foreign key in a table does not have to be a key of that table**

Employee	<u>ID#</u>	Name	Child	<u>ID#</u>	<u>Child</u>
	1	Alice		1	Erica
	2	Bob		1	Frank
	4	Carol		2	Bob
	5	David		2	Frank
	6	Bob		6	Frank

# *Foreign Key $\equiv$ A Binary Many-To-One Relationship Between Tables (Partial Function)*

Employee	<u>ID#</u>	Name	Child	<u>ID#</u>	<u>Child</u>
	1	Alice		1	Erica
	2	Bob		1	Frank
	4	Carol		2	Bob
	5	David		2	Frank
	6	Bob		6	Frank

- Note:
  - Every row of Child has a single value of a primary key of Employee, so every row of Child “maps” to a single row of Employee
  - Every row of Employee has zero or more rows of Child mapped into itIn other words, no constraint
- Here the function is **total**: defined everywhere

# *Foreign Key $\equiv$ A Binary Many-To-One Relationship Between Tables*



Employee	<u>ID#</u>	Name		Child	<u>Parent</u>	<u>Child</u>
	1	Alice			1	Erica
	2	Bob	←		1	Frank
	4	Carol	←		2	Bob
	5	David	←		2	Frank
	6	Bob	←		6	Frank

- Another option
- Note column names do not have to be the same for the mapping to take place
- But you need to specify which column is foreign key referring to what
- Here: Parent in Child is a foreign key referencing Employee
- In SQL DDL the syntax is a little different

# ***Born***

- Born needs to specify which employees were born in which countries (for whom this information is known)
- We can list what is the current state
  - Employee identified by 1 was born in country identified by US
  - Employee identified by 2 was born in country identified by IN
  - Employee identified by 5 was born in country identified by IN
  - Employee identified by 6 was born in country identified by CN

# Born

- Born needs to specify who was born where 
- We have tables for
  - Employee 
  - Country
- We know that **each employee was born in at most one country** (actually was born in exactly one country but we may not know what it is)
- We have a **binary many-to-one relationship** between Employee and Country, but no implementation yet

Employee	<u>ID#</u>	Name	Country	<u>CName</u>	Population
	1	Alice	→	US	
	2	Bob	→	IN	1347
	4	Carol	→	CN	1412
	5	David	→	RU	
	6	Bob	→		

# Implementation for Born

- Augment Employee so instead of

Employee	<u>ID#</u>	Name
	1	Alice
	2	Bob
	4	Carol
	5	David
	6	Bob

Country	<u>CName</u>	Population
	US	
	IN	1347
	CN	1412
	RU	

we have

Employee	<u>ID#</u>	Name	CName
	1	Alice	US
	2	Bob	IN
	4	Carol	
	5	David	IN
	6	Bob	CN

Country	<u>CName</u>	Population
	US	
	IN	1347
	CN	1412
	RU	

# Implementation for Born

- Augment Employee so instead of

Employee	<u>ID#</u>	Name
	1	Alice
	2	Bob
	4	Carol
	5	David
	6	Bob

Country	<u>CName</u>	Population
	US	
	IN	1347
	CN	1412
	RU	

we have two tables and a binary many-to-one mapping

Employee	<u>ID#</u>	Name	CName		Country	<u>CName</u>	Population
	1	Alice	US	→		US	
	2	Bob	IN	→		IN	1347
	4	Carol		→		CN	1412
	5	David	IN	→		RU	
	6	Bob	CN	→			

# Foreign Key Constraint Implementing Born

- We have again a foreign key constraint
- Any value of CName in Employee must also appear in Country as a primary key in some row
- CName in Employee is a foreign key referencing Country
- Note that CName in Employee is not even a part of its primary key

Employee	<u>ID#</u>	Name	CName		Country	<u>CName</u>	Population
	1	Alice	US	→		US	
	2	Bob	IN	→		IN	1347
	4	Carol		→		CN	1412
	5	David	IN	→		RU	
	6	Bob	CN	→			



# Foreign Key Constraint Implementing Born

- Perhaps better (and frequently done in practice) use a different name for foreign keys
- Any value of CBirth in Employee must also appear in Country as a primary key in some row
- CBirth in Employee is a foreign key referencing Country
- We will not talk about such, possibly convenient, renaming

Employee	<u>ID#</u>	Name	CBirth		Country	<u>CName</u>	Population
	1	Alice	US	→		US	
	2	Bob	IN	→		IN	1347
	4	Carol		→		CN	1412
	5	David	IN	→		RU	
	6	Bob	CN	→			

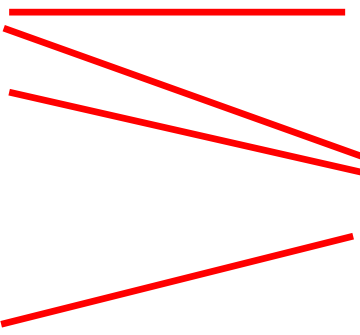
# *Likes*

- Likes needs to specify which employees like which animals
- We can list what is the current state:
  - Employee identified by 1 likes animal identified by Horse
  - Employee identified by 1 likes animal identified by Cat
  - Employee identified by 2 likes animal identified by Cat
  - Employee identified by 6 likes animal identified by Yak

# Likes

- We can describe Likes by drawing lines between the two tables
- We need to “store” this set of red lines
- ***Likes is a many-to-many relationship***
  - ***It is not a many-to-one relationship and therefore it is not a partial function***

Employee	<u>ID#</u>	Name	Animal	<u>Species</u>	Discovered
	1	Alice		Horse	Asia
	2	Bob		Wolf	Asia
	4	Carol		Cat	Africa
	5	David		Yak	Asia
	6	Bob		Zebra	Africa



## *Likes (impossible implementation)*

- Cannot store with Employee (there is no limit on the number of animals an employee likes)

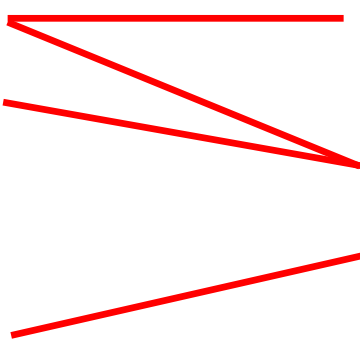
Employee	<u>ID#</u>	Name	Animal	<u>Species</u>	Discovered
	1	Alice		Horse	Asia
	2	Bob		Wolf	Asia
	4	Carol		Cat	Africa
	5	David		Yak	Asia
	6	Bob		Zebra	Africa

Employee	<u>ID#</u>	Name	Species	Species
	1	Alice	Horse	Cat
	2	Bob	Cat	
	4	Carol		
	5	David		
	6	Bob	Yak	

## *Likes (impossible implementation)*

- Cannot store with Animal (there is no limit on the number of employees who like an animal)

Employee	<u>ID#</u>	Name	Animal	<u>Species</u>	Discovered
	1	Alice		Horse	Asia
	2	Bob		Wolf	Asia
	4	Carol		Cat	Africa
	5	David		Yak	Asia
	6	Bob		Zebra	Africa



Animal	<u>Species</u>	Discovered	ID#	ID#
	Horse	Asia	1	
	Wolf	Asia		
	Cat	Africa	1	2
	Yak	Asia	6	
	Zebra	Africa		

# Likes

- Each red line is an **edge** defined by its vertices
  - We create a table storing the red lines; that is, its vertices
  - We can do this using the **primary keys** of the entities
- We do not need other attributes such as Name or Discovered
- The table for Likes contains tuples:
    - 1 likes Horse
    - 1 likes Cat
    - 2 likes Cat
    - 6 likes Yak

Likes	<u>ID#</u>	<u>Species</u>
	1	Horse
	1	Cat
	2	Cat
	6	Yak

# Likes

- Note that ***there are foreign key constraints***
  - ID# appearing in Likes is a foreign key referencing Employee
  - Species appearing in Likes is a foreign key referencing Animal
- And two many-to-one mappings are induced
- Note: ***a binary many-to-many relationship was replaced by a new table and two many-to one relationships***

Employee	<u>ID#</u>	Name	Likes	<u>ID#</u>	<u>Species</u>	Animal	<u>Species</u>	Discovered
	1	Alice		1	Horse		Horse	Asia
	2	Bob		1	Cat		Wolf	Asia
	4	Carol		2	Cat		Cat	Africa
	5	David		6	Yak		Yak	Asia
	6	Bob					Zebra	Africa

# Relational Implementation for the Example

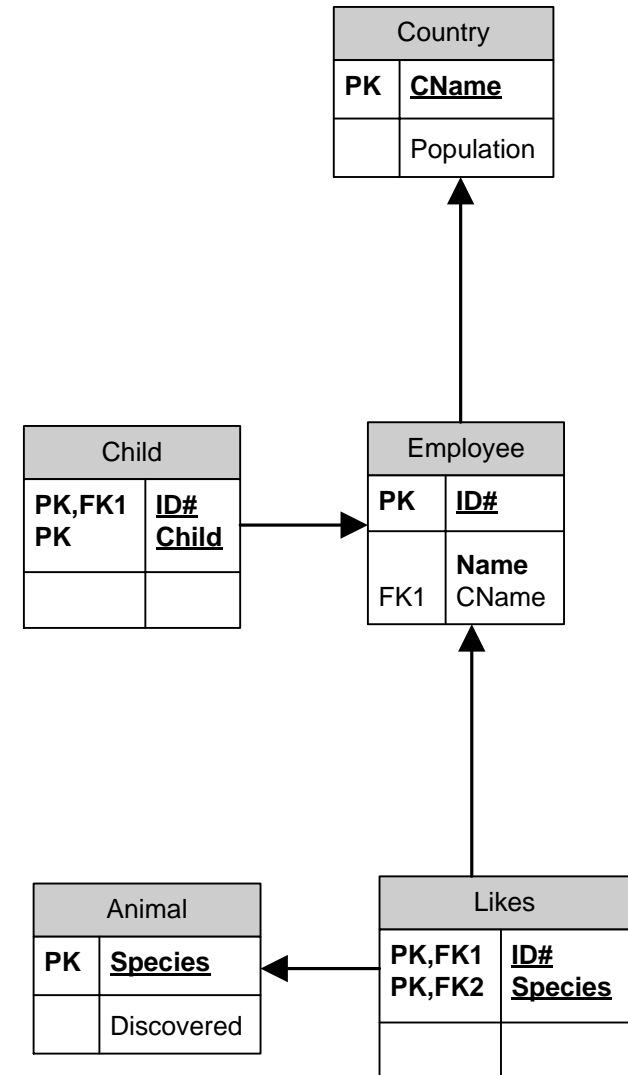
Country	<u>CName</u>	Population
	US	
	IN	1347
	CN	1412
	RU	

Child	<u>ID#</u>	<u>Child</u>
	1	Erica
	1	Frank
	2	Bob
	2	Frank
	6	Frank

Employee	<u>ID#</u>	Name	CName
	1	Alice	US
	2	Bob	IN
	4	Carol	
	5	David	IN
	6	Bob	CN

Animal	<u>Species</u>	Discovered
	Horse	Asia
	Wolf	Asia
	Cat	Africa
	Yak	Asia
	Zebra	Africa

Likes	<u>ID#</u>	<u>Species</u>
	1	Horse
	1	Cat
	2	Cat
	6	Yak





# ***Cardinality Constraints***

- The statement that a relationship is many-to-one as opposed to being a “standard” many-to-many relationship is really a cardinality constraint
- We will look at relationship Likes between Person and Country and four cases of cardinality constraints on how many Countries a Person may like
  - No constraint
  - At least one
  - At most one
  - Exactly one
- For the first two, Likes is many-to-many
- For the last two, Likes is many-to-one
- Intuitively, Likes is many to one if for every Person, when you see which Countries this Person Likes, you get 0 or 1
- If you always get 1, this is a total function, otherwise this is a partial function

# Specifying These Constraints (Revisited From Unit 2)



Every Person likes 0 or more Countries



Every Person likes 1 or more Countries



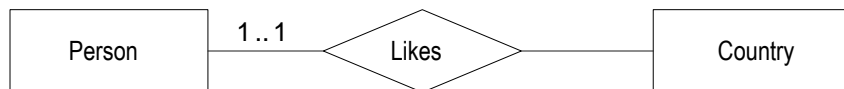
Every Person likes 0 or 1 Countries



Every Person likes 1 Country

# Crow's Feet: Improved Arrow Notation

- Note: different sides of the relationship are labeled in the two notations!**



# ***Crow's Feet***

- In general, cardinalities of both sides of the relationship may need to be specified
- We did only one, because it is sufficient to understand the notation
- We now return to the relational implementation of our example
- Visio can use the Crow's Feet notation
- SQL Power Architect uses the Crow's Feet notation
- Digress to:  
<http://www2.cs.uregina.ca/~bernatja/crowsfoot.html>

# Relational Implementation for the Example



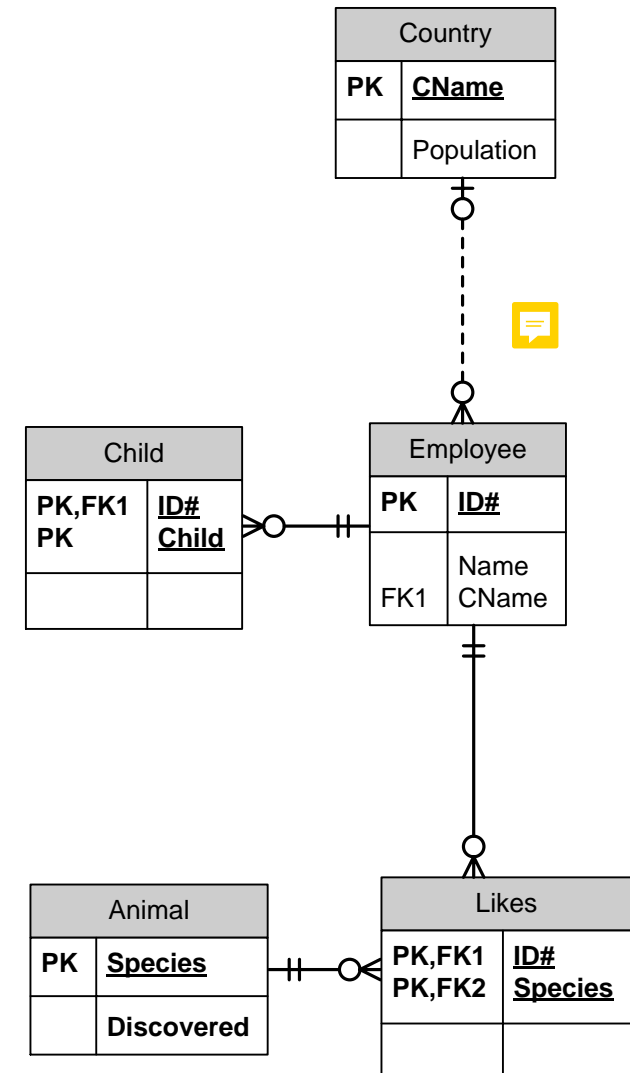
Country	<u>CName</u>	Population
	US	
	IN	1347
	CN	1412
	RU	

Child	<u>ID#</u>	<u>Child</u>
	1	Erica
	1	Frank
	2	Bob
	2	Frank
	6	Frank

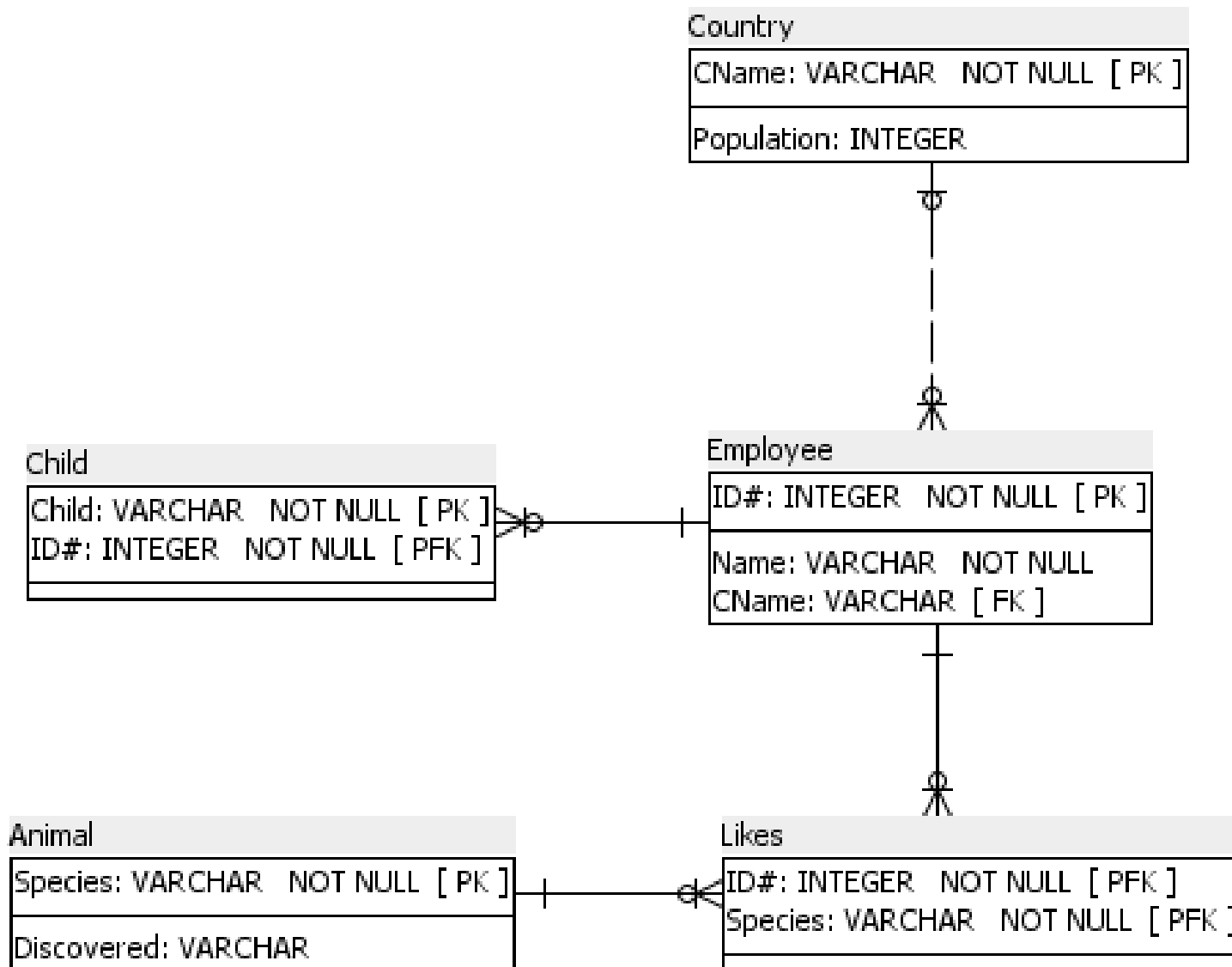
Employee	<u>ID#</u>	Name	CName
	1	Alice	US
	2	Bob	IN
	4	Carol	
	5	David	IN
	6	Bob	CN

Animal	<u>Species</u>	Discovered
	Horse	Asia
	Wolf	Asia
	Cat	Africa
	Yak	Asia
	Zebra	Africa

Likes	<u>ID#</u>	<u>Species</u>
	1	Horse
	1	Cat
	2	Cat
	6	Yak

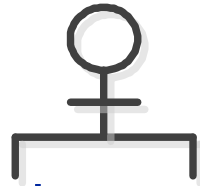


# Relational Implementation Using SQL Power Architect

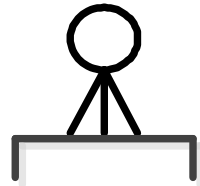


# *End of Lines In Crow's Feet Notation*

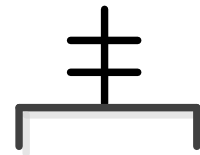
- 0..1: for each of the other entity, zero or 1 of this one



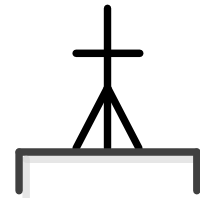
- \* 0..\*



- 1..1



- 1..\*: for each of the other entity, 1 or more of this one.



