

GET NORMAL



Copyright 1995 Craig Edward Given
Second Edition 2017

TABLE OF CONTENTS

Course Overview	3
What is Normalization?	3
Why Normalize?	3
About This Tutorial	3
What will I learn from this tutorial?	3
How Long Will the Tutorial Take?	3
Example Scenario	4
What is a “Relational” Database?	5
Where Did Relational Databases Come From?	5
Primary Attributes of a Relational Database	5
Some Relational Operators	6
Make it Relational	7
Identify Entities	7
Domain Decisions	7
What’s Still Wrong?	8
The First Normal Form (1NF): Eliminate Repetition	9
Eliminate Repetition	9
Entity Relationship Diagrams (ERDs)	10
An Example ERD	10
ERD Notations	11
Interpreting Logical ERDs	12
Interpreting Physical ERDs	13
Interpreting Hybrid ERDs	14
The Second Normal Form (2NF): Eliminate Redundancy	15
Eliminate Redundancy	15
ERD for 2NF	16
DD for 2NF	16
TRICK	17
The Third Normal Form (3NF): Eliminate Transitive Dependencies	18
Eliminate Transitive Dependencies	18
ERD for the 3NF	19
Weaknesses of the 3NF	20
I/O Overhead	20
40 Years Old	20
Structured	20
SQL vs. 4GL	20
Referential Integrity	21
Prevent Orphans by Killing Children before Parents	21

Record Locking	22
Stored Procedures	22
Triggers.....	23
Hands-On Exercises	24
Instructions	24
Exercise 1	24
Exercise 2	24
Exercise 3	24
Exercise 4	24
Exercise 5	25
Solutions	26
Exercise 1: Accreditation Ranking	26
Exercise 2: Current vs. Charged Price	26
Exercise 3: Kennel / Trick Correlation	26
Exercise 4: Track Employee Trainers	27
Exercise 5: Track Employee Job Titles	27
Final Brain Buster.....	28
The Problem	28
The Solution	29
Bibliography	30

COURSE OVERVIEW

WHAT IS NORMALIZATION?

Normalization is a technique, or discipline, applied to the design of a database. If you are designing a database, you should normalize your databases to the third normal form. If you use normalized databases, understanding how they are structured can help you query or report against that database.

WHY NORMALIZE?

- Normalization offers many benefits for developers and users:
 - **Flexibility:** Normalization allows complex reporting and queries. If you can get the data into the system correctly, then getting it back out is not a problem. You will find business activities change more than business information. Storing the data flexibly facilitates a fluid business environment.
 - **Expandability:** Adding new fields to a normalized database doesn't create a lot of problems.
 - **Integrity:** Normalization allows you to maintain a greater degree of data consistency and prevent certain cases of data loss.
 - **Efficiency:** Normalization eliminates redundancies which can lead to storage and processing benefits.
 - **Performance:** The fastest speeds are obtained by planning. The initial design will impact performance more than remedial tuning, so make sure your blueprint is correct up front.
- RDMSs¹ are the de facto standard for databases. There are many vendors, many products, and multiple platforms running relational databases. Therefore, if you are building or modifying a relational database, it is important that your work be normalized. Pre-built database designs (e.g., PeopleSoft applications, Oracle applications, Kronos, etc.) are delivered normalized, so power-users and administrators for these should be well versed in normalization.
- Most relational databases now offer standardized access via SQL² which allows programmers access to a larger array of database products, portability between products, lower training costs, access to a broader range of tools, and is a marketable job skill. As a higher-level language, SQL offers more power for less code.

ABOUT THIS TUTORIAL

WHAT WILL I LEARN FROM THIS TUTORIAL?

- **Terms:** We will define the terms 3NF (third normal form), ER (entity relationship), ERD (entity relationship diagram), DD (data dictionary), and Triggers.
- **Techniques:** You will learn to read an ERD, create an ERD, and migrate a sample, unnormalized database, to the third normal form. This tutorial focuses on the practical aspects of normalization. Therefore, it doesn't discuss the (primarily) theoretical fourth or fifth normal forms. Also, we ignore the mathematical theory upon which relational databases were designed, and instead focus on what vendors have implemented.

HOW LONG WILL THE TUTORIAL TAKE?

When this material was originally taught in a classroom setting, the course took approximately two hours. This included two breaks, hands-on exercises, and a Q&A session. The audiences also included a good proportion of students without any technical background.

¹ Relational Database Management System, commonly called a "relational database." Relational databases are not necessarily normalized, but must be designed that way by their designers

² Structured Query Language is a non-procedural language used to retrieve, store, or modify database information. The example SQL statements in this tutorial are the Oracle "flavor."

EXAMPLE SCENARIO

An imaginary kennel has an unnormalized database. We examine their situation and current data layout.



Welcome to **Teacher's Pet, Inc.!** You've been drafted to serve as the RDBMS design staff.

Before I show you our data files, let me tell you about our company. Teacher's Pet, Inc. (TPI) offers *highly* specialized dog training. Have you ever seen the painting of dogs playing poker? Well, we trained those dogs! When owners of a TPI-trained dog say "Speak!" our dogs enunciate. When they say "Beg!" our dogs hold up cardboard signs reading "Will work for food." When they say "Sit!" our dogs take care of the baby while they go out for dinner.

TPI is a prestigious, exclusive, expensive, and (because we're so expensive) a dying company. Therefore, our marketing department—his name is Herb—has created a nifty ad campaign to boost business. In anticipation of the huge customer increase, the boss wants everything on computer. Here's how our data is currently being tracked on paper:

Dog Number:	3	Dog Number:	12	Dog Number:	13
Name:	Tuck Yang	Name:	Nip Yin	Name:	Shadow Muttley
Sex:	Male	Sex:	Male	Sex:	N/A
Kennel:	Ruff Haus	Kennel:	K9HQ	Kennel:	K9HQ
Address:	1200 Frank St. Chattanooga, TN 37411	Address:	4051 Lewis St. Chattanooga, TN 37412	Address:	4051 Lewis St. Chattanooga, TN 37412
Phone:	423-820-2412	Phone:	423-833-4122	Phone:	615-833-4122
Tricks:	Rollover 53	Tricks:	Roll Over 40	Tricks:	Sit 10%
	Retrieve 71		Play Dead 70		Stay 10%
	Tightrope 91		Fetch 50		Heel 12%

These 3x5 index cards, by the way, represent the sum total of our current clientele. You can see why we need the new marketing campaign. You can also see some of the problems we've had with data integrity. When our area code changed from 615 to 423, we missed Shadow's record during the update. You can also see that Nip and Tuck both know the same trick, but it's called "fetch" by one trainer, and "retrieve" by another. Also, the "Roll Over" trick was once mistakenly recorded as "Rollover." I'm also uncertain if the skill levels are recorded consistently. That is, does Nip play dead at 70% or is it 70 out a possible 300?

The salesman told the boss that a "Relational Database" would solve our problems. That's why we brought you in. Can you migrate this data to an RDMS? By the way, exactly what is a "relational" database?

WHAT IS A “RELATIONAL” DATABASE?

WHERE DID RELATIONAL DATABASES COME FROM?

- The concept of Relational Databases was formulated in 1969 by E.F. Codd of IBM
- The term “relational” comes from the mathematical studies of “relational algebra” and “relational calculus” that were used to manipulate this type of database model. Fortunately, we don’t have to know algebra nor calculus to use a relational database.
- Codd specified **333** criteria for a relational database. Of these, 12 are considered the core criteria. We'll discuss a handful of these core criteria in a minute.
- As of this writing, there are **NO** database products that qualify fully as a relational database (i.e., that meet all 333 of Codd’s criteria).

