7

Normalizing Data

Chapter 6 explained how you can make a database more flexible and robust by extracting certain business rules from the database's structure. By removing some of the more complex and changeable rules from the database's check constraints, you make it easier to change those rules later.

Another way to make a database more flexible and robust is to ''normalize'' it. Normalization makes the database more able to accommodate changes in the structure of the data. It also protects the database against certain kinds of errors.

This chapter explains what normalization is and tells how you can use it to improve your database design.

In this chapter you learn:

- What normalization is.
- What problems different types or levels of normalization address.
- How to normalize a database.
- How to know what level of normalization is best for your database.

After you normalize your relational model, you'll be ready to build the database.

What Is Normalization?

Depending on how you design a relational database, it may be susceptible so all sorts of problems. For example:

- It may contain lots of duplicated data. This not only wastes space but it also makes updating all of those duplicated values a time-consuming chore.
- It may incorrectly associate two unrelated pieces of data so you cannot delete one without deleting the other.

- It may require a piece of data that shouldn't exist in order to represent another piece of data that should exist.
- It may limit the number of values that you can enter for what should be a multi-valued piece of data.

In database terminology, these issues are called *anomalies*. (Anomaly is a euphemism for ''problem.'' I'm not sure why this needs a euphemism — I doubt the database's feelings would be hurt by the word ''problem.'')

Normalization is a process of rearranging the database to put it into a standard (normal) form that prevents these kinds of anomalies.

There are seven different levels of normalization. Each level includes those before it. For example, a database is in Third Normal Form if is in Second Normal Form plus it satisfies some extra properties. That means if a database is at one level of normalization, then by definition it gets the advantages of the ''lower'' levels.

The different levels of normalization in order from weakest to strongest are:

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce-Codd Normal Form (BCNF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)
- Domain/Key Normal Form (DKNF)

A database in DKNF has amazing powers of protection against anomalies, can leap tall buildings, and has all sorts of other super-database powers.

The following sections explain the properties that a database must satisfy to officially earn one of these coveted uber-database titles. They also explain the data anomalies that each level of normalization prevents.

First Normal Form (1NF)

First Normal Form basically says that the data is in a database. It's sort of the price to play the game if you want be a relational database.

Most of the properties needed to be in 1NF are enforced automatically by any reasonable relational database. There are a couple of extra properties added on to make the database more useful, but mostly these rules are pretty basic. The official qualifications for 1NF are:

1. Each column must have a unique name.
2. The order of the rows and columns doesn't matter.
3. Each column must have a single data type.
4. No two rows can contain identical values.
5. Each column must contain a single value.
6. Columns cannot contain repeating groups.

The first two rules basically come for free when you use a relational database product such as Access, SQL Server, or MySQL. All of these require that you give columns different names. They also don't really care about the order of rows and columns, although when you select data you will probably want to specify the order in which it is returned for consistency's sake.

Rule 3 means two rows cannot store different types of data in the same column. For example, the Value field in a table cannot hold a string in one row, a date in another, and a currency value in a third. This is almost a freebie because database products won't let you say, ''This field should hold numbers or dates.''

One way to run afoul of Rule 3 is to store values with different types converted into a common form. For example, you could store a date written as a string (such as ''3/14/2012'') and a number written as a string (such as ''17'') in a column designed to hold strings. Though this is an impressive display of your cleverness, it violates the spirit of the rule. It makes it much harder to perform queries using the field in any meaningful way. If you really need to store different kinds of data, split them apart into different columns that each holds a single kind of data. (In practice, many databases end up with just this sort of field. In particular, users often enter key data in comment or notes fields and a program must later search for values in those fields. Not the best practice but it does happen.)

Rule 4 makes sense because, if two rows *did* contain identical values, how would you tell them apart? The only reason you might be tempted to violate this rule is if you don't need to tell the rows apart. For example, suppose you fill out an order form for a pencil, some paper, and a tarantula. Oh, yeah, you also need another pencil so you add it at the end.

> *This reminds me of the joke where some guy wants to buy two new residents for his aquarium by mail-order (this was before Internet shopping) but he doesn't know whether the plural of octopus is octopi or octopuses. So he writes, ''Dear Sirs, please send me an octopus. Oh and please send me another one.''*

Now the form's list of items contains two identical rows listing a pencil. You don't care that the rows are identical because the pencils are identical. In fact, all you really know is that you want two pencils. That observation leads to the solution. Instead of using two identical rows, use one row with a new Quantity field and set Quantity to 2.

Note that Rule 4 is equivalent to saying that the table can have a primary key. Recall from Chapter 3 that a primary key is a set of columns that you can use to uniquely identify rows. If no two rows can have exactly the same values, then you must be able to pick a set of columns to uniquely identify the rows, even if it includes every column.

In fact, let's make that a new rule:

4. (continued). Every table has a primary key.

Rule 5 is the one you might be most tempted to violate. Sometimes a data entity includes a concept that needs multiple values. The semantic object models described in Chapter 5 even let you explicitly set an attribute's cardinality so you can make one attribute that holds multiple values.

For example, suppose you are building a recipe table and you give it the fields Name, Ingredients, Instructions, and Notes (which contains things such as ''Sherri loves this recipe'' and ''For extra flavor, increase ants to 3 tbl.''). This gives you enough information to print out a recipe and you can easily follow it (assuming you have some talent for cooking and a garden full of ants).

However, the Ingredients, Instructions, and Notes fields contain multiple values. Two hints that this might be the case are the fact that the column names are plural and that the column values are probably broken up into sub-values by commas, periods, carriage returns, or some other delimiter.

Storing multiple values in a single field limits the usefulness of that field. For example, suppose you decide that you want to find all of your recipes that use ants as an ingredient. Because the Ingredients field contains a bunch of different values all glommed together, you cannot easily search for a particular ingredient. You might be able to search for the word ''ants'' within the string, but you're likely to get extraneous matches such as ''currants.'' You also won't be able to use indexes to make these searches in the middle of a string faster.

The solution is to break the multiple values apart, move them into a new table, and link those records back to this one with this record's primary key. For the recipe example, you would create a RecipeIngredients table with fields RecipeId, Ingredient, and Amount. Now you can search for RecipeIngredients records where Ingredient is ''ants.''

Similarly, you could make a RecipeInstructions table with fields RecipeId, StepNumber, and Instruction. The StepNumber field is necessary because you want to perform the steps in the correct order. (I've tried rearranging the steps and it just doesn't work! Baking bread before you mix the ingredients gives you a strange little brick-like puddle.) Now you can search for Recipes records and matching RecipeInstructions records that contain the word ''preheat'' to see how hot the oven must be.

Note that you only need to separate a field's values if they are logically distinct for whatever purposes you will use them. For example, you might want to search for individual ingredients. It's a bit less clear that you'll need to search for particular instructions. It's even less sure that you'll want to search for specific values within a Notes field. Notes is more or less a free-format field, and it's not clear that any internal structure is important.

For an even more obvious example, consider an Authors table's Biography field. This field contains a brief biography of the author. You could break it into sentences (or words, or even letters), but the individual sentences don't have any real context, so there's little point. You will display the Biography as a whole anyway so there's little benefit in chopping it up arbitrarily.

Rule 6 means you cannot have multiple columns that contain values that are not distinguishable. For example, suppose you decide that each Exterminators record should be able to hold the animals for which an exterminator is qualified to remove (muskrat, ostrich, platypus, and so forth). (Don't worry, this is a humane pest control service, so the exterminators catch the critters and release them far away so they can bug someone else.)

You already know from Rule 5 that you can't just cram all of the animals into a single Critters field. Rule 6 says you also cannot create columns named Critter1, Critter2, and Critter3 to hold different animals. Instead you need to split the values out into a new table and use the Exterminators table's primary key to link back to the main records.

Figure 7-1 shows a relational model for the recipe data. The ingredients and instructions have been moved into new tables but the Notes field remains as it was originally.

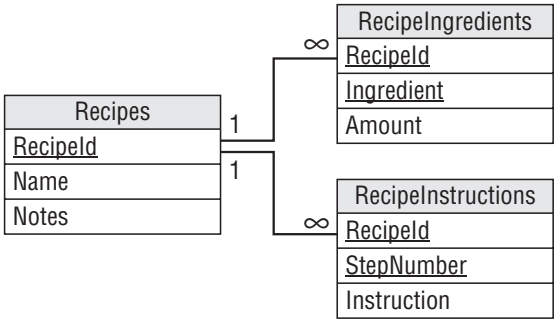


Figure 7-1

Try It Out Arranging Data in the First Normal Form

The following table contains information about airline flights. It contains data about a party of two flying from Denver to Phoenix and a party of three flying from San Diego to Los Angeles. The first two columns give the start and destination cities. The final column gives the connection cities (if any) or the number of connections. The rows are ordered so the frequent flyer passengers are at the top, in this case in the first three rows.

City	City	Connections
DEN	PHX	1
SAN	LAX	JFK, SEA, TPA
SAN	LAX	JFK, SEA, TPA
DEN	PHX	1
SAN	LAX	JFK, SEA, TPA

Your mission, should you decide to accept it, is to put this atrocity into First Normal Form:

1. Make sure every column has a unique name. This table has two columns named City. To fix these problems (sorry, I mean ''anomalies''), rename those columns to StartCity and DestinationCity.
2. Make sure the order of the rows and columns doesn't matter. If the order of the rows matters, add a column to record the information implied by their positions. In this example, make a new Priority column and explicitly list the passengers' priorities.
3. Make sure each column holds a single data type. If a column holds more than one type of data, split it into multiple columns, one for each data type. In this case, list the connecting cities, and don't even record the number of cities. Just count them when necessary.

4. Make sure no two rows can contain identical values. If two rows contain identical values, add a field to differentiate them. In this case, add a CustomerId column so you can tell the customers apart.
5. Make sure each column contains a single value. If a column holds multiple data values, split them out into a new table. Make a new Connections table and move the connection data there. To tie those records back to their original rows in this table, add columns that correspond to the primary key columns in the original table (CustomerId and Date).
6. Make sure multiple columns don't contain repeating groups. In this example, you need to think about the two city fields and decide whether they contain distinguishable values.

How It Works

1. Make sure every column has a unique name.

   Rule 1 says each column must have a unique name, but this table has two columns named City. This also sort of violates the rule that the order of the columns cannot matter because we're using the ordering to know which is the start city and which is the destination city.

   After you rename the columns, the table looks like this:

StartCity	DestinationCity	Connections
DEN	PHX	1
SAN	LAX	JFK, SEA, TPA
SAN	LAX	JFK, SEA, TPA
DEN	PHX	1
SAN	LAX	JFK, SEA, TPA

2. Make sure the order of the rows and columns doesn't matter. If the order of the rows matters, add a column to record the information implied by their positions.

   Rule 2 says the order of the rows and columns doesn't matter. After making the first change, the order of the columns doesn't matter any more because you can use the column names rather than their order to tell which city is which. However, you're using the order of the rows to determine which passengers have the highest priority (frequent flyers get the caviar and pheasant while the others get Twinkies and Spam).

   To fix this, take whatever concept the row ordering represents and move it into a new column. After you make a new Priority column and explicitly list the passengers' priorities, the ordering of the rows doesn't matter because you can retrieve the original idea of who has higher priority from the new column. Now the table looks like this:

StartCity	DestinationCity	Connections	Priority
DEN	PHX	1	1
SAN	LAX	JFK, SEA, TPA	1
SAN	LAX	JFK, SEA, TPA	1
DEN	PHX	1	2
SAN	LAX	JFK, SEA, TPA	2

3. Make sure each column holds a single data type. If a column holds more than one type of data, split it into multiple columns, one for each data type.

   Rule 3 says each column must have a single data type. Here the Connections column holds either a list of connecting cities or the number of connections, two different kinds of data. There are at least two reasonable solutions for this problem (at least for right now).

   First, you could make two columns, ConnectingCities and NumberOfConnections, and split these values into their proper columns. This would be the better solution if you really needed both of these types of values.