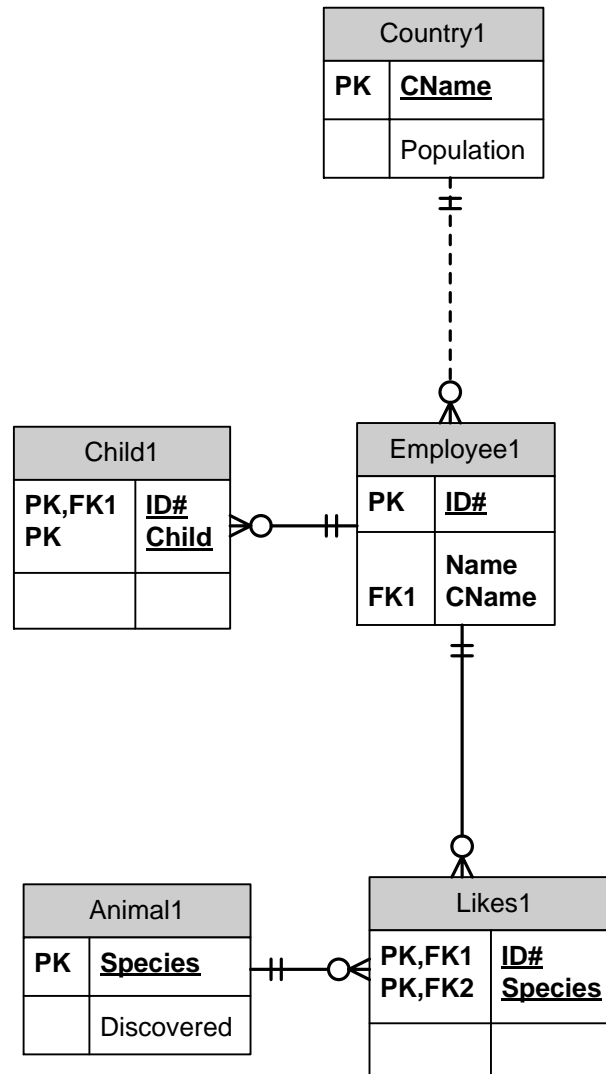
## ***Pattern of Lines***

- In the next slide, we see a slight modification of our example in which all lines have the same pair of endings
  - We required that for each Employee the Country of Birth is known
  - Nevertheless, as Cname is not part of the primary key of Country, the line is dashed
- 
- For technical reasons, the tables have slightly different names, but this has nothing to do with our point



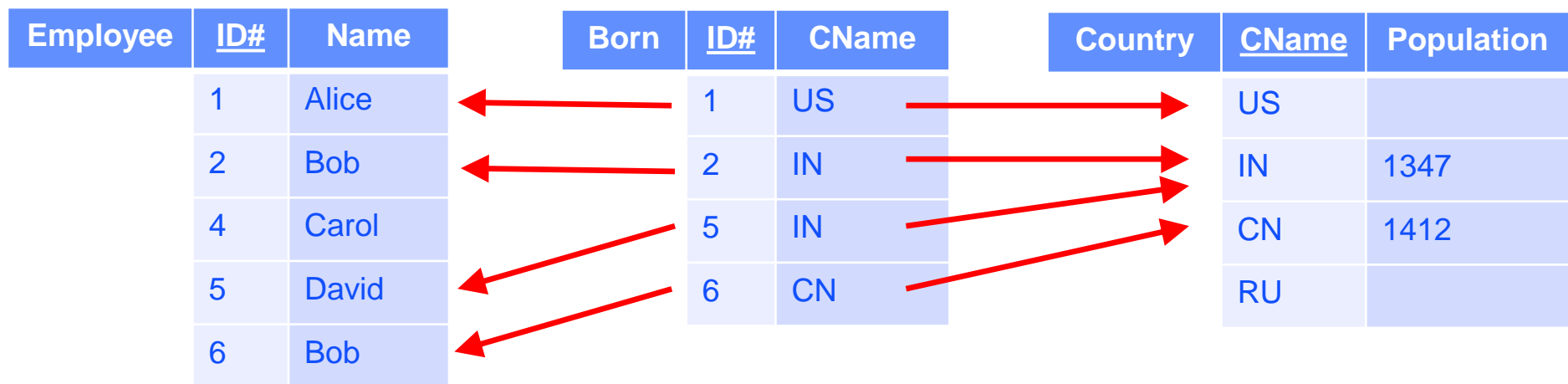
# Example

- Assume: Every employee has exactly one Country (that is we know the country of birth)



## Alternative Implementation for Born

- We **need** an “in-between” table for Likes because it is **many-to-many**
- We **do not need** an “in-between” table for Born because it is **many-to-one**
- But we can implement Born using such an “in-between” table
  - Note that CName is not part of the primary key of Born and this because Born is many-to-one



# Alternative Implementation for the Example

Country	<u>CName</u>	Population
	US	
	IN	1347
	CN	1412
	RU	

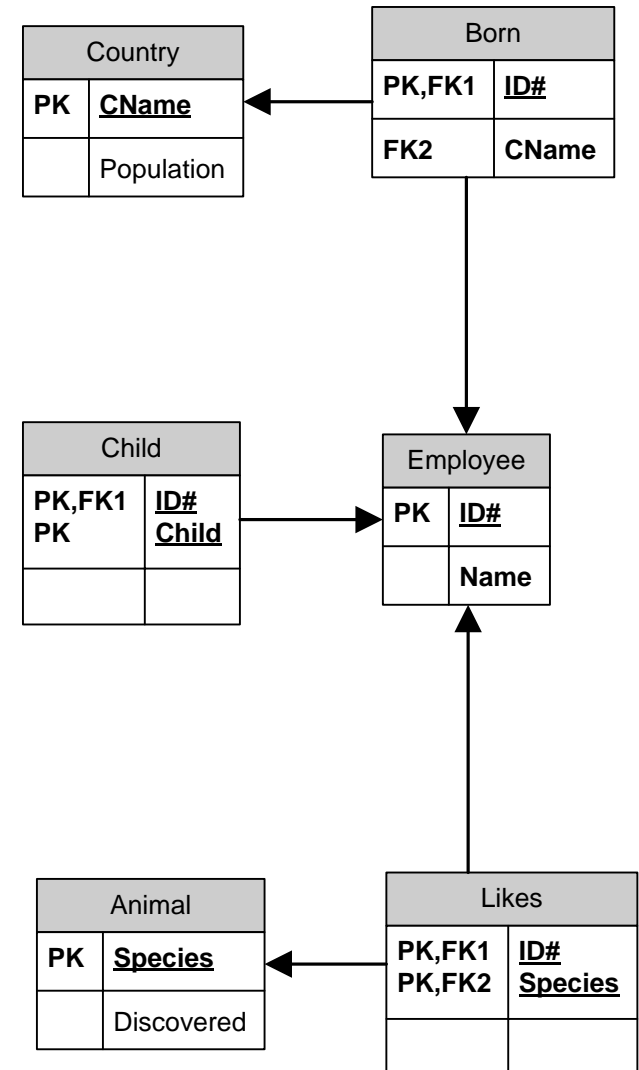
Born	<u>ID#</u>	CName
	1	US
	2	IN
	5	IN
	6	CN

Child	<u>ID#</u>	<u>Child</u>
	1	Erica
	1	Frank
	2	Bob
	2	Frank
	6	Frank

Employee	<u>ID#</u>	Name
	1	Alice
	2	Bob
	4	Carol
	5	David
	6	Bob

Animal	<u>Species</u>	Discovered
	Horse	Asia
	Wolf	Asia
	Cat	Africa
	Yak	Asia
	Zebra	Africa

Likes	<u>ID#</u>	<u>Species</u>
	1	Horse
	1	Cat
	2	Cat
	6	Yak



# Alternative Implementation for the Example

Country	<u>Cname</u>	Population
	US	
	IN	1347
	CN	1412
	RU	

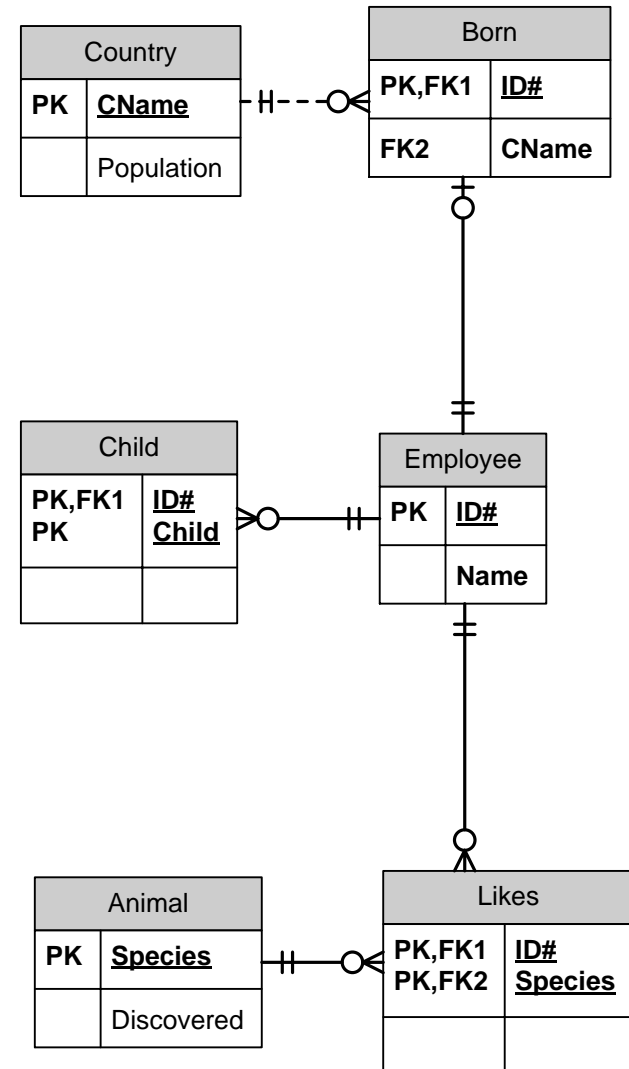
Born	<u>ID#</u>	CName
	1	US
	2	IN
	5	IN
	6	CN

Child	<u>ID#</u>	<u>Child</u>
	1	Erica
	1	Frank
	2	Bob
	2	Frank
	6	Frank

Employee	<u>ID#</u>	Name
	1	Alice
	2	Bob
	4	Carol
	5	David
	6	Bob

Animal	<u>Species</u>	Discovered
	Horse	Asia
	Wolf	Asia
	Cat	Africa
	Yak	Asia
	Zebra	Africa

Likes	<u>ID#</u>	<u>Species</u>
	1	Horse
	1	Cat
	2	Cat
	6	Yak



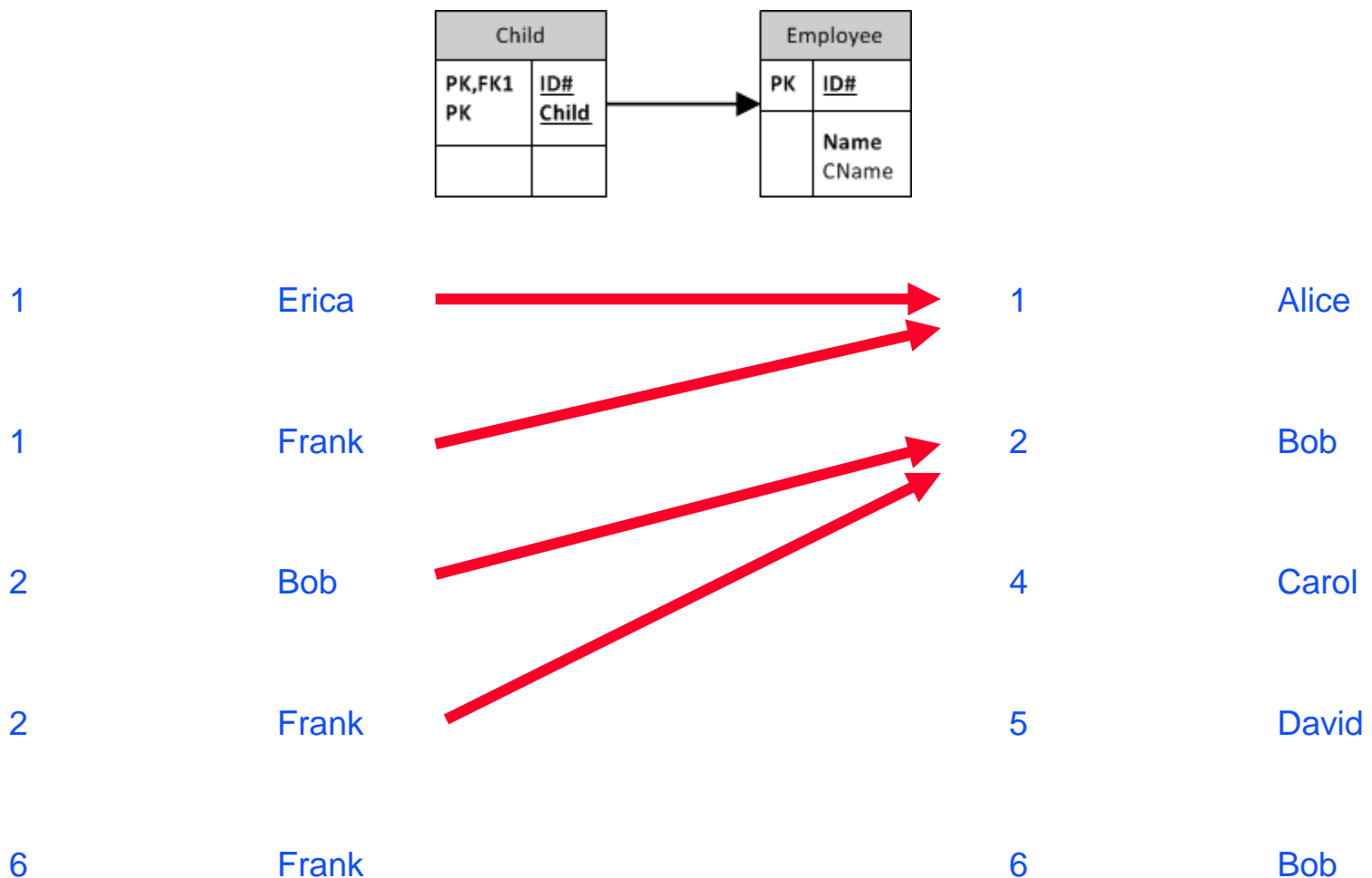
# To Remember!

- Structurally, a relational database consists of
  1. A **set of tables with identifiers (primary keys)**
  2. A **set of many-to-one binary relationships** between them, induced by foreign key constraints

In other words; **a set of functions (in general partial), each from a table into a table**
  
- When designing a relational database, you **should** specify both (or you will produce a bad specification)
  - Technically, tables are enough, but this a very bad practice as you do not specify the relationships between tables

# Many-To-One Mapping From Child to Employee (to Reiterate)

- Partial function from a set of rows into a set of rows



# Very Bad Relational Implementation

- Tables are listed with attributes, specifying only which are in the primary key
- Foreign key constraints are not specified
  - So the DB system does not know what to enforce

Country	
PK	<u>CName</u>
	Population

Child	
PK	<u>ID#</u>
PK	<u>Child</u>

Employee	
PK	<u>ID#</u>
	Name CName

Animal	
PK	<u>Species</u>
	Discovered

Likes	
PK	<u>ID#</u>
PK	<u>Species</u>

# ***Terrible Relational Implementation***

- Even primary keys are not specified

Country	
	CName Population

Child	
	ID# Child

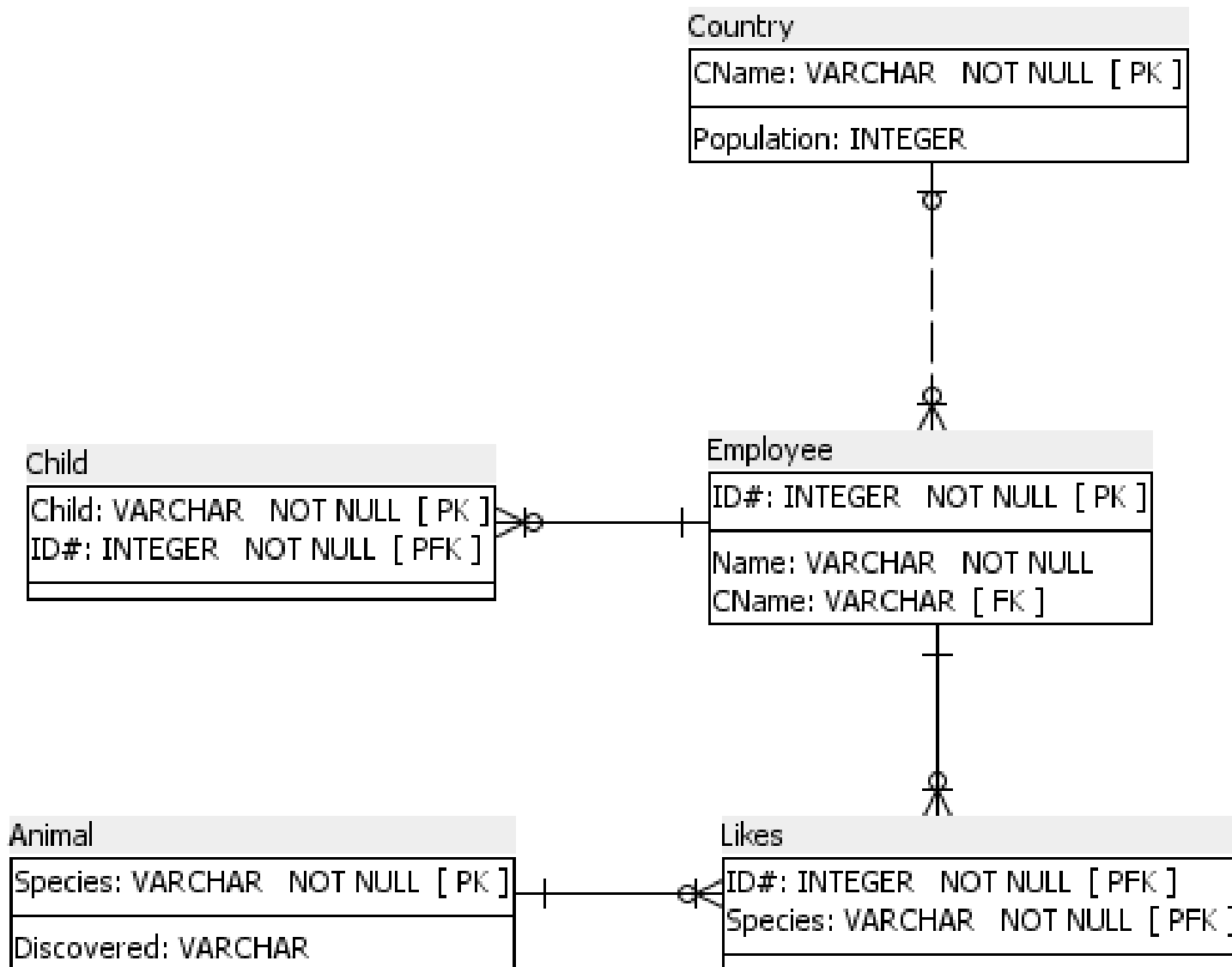
Employee	
	ID# Name CName

Animal	
	Species Discovered

Likes	
	ID# Species



# Relational Implementation Using SQL Power Architect



# ***From ER Diagram to Relational Database A Comprehensive Example***

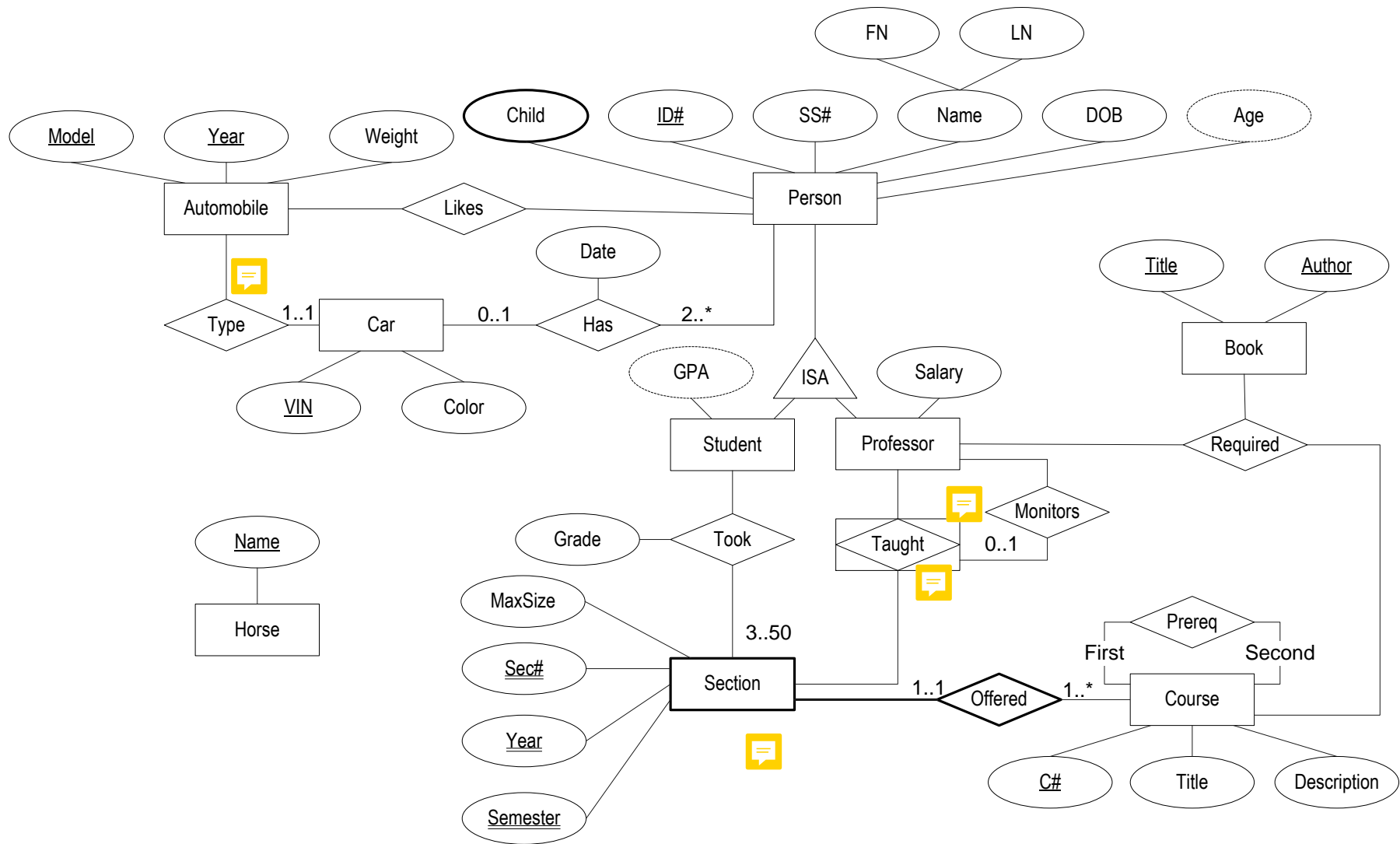
# ***From ER Diagram to Relational Database***