PRIMARY ATTRIBUTES OF A RELATIONAL DATABASE

- A relational database is a collection of two dimensional tables. Think of the horizontal dimension as rows and the vertical dimension as columns. You can also think of them as records and fields. Each table has exactly four sides: that is, each row in a table has the exact same number of columns (i.e., every record in a table has the exact same fields).
- Each table’s attributes (more commonly known as columns or fields) must all be of the same data type. For example, in our scenario, each dog can be uniquely identified by a dog number. Therefore, our table should have a field for tracking the dog number, which can only contain numbers. We can’t mix a numeric dog number with alphabetic dog codes. This can be a confusing concept at first. For example, a social security number, while it has digits is really a code. But a price is always a numeric value. You can’t have a relational database where the price is stored as 89.95 in one record, but “a whole lot” in another. They must be the same data type.
- No two tuples (more commonly known as rows or records) are identical. Why would you want duplicate rows anyway? That would be a waste. Besides, when we normalize a relational database, one step is to eliminate wasteful duplication.
- The order of the rows is not important. This rule allows us to use indices to sort the data in multiple ways. With a relational database we can insert new rows at the beginning, in the middle, or at the end. Also, when data is deleted in a relational database, it doesn’t require any kind of reorganization.
- Each attribute must have a domain. This rule states that each column has specified limits. For example, a dog number might have a limit of three digits. Its domain would be any integer between zero and 999.

SOME RELATIONAL OPERATORS

Do you remember “sets” from elementary school? Tables in a relational database are data sets, so you can perform relational operations on them. Here are a few of the relational operators which can be performed on a relational database:



$A \cup B$

UNION: data sets combined.



$A \cap B$

INTERSECTION: only the data shared between sets.



$A - B$

DIFFERENCE: only data not shared with another set.



$A \Delta B$

SYMMETRIC DIFFERENCE: only data not shared with all sets.

MAKE IT RELATIONAL

IDENTIFY ENTITIES

If we take our 3x5 index cards, and put all the data in a tabular format, this is basically what it would look like:

Dog	Name	Sex	Kennel	Address	Phone	Trick #1	Trick #2	Trick #3
3	Tuck Yang	Male	Ruff Haus	1200 Frank St., Chattanooga, TN 37411	423-820-2412	Rollover 53	Retrieve 71	Tightrope 91
12	Nip Yin	Male	K9HQ	4051 Lewis St., Chattanooga, TN 37412	423-833-4122	Roll Over 40	Play Dead 70	Fetch 50
13	Shadow Muttley	N/A	K9HQ	4051 Lewis St., Chattanooga, TN 37412	615-833-4122	Sit 10%	Stay 10%	Heel 12%

There are several problems with this data. First, we haven't accurately identified each piece of data in enough detail. For example, "Name" needs to be broken down into "First Name" and "Last Name." Also, the address needs to be broken down into street address, city, state, and ZIP code. Also, the tricks need to be separated into trick name and skill level. Identifying each entity allows for better queries in the future (e.g., when we need to send targeted advertising to just certain states, or certain ZIP codes).

One of the rules of a relational database is that attributes (columns) must have the same data type. Therefore, skill levels will be numeric percentages only (but with percent signs removed). Here is the same data in a relational database's table, but still unnormalized:

Dog	First Name	Last Name	Sex	Kennel	Street	City	State	ZIP	Phone	Trick Name #1	Trick #1 Skill	Trick Name #2	Trick #2 Skill	Trick #3	Trick #3 Skill
3	Tuck	Yang	M	Ruff Haus	1200 Frank St.	Chattanooga	TN	37411	423-820-2412	Rollover	53	Retrieve	71	Tightrope	91
12	Nip	Yin	M	K9HQ	4051 Lewis St.	Chattanooga	TN	37412	423-833-4122	Roll Over	40	Play Dead	70	Fetch	50
13	Shadow	Muttley	U	K9HQ	4051 Lewis St.	Chattanooga	TN	37412	615-833-4122	Sit	10	Stay	10	Heel	12

DOMAIN DECISIONS

We had to make some non-technical decisions, which in the corporate world might be called "redesigning our business processes." In our scenario, we decided that all skill levels would be recorded as percentages, so 70 means 70 out of 100 possible points. We also made the decision that percentages will not include any decimal points — integers only. That also meets the relational requirement that an attribute have a domain. In this case, the domain for a skill level is an integer between 0 and 100. You'll also notice that we conserved space by using a single-letter code for sex: M=Male, F=Female, and U=Unknown. We can programmatically enforce these domains, either in the applications or through triggers (more on triggers in a following page).

From here on, I'll refer to columns as fields, and rows as records. As you can see, the order of the records doesn't change the data, there are no duplicate records, each record has the same number of fields (making it a two-dimensional table), any given field is of the same data type, and all the fields have a domain. This last characteristic isn't obvious, but when working with a database tool, you would define a field's domain. For example, you would set "First Name" to accept characters, with a maximum length of 30. By setting this field to be required (often called a "Not Null" constraint) we enforce a domain of "at least one character, but no more than 30 characters."

WHAT'S STILL WRONG?

Our data may be stored in a relational database, but there are some serious flaws in this table. Let's identify the problems before we solve them using normalization.

- First, if we ever delete Tuck's record, we lose the address for Ruff Haus. We want to track all area kennels that we've worked with before, even if they don't have dogs enrolled in TPI. All those kennels are a major thrust for our advertising campaign.
- Second, we've only allowed for three tricks. What happens when a dog learns four tricks? A table change would require major rewriting to all reports and programs if we leave the data in this layout. That's a massive headache and a sure sign of sloppy design work. Normalization will fix the problem and give us all the flexibility we need, even if a dog learns a thousand tricks! Normalization will also fix the problem of wasted space (e.g., when a dog first enrolls and knows just one, or even zero tricks, we have blank fields with this current layout).
- Third, we still have the problem with data consistency. The trick "Roll Over" is spelled two different ways, and the trick "Fetch" is also listed as "Retrieve" even though they are the same trick.
- Fourth, we have a maintenance nightmare, especially if our database grows to thousands of records (which we want to happen). Specifically, if a kennel changes its phone number we must manually change every record where it occurs. In our scenario, one record was missed, and we've got bad data in our database.

THE FIRST NORMAL FORM (1NF): ELIMINATE REPETITION

ELIMINATE REPETITION

Normalizing a database begins by eliminating repetitions. A database that is free of repetitions is said to be in the first normal form (or 1NF). Our sample database, on the previous page, has repetitions: the tricks are repeated three times.

If we break the data into two separate tables, we eliminate the repetition, and we fix several flaws. With two tables we can now store as few or as many tricks as we want (storage space permitting). Also, there is no wasted space because we only insert a record into the tricks table as necessary.

Here are the two tables in the first normal form:

DOG Table

Dog	First Name	Last Name	Sex	Kennel	Street	City	State	ZIP	Phone
3	Tuck	Yang	M	Ruff Haus	1200 Frank St.	Chattanooga	TN	37411	423-820-2412
12	Nip	Yin	M	K9HQ	4051 Lewis St.	Chattanooga	TN	37412	423-833-4122
13	Shadow	Muttley	U	K9HQ	4051 Lewis St.	Chattanooga	TN	37412	615-833-4122

SKILL Table

Dog	Trick	Skill
3	Rollover	53
3	Retrieve	71
3	Tightrope	91
12	Roll Over	40
12	Play Dead	70
12	Fetch	50
13	Sit	10
13	Stay	10
13	Heel	12

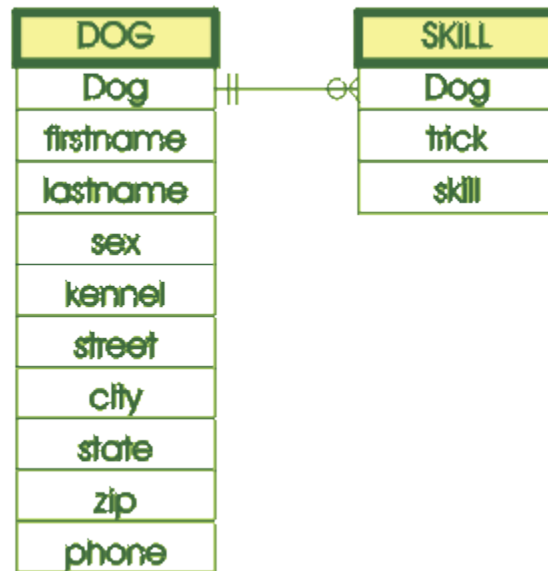
We get an efficiency benefit from the first normal form. When this database grows to a larger size we can easily query the database to determine which dogs know a particular trick. Not only will the queries be easier to write, they will also run faster! In an unnormalized database, we're basically running three queries! The queries below are written in SQL, which you aren't expected to know at this point, but they should be fairly easy to read.

