Chapter 3

Data Normalization

Chapter Outline

Introduction, 112

Two-Minute Chapter, 113

Tables, Classes, and Keys, 113
- *Composite Keys 114*
- *Surrogate Keys 115*
- *Notation 116*

Database Normalization: Atomic Values and Dependency, 117
- *Atomic Data Values 117*
- *Dependency 119*

Sample Database for Typical Sales, 121
- *Initial Objects 122*
- *Initial Form Evaluation 123*
- *Problems with Repeating Sections 125*

First Normal Form, 125
- *Repeating Groups 125*
- *Multiple Repeating Groups 127*
- *Nested Repeating Groups 127*

Second Normal Form, 128
- *Problems with First Normal Form 128*
- *Second Normal Form Definition 129*

Third Normal Form, 132
- *Problems with Second Normal Form 132*
- *Third Normal Form Definition 132*
- *Checking Your Work 135*

Beyond Third Normal Form, 135
- *Boyce-Codd Normal Form 136*
- *Fourth Normal Form 137*
- *Domain-Key Normal Form 137*
- *Summary 139*

Data Rules and Integrity, 139

The Effects of Business Rules, 141

Converting a Class Diagram to Normalized Tables, 143
- *One-to-Many Relationships 144*
- *Many-to-Many Relationships 146*
- *N-ary Associations 147*
- *Generalization or Subtypes 149*
- *Composition 150*
- *Recursive (Reflexive) Associations 151*

The Pet Store Example, 151

View Integration, 153
- *The Pet Store Example 154*
- *Rolling Thunder Sample Integration Problem 156*

Data Dictionary, 162
- *DBMS Table Definition 163*
- *Data Volume and Usage 166*

Summary, 168

Key Terms, 170

Review Questions, 170

Exercises, 171

Web Site References, 179

Additional Reading, 179

Appendix: Normalization, 180

What You Will Learn in This Chapter

- Why is database design important?
- What is a table and how do you choose keys?
- What are the fundamental rules of database normalization?
- How do you begin analyzing a form to create normalized tables?
- How do you create a design in first normal form?
- What is second normal form?
- What is third normal form?
- What problems exist beyond third normal form?
- How does a database record constraints?
- How do business rules change the database design?
- What problems arise when converting a class diagram to normalized tables?
- What tables are needed for the Sally's Pet Store?
- How do you combine tables from multiple forms and many developers?
- How do you record the details for all of the columns and tables?

A Developer's View

Miranda: That was actually fun. I learned a lot about the company's procedures and rules. I think I have everything recorded properly on the class diagram; along with some notes in the data dictionary.

Ariel: Great! We should go to the concert tonight and celebrate.

Miranda: I could use a night off. Maybe giving my brain cells a rest will help me figure out what to do next.

Ariel: What do you mean? How much longer do you think the project will take?

Miranda: That's the problem. I put all this time in, and I don't really have a start on the application at all.

Ariel: Well, isn't the data the most important aspect to building a database application? I heard that database systems are touchy. You have to define the data correctly the first time; otherwise, you will have to start over.

Miranda: Maybe you're right. I'll take the night off; then I'll study these rules to see how I can turn the class diagram into a set of database tables.

Getting Started
Refine your table definitions by double-checking the primary keys. Then examine each non-key column to ensure that it depends on the whole key and nothing but the key. Each table should represent a single concept and all business rules should be explicit. There can be no hidden dependencies.

Introduction

Why is database design important? A database management system is a powerful tool. It provides many advantages over traditional programming and hierarchical files. However, you get these advantages only if you design the database correctly. Recall that a database is a collection of tables. The goal of this chapter is to show you how to design the tables for your database.

The essence of data normalization is to split your data into several tables that will be connected to each other based on the data within them. Mechanically, this process is not very difficult. There are perhaps four rules that you need to learn. On the other hand, the tables have to be created specifically for the business or application that you are dealing with. Therefore, you must first understand the business, and your tables must match the rules of the business. So the challenge in designing a database is to first understand how the business operates and what its rules are. Some of these rules were hinted at in Chapter 2 with the focus on relationships. Business relationships (one-to-one and one-to-many) form the foundation of data normalization. These relationships are crucial to determining how to set up your database. These rules vary from firm to firm and sometimes even depend on which person you talk to in the organization. So when you create your database, you have to build a picture of how the company works. You talk to many people to understand the relationships among the data. The goal of data normalization is to identify the business rules so that you can design good database tables.

By designing database tables carefully, you (1) save space, (2) minimize duplication, (3) protect the data to ensure its consistency, and (4) provide faster transactions by sending less data. One method for defining database tables is to use the graphical approach presented in Chapter 2 and build a class diagram. A related method is to collect the basic paperwork, starting with every form and every report you might use. Then take apart each collection of data and break it down into respective tables. Most people find that a combination of both approaches helps them find the answer. However, the discussion will begin by describing the two methods separately.

Two-Minute Chapter

Database design takes practice and experience. In the end, the design encapsulates the rules and relationships in the underlying business problem. Chapter 2 emphasized that tables represent business objects and each table must have a primary key. Composite keys (multiple columns) in a table indicate many-to-many relationships. Tables are linked together by the data in keys. For example, The Sale table has a primary key of SaleID but CustomerID is a foreign key column in the Sale table. This approach saves space and prevents other problems because only the CustomerID number is stored for each Sale. Instead of repeating all Customer data for every Sale, the number refers back to the detailed information in the Customer table.

When deciding which columns belong in each table, the three primary rules of normalization are: (1) each entry must be atomic or single-valued not repeating, (2) each non-key column must depend on the whole key, and (3) each non-key column must depend on nothing but the key. Another general way to look at the problem is to note that there can be no hidden dependencies. If some business relationship or rule exists, it needs to be defined as its own table. The hard part is identifying these business rules. Some of them can be determined from existing forms and data. Others have to be elicited through interviews and discussions with managers.

This approach leads to tables that can efficiently store data with minimal problems. However, you have to carefully evaluate every table, every key, and every column. Watch for many-to-many relationships, and understand the concept of *dependence*.

Tables, Classes, and Keys

What is a table and how do you choose keys? Chapter 2 focuses on identifying the business classes and associations. Now, these classes need to be more carefully defined so they can be converted into database tables. Of course, as you modify the tables, you will also update the class diagram. The relationships among the classes are critical to determining the final form of the tables. These relationships are also expressed in terms of the primary keys of the tables. Remember that a primary key consists of a collection of columns that uniquely identify each row. Since the key must be guaranteed to always be unique, it is common to create a new key column that holds generated keys. But, in many cases, you will use multiple columns to make up the primary key. These situations are important enough to require a detailed explanation.

Orders

OrderID	Date	Customer
8367	5-5-04	6794
8368	5-6-04	9263

OrderItems

OrderID	Item	Quantity
8367	229	2
8367	253	4
8367	876	1
8368	555	4
8368	229	1

Figure 3.1

Composite keys. OrderItems uses a composite key (OrderID + Item) because there is a many-to-many relationship. Each order can contain many items (shown by the solid arrows). Each item can show up on many different orders (dotted arrows).

Composite Keys

In many cases, as you design a database, you will have tables that will use more than one column as part of the primary key. These are called **composite keys**. You need composite keys when the table contains a one-to-many or many-to-many relationship with another table.

As an example of composite keys, look at the OrderItems table in Figure 3.1. These two tables are common in business and they form a **master-detail** or parent-child relationship. The Orders table is straightforward. It has one column as a primary key, where you created the OrderID. This table contains the basic information about an order, including the date and the customer. The OrderItems table has two columns as keys: OrderID and Item. The purpose of the OrderItems table is to show which products the customers chose to buy. In terms of keys the important point is that each order can contain many different items. In the example OrderID 8367 has three items. Because each order can have many different items, Item must be part of the key. Reading the table description from left to right you can say that each OrderID may have many Items. The "many" says that Item must be keyed. What about the other direction in the OrderItems table? Do you really need to key OrderID? The answer is yes because the firm can sell the same item to many different people (or to the same customer at different times). For example, Item 229 appears on OrderIDs 8367 and 8368. Because each item can appear on many different orders, the OrderID must be part of the primary key. For comparison, reconsider the Orders table in Figure 3.1. Each OrderID can have only one Customer, so Customer is not keyed.

To be sure you understand how keys and relationships interact, look again at the OrderItems table. Looking at the ItemID column, ask yourself: For each OrderID, can there be one or many ItemIDs? If the answer is many, then ItemID must be keyed (underlined). Now, look at OrderID and ask yourself, Can an ItemID appear on one or many orders? Again, the answer is many, so OrderID must also be keyed.

Look at the CustomerID column in the Order table and ask, For each order, can there be one or many customers? The common business rule says there is only

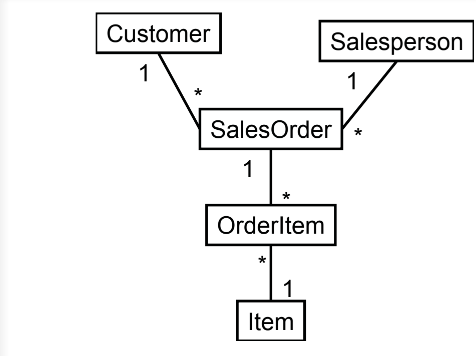


Figure 3.2

A small class diagram for a basic order system. The numbers indicate relationships. For instance, each customer can place many orders, but a given order can come from only one customer.

one customer per order, so CustomerID is not part of the primary key. On the other hand, because CustomerID is a primary key within the Customer table, it is known as a **foreign key** in the Order table. Think of it as a foreign dignitary visiting a different country (table). It is required in the Order table because it serves as a link to the rest of the customer data in the Customer table; but it does not have to be a key (king) in that table.

To properly normalize the data and store the data as efficiently as possible, you must identify keys properly. Your choice of the key depends on the business relationships, the terminology in the organization, and the one-to-many and many-to-many relationships within the company.

Surrogate Keys

It can be difficult to ensure that any real-world data will always generate a unique key. Consequently, you will often ask the database system to generate its own key values. These **surrogate keys** are used only within the database and are often hidden so users do not even know they exist. For example, the database system could assign a unique key to each customer, but clerks would look up customers by conventional data such as name and address. Surrogate keys are especially useful when there is some uncertainty with the business key. Think about SalesID or PurchaseOrderID which need to be assigned at the time of each sale or purchase. How can a person create these to guarantee they are unique? Numbers such as

Figure 3.3

Table notation. Column details are easier to see in a simple listing of the tables. This list is also useful when the tables are entered into the database.

Customer(CustomerID, Name, Address, City, Phone)
Salesperson(EmployeeID, Name, Commission, Datehired)
SaleOrder(OrderID, OrderDate, CustomerID, EmployeeID)
OrderItem(OrderID, ItemID, Quantity, SalePrice)
Item(ItemID, Description, ListPrice)

1. Each cell in a table contains atomic (single-valued) data.
2. Each non-key column depends on all of the primary key columns (not just some of the columns).
3. Each non-key column depends on nothing outside of the key columns.

Figure 3.4

The three main rules for data normalization. Essentially, each table has to accurately represent the business definitions. Each table represents a single entity and the keys accurately identify the entity and represent the one-to-many relationships among the attributes.

CustomerID and EmployeeID could be defined by the marketing or HRM departments, but it is simpler to just let the DBMS create unique values when they are needed.

The use of surrogate keys can be tricky when the database becomes large. With many simultaneous users, creating unique numbers becomes more challenging. Additionally, several performance questions arise involving surrogate keys in large databases. For example, a common method of generating a surrogate key is to find the largest existing key value and increment it. But what happens if two users attempt to generate a new key at the same time? A good DBMS handles these problems automatically.

Microsoft Access uses the **autonumber** data type to generate unique numbers for key columns. Similarly, SQL Server uses the Identity data type. Oracle has a SEQUENCES command to generate unique numbers, but it operates differently than the Microsoft approaches. As a programmer, you generate and use the new values when a new row is inserted—the process is not automatic (but you can automate it with a couple lines of code). There are advantages and drawbacks to both approaches. The biggest difficulty with the Microsoft approach is that it is sometimes difficult to obtain the new value that was generated when a row is inserted. The drawback to the Oracle approach is that you must ensure that all users and developers use the proper number generation commands throughout the application. Additionally, both approaches can cause problems when transferring data—particularly to other database systems.

Generated numbers are even trickier in distributed databases—where new numbers must be generated in multiple locations. One approach is to assign different ranges to each location so each location generates a different type of number. A second approach is the **globally-unique identifier (GUID)** which is essentially a very large random number. Microsoft software has tools for generating 128-bit GUIDs. Oracle also has functions for generating GUIDs. GUIDs are rarely sequential and they are large numbers. The point is that although they are useful for creating key values, think of generated keys as random values, and you almost always want to hide these numbers from the users.

Notation

A detailed class diagram can describe each table and include all properties within each class and marked key columns. The advantage to using class diagrams is that they highlight the associations among the classes. Additionally, some people understand the system better with a visual representation. Figure 3.2 shows a simple example class diagram, but it leaves out the properties.

The drawback to class diagrams is that they can become very large. By the time you get to 30 classes, it is hard to fit all the information on one page. Also, many of the association lines will cross, making the diagram harder to read. CASE tools help resolve some of these problems by enabling you to examine a smaller section of the diagram.

However, you can also use a shorter notation, as shown in Figure 3.3. The notation consists of a straight listing of the tables. Each column is listed with the table name. The primary keys are underlined and generally listed first. This notation is easy to write by hand or to type, and it can display many tables in a compact space. However, it is hard to show the relationships between the tables. You can draw arrows between the tables, but your page can become messy.

Designers frequently create both the class diagram and the list of tables. The list identifies all of the columns and the keys. The class diagram shows the relationships between the tables. The class diagram can also contain additional details, such as existence constraints and minimum requirements.

Database Normalization: Atomic Values and Dependency

What are the fundamental rules of database normalization? Database researchers have shown that if tables are not designed carefully, several serious problems can arise. These problems can be avoided by following some basic rules. The primary rules are written as the first three normal forms. Figure 3.4 lists the most important rules. These rules are explained in detail in the following sections. You need to understand each rule in detail because you use them to improve your designs and ensure that the tables you create accurately reflect the business rules. One way to think about the rules is that every database table represents the business rules so that there are no hidden relationships. Everything is spelled out correctly in the tables.

When you first read the rules, they seem slightly confusing because they rely on some special terms. In fact, understanding the rules basically comes down to understanding two specific concepts: atomicity and dependency. These terms are described in this section and the following sections describe how to apply them to real-world problems.

Atomic Data Values

The first rule is the easiest to understand and one of the most important. However, sometimes it can be tricky to apply. A table cell contains an **atomic** value if there is only a single non-repeating item. Consider a Customer table that shows up in most business databases. A Name column in a Customer table can contain only one name on each row. That example certainly seems simple. Why would anyone ever try to store multiple names in one cell? People have only one name anyway? Wait a second. What about first name and last (family) name? If you enter "John Doe" into a Name column, is that one name or two? The answer is that it depends on how the data will be used. If you want to sort the rows based on last name and then first name, you really need to create two columns: LastName and FirstName instead of just the Name column. Yes, you could write a special function that would split a single name into its two components. But, needing to write special functions to extract data from a cell is a major sign that your database is violating the rule of atomic data. Now, if the users always look at the entire name as a single thing, it is fine to store the entire name in one column. This example highlights one of the most important things you need to learn: the database design depends

CustomerID	LastName	FirstName	Phone	Fax	CellPhone
15023	Jones	Mary	222-3034	222-4094	223-0984
63478	Sanchez	Miguel	030-9693	403-4094	
94552	O'Reilly	Madelline	849-4948	292-3332	139-3831
45791	Stein	Marta	294-4421		
49004	Brise	Mer	764-5103		

Figure 3.5

Atomic values for phone numbers. Is phone number atomic (single-valued) or repeating? You might add a column for each possible type of phone number. But how many customers have each type of phone and what if more types are needed in the future?

heavily on the business rules and assumptions. In fact, the design is a model of the business because the tables, columns, and keys reflect the business rules.