- We now convert our big ER diagram into a relational database
- We specify
  - Attributes that must not be NULL
  - Primary keys
  - Keys (beyond primary)
  - Foreign keys and what they reference
  - Cardinality constraints
  - Some additional “stubs”

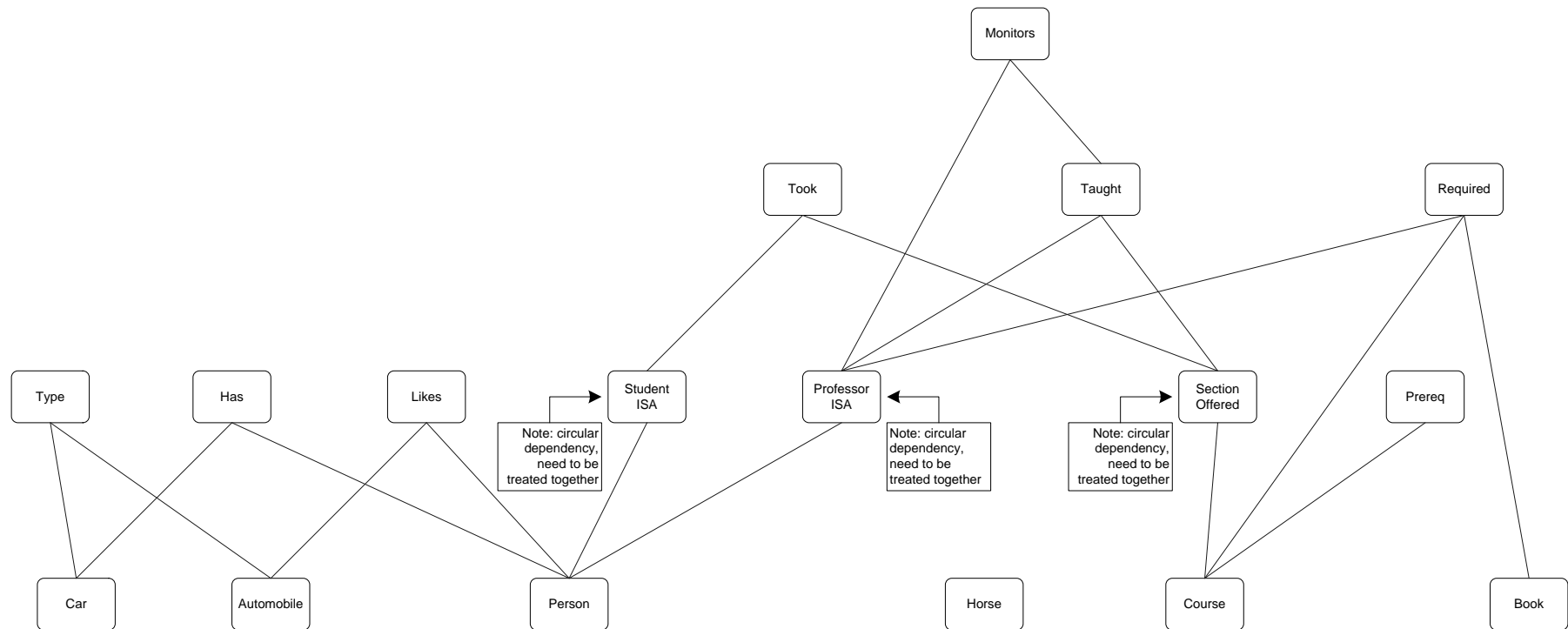
# ***From ER Diagram to Relational Database***

- We both give a narrative description, similar to actual SQL DDL (so we are learning about actual relational databases) and Visio diagrams
- We could have used SQL Power Architect (not that nice)
- We should specify domains also, but we would not learn anything from this here, so we do not do it
- This is really a comprehensive example and not an algorithmic procedure
- It provides intuition to handle cases not explicitly appearing in the example
- We are **not** trying to improve the design: we will do that later in the Normalization Unit: we just implement the ER diagram, but we may discuss “mistakes”
- We go bottom up, in the same order as the one we used in constructing the ER diagram

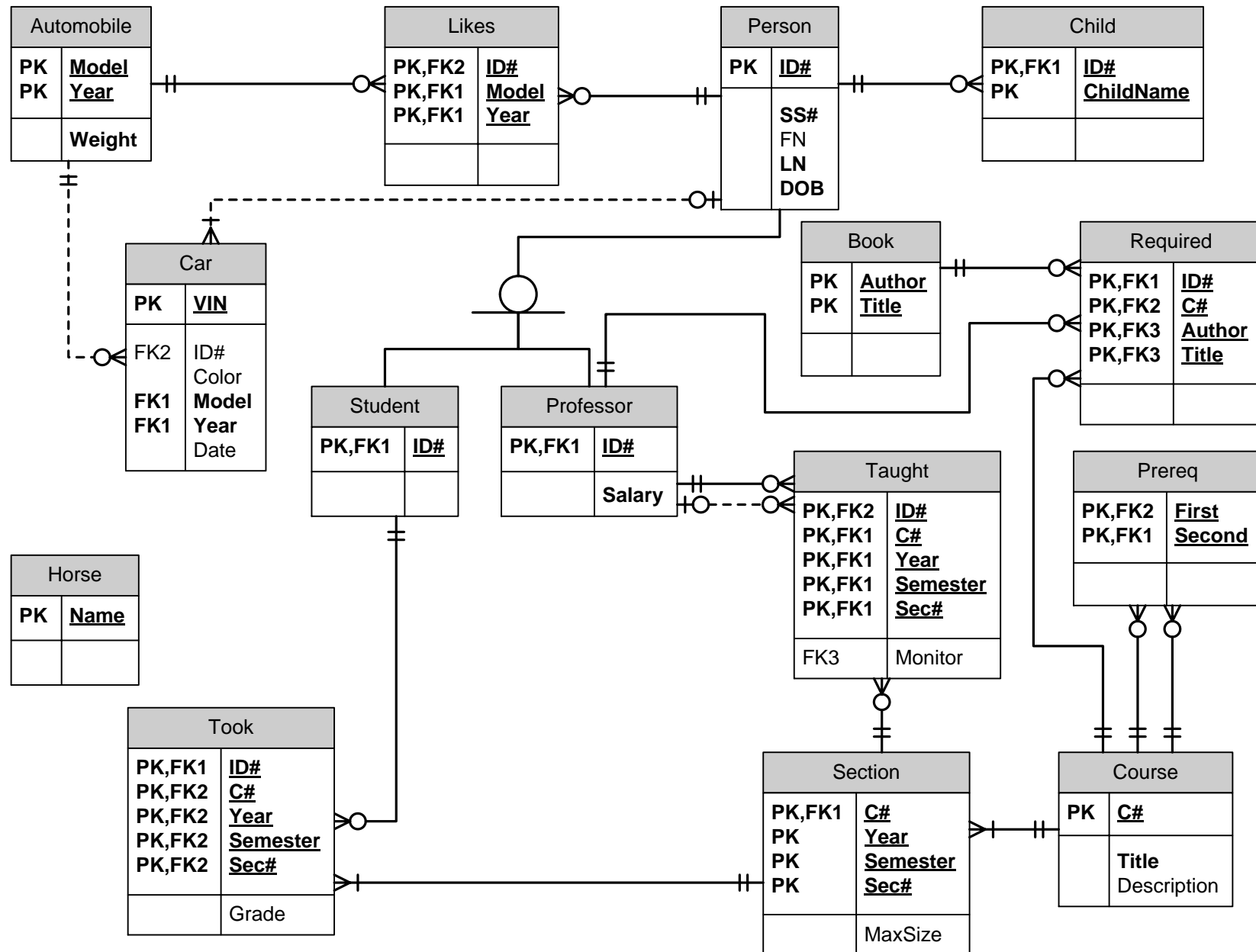
# Our ER Diagram



# *Hierarchy for Our ER Diagram*



# We Will Produce



# *Horse*

- Define Table Horse (  
Name NOT NULL,  
Primary Key (Name));
- This represents the simplest possible relational database
  - One table with one attribute



# *Horse*

Horse	
<b>PK</b>	<b><u>Name</u></b>

# ***Person***

- Person has some interesting attributes
- Multivalued attribute: we will create another table
- Derived attribute: we do not create a column for it, it will be computed as needed
- Composite attribute: we “flatten” it

# *Person*

- Define Table Person (  
ID# NOT NULL,  
SS# NOT NULL,  
FN,  
LN NOT NULL,  
DOB NOT NULL,  
Primary Key (ID#),  
Candidate Key (SS#),  
Age (computed by procedure ...) );
- In SQL DDL, the keyword UNIQUE is used instead of Candidate Key, but “Candidate Key” is better for reminding us what this could be
- Age would likely not be stored but defined in some view

# Person

Person	
PK	<u>ID#</u>
	SS# FN LN DOB

Horse	
PK	<u>Name</u>

# *Child*

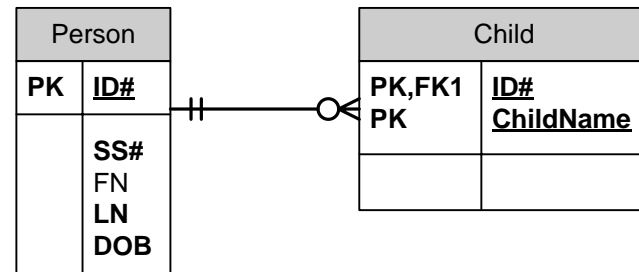
- Define Table Child (  
ID# NOT NULL,  
ChildName NOT NULL,  
Primary Key (ID#,ChildName),  
Foreign Key (ID#) References Person );
- This lists all pairs (ID# of person, a child's name)
  - We have chosen a more descriptive attribute name than the one in the ER diagram for children's names
- Note
  - A person may have several children, each with a different name
  - Two different persons may have children with the same name
- Because of this, no single attribute can serve as primary key of Child

## ***Person and Child***

- Note that some attributes are not bold, such as FN here
- This means that FN could be NULL (in this context, meaning empty)
  
- Note the induced many-to-one relationship
- We need to make sure we understand what the line ends indicate
  - A person may have 0 or more children (unbounded)
  - A child has exactly 1 person to whom it is attached
  
- We need to pay attention to such matters, though we are generally not going to be listing them here

But you should look at all lines and understand the ends and the patterns (solid or dashed)

# Person and Child



Horse	
PK	<u>Name</u>

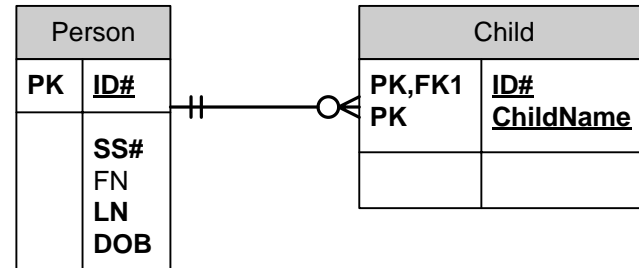
# ***Automobile***

- Define Table Automobile (  
Model NOT NULL,  
Year NOT NULL,  
Weight NOT NULL,  
Primary Key (Model,Year) );



# Automobile

Automobile	
PK	<u>Model</u>
PK	Year
	Weight

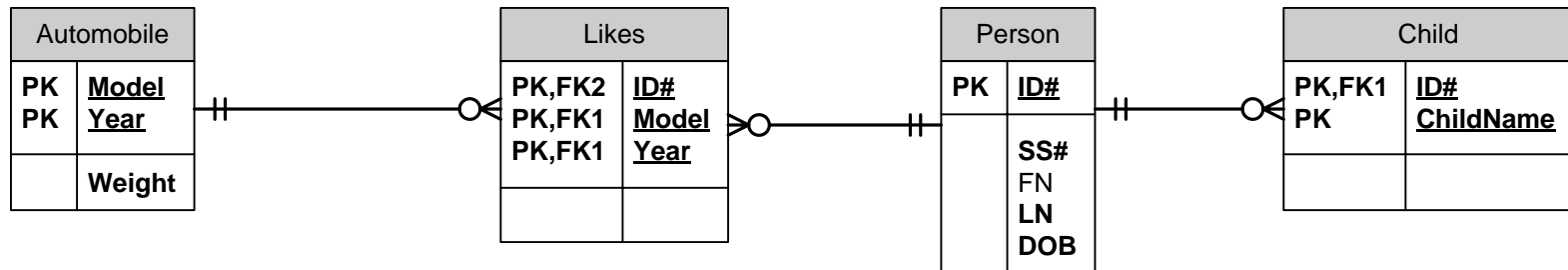


Horse	
PK	<u>Name</u>

# Likes

- Define Table Likes (  
ID# NOT NULL,  
Model NOT NULL,  
Year NOT NULL,  
Primary Key (ID#,Model,Year),  
Foreign Key (ID#) References Person,  
Foreign Key (Model,Year) References Automobile );
- ***Note: the following is bad/incorrect***, replacing one line by two lines
  - Foreign Key (Model) References Automobile
  - Foreign Key (Year) References Automobile
- There are induced binary many-to-one relationships between
  - Likes and Person
  - Likes and Automobile

# Likes

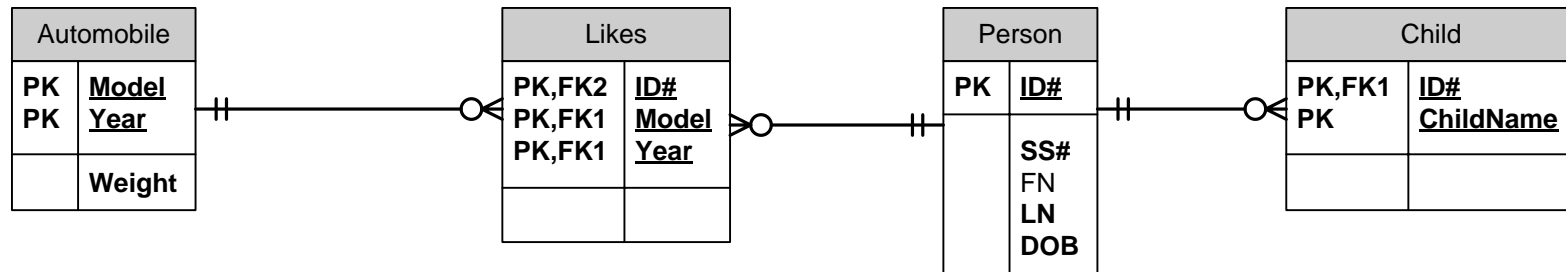


Horse	
PK	<u>Name</u>

## ***Car***

- Define Table Car (  
VIN NOT NULL,  
Color,  
Primary Key (VIN) );

# Car



Car	
PK	<u>VIN</u>
	Color

Horse	
PK	<u>Name</u>

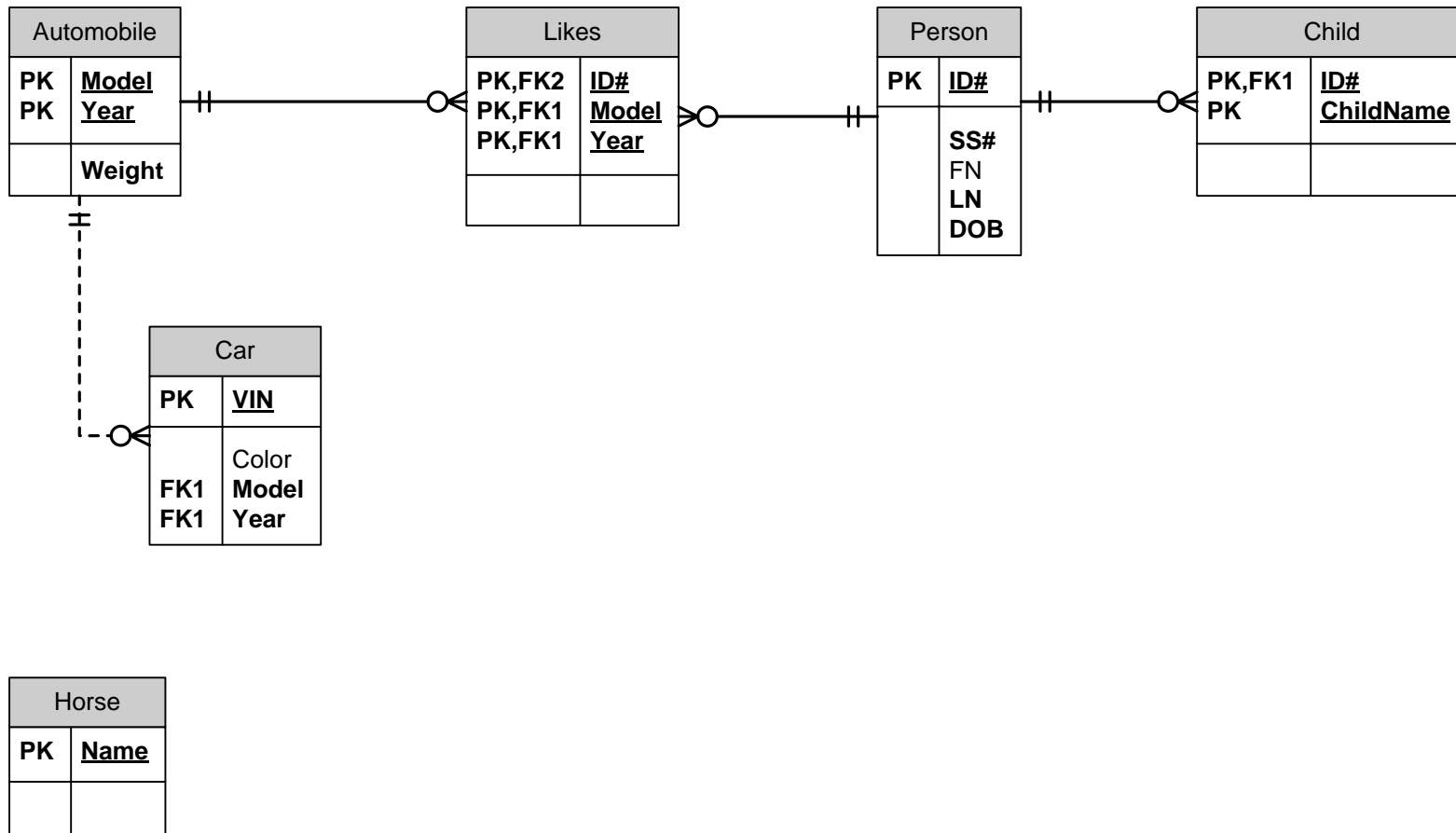
# *Type*

- There is no need for a table for Type as Type is a binary many-to-one relationship
- Type is actually “stored” in the “many” side, that is in Car

# ***Car***

- Define Table Car (  
VIN NOT NULL,  
Color,  
Model NOT NULL,  
Year NOT NULL,  
Primary Key (VIN),  
Foreign Key (Model,Year) References Automobile );