Querying an unnormalized Table	Querying a 1NF Table
<pre>SELECT dog FROM dog WHERE trick1 = 'Sit' OR trick2 = 'Sit' OR trick3 = 'Sit';</pre>	<pre>SELECT dog FROM skill WHERE trick = 'Sit';</pre>

ENTITY RELATIONSHIP DIAGRAMS (ERDs)

AN EXAMPLE ERD

We've been using tables with columns and rows to represent our databases. That's fine when we only have three dogs. But when a database has millions of records, we need a better (smaller) way to show the relationships between the entities in each table. Such a tool exists, and it's called an ERD (Entity Relationship Diagram). Below is an ERD of our database as it is structured so far (1NF):



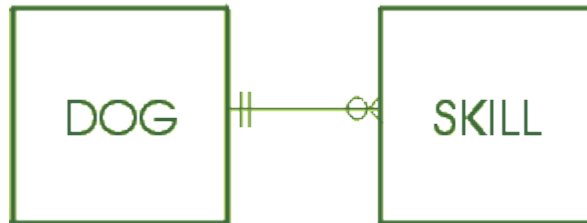
This ERD shows two tables: DOG and SKILL. Tables should be named in the singular (e.g., not “DOGS” and not “SKILLS,” but your employer may have a different standard). The top box is used to show the name of each table, and then the fields in the table are listed below that. You will also notice a line drawn between the two tables, which shows a relationship. On the left side of this line, there are two vertical slashes which means “must exist.” On the right side of this line is a circle and a three-pronged trident, which means “zero or more.” When you put it all together, it reads “One dog can have zero or more skills” with the understanding that the dog must exist before it can have any skills. This type of set up is known as a “one-to-many relationship.”

One way to view an ERD is to see the tables as the “nouns” of your data: person, places, and things. Attributes (columns or fields) are the “adjectives” for those nouns, because they describe (a person’s name, a person’s age, a person’s address, etc.). The relationships between entities are the “verbs.”

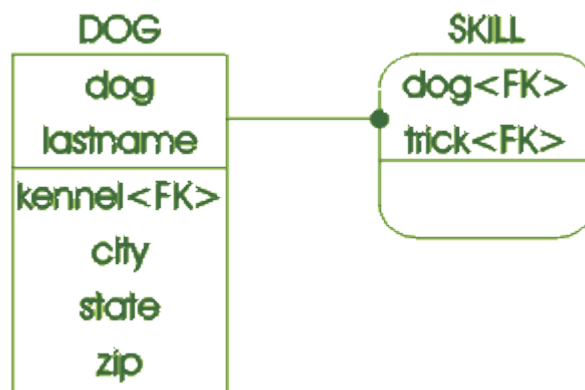
ERD NOTATIONS

Entity relationship diagrams come in various flavors, and different vendors and scholars³ have invented their own notation systems. The notation used in the ERD above is a hybrid cross between two very common notational systems (one for the logical, and one for the physical). A logical ERD is a high-level view that focuses on just the table relationships, and doesn't include details about fields. On the other hand, the physical ERD includes information on the fields in a table, and shows how they work as keys to connect one table to the next. Here is how our example database would look in each diagram type:

LOGICAL ERD



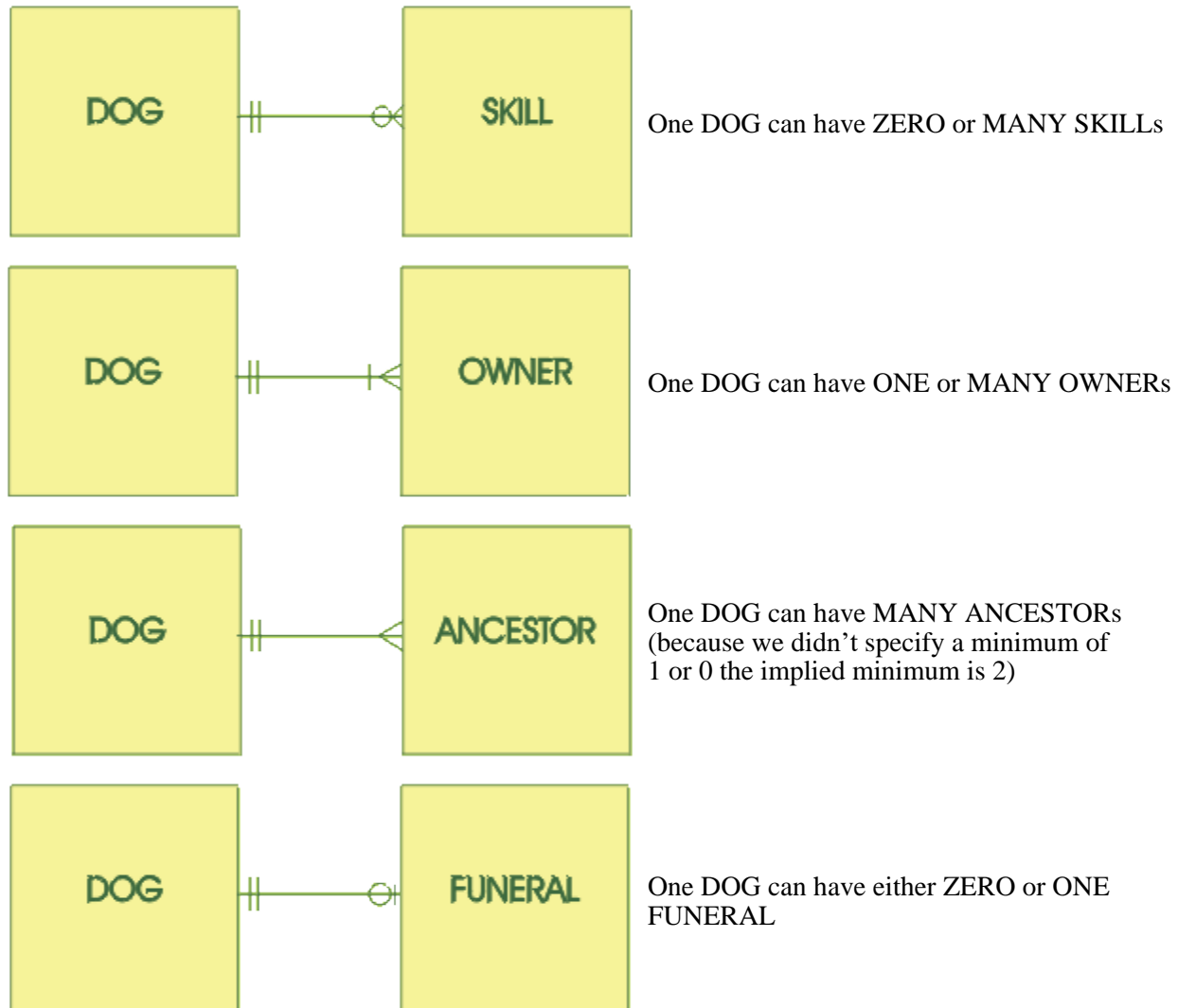
PHYSICAL ERD



³ Examples include Chen, Bachman, Martin, IE, Merise, Shlaer/Mellor, IDEF1X, and proprietary CASE notations. PeopleSoft, for example uses EasyCASE to produce some of their ERDs

INTERPRETING LOGICAL ERDs

When interpreting a logical ERD, double vertical bars on the connecting line mean that entity “must exist” before the relationship can be established. In all the examples below, a dog must exist before it can have a skill, owner, funeral, etc. The symbols on the end of the lines are generally a combination of one or two symbols. The first symbol represents the minimum, and the second (or only symbol in some cases) represents the maximum. The symbols are a circle (means “zero”), single vertical slash (means “one”), and a three-pronged triton (means “many”). Here are some examples and how they are read:



Try your hand at it now, with these examples:

- An employee can be single, but if married they can have only one spouse.
- An employee works in at least one department, but they could work in several.
- Our network connects many different computers together. Obviously, you need at least two computers, or it wouldn't be a network, would it?
- An employee may not be eligible to attend a service awards banquet, but if they've been here long enough, they could have possibly attended several.