The Name column is relatively easy. In practice, most designers do split Name into LastName and FirstName columns. Now, look at some other columns in the potential Customer table. Many companies want to store the customer's phone number, so you can add a Phone column to the table. But once again, you have to ask: Can a customer have more than one phone number? Twenty years ago, this question was easy to answer as "no," customers have only one phone number. Today, you might want to add a cell phone number or business number. Figure 3.5 shows how you might add three types of phone numbers to a table. It certainly appears that each cell contains a single value. However, notice that customers might not have all three numbers. Also, what will you do if a customer has a fourth or fifth phone number?

Figure 3.6

Repeating values for phone numbers. Each customer can have from one to many phone numbers. This table would be a bad design because it would cause problems with retrieving or editing an individual phone number.

CustomerID	LastName	FirstName	Phone
15023	Jones	Mary	222-3034 222-4094 223-0984
63478	Sanchez	Miguel	030-9693 403-4094
94552	O'Reilly	Madeline	849-4948 292-3332 139-3831 339-4040
45791	Stein	Marta	294-4421
49004	Brise	Mer	764-5103

CustomerID	LastName	FirstName
15023	Jones	Mary
63478	Sanchez	Miguel
94552	O'Reilly	Madeline
45791	Stein	Marta
49004	Brise	Mer

CustID	PhoneType	Phone
15023	Land	222-3034
15023	Fax	222-4094
15023	Cell	223-0984
63478	Land	030-9693
63478	Fax	403-4094
94552	Land	849-4948
94552	Fax	292-3332
94552	Cell	139-3831
94552	Laptop	339-4040
45791	Land	294-4421
49004	Land	764-5103

Figure 3.7

Repeating values for phone numbers. Split the phone numbers into a separate table. Include the key (CustomerID) to link back to the original Customer table.

Today, it is possible that phone number is a repeating (or multi-valued) entity. Figure 3.6 shows another possible way of looking at the phone number data. At least with this approach, you would not have to guess at the number of phone number columns. However, it would be difficult to find or edit individual phone numbers, so you would never actually store the data this way. You could make do with the multiple columns shown in Figure 3.5, but it will waste space when you get beyond two or three types of numbers.

So what is a better answer? The solution to each of the normalization steps is always the same: Split the table into two tables. In this case, the phone number is causing the problem, so you need to put the phone numbers into a separate table. The one catch is that you must bring along the key column (CustomerID) so you will be able to join the phone numbers back to the customers.

Figure 3.7 shows the resulting two tables. Look at the sample data to see how the problems have been solved. Each phone number is listed in a separate row in the CustomerPhones table. A customer with a single phone number takes only one row of space. Yet, the table can hold data for as many phone numbers per person as you will need—even if new phone types are added later.

The example of phone numbers as repeating data is tricky. Most designers would probably choose to go with the method in Figure 3.5 by using multiple columns. However, as the number of phone types proliferates, you might need to switch to the version in Figure 3.7 to reduce the wasted space. In every case, the overtly repeating version in Figure 3.6 would be wrong. Fortunately, most situations of repeating data are considerably more obvious. Usually, you can identify repeating sections of data on a form because multiple lines are provided for entering data.

Dependency

Although researchers have defined dependency in mathematical terms, the concept is truly an issue of business rules. You can read the formal definitions in

CustomerID Company Name
City
Contact Last Name, First Name
Phone

Figure 3.8

A portion of a form for a firm that sells products to other companies. The dependencies among the attributes have to be identified based on common business practices.

the appendix, but the discussion in the chapter focuses on the business rules and uses language to explain dependency. If one attribute always identifies a specific value for another attribute, the second attribute is said to **depend** on the first. For instance, if someone gives you the CustomerID of 15023, you know that the LastName will always be Jones. It can never be anything else. Hence, LastName depends on CustomerID. Which would mean that CustomerID is a good candidate for becoming the primary key. On the other hand, if you were given the LastName of Jones, you probably would not be able to identify just one phone number. Jones is a fairly common name and the company probably has many customers with that name, so many phone numbers would show up for that name. Consequently, it is wrong to say that Phone depends on LastName. LastName would be a bad choice for a key column.

How do you know when one column depends on another? This question is the most difficult problem you face when designing a database. In fact, if someone were to tell you every single dependency in a case, it is easy to create the appropriate database design. In fact, you could create a program to build the design automatically (it has been done already). In other words, when you build a database design, you are really just identifying the business rules that specify how attributes (columns) depend on each other. Did you notice that the original question has not been answered? The answer is that you must talk with the users, study the forms and reports, and identify the dependency rules specifically for each situation.

To create an example, you need to know the basic business rules. Writing the business rules out would give away the answers, so usually you will be asked to study forms and reports to identify dependencies. However, you still need to use your business judgment. (It truly is important to take all of those other business courses—they will help you identify common business rules.) Consider a firm that sells products to other companies. Figure 3.8 shows a portion of an order form. You need to use your knowledge of common business rules to determine the dependencies among the attributes.

As a first attempt, you might try putting all of the attributes together into a single table. The CustomerID looks like it would make a good primary key. But, now you have to look at each potential column and ask yourself a basic question: (1) For a given value of the CustomerID, can there be more than one value of this attribute? If the answer is "yes," you must move the questioned attribute into a new table.

In the customer example, it is clear that the CompanyName depends on the CustomerID. The City attribute might be trickier. A customer could have offices in several cities. And, it might be critically important to your application to store the data this way. However, in many situations you can simply assume that City refers

Customer(CustomerID, CompanyName, City)
Contact(ContactID, CustomerID, LastName, FirstName)

Figure 3.9

The two main tables for the customer case. The key columns in the Contact table reveal that there can be many ContactIDs at each customer, but each contact works for only one customer. Where does the phone number belong?

to the corporate headquarters in a single city. You should record this assumption in your design notes so others can understand your design decisions. The Contact last and first names are more important. Although you might currently have only one contact at a customer firm, it is more reasonable to assume that in the future you will have multiple contacts at a company. From the repeating rule, you need to create a new table for the contacts. Figure 3.9 shows the initial tables. Notice that CustomerID is not part of the key in the Contact table because each contact person works for only one customer.

Now, where do you put the phone number column? This question illustrates the concept of dependency. Does the phone depend on (refer to) the customer or a contact? Technically, it could refer to either one. Most companies have a primary switchboard number that you could call. However, it is a generic number and requires you to go through several steps to find the person you want. It is more likely that the users of the system want the phone number of the specific contact person so that person can be reached directly. Consequently, Phone depends on ContactID and belongs in the Contact table instead of the Customer table. Figure 3.10 shows the resulting database design section.

Sample Database for Typical Sales

How do you begin analyzing a form to create normalized tables? The best way to illustrate data normalization is to examine a sample problem. Remember that the results you get (the tables you create) depend heavily on the specific example and the assumptions you make. The following example uses a basic Sales Order form. The sample data are from fictional sales at a SCUBA dive shop, but the actual items could be anything and the principles will remain the same.

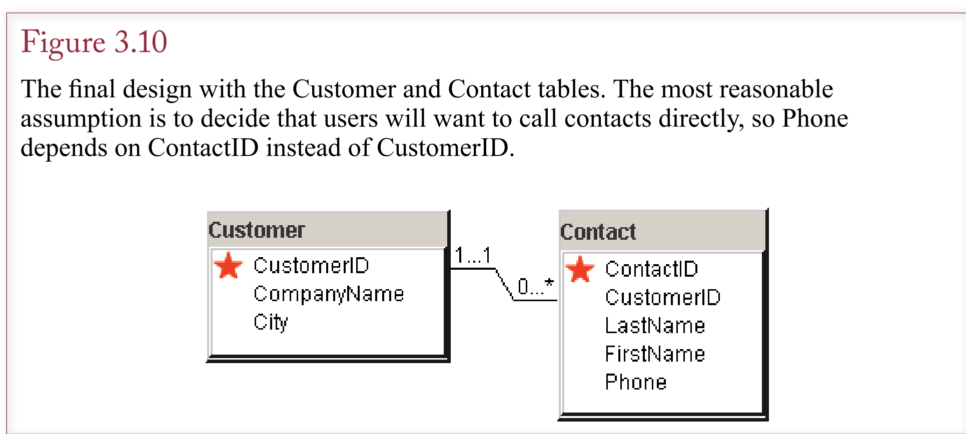


Figure 3.10

The final design with the Customer and Contact tables. The most reasonable assumption is to decide that users will want to call contacts directly, so Phone depends on ContactID instead of CustomerID.

SaleID Date
Customer
First Name
Last Name
Address
City, State ZIPCode

ItemID	Description	List Price	Quantity	QOH	Value
					Total

Figure 3.11

Sample sales form. First look for possible keys, keeping in mind that repeating sections (one-to-many relationships) will eventually need composite keys.

Figure 3.11 shows a basic sales order form. The main components of the sample form are the customer and the items being sold. When the form is built in the database, it will automatically keep track of the total amount due. It should also automatically assign a SaleID that is unique. The database form will also have buttons and drop-down lists to help the user enter data with a minimum of effort. For now, as you talk with the manager, you should sketch the desired features of the form. Values that can be computed (e.g., subtotals) should be marked, and the appropriate equations provided if needed. For the most part you do not want to store computed values in the data tables.

Initial Objects

One way to begin the design process is to identify the primary objects on a form. This step helps you think about the overall design and provides a start at identifying the tables. A form generally has several obvious entities. In this case, the obvious ones are customers and items. In real life you would also have employees. Managers also need to keep track of who purchased specific items. For example, if a manufacturer finds a problem with a piece of diving gear, the manager wants to send a notice to any customer who purchased that item. Hence you need two additional objects. The first is a transaction that records the date and the customer. It represents the overall form itself. The second is a list of the items purchased by that customer at that time. It represents the repeating section of the form.

Examine the initial objects in Figure 3.12. You need a primary key for customers (and items). Clearly, Name will not work, but you might consider using the Phone number. This approach would probably work, but it might cause some minor difficulties down the road. For example, if a customer gets a new phone number, you would have to change the corresponding phone number in every table that referred to it. As a primary key, it could appear in several different tables. A bigger problem would arise if a customer (Adams) moves, freeing up the phone number, which the phone company reassigns to another person (Brown) several months later. If Brown opens an account at your store, your database might mis-

Initial Object	Key	Sample Properties
Customer	Assign CustomerID	Name Address Phone
Item	Assign ItemID	Description List Price Quantity On Hand
Sale	Assign SaleID	Sale Date
SaleItems	SaleID + ItemID	Quantity

Figure 3.12

Initial objects for the sale form. Note that the transaction has two parts, Sale and SaleItems because many items can be sold at many different times.

takenly identify customer Brown as the customer Adams. The safest approach is to have the database create a new number for every customer.

The Item object also needs a key. In practice you might be able to use the product identifiers created by the manufacturer. For now, it is easiest to assign a separate number. Basic properties include the item description, list price and quantity on hand (QOH). More attributes (such as size) can be added later if necessary.

Every transaction must be recorded. The transaction in this case is the entire sale. This object refers to the overall sale form and is also assigned a unique key value. Remember this approach. Almost all of the problems you encounter will end up with a table to hold data for the base form or report.

An important issue in many situations is the presence of a repeating section, which can cause problems for storing data. Hence, the section is split from the main transaction and stored in its own table. Keys here include the SaleID from the Sale table and the ItemID. Note that the key is composite because a many-to-many relationship exists. A customer can buy many products at one time and a product can be purchased (at different times) by more than one customer.

Initial Form Evaluation

Practice is required to identify all of the tables needed for a form or report. In the Sales Order example, most people should be able to identify the Customer and Item tables. Some will recognize the need for a Sale table. However, the purpose of the SaleItems table is not as clear. Fortunately, there is a method to derive the individual tables by starting with the entire form and breaking it into pieces. This method is the data normalization approach, and it is a mechanical process that follows from the business assumptions.

Figure 3.13 shows the first step in the evaluation. As you learn normalization, you should be careful to write out this first step. As you gain experience, you might choose to skip this step. The procedure is to look through the form or report and write down everything that you want to store. The objective is to write it in a structured format. Give the form a name and list the items as column names. You can generally start at the top left of the form and write a column name for each data element. Try to list items together that fall into natural groupings—such as all customer data. The SaleForm begins with the SaleID, which looks like it would make a good key. The SaleDate and CustomerID are listed next, followed by the

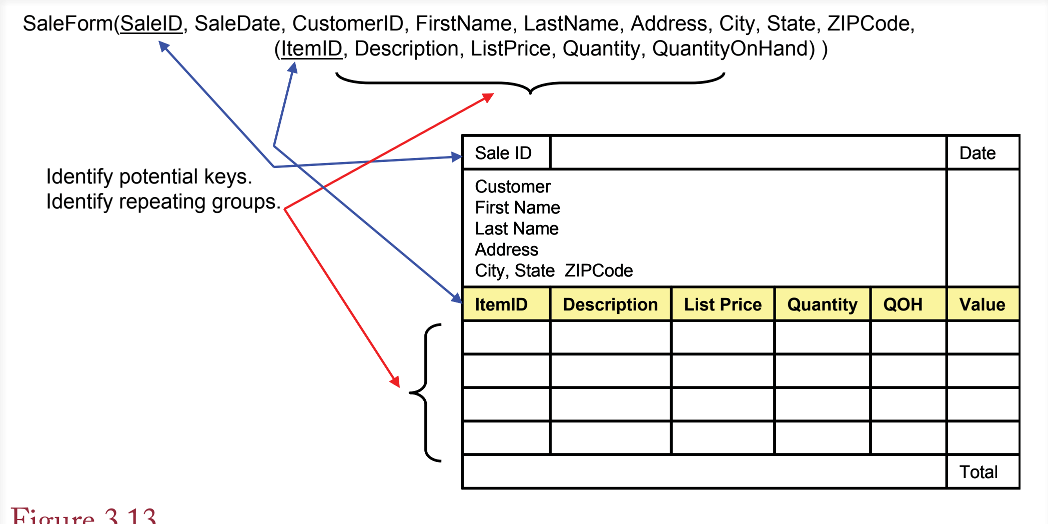


Figure 3.13

Initial form evaluation. Once you have collected basic user forms, you can convert them into a more compact notation. The notation makes the normalization steps easier by highlighting potential issues.

basic customer data. The next step is slightly more complicated because you have to signify that the section with the items contains repeating data. That is, it has multiple lines of data or the potential for several similar entries. Repeating data represents a one-to-many relationship that must be handled carefully. An easy way to signify the repeating section is to put it inside another set of parentheses. Some people also list it on a new line.

Observe that the computed total was not included, since it can be recalculated as needed. However, in some cases you might want to store computed data. For instance, if you compute a sales tax with each order, it is convenient to store the computed value. Even though the tax could be recomputed later, changing tax rates and round-off differences might lead to errors in the later calculations. The reduction in risk is worth the small extra storage of data. However, you should mark the items or add a description so you remember they are computed values.

While you are working on the first step, be sure to write down every item that you want to store in the database. In addition, make sure to identify every repeating section. Here you have to be careful. Sometimes repeating sections are obvious: They might be in a separate section, highlighted by a different color, or contain sample data so you can see the repetition. Other times, repeating sections are less obvious. For example, on large forms repeating sections might appear on separate pages. Other times, some entries might not seem to be repeating. You should also try to mark potential keys at this point, both to indicate repeating sections and to highlight columns that you know will contain unique data. On some DBMSs, you can create a **pseudo column** to define a computed value. This value is not actually stored, but recomputed as needed. For instance, the Value column could be computed as price times quantity. Finally, when you write down a column name, you should add it to a data dictionary and record attributes such as the data type and which person is responsible for the item.

SaleForm(<u>SaleID</u>, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode, (<u>ItemID</u>, Description, ListPrice, Quantity, QuantityOnHand))

Repeating section

Duplication

Not atomic

<u>SaleID</u>	Date	CID	FirstName	LastName	Address	City	State	ZIP	ItemID	Description	ListPrice	Quantity	QOH
11851	7/15	15023	Mary	Jones	111 Elm	Chicago	IL	60601	15 27 32	Air Tank Regulator Mask 1557	192.00 251.00 65.00	2 1 1	15 5 6
11852	7/15	63478	Miguel	Sanchez	222 Oro	Madrid			15 33	Air Tank Mask 2020	192.00 91.00	4 1	15 3
11853	7/16	15023	Mary	Jones	111 Elm	Chicago	IL	60601	41 75	Snorkel 71 Wet suit-S	44.00 215.00	2 1	15 3
11854	7/17	94552	Madeline	O'Reilly	333 Tam	Dublin			75 32 57	Wet suit-S Mask 1557 Snorkel 95	215.00 65.00 83.00	2 1 1	3 6 17

Figure 3.14

Problems with repeating data. Storing repeating data with the main form results in either non-atomic cells or substantial duplication of data.