# Type





# *Has*

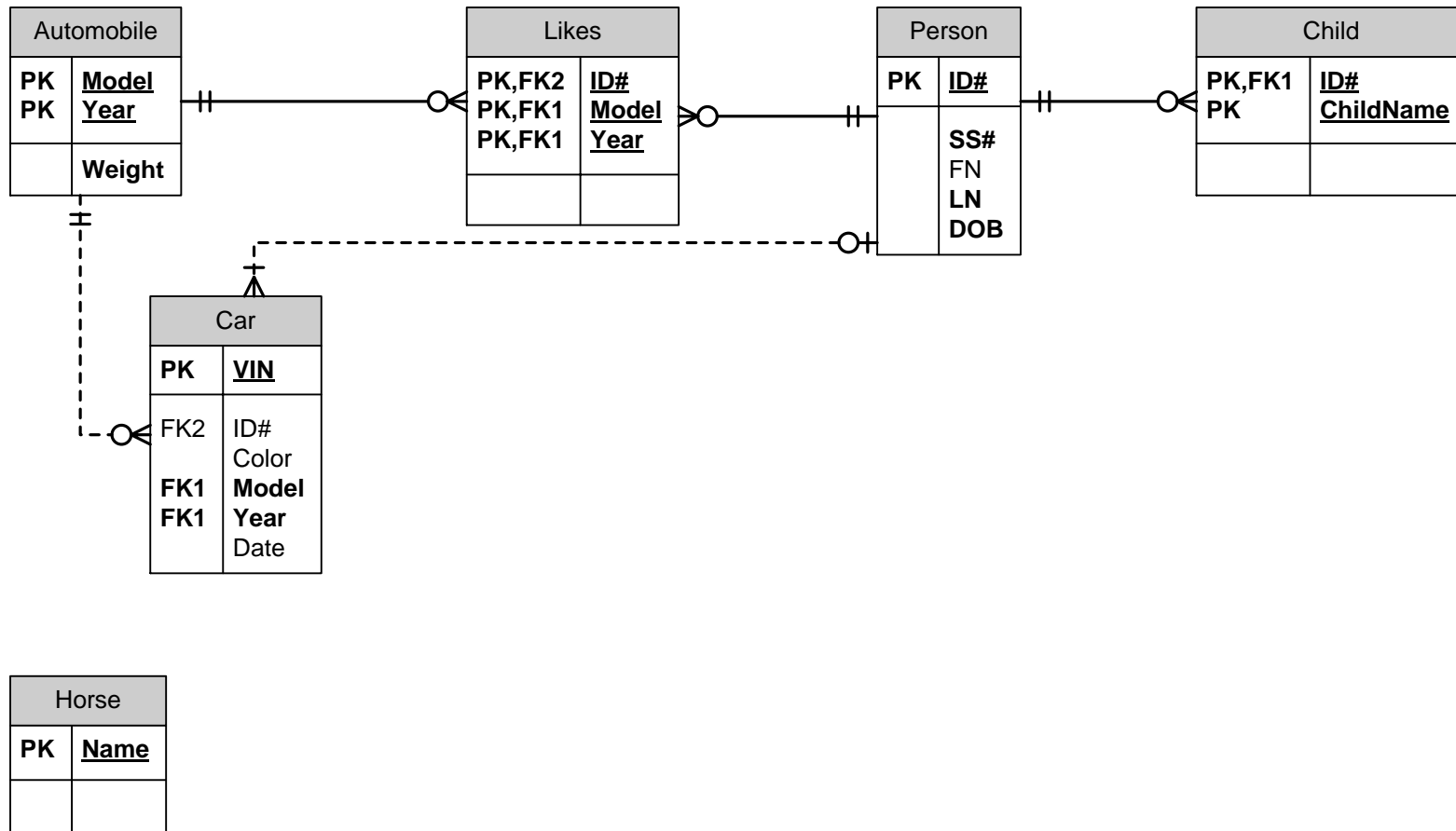
- As Has is a binary many-to-one relationship, the attributed of this relationship, Date, is stored in the “many” side, Car
- There is no need for a table for Has as Has is a binary many-to-one relationship
- It is essentially “stored” in the “many” side, that is in Car
- We can only specify that a Person has at least 1 Car with the notation we currently use
- The CHECK condition is specified using appropriate SQL constraint syntax

This can actually be done in Visio also, and it is done in the examples in ExtrasForUnit03

# ***Car***

- Define Table Car (  
VIN NOT NULL,  
Color,  
Model NOT NULL,  
Year NOT NULL,  
ID#,  
Date,  
Primary Key (VIN),  
Foreign Key (Model,Year) References Automobile  
Foreign Key (ID#) References Person );

# Has



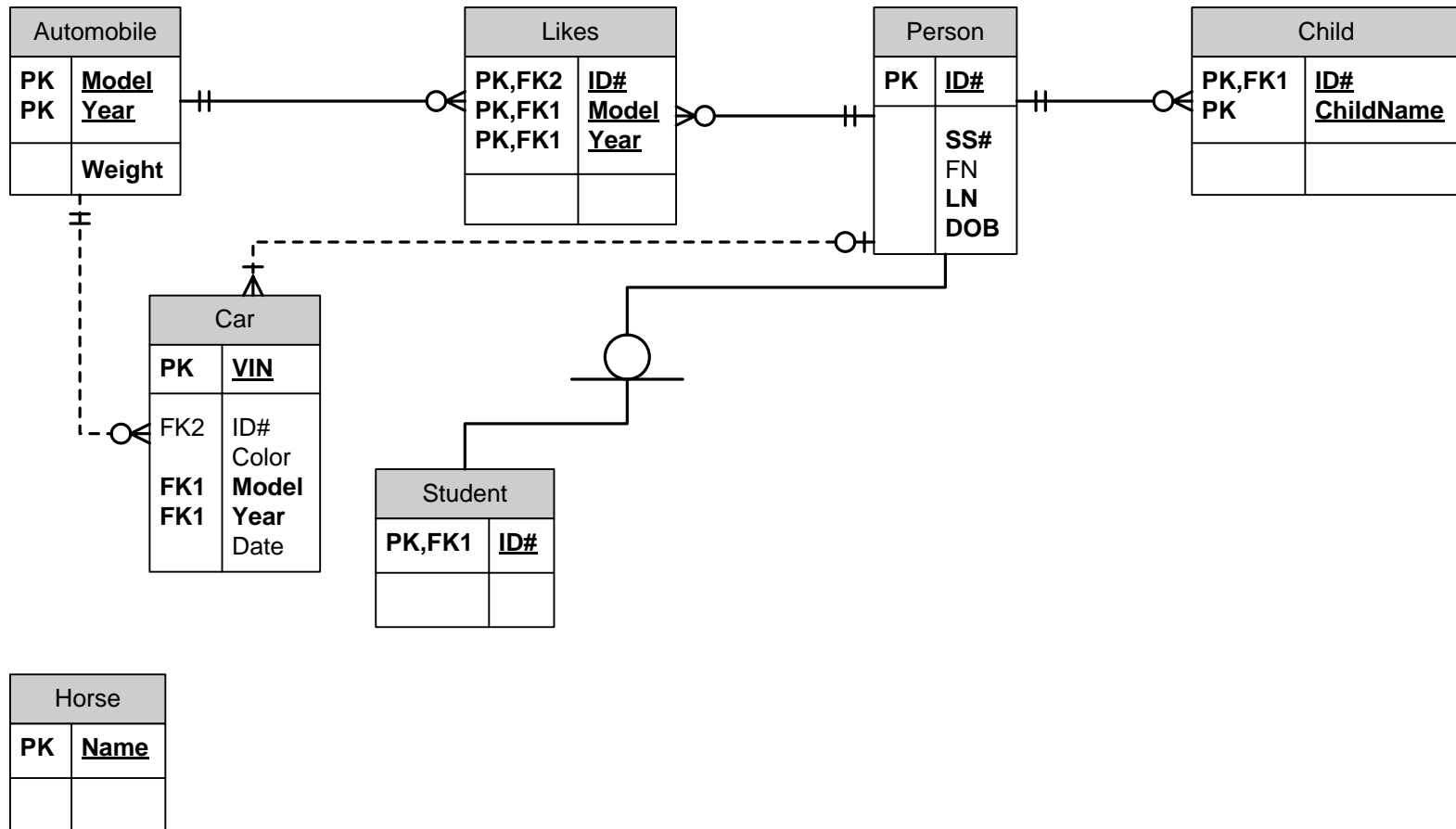
# ***ISA***

- We do not define a table for ISA
- This/these relationship/s is/are “embedded” in Student and Professor

# ***Student***

- Define Table Student (  
ID# NOT NULL,  
Primary Key (ID#),  
Foreign Key (ID#) References Person,  
GPA (computed by procedure ...) );
- Note, how ISA, the class/subclass (set/subset) relations,  
is modeled by Visio
- This is not the case in SQL Power Architect

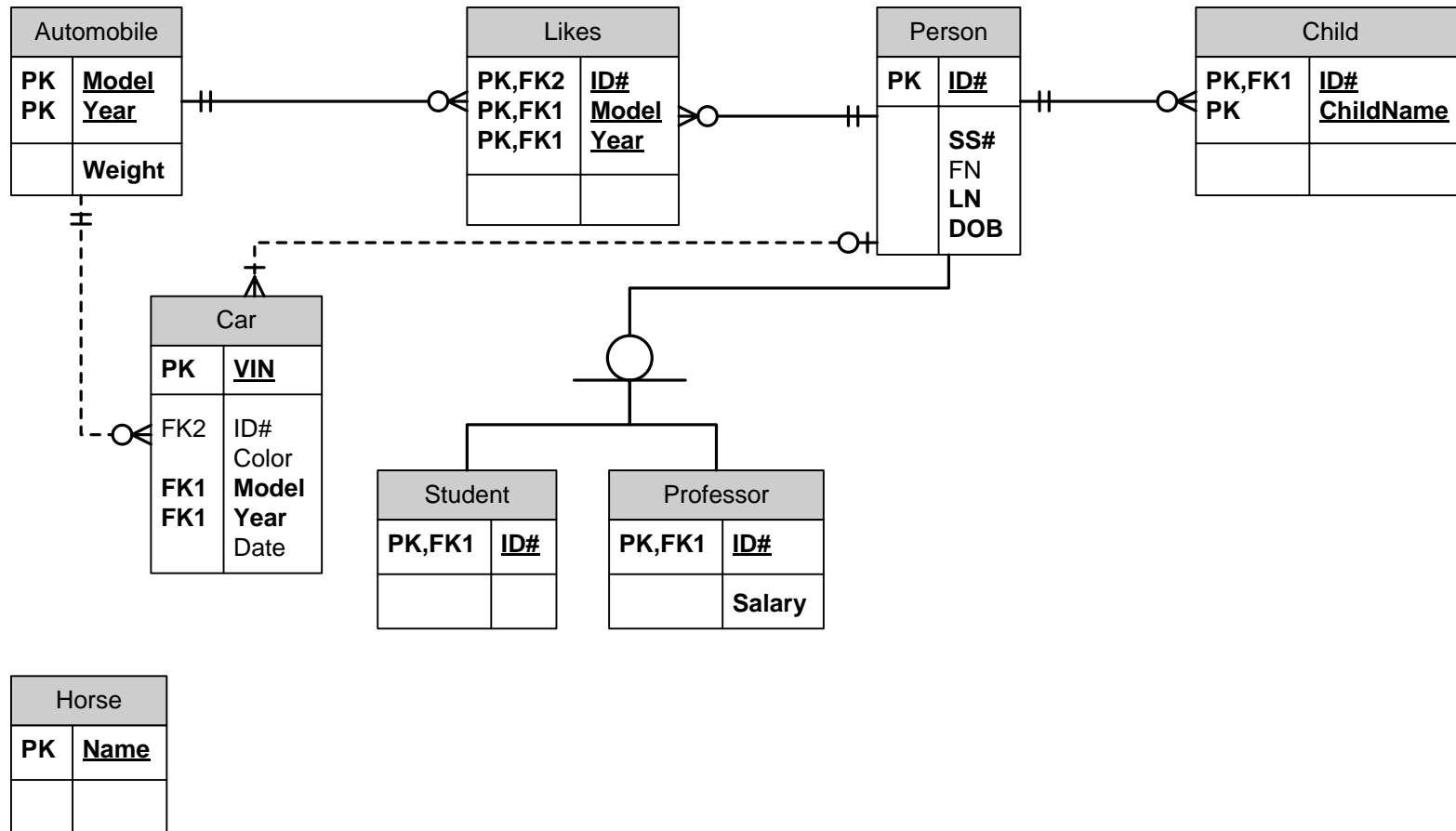
# Student and ISA



# ***Professor***

- Define Table Professor (  
ID# NOT NULL,  
Salary NOT NULL,  
Primary Key (ID#),  
Foreign Key (ID#) References Person );

# Professor and ISA

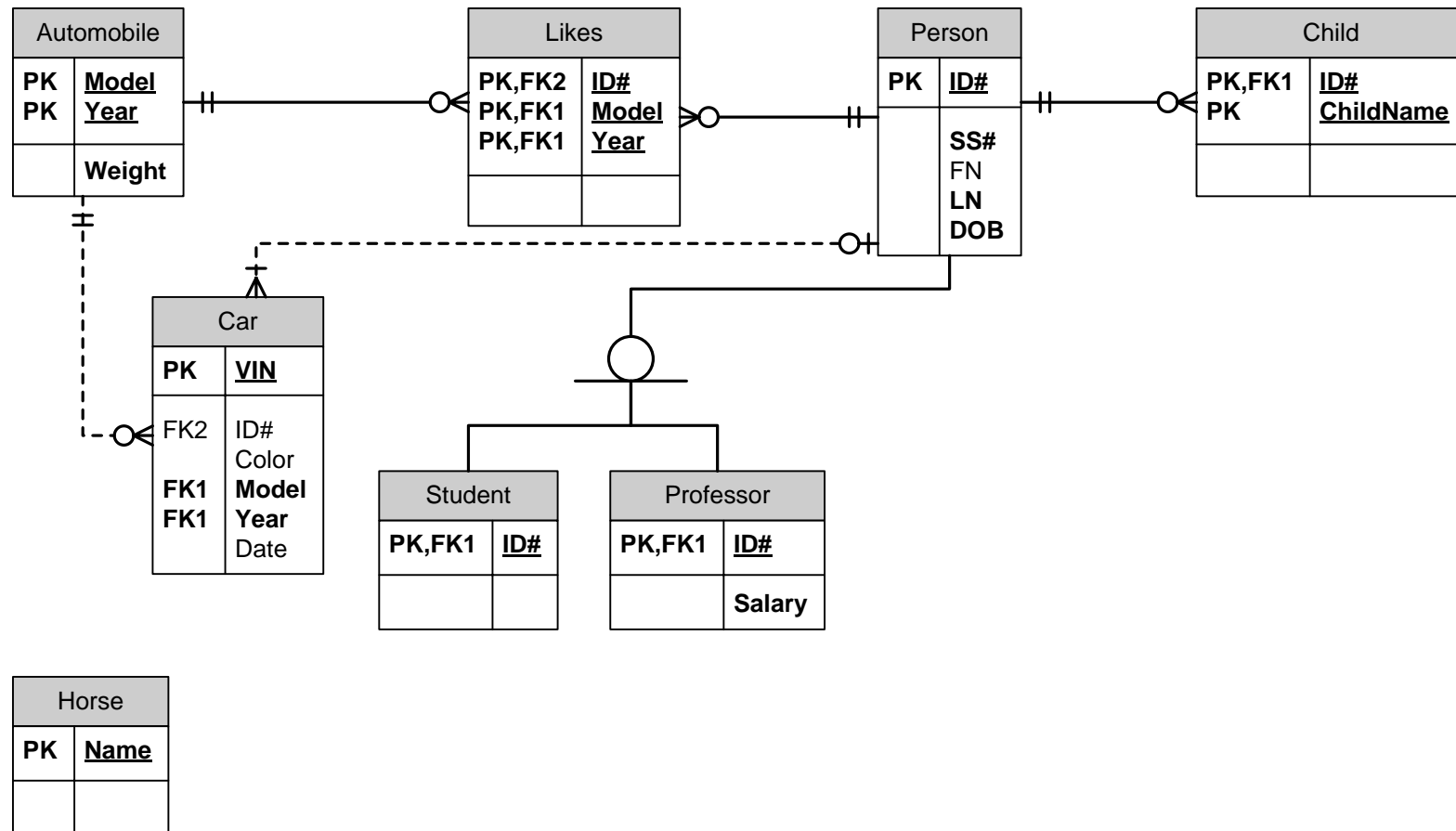




# ***Course***

- Define Table Course (  
C# NOT NULL,  
Title NOT NULL,  
Description,  
Primary Key (C#) );

# Course



Course	
PK	<u>C#</u>
	Title Description

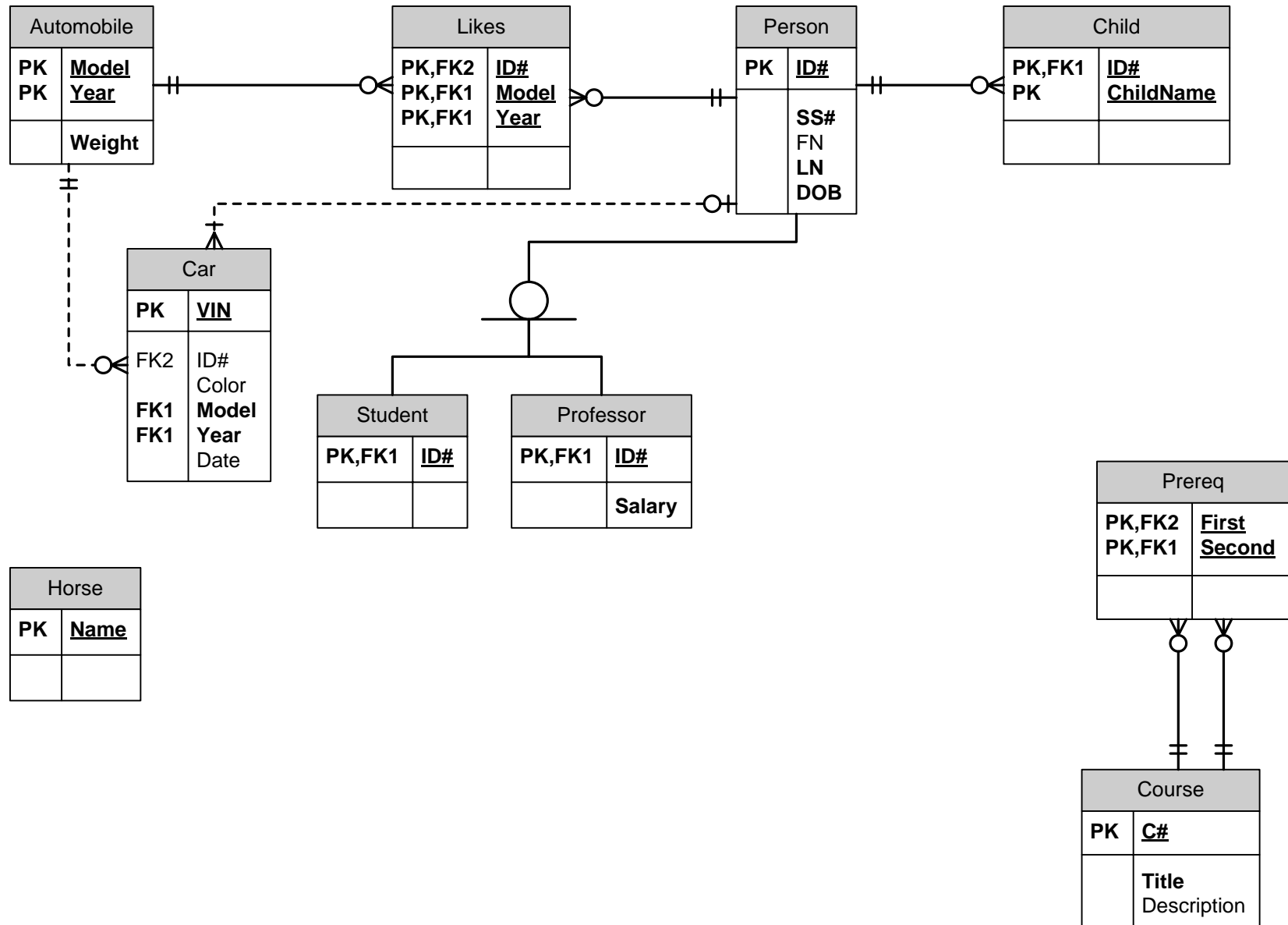
# ***Prerequisite***

- Define Table Prereq (  
First NOT NULL,  
Second NOT NULL,  
Primary Key (First,Second),  
Foreign Key (First) References Course,  
Foreign Key (Second) References Course );

# ***Prereq***

- This is our first example of a table modeling a recursive relationship, between an entity set and itself
- We decide to name the table Prereq, as this is shorter than Prerequisite
- Note that it is perfectly clear and acceptable to refer here to C# by new names: First and Second
  - Similarly, to using ChildName in the Child table
- We should add some constraint to indicate that this (directed graph) should be acyclic (but as annotations)
  - Maybe other conditions, based on numbering conventions specifying course levels

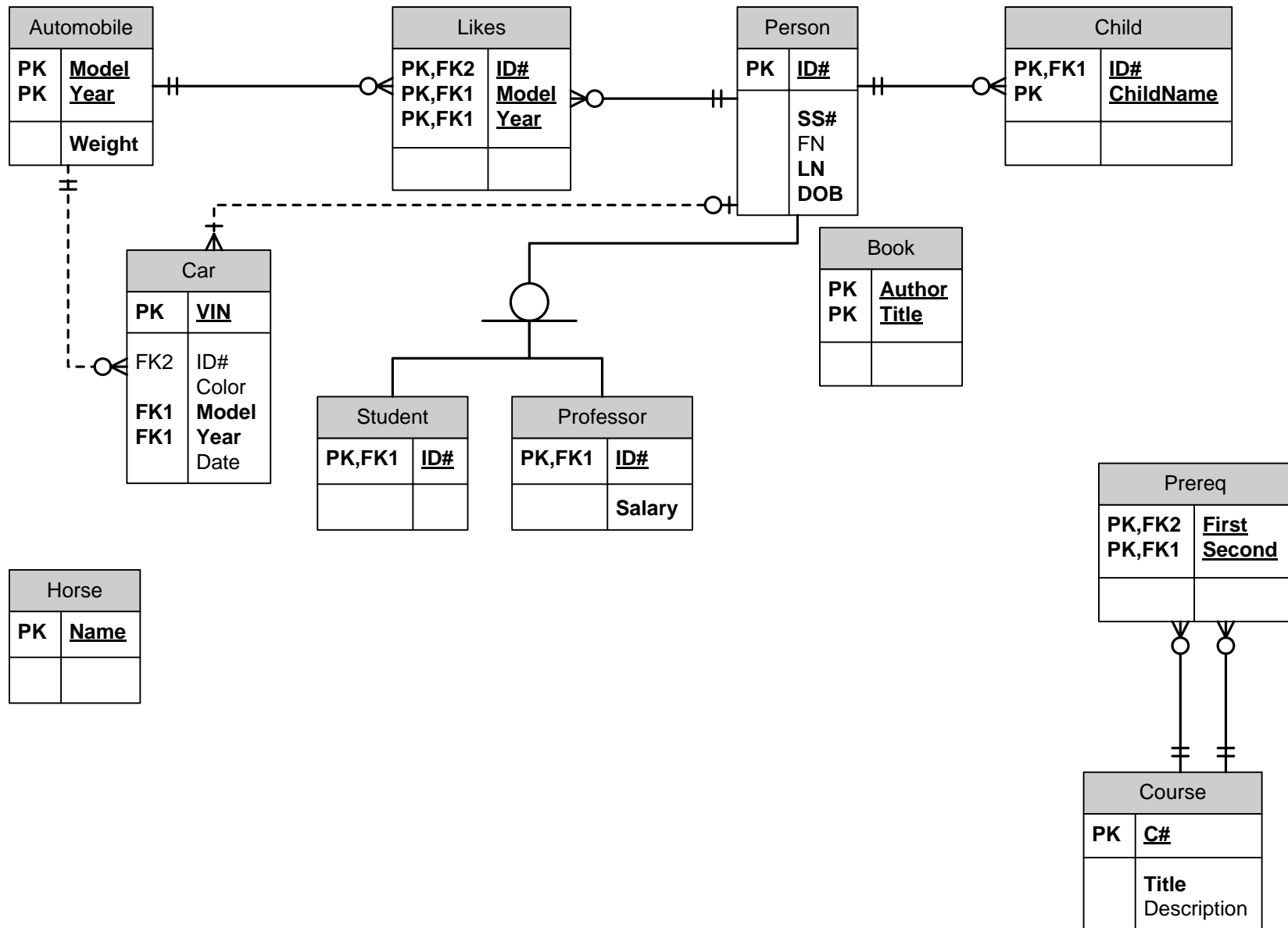
# Prereq



# ***Book***

- Define Table Book (  
Author NOT NULL,  
Title NOT NULL,  
Primary Key (Author,Title) );