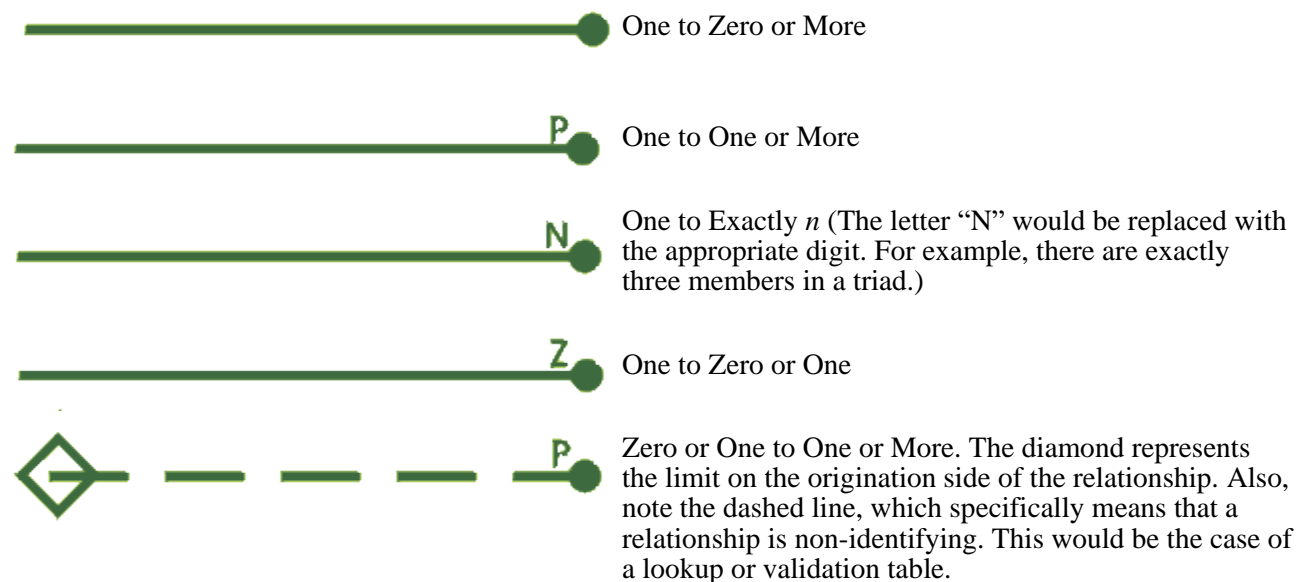
These statements could easily have come from a design session, where a database designer interviewed a user. By asking specific questions he can identify the entities to be tracked, and how those entities relate to each other. Those complex relationships can be visualized in an ERD produced from such an interview.

INTERPRETING PHYSICAL ERDS

Logical ERDs are great for overviews and planning, but eventually you'll have to deal with the details. The physical ERD gives that kind of detail. Specifically, it shows the keys (index keys) that are used to link two tables in a relationship.

In a physical ERD, the shape of a box is significant. Boxes with square corners are considered independent entities. Boxes with curved corners are dependent. You will also note that a box is divided into two sections. Fields listed above the horizontal divider are primary keys. Fields below the line are secondary keys. In a dependent entity, you'll also see the suffix <FK> after a field name. This stands for "Foreign Key." A foreign key is simply a link to a Primary Key in another table (i.e., the link from a child record to a parent record). The primary key in a table must be unique, but that key may be duplicated many times in the dependent (child) table as a foreign key. For example, Nip is dog number 12. There will be only one dog #12 in the DOG table, but there can be many records in the SKILL table with dog #12 entries. That's because a single dog can have many skills.

Here are the meanings for the different connecting lines in a physical ERD:



INTERPRETING HYBRID ERDS

This is the notation we'll use for this tutorial. It borrows the best from several different notation systems, including the physical and logical notations already shown, so that we can more accurately diagram a wider selection of relationships. In the section “An Example ERD” (at the top of this page), I show the hybrid notation in use: The boldly outlined box at the top of each column contains the table name in all capital letters. The fields in each table are listed below the table name. Relationship lines are drawn between the linking fields, and not just between table names. Therefore, the ERD at the top of the page shows that DOG.DOG⁴ must pre-exist, and may have zero or more occurrences in SKILL.DOG.


Read the lines below from left to right. The different symbols are: vertical slash (one), circle (zero), three-pronged triton (many), double-vertical slash (must pre-exist), and solid circles (mutually exclusive). The dashed lines have a house-shaped arrow head (initialized from, or copied from) while the dotted lines have a triangular arrow head (conditional). We use two different arrow head shapes just to make it easier to distinguish dotted and dashed lines.


 One to One


 One to One or More


 One to Zero or One


 One to Zero or More

 One to Many (2 minimum, because neither 1 or 0 is “many”)

 One (must pre-exist) to Zero or More

 source is copied to destination (non-identifying relationship)

 Conditional relationship exists. E.g., an INVENTORY table has a COMPANY field which is conditionally interpreted by COMPANY.TYPE (where type might be “Distributor” or “Manufacturer”). We would draw a relationship from COMPANY.COMPANY to INVENTORY.COMPANY, and a conditional line from COMPANY.TYPE to INVENTORY.COMPANY.

 One (must pre-exist) to Zero or More. However, the vertical slash before the line divides means you must choose one path: there is a “fork in the road.” The solid circles after the fork mean “mutually exclusive.” That is, take the top fork or take the bottom fork, but it can’t be both simultaneously.

⁴ I’m using a dot notation here, where I reference tables and fields in the format of TABLE.FIELD

THE SECOND NORMAL FORM (2NF): ELIMINATE REDUNDANCY

ELIMINATE REDUNDANCY

Our SKILL table still has flaws. Here it is again for reference, in the 1NF:

SKILL Table

Dog	Trick	Skill
3	Rollover	53
3	Retrieve	71
3	Tightrope	91
12	Roll Over	40
12	Play Dead	70
12	Fetch	50
13	Sit	10
13	Stay	10
13	Heel	12

We want to eliminate redundancy by using shorter codes for the trick names. Codes save storage space, smaller files process faster, and short codes are easier for users to enter⁵ into applications. This is especially true if the descriptions were a lot longer than our example data. Then the savings and performance gains could be quite significant.

By storing codes, we also gain an integrity benefit. You don't have to worry about variations in spelling, capitalization, or the use of synonyms. You'll want to limit who can manage the list of trick definitions, so that consistency is enforced. This is because consistency is especially important when queries are made (e.g., "Give a list of all dogs who know the trick **Fetch**") could return invalid results in the 1NF because it would miss dogs who know that trick as **Retrieve**.)