   In this case, however, the number of connections is just a count of the number of connecting cities so, if you knew the cities, you could just count them to get the number of cities. The better solution in this case is to list the connecting cities and calculate the number of those cities when necessary. Here's the new table:

StartCity	DestinationCity	Connections	Priority
DEN	PHX	LON	1
SAN	LAX	JFK, SEA, TPA	1
SAN	LAX	JFK, SEA, TPA	1
DEN	PHX	LON	2
SAN	LAX	JFK, SEA, TPA	2

4. Make sure no two rows can contain identical values. If two rows contain identical values, add a field to differentiate them.

   Rule 4 says no two rows can contain identical values. Unfortunately this table's second and third rows are identical. The question now becomes, ''Do you care that you cannot tell these records apart?''

If you are a cold, heartless, big corporation airline and you don't care who is flying, just that *someone* is flying, then you don't care. In that case, add a Count field to the table and use it to track the number of identical rows. This would be the new design:

StartCity	DestinationCity	Connections	Priority	Count
DEN	PHX	LON	1	1
SAN	LAX	JFK, SEA, TPA	1	2
DEN	PHX	LON	2	1
SAN	LAX	JFK, SEA, TPA	2	1

However, if you're a warm, friendly, mom-and-pop airline, then you do care who has which flight. In that case, what is the difference between the two identical rows? The answer is that they represent different customers so the solution is to add a column to differentiate between the customers. If you add a CustomerId column, then you don't need the Count column and the table becomes:

StartCity	DestinationCity	Connections	Priority	CustomerId
DEN	PHX	LON	1	4637
SAN	LAX	JFK, SEA, TPA	1	12878
SAN	LAX	JFK, SEA, TPA	1	2871
DEN	PHX	LON	2	28718
SAN	LAX	JFK, SEA, TPA	2	9287

This works for now, but what if one of these customers wants to make the same trip more than once on different dates? In that case the table will hold two identical records again. Again you can ask yourself, what is the difference between the two identical rows? The answer this time is that the trips take place on different dates so you can fix it by adding a Date column.

StartCity	DestinationCity	Connections	Priority	CustomerId	Date
DEN	PHX	LON	1	4637	4/1/10
SAN	LAX	JFK, SEA, TPA	1	12878	6/21/10
SAN	LAX	JFK, SEA, TPA	1	2871	6/21/10
DEN	PHX	LON	2	28718	4/1/10
SAN	LAX	JFK, SEA, TPA	2	9287	6/21/10

Rule 4a says the table should have a primary key. The combination of CustomerId and Date can uniquely identify the rows. This seems like a safe combination because one customer cannot take two flights at the same time (although customers have been known to book multiple flights at the same time and then cancel all but one).

5. Make sure each column contains a single value. If a column holds multiple data values, split them out into a new table.

   Rule 5 says each column must contain a single value. This table's Connections column clearly violates the rule. To solve this problem, make a new Connections table and move the connection data there. To tie those records back to their original rows in this table, add columns that correspond to the primary key columns in the original table (CustomerId and Date).

   That single change leads to a big problem, however. The values in the combined Connections column implicitly defined the connections' order. Using the new table, you cannot tell which connections come before which. (Unless you use the ordering of the rows to decide which comes first, and you know that's not allowed!)

   The solution is to add a ConnectionNumber field to the new table so you can figure out how to order the connections.

   Figure 7-2 shows the tables together with lines connecting the corresponding records.

StartCity	DestinationCity	Priority	CustomerId	Date
DEN	PHX	1	4637	4/1/10
SAN	LAX	1	12878	6/21/10
SAN	LAX	1	2871	6/21/10
DEN	PHX	2	28718	4/1/10
SAN	LAX	2	9287	6/21/10

CustomerId	Date	ConnectionNumber	City
4637	4/1/10	1	LON
12878	6/21/10	1	JFK
12878	6/21/10	2	SEA
12878	6/21/10	3	TPA
2871	6/21/10	1	JFK
2871	6/21/10	2	SEA
2871	6/21/10	3	TPA
28718	4/1/10	1	LON
9287	6/21/10	1	JFK
9287	6/21/10	2	SEA
9287	6/21/10	3	TPA

Figure 7-2

Figure 7-3 shows a relational model for these tables. (In the context of airline connections, that infinity symbol is kind of depressing.)

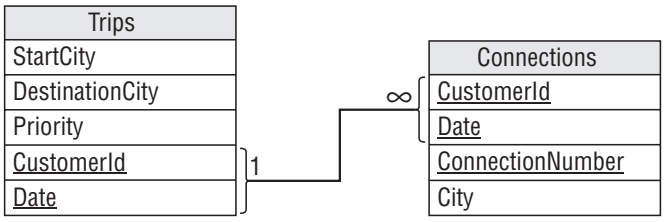


Figure 7-3

6. Make sure multiple columns don't contain repeating groups. In this example, you need to think about the two city fields and decide whether they contain distinguishable values.

Rule 6 says that multiple columns don't contain repeating groups. In this example, the two city fields contain very similar types of values: cities. Unlike the exterminator example described earlier, however, these values are distinguishable. Starting and ending destination are not the same thing, and they are not the same as connecting cities. (Although I had a travel agent once who may not have fully understood the difference, judging by where my rental car was reserved.)

Database purists would say that having two fields containing the same kind of data is a bad thing and you should move the values into a new table.

My take on the issue is that it depends on how you are going to use the values. Will you ever want to ask, ''Which customers ever visit San Jose?'' If so, then having these cities in two separate fields is cumbersome because you'll have to ask the same question about each field. In fact, you'll also have to ask about the connecting cities. In this case, it might be better to move the start and destination cities out of this table.

In contrast, suppose you never ask what cities customers visit and instead ask, ''Which customers start from Berlin?'' or ''Which customers finish in Madrid?'' In that case, keeping these values in separate fields does no harm.

For now I'll leave them alone and take up this issue again in the next section.

As a quick check, you should also verify that the new table is in 1NF. If you run through the rules, you'll find that most of them are satisfied trivially but a few are worth a moment of reflection.

Rules 4 says no two rows can contain identical values and the table must have a primary key. Assuming no customer takes more than one trip per day, then the Trips table's CustomerId/Date fields and the Connections table's CustomerId/Date/ConnectionNumber fields make reasonable primary keys. If you need to allow customers to take more than one trip per day (which happens in the real world), you probably need to add another TripNumber field to both tables.

Rule 5 says every column must contain a single value. If we split apart the values in the original table's Connections column, the new table should be okay.

But we'll need a separate table to track where the luggage goes.

Second Normal Form (2NF)

A table is in 2NF if:

1. It is in 1NF.
2. All of the non-key fields depend on all of the key fields.

To see what this means, consider the alligator wrestling schedule shown in the following table. It lists the name, class (amateur or professional), and ranking for each wrestler, together with the time when this wrestler will perform. The Time/Wrestler combination forms the table's primary key.

Time	Wrestler	Class	Rank
1:30	Annette Cart	Pro	3
1:30	Ben Jones	Pro	2
2:00	Sydney Dart	Amateur	1

Time	Wrestler	Class	Rank
2:15	Ben Jones	Pro	2
2:30	Annette Cart	Pro	3
3:30	Sydney Dart	Amateur	1
3:30	Mike Acosta	Amateur	6
3:45	Annette Cart	Pro	3

Though this table is in 1NF (don't take my word for it, verify it yourself), it is trying to do too much work all by itself and that leads to several problems.

> *Note that the Wrestler field contains both first and last names. This would violate 1NF if you consider those as two separate pieces of information. For this example, assume you only need to display first and last name together and will never need to perform searches on last name only, for example. This is confusing enough without adding extra columns.*

First, this table is vulnerable to update anomalies. An *update anomaly* occurs when a change to a row leads to inconsistent data. In this case, update anomalies are caused by the fact that this table holds a lot of repeated data. For example, suppose Sydney Dart decides to turn pro, so you update the Class entry in the third row. Now that row is inconsistent with the Class entry in row 6 that still shows Sydney as an amateur. You'll need to update every row in the table that mentions Sydney to fix this problem.

Second, this table is susceptible to deletion anomalies. A *deletion anomaly* occurs when deleting a record can destroy information that you might need later. In this example, suppose you cancel the 3:30 match featuring Mike Acosta. In that case you lose the entire 7th record in the table, so you lose the fact that Mike is an amateur, that he's ranked 6th, and even that he exists (presumably he disappears in a puff of smoke).

Third, this table is subject to insertion anomalies. An *insertion anomaly* occurs when you cannot store certain kinds of information because it would violate the table's primary key constraints. Suppose you want to add a new wrestler Nate Waffle to the roster but you have not yet scheduled any matches for him. (Nate's actually the contest organizer's nephew so he doesn't really wrestle alligators; he just wants to be listed in the program to impress his friends.) To add Nate to this table, you would have to assign him a wrestling match, and Nate would probably have a heart attack. Similarly, you cannot create a new time for a match without assigning a wrestler to it.

Okay, I confess I pulled a fast one here. You *could* create a record for Nate that had Time set to null. That would be really bad form, however, because all of the fields that make up a primary key should have non-null values. Many databases require that all primary key fields not allow nulls. Because Time/Wrestler is the table's primary key, you cannot give Nate a record without assigning a Time and you're stuck.

The underlying problem is that some of the table's columns do not depend on all of the primary key fields. The Class and Rank fields depend on Wrestler but not on Time. Annette Cart is a professional whether she wrestles at 1:30, 2:30, or 3:45.

The solution is to pull the columns that do not depend on the *entire* primary key out of the table and put them in a new table. In this case, you could create a new Wrestlers table and move the Class and Rank fields into it. You would add a WrestlerName field to link back to the original table.

Figure 7-4 shows a relational model for the new tables.

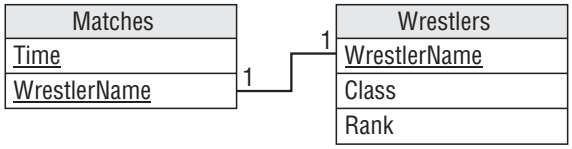
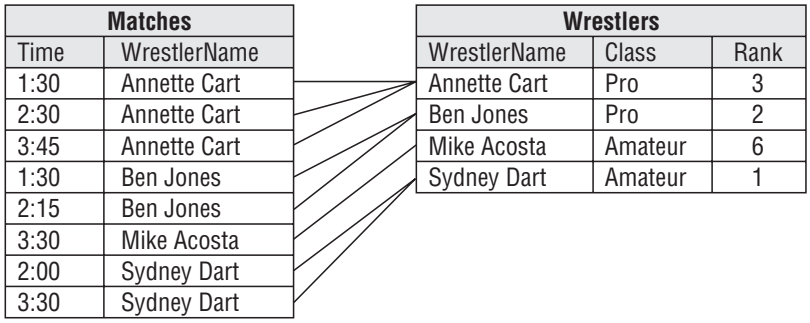


Figure 7-4

Figure 7-5 shows the new tables holding the original data. Here I've sorted the matches by wrestler name to make it easier to see the relationship between the two tables. (It's a mess if you sort the matches by time.)



Matches

Time	WrestlerName
1:30	Annette Cart
2:30	Annette Cart
3:45	Annette Cart
1:30	Ben Jones
2:15	Ben Jones
3:30	Mike Acosta
2:00	Sydney Dart
3:30	Sydney Dart

Wrestlers

WrestlerName	Class	Rank
Annette Cart	Pro	3
Ben Jones	Pro	2
Mike Acosta	Amateur	6
Sydney Dart	Amateur	1

Figure 7-5

The new arrangement is immune to the three anomalies described earlier. To make Sydney Dart a professional, you only need to change her Wrestlers record. You can cancel the 3:30 match between Mike Acosta and Hungry Bob without losing Mike's information in the Wrestler's table. Finally, you can make a row for Nate in the Wrestlers table without making one in the Matches table.

You should also verify that all of the new tables satisfy the 2NF rule, ''All of the non-key fields depend on all of the key fields.'' The Matches table contains no fields that are not part of the primary key so it satisfies this requirement trivially.

The primary key for the Wrestlers table is the WrestlerName field, and the Class and Rank fields depend directly on the value of WrestlerName. If you move to a different WrestlerName, you get different values for Class and Rank. Note that the second wrestler might have the same Class and Rank but that would be mere coincidence. The new values belong to the new wrestler.

Intuitively, the original table had problems because it was trying to hold two kinds of information: information about matches and information about wrestlers. To fix the problem, we broke the table into two tables to hold those two kinds of information separately.