# Book



# *Required*

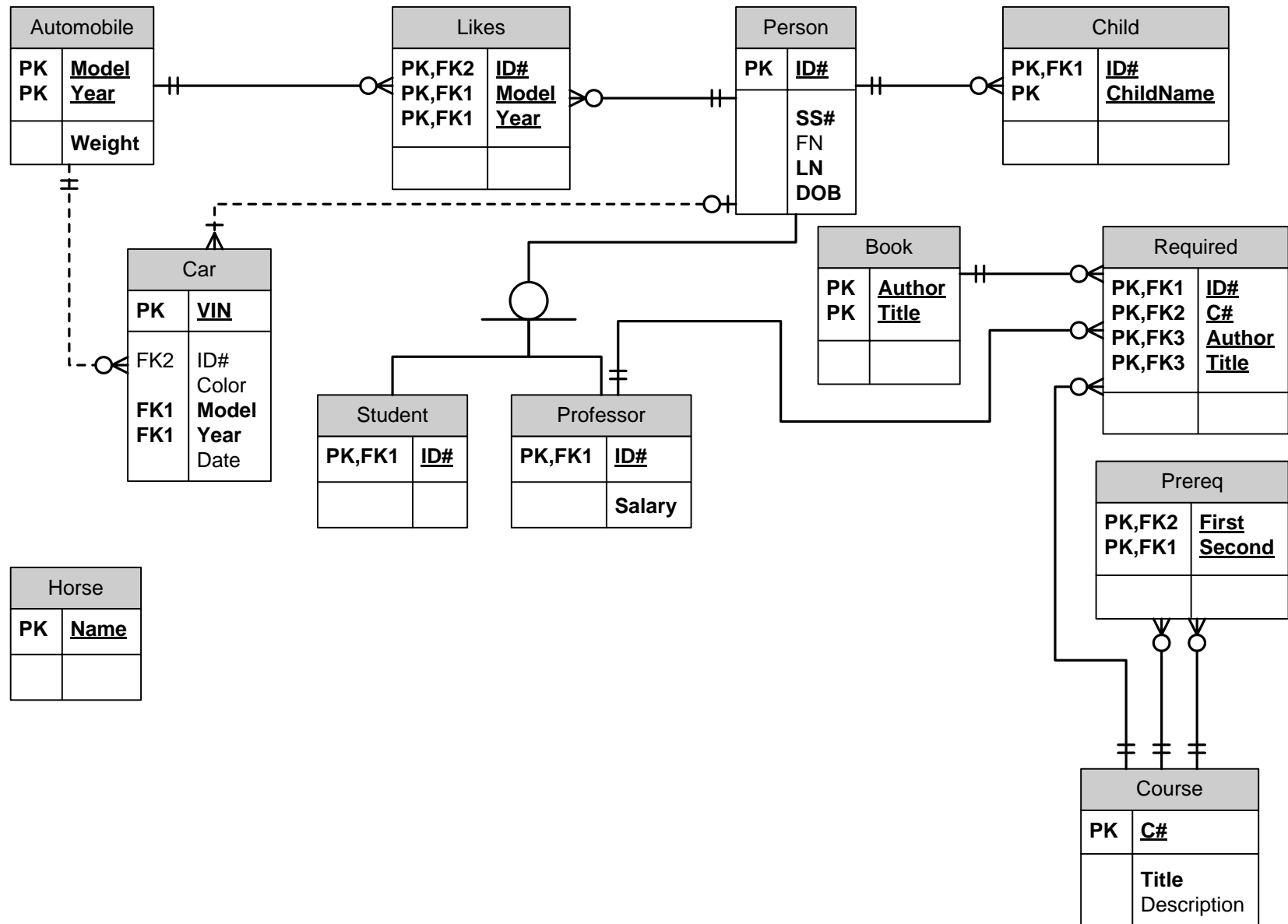
- Define Table Required (  
ID# NOT NULL,  
C# NOT NULL,  
Author NOT NULL,  
Title NOT NULL,  
Primary Key (ID#,C#,Author,Title),  
Foreign Key (ID#) References Professor,  
Foreign Key (C#) References Course,  
Foreign Key (Author,Title) References Book );
- Why is it *bad* to have  
Foreign Key (ID#) References Person,  
instead of  
Foreign Key (ID#) References Professor?  
Because only a Professor can Require a Book



## ***Required***

- This is our first example of a table modeling a relationship that is not binary
- Relationship Required was ternary: it involved three entity sets
- There is nothing unusual about handling it
- We still have as foreign keys the primary keys of the “participating” entities

# Required



## ***Section***

- Define Table Section (  
C# NOT NULL,  
Year NOT NULL,  
Semester NOT NULL,  
Sec# NOT NULL,  
MaxSize,  
Primary Key (C#,Year,Semester,Sec#),  
Foreign Key (C#) References Course );
- Look at the edge between Course and Section, and note how the Section end specifies the requirement of having at least one Section

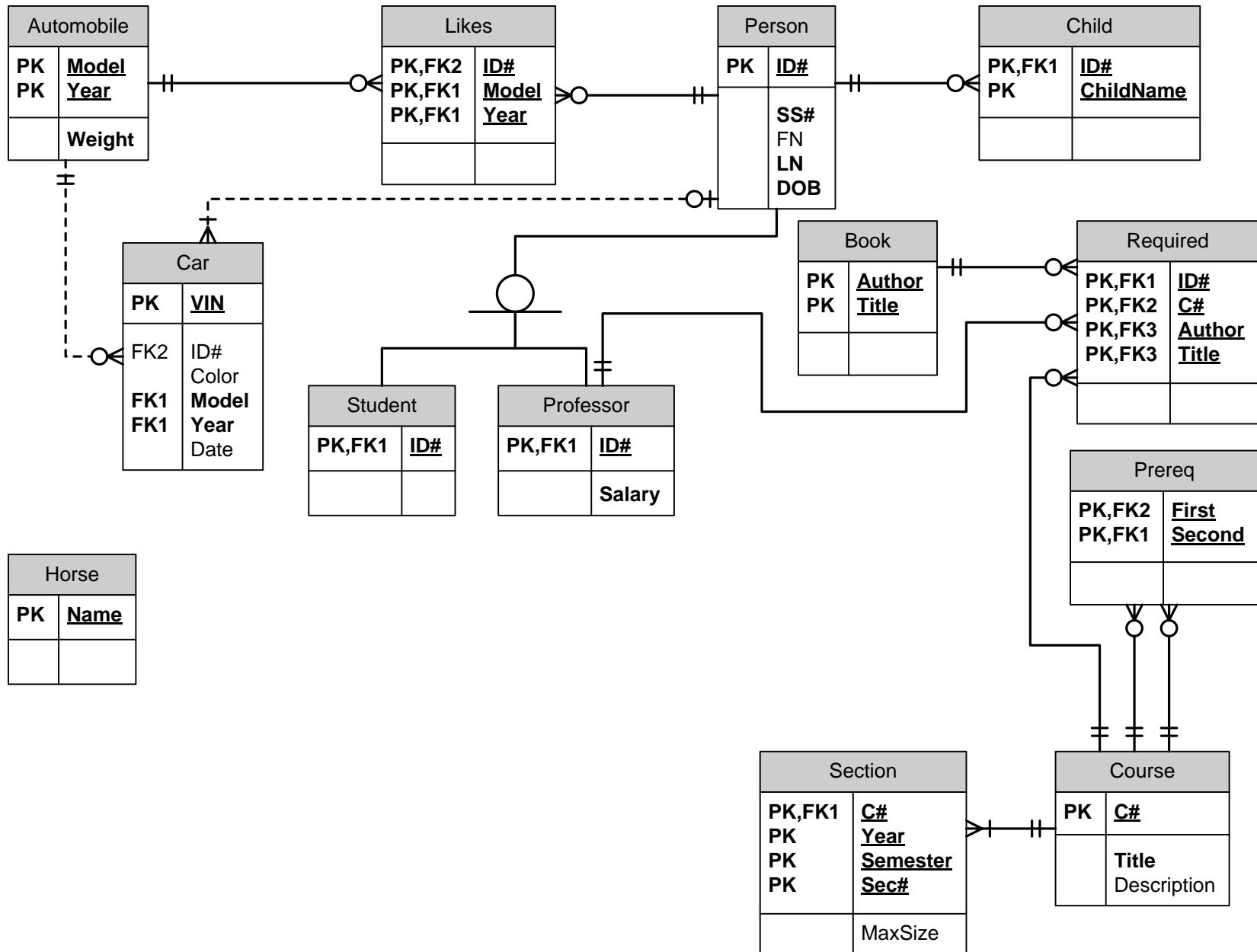
# ***Section***

- Section is our first example of a weak entity

## ***Offered***

- We do not define a table for Offered
- Relationship Offered is implicit in the foreign key constraint

# Section + Offered



# Took

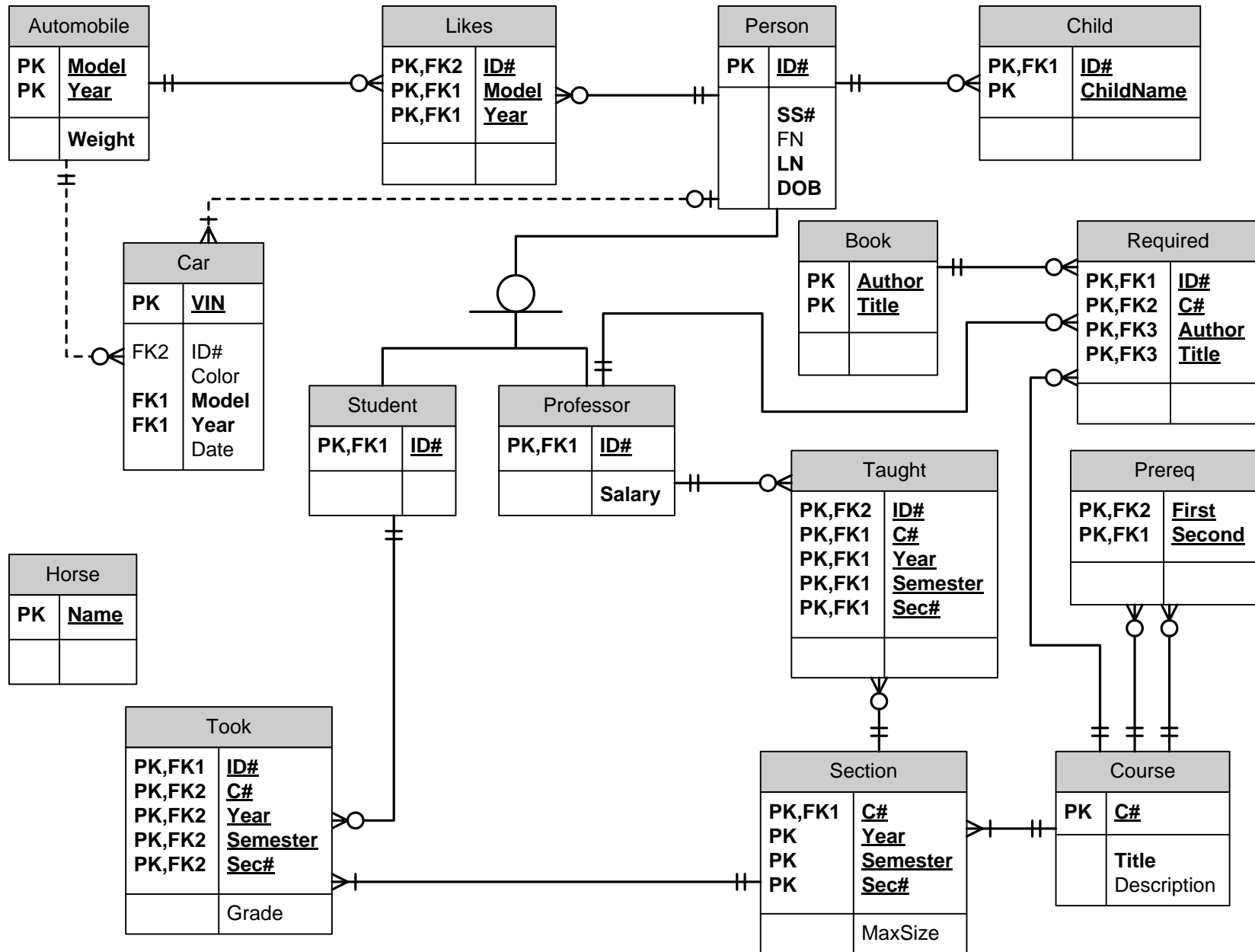
- Define Table Took (  
ID# NOT NULL,  
C# NOT NULL,  
Year NOT NULL,  
Semester NOT NULL,  
Sec# NOT NULL,  
Grade,  
Primary Key (ID#,C#,Year,Semester,Sec#),  
Foreign Key (ID#) References Student,  
Foreign Key (C#,Year,Semester, Sec#) References  
Section );
- Look at the edge between Section and Took, and note how Took end specifies the requirement of having between 3 and 50 students in a section is not fully modeled
- We can only show 1 or more using current notation

# ***Took***

- Because Took is a many-to-many relationship we store its attribute, Grade, in its table
- We cannot store Grade in any of the two
  - Section
  - Student



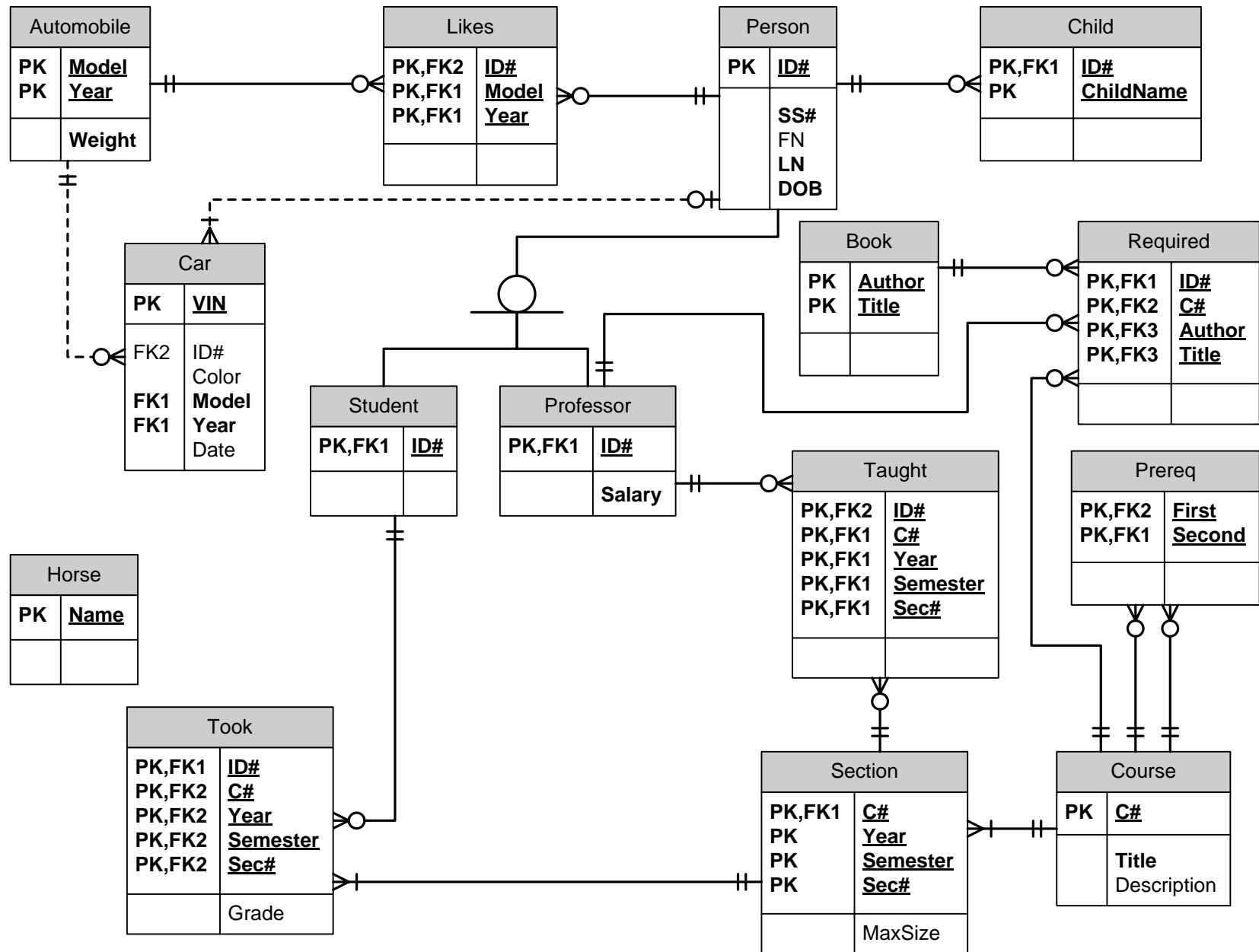
# Took



# ***Taught***

- Define Table Taught (  
ID# NOT NULL,  
C# NOT NULL,  
Year NOT NULL,  
Semester NOT NULL,  
Sec# NOT NULL,  
Primary Key (ID#,C#,Year,Semester,Sec#),  
Foreign Key (ID#), References Professor,  
Foreign Key (C#,Year,Semester,Sec#) References  
Section );

# Taught



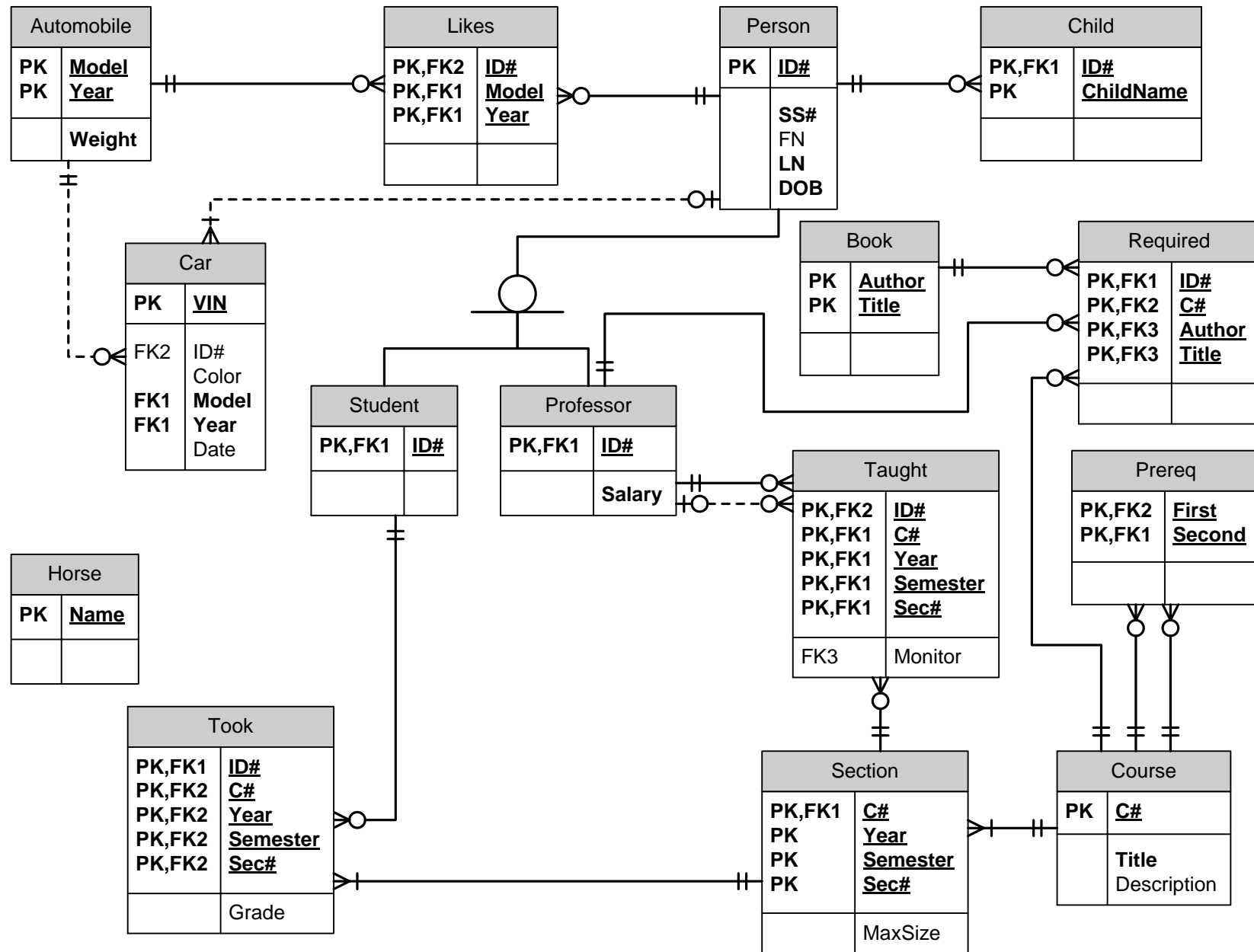
# ***Monitors***

- This is our first example in which a table, Taught, that “came from” a relationship is treated as if it came from an entity and participates in a relationship with other tables
- Nothing special needs to be done to “convert” a table that models a relationship, to be also treated as a table modeling an entity
- In this case, Monitors is a binary many-to-one relationship, so we do not need to create a table for it, and it can be stored in the “many” side, Taught