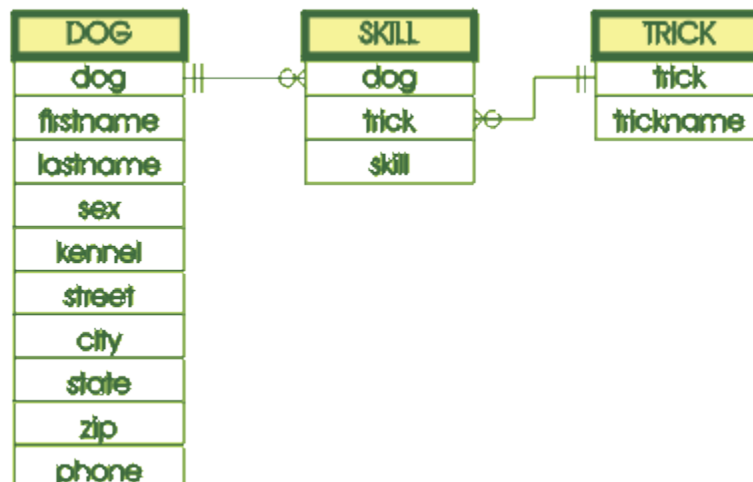
⁵A well-designed application should allow users to enter both codes or select from a list of full descriptions. Expert users won't be slowed down, and novice users can learn the codes at their own pace. Even expert users benefit from full descriptions because there will be some codes that they rarely use. If the list of codes is long, you'll also want to make the list searchable

When we move the redundant data to a different table, here is how the data would look:

SKILL Table			TRICK Table	
Dog	Trick	Skill	Trick	Name
3	1	53	1	Roll Over
3	3	71	2	Play Dead
3	7	91	3	Retrieve
12	1	40	4	Sit
12	2	70	5	Stay
12	3	50	6	Heel
13	4	10	7	Tightrope
13	5	10		
13	6	12		

ERD FOR 2NF

Below is the ERD for our two new tables in the second normal form (the DOG table isn't in the 2NF, but it will be fixed when we bring that table to the 3NF). You can see that the SKILL table is the "many" side for two relationships. That is, one dog can learn zero or more skills, and that one trick can occur zero or more times in the skills table. The skills table is in (what is referred to as) a "Many-to-Many Relationship."



DD FOR 2NF

The 3NF is just a step away, but before we reach that destination, I want to talk about Data Dictionaries (abbreviated DD). Below is a DD entry for our TRICK table. A DD gives detailed information about each field in a record. This is where we document the data type, domain (limits such as size), and what role a field might play in a relationship between other tables. In our example, the notes for the TRICK field show how the trick number links the table TRICK with the table SKILL. The comments use the "dot" notation. With dot notation, the word before the period is the table, and the field follows the period. Since TRICK is a field in both the SKILL and TRICK tables, it helps identify which we're talking about. Take a look at the example below, and we'll discuss some finer points below.

TRICK

TRICK

Type: Numeric
Size: 5 (99999)
Index: Primary Key
Constraints: Required, Unique.
Notes: Internal trick number. There may be many SKILL.TRICK entries for a single TRICK.TRICK

TRICKNAME

Type: Character
Size: 40
Index: Secondary Key
Constraints: Required. Always Title Case.
Notes: Full description for the trick.

There's nothing magic about the layout of this example DD. It's just a document created by the database designer to record the decisions made during development. Any changes that follow implementation should also be recorded in your DD. You might want to use a different layout, record additional details, or your database tool might have a place for this information to be stored. Regardless of location, it's important to document these details so they won't be forgotten. I would also encourage you to document, not only the WHAT, but also the WHY things were set up a certain way.

Looking at our specific example, you can see that the TRICK field is the primary key used to link to the SKILL table. Earlier I spoke of using short, mnemonic codes, so why are we using numbers here? First, primary keys used to link must be unique. Second, to ensure that link integrity is enforced, professional database designers hide these links from the user (for both their benefit and for integrity's sake). Third, professional database designers prefer to use *arbitrary* keys, so maintenance is minimal. Numeric fields are used, by the clear majority of professional developers, as primary keys. That's because a new record can be added by simply adding one to the highest key that exists (e.g., the next trick in our sample table of seven tricks would be trick #8).

If an application is designed to use codes for expert user's efficiency, we could add a field named CODE to the TRICK table. The user could use CODE, but behind the scenes the TRICK field transparently links to the SKILL table. We don't sacrifice maintainability, either. For example, let's say we added a trick #8 that's named "Triple Reverse Somersault." We want to use the code "TR" but it is already used by "Tightrope." To solve our quandary, we'll make "Tightrope" code "T" and "Triple Reverse Somersault" as code "TR." This change only affects two records if we used arbitrary key. However, if we had used a code to link the tables, we'd have to replace ALL occurrences of "TR" with "T" first, and then all occurrences for "Triple Reverse Somersault" to "TR." If there were millions of records, such changes become a maintenance nightmare. Also, if the sequence isn't correct, you can accidentally corrupt a ton of data!

THE THIRD NORMAL FORM (3NF): ELIMINATE TRANSITIVE DEPENDENCIES

ELIMINATE TRANSITIVE DEPENDENCIES

We left our DOG table in the 1NF because I wanted to illustrate how to eliminate transitive dependencies (i.e., bring it to the third normal form). Here is our flawed table for reference:

DOG Table

Dog	First Name	Last Name	Sex	Kennel	Street	City	State	ZIP	Phone
3	Tuck	Yang	M	Ruff Haus	1200 Frank St.	Chattanooga	TN	37411	423-820-2412
12	Nip	Yin	M	K9HQ	4051 Lewis St.	Chattanooga	TN	37412	423-833-4122
13	Shadow	Muttley	U	K9HQ	4051 Lewis St.	Chattanooga	TN	37412	615-833-4122

The problem is with the kennel information. Specifically, if we delete Tuck's record, we lose all the information for the Ruff Haus kennel. In our scenario that's especially bad because we want to send advertising mailouts to all area kennels to increase our business. That certainly includes kennels for which we have no clients! In simple terms, the Ruff Haus information **DEPENDS** on Tuck's record: it's a "transitive dependency."

We eliminate transitive dependencies the same way we've solved the other problems: split it into another table. The 3NF also solves some 2NF problems as well. First, data consistency is maintained. The phone number for K9HQ, for example, only has to be maintained in one location. This demonstrates the second advantage: maintenance efficiency. That is, all dog records will automatically refer to the correct phone number. In a large database, this means one update instead of millions. Also, consider the maintenance impact if we added Fax Number for our kennel customers!

Third, we realize a tremendous space savings. Kennel information could be considerably more extensive. One copy isn't a burden to store, but if it were stored with every dog record, its impact is immense on a large database. An unnormalized database can dramatically increase your storage costs and impact performance. Now let's look at how DOG table is split to conform to the 3NF:

DOG Table

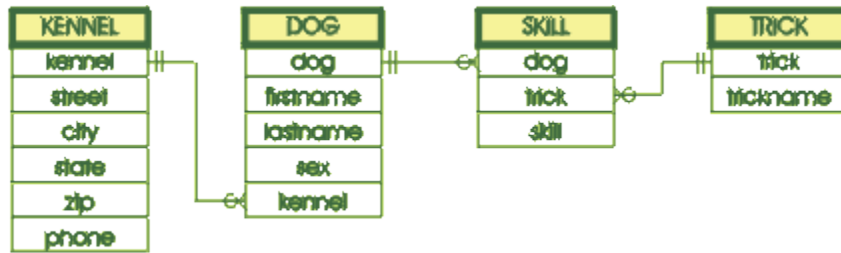
Dog	First Name	Last Name	Sex	Kennel
3	Tuck	Yang	M	1
12	Nip	Yin	M	2
13	Shadow	Muttley	U	2

KENNEL Table

Kennel	Name	Street	City	State	ZIP	Phone
1	Ruff Haus	1200 Frank St.	Chattanooga	TN	37411	423-820-2412
2	K9HQ	4051 Lewis St.	Chattanooga	TN	37412	423-833-4122

ERD FOR THE 3NF

Drum roll please! Here it is, our scenario database in the third normal form:



Now consider that this is a very simplistic database! An application such as PeopleSoft's HRMS/Payroll system is composed of thousands of tables. The ERD for such a system won't even fit on a large poster — it's more like an entire wall! But normalization is worth it. The 3NF gives us amazing query flexibility, an environment that promotes data integrity/consistency, and reduces storage requirements. If some ERDs seem complex, it is because the power of the 3NF can be used to represent extremely complex real-world relationships. Only the 3NF can model the complexity of something like a payroll system, and do it with both accuracy and efficiency. But there is a dark side to normalization, and that's our next topic.