Problems with Repeating Sections

The reason you have to be so careful in identifying repeated sections or one-to-many relationships is that they can cause problems in the database. The situation in Figure 3.14 shows what happens when you try to store the data from the form exactly the way it is written now. In particular, the repeating section causes problems. As it is displayed, it results in non-atomic data in the cells because of the need to store multiple items for each order. You might try to avoid the problem by storing each item in a separate row, but then you would have to duplicate the sale and customer data for each item being sold. The other problems with this attempt are explained in the next sections, but you might as well solve the problem now.

Several other problems arise because of this weak design. What do you know about products that have not been sold yet? Conversely, what if you delete old data, such as all of last year's sales? As you delete sales, you also delete item and customer data. Suddenly, you notice that you deleted half of the customer base. Technically, these problems are known as an **insertion anomaly** and a **deletion anomaly**; that is, when the data is not stored in a proper format, you encounter difficulties as you try to add or delete data. These problems arise because you tried to store all the data in one table.

First Normal Form

How do you create a design in first normal form? The answer to the problem with repeating sections is to put them into a separate table. When all cells contain atomic data, (for example, a table has no repeating groups), it is said to be in **first normal form (1NF)**. That is, for each cell in a table (one row and one column), there can be only one value. This value is atomic in the sense that it cannot be decomposed into smaller pieces.

Repeating Groups

As shown in some of the prior examples, some **repeating groups** are obvious. Others are more subtle and deciding whether to split them into a separate table is more difficult. The first normalization rule is clear: If a group of items repeats,

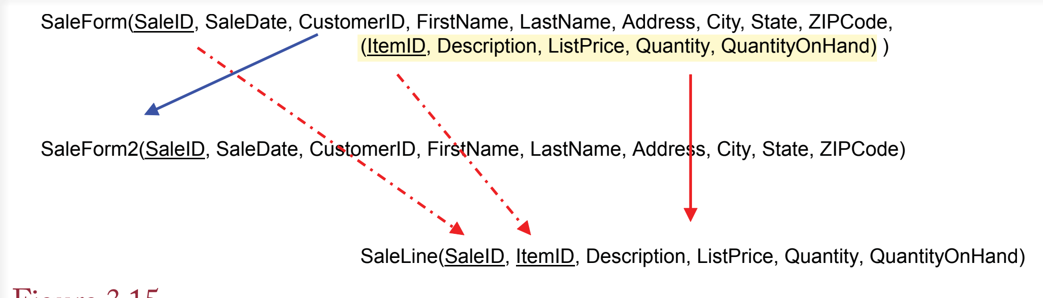


Figure 3.15

First normal form. All repeating (non-atomic) groups must be split into new tables. Be sure that the new table includes a copy of the key from the original table. The table holding the repeating group must have a composite key so that the data can be recombined in queries.

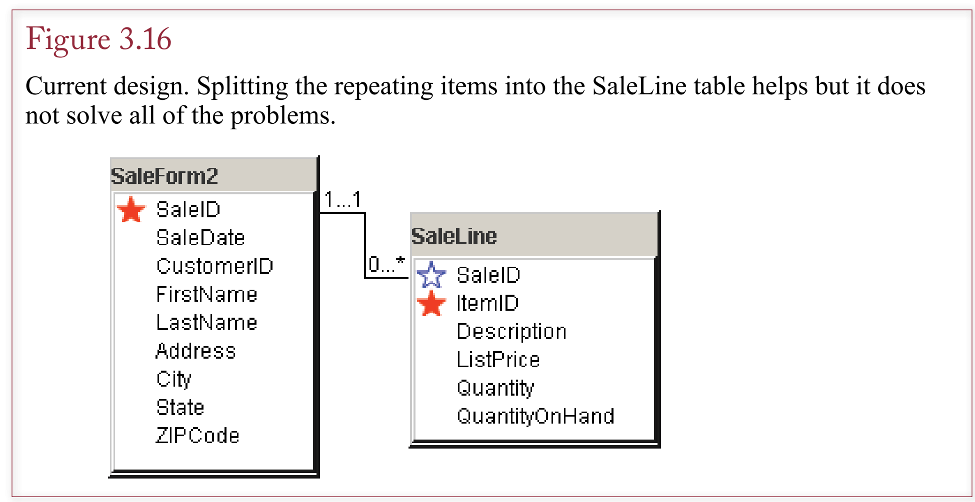
it should be split into a new table. The solution in all cases is the same: Split the design into two tables. If repeating groups remain, split the design again.

Return to the scuba store example, as shown in Figure 3.15, and notice the repeating section that is highlighted by the parentheses. To split this form, first separate everything that is not in the repeating group. These columns might need other changes later, but the section contains no repeating groups. Second, put all the columns from the repeating item sales section into a new table. However, be careful. When you pull out a repeating section, you must bring down the key from the original table. The SaleForm table has SaleID as a primary key. This key, along with the ItemID key, must become part of the new table SaleLine. You need the Sale key so that the data from the two tables can be recombined later. Note that the new table (SaleLine) will always have a composite key—signifying the many-to-many relationship between sales and items.

Figure 3.16 shows the current design in the database design system. Keep in mind that this design is merely the first step. Just glancing at the figure should tell you there is a serious problem—because the SaleLine table contains both a generated key and a non-generated key column. Remember, because there is a many-to-many relationship between Sale and Item, you eventually need a table

Figure 3.16

Current design. Splitting the repeating items into the SaleLine table helps but it does not solve all of the problems.



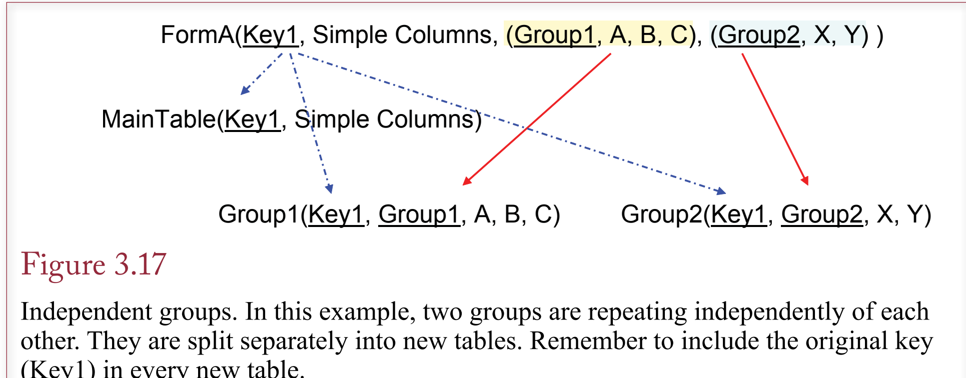


Figure 3.17

Independent groups. In this example, two groups are repeating independently of each other. They are split separately into new tables. Remember to include the original key (Key1) in every new table.

that contains both SaleID and ItemID as keys. The point to remember is that this design is in 1NF, which means it is better than putting everything into a single table—but not much better.

Splitting off the repeating groups solves several basic problems. First, it reduces the duplication: You no longer have to enter customer data for every item that is sold. In addition, you do not have to worry about allocating storage space: Each item sold will be allocated to a new row.

Multiple Repeating Groups

Before looking at the next step in normalization, you should realize that repeating sections can be considerably more complicated. Remember that you pick up the initial design from forms and reports, and you will be amazed at the complexity that can arise on business forms. Two common situations are: (1) independently repeating groups, and (2) nested repeating groups.

Many forms will have several different groups that repeat. As shown in Figure 3.17, if they repeat independently of each other, the split is straightforward; each group becomes a new table. Just be careful to include the original key in every new table so the tables can be linked together later. Using the base notation, groups are independent if the parentheses do not overlap. For example, a more complex sales case could have a second repeating group of items that are being leased. The leased items would often be stored separately because additional lease data is needed for each item.

Nested Repeating Groups

More complicated situations arise when several different repeating groups occur within a table—particularly when one repeating group is nested inside another group. The greatest difficulty lies in identifying the nested nature of the groups. As illustrated in Figure 3.18, after you identify the relationships, splitting the tables is straightforward. Just go one step at a time, pulling the outermost groups first. Always remember to bring along the prior key each time you split the tables. So when you pull the second group (Key2 … (Key3 …)) from the first group (Key1 …), the new TableA must include Key1 and Key2. When you pull Table3 from TableA, you must bring along all the prior keys (Key1 and Key2) and then add the third key (Key3).

A more sophisticated sale problem could encounter nested repeating groups. For example, the store might sell (and ship) items to several departments for the

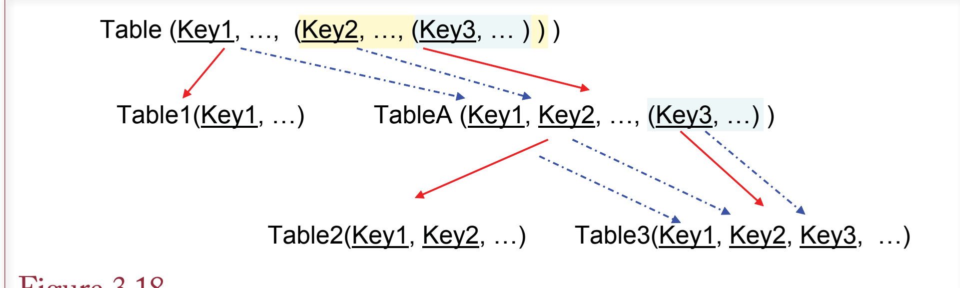


Figure 3.18

Nested repeating groups. Groups are nested when they repeat within another group (key3 inside key2 inside key1). Split them in steps: Pull all of group2 from group1, then pull group3 from group2. Note that every table will contain the original key (key1). With three levels, the final table (Table3) must contain three columns in the key.

customer. The Sale would be the top level, departments would be next, and individual sale items would be nested within the departments. The table resulting from the innermost nesting would have a primary key consisting of the SaleID, Department, and ItemID columns.

Second Normal Form

What is second normal form? It was straightforward to reach first normal form: Just identify the repeating groups and put them into their own table that is linked to the main table through the initial key. The next step is a little more complicated because you have to look at relationships between the key value and the other (nonkey) columns in the table. Correct specification of the keys is crucial. At this point it would be wise to double-check all the keys to make sure they are unique and that they correctly identify many-to-many relationships. In particular, focus on tables where the primary key consists of more than one column. Second normal form is concerned with the situation where a nonkey column depends on only part of the key.

Problems with First Normal Form

It is fairly clear that 1NF is not the final answer. You still face a couple of major design issues. The DB Design system highlights some of the issues with the keys in the SaleLine group. A generated key must always be the only key column in a table, but the SaleLine table needs both SaleID and ItemID as keys. One solution might be to just remove the generator from ItemID so it is a simple key column. But, if you make that change, where would you get values for the ItemID? The problem is that the SaleLine table is trying to do two things: show which items were sold and describe individual items.

You can guess by the temporary names of the tables in Figure 3.15 that first normal form might still have problems storing data efficiently. Consider the situation in Figure 3.19 that illustrates the current Sale Item table. Every time someone buys item 15, the database stores the *Air Tank* description. The problem is that the description depends on only part of the key (ItemID). If you know the ItemID, you always know the corresponding description. The description does not change with every transaction. Beyond the waste of space and clerical time, there is an

SaleLine(<u>SaleID</u>, <u>ItemID</u>, Description, ListPrice, Quantity, QuantityOnHand)

Duplication for columns that depend only on ItemID

<u>SaleID</u>	<u>ItemID</u>	Description	ListPrice	Quantity	QOH
11851	15	Air Tank	192.00	2	15
11851	27	Regulator	251.00	1	5
11851	32	Mask 1557	65.00	1	6
11852	15	Air Tank	192.00	4	15
11852	33	Mask 2020	91.00	1	3
11853	41	Snorkel 71	44.00	2	15
11853	75	West suit-S	215.00	1	3
11854	75	Wet suit-S	215.00	2	3
11854	32	Mask 1557	65.00	1	6
11854	57	Snorkel 95	83.00	1	17

Figure 3.19

Problems with first normal form. This design is in 1NF but it still contains duplicated data. Every time an item is sold, the clerk has to reenter its description, list price, and quantity on hand. Also, if an item has not yet been sold, what is its ListPrice? The problems arise because these columns depend only on the ItemID, not on the SaleID.

additional problem: If an item has not yet been sold, what is its description (and price)? Because items are only entered into the database with a transaction, this data will not be stored in the database. Similarly, if all the rows for item 15 are deleted, you will lose all the associated information about that product.

Second Normal Form Definition

The problem with the preceding example is that once you know the ItemID, you always know the description. A one-to-one relationship exists between the ItemID and the Description (perhaps many-to-one). As shown in Figure 3.20, the important point is that the sale transaction does not matter. If someone buys item 15 in June, the description is *Air Tank*. If someone buys item 15 in December, the description is still *Air Tank*. Hence, the description depends on only part of the key (the ItemID and not the SaleID). A table is in **second normal form (2NF)** if every nonkey column depends on the entire key (not just part of it). Note that this issue arises only for composite keys (with multiple columns).

The solution is to split the table. Pull out the columns that depend on part of the key. Remember to include that part of the key in the new table. The new tables (SaleItems and Item) are shown in Figure 3.21. Note that ItemID must be in both tables. It stays in the SaleItems table to indicate which items have been purchased at each time. It is the primary key in the Item table because it is the unique identifier. Including the column in both tables enables you to link the data together later.

In creating the new Item table, you are faced with the interesting question of where to put the price. There are two choices: in the SaleItems table or in the Item table. The answer depends on the operations and rules used in the business. From a technical standpoint you can choose either table. However, from a business standpoint there is a big difference. Consider the case where the price is in

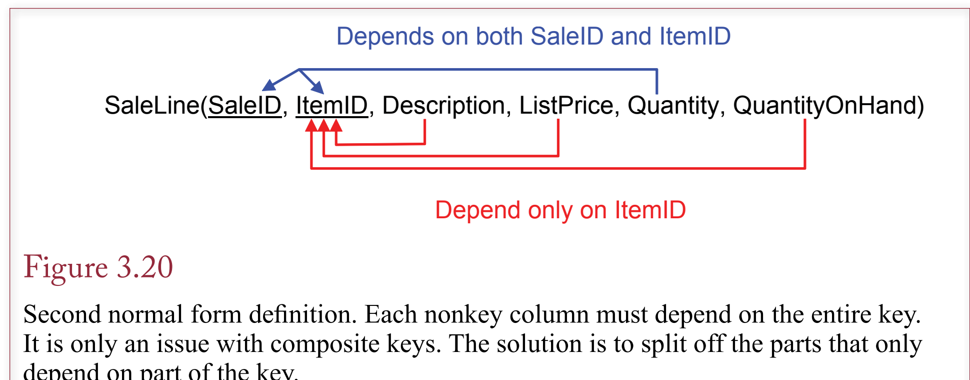


Figure 3.20

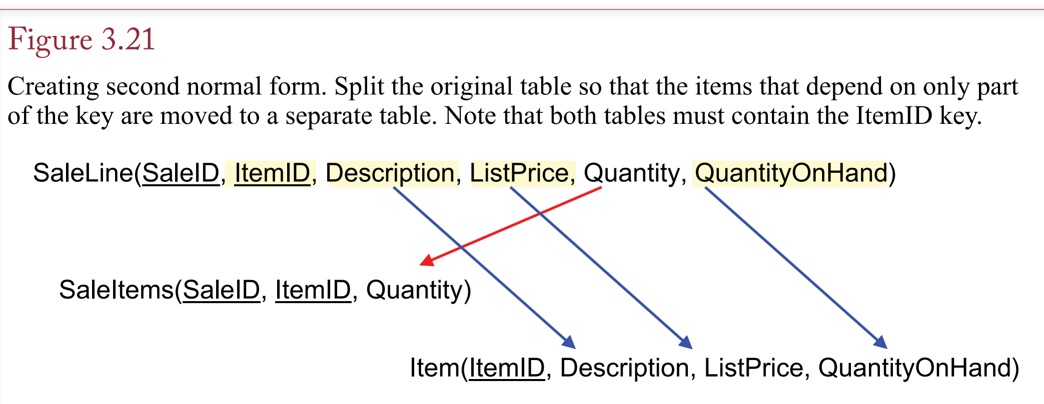
Second normal form definition. Each nonkey column must depend on the entire key. It is only an issue with composite keys. The solution is to split off the parts that only depend on part of the key.

the Item table. This model of the firm says that if you know the ItemID, you always know the price. In other words, the price is fixed for each item and does not change over time. Now consider the interpretation when the price is stored in the Item table. Here you are explicitly saying that the price depends on both the ItemID and on the specific sale. In other words, for one customer the price for *Air Tank* might be $192, whereas another customer might pay only $175. The price difference might arise because you give end-of-season discounts, or if someone purchases several items at one time. Most business database designers quickly encounter the problem of where to store prices. One solution is to store prices in both tables. That is, the price in the Items table (ListPrice) would be the list price that the business intends to charge. The price in the SaleItems table would be the actual price paid that incorporates various discounts (SalePrice). The key point is that the final list of tables depends not just on mechanical rules but is also determined by the operations of the business. The assumptions you make about how a particular business operates determine the tables you get. For now, you will stick with the simpler assumption that assigns a fixed ListPrice to each item.