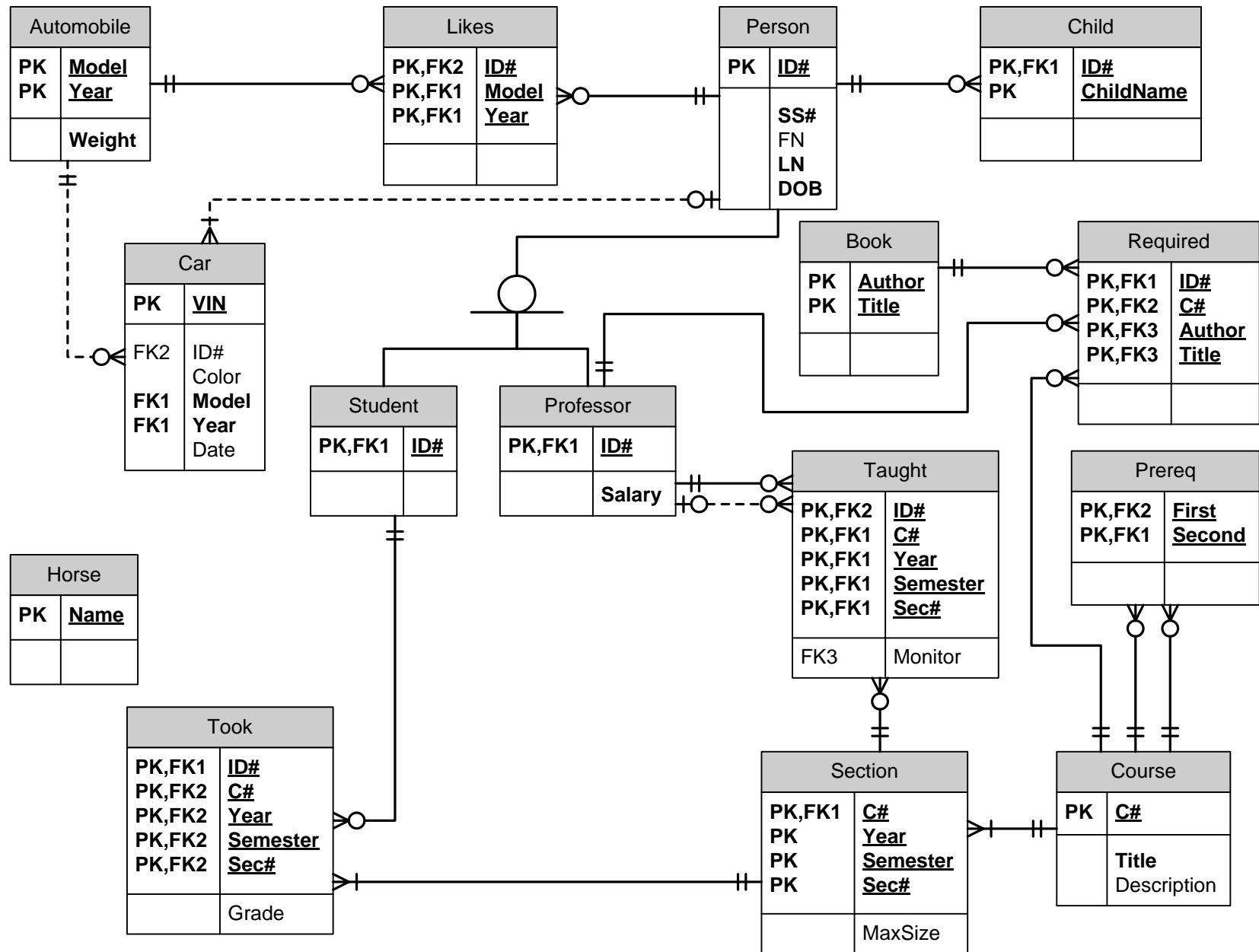
# ***Monitors***

- Define Table Taught (  
ID# NOT NULL,  
C# NOT NULL,  
Year NOT NULL,  
Semester NOT NULL,  
Sec# NOT NULL,  
Monitor  
Primary Key (ID#,C#,Year,Semester,Sec#),  
Foreign Key (ID#), References Professor,  
Foreign Key (C#,Year,Semester,Sec#) References  
Section  
Foreign Key (Monitor) References Professor );
- **Note:** this definition of Taught replaces the original definition as, of course, we do not have two copies of the table Taught

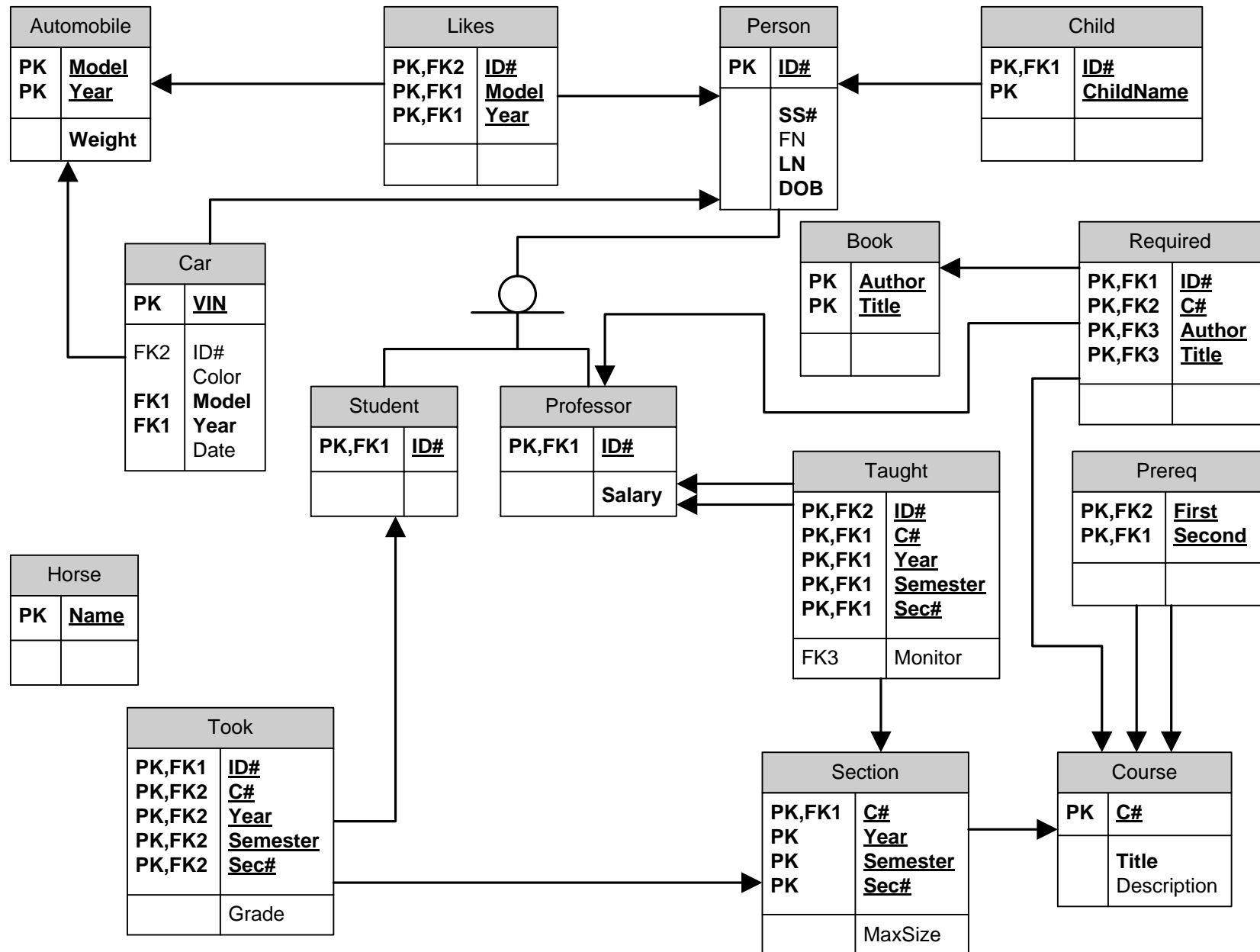
# Monitors



# We Are Done

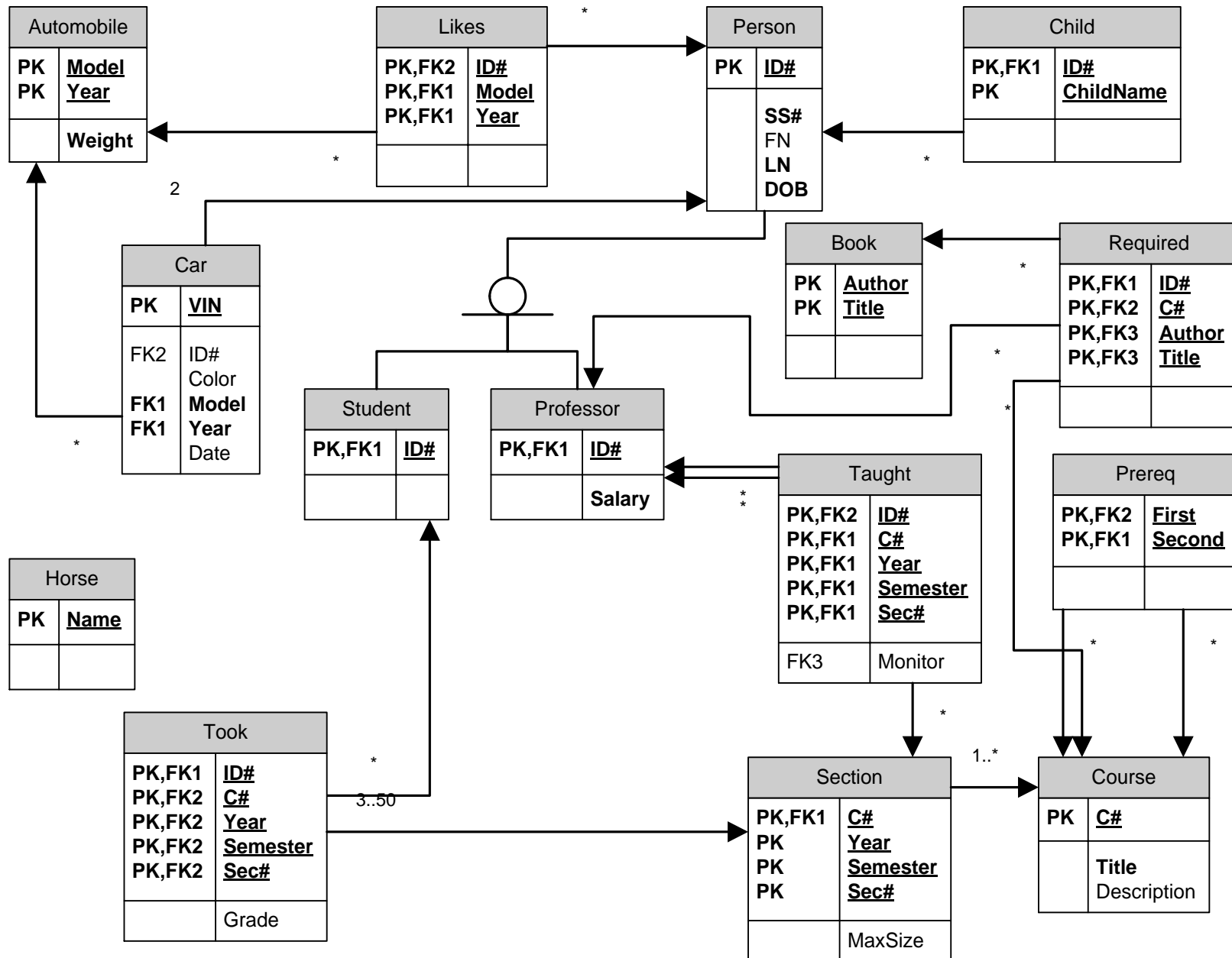


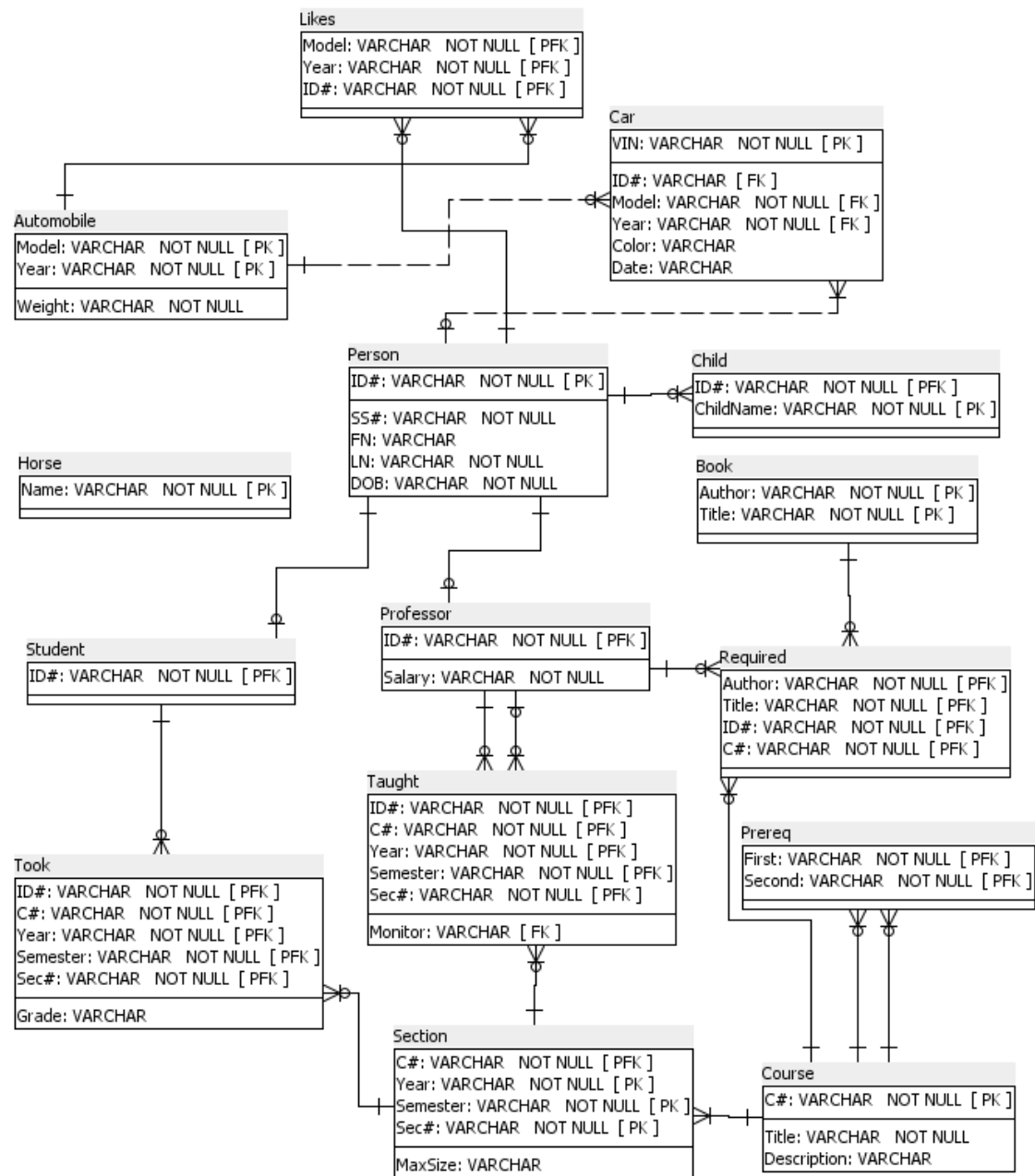
# Arrows Notation



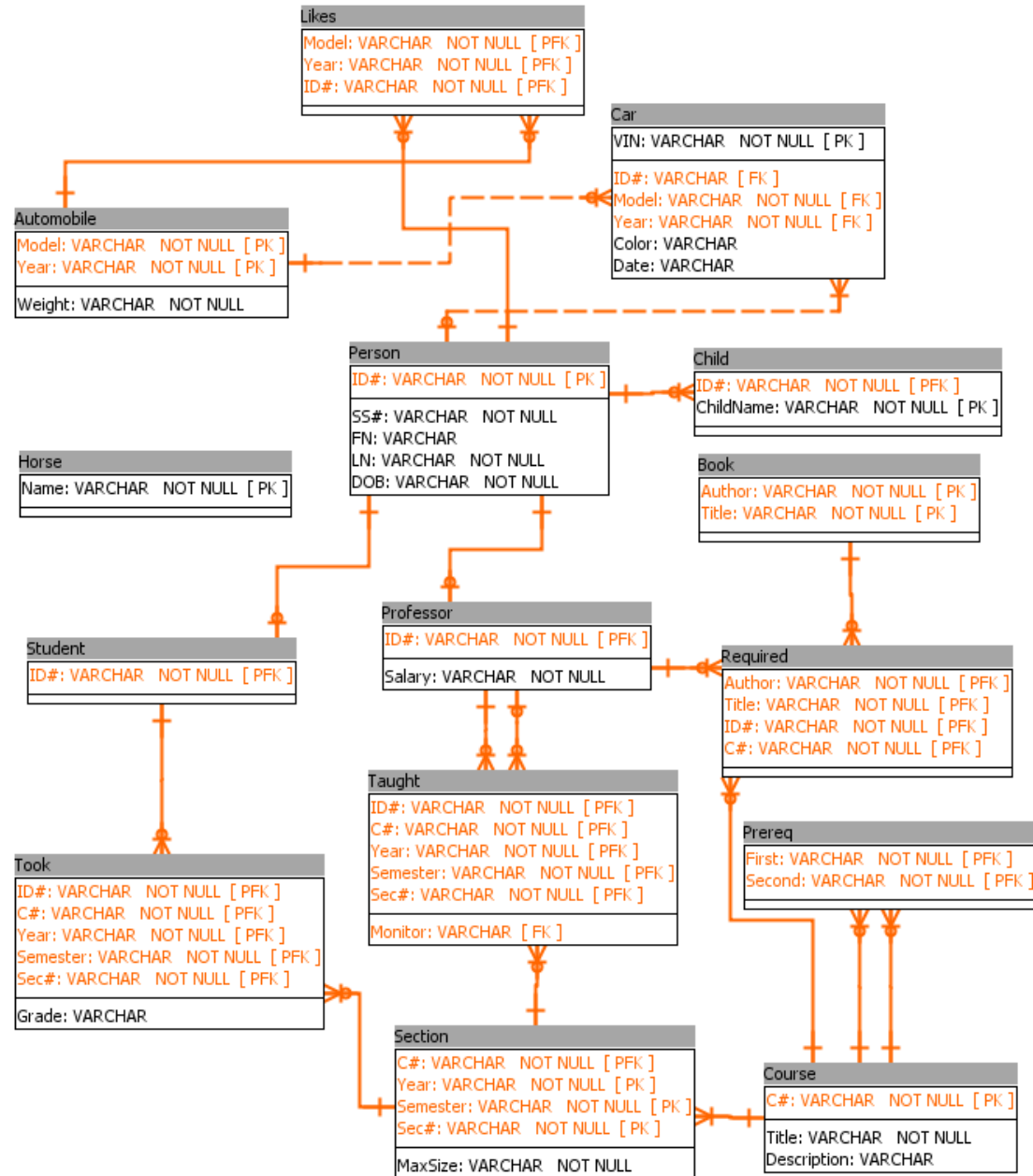


# Arrows and Cardinality Notation





# In SQL Power Architect With Mappings Highlighted



## **Annotations**

### **(The Necessary And Only The Necessary)**

- Unique: SS#
  - A Person has at least 2 Cars
  - A Section has between 3 and 50 students
  - Computed Age in Person using formula ...
  - Computed GPA in Student using formula ...
- 
- Note that we assume that the original .architect file is available and therefore we can click on the lines and see the mappings explicitly
- 
- We only look at annotations for the .architect file because this will be the format in the homework and on an exam