If you ensure that every table represents one single, unified concept such as wrestler or match, the table will be in 2NF. It's when a table tries to play multiple roles, such as storing wrestler and match information at the same time, that it is open to data anomalies.

Try It Out Arranging Data in the Second Normal Form

Suppose you just graduated from the East Los Angeles Space Academy and you rush to the posting board to find your ship assignments:

Cadet	Position	Ship
Ash, Joshua	Fuse Tender	Frieda's Glory
Barker, Sally	Pilot	Scrat
Barker, Sally	Arms Master	Scrat
Cumin, Bil	Cook's Mate	Scrat
Farnsworth, Al	Arc Tauran Liaison	Frieda's Glory
Farnsworth, Al	Interpreter	Frieda's Glory
Major, Major	Cook's Mate	Scrat
Pickover, Bud	Captain	Athena Ascendant

This table uses the Cadet/Position combination as a primary key. Note that some cadets have more than one job.

To earn your posting as Data Minder First Class:

1. Describe the table's update, deletion, and insertion anomalies.
2. Put it in 2NF. Find any fields that don't depend on the entire primary key and move them into a new table. In this case, split the table into two new tables: CadetPositions and CadetShips. The CadetPositions table is similar to the original table with the Ship field removed. The CadetShips table links the cadets with their ships.

How It Works

1. Describe the table's update, deletion, and insertion anomalies.

   This table allows update anomalies because it contains repeated values. For example, if you change Sally Barker's Ship in row 2 to Athena Ascendant, it would conflict with the Ship value in row 3 that says Sally is on the Scrat.

   This table also allows deletion anomalies. If you delete the last row, you no longer know that the Athena Ascendant is a Courier class ship or that she even exists. (Her crew will be mad when headquarters stops sending paychecks.)

   Finally, the table allows insertion anomalies because you cannot store information about a cadet without a ship or a ship without a cadet.

2. Put it in 2NF. Find any fields that don't depend on the entire primary key and move them into a new table.

This table has problems because some of the fields don't depend on the entire primary key. In particular, the Ship field depends on the Cadet (it's the ship where this cadet is assigned) but it does not depend on the Position. You could solve the problem if you could remove Position from the primary key but we need both Cadet and Position to uniquely identify a record.

The solution is to split the table into two new tables: CadetPositions and CadetShips, and move the ship information (which doesn't depend on the entire primary key) into the new table. Figure 7-6 shows the relational model for this new design.

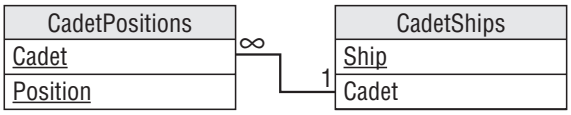
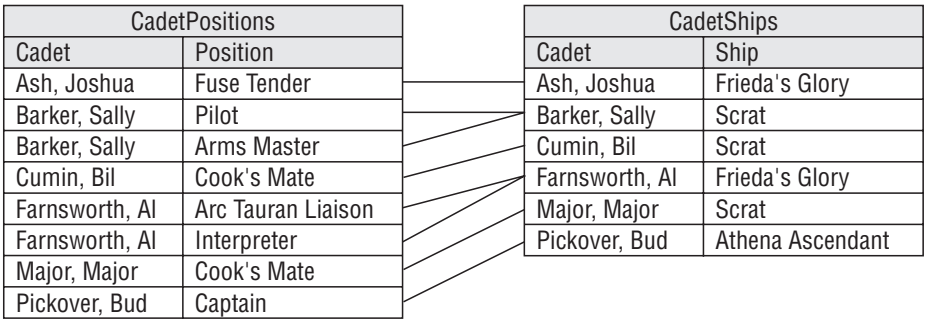


Figure 7-6

Figure 7-7 shows the new tables and their data.



CadetPositions

Cadet	Position
Ash, Joshua	Fuse Tender
Barker, Sally	Pilot
Barker, Sally	Arms Master
Cumin, Bil	Cook's Mate
Farnsworth, Al	Arc Tauran Liaison
Farnsworth, Al	Interpreter
Major, Major	Cook's Mate
Pickover, Bud	Captain

CadetShips

Cadet	Ship
Ash, Joshua	Frieda's Glory
Barker, Sally	Scrat
Cumin, Bil	Scrat
Farnsworth, Al	Frieda's Glory
Major, Major	Scrat
Pickover, Bud	Athena Ascendant

Figure 7-7

You can easily verify that these tables are in 1NF.

The CadetPositions table is in 2NF trivially because every field is part of the primary key.

The CadetShips table is in 2NF because the only field that is not part of the primary key (Ship) depends on the single primary key field (Cadet).

Third Normal Form (3NF)

A table is in 3NF if:

1. It is in 2NF.
2. It contains no transitive dependencies.

A *transitive dependency* is when one non-key field's value depends on another non-key field's value.

For example, suppose you and your friends decide to start a book club. To see what kinds of books people like, you put together the following table listing everyone's favorite books. It uses Person as the primary

key. (Again the Author field might violate 1NF if you consider it as containing multiple values: first and last name. For simplicity, and because you won't ever want to search for books written by authors with the first name ''Orson,'' I'll treat this as a single value.)

Person	Title	Author	Pages	Year
Amy	Support Your Local Wizard	Duane, Diane	473	1990
Becky	Three to Dorsai!	Dickson, Gordon	532	1975
Jon	Chronicles of the Black Company	Cook, Glen	704	2007
Ken	Three to Dorsai!	Dickson, Gordon	532	1975
Wendy	Support Your Local Wizard	Duane, Diane	473	1990

You can easily show that this table is 1NF. It uses a single field as primary key so every field in the table depends on the entire primary key, so it's also 2NF. (Each row represents that Person's favorite book so every field must depend on that Person.)

However, this table contains a lot of duplication, so it is subject to modification anomalies. (At this point you probably knew that!) If you discover that the Year for *Support Your Local Wizard* is wrong and fix it in row 1, it will conflict with the last row.

It's also subject to deletion anomalies (if Jon insults everyone and gets kicked out of the group so you remove the third row, you lose all of the information about *Chronicles of the Black Company*) and insertion anomalies (you cannot save Title, Author, Pages, and Year information about a book unless it's someone's favorite, and you cannot allow someone to join the group until he or she decides on a favorite).

The problem here is that some of the fields are related to others. In this example, Author, Pages, and Year are related to Title. If you know a book's Title, you could look up its Author, Pages, and Year.

In this example, the primary key Person doesn't exactly drive the Author, Pages, and Year fields. Instead it selects the Person's favorite Title and then Title determines the other values. This is a *transitive dependency*. Title depends on Person and the other fields depend on Title.

The main clue that there is a transitive dependency is that there are lots of duplicate values in the table.

You can fix this problem in a way similar to the way you put a table into 2NF: find the fields that are causing the problem and pull them into a separate table. Add an extra field to contain the original field on which those were dependent so you can link back to the original table.

In this case, you could make a Books table to hold the Author, Pages, and Year fields. You would then add a Title field to link the new records back to the original table.

Figure 7-8 shows a relational model for the new design.

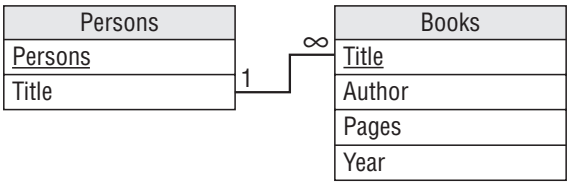
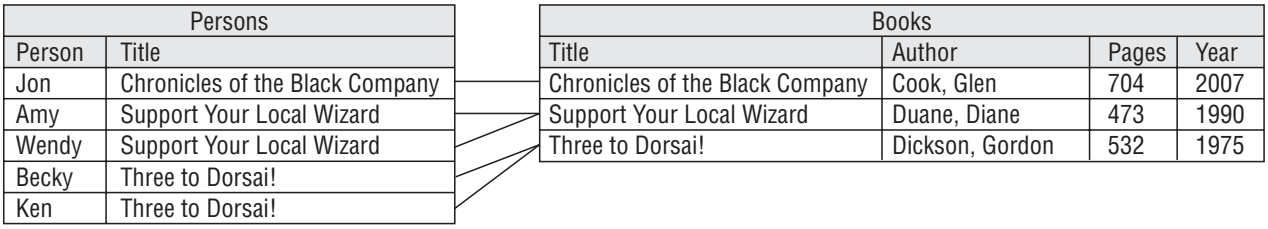


Figure 7-8

Figure 7-9 shows the new tables containing the original data.



Persons

Person	Title
Jon	Chronicles of the Black Company
Amy	Support Your Local Wizard
Wendy	Support Your Local Wizard
Becky	Three to Dorsai!
Ken	Three to Dorsai!

Books

Title	Author	Pages	Year
Chronicles of the Black Company	Cook, Glen	704	2007
Support Your Local Wizard	Duane, Diane	473	1990
Three to Dorsai!	Dickson, Gordon	532	1975

Figure 7-9

Try It Out Arranging Data in the Third Normal Form

Suppose you're helping to organize the 19,524th Jedi Olympics. Mostly the contestants stand around bragging about how they don't need to use violence because the Force is strong in them. You also often hear the phrases, ''I was just following the Force,'' and ''The Force made me do it.''

But to keep television ratings up, there are some athletic events. The following table shows the day's schedule. It uses Contestant as the primary key.

Contestant	Time	Event	Venue
Boyce Codd	2:00	Monster Mayhem	Monster Pit
General Mills	1:30	Pebble Levitating	Windy Plains Arena
Master Plethora	4:00	X-wing Lifting	Windy Plains Arena
Master Tor	1:00	Monster Mayhem	Monster Pit
Glenn	5:00	Poker	Dark Force Casino
Xzktp! Krffzk	5:00	Poker	Dark Force Casino

As part of your data processing padawan training:

1. Describe the data anomalies that this table allows.
2. Put the table in 3NF. If some fields are dependent on other non-key fields, pull the dependent fields out into a new table.

How It Works

1. Describe the data anomalies that this table allows.

   This table allows update anomalies because it contains lots of repeated values. For example, if you changed the Venue for Monster Mayhem in row 1, it would conflict with the Venue for Monster Mayhem in row 4. It also allows deletion anomalies (if Master Plethora is caught metaclorian doping and he drops out, you lose the fact that X-wing Lifting occurs in the Windy Plains Arena) and insertion anomalies (you cannot add a new contestant without an event or an event without a contestant).

2. Put the table in 3NF. If some fields are dependent on other non-key fields, pull the dependent fields out into a new table. Pull the dependent Venue field out into a new table. Add the field it depends upon (Event) as a key to link back to the original table.

   The problem is that the Event and Venue fields are dependent on each other in some manner.

   The solution is to pull the dependent fields out and put them in a new table. Then add a field linking back to the field on which they depend. The next question is, ''Which of these two related fields should be the one that you leave in the original table to use as the link?'' Does Venue depend on Event? Or does Event depend on Venue? Or does it matter which one you consider dependent on the other?

   In this example, it does matter.

   Notice that Event determines Venue. In other words, each particular Event occurs in only one Venue, so if you know the Event, you know the Venue. For example, all Pebble Levitating events occur in Windy Plains Arena. This means Venue is dependent on Event.

   However, the reverse is not true. Venue does not determine Event. If you know the Venue, you do not necessarily know the Event. For example, the Windy Plains Arena is the venue for both Pebble Levitating and X-wing Lifting. This means Event is not dependent on Venue.

   (If there were a one-to-one mapping of Event to Venue, each field would determine the other so you could use either as the key field. Although to me it makes more intuitive sense to use Event as the key.)

   Figure 7-10 shows the new model.

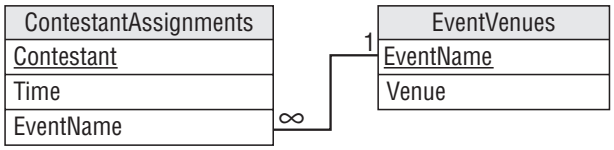


Figure 7-10

Figure 7-11 shows the tables containing the original data.