Figure 3.22 gives sample data for the new tables. Notice that 2NF resolves the problem of repeating the description each time an item is sold. The base product data is stored one time in the Item table. It is referenced in the SaleItem table by the ItemID. Looking through the SaleItem table, you can easily get the corresponding description by finding the matching ID in the Item table. Chapter 4 explains how the database query system handles this link automatically.

Figure 3.21

Creating second normal form. Split the original table so that the items that depend on only part of the key are moved to a separate table. Note that both tables must contain the ItemID key.



SaleID	ItemID	Quantity
11851	15	2
11851	27	1
11851	32	1
11852	15	4
11852	33	1
11853	41	2
11853	75	1
11854	75	2
11854	32	1
11854	57	1

SaleItems(SaleID, ItemID, Quantity)

ItemID	Description	ListPrice	QOH
15	Air Tank	192.00	15
27	Regulator	251.00	5
32	Mask 1557	65.00	6
33	Mask 2020	91.00	3
41	Snorkel 71	44.00	15
57	Snorkel 95	83.00	17
75	Wet suit-S	215.00	3
77	Wet suit-M	215.00	7

Item(ItemID, Description, ListPrice, QuantityOnHand)

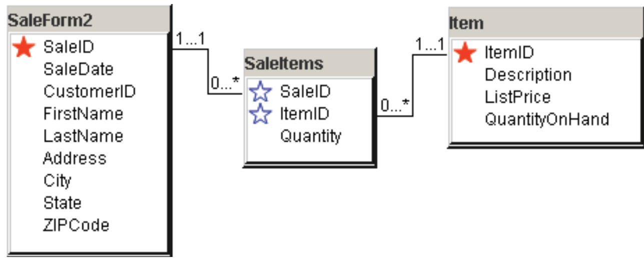
Figure 3.22

Second normal form data. Product items are now stored only one time. Other tables (SaleItems) can refer to an item just by its key (ItemID), which provides a link back to the Item table.

Figure 3.23 shows the current status of the tables in DB Design. The problem with the ItemID key has been solved. The ItemID key values are generated in the Item table whenever a new product is added to inventory. Every column in the Item table depends only on the ItemID key. The ItemID values are used along with the SaleID values in the SaleItems table to show exactly which items are purchased on each sale. Almost any time you have a many-to-many relationship, you will see a similar pattern. One table will be used to generate each key column, and the intermediate or junction table will use the two keys to handle the many-to-many relationship. The challenge is to identify exactly which columns depend on both key columns and belong in the intermediate table.

Figure 3.23

Second normal form in DB Design. ItemID is generated in a table that only refers to items. This value is used along with SaleID in a table that shows which items were purchased on each sale.



SaleForm2(SaleID, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode)

SaleID	Date	CustomerID	FirstName	LastName	Address	City	State	ZIP
11851	7/15	15023	Mary	Jones	111 Elm	Chicago	IL	60601
11852	7/15	63478	Miguel	Sanchez	222 Oro	Madrid		
11853	7/16	15023	Mary	Jones	111 Elm	Chicago	IL	60601
11854	7/17	94552	Madeline	O'Reilly	333 Tam	Dublin		

Duplication

Figure 3.24

Problems with second normal form. The hidden dependency in the customer data leads to duplicating the customer address each time a customer rents videos from the store. Similarly, if old transaction rows are deleted, the firm might lose all of the data for some customers.

Third Normal Form

What is third normal form? The logic, analysis, and elements of designing for **third normal form (3NF)** are similar to those used in deriving 2NF. In particular, you still concentrate on the issue of dependence. With experience, most designers combine the derivation of 2NF and 3NF into a single step. Technically, a table in 3NF must also be in 2NF.

Problems with Second Normal Form

At this point, you need to examine the SaleForm2 table that was ignored in the earlier analysis. It is displayed in Figure 3.24. In particular, notice that SaleID is the key. The problem can be seen in the sample data. Every time a customer participates in a sale, the database stores his or her name, address, and phone number again. This unnecessary duplication is a waste of space and probably a waste of the clerk's data entry time. Consider what happens when a customer moves. You would have to find the address and change it for every transaction the customer had with the store. Likewise, if the customer has not yet purchased any items, you do not have a place to store the customer data. Similarly, if you delete old transactions from the database, you risk losing customer data. The problem arises because you have a hidden dependency. The solution is to make the dependency explicit.

Third Normal Form Definition

The problems in the previous section are fairly clear. The customer name, address, phone, and so on depend on the CustomerID. Given a specific value for CustomerID, you immediately know the rest of the customer data. The problem with the design at this point is that CustomerID is not part of the key for the table. In other words, some nonkey columns do not depend on the key. So why are they in this table? The question also provides the solution. If columns do not depend on the primary key, they should be placed in a separate table.

To be in 3NF a table must already be in 2NF, and every nonkey column must depend on nothing but the key. In the video example in Figure 3.25, the problem

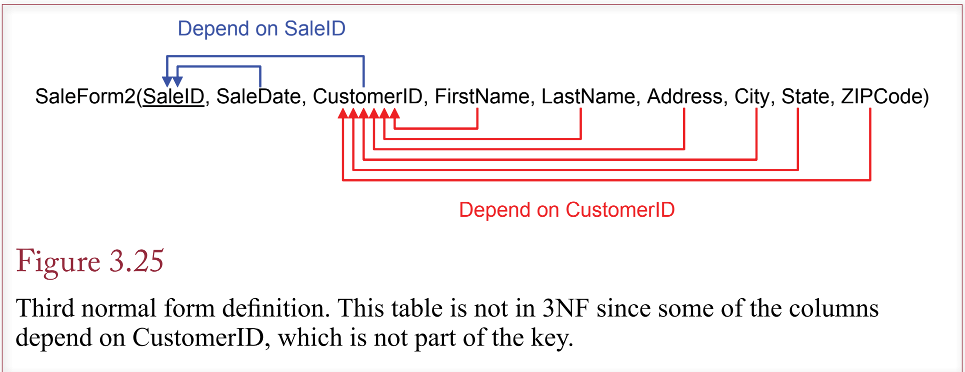


Figure 3.25

Third normal form definition. This table is not in 3NF since some of the columns depend on CustomerID, which is not part of the key.

is that basic customer data columns depend on the CustomerID, which is not part of the key.

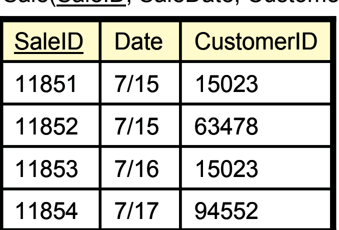
At first glance, two solutions seem possible: (1) make CustomerID part of the key or (2) split the table. If the table is already in 2NF, option (2) is the only choice that will work. The problem with the first option is that making CustomerID part of the key is equivalent to stating that each transaction can involve many customers. This assumption is not likely to be true. However, even if it is, your table would no longer be in 2NF, since the customer data would then depend on

Figure 3.26

Third normal form. Putting customer data into a separate table eliminates the hidden dependency and resolves the problems with duplicate data. Note that CustomerID remains in both tables, but it is still not a key in the Sale table because only one customer participates in a given sale.

SaleForm2(SaleID, SaleDate, CustomerID, FirstName, LastName, Address, City, State, ZIPCode)

Sale(SaleID, SaleDate, CustomerID)



SaleID	Date	CustomerID
11851	7/15	15023
11852	7/15	63478
11853	7/16	15023
11854	7/17	94552

Customer(CustomerID, FirstName, LastName, Address, City, State, ZIPCode)

CustomerID	FirstName	LastName	Address	City	State	ZIP
15023	Mary	Jones	111 Elm	Chicago	IL	60601
63478	Miguel	Sanchez	222 Oro	Madrid		
94552	Madeline	O'Reilly	333 Tam	Dublin		

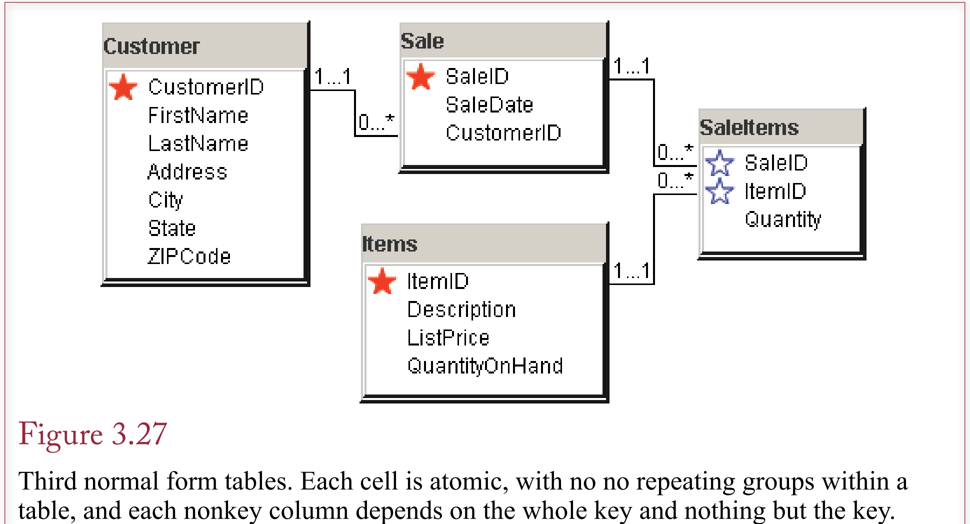


Figure 3.27

Third normal form tables. Each cell is atomic, with no no repeating groups within a table, and each nonkey column depends on the whole key and nothing but the key.

only part of the key (CustomerID and not TransID). Hence the correct solution is to split the table into two parts: the columns that depend on the whole key and the columns that depend on something else (CustomerID).

The solution in the sale example is to pull out the columns that are determined by the CustomerID. Remember to include the CustomerID column in both tables so they can be relinked later. The resulting tables are displayed in Figure 3.26. Notice that CustomerID is not a key in the Sale table because only one customer participates in any given sale. Figure 3.26 also illustrates how splitting the tables resolves the problems from the hidden dependency.

The final collection of tables is presented in Figure 3.27. This list is in 3NF: Each cell is atomic and there are no repeating groups within a table (1NF), and each nonkey column depends on the whole key (2NF) and nothing but the key (3NF). You can change the layout of the tables, but the relationships remain the same. As shown in Figure 3.28, you can also write the list of tables and their column names. This approach makes it easy to fit dozens of tables on one page; however, the relationships are more difficult to see.

The astute reader should raise a question about the address data. That is, City, State, and ZipCode have some type of dependent relationship. Perhaps the Customer table is not really in 3NF? In theory, it is true: ZIP codes were created as a means to identify locations. The catch is that at a five-digit level, the relationship is relatively weak. A ZIP code identifies an individual post office. Each city can have many ZIP codes, and a ZIP code can be used for more than one city. At the moment, it is true that a ZIP code always identifies one state. However, can you be certain that this relationship will always hold—even in an international setting? Hence it is generally acceptable to include all three items in the same table. On the other hand, as pointed out in the pet store discussion in Chapter 2, there are some advantages to creating a separate City table. The most important advantage is that you can reduce data entry time and errors by selecting a city from a predefined list.

Customer(<u>CustomerID</u>, FirstName, LastName, Address, City, State, ZIPCode)
Sale(<u>SaleID</u>, SaleDate, CustomerID)
SaleItems(<u>SaleID</u>, <u>ItemID</u>, Quantity)
Item(<u>ItemID</u>, Description, ListPrice, QuantityOnHand)

Figure 3.28

Third normal form table list. The list is an easy way to fit dozens of tables on a page but does not show the relationships.

Checking Your Work

At this critical point, you must double-check your work. In large projects it is beneficial to have several team members participate in the review to make sure the assumptions used in defining the data tables match the business operations.

The essence of data normalization is to collect all the forms and reports and then to inspect each form to identify the data that will be stored. Writing the columns in a standard notation makes the normalization process more mechanical, minimizing the potential for mistakes. In particular, look for keys and highlight one-to-one and one-to-many relationships. To check your work, you need to examine each table to make sure it demonstrates the assumptions and operations of the firm.

To check your tables, you essentially repeat the steps in normalization. First, make sure that you have pulled out every repeating group. While you are at it, double-check your keys. Be sure you know exactly where each key value is generated. To verify the key columns, start with the first key column in a table and ask yourself if there is a one-to-one or a one-to-many relationship with each of the other columns. If it is a one-to-many relationship (or many-to-many), you need to underline the column title. If it is one-to-one (or many-to-one), the column in question should not be underlined. The second step is to look at each nonkey column and ask yourself if it depends on the whole key and nothing but the key. Third, verify that the tables can be reconnected. Try drawing lines between each table. Tables that do not connect with the others are probably wrong. Fourth, ask yourself if each table represents a single object. Try giving it a name. If you cannot find a good single name for the table, it probably represents more than one object and needs to be split. Finally, enter sample data for each table and make sure that you are not entering duplicate rows. Some underlying problems may become obvious when you begin to enter data. It is best to enter test data during the design stage, instead of waiting until the final implementation.

Beyond Third Normal Form

What problems exist beyond third normal form? In designing relational database theory, E. F. Codd first proposed the three normalization rules. On examining real-world situations, he and other writers realized that additional problems could occur in some situations. In particular, Codd's initial formal definition of 3NF was probably too narrow. Hence he and Boyce defined a new version, which is called **Boyce-Codd normal form (BCNF)**.

Other writers eventually identified additional problems that could arise and created further "normal forms." If you are careful in designing your database—particularly in creating keys—you should not have too many problems with these

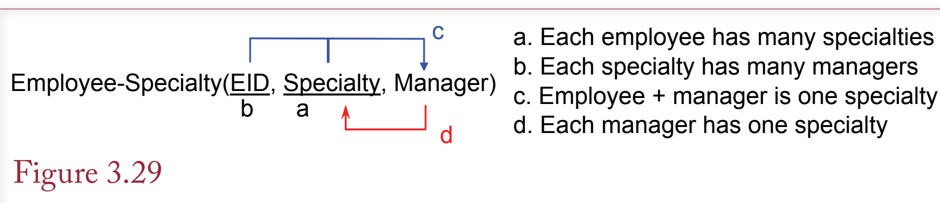


Figure 3.29

Boyce-Codd normal form. There is a hidden dependency (d) between manager and specialty. If we delete rows from the original table, we risk losing data about our managers. The solution is to add a table to make the dependency explicit.

issues. However, occasionally problems arise, so a good database designer will check for the problems described in the following sections. In particular, in large projects with many designers, one member of the team should check the final list of tables.