# ***Comparing Notations***

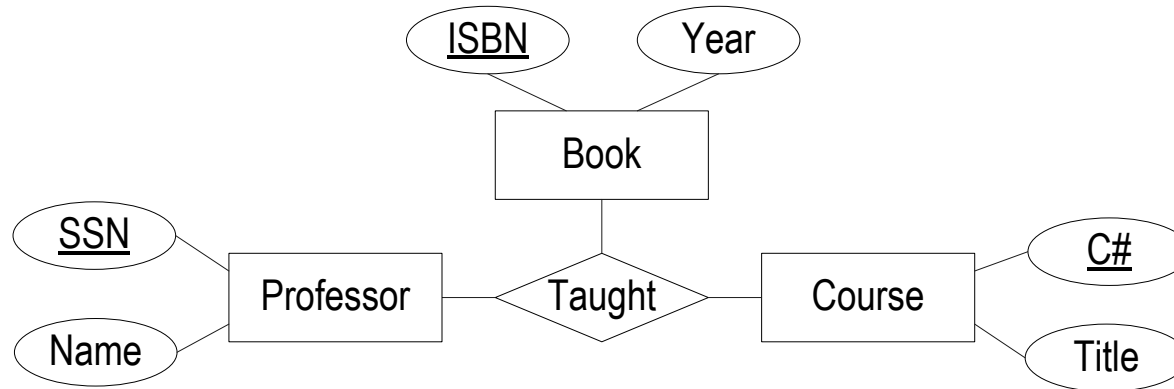
- It is useful to go over the last slides to see the different ways the binary many-to-one mappings are represented
- Note specifically how the endings of lines are represented in Visio (more conventional way) and in SQL Power Architect

# ***Non-Binary Relationships That Are Many-To-One***

# ***Non-Binary Relationships That Are Many-To-One***

- We could not discuss this in the context of the University database, which was already quite complex
- So we will discuss this now on a number of small examples
- So issue will be how to select the primary key
- Our example will be a tiny database, essentially “extracted” from the University database
- The example will deal with professors using books in courses

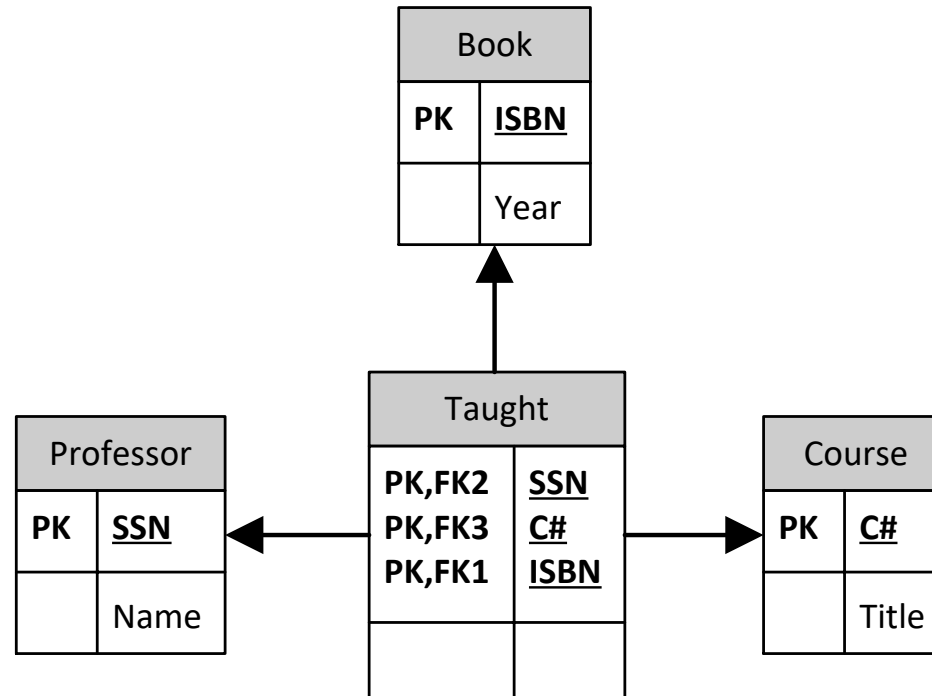
# ***Non-Binary Relationships That Are Many-To-One***



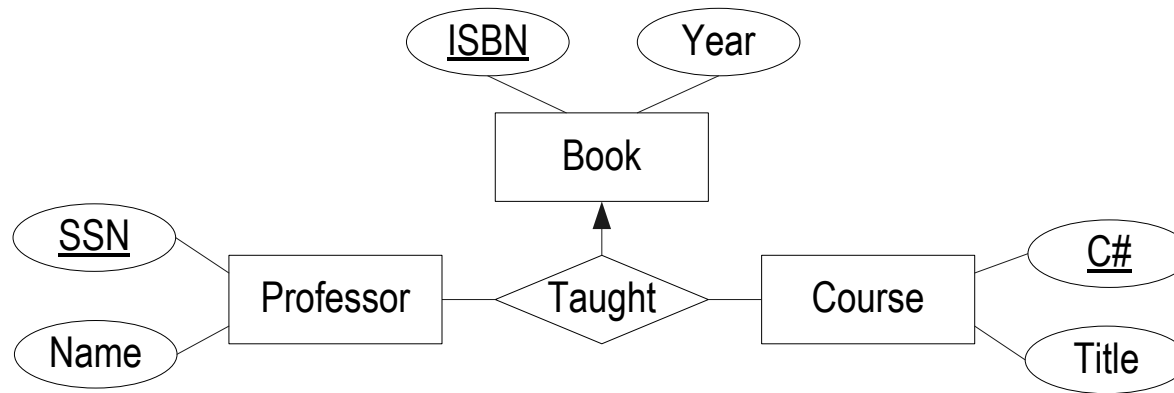
- No additional specifications



# *Non-Binary Relationships That Are Many-To-One*

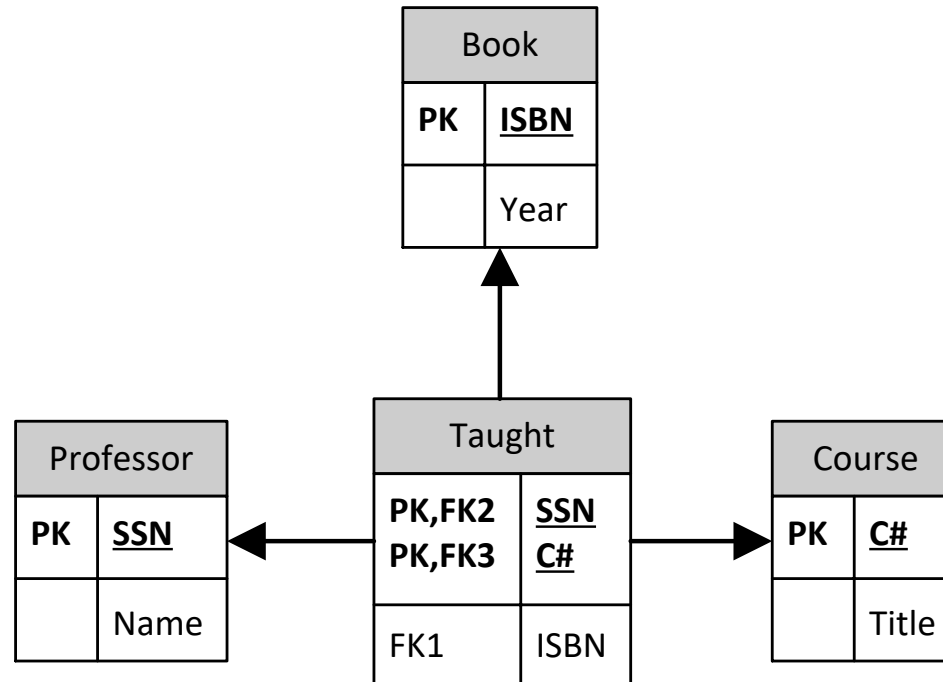


# ***Non-Binary Relationships That Are Many-To-One***

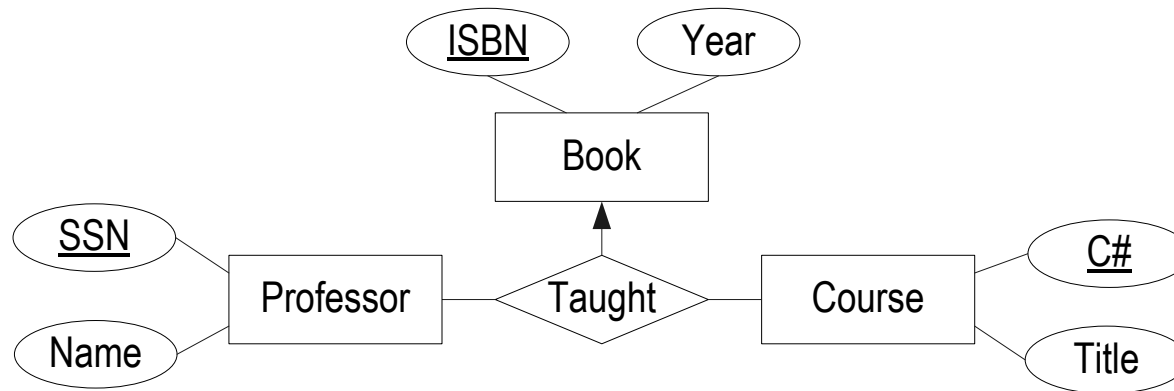


- No additional specification
- We understand this to mean that a professor taught a course using at most one book
- We understand this to mean: Book is a function of (Professor, Course)
  - And this is actually a clearer phrasing
- But in a specific course each professor can use a different book
- But a specific professor can use a different book in each course

# ***Non-Binary Relationships That Are Many-To-One***

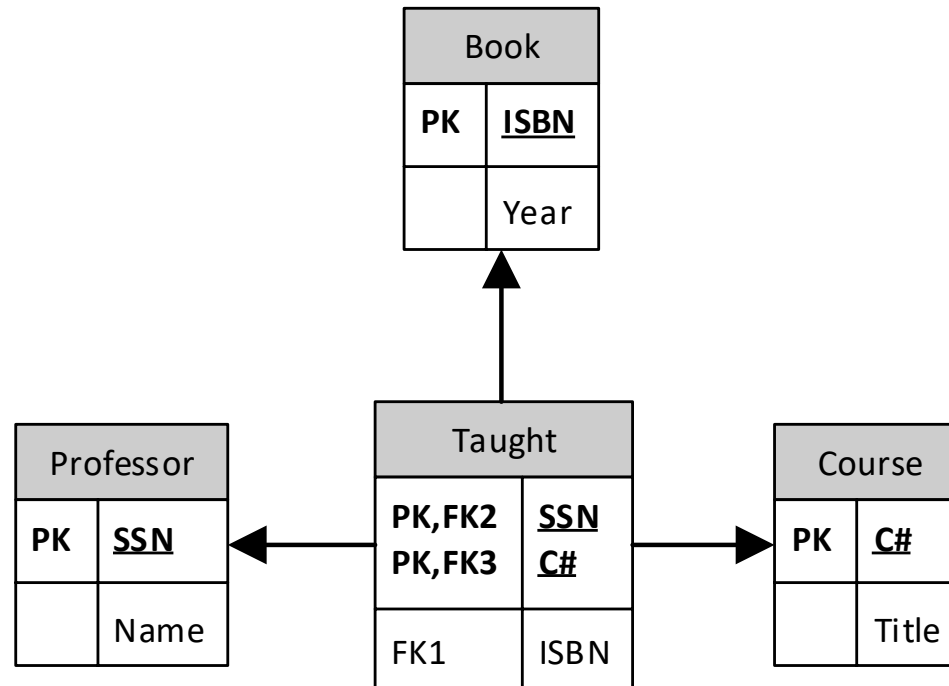


# ***Non-Binary Relationships That Are Many-To-One***



- We are told by an annotation that each course was taught using at most the same (one) book only, no matter which professor taught it
- We understand this to mean: Book is a function of Course
  - And this actually a clearer phrasing but we cannot specify this using our ER diagrams conventions and therefore an annotation is needed
- So in such cases you need to
  - Specify in a drawing using an arrow into the “target” entity set
  - Add an annotation on the “source,” or “sources” entity set(s)

# Non-Binary Relationships That Are Many-To-One

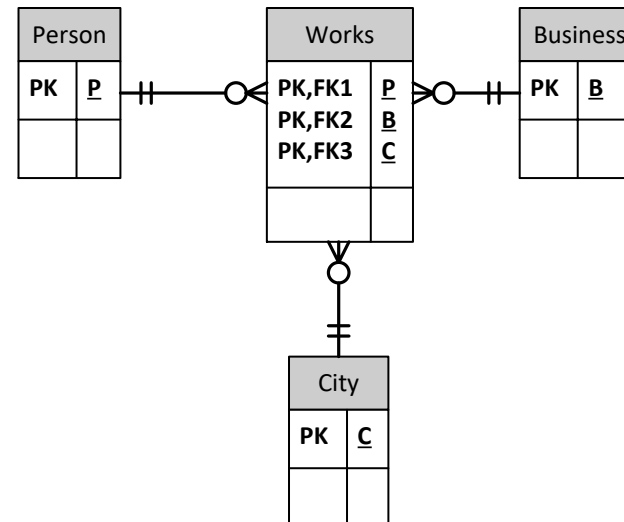
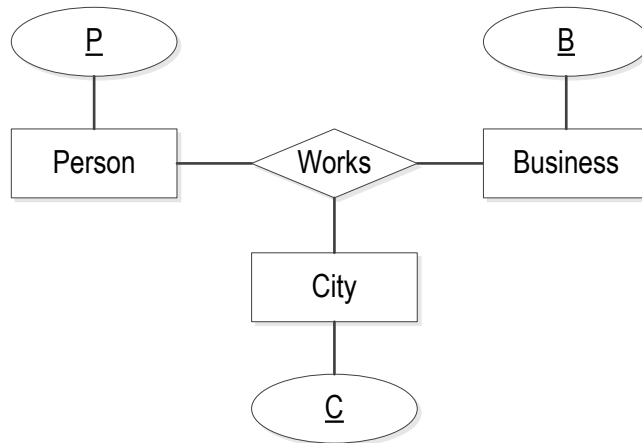


- ***We annotate that ISBN in Taught is only a function of C#***
- Incidentally, this is a bad design, which was derived from the ER diagram without further consideration
- In the Normalization unit we will formally understand why the design was bad and will learn algorithms for automatically fixing such bad designs

## ***More About Non-Binary Relationships***

# General Relationship Implemented As One Table And Several Binary Many-To-One Mappings

- Let's look at an example of a ternary relationship and its implementation



# General Relationship Implemented As One Table And Several Binary Many-To-One Mappings

- Let's look at an instance

Person	<u>P</u>
	a
	b
	c

Works	<u>P</u>	<u>C</u>	<u>B</u>
	a	Boston	Google
	a	Boston	Apple
	a	Chicago	Google
	b	Boston	Google
	b	Chicago	Facebook

Business	<u>B</u>
	Facebook
	Google
	Apple
	Amazon

City	<u>C</u>
	Boston
	Miami
	Chicago

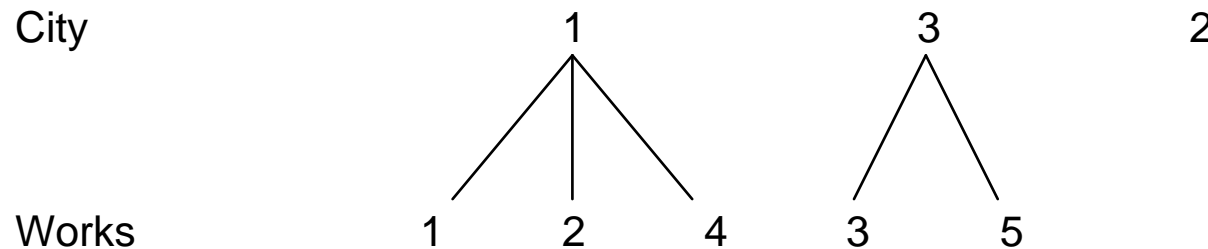
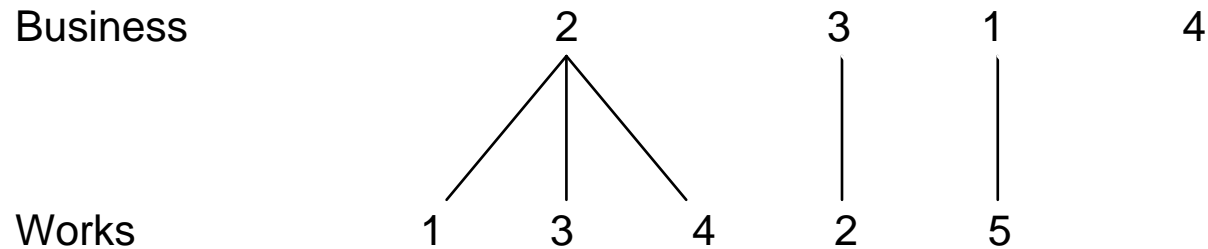
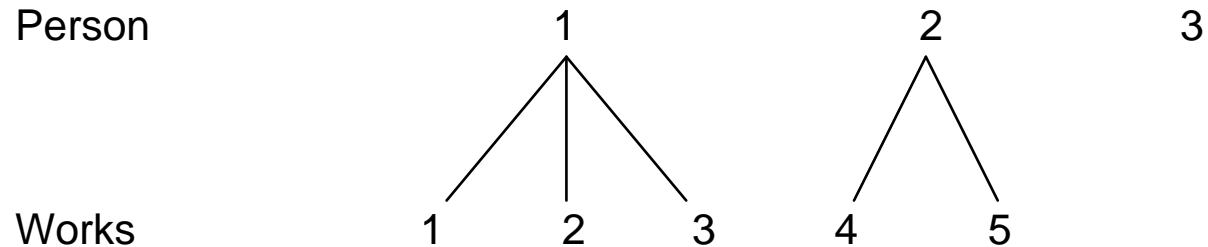


# ***General Relationship Implemented As One Table And Several Binary Many-To-One Mappings***

- Many-to-one from Works to Person
  - Rows 1 2 3 of Works map on Row 1 of Person
  - Rows 3 4 of Works map on Row 2 of Person
  
- Many-to-one from Works to Business
  - Rows 1 3 4 of Works map on Row 2 of Business
  - Row 2 of Works maps on Row 3 of Business
  - Row 5 of Works maps on Row 1 of Business
  
- Many-to-one from Works to City
  - Rows 1 2 4 of Works map on Row 1 of City
  - Rows 3 5 of Works map on Row 3 of City

# General Relationship Implemented As One Table And Several Binary Many-To-One Mappings

- The numbers indicate rows in tables

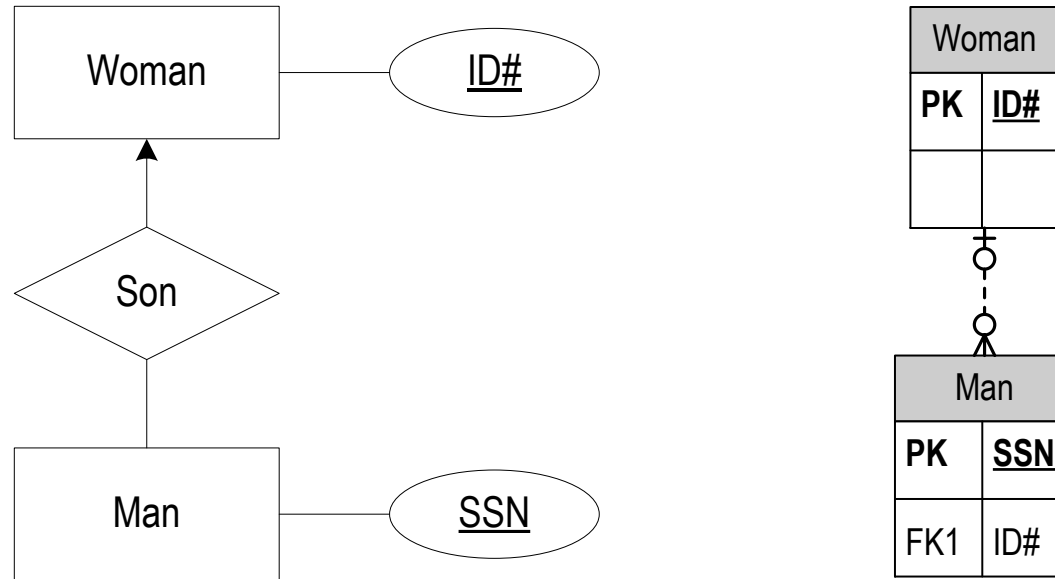


# ***Reiteration on Pattern of Lines***

## ***Dashed Or Solid Lines***

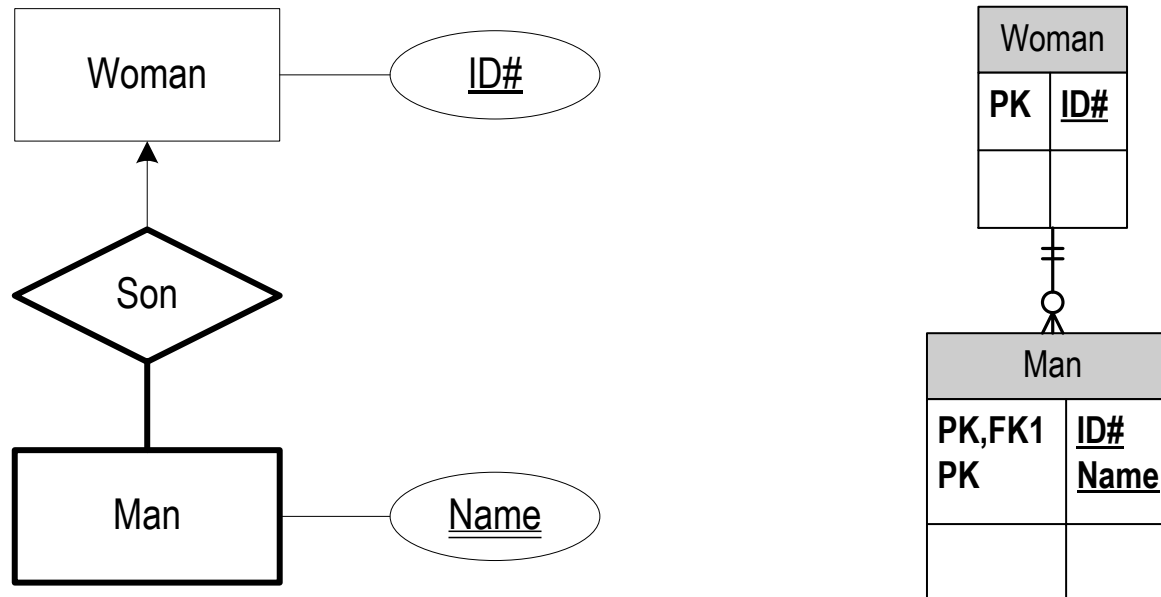
- We are interested in storing information about
  1. Entity Man
  2. Entity Woman
  3. Relationship Son
- We looked at this already when we studied ER diagrams
- We had two cases
  1. Woman was a strong entity and Man was a strong entity
  2. Woman was a strong entity and Man was a weak entity: it was not possible to identify a man based on its own attributes only
- We will now elaborate on relational implementations
- Son is binary many-to-one
  1. One woman has any number of sons
  2. One man has at most one mother (maybe zero in the database)

