Normalization Techniques

No matter what database system is being designed, normalization is a vital process. Normalization ensures that the database contained within the system is both accurate and doesn't contain duplicated information. Normalization can be summed up in a single sentence: "Every entity in a database needs to have a single theme" (see http://www.gslis.utexas.edu/~l384k11w/normstep.html). By theme, what is being said is that one entity models one "thing" and is used for just one purpose.

The term "normalized database" is used quite frequently, but it really is a misnomer, because there is no true or false criterion to determine if a database is normal, or otherwise. Normalization is a matter of levels, or steps. The different levels of normalization of a database indicate how well the structure adheres to the recognized standards of database design; although after the third level there is some question as to how useful the further levels are. You'll consider these arguments in the next chapter; this chapter will focus only on the first three levels of normalization, plus one additional level, which is a clearer restatement of numbers two and three.

As you might have guessed, database structure is a polarizing topic for the database architect. Disagreements often arise between database architects and client developers over how to store data. Why? Because no two database designs are the same and there are always several correct ways to structure a database, not to mention that it can take a good deal more time initially to build applications on top of properly structured tables. Logically, if you stored all of your data in one table, you would only need a single interface to modify the data. You can develop this much faster than if you had twenty tables, right? For the most part, there is no easy way to defend this statement. It's clearly true on one level, but completely wrong on many others. This chapter and the next will look at some of the problems with the statement, showing some of the programming anomalies that will be caused by not breaking your entities down into smaller "single-themed" packages.

An OLTP (online transaction processing) database that isn't extremely normalized is generally quicker to build for a client the first time because there are far fewer tables. This benefit soon disappears when minor user changes are required after the system is in production. Expert database designers realize that changes in data structure have large costs associated with them. When there are no rows in a table and no code to access the tables, structure changes may simply take minutes. When you've added a hundred rows to a table, with seven of its related tables having ten rows each, and programmers having written hundreds of lines of code, you can easily burn an hour of prime video game time changing the table structures, domains, and supporting code. Data conversion is also a problem, because if you have to change the table, then you have to fit existing data into the new structures. If there are a million rows in the table, forget about it. Once you've changed the table structure, you then have to change all of the code that accesses the table. Normalization also saves a great deal of storage space, because a major goal is to avoid repeating data. Note also that you do normalization as part of the logical modeling phase. This is because its implementation isn't specific. No matter what RDBMS (relational database management system) you use, you still have the same goals for the normalization process.

The process of normalization also ensures consistent entity structures, which will help you avoid modifying your existing entities in the future. If you've done a decent job on the requirements portion of the project, when you're finished, the normalization process should have weeded out most of the trouble areas in the database. Obviously I'm not saying that it can't be changed once the design has been implemented, because expansions and changes in business practice could create much needed amendments. However, it's easier to add new information and new entities to a fully normalized database.

This chapter and the next will go through the recognized techniques of normalization, and you'll see that the results of normalization minimize the inclusion of duplicated information, and make entities easier to manage and modify as well as more secure–although it will create a lot more entities.

Why Normalize?

There are many reasons to normalize data structures, as you'll see in the following sections.

Reducing NULLs

`NULL`s can be very detrimental to data integrity and performance, and can make querying the database very confusing. However, `NULL`s are a tricky subject in their own right; a lot of space can be devoted to how SQL Server 2000 handles them and how to avoid them or minimize their impact. So, rather than digressing here, the rules of `NULL` handling will be covered in Chapter 14 in more depth. It's enough here to note that the process of normalization can lead to there being fewer `NULL` values contained in the database, a definite good thing.

Eliminating Redundant Data

Any editable piece of data that isn't a primary key of a table (or part of one) but is a foreign key (or part of one) that occurs more than once in the database, is an error waiting to happen. No doubt you've all seen it before: a person's name stored in two places, then one version gets modified and the other doesn't–and suddenly you have two names where before there was just one.

The problem with storing redundant data will be very obvious to anyone who has moved to a new address. Every government authority requires its loyal citizens to individually change their address information on tax forms, driver's licenses, auto registrations, and so on, rather than one change being made centrally.

Avoiding Unnecessary Coding

Extra programming in triggers, stored procedures, or even in the business logic tier can be required to handle the poorly structured data and this in turn can impair performance significantly. This isn't to mention that extra coding increases the chance of introducing new bugs into a labyrinth of code required to maintain redundant data. Many database projects fail to meet their potential due to the enormous requirement of keeping redundant data in sync.

Maximizing Clustered Indexes

Clustered indexes are used to natively order a table in SQL Server. They are special indexes in which the physical storage of the data matches the order of the indexed columns, which allow for better performance of queries using that index. Typically, the indexes are used to order a table in a convenient manner to enhance performance. Each table may have only a single clustered index. The more clustered indexes in the database the less sorting that will need to be performed, and the more likely it is that queries will be able to use the `MERGE JOIN`–a special type of very fast join technique that requires sorted data. Sorting is a very costly operation that should be avoided if possible. Clustered indexes and indexes in general will be covered in great detail in Chapters 10 and 11.

Lowering the Number of Indexes per Table

The fewer indexes per table, the fewer the number of **pages** that might be moved around on a modification or insertion into the table. By pages, we're referring to pages of memory. In SQL Server, data and indexes are broken up and stored on 8KB pages. Of course SQL Server doesn't keep the whole database in memory at any one time. What it does do is keep a "snapshot" of what is currently being looked at. To keep the illusion of having the whole database in memory, SQL Server moves the pages in and out of a high-speed fast-access storage space when those pages are required, but this space can only contain a limited number of pages at any one time. The pages are therefore moved in and out of the space on the principle that the most frequently accessed remain in. The operation of moving pages in and out of memory is costly in terms of performance, and especially painful if one or more clients are twiddling their thumbs waiting for an operation to complete. Therefore, to keep performance as high as possible, you should minimize page transfers as much as possible.

When a table has many columns, it may be required to apply quite a few indexes on a table to avoid retrieval performance problems. While these indexes may give great retrieval gains, maintaining indexes can be very costly. Indexes are a very tricky topic because they have both positive and negative effects on performance and require a fine balance for optimal utilization.

Keeping Tables Thin

When we refer to a thinner table, we mean that the table has a relatively small number of columns. Thinner tables mean more data fits on a given page in the database, thus allowing the database server to retrieve more rows for a table in a single read than would be otherwise possible. This all means that there will be more tables in the system when you're finished normalizing. There is, however, a common sense cutoff point (no single column tables!). Bear in mind that in a typical OLTP system, very few columns of data are touched on every data modification, and frequently queries are used to gather the summary of a single value, like an account balance.

The Process of Normalization

If you recall your single sentence description of normalization from the first paragraph of this chapter–"Every table in the database needs to have a single theme"–you can take this to mean that each table should endeavor to represent a single entity. This concept will become more apparent over the next two chapters as you work through the process of normalization.

In 1972, E. F. Codd presented the world with the First Normal Form, based on the shape of the data, and the Second and Third Normal Forms, based on functional dependencies in the data. These were further refined by Codd and Boyce in the Boyce-Codd Normal Form. This chapter will cover these four in some detail.

During the discussion of normalization in this chapter and the next, you'll step through each of the different types, looking to eliminate all of the violations you find by following the rules specified in each type. You might decide to ignore some of the violations for expediency. It's also critical not only to read through each form in the book, but to consider each form while performing logical modeling.

Normalization is a process that can be broken down into three categories:

- Attribute shape
- Relationships between attributes
- Advanced relationship resolution

The discussion of advanced relationship topics will occur in the next chapter.

Attribute Shape

When you investigate the shape of attributes, you're looking at how the data is allowed to look in any single attribute. The **First Normal Form** deals with this topic, and is one of the most important normal forms.

This form is also used in the definition of the relational model of databases, and the definition of the First Normal Form is one of Codd's Twelve Rules, which aren't actually about normalization, but rather a set of rules that define a relational database. These rules are listed and discussed in Appendix A (see the Apress web site).

Entities in First Normal Form will have the following characteristics:

- All attributes must be atomic; that is, only one single value represented in a single attribute in a single instance of an entity
- All instances in a table must contain the same number of values
- All instances in a table must be different

First Normal Form violations manifest themselves in the physical model with messy data handling situations as you'll see shortly.

All Attributes Must Be Atomic

An attribute can only represent one single value; it may not be a group. This means there can be no arrays, no delimited data, and no multivalued attributes. To put it another way, the values stored in an attribute cannot be split into smaller parts. As examples, you'll explore some common violations of this rule of the First Normal Form.

E-mail Addresses

In an e-mail message, the e-mail address is typically stored in a format such as the following:

`name1@domain1.com;name2@domain2.com;name3@domain3.com`.

This is a clear violation of the First Normal Form because you're trying to store more than one e-mail address in a single `e-mail` attribute. Each e-mail address should form one separate attribute.