WEAKNESSES OF THE 3NF

Nobody's perfect, and a normalized database is no exception. While the 3NF has weaknesses, very few people have ever switched from a relational database model because the payoffs vastly outweigh these inconveniences. But, in fairness, here are the major weak points of a normalized relational database:

I/O OVERHEAD

The input/output load can be greater for a normalized database than certain non-relational counterparts. Since normalization is primarily a divide-and-conquer strategy, the 3NF means more tables. Think of it as having to assemble your car every time you had to go to the grocery store. The cost for enormous query flexibility is that we must query more tables when assembling data. Normally the performance impact is offset by this power and flexibility. But, in a poorly designed implementation, there can be degradation. Sometimes a database designer will purposely leave certain tables unnormalized for efficiency of processing. This is especially true when the other table will have very few fields.

40 YEARS OLD

Since 1969, newer and more complex data models have evolved. Object Orientated Databases (OOD) and programming (OOP) have introduced the concepts of objects, abstraction, and inheritance. Also, the two-dimensional tables of the relational database aren't as well suited to handle multi-dimensional data or hierarchical data structures.

STRUCTURED

The two-dimensional structure of a relational database table is both a benefit and a liability. For organized and structured data, it's a boon, but for free-form data it's a bane. Other data models can handle certain data types, such as large volumes of text, more efficiently.

SQL vs. 4GL

Most relational databases use SQL as their native tongue. SQL is a non-procedural syntax that drives procedural programmers to distraction. It is also limited in some processing capabilities. Often a vendor provides a procedural tool to accompany SQL to overcome these shortfalls (e.g., Oracle has PL/SQL [procedural language/SQL] and PeopleSoft uses SQR [structured query and reporting]). Also, fourth generation languages (4GL) have tried to make a debut, but a lack of standardization has prevented them from supplanting 3GL (third generation languages). "Natural Language" is an example of a 4GL, where a user could type in a question and the computer understands what is being requested. For example: "List the names of all the dogs who know how to fetch and who are at the Ruff Haus kennel and list them in alphabetical order by last name, then first name." Obviously, once programmers actually get this working, it won't be long before you can verbally ask a database a question like I just posited. Star Trek here we come!

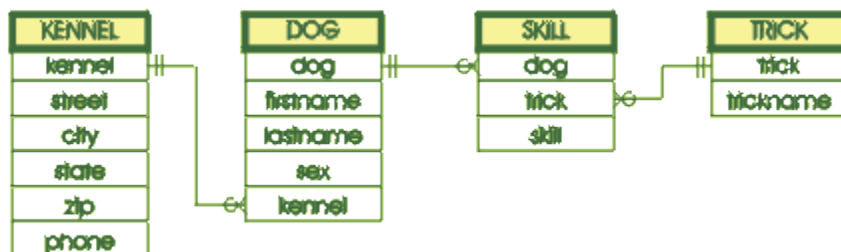
REFERENTIAL INTEGRITY

PREVENT ORPHANS BY KILLING CHILDREN BEFORE PARENTS

I think it was in 2000 when I overheard saying “To prevent orphans you must kill the children before killing the parents.” I was with friends at Sushi Nabe (an excellent Sushi restaurant in Chattanooga), and we received some unusual looks from nearby tables. We weren’t talking about biological children, but about data. The topic was database design and we were explaining to a friend the concept of referential integrity. The term “parent” is applied to a record in an independent table, while the dependent records for that parent record are called “children.” In the relationship “One dog can learn zero or more skills” we’re saying that the DOG record is the parent record, and it can have zero or more child records in the SKILL table.

Database programmers often write code to ensure database integrity. One situation they want to avoid is orphaned records. These are child records with linking fields (foreign keys) that point to non-existent parent records. It’s important to delete a parent’s child records first, before deleting that parent record. In a complex database, a parent record could have children in many different tables, so it’s easy to end up with orphaned records if you have a sloppy programmer.

Take our database scenario and look at the table in its third normal form and see if you can answer the following question: “What would be required to delete a trick? A dog?”



Some joker in the office added a new trick (#13) called “Pant.” We don’t want to track a dog’s skill level at panting, so the boss wants all traces of trick #13 removed from the database. First, we delete all occurrences of the trick from the child records (i.e., the SKILL table in our scenario). If we were using SQL it would look like this:

```
DELETE FROM skill WHERE trick = 13;
```

Now that the children are gone, we can safely delete the parent record (i.e., the record in the TRICK table). In SQL it would look like this:

```
DELETE FROM trick WHERE trick = 13;
```

After that was done we learned that a new dog, Rover, was entered into our database but never actually enrolled. To complicate matters, an idiot of an employee was confused and entered data from several different registrations. Consequently, Rover’s record is trash, and it’s useless even for marketing purposes. Therefore, we first delete any SKILL records for Rover (he’s dog #45) before deleting dog record #45. In SQL that would be two commands:

```
DELETE FROM skill WHERE dog = 45;
DELETE FROM dog WHERE dog = 45;
```

RECORD LOCKING

The deletion of records often requires an exclusive lock on those records. Record locking is a database operation that prevents other users from accessing a record while it is locked. This prevents two people from editing the same record simultaneously (either somebody's changes will be lost, or the data will be mixed!). Different database products have different features and methods for locking, but I wanted you to know about them.

Now let's think about a record locking example. What tables or records need to be locked on the network during modification or deletion of a dog? An elegant solution is to only lock the dog record, and apply inherited locking to all child records. Your program would be written so that an attempt to delete a child record would first lock the parent. The same would apply to record changes: any modifications to a child record require that you have a lock on its parent.

STORED PROCEDURES

Normally, you don't want database integrity to be the responsibility of your end users. Imagine the horrified looks on their faces as you insist that they must "prevent orphans by killing the children before killing the parents!" And do you want to teach SQL to every user? I'm not talking about the enormous workload the training would require — I'm talking about the dangers of nuclear-grade weapons in inexperienced hands. An entire database can be devastated by an incompetent or disgruntled employee in a matter of minutes.

For your sake and theirs, database access is handled through pre-written programs. The programs control who can do what, and hide the complexity of the programming commands. Only developers, system administrators, and select power-users would be given access to the database via SQL commands (and even then, a database administrator might only grant query access — inserts, updates, and deletions would be disabled).

There are some data integrity rules that should be universally enforced throughout an application. Some of these rules can be enforced by the database itself. For example, an employee's name is normally a required field in an employee table! Most databases let you set a "Not Null" constraint so that a record can NEVER be added or changed so that the name is missing. Another common constraint is the Unique Constraint. The unique constraint is frequently used on the linking fields (i.e., primary keys) of a table.

Then there are constraints that are much more complex and that are not delivered functionality of the database. You must code these constraints in your program, and perform data validation before data is added, deleted, or stored. For example, you might have a business rule that states:

Employees cannot take more than a week of vacation until they've worked with the company for more than two years. All employees, regardless of seniority, cannot take more than two contiguous weeks of vacation. Also, anyone taking more than four weeks of vacation in a calendar year will be disciplined.

Enforcing that business rule is going to take quite a bit of programming, and a single SQL statement isn't going to do it! Once you've programmed a validation routine, you've got to implement it somewhere. You could store it in an application, but there is another location available. Many databases allow you to store code in the database itself! If a rule or program universally applies to a table or field, why not store it there? Programs that are stored in a database are called "Stored Procedures" or sometimes "Stored Statements." A special type of stored procedure is a "Trigger" which is what we'll discuss next.