Contestants		
ContestantName	Time	Event
Boyce Codd	2:00	Monster Mayhem
Master Tor	1:00	Monster Mayhem
General Mills	1:30	Pebble Levitating
Glenn	5:00	Poker
Xzktp! Krffzk	5:00	Poker
Master Plethora	4:00	X-wing Lifting

EventVenues	
Event	Venue
Monster Mayhem	Monster Pit
Pebble Levitating	Windy Plains Arena
Poker	Dark Force Casino
X-wing Lifting	Windy Plains Arena

Figure 7-11

Stopping at Third Normal Form

Many database designers stop normalizing the database at 3NF because it provides the most bang for the buck. It's fairly easy to convert a database to 3NF and that level of normalization prevents the most common data anomalies. It stores separate data separately so you can add and remove pieces of information without destroying unrelated data. It also removes redundant data so the database isn't full of a zillion copies of the same information that waste space and make updating values difficult.

However, the database may still be vulnerable to some less common anomalies that are prevented by the more complete normalizations described in the following sections. These greater levels of normalization are rather technical and confusing. They can also lead to unnecessarily complicated data models that are hard to implement, hard to maintain, and hard to use. In some cases, they can give worse performance than less completely normalized designs.

Though you may not always need to use these super-normalized databases, it's still good to understand them and the problems that they prevent. Then you can decide whether those problems are a big enough issue to justify including them in your design. (Besides, they make great ice breakers at parties. ''Hey everyone! Let's see who can put the guest list in 4NF the fastest!'')

Boyce-Codd Normal Form (BCNF)

This one is kind of technical, so to understand it you need to know some terms.

Recall from Chapter 3 that a *superkey* is a set of fields that contain unique values. You can use a superkey to uniquely identify the records in a table.

Also recall that a *candidate key* is a minimal superkey. In other words, if you remove any of the fields from the candidate key, it won't be a superkey anymore.

Now for a new term. A *determinant* is a field that at least partly determines the value in another field. Note that the definition of 3NF worries about fields that are dependent on another field that is not part of the primary key. Now we're talking about fields that might be dependent on fields that *are* part of the primary key (or any candidate key).

A table is in BCNF if:

1. It is in 3NF.
2. Every determinant is a candidate key.

For example, suppose you are attending the Wizards, Knights, and Farriers Convention hosted by three nearby castles: Castle Blue, Castle Green, and Le Château du Chevalier Rouge. Each attendee must select a track: Wizard, Knight, or Farrier. Each castle hosts three seminars, one for each track.

During the conference, you may attend seminars at any of the three castles, but you can only attend the one for your track. That means if you pick a castle, I can deduce which session you will attend there.

Here's part of the attendee schedule. The letters in parentheses show the attendee and seminar tracks to make the table easier to read and they are not really part of this data.

Attendee	Castle	Seminar
Agress Profundus (w)	Green	Poisons for Fun and Profit (w)
Anabel (k)	Blue	Terrific Tilting (k)
Anabel (k)	Rouge	Clubs 'N Things (k)
Frock Smith (f)	Blue	Dealing with Difficult Destriers (f)
Lady Mismyth (w)	Green	Poisons for Fun and Profit (w)
Sten Bors (f)	Blue	Dealing with Difficult Destriers (f)
The Mighty Brak (k)	Green	Siege Engine Maintenance (k)
The Mighty Brak (k)	Rouge	Clubs 'N Things (k)

This table is susceptible to update anomalies because it contains duplicated data. If you moved the Poisons for Fun and Profit seminar to Castle Blue in the first record, it would contradict the Castle value in row 5.

It's also vulnerable to deletion anomalies because the relationship between Castle and Seminar is stored implicitly in this table. If you deleted the second record, you would lose the fact that the Terrific Tilting seminar is taking place in Castle Blue.

Finally, this table suffers from insertion anomalies. For example, you cannot create a record for a new seminar without assigning an attendee to it.

In short, this table has a problem because it has multiple overlapping candidate keys.

This table has two candidate keys: Attendee/Castle and Attendee/Seminar. Either of those combinations will uniquely identify a record.

The remaining combination, Castle/Seminar, cannot identify the Attendee so it's not a candidate key.

The Castle and Seminar fields have a dependency: Seminar determines Castle (but not vice versa). In other words, Seminar is a determinant of Castle.

This table is not in BCNF because Seminar is a determinant but is not a candidate key.

You can put this table in BCNF by pulling out the dependent data and linking it to the determinant. In this case, that means moving the Castle data into a new table and linking it back to its determinant, Seminar.

Figure 7-12 shows the new design.

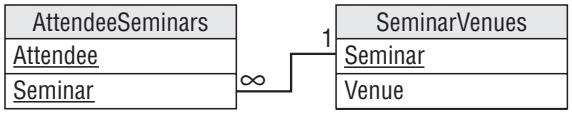
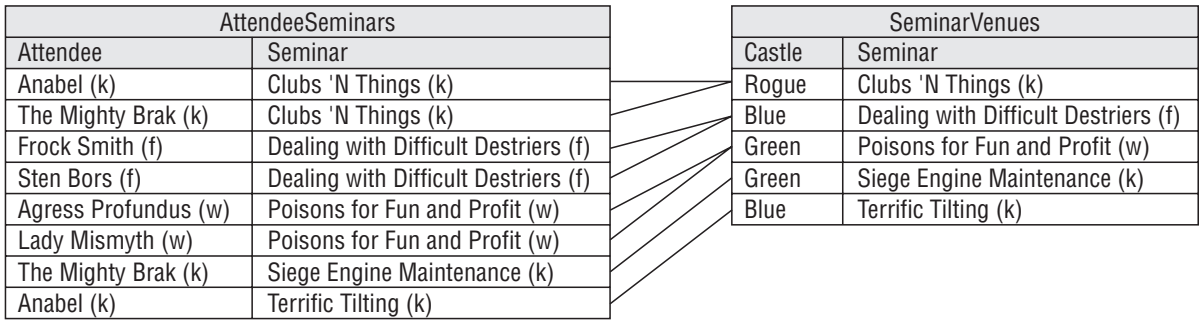


Figure 7-12

Figure 7-13 shows the new tables containing the original data.



AttendeeSeminars	
Attendee	Seminar
Anabel (k)	Clubs 'N Things (k)
The Mighty Brak (k)	Clubs 'N Things (k)
Frock Smith (f)	Dealing with Difficult Destriers (f)
Sten Bors (f)	Dealing with Difficult Destriers (f)
Agress Profundus (w)	Poisons for Fun and Profit (w)
Lady Mismyth (w)	Poisons for Fun and Profit (w)
The Mighty Brak (k)	Siege Engine Maintenance (k)
Anabel (k)	Terrific Tilting (k)

SeminarVenues	
Castle	Seminar
Rogue	Clubs 'N Things (k)
Blue	Dealing with Difficult Destriers (f)
Green	Poisons for Fun and Profit (w)
Green	Siege Engine Maintenance (k)
Blue	Terrific Tilting (k)

Figure 7-13

Now you can move the Poisons for Fun and Profit seminar to Castle Blue by changing a single record in the SeminarVenues table. You can delete Anabel's record for Terrific Tilting without losing the fact that Terrific Tilting takes place in Castle Blue because that information is in the SeminarVenues table. Finally, you can add a new record to the SeminarVenues table without assigning any attendees to it.

For another example, suppose you have an Employees table with columns EmployeeId, FirstName, LastName, SocialSecurityNumber, and Phone. Assume you don't need to worry about weird special cases such as roommates sharing a phone number or multiple employees with the same name.

This table has several determinants. For example, EmployeeId determines every other field's value. If you know an employee's ID, then all of the other values are fixed. This doesn't violate BCNF because EmployeeId is also a candidate key.

Similarly SocialSecurityNumber and Phone are each determinants of all of the other fields. Fortunately they, too, are candidate keys.

So far so good. Now for a stranger case. The combination FirstName/LastName is a determinant for all of the other fields. If you know an employee's first and last names, the corresponding EmployeeId, SocialSecurityNumber, and Phone values are set. Fortunately FirstName/LastName is also a candidate key so even that doesn't break the table's BCNF-ness.

This table is in BCNF because every determinant is also a candidate key. Intuitively the table is in BCNF because it represents a single entity: an employee.

The previous example was not in BCNF because it represented two concepts at the same time: attendees and the seminars they're attending, and the locations of the seminars. We solved that problem by splitting the table into two tables that each represented only one of those concepts.

Generally if every table represents a single concept or entity, it will be in pretty good shape. It's when you ask a table to do too much that you run into problems.

Try It Out Arranging Data in the BCNF

Consider an EmployeeAssignments table with the fields EmployeeId, FirstName, LastName, and Project. Each employee can be assigned to multiple projects and each project can have multiple employees. Ignore weirdnesses such as two employees having the same name.

To become a Data Wizard:

1. Explain why this table is not in BCNF. (Find a determinant that this not also a candidate key.)
2. Describe data anomalies that might befall this table.
3. Put this table in BCNF by pulling the FirstName and LastName fields out of the table and moving them into a new EmployeeData table. Add an EmployeeId field to link the new table's records back to the original records.

How It Works

1. Explain why this table is not in BCNF. (Find a determinant that this not also a candidate key.)

   Intuitively the problem with this table is that it includes two different concepts. It contains multiple pieces of employee data (EmployeeId, FirstName, LastName) together with employee project assignment data.

   More technically, this table's candidate keys are EmployeeId/Project and FirstName/LastName/Project. (Take a few minutes to verify that these combinations specify the records uniquely, that you cannot remove any fields from them, and that the remaining combination EmployeeId/FirstName/LastName doesn't work.)

   The problem occurs because these two candidate keys are partially overlapping. If you subtract out the overlapping field (Project), you get EmployeeId and FirstName/LastName. These are in the two candidate keys because they specify the same thing: the employee. That means they are determinants of each other. Unfortunately neither of these is a candidate key by itself so the table is not in BCNF.

2. Describe data anomalies that might befall this table.

   Suppose an employee is involved with multiple projects. If you change the employee's FirstName in one of that employee's rows, it will contradict the FirstName in the employee's other rows. This is a modification anomaly.

   Suppose you delete the only record containing employee Milton Waddams. You no longer have a record of his EmployeeId. In fact, you no longer have a record of him at all. (Perhaps that's the way he got deleted from the database in the movie *Office Space*.) This is a deletion anomaly.

   You also can't add an employee record without assigning the employee to a project, and you can't create a project without assigning an employee to it. These are insertion anomalies.

3. Explain how to put this table in BCNF.

The solution is to move one of the dependent fields into a new table. In this case, you could pull the FirstName and LastName fields out of the table and move them into a new EmployeeData table. Add an EmployeeId field to link the new table's records back to the original records.

Figure 7-14 shows the new model.

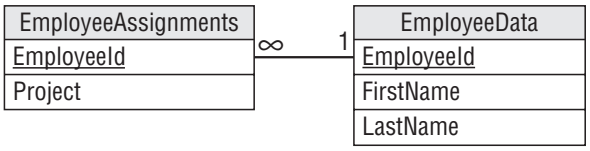


Figure 7-14

Fourth Normal Form (4NF)

Suppose you run a home fixit service. Each employee has a set of skills and each drives a particular truck that contains useful special equipment. They are all qualified to use any of the equipment. The following table shows a really bad attempt to store this information.

Employee	Skills	Tools
Gina Harris	Electric, Plumbing	Chop saw, Impact hammer
Pease Marks	Electric, Tile	Chain saw
Rick Shaw	Plumbing	Milling machine, Lathe

You should instantly notice that this table isn't even in 1NF because the Skills and Tools columns contain multiple values.

The following table shows an improved version. Here each row holds only one skill and tool.

Employee	Skill	Tool
Gina Harris	Electric	Chop saw
Gina Harris	Electric	Impact hammer
Gina Harris	Plumbing	Chop saw
Gina Harris	Plumbing	Impact hammer
Pease Marks	Electric	Chain saw
Pease Marks	Tile	Chain saw
Rick Shaw	Plumbing	Milling machine
Rick Shaw	Plumbing	Lathe

Unfortunately, to capture all of the data about each employee, this table must include a lot of duplication. To record the fact that Gina Harris has the electric and plumbing skills, and that her truck contains a chop saw and an impact hammer, you need four rows showing the four possible combinations of values.