Names

Consider the name "John Q Public." This is less obviously a problem, but from what you understand about people's names (in English-based cultures at least) it is clear that you have the first name, middle initial, and last name. After breaking down the name into three parts, you get the attributes `first_name`, `middle_initial` (or `middle_name`, which I prefer), and `last_name`. This is usually fine, since a person's name in the US is generally considered to have three parts. In some situations, this may not be enough, and the number of parts may not be known until the user enters the data. Knowing the data requirements is very important in these kinds of situations.

Telephone Numbers

Consider the case of the telephone number. American telephone numbers take the form "1-423-555-1212" plus a possible extension number. From the previous examples, you can see that there are several values embedded in that telephone number, not to mention possible extensions. Additionally, there is frequently the need to store more than just American telephone numbers in a database. The decision on how to handle this situation may be totally based on how often the users store international phone numbers, because it would be a really hard task to build a table or set of entities to handle every situation.

So, for an American-style telephone number, you would need five attributes for each of the following parts in C-AAA-EEE-NNNN-XXXX:

- (C) Country code: This is the one that you dial for numbers that aren't within the area code, and signals to the phone that you're dialing a nonlocal number.
- (AAA) Area code: Indicates a calling area that is located within a state.
- (EEE) Exchange: Indicates a set of numbers within an area code.
- (NNNN) Number: Number used to make individual phone numbers unique.
- (XXXX) Extension: A number that must be dialed once after connecting using the previous numbers.

One of the coding examples later in this section will cover the sorts of issues that come up with storing phone numbers.

Addresses

It should be clear by now that all of an address should be broken up into attributes for street address, city, state, and postal code (from here on you'll ignore the internationalization factor, for brevity). However, street address can be broken down, in most cases, into number, street name, suite number, apartment number, and post office box. You'll look at street addresses again shortly.

IP Addresses

IP addresses are a very interesting case because they appear to be four pieces of data, formed like BY1.BY2.BY3.BY4 where BY is short for byte. This appears to be four different attributes, but is actually a representation of a single unsigned integer value based on the mathematical formula (BY1 * 256^3) + (BY2 * 256^2) + (BY3 * 256^1) + (BY4 * 256^0). Therefore, say you have an IP address 24.32.1.128 that could be stored as 404750720 = (24 * 256^3)+ (32 * 256^2) + (1 * 256^1) + (128). How the value is actually stored will depend in great part to what will be done with the data, for example:

- If the IP address is used as four distinct values, you might organize it in four different attributes, possibly named `ipAddressPart1`, `ipAddressPart2`, `ipAddressPart3`, and `ipAddressPart4`. Note that because there will always be exactly four parts, this type of storage doesn't violate the First Normal Form rules: All instances in an entity must contain the same number of values. This is covered in the next section.
- One valid reason to store the value as a single value is range checking. To determine when one IP address falls between the two other IP addresses, storing the addresses as integers allows you to simply search for data with a where clause, such as the following: "where `ipAddress` is between `ipAddressLow` and `ipAddressHigh`".

One thing should be clear however, in logical modeling; it's enough to understand that there is an IP address, and that it is in fact one value. While interesting to consider the implications of how you model and store the value, both storage possibilities are equivalent in that in the end you have some representation of the IP address. You may store it as either or both possible values (only one editable value, however) when you actually come to implement your system.

All Instances in an Entity Must Contain the Same Number of Values

This is best illustrated with a quick example. If you're building an entity that stores a person's name, then, if one row has one name, all rows must only have one name. If they might have two, all instances must be able to have two. If they may have a different number, you have to deal with this a different way.

An example of a violation of this rule of the First Normal Form can be found in entities that have several attributes with the same base name suffixed (or prefixed) with a number, such as `address_line_1`, `address_line_2`, and so on. Usually this is an attempt to allow multiple values for a single attribute in an entity. In the rare cases where there is always precisely the same number of values, then there is technically no violation of the First Normal Form. Even in such cases, it still isn't generally a good design decision, because users can change their minds frequently. To overcome all of this, you would create a child entity to hold the values in the malformed entity. This will also allow you to have a virtually unlimited number of values where the previous solution had a finite (and small) number of possible values. One of the issues to deal with is that the child rows created will require sequencing information to get the actual rows properly organized for usage. Note that the actual implementation of addresses will certainly be based on requirements for the system that is being created.

You can use cardinality rules as described in the previous chapter to constrain the number of possible values. If you need to choke things back because your model states that you only need a maximum of two children, and a minimum of one child, cardinality provides the mechanism for this.

All Occurrences of a Row Type in an Entity Must Be Different

This one seems obvious, but needs to be stated. Basically, this indicates that every First Normal Form entity must have a primary (or unique) key. Take care, however, because just adding an artificial key to an entity might technically make the entity comply with the letter of the rule, but certainly not the purpose. The purpose is that no two rows represent the same thing: Because an artificial key has no meaning by definition, it will not fix the problem.

Another issue is keys that use a date and time value to differentiate between rows. If the date and time value is part of the identification of the row, like a calendar entry or a log row, this is acceptable. If the value is simply used like the artificial key to force uniqueness (like indicating when the row was created) then it's a less than desirable solution.

Programming Anomalies Avoided by the First Normal Form

Violations of the First Normal Form are obvious and often very awkward if the columns affected are frequently accessed. The following examples will identify some of the situations that you can avoid by putting your entities in the First Normal Form.

Note that for these programming anomalies you'll switch over into using tables, rather than entities. This is because any issues you overlook now will eventually be manifested in your physical structures. It also gives some detail as to *why* this process is actually useful in your physical tables, and not just academic hoo-haa.

Modifying Lists in a Single Column

The big problem with First Normal Form violations is that relational theory and SQL aren't set up to handle nonatomic columns. Considering the previous example of the e-mail addresses attribute, suppose that you have a table named person with the following schema:

```
CREATE TABLE person
(
   personId int NOT NULL IDENTITY,
   name varchar(100) NOT NULL,
   emailAddress varchar(1000) NOT NULL
)
```

If you let users have more than one e-mail address and store it in a single e-mail address attribute, your emailAddress column might look like the following: "Davidsons@d.com;ldavidson@email.com".

Also consider that many different users in the database might use the Davidsons@d.com e-mail address. If you need to change the e-mail address from Davidsons@d.com to Davidsons@domain.com, you would need to execute code like the following for every person that uses this e-mail address:

```
UPDATE person
SET emailAddress = replace(emailAddress,'Davidsons@d.com',
        'Davidsons@domain.com')
WHERE emailAddress like '%Davidsons@d.com%'
```

This code doesn't seem like trouble, but what about the case where there is also the e-mail address theDavidsons@d.com? You would have to take that into account, and it would at the very least be messy. If the e-mail address were split off as its own table apart from the person entity, it would be easier and cleaner to deal with.

Modifying Multipart Values

The programming logic required to change part of the multipart values can be very confusing. Take, for example, the case of telephone area codes. In the United States, you have more phones, pagers, cell phones, and so on than the creators of the area code system ever thought of, and so the communications companies frequently change or introduce new area codes.

If all values are stored in atomic containers, updating the area code would take a single, easy-to-follow, one-line SQL statement like this:

```
UPDATE phoneNumber
SET areaCode = '423'
WHERE areaCode = '615' AND exchange IN ('232','323',…,'989')
```

Instead, if the phone number is stored in unformatted text, you get the following:

```
UPDATE phoneNumber
SET phoneNumber = REPLACE(phoneNumber,'-615-','-423-')
WHERE phoneNumber LIKE '_-615-___-____'
   AND substring(phoneNumber,7,3) IN ('232','323',…,'989') --area codes generally
                                                           --change for certain
                                                           --exchanges
```

This example requires perfect formatting of the phone number data to work, and unless that is forced upon the users, this is unlikely to be the case.

Modifying Rows with Variable Numbers of Facts

One of the main problems with allowing variable numbers of facts in a given row, is dealing with the different situations that occur when one needs to deal with just one of the columns instead of the other. Say you have a very basic structured table such as the following:

```
CREATE TABLE payments
(
   paymentsId int NOT NULL IDENTITY,   accountId int NOT NULL,
   payment1 money NOT NULL,
   payment2 money NULL
)
```

Wherever the user has to make two payments (for some reason), he would be required to enter a payment as the following code snippet shows:

```
UPDATE payments
SET payment1 = case WHEN payment1 IS NULL THEN 1000.00 ELSE payment1 END,
    payment2 = case WHEN payment1 IS NOT NULL AND payment2 IS NULL THEN
    1000.00 ELSE payment2 END
WHERE accountId = 1
```

Of course, you probably will not be using SQL to deal with columns such as this, but even if this logic is done on the client side by giving multiple blocks for payments, or you predetermine which of the payments needs to be made, it should still be clear that it's going to be problematic to deal with.