TRIGGERS

Triggers are stored procedures that are automatically run by the database whenever certain “trigger” events occur. These events are normally a record INSERT, UPDATE, or DELETE. For example, you could create a DELETE trigger for the DOG table that automatically deletes all child records of that dog in the SKILL table. Since the trigger is stored in the database it doesn’t matter what application or program accesses your database — you can enforce data integrity through your trigger! Stored procedures and triggers give you a centralized location for core logic, prevent the circumvention of integrity rules, and are integrated with the data they protect.

HANDS-ON EXERCISES**INSTRUCTIONS**

Modify the ERD that we've just normalized. Use pencil so you can erase and don't worry about making mistakes — you're learning!

Add the minimal number of fields to solve the problem. You will come up with different ER diagrams if you try to track associated data that isn't requested by the problem! Don't assume and don't track related data that may seem implied.

Solutions are given at the end of this page, but if you look there without giving due diligence, then you're not going to remember anything that you've read. Be sure to work the exercises and THEN compare them to the solutions. Learning dramatically improves when immediate feedback is received on an incorrect attempt.

EXERCISE 1

TPI is a member of F.L.E.A., and in order to retain our accreditation we must rank each dog using a formula that F.L.E.A. provided:

$$(\text{SUM}((\text{skill/difficulty}) * 100)) / \text{tricks}$$

That is, we need to assign each trick a difficulty rating between 01 and 100. To calculate a dog's rank, we divide each skill he's learned by the difficulty rating for that trick. Each of these results are then summed and the total is multiplied by 100. That number is then divided by the number of tricks known by the dog. This final figure is the dog's rank, and the ranks for all dogs enrolled at TPI will be printed and mailed to the F.L.E.A. headquarters each month. Please modify the ERD to show any new fields, tables, and relationships required to support this business requirement.

EXERCISE 2

TPI is charging per trick taught. Therefore, we need to track the price paid for each trick. We want to know what we CHARGED for the class (i.e., what did we charge that DAY when training began for that dog?). This price won't be updated, even if the price of the trick should change. For example, Fido learns to "Jump" for \$10.00, but a year later the cost is \$12.00. Despite the price hike, we can still see that Fido learned "Jump" for just \$10.00.

We're not concerned with tracking the history of prices. The boss only wants to know what was paid to learn a skill, and what the price of a trick is currently. He does NOT want to store a list of "The price is \$12.00 now, but before that it was \$11.50, and before that it was \$10.00, and before that it was \$9.75."

EXERCISE 3

The boss feels like a dog's ability to learn is influenced by the kennel where he's housed. When we have a year's worth of data, we'll analyze this and see if there's any correlation. However, we need to begin collecting data now, so we'll have something to work with next year. Therefore, we want to track where a dog was staying when he learned one of our tricks. We've examined our business process, and we've decided that we will record the information on the first day of class. We don't care where the dog was staying after that day (in case the dog switches kennels mid-class).

EXERCISE 4

The boss wants to track information on each employee at TPI. He unwisely insists on using each employee's social security number as their identifying and linking key. Other than SSN, all that is needed right now is the employee's first name, middle name, and last name. He also wants you to change the database model, so we can track which employee taught a dog any new skills.

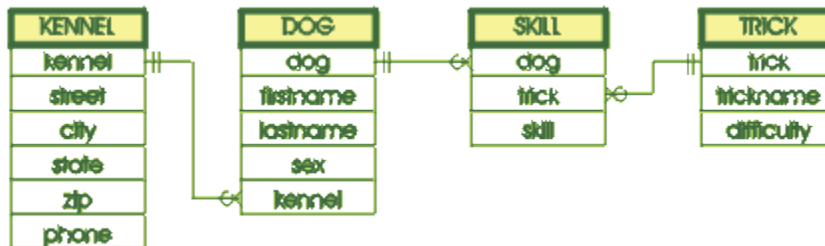
EXERCISE 5

Since not all TPI employees are trainers, the boss wants to add a way to track an employee's job title. Employees may perform several job functions, but they will be assigned only one primary job based on their expertise. Change the database model to allow us to track job titles.

SOLUTIONS

EXERCISE 1: ACCREDITATION RANKING

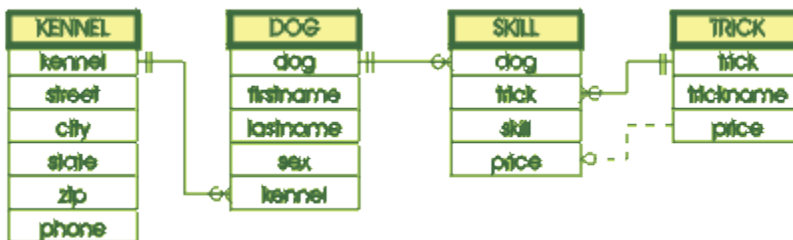
The only new entity is the DIFFICULTY rating. The rank can be calculated, so it doesn't need to be (and shouldn't be) stored in the database. Difficulty is an attribute of a trick, so it is placed in the TRICK table.



There might be a situation where you might break the normalization, and store the calculation results in a RANK field: If RANK was frequently needed and you were experiencing a performance degradation that couldn't be address in any other way. Then you might break the rules by adding a non-editable RANK field to the DOG table. If you did this, then you would also have to create triggers for both the SKILL and TRICK tables. These triggers would recalculate the RANK and update the DOG table whenever any field affecting a ranking was inserted, updated, or deleted.

EXERCISE 2: CURRENT VS. CHARGED PRICE

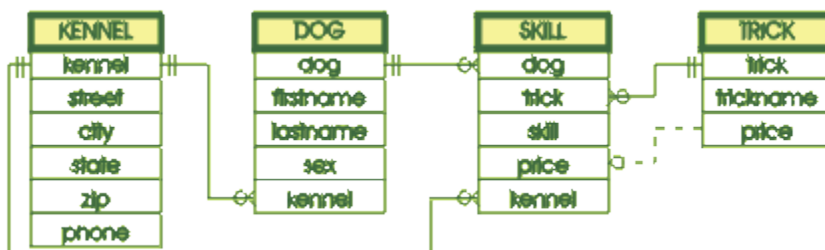
The only new entity is PRICE. However, we want to keep the paid price distinct from the current price. Therefore, the PRICE paid is an attribute of the SKILL learned. So, the first PRICE field is added to the SKILL table. The current PRICE is an attribute of the TRICK, so we add a PRICE field to the TRICK table that has the same domain. When a record is inserted into the SKILL table, we simply copy the TRICK.PRICE into SKILL.PRICE, and from there on, we don't touch SKILL.PRICE.



You'll notice that we show a special relationship that was only hinted at before. Notice the dashed line connecting the two PRICE fields. If you look back to the session on ERD notation, you'll see that the physical ERD uses a dashed line to indicate a "non-identifying" relationship. In this hybrid ERD the arrow head shows that SKILL.PRICE comes from TRICK.PRICE, but it isn't linked to TRICK.PRICE (it's not a linking key).

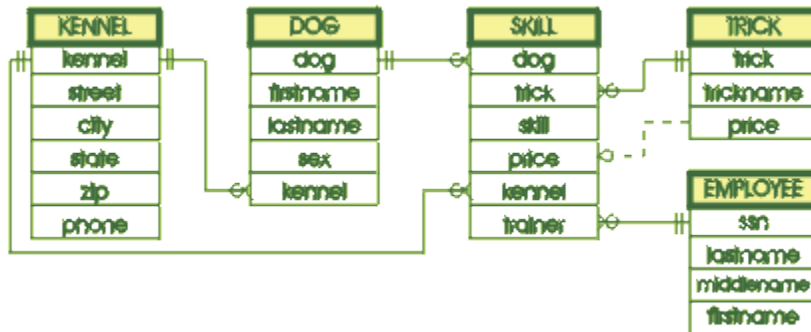
EXERCISE 3: KENNEL / TRICK CORRELATION

This problem was straight forward: we wanted to track the relationship between the KENNEL and the SKILL learned by the dog. So, we simply link SKILL to KENNEL with a linking field. We add SKILL.KENNEL (a foreign key) which is the link to KENNEL.KENNEL (the KENNEL table's primary key). You'll notice that KENNEL.KENNEL has two links going "out" to other tables. In core tables of a commercial database, you can see many, many relationships tied to a single field.