In general, if an employee has S skills and T tools, the table would need S × T rows to hold all of the combinations.

This leads to the usual assortment of problems. If you modify the first row's Skill to Tile, it contradicts the second row, causing a modification anomaly. If Gina loses her impact hammer, you must delete two rows to prevent inconsistencies. If Gina takes classes in Painting, you need to add two new rows to cover all of the new combinations. If she then decides to add a spray gun, too, you need to add three more rows.

Something strange is definitely going on here. And yet this table is in BCNF!

You can easily verify that it's in 1NF.

Next note that every field must be part of the table's primary key because there can be duplicates of every other combination.

The table is in 2NF because all of the non-key fields (there are none) depend on all of the key fields (all of them). It's in 3NF because there are no transitive dependencies (every field is in the primary key so there no field is dependent on a non-key field). It's in BCNF because every determinant is a candidate key (the only determinant is Employee/Skill/Tool, which is also the only candidate key).

In this table, the problem arises because Employee implies Skill and Employee implies Tool but Skill and Tool are independent. This situation is called an *unrelated multi-valued dependency*.

A table is in 4NF if:

1. It is in BCNF.
2. It does not contain an unrelated multi-valued dependency.

A particular Employee leads to multiple Skills and for any given Skill there can be many Employees, so there is a many-to-many relationship between Employee and Skill. Similarly there is a many-to-many relationship between Employee and Tool. (Note that there is no relationship between Skill and Tool.)

Figure 7-15 shows an ER diagram for the entities involved: Employee, Skill, and Tool.

The solution to the problem is to find the field that drives the unrelated multi-valued dependency. In this case, Employee is the central field. It's in the middle of the ER diagram shown in Figure 7-15 and it forms one part of each of the many-to-many relationships.

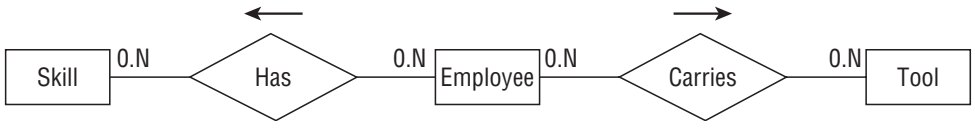


Figure 7-15

To fix the table, pull one of the other fields out into a new table. Add the central field (Employee) to link the new records back to the original ones.

Figure 7-16 shows the new model.

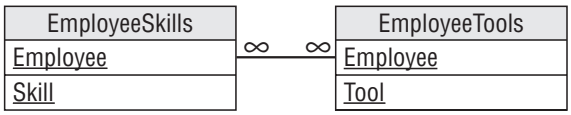
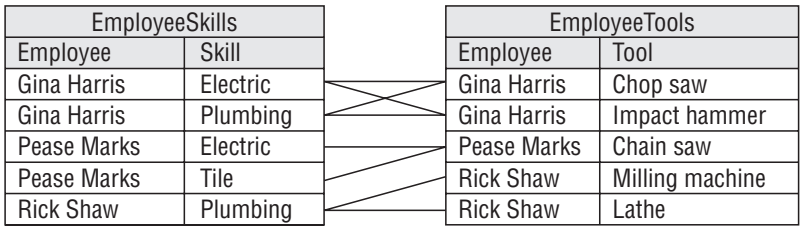


Figure 7-16

Figure 7-17 shows the original data in the new tables.



EmployeeSkills

Employee	Skill
Gina Harris	Electric
Gina Harris	Plumbing
Pease Marks	Electric
Pease Marks	Tile
Rick Shaw	Plumbing

EmployeeTools

Employee	Tool
Gina Harris	Chop saw
Gina Harris	Impact hammer
Pease Marks	Chain saw
Rick Shaw	Milling machine
Rick Shaw	Lathe

Figure 7-17

Try It Out Arranging Data in Fourth Normal Form

Consider the following artist's directory that lists artist names, genres, and shows they will attend this year.

Artist	Genre	Show
Ben Winsh	Metalwork	Makers of the Lost Art
Ben Winsh	Metalwork	Tribal Confusion
Harriette Laff	Textile	Fuzzy Mountain Alpaca Festival
Harriette Laff	Textile	Tribal Confusion
Harriette Laff	Sculpture	Fuzzy Mountain Alpaca Festival
Harriette Laff	Sculpture	Tribal Confusion
Mark Winslow	Sculpture	Green Mountain Arts Festival

A true database artist should be able to:

1. Identify the table's unrelated multi-valued dependency.
2. Draw an ER diagram showing the table's many-to-many relationships.

3. Put the table into 4NF by pulling the Show field out into a new table and adding an Artist column to link back to the original records. Then draw a relational model for the result.
4. Display the new tables with their data.

How It Works

In this example, there is a many-to-many relationship between Genre and Artist, and a second many-to-many relationship between Show and Artist.

1. Identify the table's unrelated multi-valued dependency.

   The Artist determines Genre and Show but Genre and Show are unrelated.

2. Draw an ER diagram showing the table's many-to-many relationships.

   Figure 7-18 shows an ER diagram for this situation.

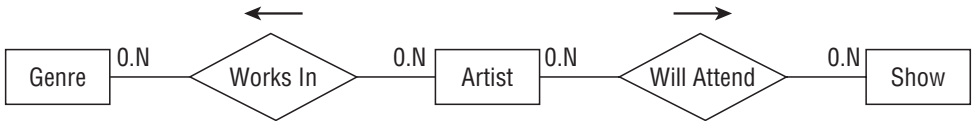


Figure 7-18

3. Put the table into 4NF and draw a relational model for the result.

   The central entity is Artist. One solution to this puzzle is to pull the Show field out into a new table and add an Artist column to link back to the original records. Figure 7-19 shows the result.

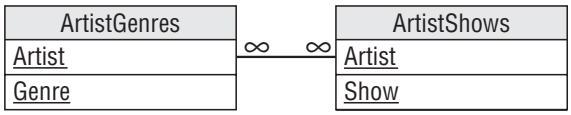
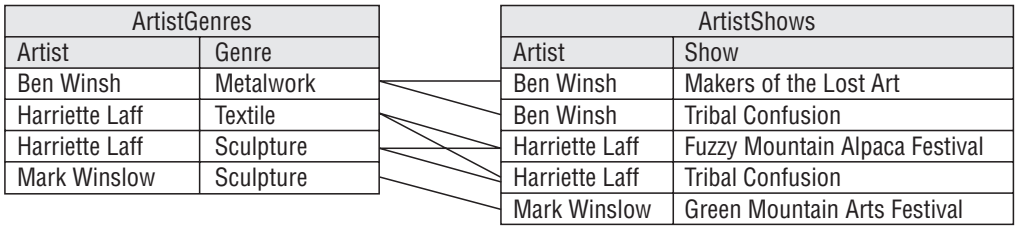


Figure 7-19

4. Display the new tables with their data.

   Figure 7-20 shows the new tables holding the original data.



ArtistGenres	
Artist	Genre
Ben Winsh	Metalwork
Harriette Laff	Textile
Harriette Laff	Sculpture
Mark Winslow	Sculpture

ArtistShows	
Artist	Show
Ben Winsh	Makers of the Lost Art
Ben Winsh	Tribal Confusion
Harriette Laff	Fuzzy Mountain Alpaca Festival
Harriette Laff	Tribal Confusion
Mark Winslow	Green Mountain Arts Festival

Figure 7-20

Fifth Normal Form (5NF)

A table is in 5NF (also called ''Project-Join Normal Form'') if:

1. It is in 4NF.
2. It contains no related multi-valued dependencies.

For example, suppose you run an auto repair shop. The grease monkeys who work there may be certified to work on particular makes of vehicles (Honda, Hummer, Yugo) and on particular types of engines (gas, diesel, hybrid, matter-antimatter).

If a grease monkey is certified for a particular make and for a particular engine, that person *must* provide service for that make and engine (if that combination exists). For example, suppose Joe Quark is certified to repair Hondas and Diesel. Then he must be able to repair Diesel engines made by Honda.

Now consider the following table showing which grease monkey can repair which combinations of make and engine.

GreaseMonkey	Make	Engine
Cindy Oyle	Honda	Gas
Cindy Oyle	Hummer	Gas
Eric Wander	Honda	Gas
Eric Wander	Honda	Hybrid
Eric Wander	Hummer	Gas
Joe Quark	Honda	Diesel
Joe Quark	Honda	Gas
Joe Quark	Honda	Hybrid
Joe Quark	Toyota	Diesel
Joe Quark	Toyota	Gas
Joe Quark	Toyota	Hybrid

In this case, GreaseMonkey determines Make. For a given GreaseMonkey, there are certain Makes that this person can repair.

Similarly, GreaseMonkey determines Engine. For a given GreaseMonkey, there are certain Engines that this person can repair.

Up to this point, the table is very similar to the Employee/Skill/Tool table described in the previous section about 4NF. Here comes the difference.

In the Employee/Skill/Tool table, Skill and Tool were unrelated. In this table, however, Make and Engine are related. For example, Eric Wander is certified in the Makes Honda and Hummer. He is also certified in the Engines Gas and Hybrid. The rules state that he must repair Gas and Hybrid engines for Honda and Hummer vehicles, if they provide those Makes. But Hummer doesn't make a hybrid, so Eric doesn't need to service that combination.

There's the dependency between Make and Engine. While the GreaseMonkey determines the Make and Engine, Make also influences Engine.

So how can you remove this dependency? Break the single table into three new tables that record the three different relationships: GreaseMonkey/Make, GreaseMonkey/Engine, and Make/Engine.

Figure 7-21 shows the new relational model.

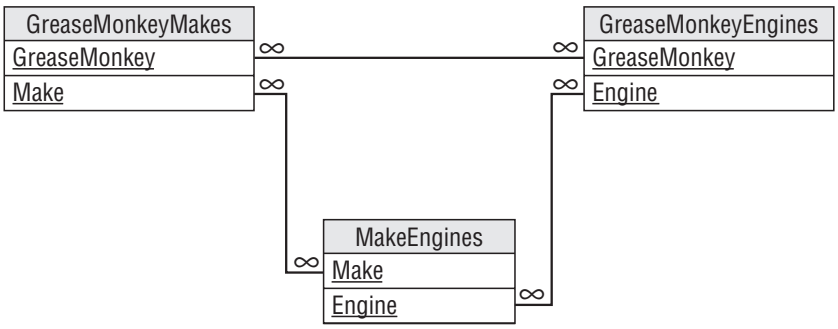


Figure 7-21

Figure 7-22 shows the new tables holding the original data. I haven't drawn lines connecting related records because it would make a big mess.

GreaseMonkey	Make
Cindy Oyle	Honda
Cindy Oyle	Hummer
Eric Wander	Honda
Eric Wander	Hummer
Joe Quark	Honda
Joe Quark	Toyota

GreaseMonkey	Engine
Cindy Oyle	Gas
Eric Wander	Gas
Eric Wander	Hybrid
Joe Quark	Diesel
Joe Quark	Gas
Joe Quark	Hybrid

Make	Engine
Honda	Diesel
Honda	Gas
Honda	Hybrid
Hummer	Gas
Hummer	Diesel
Toyota	Diesel
Toyota	Gas
Toyota	Hybrid

Figure 7-22

Try It Out — Working with the Fifth Normal Form

Remember the artist's directory from the previous sections about 4NF? The rules have changed slightly. The directory still lists artist names, genres, and shows they will attend, but now each show allows only certain genres. Now the Fuzzy Mountain Alpaca Festival includes only textile arts and Tribal Confusion includes only metalwork and sculpture. (Also, Ben Winsh started making sculptures.)

Here's the new schedule:

Artist	Genre	Show
Ben Winsh	Metalwork	Makers of the Lost Art
Ben Winsh	Metalwork	Tribal Confusion
Ben Winsh	Sculpture	Makers of the Lost Art
Harriette Laff	Textile	Fuzzy Mountain Alpaca Festival
Harriette Laff	Sculpture	Tribal Confusion
Mark Winslow	Sculpture	Green Mountain Arts Festival

To prove you're a true database artist:

1. Identify the table's related multi-valued dependency.
2. Draw an ER diagram showing the table's many-to-many relationships.
3. Put the table into 5NF and draw a relational model for the result.
4. Display the new tables with their data.