The alternative would be to have a table that's built like this:

```
CREATE TABLE payment
(
   paymentId int NOT NULL IDENTITY,
   accountId int NOT NULL, --foreign key to account table
   date datetime NOT NULL,
   amount money
)
```

Then, adding a payment would be as simple as adding a new row to the payment table as follows:

```
INSERT payment (accountId, date, amount)
VALUES (1, 'June 12 2000', $300.00)
```

This is certainly a better solution. One thing to note here is that `date` is seemingly not an atomic datatype (in other words, it is a multivalued column). In fact, SQL Server simply has a `datetime` datatype that can be thought of as several columns all rolled up into one: day, month, year, hour, minute, seconds, milliseconds, and so on. In all actuality, it technically isn't a problem, because what the column stores is the number of ticks (.003 second per) since a date in the past. The only problem is you quite often deal with the date value and the time value separately.

The `datetime` datatype is used often for convenience reasons, but it gives you quite the same issues as if it were not in the First Normal Form. Because of this, there are a host of functions (`DATEPART`, `DATEDIFF`) in SQL Server designed to make up for the fact that `datetime` and `smalldatetime` aren't proper relational datatypes. There are situations in which it's prudent to implement your own date or time user-defined datatypes and deal with time variables in a reasonable manner.

Clues That Existing Data Isn't in First Normal Form

Next you're going to take a look at how you might go about recognizing whether the data in a given database is already in First Normal Form or not.

Data That Contains Nonalphanumeric Characters

Nonalphabetic characters include commas, brackets, parentheses, pipe characters, and so on. These act as warning signs that you're dealing with a multivalued column. However, be careful not to go too far. For instance, if you were designing a solution to hold a block of text, you have probably normalized too much if you have a word entity, a sentence entity, and a paragraph entity (if you had already been considering it, give yourself three points for thinking ahead). This clue is more applicable to entities that have delimited lists.

Column Names with Numbers at the End

As noted previously, an obvious example of column names with numbers at the end would be finding entities with *child1*, *child2*, and so on, attributes or, my favorite, *UserDefined1*, *UserDefined2*, and others. These are usually pretty messy to deal with and should be candidates for their own table. They don't have to be wrong; for example there always exist two values–this is perfectly proper–but be careful that what is thought of as always, is actually always. Often there will arise "exceptions" that will cause the solution to fail. "A person always has two forms of identification noted in columns ID1 and ID2, except when the manager says they don't have to" In this case always doesn't mean always. `Coordinate_1`, `Coordinate_2` might be acceptable, because it will always require two coordinates to find a point in a two-dimensional space, never any more, or never any less.

These kinds of columns are a very common holdover from the days of flat file databases. Multitable data access was costly, so they put many columns in a single table. This is very detrimental for relational database systems.

Relationships Between Attributes

The next normal forms you'll look at are concerned with the relationships between attributes in an entity and most importantly, the key in that entity. These normal forms deal with minimizing improper functional dependencies. As discussed in Chapter 3, being functionally dependent implies that the output of a function on one value (call it Value1) is *always* the exact same value (call it Value2): Value2 is functionally dependent on Value1.

For example, consider the following situation. You have three values: Owner Name, Product Type, and Serial Number. Serial Numbers imply a particular Product Type, so they are functionally dependent. If you change the Product Type but fail to change the Serial Number, then your Serial Number and Product Type will no longer match and the three values will no longer be of any value. Because the two values are functionally dependent on one another, then they both must be modified.

On this note, let's say that you have two sets of the same values, each alike. If you change the non-key values on one of the sets, and not the others, then there are two sets with the same key, with different values. Now the unique identifier refers to two different values, usually not a desirable condition, and certainly not within a single table. The normal forms you'll look at for the rest of this chapter deal with making sure there are none of these relationships "hiding" in entities.

The following quote will help you understand what the Second Normal Form, Third Normal Form, and Boyce-Codd Normal Form are concerned with:

> *Non-key attributes must provide a detail about the key, the whole key, and nothing but the key.*

This means that non-key attribute have to further describe the key of the entity, and not describe any other attributes. The forms that deal with relationships between attributes are as follows:

- Second Normal Form
- Third Normal Form
- Boyce-Codd Normal Form

Second Normal Form

The first attribute relationship normal form is the **Second Normal Form**. An entity complying with Second Normal Form has to have the following characteristics:

- The entity must be in the First Normal Form
- Each attribute must be a fact describing the entire key

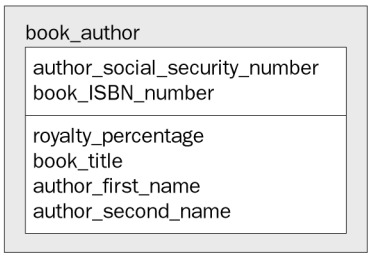
> **The Second Normal Form is only relevant when a composite key (a key composed of two or more columns) exists in the entity.**

The Entity Must Be in the First Normal Form

This is very important; it's essential to go through each step of the normalization process to eliminate problems in data. It may be hard to locate the Second Normal Form problems if you still have First Normal Form problems. Otherwise, some of the problems you're trying to fix in this rule may show up in any misshapen attributes not dealt with in the previous rule.

Each Non-key Attribute Must Describe the Entire Key

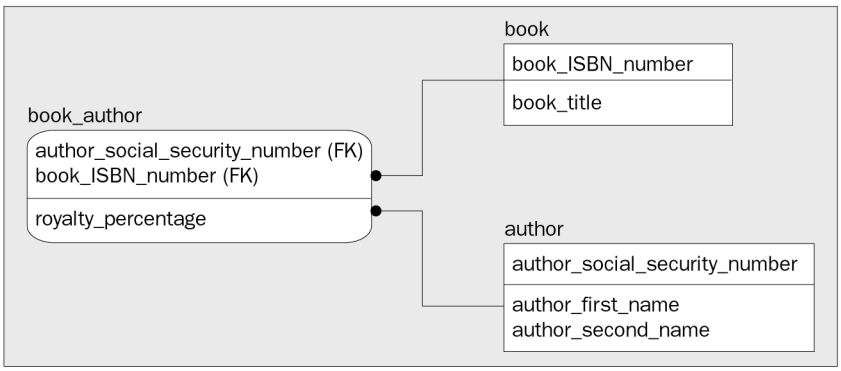
What is being described here is that each non-key attribute must depict the entity described by **all** attributes in the key, and not simply parts. If this isn't true and any of the non-key attributes are functionally dependent on a subset of the attributes in the key, then the attributes will be data modification anomalies. For example, consider the following structure:



The `book_ISBN_number` attribute uniquely identifies the book, and `author_social_security_number` uniquely identifies the author. Hence, these two columns create one key that uniquely identifies an author for a book. The problem is with the other attributes. The `royalty_percentage` attribute defines the royalty that the author is receiving for the book, so this refers to the entire key. The `book_title` describes the book, but doesn't describe the author at all. The same goes for the `author_first_name` and `author_last_name` attributes. They describe the author, but not the book at all.

This is a prime example of a functional dependency. For every value in the `book_ISBN_number` column, there must exist the same book title and author (since it's the same book). But for every `book_ISBN_number`, it isn't true that you'll definitely have the same `royalty_percentage`–this is actually dependent on *both* the author and the book, and not one or the other, because when books are cowritten, the split might be based on many factors, such as celebrity, how many copies of the book are produced, and so on.

Hence you have problems, so you need to create three separate entities to store this data.



After you split the entities, you'll see that the `royalty_percentage` attribute is still a fact that describes the author writing the book, the `book_title` is now an attribute describing the entity defined by the `book_ISBN_number`, and the author's name attributes are attributes of the `author` entity, identified by the `author_social_security_number`.

Note that the `book` to `book_author` relationship is an identifying type relationship. Second Normal Form violations are usually logically modeled as identifying relationships where the primary key of the new entity is migrated to the entity where the original problem occurred.

The previous illustration demonstrates this concept quite well. In the corrected example you have isolated the functional dependencies so that attributes that are functionally dependent on another attribute are functionally dependent on the key.

Because there are no columns that aren't functionally dependent on a part of the key, these entities are in the Second Normal Form.