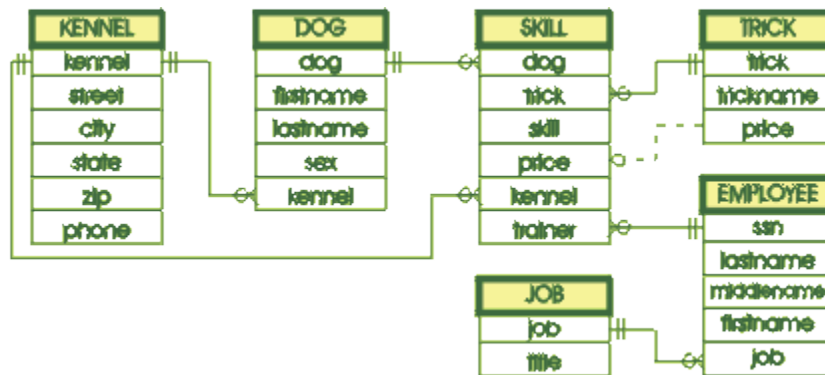
EXERCISE 4: TRACK EMPLOYEE TRAINERS

The new entities SSN⁶, LASTNAME, FIRSTNAME, and MIDDLENAME are added to an EMPLOYEE table. To link EMPLOYEE with SKILL we add SKILL.TRAINER which links to EMPLOYEE.SSN. The ERD signifies that one employee may be the trainer of zero or more skills



EXERCISE 5: TRACK EMPLOYEE JOB TITLES

The new entity is primarily TITLE which we store in a JOB table. We relate JOB.TITLE to the EMPLOYEE via the JOB.JOB and EMPLOYEE.JOB fields. The relationship is diagrammed to show that one job may be held by zero or more employees.



⁶ As mentioned previously, primary keys should be arbitrary, and we'd want to convince the boss to use an "employee number" field instead. The current requirement the boss demanded could expose confidential SSN data in a privacy violating manner if we're not careful.

FINAL BRAIN BUSTER

THE PROBLEM

Add a field or fields that will allow the creation of a mailing list report. This report should be able to create the listing with a single database query. (By single, I mean “in one trip”). The database model should be designed so that each of these scenarios can be retrieved using a single query (although each query will be different, only one of them is needed each time):

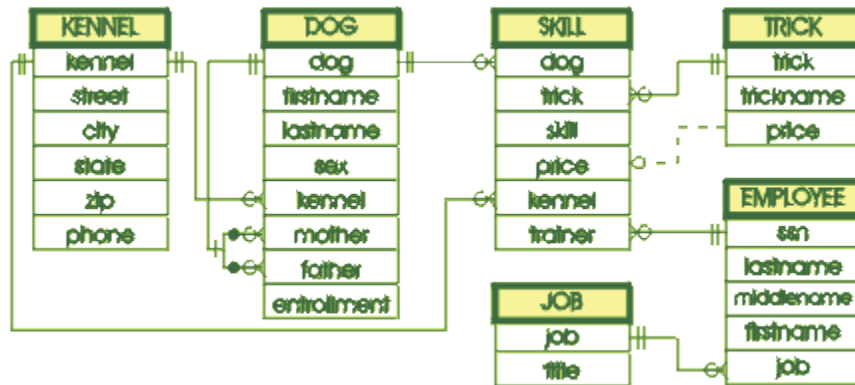
- A dog’s parents,
or
- All of a dog’s children,
or
- All of a dog’s children by one specific mate.

The query can also have the following filter options:

- List only dogs who are currently enrolled at TPI (Teacher’s Pet, Inc.). We’ll use this to mail out our newsletter.
- List only dogs who are NOT currently enrolled at TPI but have attended in the past. We’ll use this to mail alumni meeting notices.
- List only dogs who are currently or previously enrolled at TPI. We’ll use this to mail out “preferred customer” discount coupons.
- List only dogs who are not currently enrolled, and have never been enrolled. We’ll use this to send out promotional coupons.

THE SOLUTION

While it may not have been obvious, relationships don't always have to be between tables. The most elegant solution is to relate a table back to itself (this type of relationship is called a "pig's ear."). Therefore, both DOG.MOTHER and DOG.FATHER contain links to DOG.DOG. You'll also see that we used the "mutually exclusive" notation. That is, a dog can FATHER zero or more children, or a dog can MOTHER zero or more children, but a dog cannot both FATHER and MOTHER children!



In addition to the two parent fields, we also added an ENROLLMENT field to track a dog's enrollment status. Our ERD doesn't show how this works, but our data dictionary would probably show that this field contains three values: 0 = not enrolled, 1 = currently enrolled, and 2 = alumnus.

Now, let's take one of the query scenarios and show how this would look in SQL. If we wanted to list all of Fido's children (let's say Fido is dog #33) by mate Fifi (dog #99), but only those offspring who have been enrolled or are currently enrolled:

```
SELECT dog.firstname,  
       dog.lastname  
FROM   dog  
WHERE  dog.father = 33  
       AND dog.mother = 99  
       AND dog.enrollment > 0;
```

BIBLIOGRAPHY

- Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," reprinted in *Readings in Database Systems*, M. Stonebraker, ed., Morgan Kaufmann, 1988. (Originally published in 1970.)
- Codd, E. F., "Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks," *IBM Research Journal*, R J 599 (#12343), August 19, 1969.
- Codd, E. F., "Extending the Database Relational Model to Capture More Meaning," reprinted in *Reading in Database Systems*, M. Stonebraker, ed., Morgan Kaufmann, 1988. (Originally published in 1979.)
- Codd, E. F., "How Relational Is Your Database Management System?" *Computerworld*, October 14 and 21, 1985.
- Codd, E. F., "The Relational Model for Database Management," *Version 2*, Addison-Wesley, 1990.
- Codd, E.F.. "A Relational Model of Data for Large Shared Data Banks." *Communications of the Association for Computing Machinery*. Issue 13, 1970. p.6.
- Date, C. J. and McGoveran, D., "A New Database Design Principle," *Database Programming & Design*, 7(7):46-53, July 1994.
- Date, C. J. and McGoveran, D., "Updating Joins and Other Views," *Database Programming & Design*, 7(8):43-49, August 1994.
- Date, C. J. and McGoveran, D., "Updating Union, Intersection, and Difference Views," *Database Programming & Design*, 7(6):46-53, June 1994.
- Halpin, T.A.. "Conceptual Schema and Relational Database Design." Prentice Hall, 1993.
- McGoveran, D., "Classical Logic: Nothing Compares 2 U," *Database Programming & Design*, 7(1):54-61, January 1994.
- McGoveran, D., "Nothing From Nothing (or, What's Logic Got to Do With It?)," *Database Programming & Design*, 6(12):32-41, December 1993.
- McGoveran, D., "Nothing From Nothing Part III: Can't Lose What You Never Had," *Database Programming & Design*, 7(2):42-48, February 1994.
- McGoveran, D., "Nothing From Nothing Part IV: It's In the Way That You Use It," *Database Programming & Design*, 7(6):54-63, March 1994.
- McGoveran, David. "The relational model turns 25... and we're still trying to get it right." *DBMS* Oct 1994 v7 n11 p46(7)
- Mezick, Dan. "Data Modeling: The Building Blocks to Better Apps." *Database Advisor*, October 1993. p. 79. Discusses the ERA methodology originated by Relational Systems Corp. of Birmingham, Alabama.