# Dashed Or Solid Lines



- To store Son we use a **foreign key**
- Man **is completely identified** by its own original attributes: in the table SSN serves as the primary key
- Relationship Son **is not needed** for identifying a Man: **foreign key attribute ID# is not a part of the primary key** of Man
- Therefore the **line is dashed**

# Dashed Or Solid Lines



- To store Son we use a **foreign key**
- Man **is not completely identified** by its own original attributes: in the table SSN and ID# together serve as the primary key
- Relationship Son **is needed** for identifying a Man: **foreign key attribute ID# is a part of the primary key** of Man
- Therefore the **line is solid**

## ***Additional Points***

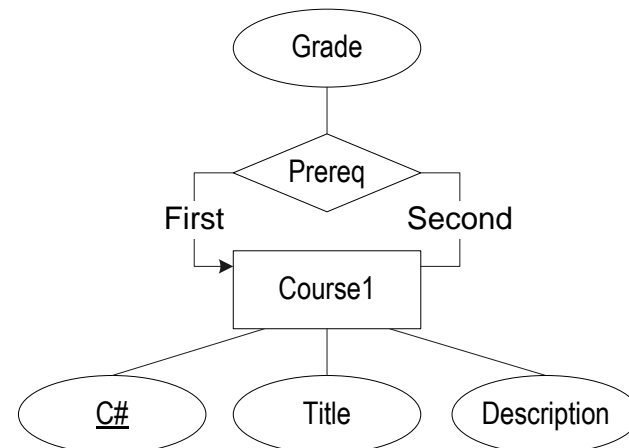
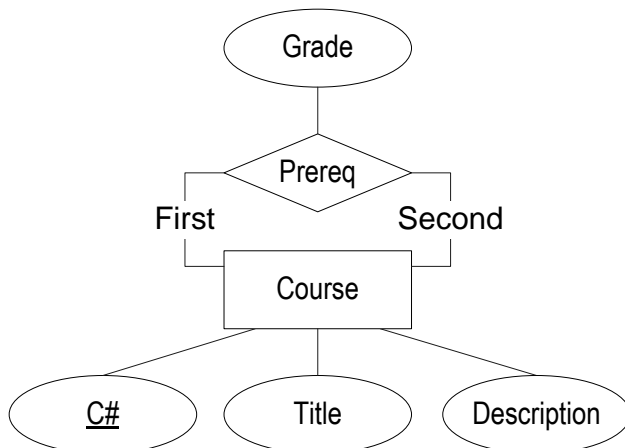
## ***Additional Points***

- We did not write out on the diagram various constraints that must be known, such as
  - At least preliminary domains, e.g., number, string, etc.
  - What is the maximum permitted section size
- ***This must be done for a proper documentation of the application's requirements***
  
- We will discuss some additional, important, points
  - Elaboration on recursive relationships
  - Referential Integrity
  - Temporal databases



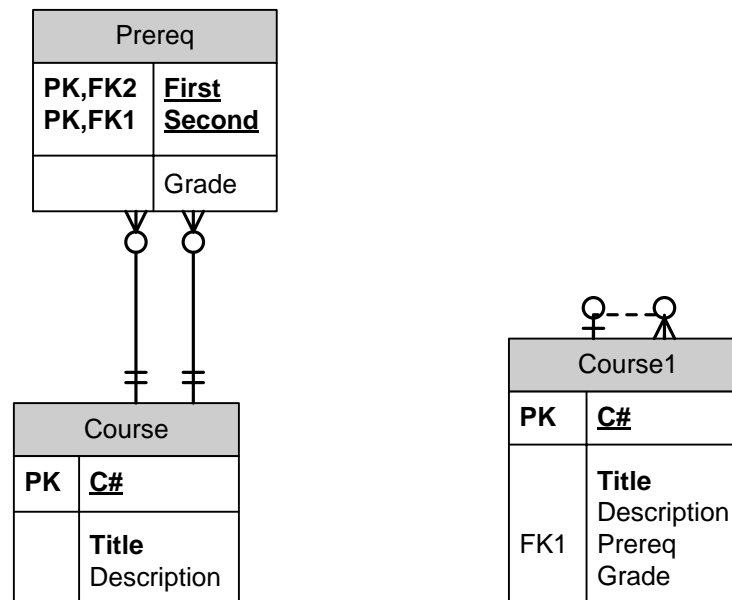
# ***Recursive Relationships: Example***

- Assume now that a prerequisite course, “First” course, must be taken with at least some Grade to count as a prerequisite
- This to make an example a little “richer”
- Two cases:
  1. A course may have any number of prerequisites:  
Prereq is binary many-to-many
  2. A course may have at most one prerequisite:  
Prereq is binary many to one (Second (a more advanced course) is the many side, a single First could be a prerequisite for many Second courses)



# Recursive Relationships: Example

- Nothing special, we handle the second case of Prereq by storing it in the “many” side of the relationship
- So there are two additional attributes in Course1
  - The prerequisite course, if any
  - The required grade in the prerequisite course, if any



# ***Temporal Databases***

- Of course, we may want to maintain historical data
- So, in practice one may have some indication that the professor no longer works, but still keep historical information about the past
- But we do not assume this for our example

# ***Referential Integrity***

## ***Referential Integrity: Example***

Professor	ID#	Salary
	5	1
	7	2

Taught	ID#	C#	Year	Semester	Sec#	Monitor
	5	G22.2433	2009	Spring	001	7

- Assume that we have some professors in table Professor, with rows: 5,1 and 7,2
- There is a row in Taught 5,G22.2433,2009,Spring,001,7
- This means that 5 teaches a specific section and 7 monitors this assignment

# Referential Integrity: Example

Professor	ID#	Salary
	5	1
	7	2

Taught	ID#	C#	Year	Semester	Sec#	Monitors
	5	G22.2433	2009	Spring	001	7

- A user accesses the database and **attempts to delete a row** (or all rows like this, recall that duplicates are permitted) 5,1 **from Professor**
- What should happen, as there is a row in Taught referencing this row in Professor?
- A user accesses the database and attempts to delete row 7,2 from Professor?
- What should happen, as there is a row in Taught referencing this row in Professor?

# ***Referential Integrity: Example***

- Part of specification of foreign key in in Taught
- An action on Professor can be denied, or can trigger an action on Taught
- For example
  - ON DELETE NO ACTION

This means that the “needed” row in Professor cannot be deleted

Of course, it is possible to delete the row from Taught and then from the Professor (if no other row in in any table in the database “needs” the row in Professor)
  - ON DELETE CASCADE

This means that if the a row is deleted from Professor, all the rows in Taught referring to it are deleted too
  - ON DELETE SET NULL

This means, that the value referring to no-longer-existing professor is replaced by NULL

In our example, this is not possible for ID# as it is a part of the primary key of Taught, but is possible for Monitor

## ***Referential Integrity: Another Example***

- Part of specification of foreign key in in Professor
- An action on Person can be denied, or can trigger an action on Professor
- For example
  - ON UPDATE CASCADE  
This means that if the value of ID# in Person is changed, this value of ID# also propagates to Professor
- Could (and probably should) add to Taught and Required:
  - ON UPDATE CASCADE  
In appropriate attributes, so that the change of ID# in Professor also propagates to them  
In Taught in both ID# and Monitor  
In Required in ID#
- Excellent mechanism for centralized maintenance



# ***From ER Diagrams to Relational Database Summary of the Process***

# ***Summary of the Process***

- The following will be reiteration of some points
- This is not complete, but you are now able to apply your understanding to more complex cases that we have not covered
  - For example, a multivalued attribute that is composite, such as a person has a number of addresses, each consisting of country, state, city, street, street number

## ***Summary: Strong Entity***

- Example: **Person**
- Create a table for the entity without multivalued and derived attributes, flattening composite attributes

The primary key of this table will consist of the attributes serving as primary key of the entity

Example table: Person

- If there is a derived attribute, describe how it is computed, but do not store it
- If there is a multivalued attribute, create a table for it consisting of it and attributes of the primary key of the entity; do not put it in the table for the entity

Example table: Child

The primary key of this table will consist of all its attributes

## ***Summary: Strong Entity***

- There could be an attribute that is composite with some components being multivalued and some derived
- And similar complexities
- Example, without drawing the appropriate entity using the ER model (this is getting too hairy)
  - A person has many children (multivalued)
  - Each child has both FirstName and MiddleName
  - The child has DOB
  - The child has Age
- Then the table for child will look like

Child	ID#	FirstName	MiddleName	DOB
	5432	Krishna	Satya	2006-11-05

## ***Summary: ISA and a Subclass***

- Example: **ISA** and **Professor**
  - Do not produce anything for ISA
  - The class “above” ISA (here Person) has already been implemented as a table
  - Create a table with all the attributes of the subclass (as for strong entity above) augmented with the primary key of the table “above” ISA, and no other attributes from it
- The primary key is the same as the primary key of the table “above” ISA

Example table: Professor

## ***Summary: Weak Entity and Defining Relationship***

- Example: **Offered** and **Section**
- Do not produce anything for the defining relationship, here Offered
- Imagine that the weak entity is augmented by the primary key of the “stronger” table through which it is defined (the table for it has been created already)

Treat the augmented weak entity the same way as a strong entity

The primary key is the primary key of the “stronger” table augmented by the attributes in the discriminant of the weak entity (a discriminant may consist of more than one attribute)

Example table: Section and Offered

## ***Summary: A Relationship That Is Not Many-To-One***

- Example **Took**

The tables for the participating entities have already been created

Create a table consisting of the primary keys of the participating tables and the attributes of the relationship itself

Of course, treat attributes of the relationship that are derived, multivalued, or composite, appropriately, not storing them, producing additional tables, flattening them

The primary key consists of generally all the attributes of the primary keys of the participating tables

Example table: Took

## ***Summary: A Relationship That Is Many-To-One and Non-Binary***

- Example ***Taught (not in University example, but the later one)***

The tables for the participating entities have already been created

Create a table consisting of the primary keys of the participating tables and the attributes of the relationship itself

Of course, treat attributes of the relationship that are derived, multivalued, or composite, appropriately, not storing them, producing additional tables, flattening them

The primary key consists of a ***subset*** of all the attributes of the primary keys of the participating tables

Annotations may need to be added

Example table: Taught



## ***Summary: A Relationship That Is Binary Many-To-One***

- Example: **Has**

Do not create a table for this relationship

Put the attributes of the primary key of the “one” side and the attributes of the relationship itself into the table of the “many” side

Of course, treat attributes of the relation that are derived, multivalued, or composite, appropriately, not storing them, producing additional tables, flattening them, as the case may be

You may decide to treat such a relationship the way you treat a relationship that is not binary many to one (but not in our class)

If the relationship is one-to-one, choose which side to treat as if it were “many”

Example table: Has

## ***Summary: Treating A Relationship As an Entity***

- Example: **Taught** (before it was modified by adding Monitor)

We have a table for that was created when we treated it as a relationship

We do not need to do anything else to this table

Example table: Taught

# ***Surrogate Primary Keys***

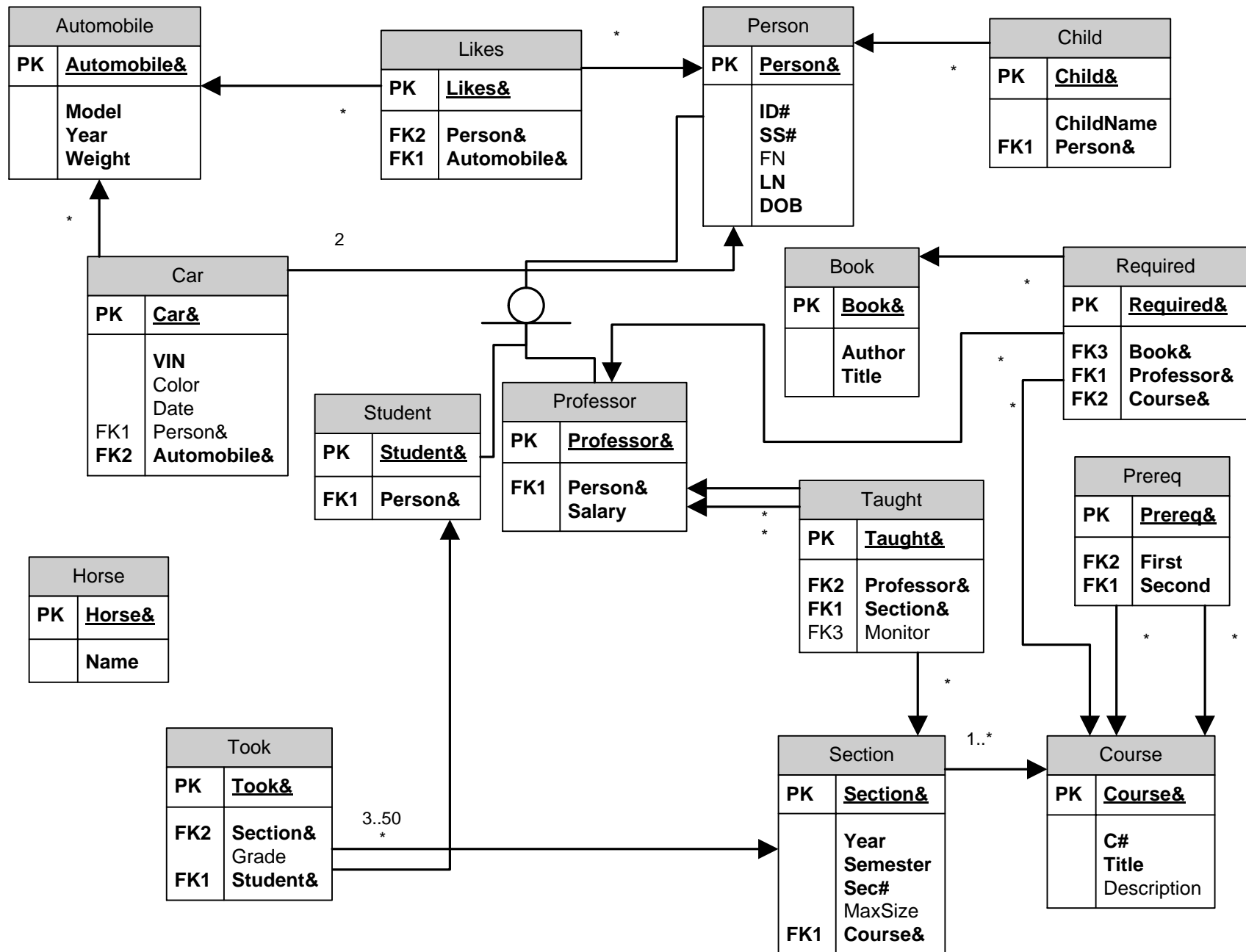
# ***Surrogate Primary Keys Replacing Former “Semantic/Natural” Primary Keys***

- Frequently it is desirable that the primary keys not be interesting by themselves but serve as internal identifiers of tuples in a table
- It is then less likely that the real world will impose legal or practical constraints on their use
  - For example, NYU used Social Security Numbers as identifiers, but a file of them was hacked
  - Also, in future, maybe it will be illegal to use Social Security Numbers for any purpose other than Social Security taxes
  - Maybe the manufacturer of Automobile will change retroactively the value of Model (name change to enhance sales)

# ***Surrogate Primary Keys Replacing Former “Semantic/Natural” Primary Keys***

- We can redesign our application so that the primary keys are all surrogate and of one attribute only
- A very obvious way is to assign serial numbers to entities in an entity set (and in other places too)
  - We discuss this in the unit dealing with SQL
  - And, very loosely speaking in that Unit, as an entity set becomes a table in a relational implementation with each entity in its own row, make the (surrogate) key of the 1<sup>st</sup> row to be 1, of the 2<sup>nd</sup> row to be 2, etc.
- There are advantages to indexing in the presence of surrogate keys
  - We will discuss that in the Unit dealing with physical design
- Surrogate attributes are denoted in the following example by ending with “&”
- ***Do not use surrogate keys in homework or exams***
  - They are important in practice but obscure some semantics

# Our Application with Surrogate Primary Keys



## ***Important: Former Primary Keys Must Be Made UNIQUE and NOT NULL***

- Note that some of the former primary keys are now surrogates
  - For example, former primary key of Professor (ID#) is now replaced by a surrogate (Person&)
- Note that some of the former primary key are now partially surrogates
  - For example, former primary key of Child (ID#,Childname) is now replaced by (Person&,Childname)
- Note that sometimes we do something else, but it is intuitively clear what needs to be done
  - ***But we need to tell the database that former primary keys are UNIQUE, as, e.g., no two Person& could have the same ID# and they cannot be NULL***

## ***Important: Former Primary Keys Must Be Made UNIQUE and NOT NULL***

- Horse: UNIQUE(Name), Name NOT NULL
- Person: UNIQUE(ID#), ID# NOT NULL, UNIQUE(SS#), SS# NOT NULL
- Automobile: UNIQUE(Model,Year), Model NOT NULL, Year NOT NULL
- Car: UNIQUE(VIN), VIN NOT NULL
- Book: UNIQUE(Author,Title), Author NOT NULL, Title NOT NULL
- Course: UNIQUE(C#), C# NOT NULL



## ***Important: Former Primary Keys Must Be Made UNIQUE and NOT NULL***

- Prereq: UNIQUE(First,Second), First NOT NULL, Second NOT NULL
- Required: UNIQUE(Book&,Professor&,Course&), Book& NOT NULL, Professor& NOT NULL, Course& NOT NULL
- Section: UNIQUE(Year,Semester,Sec#,Course&), Year NOT NULL, Semester, NOT NULL, Sec# NOT NULL, Course& NOT NULL
- Took: UNIQUE(Section&,Student&), Section& NOT NULL, Student& NOT NULL
- Taught: UNIQUE(Professor&,Section&), Professor& NOT NULL, Section& NOT NULL

# ***Caveat***

# ***Caveat***

- We have gone through a pretty complex example and examined important cases of transforming an ER diagram into a relational implementation
- However, there is no complete set of “recipes” for doing it automatically
- Just do a reasonable thing and be ready to justify it (to yourself and to your customer)

# ***Annotate, Annotate, Annotate ...***

- **A relational schema specification should be annotated with all known constraints**
- Annotations are needed so that whoever converts a relational schema specification + annotations into an SQL DDL + any added necessary constraints can account for all the constraints imposed on the application
- You do not need to put in annotations any constraints that are reflected in the schema specification, using a tool such as SQL Power Architect
- ***In the homework and exams, you must not put such constraints (that is constraints implicit in what SQL Power Architect can do) in annotations just to make sure that you realize which constraints are already reflected in SQL Power Architect specification***

# ***Key Ideas***

# ***Key Ideas***

- Sets
- Relations and tables
- Relational schema
- Primary keys
- Implementing an ER diagram as a relational schema (relational database)
- General implementation of strong entities
- Handling attributes of different types
- General implementation of relationships
- Possible special implementation of binary many-to-one relationships
- Implementation of ISA
- Implementation of weak entities

# ***Key Ideas***

- Foreign keys
- Primary key / foreign key constraints inducing many-to-one relationships between tables
- Concept of referential integrity
- SQL Power Architect
- Crow's feet notation: ends of lines
- Crow's feet notation: pattern of lines
- Non-binary many-to-one relationships
- Surrogate keys