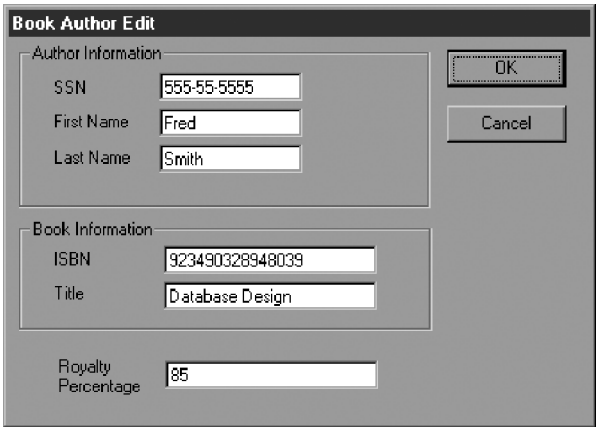
Programming Problems Avoided by the Second Normal Form

All of the programming issues that arise with the Second Normal Form, as well as the Third and Boyce-Codd Normal Forms, deal with functional dependencies. The programming issue is quite simple. Consider the data you're modeling, and how it would be affected if it were physicalized, and if you executed a statement like this, where you're updating an attribute that clearly represents a singular thing (in this case a book's title):

```
UPDATE book_author
SET book_title = 'Database Design'
WHERE book_ISBN_number = '923490328948039'
```

If it were possible to touch more than one row, there are problems with your structures.

The crux of the problems are that many programmers don't have their database design thinking caps on when they are churning out applications, so you get tables created with client screens like this one.



Consider what happens if you use this screen to change the title of a multiauthor book in a database that has `book_author` tables like the one shown in the first diagram in the previous section, "Each Non-Key Attribute Must Describe the Entire Key." If the book has two authors, there will be two `book_author` tables for this book. Now a user opens the editing screen and changes the title, as shown here. When he saves the change, it will only alter the `book_author` table for Fred Smith, not the `book_author` table for his coauthor. The two `book_author` tables, originally for the same book, now show different titles.

This problem is rectified by using the Second Normal Form, as shown in the second diagram in that section. In this form, the `book` table will connect to two `book_author` tables. Changing the title in this editor screen will change the `book_title` column in this single `book` table; the two `book_author` tables are only linked to the `book` table by the `book_ISBN_number` column, so the database will still show both authors as having coauthored the same book. Everything remains in sync.

> **Reality Check: There is nothing sinfully wrong with this screen. If the design calls for this sort of thing to be entered, and this is what the client wants, fine. However, it's clearly best if the underlying structures don't look like this, for the reasons stated. It would also probably be better to give rich functionality to look up the book from the ISBN number and populate the rest of the columns for the book (obviously a table for a book will include more information than its ISBN number and title) but clearly this is a UI design issue, and not a question of database structure.**

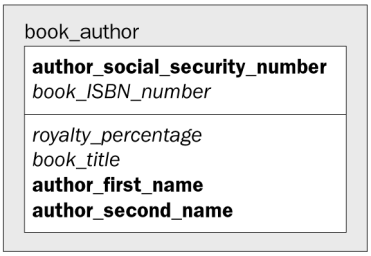
Clues That a Table Isn't in the Second Normal Form

The clues for detecting whether entities are in the Second Normal Form aren't quite as straightforward as the clues for the First Normal Form. They take some careful thought, however, and some thorough examination of your structures such as the following:

- Repeating Key Column Name Prefixes
- Repeating Groups of Data
- Composite Keys Without a Foreign Key

Repeating Key Column Name Prefixes

This situation is one of the dead giveaways. Let's revisit the previous example.



You have `author_first_name` and `author_second_name`, which are functionally dependent on `author_social_security_number`. You also have `book_title` and `book_ISBN_number` with the same situation.

Having such obvious prefixes isn't always the case, but it's a good thing to look for because this is a rather common mistake made by novice designers.

Repeating Groups of Data

More difficult to recognize are the repeating groups of data. Imagine executing multiple `SELECT` statements on a table, each time retrieving all rows (if possible), ordered by each of the important columns. If there is a functionally dependent attribute on one of the attributes, anywhere one of the columns is equal to X, you'll see the dependent column, Y.

Take a look at some example entries from the following table.

author_social_ security_number	**book**_ISBN_number	**royalty_ percentage**
DELA-777-888	1-861-000-156-7	2
DELA-777-888	1-861-000-338-1	3
GIBB-423-4421	1-861-000-156-7	3
Book_title	**author**_first_name	**author**_second_name
Instant Tiddlywinks	Vervain	Delaware
Beginning Ludo	Vervain	Delaware
Instant Tiddlywinks	Gordon	Gibbon

> `Book_title` *is, of course, dependent on* `book_ISBN_Number`*, so any time you see an ISBN number = 1-861-000-156-7, you can be sure that the book title is "Instant Tiddlywinks." If it isn't, then there's something wrong in the database.*

Composite Keys Without a Foreign Key

If there is more than one attribute in the key that isn't a foreign key, any attributes that describe those attributes are likely to be violating the Second Normal Form. This isn't always true of course, considering the previous example of a phone number in which each of the pieces made up the key. However, consider what the composite key values are made up of. In the case of the phone number, the different parts of the phone number come together to form a singular thing, a phone number.

However, the previous example of a key that was made up of the Book ISBN number and Author Identification clearly give you two keys that don't represent the same thing, in this case a Book and an Author.

Coding Around the Problem

Scouring any existing database code is one good way of discovering problems, based on the lifespan of the system you're analyzing. Many times, a programmer will simply write code to make sure the Second Normal Form violation isn't harmful to the data, rather than remodeling it into a proper structure. At one time in the history of the RDBMS (relational database management system), this may have been the only way to handle this situation; however, now that technology has caught up with the relational theory, this isn't the case.

It's important to understand that the case isn't trying to be made that theory has changed a bit due to technology. Actually, relational theory has been very stable throughout the years with few of the basic concepts changing in the past decade. Ten years ago, I had quite a few problems making a normalized system operational in the hardware and operating system constraints I had to deal with, so I cut corners in my models for "performance" reasons.

Using current hardware, there is no need to even begin to cut normalization corners for performance. It's best to resolve these issues with SQL joins instead of spaghetti code to maintain denormalized data. Of course, at this point in the design process, it's best to not even consider the topic of implementation, performance, or any subject in which you aren't simply working towards proper logical storage of data.

Third Normal Form

An entity that's in the **Third Normal Form** will have the following characteristics:

- The entity must be in the Second Normal Form
- An entity is in violation of the Third Normal Form if a non-key attribute is a fact about another non-key attribute

You can rephrase the second bullet like this:

> *All attributes must be a fact about the key, and nothing but the key.*

The Third Normal Form differs from the Second Normal Form in that it deals with the relationship of non-key data to non-key data. The problems are the same and many of the symptoms are the same, but it can be harder to locate the general kind of violations that this form tends to deal with. Basically, the main difference is that data in one attribute, instead of being dependent on the key, is actually dependent on data in another non-key attribute. The requirements for the Third Normal Form are as follows.

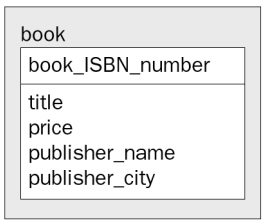
The Entity Must Be in the Second Normal Form

Once again, this is very important. It may be hard to locate the Third Normal Form problems if the Second Normal Form problems still remain.

Non-Key Attributes Cannot Describe Other Non-Key Attributes

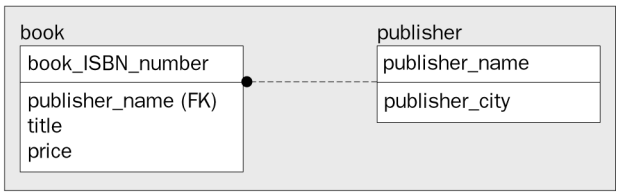
If any of the attributes are functionally dependent on an attribute other than the key, then you're again going to have data modification anomalies. Because you're in the Second Normal Form already, you've proven that all of your attributes are reliant on the whole key, but you haven't looked at the relationship of the attributes to one another.

In the following diagram, you take your `book` entity and extend it to include the publisher and the city where the publisher is located.



`Title` defines the title for the book defined by the `book_ISBN_number`, `price` indicates the price of the book. The case can clearly be made `publisher_name` describes the book's publisher, but it's clear that the `publisher_city` doesn't make sense in this context, because it doesn't directly describe the book.

To correct this situation, you need to create a different entity to identify the publisher information.



Now the `publisher` entity has only data concerning the publisher, and the `book` entity has book information. What makes this so valuable is that now, if you want to add information to your schema concerning the publisher, for instance contact information or an address, it's very obvious where you add that information. Now you have your `publisher_city` attribute identifying the publisher, not the book. Once we get into physical modeling, we'll discuss the merits of having the `publisher_name` attribute as the primary key, but for now this is a reasonable primary key, and a reasonable set of attributes.

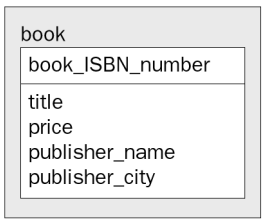
Note that the resolution of this problem was to create a **nonidentifying** relationship: publisher is related to book. Because the malevolent attributes weren't in the key to begin with, they don't go there now.