Boyce-Codd Normal Form

You have already seen how problems can arise when hidden dependencies occur within a table. A secondary relationship between columns within a table can cause problems with duplication and lost data. Consider the example in Figure 3.29, which contains data about employees. From the business rules, it is clear that the table is in 3NF. The keys are correct, and from rule (c) the nonkey column (Manager) depends on the entire key. That is, each employee can have a different manager for each specialty. The problem arises because of business rule (d): Each manager has only one specialty. The manager determines the specialty, but since Manager can never be a key for the entire table, you have a hidden dependency (Manager → Specialty) in the table. A **hidden dependency** arises when there is a functional rule that is not part of the primary key. What if you delete old data rows and delete all references to one manager? Then you lose the data that revealed that manager's specialty. BCNF prevents this problem by stating that any dependency must be explicitly shown in the keys.

The solution is to add a table to make the dependency explicit. Because each specialty can have many managers, the best solution is to add the table Manager(Manager, Specialty). Note that technically, you can now remove the Specialty column from the original table (and key Manager). Because a manager can have only one specialty, as soon as you know the manager, you can use a link to obtain the specialty. However, as a designer, you have to question to rules. This situation requires some unusual rules. If the manager-specialty rule is relaxed in the future, allowing managers to have multiple specialties, you would have to redesign the tables (and forms and reports). In the example it is not very realistic to believe the firm will always have managers with only one specialty. It is better to leave the original table and add the new Manager table. Then if the assumptions change, you simply need to make Specialty a key in the Manager table. The main point is that you have solved the BCNF problem by explicitly recording the hidden relationship—so you no longer need to worry about losing important relationships when you delete rows.

EmployeeTasks(EID, Specialty, ToolID)

Business rules.

(a) Each employee has many specialties.

(b) Each employee has many tools.

(c) Tools and specialties are unrelated.

EmployeeSpecialty(EID, Specialty)
EmployeeTools(EID, ToolID)

Figure 3.30

Fourth normal form. The original table is 3NF because there are no nonkey columns. The keys are legitimate, but there is a hidden (multivalued) dependency because Specialty and ToolID are unrelated. The solution is to create two tables—one to show each of the two dependencies.

Fourth Normal Form

Fourth normal form (4NF) problems arise when there are two binary relationships, but the modeler attempts to show them as one combined relationship. An example should clarify the situation.

In Figure 3.30, employees can have many specialties, and they perform many tasks for each specialty. Because all three columns are keyed, the table must be in 3NF. From the business rules, you can see that the keys are legitimate. However, there are really two binary relationships instead of one ternary relationship: Employee → Specialty and Employee → Tool.

Since the third business rule specifies that Specialty and ToolID are not directly dependent on each other, you need to break up the original table into two tables to remove the hidden dependency. The problem you would face with the original is that there could be considerable duplication of data if for every employee you have to list each tool for every specialty. It is more efficient to list specialties and tools separately.

Fourth normal form problems can occur and they can cause problems, so you should be able to spot them. The main trick is to watch for hidden dependencies, and make sure they are made explicit.

Domain-Key Normal Form

In 1981 Fagin described a different approach to normalized tables when he proposed the **domain-key normal form (DKNF)**. DKNF describes the ultimate goal in designing a database. If a table is in DKNF, Fagin proved that it must also be in 4NF, 3NF, and all of the other normal forms. The catch is that there is no defined method to get a table into DKNF. In fact, it is possible that some tables can never be converted to DKNF.

Despite these difficulties, DKNF is important for application developers because it is a goal to work toward when designing applications. Think of it as driving to the mall when you do not have exact directions. You can still get there as long as you know how to start (1NF, 2NF, and 3NF are well-defined) and can recognize the mall when you arrive (DKNF).

The goal of DKNF is to have each table represent one topic and for all the business rules to be expressed in terms of domain constraints and key relationships.

EmployeeTask(<u>EmployeeID</u>, <u>TaskID</u>, <u>ToolID</u>)
Defined business rules (a) Each employee performs many tasks with many tools.
But, maybe you need a second rule. (b) Each task has commonly used tools. RequiredTools(<u>TaskID</u>, <u>ToolID</u>)

Figure 3.31

DKNF example. With the stated rule, the tables are in BCNF but might not be in DKNF. The initial table combines information about tasks and tools. Maybe there is an additional undefined dependency between task and tool, where employees commonly use the same tools for each task.

That is, all business rules are explicitly described by the table rules. Domain constraints are straightforward—they represent limitations placed on the data held in a column. For example, prices cannot be negative.

All other business rules must be expressed in terms of relationships with keys. In particular, there can be no hidden relationships. Consider the example in Figure 3.31, which shows a table that records the tasks performed by employees and the tools they used. The primary business rule you were given states that each employee performs many tasks with many tools, so all three columns need to be part of the primary key. The key columns are legitimate and the table is in 3NF. Since only one rule (dependency) has been specified, the table is also in BCNF. However, if you think about the business problem for a few minutes, you can see that the table might be used to cover two topics: (1) The tools that employees actually used, and (2) The tools that are commonly used for a specific task. Think about the problem from the perspective of a novice employee who needs to know which tools to pick up for a specific task. You could query the database to see what tools other employees used in the past, but there could be considerable variation. Perhaps there needs to be a second dependency that lists the minimum set of tools required for each task. If you know about this rule at the start, you can see that the single EmployeeTask table violates BCNF because it ignores a hidden rule. But, since the rule was not explicitly stated, you used the DKNF approach to realize the initial table was trying to cover two different facts. With this insight, you can look harder to identify formal rules.

This example also shows you the challenge of DKNF. There is no formal method to arrive at DKNF. To define a set of tables in DKNF, you can start by working through the 3NF rules. Then look carefully for hidden dependencies and add tables to reach BCNF. Then, verify keys and ensure that each table describes a single fact, and that facts are stored in only one location. Domain-key normal form returns to the beginning of Chapter 2. The goal in designing the database is to build a model of the organization, and DKNF clarifies this goal by stating that the best database design is one that explicitly states all business rules as database rules.

In theory, there can be no normal forms beyond DKNF. That is a nice theory, but since there is no well-defined way to put a set of tables in DKNF, it is not always helpful. Several authors have identified other potential problems and derived additional versions of normal forms, such as fifth normal form. For the most part these definitions are not very useful in practice; they will not be described here. You can consult C. J. Date's textbooks for details and examples of more theoretical concepts.

Summary

You can review the technical definitions in the appendix for a formal statement of the normalization conditions. However, the most important thing to remember is that normalization ultimately comes down to properly understanding the business rules (dependencies). The first rule is straightforward: Each cell contains atomic, non-repeating data. The second and third rule can be summarized by remembering that: Each nonkey column depends on the whole key and nothing but the key. (So help me Codd.) BCNF seems slightly trickier, but a simple rule can be used to represent the entire process: There must be no hidden dependencies. All dependencies should be explicitly stated within the primary keys.

Data Rules and Integrity

How does a database record constraints? As you talk to users and managers to design reports and tables, you also need to think about what business rules need to be enforced. One of the goals of a database designer is to ensure that the data remains accurate. Many cases have straightforward business rules. For example, you typically want to make sure that price is greater than zero. Similarly, you may have a constraint that salaries should not exceed some number like $100,000 or that the date hired has to be greater than the date the company was founded. These **data integrity** constraints are easy to assign in most databases. Typically, you can go to the table definitions and add the simple constraints along with a message. The advantage of storing these constraints with the tables is that the DBMS enforces the conditions for every operation on the table, regardless of the source or method of data entry. No programming is necessary, and the constraint is stored in one location. If you need to change the condition, it is readily accessible (to authorized users).

A second type of constraint is to choose data from a set of predefined options. For example, gender may be listed as male, female, or unavailable. Providing a list helps clerks enter data, and it forces them to enter only the choices provided. For instance, you do not have to worry whether someone might enter *f*, *F*, or *fem*. The data is more consistent.

A third type of data integrity is a bit more complicated but crucial in a relational database. The tables are nicely organized with properties that ensure efficient storage of the data. Yet you need to be able to reconnect the data in the tables to get the reports and forms the users need. Consider the sale example in Figure 3.32 when a clerk enters a customer number in the Sale table. What happens if the clerk enters a customer number that does not exist in the Customer table? If you want to check later on customer purchases, you will be unable to find matching data for that customer. Hence you need a constraint to ensure that when a customer number is entered into the Sale table that number must already exist in the Customer table. The CustomerID in the Sale table is a foreign key in that table, and the constraint you need is known as referential integrity. **Referential integrity** exists

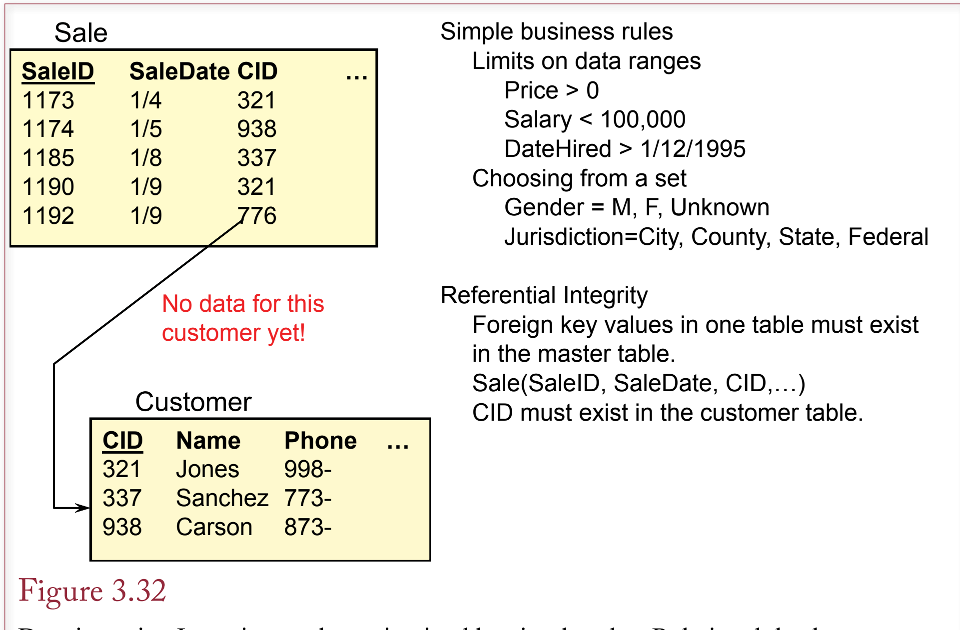


Figure 3.32

Data integrity. Integrity can be maintained by simple rules. Relational databases rely on referential integrity constraints to ensure that customer data exists before the customer number can be entered in the Rental table.

when a value for a foreign key can be entered only if the corresponding value already exists in the originating table.

Essentially, once you define the relationship between tables, you can tell the DBMS to enforce referential integrity. The method for defining referential integrity depends on the specific DBMS. Generally, the constraint is specified in the CREATE TABLE command. Most relational databases also support **cascading delete**, which uses the same concepts. If a user deletes a row in the Customer table, you also need to delete the related entries in the Sale table. Then you need to delete the corresponding rows in the SaleItems table. If you build the relationships and specify cascade on delete, the database will automatically delete the related

Figure 3.33

SQL referential integrity definition. In the Sale table, declaring a column as a foreign key tells the DBMS to check each value in this table to find a matching value in the referenced (e.g., Customer) table.

```
CREATE TABLE Sale
(  SaleID        Integer NOT NULL,
   SaleDate      Date,
   CustomerID    Integer,
     CONSTRAINT pk_Sale PRIMARY KEY (SaleID),
     CONSTRAINT fk_SaleCustomer FOREIGN KEY (CustomerID)
       REFERENCES Customer (CustomerID)
       ON DELETE CASCADE
)
```

<table>
<tr><td colspan="5">Location
Date Played</td><td colspan="5">Referee Name
Phone Number, Address</td></tr>
<tr><td colspan="2">Team 1
Name
Sponsor</td><td colspan="3">Score</td><td colspan="2">Team 2
Name
Sponsor</td><td colspan="3">Score</td></tr>
<tr><td>Player Name</td><td>Phone</td><td>Age</td><td>Points</td><td>Penal.</td><td>Player Name</td><td>Phone</td><td>Age</td><td>Points</td><td>Penal.</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Figure 3.34

Database design for a soccer league. The design and normalized tables depend on the business rules. Some of the rules are shown on the form.

rows when a user deletes an entry in the Customer table. These actions maintain the consistency of the database by ensuring that links between the tables always refer to legitimate rows.

Oracle and SQL Server support referential integrity by declaring a foreign key when you create a table. Figure 3.33 shows the command that can be used to create a Sale table with three columns. The company wants to make sure that all orders are sent to legitimate customers, so the customer number (CustomerID) in the Sale table must exist in the Customer table. The foreign key constraint enforces this relationship. The constraint also specifies that the relationship should handle cascading deletes. Oracle and SQL Server use the standard SQL language to create tables.

When you start to enter data into a DBMS, you will quickly see the role played by referential integrity. Consider two tables: Sale(SaleID, SaleDate, CustomerID) and Customer(CustomerID, Name, Address, etc.). You have a referential integrity constraint that links the CustomerID column in the Sale table to the CustomerID column in the Customer table. As you enter sample data, begin with the Sale table. The DBMS will not accept any data—because the corresponding CustomerID must already exist in the Customer table. That is, the referential integrity rules force you to enter data in a certain order. Clearly, these rules would present problems to users, so you cannot expect users to enter data directly into tables. Chapters 6, 7, and 8 explain how forms and applications will automatically ensure that the user enters data in the proper sequence.

The Effects of Business Rules

How do business rules change the database design? It is important to understand how different business rules affect the database design and the normalization process. As a database designer, you must identify the basic rules and build the database to match them. However, be careful because business rules can change. If you think a current business rule is too restrictive, you should design the database with a more flexible structure.

There is one referee per match.
A player can play on only one team.

Match(<u>MatchID</u>, DatePlayed, Location, RefID)
Score(<u>MatchID</u>, <u>TeamID</u>, Score)
Referee(<u>RefID</u>, Phone, Address)
Team(<u>TeamID</u>, Name, Sponsor)
Player(<u>PlayerID</u>, Name, Phone, DoB, TeamID)
PlayerStats(<u>MatchID</u>, <u>PlayerID</u>, Points, Penalties)

Figure 3.35

Restrictive rules. With only one referee per match, the referee key is added to the Match table. Similarly, the TeamID column is placed in the Player table.

Consider the example shown in Figure 3.34. The local parks and recreation department runs a soccer league and collects basic statistics at the end of every match. You need to design the data tables for this problem.

To illustrate the effect of different rules, consider the two main rules and the resulting tables displayed in Figure 3.35. The first rule states that there can be only one referee per match. Hence the RefID can be placed in the Match table. Note that it is not part of the primary key. The second rule states that a player can play on only one team; therefore, the appropriate TeamID can be placed in the Player table.

Now consider what happens if these two rules are relaxed as shown in Figure 3.36. The department manager believes that some day there might be several referees per match. Also, the issue of substitute players presents a problem. A substitute might play on several different teams in a season—but only for one team during a match. To handle these new rules, the key values must change. You might be tempted to make the simple changes indicated in Figure 3.36; that is, make RefID part of the key in the Match table and make TeamID part of the primary key in the Player table. Now each Match can have many Referees, and each Player can play on many teams. The problem with this approach is that the Match and Player

Figure 3.36

Relaxing the rules to allow many-to-many relationships. You might try to make the RefID and TeamID columns part of the primary key, but the resulting tables are not in 3NF. Location does not depend on RefID, and Player Name does not depend on TeamID.

There can be several referees per match.
A player can play on several teams (substitute), but only one team per match.

Match(<u>MatchID</u>, DatePlayed, Location, <u>RefID</u>)
Score(<u>MatchID</u>, <u>TeamID</u>, Score)
Referee(<u>RefID</u>, Phone, Address)
Team(<u>TeamID</u>, Name, Sponsor)
Player(<u>PlayerID</u>, Name, Phone, DoB, <u>TeamID</u>)
PlayerStats(<u>MatchID</u>, <u>PlayerID</u>, Points, Penalties)

There can be several referees per match.
A player can play on several teams (substitute), but only one team per match.