How It Works

1. Identify the table's related multi-valued dependency.

   Artist determines Genre, Artist determines Show, and Show determines Genre.

2. Draw an ER diagram showing the table's many-to-many relationships.

   Figure 7-23 shows an ER diagram for this situation.

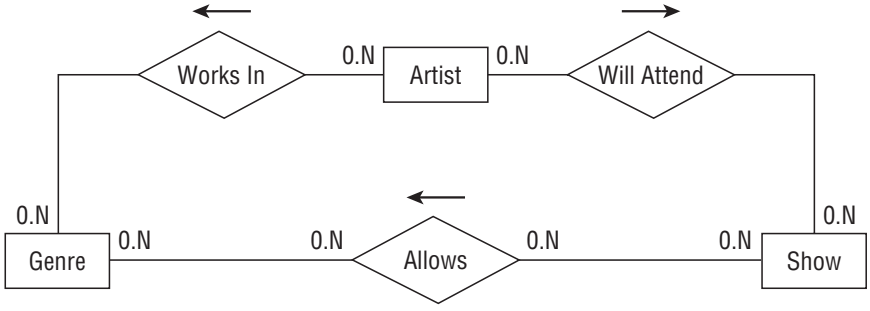


Figure 7-23

3. Put the table into 5NF and draw a relational model for the result.

   The new model should make separate tables to store the relationships between Artist and Genre, Artist and Show, and Show and Genre. Figure 7-24 shows this 5NF model.

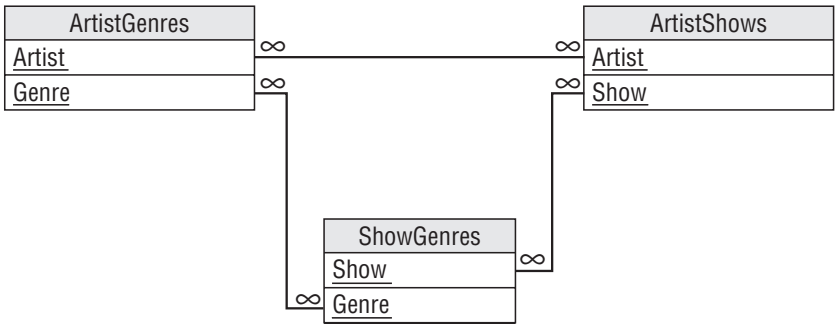


Figure 7-24

4. Display the new tables with their data.

   Figure 7-25 shows the new tables holding the original data.

ArtistGenres	
Artist	Genre
Ben Winsh	Metalwork
Ben Winsh	Sculpture
Harriette Laff	Sculpture
Harriette Laff	Textile
Mark Winslow	Sculpture

ArtistShows	
Artist	Show
Ben Winsh	Markers of the Lost Art
Ben Winsh	Trible Confusion
Harriette Laff	Fuzzy Mountain Alpaca Festival
Harriette Laff	Tribal Confusion
Mark Winslow	Green Mountain Arts Festival

ShowGenres	
Show	Genre
Fuzzy Mountain Alpaca Festival	Textile
Green Mountain Arts Festival	Sculpture
Markers of the Lost Art	Metalwork
Markers of the Lost Art	Sculpture
Tribal Confusion	Metalwork
Tribal Confusion	Sculpture

Figure 7-25

Domain/Key Normal Form (DKNF)

A table is in DKNF if:

1. The table contains no constraints except domain constraints and key constraints.

In other words, a table is on DKNF if every constraint is a consequence of domain and key constraints.

Recall from Chapter 3 that a field's domain consists of its allowed values. A domain constraint simply means that a field has a value that is in its domain. It's easy to check that a domain constraint is satisfied by simply examining all of the field's values.

A key constraint means the values in the fields that make up a key are unique.

So if a table is in DKNF, to validate all constraints on the data it is sufficient to validate the domain constraints and key constraints.

For example, consider a typical Employees table with fields FirstName, LastName, Street, City, State, and Zip. There is a hidden constraint between Street/City/State and Zip because a particular Street/City/State defines a ZIP Code and a ZIP Code defines City/State. You could validate new

addresses in a table-level check constraint that looked up Street/City/State/Zip to make sure it was a valid combination.

This table contains a constraint that is neither a domain constraint nor a key constraint so it is not in DKNF.

You can make the table DKNF by simply removing the Zip field. Now instead of validating a new Street/City/State/Zip, you look up the address's ZIP Code whenever you need it. (You would use whatever method you had been using before to validate the ZIP Code. For example, if the table-level check constraint was looking it up in a table, you would use that table to look it up now.)

It can be proven (although not by me) that a database in DKNF is immune to all data anomalies. So why would you bother with lesser forms of normalization? Mostly because it can be confusing and difficult to build a database in DKNF. (For example, there are about 45,000 U.S. ZIP Codes and they are constantly changing. That would make a whopper of a table and a maintenance nightmare!)

Lesser forms of normalization also usually give good enough results for most practical database applications so there's no need for DKNF under most circumstances.

However, it's nice to know what DKNF means so you won't feel left out at cocktail parties when everyone else is talking about DKNF.

Try It Out — Arranging Data in the Domain/Key Normal Form

Consider the following student/class assignment table:

Student	Class	Department
Annette Silver	First Order Logic	Philosophy
Annette Silver	Real Analysis II	Mathematics
Janet Wilkes	Fluid Dynamics I	Physics
Janet Wilkes	Real Analysis II	Mathematics
Mark Hardaway	First Order Logic	Philosophy
Mark Hardaway	Topology I	Mathematics

This table has a dependency between Class and Department because each class is within a single department.

To ace this class:

1. Explain why this table is not in DKNF.
2. Explain how you could put the table in DKNF.
3. Draw a relational model for the result.
4. Show the table(s) you create containing the original data.

How It Works

1. Explain why table is not in DKNF.

   The dependency between Class and Department is not a domain constraint or a key constraint.

2. Explain how you could put the table in DKNF.

   Pull the Department data out of the table and make a new table to relate Department and Class. Now instead of storing the Department data in the original table, you store only the Student and Class. When you need to know the Department, you look it up in the new table.

3. Draw a relational model for the result.

   Figure 7-26 shows a relational model in DKNF.

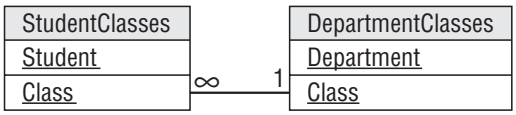
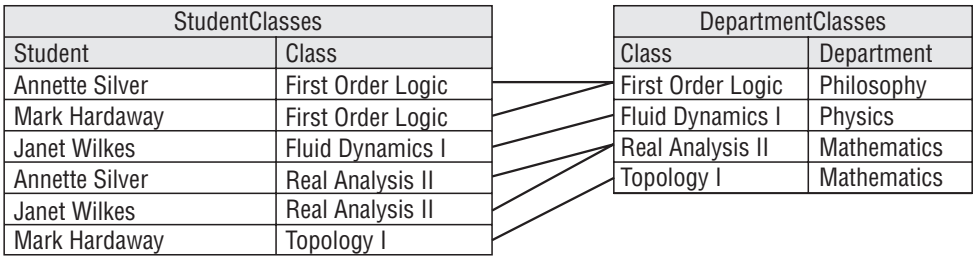


Figure 7-26

4. Show the table(s) you create containing the original data.

   Figure 7-27 shows the new tables holding the original data.



StudentClasses	
Student	Class
Annette Silver	First Order Logic
Mark Hardaway	First Order Logic
Janet Wilkes	Fluid Dynamics I
Annette Silver	Real Analysis II
Janet Wilkes	Real Analysis II
Mark Hardaway	Topology I

DepartmentClasses	
Class	Department
First Order Logic	Philosophy
Fluid Dynamics I	Physics
Real Analysis II	Mathematics
Topology I	Mathematics

Figure 7-27

Essential Redundancy

One of the major data anomaly themes is redundancy. If a table contains a lot of redundant data, it's probably vulnerable to data anomalies, particularly modification anomalies.

However, this is not true if the redundant data is in keys. For example, look again at Figure 7-27. The StudentClasses table contains several repeated student names and class names. Similarly the DepartmentClasses table contains repeated Department names. You might think these create a modification anomaly hazard.

In fact, if you look at Figure 7-26, you'll see that all of these fields are all part of the tables' keys. Their repetition is necessary to represent the data that the tables hold. For example, the repeated Department values in the DepartmentClasses table are part of the data showing which departments hold which classes. Similarly the repeated Student and Class data in the StudentClasses table is needed to represent the students' class assignments.

Though these repeated values are necessary, they do create a different potential problem. Suppose you want to change the name of the class ''Real Analysis II'' to ''Getting Real, the Sequel'' because you think it will make more students sign up for it.

Unfortunately you're not supposed to change the value of a primary key. If you could change the value, you might need to update a large number of records and that could lead to problems like any other modification anomaly would.

The real problem here is that you decided that the class's name should be changed. Because you can't change key values, the solution is to use something else instead of the class's name for the key. Typically a database will use an arbitrary ID number to represent the entity and then move the real data (in this case the class's name) into another table. Because the ID is arbitrary, you should never need to change it.

Figure 7-28 shows one way to replace these fields used as keys with arbitrary IDs that you will never need to change.

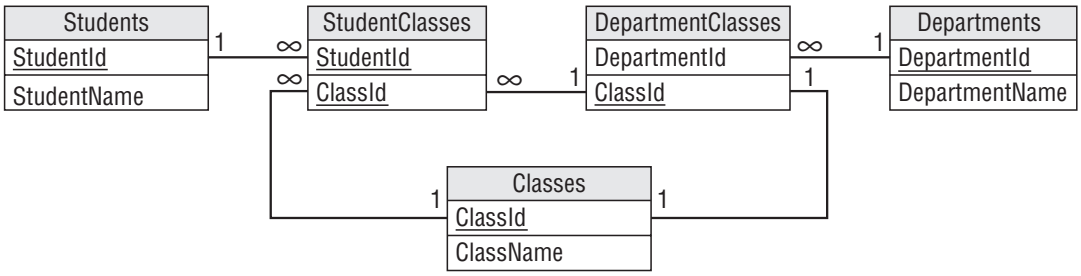


Figure 7-28

For bonus points, you can notice that you can combine the DepartmentClasses and Classes tables to give the simpler model shown in Figure 7-29.

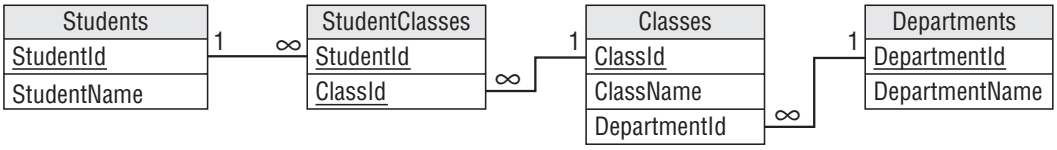


Figure 7-29

This is a reasonable model. Each table represents a single, well-defined entity (student, class, department, and the relationship between students and classes).

The Best Level of Normalization

Domain/Key Normal Form makes a database provably immune to data anomalies, but it can be tricky to implement and it's not usually necessary. The higher levels of normalization may also require you to split tables into many pieces, making it harder and more time-consuming to reassemble the pieces when you need them.

For example, the previous section explained that an Employees table containing Street, City, State, and Zip fields was not in DKNF because the Street/City/State combination duplicates some of the information in the Zip field. The solution was to remove the Zip field and to look up an employee's ZIP Code whenever it was needed. To see whether this change is reasonable, look at the costs and benefits.

The extra cost is that you must perform an extra lookup every time you need to display an employee's address with the ZIP Code. Just about any time you display an employee's address you will need the ZIP Code, so you will probably perform this lookup a lot.

The benefit is that it makes the data less susceptible to data modification anomalies if you need to change a ZIP Code value. But how often do ZIP Codes change? On a national level, ZIP Codes change all the time but unless you have millions of employees, your employees' ZIP Codes probably won't change all that frequently. This seems like a rare problem. It is probably better to use a table-level check constraint to validate the Street/City/State/Zip combination when the employee's data is created or modified and then leave well enough alone. On the rare occasion when a ZIP Code really does change, you can do the extra work to update all of the employees' ZIP Codes.