All Attributes Must Be a Fact Describing the Key, the Whole Key, and Nothing but the Key

If it sounds familiar, it should. This little saying is the backbone for the whole group of normal forms concerned with the relationship between the key and non-key attributes.

Programming Problems Avoided by the Third Normal Form

While the methods of violating the Third Normal Form are very close to the violations of the Second Normal Form, there are a few important differences. Because you aren't dealing with key values, every attribute's relationship to every non-key attribute needs to be considered, and so does every combination of attributes. In the book example, you had the following entity structure:

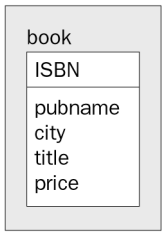


Every attribute should be considered against every other attribute. If entities are of reasonable size (10 to 20 attributes in an entity is probably as many as seems reasonable without violating some normalization rule. Of course, this doesn't always hold, but it's a good rule of thumb.), then the process of weeding out Third Normal Form problems won't be too lengthy a process. In this example, in order to do a "perfect" job of comparing each attribute against each of the other attributes, you need to check each attribute against the other three attributes. Because there are four attributes, you need to consider the N * (N – 1) or (4 * 3) = 12 different permutations of attribute relations to be safe (ignoring the fact that you'll be checking some values more than once). In this example entity you must check the following:

- `title` against: `price`, `publisher_name`, and `publisher_city`
- `price` against: `title`, `publisher_name`, and `publisher_city`
- `publisher_name` against: `price`, `title`, and `publisher_city`
- `publisher_city` against: `price`, `title`, and `publisher_name`

From this you'll notice that, when you check `publisher_name` against the other three attributes, it becomes clear that `publisher_city` is functionally dependent on it, hence there's a Third Normal Form violation.

After designing a few thousand entities, some common attributes will jump out as problems, and only a few attributes will have to be considered in routine normalization checks. Note that this example has tailored names to make it seem simple, but in reality, names are often far more cryptic. Consider the following entity.



These names are probably less cryptic than those that might actually exist in some legacy database entities; however, they're already ambiguous enough to cause problems. `City` seems almost fine here, unless you carefully consider that most books don't have a `city`, but publishers probably will. The following example code shows what happens if you want to change the `city` attribute and keep it in sync.

Take, for example, the situation where you have the table as it was built previously:

```
CREATE TABLE book
(
   ISBN varchar(20) NOT NULL,
   pubname varchar(60) NOT NULL,
   city varchar(60) NOT NULL,
   title varchar(60) NOT NULL,
   price money NOT NULL
)
```

This has the Third Normal Form violations that were identified. Consider the situation in which you want to update the `city` column for `ISBN 23232380237` from a value of `Virginia Beach` to a value of `Nashville`. You first would update the single row as follows:

```
UPDATE book
SET city = 'Nashville'
WHERE ISBN = '23232380237'
```

But because you had the functional dependency of the `publisher` to `city` relationship, you now have to update all of the books that have the same publisher name to the new value as well:

```
UPDATE book
SET city = 'Nashville'
WHERE city = 'Virginia Beach'
   AND pubname = 'Phantom Publishing' --publisher name
```

Although this is the proper way to ensure that the batch code updates the `city` properly as well as the book, in most cases this code will be buried in the application, not tied together with a transaction, much less one in a batch. It's also easy to forget these types of relationships within the row when writing new processes that are required as the system changes over time.

Any errors in one `UPDATE` statement, and data can be compromised (clearly something you're trying to avoid by spending all of this time working on your structures). For existing SQL Server applications that are being redesigned, employ the SQL Server Profiler to check what SQL is actually being sent to SQL Server from the application.

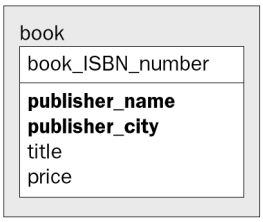
Clues That a Table Isn't in the Third Normal Form

The clues for the Third Normal Form are quite similar to those for the Second Normal Form, because they're trying to solve the same sort of problem–making sure that all non-key attributes refer to the key of the entity. These clues are as follows:

- Multiple Columns with the Same Prefix
- Repeating Groups of Data
- Summary Data

Multiple Columns with the Same Prefix

If you revisit the previous example, you'll see that it's obvious that `publisher_name` and `publisher_city` are multiple columns with the same prefix.



In some cases the prefix used may not be so obvious, such as `pub_name`, `pblish_city`, or even `location_pub`; all good reasons to establish a decent naming standard.

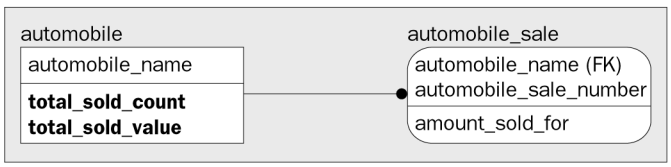
Repeating Groups of Data

You do this the same way that you did for the Second Normal Form, but you'll need to consider more permutations, as discussed.

Summary Data

One of the common violations of the Third Normal Form that may not seem blindingly obvious is **summary data**. This is where columns are added to the parent entity, which refers to the child rows and summarizes them. Summary data has been one of the most frequently necessary evils that we've had to deal with throughout the history of the relational database. There are several new features in SQL Server 2000 that you'll employ to help avoid summary data in your implementation, but in logical modeling there is *absolutely* no place for it. Not only is summary data not functionally dependent on non-key columns, but it's dependent on nonentity columns. This causes all sorts of confusion as I shall demonstrate. Summary data should be reserved either for physical design or the data warehousing systems we introduced in Chapter 1.

Take the following example of an auto dealer. The dealer has an entity listing all of the automobiles it sells, and it has an entity recording of each automobile sale.



This kind of thing probably has no part in logical modeling, because the sales data is available. Instead of accepting that the total number of vehicles sold and their value is available, the designer has decided to add columns in the parent entity that refer to the child rows and summarizes them.

Is this required during the physical implementation? Possibly, depending on performance, though it's true that the complexity of the implemented system has most likely increased by an order of magnitude, because you'll have to have triggers on the `automobile_sale` entity that calculate these values for any change in the `automobile_sale` entity. If this is a highly active database with frequent rows added to the `automobile_sale` entity, this will tend to slow the database down considerably. On the other hand, if it's an often inactive database, then there will be very few rows in the child entity, and so the performance gains made by being able to quickly find the numbers of vehicles sold and their value will be very small anyway.

The key is that in logical modeling it isn't required, because the data modeled in the `total_ columns` is actually in existence in the sales table. The information you are identifying in the logical model should be modeled in one and only one spot.

Boyce-Codd Normal Form

During the discussions of the Second and Third Normal Forms, I was purposefully vague with the word **key**. As mentioned before, a key can be *any* candidate key, whether the primary key or an alternate key.

The **Boyce-Codd Normal Form** is actually a better constructed replacement for both the Second and Third Normal Forms; it takes the meaning of the Second and Third Normal Forms and restates it in a stricter way. Note that, to be in Boyce-Codd Normal Form, there is no mention of Second Normal Form or Third Normal Form. The Boyce-Codd Normal Form actually encompasses them both, and is defined as follows:

- The entity is in First Normal Form
- All attributes are fully dependent on a key
- An entity is in Boyce-Codd Normal Form if every determinant is a key

So let's look at each of these rules individually, except for the First Normal Form, which was covered in the Second Normal Form discussion.

All Attributes Are Fully Dependent on a Key

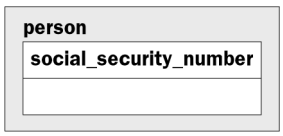
You can rephrase this like so:

> *All attributes must be a fact about **a** key, and nothing but **a** key.*

This is a slight but important deviation from the previous rules for Second Normal Form and Third Normal Form. In this case, you don't specify *the entire* key or just *the* key–now it's *a* key. How does this differ? Well, it does and it doesn't. It basically expands the meaning of Second Normal Form and Third Normal Form to deal with the very typical situation where you have more than one key.

It's noted that the attribute must be fully dependent on a key, and this key is defined as the **unique identifier**. The unique identifier should be thought of as the address or pointer for the entity, regardless of whether you use a natural key or otherwise. The **entity**, as defined, is the logical representation of a single object, either real or imaginary. Think of *every* key as the entity's ID badge, or SSN (social security number), or whole name, and it's easy to begin to understand what each of the attributes should be like.

For example, let's take a person who works for a company and model them. First you choose your key: Let's use the SSN.



Then you start adding the other attributes you know about your employee–her name, hair color, eye color, the badge number she was given, her driver's license number, and so on. Now you have the following entity:

Careful study of the entity shows that it's in the Third Normal Form, because each of the attributes further describes the entity. The `first_name`, `height`, `badge_number`, and others all refer to the entity. The same may be said for the `social_security_number`. It has been chosen as the key, primarily because it was the first thing you saw. In logical modeling, the choice of which attribute(s) are the primary key isn't all that meaningful, and you can change them at any time. (A large discussion of proper primary keys will be dealt with in Chapter 10.) In most cases, even in the logical modeling phase, you'll simply use a pointer as discussed in Chapter 3.