Match(MatchID, DatePlayed, Location)
RefereeMatch(MatchID, RefID)
Score(MatchID, TeamID, Score)
Referee(RefID, Phone, Address)
Team(TeamID, Name, Sponsor)
Player(PlayerID, Name, Phone, DoB)
PlayerStats(MatchID, PlayerID, TeamID, Points, Penalties)

Figure 3.37

Relaxing the rules and normalizing the tables. The RefereeMatch table enables the department to have more than one referee per match. Moving the TeamID to the PlayerStats table indicates that someone can play for more than one team—but for only one team during a given match.

tables are no longer in 3NF. For example, DatePlayed does not depend on RefID. Likewise, Name in the Player table does not depend on the TeamID. For example, Paul Ruiz does not change his name every time he plays on a different team.

The solution is displayed in Figure 3.37. A new table is added to handle the many-to-many relationship between referees and matches. Similarly, the player's TeamID is moved to the PlayerStats table, but it is not part of the primary key. In this solution, each match has many players, and players can participate in many matches. Yet, for each match, each player plays for only one team. This new database design is different from the initial design. More importantly it is less restrictive. As a designer, you must look ahead and build the database so that it can handle future needs of the department.

Which of these database designs is correct? The answer depends on the needs of the department. In practice, it would be wiser to choose the more flexible design that can assign several referees to a match and allows players to substitute for different teams throughout the season. However, in practice you should make one minor change to this database design. If no matches have been played, how do you know which players are on each team? As it stands, the database cannot answer this question. The solution is to add a BaseTeamID to the Player table. At the start of the season, each team will submit a roster that lists the initial team members. Players can be listed on only one initial team roster. If someone substitutes or changes teams, the data can be recorded in the PlayerStats table.

Converting a Class Diagram to Normalized Tables

What problems arise when converting a class diagram to normalized tables? Each normalized table represents a business entity or class. Hence a class diagram can be converted into a list of normalized tables. Likewise, a list of normalized tables can be drawn as a class diagram. Technically, the entities in a class diagram do not have to be in 3NF (or higher). Some designers use a class diagram as an overview, or big picture, of the business, and they leave out some of the normalized details. In this situation you will have to convert the classes into a list of normalized tables. As noted in Chapter 2, some features commonly arise on a class diagram, so you should learn how to handle these basic conversions.

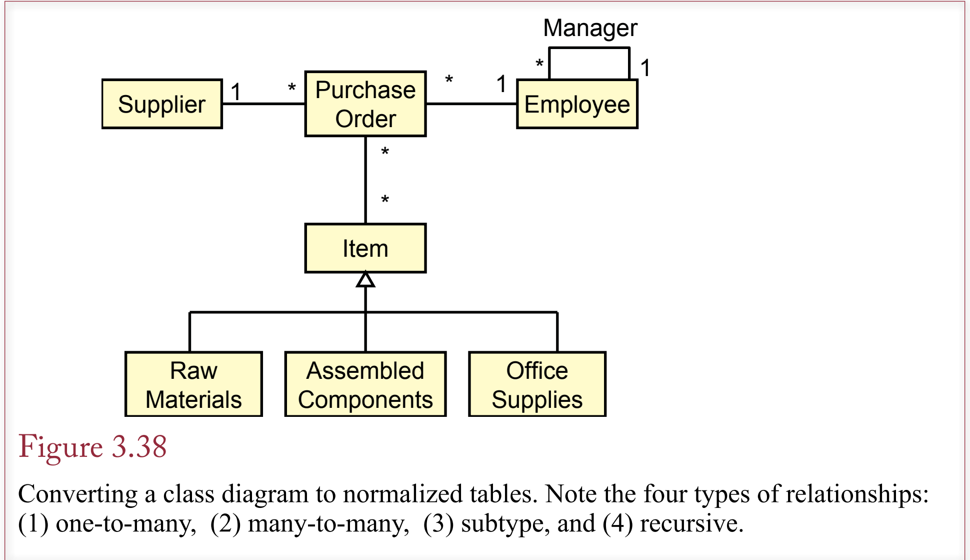


Figure 3.38

Converting a class diagram to normalized tables. Note the four types of relationships: (1) one-to-many, (2) many-to-many, (3) subtype, and (4) recursive.

The most challenging problems you will encounter are from class diagrams that utilize object-oriented features, such as subclasses and composition. Some relational database systems have added object-oriented features to make it easier to handle these issues. For example, you can store object data in a cell. However, storing object data (including XML) in a cell often violates the first rule of normalization. From the database perspective, the data is no longer atomic, but requires special routines to examine and compare the data within a cell. Sometimes it makes sense to use these extensions, but you will have to weigh the tradeoffs. For example, most systems enable you to define customized data types. A common use is for spatial data where a location is stored as a single GPS (latitude, longitude, and altitude) coordinate instead of three separate columns. For the approach to be successful, you (or the DBMS vendor) need to write customized functions to use this new object type. Some objects, such as location, are commonly used by many organizations, so it does make sense. Other, highly customized objects, could require considerable additional effort, and you need to evaluate the tradeoffs before creating the custom objects.

Figure 3.38 illustrates a typical class diagram for a purchase order with four basic types of relationships: (1) a one-to-many relationship between supplier and the purchase order, (2) a many-to-many relationship between the purchase order and the items, (3) a subtype relationship that contains different attributes, and (4) a recursive relationship within the Employee entity to indicate that some employees are managers of others.

One-to-Many Relationships

The most important rule in converting class diagrams to normalized tables is that relationships are handled by placing a common column in each of the related tables. This column is usually a key column in one of the tables. This process is easy to see with one-to-many relationships.

The purchase order example has two one-to-many relationships. (1) Many different purchase orders can be sent to each supplier, but only one supplier appears on a purchase order. (2) Each purchase order is created by only one employee, but

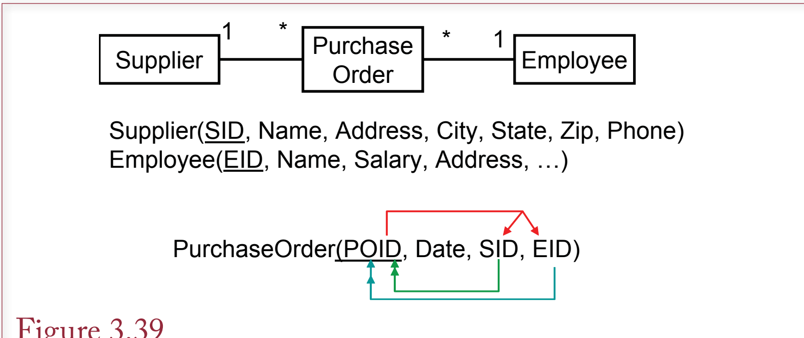


Figure 3.39

Converting one-to-many relationships. Add the primary key from the one-side into the many-side table. In the example SID and EID are added to the PurchaseOrder table. Note that they are not primary keys in the PurchaseOrder table.

Figure 3.40

Sample data for one-to-many relationships. The Supplier and PurchaseOrder tables are linked through the SID column. Similarly, the Employee table is linked through the data in the EID column. Both the SID and EID columns are foreign keys in the PurchaseOrder table, but they are not primary keys in that table.

Supplier

ID	Name	Address	City	State	Zip	Phone
5676	Jones	123 Elm	Ames	IA	50010	515-777-8988
6731	Markle	938 Oak	Boston	MA	02109	617-222-9999
7831	Paniche	873 Hickory	Jackson	MS	39205	601-333-9932
8872	Swensen	773 Poplar	Wichita	KS	67209	316-999-3312

Purchase Order

POID	Date	SID	EID
22234	9/9	5676	221
22235	9/10	5676	554
22236	9/10	7831	221
22237	9/11	8872	335

Employee

EID	Name	Salary	Address
221	Smith	67,000	223 W. 2300
335	Sanchez	82,000	37 W. 7200
554	Johnson	35,000	440 E. 5200

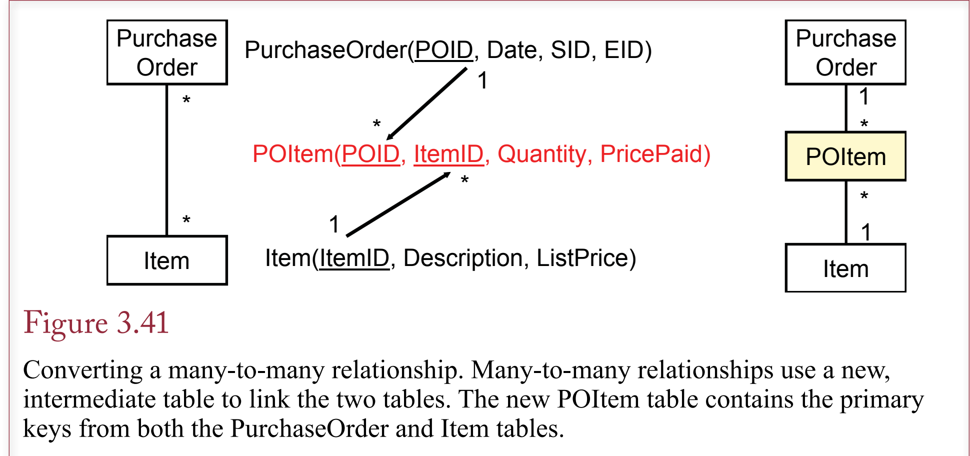


Figure 3.41

Converting a many-to-many relationship. Many-to-many relationships use a new, intermediate table to link the two tables. The new POItem table contains the primary keys from both the PurchaseOrder and Item tables.

an employee can create many purchase orders. To create the normalized tables, first create a primary key for each entity (Supplier, Employee, and PurchaseOrder). As shown in Figure 3.39, the normalized tables can be linked by placing the Supplier key (SID) and Employee key (EID) into the PurchaseOrder table. Note carefully that all class diagram associations are expressed as relationships between keys.

Note also that SID and EID are not key columns in the PurchaseOrder table. You can verify which columns should be keyed. Start with the POID column. For each PurchaseOrder (POID), how many suppliers are there? The business rule says only one supplier for a purchase order; therefore, SID should not be keyed, so do not underline SID. Now start with SID and work in the other direction. For each supplier, how many purchase orders are there? The business rule says many purchase orders can be sent to a given supplier, so the PID column needs to be a key. The same process indicates that EID should not be a key; it belongs in the PurchaseOrder table, since each Employee can place many orders. Figure 3.40 uses sample data to show how the tables are linked through the key columns.

Many-to-Many Relationships

Overview class diagrams often contain many-to-many relationships. However, in a relational database many-to-many relationships must be split into two one-to-many relationships to get to BCNF. Figure 3.41 illustrates the process with the PurchaseOrder and Item tables.

Each of the two initial entities becomes a table (PurchaseOrder and Item). The next step is to create a new intermediate table (POItem) that contains the primary keys from both of the other tables (POID and ItemID). This table represents the many-to-many relationship. Each purchase order (POID) can contain many items, so ItemID must be a key. Similarly, each item can be ordered on many purchase orders, so POID must be a key. Think of the PurchaseOrder and Item tables as the base tables that generate the purchase order and item data respectively. If you use generated key columns, new key values will be generated within those two tables when rows are added. The POItem table links the other two by using the existing key values. It indicates the individual items being purchased on a specific order.

You must have a table that contains both POID and ItemID as keys. Can you create this relationship without creating a third table? In most cases the answer is

Purchase Order

POID	Date	SID	EID
22234	9/9	5676	221
22235	9/10	5676	554
22236	9/10	7831	221
22237	9/11	8872	335

POItem

POID	ItemID	Quantity	Price
22234	444098	3	2.00
22234	444185	1	25.00
22235	444185	4	24.00
22236	555828	10	150.00
22236	555982	1	5800.00

Item

ItemID	Description	ListPrice
444098	Staples	2.00
444185	Paper	28.00
555828	Wire	158.00
555982	Sheet steel	5928.00
888371	Brake assembly	152.00

Figure 3.42

Sample data for the many-to-many relationship. Note that the intermediate POItem table links the other two tables. Verify that the three tables are in 3NF, where each nonkey column depends on the whole key and nothing but the key.

no. Consider what happens if you try to put the ItemID column into the Purchase-Order table and make it part of the primary key. The resulting entity would not be a 3NF table, because Date, SID, and EID do not depend on the ItemID. A similar problem arises if you try to place the POID key into the Item table. Hence the intermediate table is required. Figure 3.42 uses sample data to show how the three tables are linked through the keys.

N-ary Associations

As noted in Chapter 2, n-ary associations are denoted with a diamond. This diamond association also becomes a class. In a sense, an n-ary association is simply a set of several binary associations. As shown in Figure 3.43, the new association class holds the primary key from each of the other classes. As long as the binary associations are one-to-many, each column in the Assembly class will be part of the primary key. If for some reason a binary association is one-to-one, then the corresponding column would not be keyed.

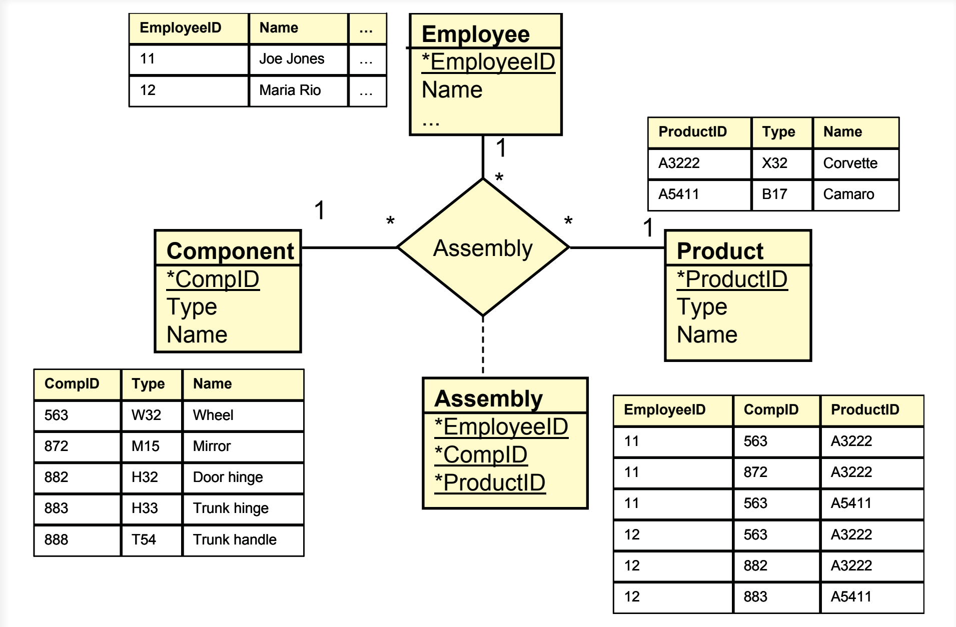
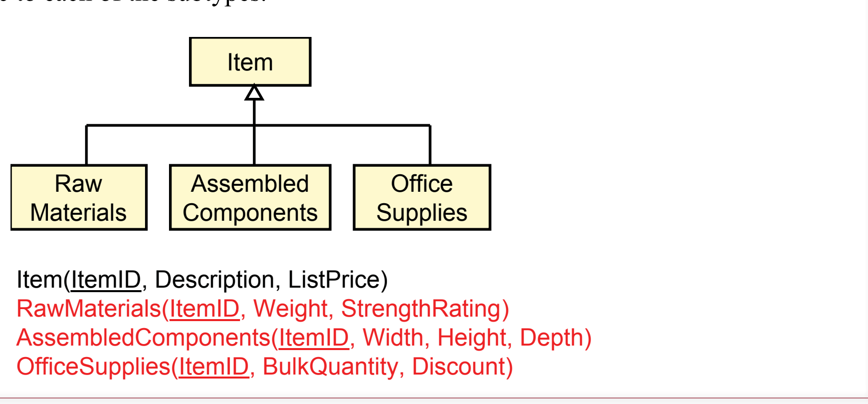


Figure 3.43

N-ary association. The Assembly association is also a class. It can be modeled as a set of binary (one-to-many) associations. The primary key from each of the main classes is included in the new Assembly class.

Figure 3.44

Converting subtypes. Every item purchased has basic attributes, which are recorded in the Item table. Each item can be placed in one of three categories, which have different attributes. To convert these relationships to 3NF, create new tables for each subtype. Use the same key in the new tables and in the generic table. Add attributes specific to each of the subtypes.