Often 3NF reduces the chances of anomalies to a reasonable level without requiring confusing and complex modifications to the database's structure.

When you design your database, put it in 3NF. Then look for redundancy that could lead to anomalies. If the kinds of changes that would cause problems in your application seem like they may happen often, then you can think about using the more advanced normalizations. If those sorts of modifications seem rare, you may prefer to leave the database less normalized.

Summary

Normalization is the process of rearranging a database's table designs to prevent certain kinds of data anomalies. Different levels of normalization protect against different kinds of errors.

If every table represents a single, clearly defined entity, you've already gone a long way toward making your database safe from data anomalies. You can use normalization to further safeguard the database.

In this chapter you learned about:

- Different kinds of anomalies that can afflict a database.
- Different levels of normalization and the anomalies they prevent.
- Methods for normalizing database tables.

The next chapter discusses another way you can reduce the chances of errors entering a database. It explains design techniques other than normalization that can make it safer for a software application to manipulate the database.

Before you move on to Chapter 8, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

Exercises

1. Suppose a student contact list contains the fields Name, Email, Email, Phone1, PhoneType1, Phone2, PhoneType2, and MajorOrSchool. The student's preferred email address is listed in the first Email field. Similarly the preferred phone number is in the Phone1 field. The MajorOrSchool field stores the student's major if he or she has picked one and the student's school (School of Engineering, School of Liberal Arts, School of Metaphysics, and so forth) otherwise.

   a. Explain why this list isn't in 1NF.

   b. Convert it into 1NF. Draw a relational diagram for it.

2. Consider the following table that lists errands that you need to run. The list shows the most important items at the top.

Location	Items
Grocery store	milk, eggs, bananas
Office supply store	paper, pencils, divining rod
Post Office	stamps
Computer store	flash drive, 8'' floppy disks

   a. Explain why this list isn't in 1NF.

   b. Convert this list into a single 1NF table. Be sure to define a primary key.

3. For the table you built for Exercise 2:

   a. Explain why the table isn't in 2NF.

   b. Convert the table into 2NF.

4. Consider the following employee assignments table, which uses Employee as its primary key.

Employee	Project	Department
Alice Most	Work Assignment	Network Lab
Bill Michaels	Network Routing	Network Lab
Deanna Fole	Survey Design	Human Factors
Josh Farfar	Work Assignment	Network Lab

Employee	Project	Department
Julie Wish	Survey Design	Human Factors
Mandy Ponem	Network Routing	Network Lab
Mike Mix	New Services Analysis	Human Factors

a. Explain why the table isn't in 3NF.

b. Convert the table into 3NF.

5. One of your friends has decided to start a breakfast club. What each member can cook depends on his or her skills and equipment. Your friend built the following table to record all of the combinations.

Person	Food	Tool
Alice	Muffins	Muffin tin
Alice	Muffins	Omelet pan
Alice	Muffins	Pancake griddle
Alice	Omelets	Muffin tin
Alice	Omelets	Omelet pan
Alice	Omelets	Pancake griddle
Alice	Pancakes	Muffin tin
Alice	Pancakes	Omelet pan
Alice	Pancakes	Pancake griddle
Bob	Muffins	Omelet pan
Bob	Omelets	Omelet pan
Bob	Pancakes	Omelet pan
Cyndi	Omelets	Muffin tin
Cyndi	Omelets	Pancake griddle

Fortunately you know all about normalization, so help your friend by:

a. Explaining why the table isn't in 5NF.

b. Converting the table into 5NF.

6. In Figure 7-30, match the normal forms on the left with their corresponding rules on the right.

First Normal Form	• It contains no transitive dependencies.
Second Normal Form	• It does not contain an unrelated multi-valued dependency.
Third Normal Form	• Each column must have a unique name. • The order of the rows and columns doesn't matter. • Each column must have a single data type. • No two rows can contain identical values. • Each column must contain a single value. • Columns cannot contain repeating groups.
Boyce/Codd Normal form	• All of the non-key fields depend on all of the key fields.
Fourth Normal form	• Every determinant is a candidate key.
Fifth Normal form	• It contains no related multi-valued dependencies.
Domain/Key Normal Form	• The table contains no constraints except domain constraints and key constraints.

Figure 7-30

8

Designing Databases to Support Software Applications

The previous chapters showed how to gather user requirements, build a database model, and normalize the database to improve its performance and robustness. Those chapters showed how to look at the database from the customers' perspective, from the end user's perspective, and from a database normalization perspective, but there's one other point of view that you should consider before you open your database product and start slapping tables together: the programmer's.

You may not be responsible for writing a program to work with a database. The database may not ever directly interact with a program (although that's rare). In any case, the techniques that you would use to make a database easier for a program to use often apply to other situations. Learning how to help a database support software applications can make the database easier to use in general.

In this chapter you learn:

- Steps you can take to make the database more efficient in practical use.
- Methods for making validation easier in the user interface.
- Ways to easily manage non-searchable data.

This chapter describes several things that you can do to make the database more program-friendly.

A few of these ideas (such as multi-tier architecture) have been covered in earlier chapters. They are repeated in brief here to tie them together with other programming-related topics, but you should refer to the original chapters for more detailed information.

Plan Ahead

Any complicated task benefits from prior planning, and software development is no exception. It has been shown that the longer an error remains in a project the longer it takes to fix it. Database

design occurs very early in the development process, so mistakes made here can be very costly. A badly designed database provides the perfect proving ground for the expression, ''Act in haste, repent at leisure.'' Do your work up front or be prepared to spend a lot of extra time fixing mistakes.

Practically all later development depends directly or indirectly on the database design. The database design acts like a building's foundation. If you build a weak foundation, the building on top of it will be wobbly and unsound. The Leaning Tower of Pisa is a beautiful result built on a weak foundation, but it's the result of luck more than planning and people have spent hundreds of years trying to keep it from falling down. If you try to build on a wobbly foundation, you're more likely to end up with a pile of broken rubble than an interesting building.

After you get some experience with database design, it's very tempting to just start cranking out table and field definitions without any prior planning, but that's almost always a mistake. Don't immediately start building a database or even creating a relational object model. At least sketch out an ER diagram to better understand the entities that the database must represent before you start building.

Document Everything

Write everything down. This helps prevent disagreements about who promised what to whom. (''But you promised that the database could look up a customer's credit rating and Amazon password.'')

Good documentation also keeps everyone on the same wavelength. If you have done a good job of writing requirements, use cases, database models, design specifications, and all of the other paperwork that describes the system, the developers can scurry off into their own little burrows and start working on their parts of the system without fear of building components that won't work together.

You can make programmers' lives a lot easier if you specify table and field definitions in great detail. Write down the fields that belong in each table. Also write down each field's properties: name, data type, length, whether it can be null, string format (such as ''mm/dd/yyyy'' or ''###-####''), allowed ranges (1–100), default values, and other more complex constraints.

Programmers will need this information to figure out how to build the user interface and the code that sits behind it (and the middle tiers if you use a multi-tier architecture). Make sure the information is correct and complete at the start so the programmers don't need to make a bunch of changes later.

For example, suppose Price must be greater than $1.00. The programmers get started and build a whole slew of screens that assume Price is greater than $1.00. Now it turns out that you meant Price must be *at least* $1.00 not *greater than* $1.00. This is a trivial change to the design and to the database but the programmers will need to go fix the whole bunch of screens that contain the incorrect assumption. (Actually, good programming practices will minimize the problem, but you can't assume everyone is a top-notch developer.)

After you have all of this information, don't just put it in the database and assume that everyone can get the information from there. Believe it or not, some developers don't know how to use every conceivable type of database product (MySQL, SQL Server, Access, Informix, Oracle, DB2, Paradox, Sybase, PostgreSQL, FoxPro — there are hundreds) so making them dig this information out of the database can be a time-consuming hassle. Besides, writing it all up gives you something to show management to prove that you're making progress.

Consider Multi-Tier Architecture

A multi-tier architecture can help isolate the database and user interface development so programmers and database developers can work independently. This approach can also make a database more flexible and amenable to change. Unless you're a project architect, you probably can't decide to use this kind of architecture by yourself but you can make sure it is considered. See Chapter 6 for more details about multi-tier architectures.

Convert Domains into Tables

It's easy enough to validate a field against its domain by using check constraints. For example, suppose you know that the Addresses table's State field must always hold one of the values CA, OR, or WA. You can verify that a field contains one of those values with a field-level check constraint. In Access, you could set the State field's Validation Rule property to:

```
='CA' Or ='OR' Or ='WA'
```

Other databases use different syntax.

Although this is simple and it's easy for you to change, it's not easily visible to programmers building the application. That means they need to write those values into the code. Later if you change the list, the programmers need to change the code.

Even worse, someone needs to remember that the code needs to be changed! It's fairly common to change one part of an application and forget to make a corresponding change elsewhere. Those kinds of mistakes can lead to some bugs that are very hard bugs.

A better approach is to move the domain information into a new table. Create a States table and put the values CA, OR, and WA in it. Then make a foreign key constraint that requires the Addresses table's States field to allow only values that are in the States table. Programmers can query the database at run time to learn what values are allowed and can then do things such as making a combo box that allows only those choices. Now if you need to change the allowed values, you only need to update the States table's data and the program automatically picks up the change.

Wherever possible, convert database structure into data so everyone can read it easily.

Try It Out — Lookup Tables

Okay, this is really easy but it's important so bear with me. Suppose your Phoenician restaurant offers delivery service for customers with ZIP Codes between 02154 and 02159. In Access, you could validate the customer's Zip field with the following field-level check constraint:

```
>='02154' And <='02159'
```

Unfortunately that constraint is hidden from the programmers building the user interface. These checks may also not be in a form that's easy for the program to understand. Most programmers don't know how to open Access, read a field's check constraints, and parse an expression such as this one to figure out what it does.

How could you make this condition easier for programmers to discover and use at run time?

How It Works

Though reading and parsing check constraints is hard, it's fairly easy for a program to read the values from a table. The answer is to make a DeliveryZips table that lists the ZIP Codes in the delivery area:

Zip
02154
02155
02156
02157
02158
02159

This seems less elegant than the field-level check constraint but it's a lot easier for the program to understand.

Keep Tables Focused

If you keep each table well-focused, there will be fewer chances for misunderstandings. Different developers will have an easier time keeping each table's purpose straight so different parts of the application will use the data in the same (intended) way.

Modern object-oriented programming languages also use objects and classes that are very similar to the objects and classes used in semantic object modeling and that are similar to the entities and entity sets used by entity-relationship diagrams. If you do a good job of modeling, and keep object and table definitions well-focused, those models practically write the code by themselves. There are still plenty of other things for the programmers to do but at least they'll be able to make programming object models that closely resemble the database structure.

Use Three Kinds of Tables

One tip that can help you keep tables focused is to note that there are three basic kinds of tables. The first kind stores information about objects of a particular type. These hold the bulk of the application's data.

The second kind of table represents a link between two or more objects. For example, the association tables described in Chapter 5 represent a link between two types of objects.

Figure 8-1 shows an ER diagram to model employees and projects. Each employee can be assigned to multiple projects and each project can involve multiple employees. The EmployeeRoles table provides a

link between the other two object tables. It also stores additional information about the link. In this case, it stores the role that an employee played on each project.

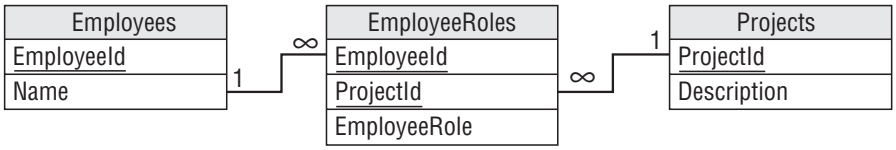


Figure 8-1

The third basic kind of table is a lookup table. These are tables created simply to use in foreign key constraints. For example, the States table described in the earlier section ''Convert Domains into Tables'' is a lookup table.

When you build a table, ask yourself whether it represents an object, a link, or a lookup. If you cannot decide or if the table represents more than one of those, the table's purpose may not be clearly defined.