The following sentence basically explains the Second and Third Normal Forms:

> **All attributes must further describe the entity, the whole entity, and nothing but the entity.**

As long this concept is understood, data modeling and normalization become much easier.

An Entity Is in Boyce-Codd Normal Form if Every Determinant Is a Key

The second part of the quest for the Boyce-Codd Normal Form is to make sure that every determinant is a key, or a unique identifier for the entity. The definition of a determinant in Chapter 3 was as follows:

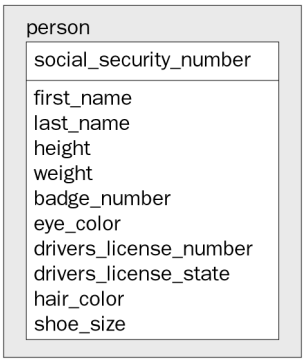
> **Any attribute or combination of attributes on which any other attribute or combination of attributes is functionally dependent.**

Based on your study of the Second and Third Normal Forms, you can see that this is basically the definition of a key. Because all attributes that aren't keys must be functionally dependent on a key, the definition of a determinant is the same as the definition of a key.

The Boyce-Codd Normal Form simply extends the previous normal forms by saying that an entity may have many keys, and all attributes must be dependent on one of these keys. I've simplified this a bit by noting that every key must uniquely identify the entity, and every non-key attribute must describe the entity.

One interesting thing that should be noted is that each key is a determinant for all other keys. This is because, in every place where one key value is seen, you can replace it with the other key value without losing the meaning of the entity. This isn't to say that an alternate key cannot change values–not at all. The driver's license number is a good key, but if the Department of Motor Vehicles issues all new numbers, it's still a key, and it will still identify and describe the entity. If the value of any candidate key changes, this is perfectly acceptable.

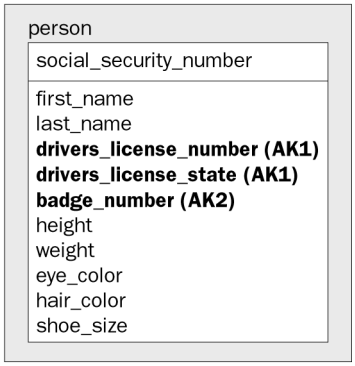
With this definition in mind, let's take the example entity you're modeling for this section and look at it again.



What you'll be looking for now are attributes or groups of attributes that are dependent on the key. The attributes or groups will also be unique to each instance of this entity.

`First_name` will not be by itself, and it would not be a very good assumption to assume that `first_name` and `last_name` are. (It all depends on the size of the target set as to whether or not the user would be willing to accept this. At the very least you would likely need to include the middle initial and title, but this still isn't a very good key.) `Height` describes the person, but isn't unique. The same is true for `weight`. `Badge_number` certainly should be unique, so you'll make it a key. (Note that you don't have `badge_issued_date`, because that would refer to the badge and doesn't help the Boyce-Codd Normal Form example.) `drivers_license_number` is probably unique, but consider variations across localities, because two governments may have similar numbering schemes that may cause duplication. `Hair_color` and `shoe_size` describe the person, but neither could be considered unique. Even taking the person's `height`, `weight`, `eye_color`, `hair_color`, and `shoe_size` together, uniqueness could not be guaranteed between two random people.

So now you model the entity as follows:



You now have three keys for this object. When you do the physical modeling, you'll choose the proper key from the keys you have defined, or use an artificial key. As discussed in Chapter 3, an artificial key is simply a value that is used as a pointer to an object, much like `badge_number` is a pointer that a company uses to identify an employee, or the government using SSNs to identify individuals in the US (of course the social security number is actually a smart key, with an artificial key embedded).

It's also worth considering that an SSN isn't always a very good key either. Even if you limit it to only US citizens, you'll find that there are plenty of people who don't have an SSN. And certainly if you want to accommodate people from outside the US, then the SSN will never work. It will certainly depend on the situation, because there are many situations in which the user is required to have an SSN, and some where an SSN or green card is required to identify the user as a valid resident of the US. The situation will always dictate the eventual solution, and it's simply up to the data architect to choose the appropriate path for the situation being modeled.

When choosing a key, you always try to make sure that keys don't completely encompass other keys. For instance, `height` isn't a unique key, but `height` and `badge_number` is! It's important to make certain that individual parts of unique keys cannot be guaranteed unique by themselves; otherwise mistakes can be made by accidentally putting non-unique data in the columns that need to be unique.

Clues and Programming Anomalies

The clues for determining that an entity is in Boyce-Codd Normal Form are the same as for the Second and Third Normal Forms. The programming anomalies solved by the Boyce-Codd Normal Form are the same too.

The main point is that once all of the determinants are modeled during this phase of the design, implementing the determinants as unique keys will be far more likely to occur. This will prevent users from entering non-unique values in the columns that need to have unique values in them.

This completes the overview of the first three normal forms. You'll look at the Fourth and Fifth Normal Forms in the next chapter, but now you'll implement your newly learned concepts into the case study.

Case Study

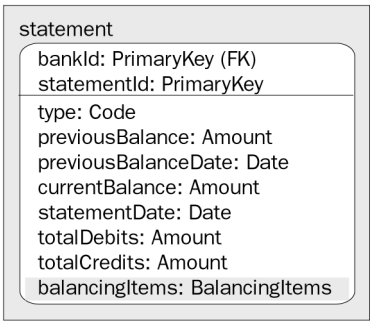
For this chapter, you'll take your model that you put together in the previous chapter and walk through each of the normal forms, correcting the model according to the rules, and ending with a completed diagram.

First Normal Form

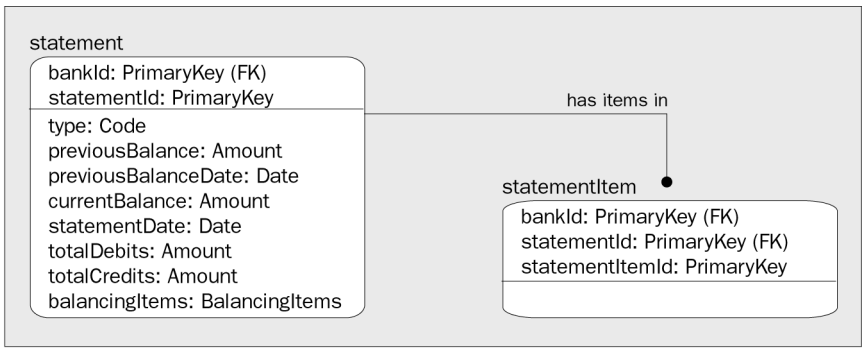
Your model has a few violations of the First Normal Form. You'll break down each of the different violations that may occur.

All Attributes Must Be Atomic

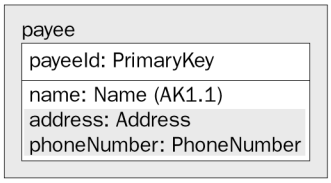
You have one example of this sort of violation. It occurs in the `statement` entity. You should recall that this entity represents the statement that the bank sends each month so that the client can reconcile all of the checks the client has hand-entered as they actually occurred. The `balancingItems` attribute contains all of the transactions that the bank has recorded and needs to be matched up to items in the register. However, this attribute will contain many rows and many columns, hardly a single atomic value, which is required by the First Normal Form.



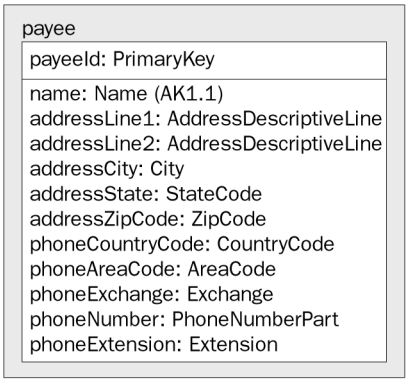
So, you need to add a new entity to contain these items. Because you don't know exactly what will go into this entity, you'll let the attributes migrate from the `statement` entity, then add another pointer to the primary key for uniqueness.



A different type of violation of this same kind must also be looked at. Consider the `payee` entity and the two attributes `address` and `phoneNumber`.