Item

ItemID	Description	ListPrice
444098	Staples	2.00
444185	Paper	28.00
555828	Wire	158.00
555982	Sheet steel	5928.00
888371	Brake assembly	152.00

RawMaterials

ItemID	Weight	StrengthRating
555828	57	2000
555982	2578	8321

AssembledComponents

ItemID	Width	Height	Depth
888371	1	3	1.5

OfficeSupplies

ItemID	BulkQuantity	Discount
444098	20	10%
444185	10	15%

Figure 3.45

Sample data for the subtype relationships. Notice how each Item has an entry in the Item table and a row in one of the three subtype tables.

Generalization or Subtypes

Some business entities are created as subtypes. Figure 3.44 illustrates this relationship with the Item entity. An item is a generic description of something that is purchased. Every item has a description and a list price. However, the company deals with three types of items: raw materials, assembled components, and office supplies. Each of these subtypes has some additional properties that you wish to track. For example, the company tracks the weight of raw materials, the dimension of assembled components, and quantity discounts for office supplies.

Two basic approaches exist for converting this design to a relational database. (1) If subtypes are similar, you could ignore the subclasses and compress all the subclasses into the main class that would contain every property for all of the subclasses. In this case each item entry would have several null values. (2) In most cases a better approach is to create separate tables for each subclass. Each table will contain the primary key from the main Item class.

As shown in Figure 3.45, each item has an entry in the Item table. The Item table contains attributes that apply to all of the subtypes (Description and ListPrice). Each item also has an entry in one of the three subtype tables, depending on the specific type of item. For example, item 444098 is described in the Item table and

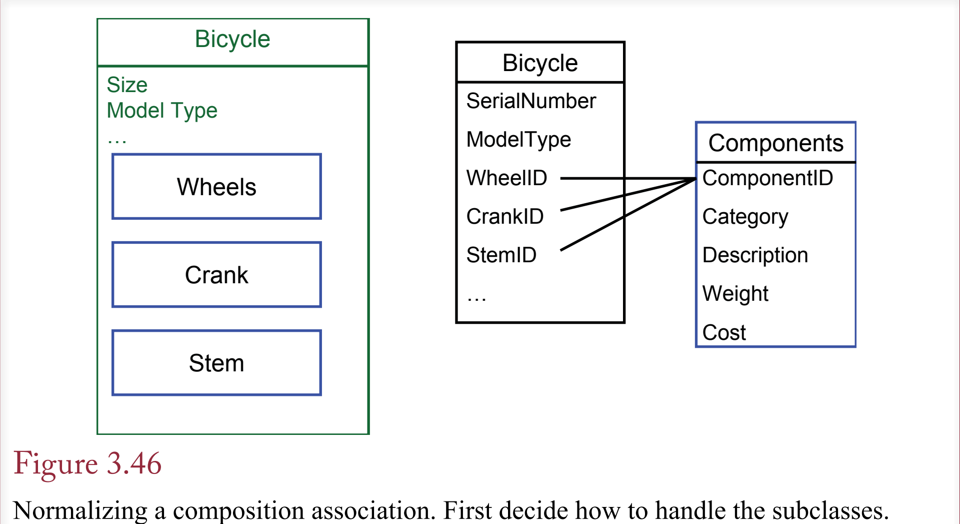


Figure 3.46

Normalizing a composition association. First decide how to handle the subclasses. In this case they are combined into one Components table. Second, handle the composition by storing the ComponentID and the SerialNumber as keys in a new junction table.

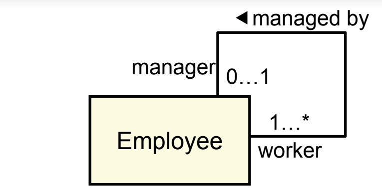
has additional data in the OfficeSupplies table. If the subclass relationships are not mutually exclusive, then each main item can have a matching row in more than one of the subclass tables.

In most cases, it is simpler to ignore the subtypes and put all of the columns into the single Item table. However, lumping the subtypes together could result in a large number of null values. If the amount of wasted space becomes a significant issue, or if you need to assign control of each subtype to a different department, you will need to create the separate tables. The major DBMSs have the ability to define subtables and can process queries with this structure automatically. If subtables are not available, or you choose not to use them, you will have to write queries to join the subtype tables back to the Item table.

Composition

In some ways composition is a combination of an n-ary association and subtypes. Consider the bicycle example in Figure 3.46, in which a bicycle is built from various components. The first decision to make is how to handle the many components. It is a question of subtypes. In this situation the business keeps almost identical data for each component (ID number, description, weight, cost, list price, and so on). Hence a good solution is to compress each subtype into a generic Component class. However, it would also make sense to handle wheels separately because they are a more complex component that is often built from other components.

You can solve the main composition problem by creating properties in the main Bicycle table for each of the component items (WheelID, CrankID, StemID, and so on). These columns are foreign keys in the Bicycle table (but not primary keys). When a bicycle is built, the ID values for the installed components are stored in the appropriate column in the Bicycle table. You can find more details by examining the actual Rolling Thunder database.



Employee(EID, Name, Salary, Address, Manager)

Employee

EID	Name	Salary	Address	Manager
221	Smith	67,000	223 W. 2300	335
335	Sanchez	82,000	37 W. 7200	
554	Johnson	35,000	440 E. 5200	335

Figure 3.47

Converting recursive relationships. An employee can have only one manager, so add a Manager column to the Employee table which contains the EID to point to the manager. In the example, Smith reports to Manager 335 (Sanchez).

Recursive (Reflexive) Associations

Occasionally, an entity may be linked to itself. A common example is shown in Figure 3.47, where employees have managers. Because managers are also employees, the entity is linked to itself. This relationship is easy to see when you create the corresponding table. Simply add a Manager column to the Employee table. The data in this column consists of an EID. For example, the first employee (Smith, EID 221) reports to manager 335 (Sanchez). Is the Manager column part of the primary key? No, because the business rule states that each employee can have only one manager.

How would you handle a situation in which an employee can have more than one manager? Key the Manager column? That would cause other problems, because the Employee table would not be in BCNF (an employee's address would not depend on the manager). The solution is to create a new table that lists EmployeeID and ManagerID—both part of the primary key. The new table would probably have additional data to describe the relationship between the employee and the manager, such as a project or task.

The Pet Store Example

What tables are needed for the Sally's Pet Store? To design the Sally's Pet Store database, you talk to the owner and investigate the way that other stores operate. In the process you collect ideas for various forms so you can learn the business rules. To expedite the development and hold down costs, you and Sally agree to begin with a simplified model and add features later. The sales form sketched in Figure 3.48 contains the primary data that will be needed when sales are made.

Sally wants you to create separate purchase orders for animals and products. She has repeatedly emphasized the importance of collecting detailed animal data. Eventually, Sally would also like to get medical records for the animals adopted through the store. Common data would include their shots, any illnesses, and any

Sales

SaleID Date

Customer	EmployeeID
Name Address City, State, ZIP	Name

Animal Adoption

ID	Name	Category	Breed	DoB	Gender	Reg	Color	Donation	Group

Merchandise Sale

Item	Description	Category	ListPrice	SalePrice	Quantity	Value

Merchandise Subtotal
Tax
Total

Figure 3.48

Pet Store sample sales form. Separate sections for selling animals and merchandise reflect a business rule to treat them differently.

medications or treatments they have received. For now, she is relying on the adoption groups to keep this information. However, once the sales and basic financial applications have been created, she wants to add these features to the database.

For the moment the most important job is to collect the transaction data. The design should make it easy to add new attributes for all of the major entities. It should also be easy to add new tables (such as health records) without making major alterations to the initial structure. In addition to sales, purchasing merchandise from suppliers is the other big transaction event.

A sample purchase order form is shown in Figure 3.49. Again, remember that Sally wants to start with a small database. Later you will have to collect additional data. For example, what happens if an order arrives and some items are missing? The current form can only record the arrival of the entire shipment. Similarly, each supplier probably uses a unique set of Item numbers. For example, a case of cat food from one supplier might be ordered with ItemID 3325, but the same case from a different supplier would be ordered with ItemID A9973. Eventually, Sally will probably want to track the numbers used by her major suppliers. That way, when invoices arrive bearing their numbers, matching the products to what she ordered will be easier.

The next step in designing the Pet Store database is to take each form and create a list of normalized tables that will be used to hold data for that form. Figure 3.50 shows the tables that were generated from the Sales form. Before examining the results in detail, you should attempt to normalize the data yourself. Then see whether you derived the same answer. You should also derive the normalized tables for the other two forms. Remember to double-check your work. First make sure the primary keys are correct, then check to see that each nonkey column depends on the whole key and nothing but the key.

Purchase Order for Merchandise

Order#

Date Ordered
Date Received

Supplier
Name
Contact
Phone
Address
City, State, ZIP Code

Employee ID
Name
Home Phone

ItemID	Description	Category	Price	Quantity	Ext.	QOH

Subtotal
Shipping Cost
Total

Figure 3.49

Pet Store sample purchase order for merchandise. Note the similarities and differences between the two types of orders. Keep in mind that additional data will have to be collected later.

Note that because each animal is unique, there is no SaleAnimal table—unlike the SaleItem table for merchandise. Because an animal can be adopted only one time, it is possible to put the Donation amount and the SaleID into the Animal table. Think about the keys for a minute. Each AnimalID can be adopted only one time, so SaleID is not a key column. But, multiple animals could be adopted at the same time, so SaleID needs to be in the Animal table where AnimalID is a primary key.

Merchandise is different because an ItemID refers to a relatively generic item—such as a bag of dog food. A bag of dog food with that ItemID can be purchased many times. Physically, it is a different bag, but a specific brand/type/size of dog food always has the same ItemID. Consequently, you need a table that links SaleID and ItemID where each SaleID can have many ItemIDs (key ItemID); and each ItemID can appear on many SaleIDs (key SaleID). In the SaleItem table, both SaleID and ItemID are part of the primary key.

View Integration

How do you combine tables from multiple forms and many developers? Up to this point, database design and normalization have been discussed using individual reports and forms, which is the basic step in designing a database. However, most projects involve many reports and forms. Some projects involve teams of designers, where each person collects forms and reports from different users and departments. Each designer creates the normalized list of tables for the individual forms, and you eventually get several collections of tables related to the same topic. At this point you need to integrate all these tables into one complete, consistent set of table definitions.

Sale(<u>SaleID</u>, Date, CustomerID, EmployeeID)
SaleAnimal(<u>SaleID</u>, <u>AnimalID</u>, SalePrice)
SaleItem(<u>SaleID</u>, <u>ItemID</u>, SalePrice, Quantity)
Customer(<u>CustomerID</u>, Name, Address, City, State, Zip)
Employee(<u>EmployeeID</u>, Name)
Animal(<u>AnimalID</u>, Name, Category, Breed, DateOfBirth,
Gender, Registration, Color, ListPrice)
Merchandise(<u>ItemID</u>, Description, Category, ListPrice)

Figure 3.50

Pet Store normalized tables for the basic sales form. You should do the normalization first and see if your results match these tables.

When you are finished with this stage, you will be able to enter the table definitions into the DBMS. Although you might end up with a large list of interrelated tables, this step is generally easier than the initial derivation of the 3NF tables. At this point you collect the tables, make sure everything is named consistently, and consolidate data from similar tables. The basic steps involved in consolidating the tables are as follows:

1. Collect the multiple views (documents, forms, etc.).
2. Create normalized tables for each document.
3. Combine the views into one complete model.

The Pet Store Example

Figure 3.51 illustrates the view integration process for the Pet Store case. The tables generated from the Sale and Purchase forms are listed first. The integration occurs by looking at each table to see which ones contain similar data. A good starting point is to look at the primary keys. If two tables have exactly the same primary keys, the tables should usually be combined. However, be careful. Sometimes the keys are wrong, and sometimes the keys might have slightly different names.

Notice that the Employee table shows up twice in the example. By carefully checking the data in each listing, you can form one new table that contains all of the columns. Hence the Phone and DateHired columns are moved to one table, and the two others are deleted. A similar process can be used for the Supplier, Animal, and Merchandise tables. The goal is to create a complete list of normalized tables that will hold the data for all the forms and reports. Be sure to double-check your work and to verify that the final list of tables is in 3NF or BCNF. Also, make sure that the tables can be joined through related columns of data.

The finalized tables can also be displayed on a detailed class diagram. The class diagram for the Pet Store is shown in Figure 3.52. A strength of the diagram is the ability to show how the classes (tables) are connected through relationships. Double-check the normalization to make sure that the basic forms can be re-created. For example, the sales form will start with the Customer, Employee, and Sale tables. The Animal table holds information about adoptions and donations. Sales of products requires the SaleItem and Merchandise tables. All of these tables can be connected by relationships on their attributes.

Sale(SaleID, Date, CustomerID, EmployeeID)
SaleAnimal(SaleID, AnimalID, SalePrice)
SaleItem(SaleID, ItemID, SalePrice, Quantity)
Customer(CustomerID, Name, Address, City, State, Zip)
Employee(EmployeeID, Name, Phone, DateHired)
Animal(AnimalID, Name, Category, Breed, DateOfBirth, Gender, Registration, Color, ListPrice)
Merchandise(ItemID, Description, Category, ListPrice, Cost)

AnimalOrder(OrderID, OrderDate, ReceiveDate, SupplierID, EmpID, ShipCost)
AnimalOrderItem(OrderID, AnimalID, Cost)
Supplier(SupplierID, Name, Contact, Phone, Address, City, State, Zip)
~~Employee(EmployeeID, Name, Phone, DateHired)~~
~~Animal(AnimalID, Name, Category, Breed, Gender, Registration, Cost)~~

MerchandiseOrder(PONumber, OrderDate, ReceiveDate, SID, EmpID, ShipCost)
MerchandiseOrderItem(PONumber, ItemID, Quantity, Cost)
~~Supplier(SupplierID, Name, Contact, Phone, Address, City, State, Zip)~~
~~Employee(EmployeeID, Name, Phone)~~
~~Merchandise(ItemID, Description, Category, QuantityOnHand)~~

Figure 3.51

Pet Store view integration. Data columns from similar tables can be combined into one table. For example, we need only one Employee table. Look for tables that have the same keys. The goal is to have one set of normalized tables that can hold the data for all the forms and reports.

Most of the relationships are one-to-many relationships, but pay attention to the direction. Access denotes the many side with an infinity (∞) sign. Of course, you first have to identify the proper relationships from the business rules. For instance, there can be many sales to each customer, but a given sale can list only one customer.

This final list shown in the class diagram in Figure 3.52 has three new tables: City, Breed, and Category. These validation tables have been added to simplify data entry and to ensure consistency of data. Without these tables employees would have to repeatedly enter text data for city name, breed, and category. There are two problems with asking people to type in these values: (1) it takes time, and (2) people might enter different data each time. By placing standardized values in these tables, employees can select the proper value from a list. Because the standard value is always copied to the new table, the data will always be entered exactly the same way each time it is used.

Asking the DBMS to enforce the specified relationships raises an interesting issue. The relationships require that data be entered in a specific sequence. The foreign key relationship specifies that a value for the customer must exist in the Customer table before it can be placed in the Sale table. From a business standpoint the rule makes sense; you must first meet customers before you can sell them something. However, this rule may cause problems for clerks who are entering sales data. You need some mechanism to help them enter new Customer data before attempting to enter the Sales data. Chapters 6 and 7 explain one way to resolve this issue.