Try It Out — Well-Focused Tables

Take a look at the following table of extra-terrestrial animals:

Animal	Size	Planet	PlanetaryMass
Hermaflamingo	Medium	Virgon 4	1.21
Shunkopotamus	Large	Dilbertopia	0.88
Mothalope	Medium	Xanth	0.01
Shunkopotamus	Large	Virgon 4	1.21
Platypus	Small	Australia	1.00

The table's primary key is the combination of Animal/Planet. (The PlanetaryMass field is measured in Earth masses.)

This isn't a very well-focused table.

1. What ideas is this table is trying to capture?
2. What types of ideas are these (object, link, or lookup)?
3. Suggest a better design that keeps the table's separate purposes separate.

How It Works

1. What ideas is this table is trying to capture?

   This table is trying to capture three different ideas: information about the animals (Animal and Size), information about planets (Planet and PlanetaryMass), and the associations between the

animals and planets. The fact that this table holds information about three different concepts is a sign that it is not well-focused.

(Also note this table is not in Second Normal Form because it has non-key fields that do not depend on the entire key. Recall that the primary key is Animal/Planet. The Size field depends on Animal but not Planet, and the PlanetaryMass field depends on Planet but not Animal.)

2. What types of ideas are these (object, link, or lookup)?

   Information about the animals and information about the planets represent objects. Information about the associations between animals and planets is a link.

3. Suggest a better design that keeps the table's separate purposes separate.

   The key is to move the three kinds of information into three different tables. Figure 8-2 shows one relational design that separates the three sets of information into three tables.

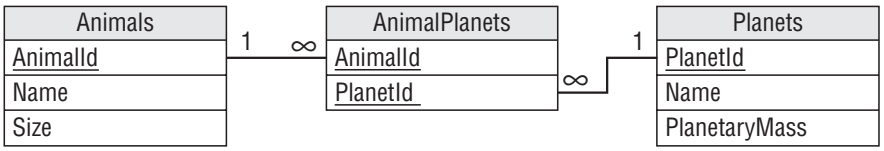


Figure 8-2

Figure 8-3 shows the tables holding their original data.

Animals

Size	Animal	AnimalId
Medium	Hermaflamingo	1
Large	Shunkopotamus	2
Medium	Mothalope	3
Small	Platypus	4

AnimalPlanets

AnimalId	PlanetId
1	101
2	102
3	103
2	101
4	104

Planets

PlanetId	HomePlanet	PlanetaryMass
101	Virgon 4	1.21
102	Dilbertopia	0.88
103	Xanth	0.01
104	Australia	1.00

Figure 8-3

Use Naming Conventions

Use consistent naming conventions when you name database objects. It doesn't matter too much what conventions you use as long as you use something consistent.

Some database developers prefix table names with tbl and field names with fld as in, ''The tblEmployees table contains the fldFirstName and fldLastName fields.'' For simple databases, I prefer to omit the prefixes because it's usually easy enough to tell which names represent objects (tables) and which represent attributes (fields).

Some developers also use a lnk prefix for tables that represent a link between objects as in, ''The lnkAnimalsPlanets table links tblAnimals and tblPlanets.'' These developers are also likely to use lu or lup as a prefix for lookup tables.

Some developers prefer to use UPPERCASE_LETTERS for table names and lowercase_letters for field names. Others use MixedCase for both.

Some prefer to make table names singular (the Employee table) and others prefer to make them plural (the Employees table).

As I said, it doesn't matter too much which of these conventions you use as long as you pick some conventions and stick to them.

However, there are three ''mandatory'' naming rules that you should follow or the other developers will snicker behind your back and openly mock you at parties.

First, don't use special characters in table names, field names, or anywhere else in database objects. For example, some databases (such as Access) allow you to include spaces in table and field names. For example, you can make a field named ''First Name.'' Those databases also provide some mechanism for making these weird names readable to the database. For example, in Access you need to use square brackets to surround a field with a name containing spaces as in ''[First Name].'' This produces hard-to-read expressions in check constraints and anywhere else you use the field. It also makes programmers using the field take similar annoying steps and that makes their code less readable, too.

Second, if two fields in different tables contain the same data, give them the same name. For example, suppose you use an ID field to link the Employees table to the EmployeePhones table. Don't call this linking field Id in the Employees table and EmpId in the EmployeePhones table. That's just asking for trouble. Call the field EmployeeId in both tables. (A corollary to this rule is that you cannot name an ID field something vague such as Id. It may make sense in the main table such as Employees but that name won't make sense in a related table such as EmployeePhones.)

The third mandatory naming rule is to use meaningful names. Don't abbreviate to the point of obscurity. It shouldn't take a team of National Security Agency cryptographers to decipher a table's field names. StudPrfCrs is much harder to read than StudentPreferredCourses. Don't be afraid to spell things out so everyone can understand them. (The exception here seems to be the military where everyone would understand ''SecInt visited NavSpecWarGru'' but saying ''the Secretary of the Interior visited the Naval Special War Group'' would brand you as an outsider.)

The section ''Poor Naming Standards'' in Chapter 10 has more to say about naming conventions and includes a few links that you can follow to learn about some specific standards that you can adopt if you like.

Allow Some Redundant Data

Chapter 7 explained that it is not always best to normalize a database as completely as possible. The higher forms of normalization usually spread data out into tables that are linked by their fields. When a program needs to display that data, it must reassemble all of that scattered data and that can take some extra time.

For example, if you allow customers to have any number of phone numbers, email addresses, postal addresses, and contacts, then what seems to the user like a simple customer record is actually spread across the Customers, CustomerPhones, CustomerEmails, CustomerAddresses, and CustomerContacts tables.

In some cases, it may be better to restrict the database's flexibility somewhat to gain speed and simplicity. For example, if you allow the customers to have only two phone numbers, one email address, and one contact, you cut the number of tables that make up the customer's information from five to two. The database won't be as infinitely flexible and it won't be quite as completely normalized, but it will be easier to use.

Usually it's also best not to store the same data in multiple ways because that can lead to modification anomalies. For example, you don't really need a Balance field in a customer's record if you can recalculate the balance by adding up all of the customer's credits and debits.

However, suppose you're running an Internet service that allows customers to download music so a typical customer makes dozens or even hundreds of purchases a month. After a year or two, adding up all of a customer's credits and debits could be time consuming. In this case, you might be better off adding a Balance field to the customer's record and exercising a little extra caution when updating the account.

Don't Squeeze in Everything

Just because you're using a database doesn't mean every piece of data that the system uses must be squeezed in there somewhere. Databases provide tools for storing and retrieving some strange pieces of data such as audio, video, images — just about anything you can cram into a computer file. That doesn't mean you should go crazy and store every file on your computer within the database.

For example, suppose an application must locate and play thousands of audio files. You could store all of them in the database or you could place the files in a directory tree somewhere and then store the locations of those files in the database. That makes the database simpler, smaller, and possibly more efficient because it doesn't need to store all of those files. It also makes it a lot easier to update the files. Instead of loading a new file into the database, you can simply replace the file on the disk.

This technique can also be useful for managing large amounts of shared data such as Web pages. You don't need to copy Wikipedia pages into your database (in fact, it would probably be a copyright violation). Instead you can store the URLs pointing to the pages you need. In this case you give up control of the data but you also don't have to store and maintain it yourself. If the data is updated, you'll see the new data the next time you visit a URL.

There are only a couple of drawbacks to this technique. First, you lose the ability to search inside any data that is not stored inside the database. I don't know of any databases that let you search inside video, audio, or jpeg data, however, so you probably shouldn't lose much sleep over giving up an ability that you don't have anyway. I wouldn't move textual data outside the database in this way, however, unless you're sure you'll never want to search inside it.

Second, you give up some of the security provided by the database. For example, you lose the database's record-locking features so you may have trouble allowing multiple concurrent users to update the data.

Try It Out First Normal Form

Suppose you're building a database of amusing commercials (see `www.veryfunnyads.com` and `giesbers.net/video` for some good ones). The Commercials table includes the fields Name, Product, Description, Length, Video (the commercial), and Still (a representative frame to remind you what the commercial is about).

Figure out which of these fields should remain in the database and which might be moved outside into the file system:

1. To figure out which fields should remain in the database, identify those that you might want to search. Include fields with simple values (such as numbers and short strings) that are easy to store in a database but that would not make a very big file on the disk.
2. To figure out which fields might be moved outside of the database into the file system, identify the fields that contain large chunks of non-searchable data.

How It Works

1. In this database, you might want to search the Name, Product, Description, and Length fields.
2. You cannot search the Video or Still fields whether you want to or not so you might as well move them into the file system. The Video field will contain particularly large amounts of data so moving it outside of the database might even make the database more efficient.

Summary

Though the focus of this book and your database design efforts is on databases, a database rarely lives in total isolation. Usually someone writes a program to interact with it. Often the database is just a backend for a complicated user interface.

To get the most out of your database, you need to consider it in its environment. In particular, you should think about the applications and programmers who will interact with it. Often a few relatively small changes to the design can make life easier for everyone who works with the database.

In this chapter you learned to:

- Plan ahead and document everything.
- Convert domains into tables to help user interface programmers and to make maintaining domain information easier.
- Keep tables well-focused and make each perform a single task.
- Use some redundancy and denormalized tables to improve performance.

The last several chapters dealt with database design techniques and considerations. Those chapters explained general techniques for building a data model and then modifying it to make it more efficient.

The next chapter switches from general discussion to more specific techniques. It summarizes some of the methods described in the previous chapters and explains some common database design patterns that you may find useful in providing specific data features.

Before you move on to Chapter 9, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

Exercises

1. Suppose you sell ocean cruises. Customers first decide what kind of Ship they want to travel on: Luxury Liner, Schooner, or Tuna Boat. Depending on that choice, they may select different classes of cabins. Luxury Liners provide 1st through 5th class. Schooners have 1st and 2nd class (basically a single or a double), and Tuna Boats have a single class (which they playfully call 1st Class) where you share a single large bunkroom with the rest of the crew.

   You could validate the Trips record's Ship and Cabin fields by using a table-level check constraint but, because you're a team player, you would rather build a foreign key constraint so the user interface can read the allowed values from the tables.

   Build such a table and display its data. Explain how this table will be used in the foreign key constraint.

2. Figure 8-4 shows a relational diagram showing the relationships between students, the classes they are taking, and the departments that hold the classes. For each table in this diagram, tell which of the three types of table it is: object, link, or lookup.

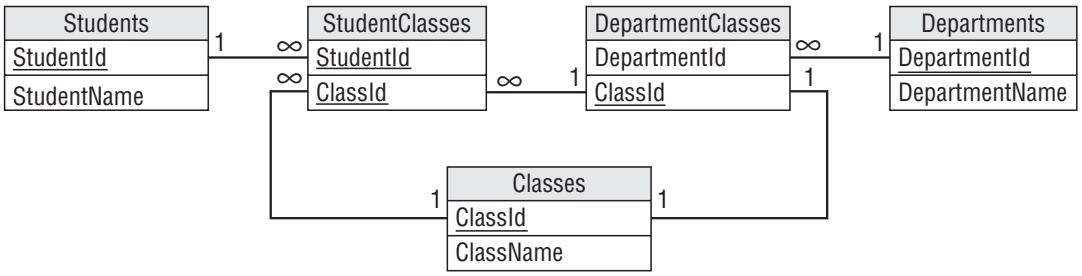


Figure 8-4

3. The following table stores information about checkers matches. Explain why it lacks focus and how you would fix it.

Player1	Player1Rank	Player2	Player2Rank	MatchTime
Smith	10	Jones	3	1:00
Marks	9	Lars	4	1:00
Aft	8	Cook	5	2:00
Mauren	7	Juno	6	2:00

4. Assume you have a large database that tracks how closely airplanes are to their scheduled departure and landing times. It tracks these values by plane (which is associated with a particular airline) and airport. It also records the weather at the starting and landing airports.

   Which of the following values should you store in a redundant variable and which should you calculate as needed?

   a. Average minutes late for an airline at a particular airport.
   b. Average minutes late for all airlines at a particular airport.
   c. Average minutes late for an airline across the country.
   d. Average minutes late for all airlines across the entire country.

   Assume that you need these numbers quickly several times per day.