For your example, yoy'll just look at the `address` attribute. The address is made up of several parts, the street address lines (usually we're happy with two lines to store street information), city, state, and zip code. You'll also expand the `phoneNumber` attribute as we discussed earlier. Once you have finished this process, you're left with the following result:

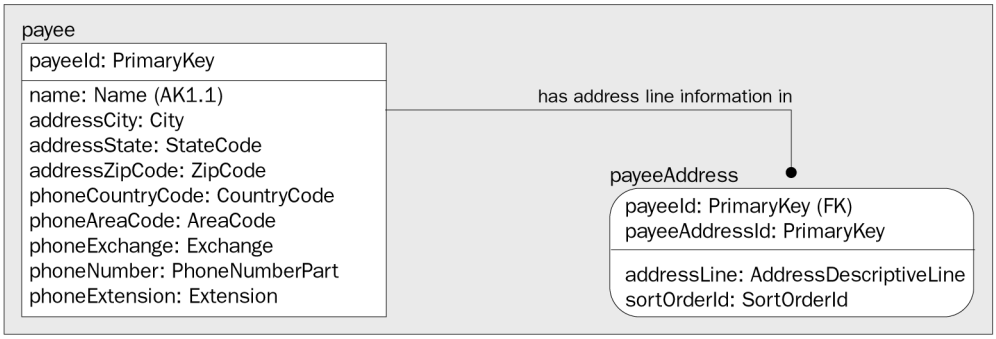


There are no attributes now in this example that overtly violate the First Normal Form and therefore aren't atomic. Actually, you could make the case that the `addressLine1` attribute isn't atomic, since it will contain street number, street name, and other pieces of information. This kind of discussion can be a fun diversion from an afternoon of actual work, but in the final analysis you generally will only model the address down to a level that meets the requirements for your purposes. There are clearly reasons to model the `addressLine1` attribute as `streetNumber`, `streetName`, etc, especially when standardizing addresses in order to save a penny or two per letter on postage fees. Not exciting or valuable unless posting thousands or millions of pieces of correspondence every year. Hence, you stick with the reasonably standard implementation of having unstructured lines for street names and number, apartment numbers, etc.

Notice that you created domains for each new attribute. `addressLine1` and `addressLine2` are the same sort of item. Also of note is the `phoneNumber` attribute. It has the same name as before, but it has a different meaning, because a phone number is made up of country code, area code, exchange, and number. As it has a different meaning, you created a new domain with a new name, since in practice you may still have entities that use the domain.

All Occurrences of a Row Type Must Contain the Same Number of Values

In your new `payee` entity, you have put together a common set of attributes that violate this part of the rule, the `addressLine1` and `addressLine2` attributes. While this is a common solution to the address problem, it's a violation nonetheless. Having a fixed number of address line attributes has bitten me several times when addresses needed more, sometimes even four or five of them. Since not every address will have the same number of items, this is a problem. You solve this by adding another child entity for the address line information:



This may seem a bit odd, and it is, considering the way databases have been developed for long periods of time. However, this address design gives you the flexibility to store as many pieces of information as you may need, instead of having to add columns if the situation requires it. This is also a good example of normalization creating more entities.

> **Reality Check: For the most part, it may not be the most user friendly way to store the address, and so the address might be stored using `addressLine1` and `addressLine2`, or considering that what separates `addressLine1` and `addressLine2` is a carriage return and a linefeed, storing it in one large column might also suffice. You'll store it in a different column for discussion purposes.**

All Occurrences of a Row Type in an Entity Must be Different

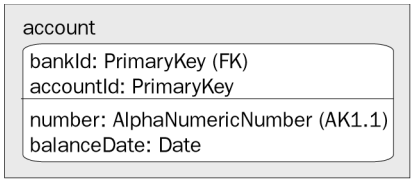
You have begun to take care of this by adding primary keys and alternate keys to your entities. Note that simply adding an artificial key will not take care of this particular rule. One of your last physical modeling tasks will be to verify that all of your entities have at least one key defined that doesn't contain a non-migrated artificial key.

Boyce-Codd Normal Form

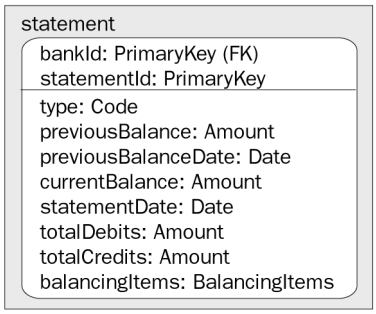
Because Boyce-Codd is actually an extension of the Second and Third Normal Forms, you can consider every one of the violations you have discussed all together.

Summary Data

Summary data attributes are generally the easiest violations to take care of. This is because you can usually just prune the values from your entities. For example, in the `account` entity, you can remove the `[Running Total]` attribute, because it can be obtained by summing values that are stored in the `transaction` entity. This leaves you with:



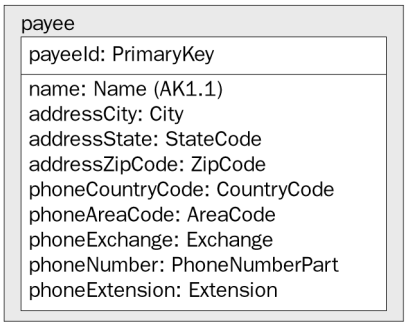
On the other hand, you have what appear to be summary attributes in the `statement` entity:



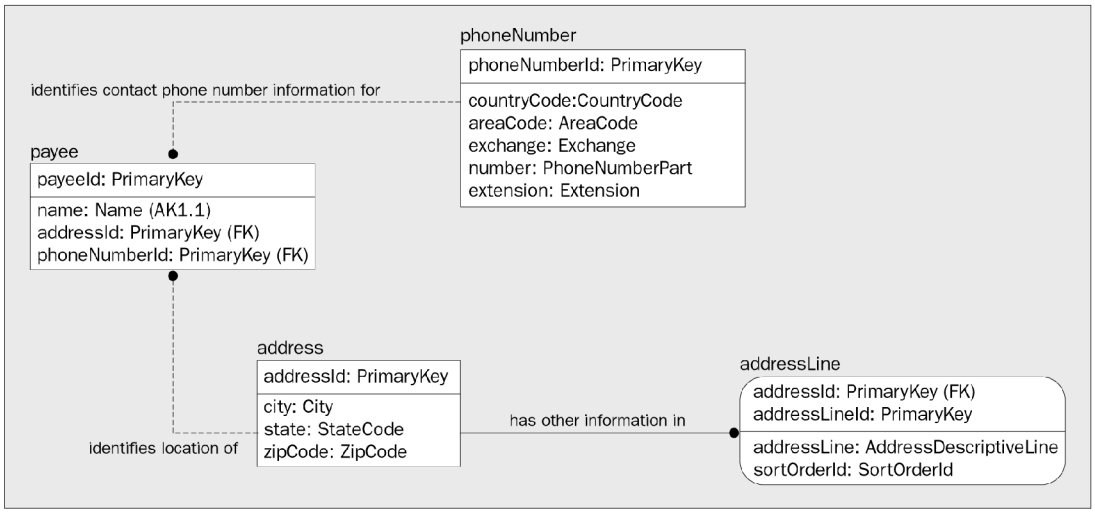
You have `previousBalance`, `currentBalance`, and so on–in fact, all of the attributes other than `type`–referring to some other entity's values. However, in this case, the `statement` entity is referring to a document, namely the statement from the bank; these attributes represent what the bank *thinks* is on the statement. You'll likely want to keep these attributes and use them to validate the data you'll be storing in the `statementItem` entity.

Multiple Attributes with the Same Prefix

You now have a very good example of this kind of problem in the `payee` entity you've created. `phoneCountryCode`, `phoneAreaCode`, and so on all have the same prefix (note that things won't always be so obvious); likewise for `addressCity`, and so on.



You can choose to say that phone numbers and addresses are logically things in and of themselves. Each of the phone attributes doesn't really describe the `payee` further, but the existence of a phone number does. The same goes for the `address` attributes. Hence you break them down further like so:



Now that you've satisfied the idea that every attribute refers to a key, because the `address` and `phoneNumber` refer to the payee's address and phone number, and the `address` is defined by its city, state, and zip, plus the street information in the `addressLine` entity. The `phoneNumber` is defined by its attributes as well.

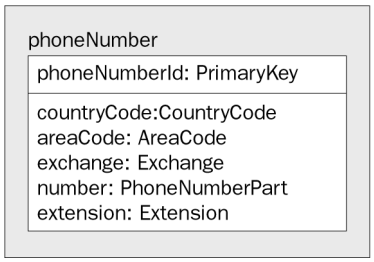
Every Determinant Must Be a Key

This is where things can get messy. Consider the `payee` entity in your previous example. The `payeeId`–a primary key–and `name` are the determinants for the entity. You have the following set of dependencies:

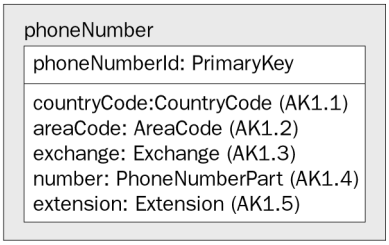
- For every value of `payeeId` you must have the same name value.
- For every value of name, you must have the same value for `addressId` and `phoneNumberId.`
- It isn't true that every value in `addressId` and `phoneNumberId` has to have the same value for name, because an address or phone number might be used for several payees.

The real issue here is in choosing a proper set of keys. Using a pointer, even in the logical modeling phase, tends to cover up a major issue. Every entity must contain unique values, and this unique value must not include meaningless values, because a pointer is meaningless except to indicate ownership in a relationship.

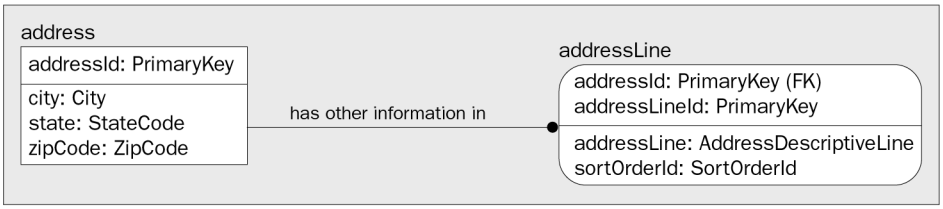
First you take the phone number entity as follows:



It has no key defined, so you must make one. In this case, the entire entity *without the primary key* will be a key. This illustrates one of the main reasons to maintain single-themed entities. Now that you have the phone number isolated, you're able to make certain that, if two payees have the same phone numbers, you don't duplicate this value. By doing this you keep things clear in your later usage of these values, because if you see duplicate phone numbers, things will get confusing.



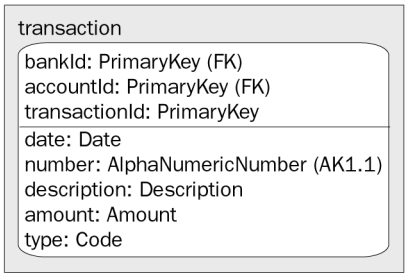
Next you'll look at the `address` entity:



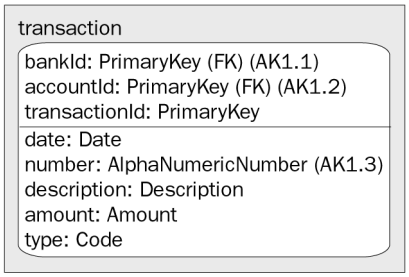
This is a very interesting case to deal with. There is no way that you can make the `city`, `state`, and `zipCode` unique, because you'll almost certainly have more than one address of the same type. In this case, the uniqueness is determined by the `city`, `state`, `zipCode`, plus any `addressLine` items, which is a valid situation. You may not have any uniqueness in the values in the `addressLine` entity either, because it's a logical part of the `address` entity, and `address` relies on it for uniqueness. You cannot model the situation that only one address may contain the information "101 Main Street" very well. This clearly will not cover every situation, unless you consider only one city, state, and zip. In cases such as this, you would have to simply document that the key of this case is actually `city`, `state`, `zipCode`, plus any number of `addressLine` values including the `sortOrderId`. Clearly this cannot be modeled in your data model, but it can be enforced using triggers.

Reality Check: The way you've modeled the address entity in two parts isn't necessarily the way an address would be actually implemented, but the situation in which two tables are required to enforce uniqueness isn't completely uncommon. What you're implementing is an attribute of type array, in which the `addressLine` attribute is actually a repeating value, which in relational language requires additional tables, whereby an array would technically be a better solution.

Finally, consider the `transaction` entity:



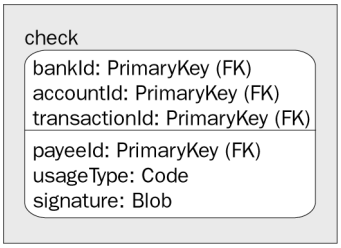
In this set of attributes, you have a key that is made up completely of migrated keys (`bankId`, `accountId`) and another artificial key. Basically, this means that the migrated account key defines part of the transaction and that it's up to you to look for the proper additional attribute. In this case it will be the `number`, because this is the number that the bank uses to identify a transaction, and you had previously chosen it as a key of its own, as follows:



You must then go through each of the other entities, making sure a proper key is assigned, and more importantly, checking that there are no attributes that have functional dependencies on another attribute.

Consider also that, now, `bankId`, `accountId`, and `transactionId` functionally determine the attribute `number`, and `bankId`, `accountId`, and `number` functionally determine the `transactionId`. This is a valid situation, which is pretty interesting!

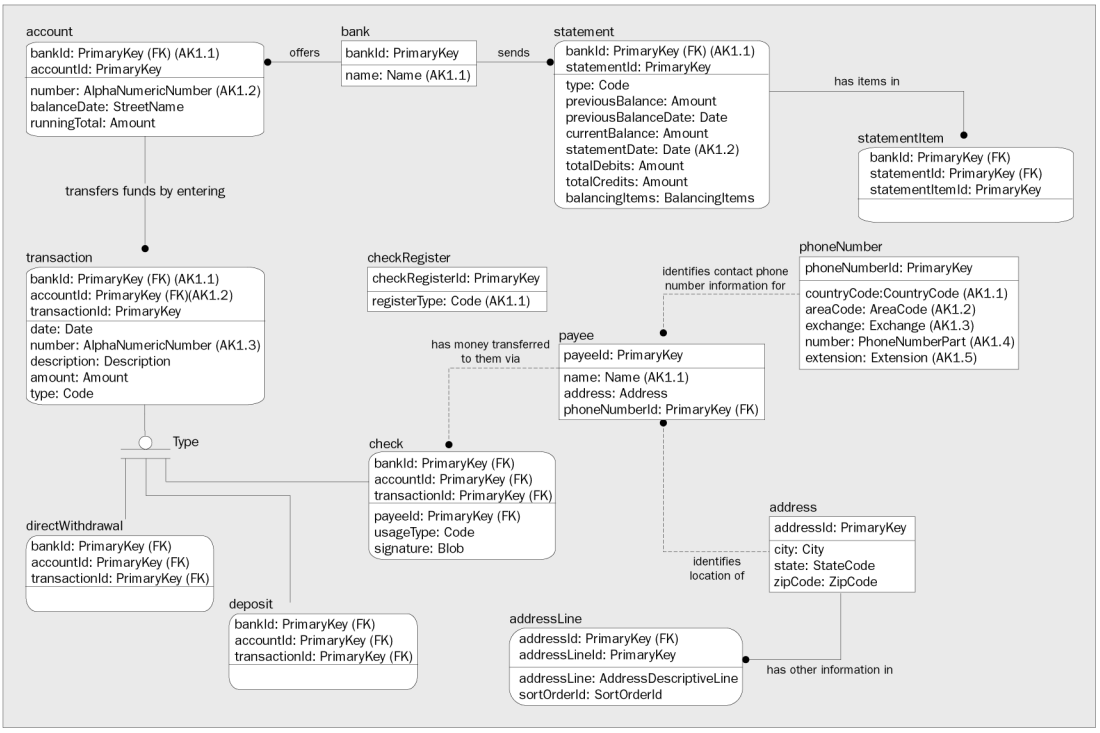
One additional point should be made concerning the following subtyped entities of `transaction`:



No additional keys are required for this entity because it's in a one-to-one relationship with the transaction entity; hence, its primary key is a valid key.

Model

Here is your model after normalizing up to the Boyce-Codd Normal Form. Note that many of the common modification anomalies should be cleared up. It's far from complete, however. To keep the model simple, you still haven't added any further information to it. You'll be presented a final model with additional columns in the last section of the logical modeling part of the book.



Best Practices

Normalization best practices are deferred until the next chapter.

Summary

In this chapter, you've begun the process of turning your random table structures into structures that will make the storage of data much more reliable. By building these structures in a very specific manner, you'll derive a set of entities that are resilient to change and that have fewer modification anomalies.

This can all seem from the outside like one of the more baffling processes within computer science. This probably explains why a lot of developers don't normalize their databases. However, there's no reason why it should be so. Behind the occasionally esoteric jargon is a set of simple, well-formed rules that lay down how databases should be designed. As for the end result, the case study should demonstrate that the resulting entities you developed are cleaner, tidier, and safer for your data entry, even though the data that is stored will look less and less meaningful to the casual observer.

Now the obvious question: Can you normalize your data any further? Because I've alerted you to the fact that there are seven normal forms of interest, and you've only looked at four, the answer to this question almost certainly is yes.

On the other hand, *should* you normalize further? Most likely! The Third Normal Form has long been identified as the most important plateau of data modeling, and it may be that, when you come to physically implement your solution, you won't need to go beyond this level. However, you're currently in the logical modeling phase and you must not concern yourself yet with performance issues; you should aim for an ideal model. Working through the next normal forms will uncover additional problems with your data that you may not have foreseen.