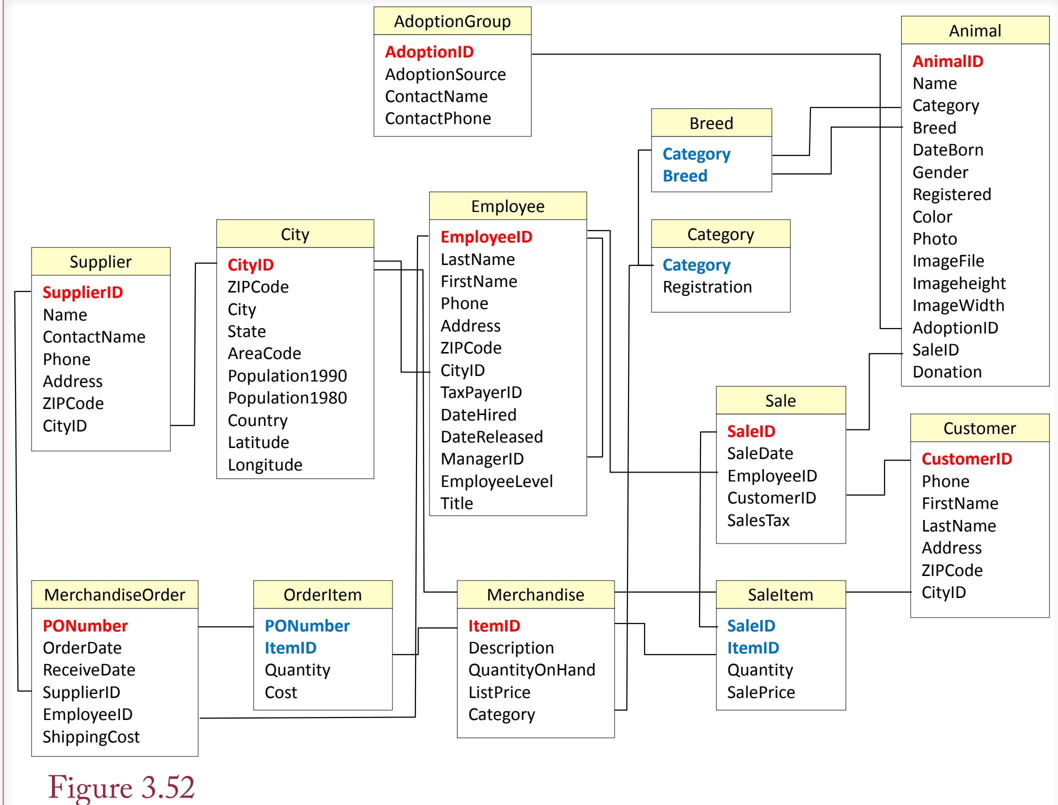


Figure 3.52

Pet Store class diagram. The tables become entities in the diagram. The relationships verify that the tables are interconnected through the data. Some new data has been added for the employees. Also, cities have been defined in a single table to simplify data entry. Likewise, the new Breed and Category tables ensure consistency of data.

Rolling Thunder Sample Integration Problem

The only way to learn database design and understand normalization is to work through more problems. To practice the concepts of data normalization and to illustrate the methods involved in combining sets of tables, consider a new problem involving a database for a small manufacturer: Rolling Thunder Bicycles. The company builds custom bicycles. Frames are built and painted in-house. Components are purchased from manufacturers and assembled on the bicycles according to the customer orders. Components (cranks, pedals, derailleurs, etc.) are typically organized into groups so that the customer orders an entire package of components without having to specify every single item. Additional details about bicycles and the company operations are available in the Rolling Thunder database.

To understand normalization and the process of integrating tables from various perspectives, consider four of the input forms: Bicycle Assembly, Manufacturer Transactions, Purchase Orders, and Components.

Builders use the Bicycle Assemble form shown in Figure 3.53 to determine the basic layout of the frame, the desired paint styles, and the components that need to be installed. As the frame is built and the components are installed, the workers check off the operations. The employee identification and the date/time are stored

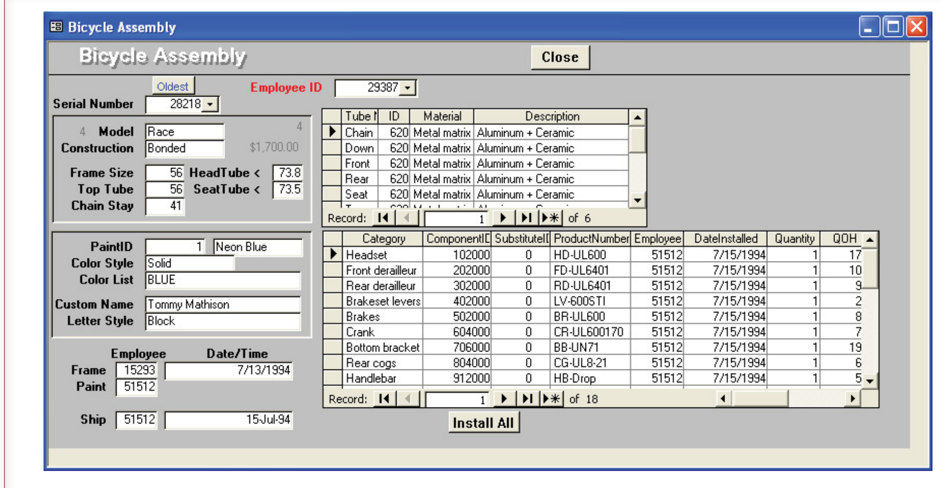


Figure 3.53

Bicycle Assembly form. The main EmployeeID control is not stored directly, but the value is entered in the FrameAssembler column of the Bicycle table when the employee clicks the Frame box.

in the database. As the parts are installed, the inventory count is automatically decreased. When the bicycle is shipped, a trigger executes code that records the price owed by the customer so a bill can be printed and sent.

Collecting the data columns from the form results in the notation displayed in Figure 3.54. Notice that two repeating groups (tubes and components) occur, but they repeat independently of each other. They are not nested.

Components and other supplies are purchased from manufacturers. Orders are placed as supplies run low and are recorded on a Purchase Order form. Shown in Figure 3.55, the Purchase Order contains standard data on the manufacturer, along with a list of components (or other supplies) that are ordered.

The notation and the 4NF tables are derived in Figure 3.56. For practice you should work through the normalization on your own. Note that the computed columns do not need to be stored. However, be careful to store the shipping cost and discount, since those might be negotiated specifically on each order.

Payments to manufacturers are collected with a basic transaction form shown in Figure 3.57. Note that the initial balance and balance due are computed by code behind the form to display the effects of adding new transactions. Row entries for purchases are automatically generated by the Purchase Order form, so this form is generally used for payments or for corrections.

The 4NF tables resulting from the manufacturer transactions are shown in Figure 3.58. Again, work through the normalization yourself. Practice and experience are the best ways to learn normalization. Do not be misled: It is always tempting to read the “answers” in the book and say that normalization is easy. Normalization becomes much more complex when you face a blank page. Investigating and determining business rules is challenging when you begin.

The Component form in Figure 3.59 is used to add new components to the list and modify the descriptions of the components. It can also be used to make changes to the manufacturer data. Notice the use of two identification numbers: one is

BicycleAssembly(SerialNumber, Model, Construction, FrameSize, TopTube,
ChainStay, HeadTube, SeatTube, PaintID, PaintColor, ColorStyle, ColorList,
CustomName, LetterStyle, EmpFrame, EmpPaint, BuildDate, ShipDate,
(Tube, TubeType, TubeMaterial, TubeDescription),
(CompCategory, ComponentID, SubstID, ProdNumber, EmpInstall, DateInstall,
Quantity, QOH)

Bicycle(SerialNumber, Model, Construction, FrameSize, TopTube, ChainStay,
HeadTube, SeatTube, PaintID, ColorStyle, CustomName, LetterStyle,
EmpFrame, EmpPaint, BuildDate, ShipDate)
Paint(PaintID, ColorList)
BikeTubes(SerialNumber, TubeID, Quantity)
TubeMaterial(TubeID, Type, Material, Description)
BikeParts(SerialNumber, ComponentID, SubstID, Quantity, DateInstalled, EmpInstalled)
Component(ComponentID, ProdNumber, Category, QOH)

Figure 3.54

Notation for the BicycleAssembly form. There are two repeating groups, but they are independent. The 4NF tables from this form are displayed, but you should try to derive the tables yourself.

assigned by Rolling Thunder, and the other is assigned by the manufacturer. Assigning our own number ensures consistency of the data format and guarantees a unique identifier. The manufacturer's product number is used to help place orders, since the manufacturer would have no use for our internal data.

The 4NF tables derived from the Component form are shown in Figure 3.60. For the most part they are straightforward. One interesting difference in Rolling Thunder is the treatment of addresses and cities. Many business tables for customers, employees, suppliers, and so on, contain columns for city, state, and ZIP code. Technically, there is a hidden dependency in this basic data because the three are

Figure 3.55

Tables from the Purchase Order form. Note that the computed columns (extension is price * quantity) are not stored in the tables.

PurchaseOrder(PurchaseID, PODate, EmployeeID, FirstName, LastName,
ManufacturerID, MfgName, Address, Phone, CityID, CurrentBalance,
ShipReceiveDate,
(ComponentID, Category,ManufacturerID, ProductNumber, Description, PricePaid,
Quantity, ReceiveQuantity, ExtendedValue, QOH, ExtendedReceived),
ShippingCost, Discount

PurchaseOrder(PurchaseID, PODate, EmpoyeeID, ManufacturerID, ShipReceiveDate,
ShippingCost, Discount)
EmployeeID(EmployeeID, FirstName, LastName)
Manufacturer(ManufacturerID, Name, Address, Phone, Address, CityID, CurrentBalance)
City(CityID, Name, ZIPCode)
PurchaseItem(PurchaseID, ComponentID, Quantity, PricePaid ReceivedQuantity)
Component(ComponentID, Category, ManufacturerID, ProductNumber, Description,
QOH)

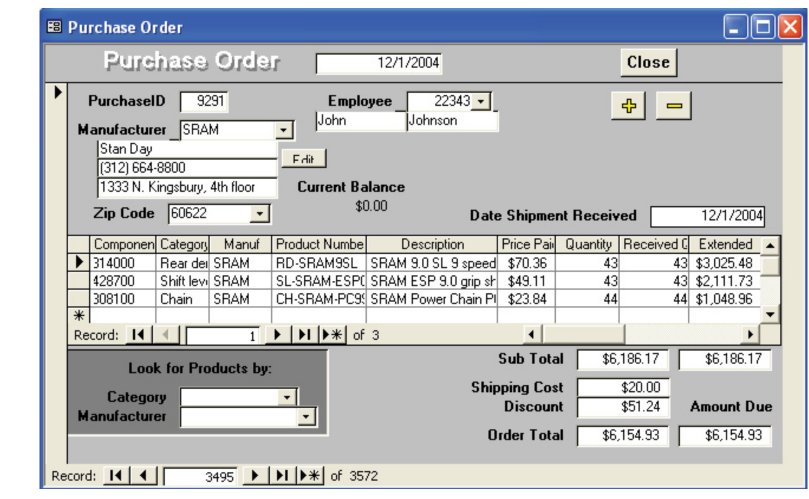
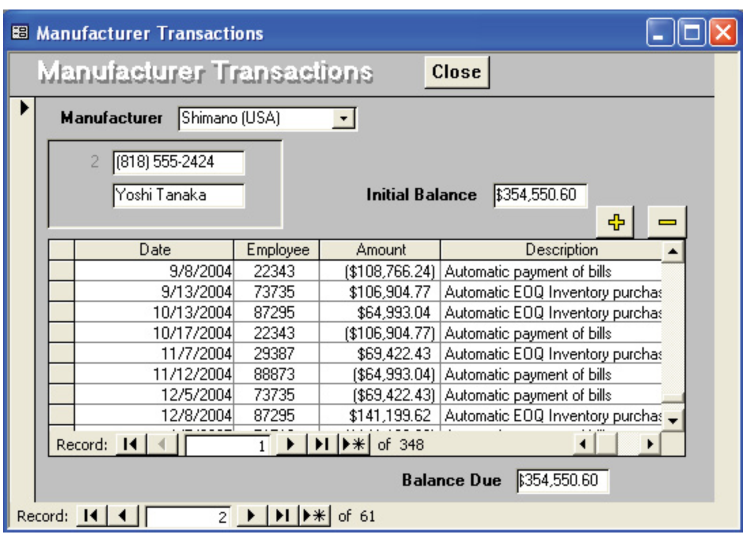


Figure 3.56

Purchase Order form. Only the items ordered is a repeating group. The Look for Products section is a convenience for users and does not store data. The Date Shipment Received box is initially blank and is filled in when the product arrives at the loading dock.

Figure 3.57

Manufacturer Transaction form. The balance due is stored in the database, but only one time. The Initial Balance and Balance Due boxes are computed by the form to display the effect of transactions added by the user.



ManufacturerTransactions(ManufacturerID, Name, Phone, Contact, BalanceDue,
(TransDate, Employee, Amount, Description)

Manufacturer(<u>ManufacturerID</u>, Name, Phone, Contact, BalanceDue)
ManufacturerTransaction(<u>ManufacturerID</u>, <u>TransactionDate</u>, EmployeeID, Amount,
Description)

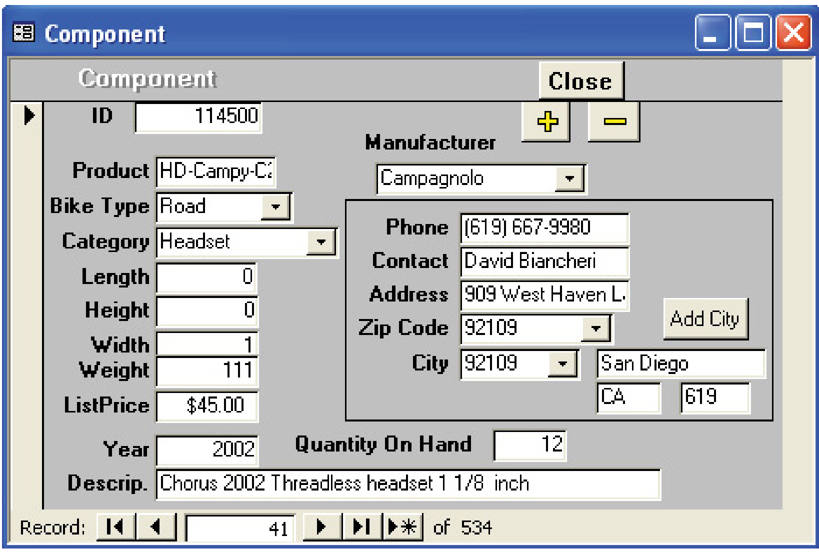
Figure 3.58

Tables for Manufacturer Transaction form. This normalization is straightforward. Note that the TransactionDate column also holds the time, so it is possible to have more than one transaction with a given manufacturer on the same day.

related. Hence a database can save space and data entry time by maintaining a separate City table. Of course, a City table for the entire United States, much less the world, could become large. A more challenging problem is that there is not a one-to-one relationship between cities and ZIP codes. Some cities have many ZIP codes, and some ZIP codes cover multiple cities. Rolling Thunder resolves these two issues by keeping a City table based on a unique CityID. If space is at a premium, the table can be reduced to contain only cities used in the database. As customers arrive from new cities, the basic city data is added. The ZIP code problem is handled by storing a base ZIP code for each city. The specific ZIP code related to each address is stored with the appropriate table (e.g., Manufacturer). This specific ZIP code could also be a nine-digit code that more closely identifies the location of the customer or manufacturer. Although it is possible to create a table

Figure 3.59

Component form. Note that components have an internal ID number that is assigned by Rolling Thunder employees. Products usually also have a Product number that is assigned by the manufacturer. It is difficult to rely on this number, since it might be duplicated across suppliers and the formats vary widely.



ComponentForm(ComponentID, Product, BikeType, Category, Length,
Height, Width, Weight, ListPrice, Description, QOH,
ManufacturerID, Name, Phone, Contact, Address, ZIPCode, CityID,
City, State, AreaCode)

Component(<u>ComponentID</u>, ProductNumber, BikeType, Category,
Length, Height, Width, Weight, ListPrice, Description, QOH,
ManufacturerID)
Manufacturer(<u>ManufacturerID</u>, Name, Phone, Contact, Address,
ZIPCode, CityID)
City(<u>CityID</u>, City, State, ZIPCode, AreaCode)

Figure 3.60

Tables derived from the Component form. The ZipCode in the Manufacturer table is specific to that company (probably a nine-digit code). The ZipCode in the City table is a base (five-digit) code that can be used for a reference point, but there are often many codes per city.

Figure 3.61

Integrated tables. Duplicate tables have been combined, and normalization (4NF) has been verified. Also draw a class diagram to be sure the tables link together. Note the addition of the Reference column as an audit trail to hold the corresponding PurchaseID. Observe that some tables (e.g., Employee) will need additional data.

Bicycle(<u>SerialNumber</u>, Model, Construction, FrameSize, TopTube, ChainStay, HeadTube,
SeatTube, PaintID, ColorStyle, CustomName, LetterStyle, EmpFrame,
EmpPaint, BuildDate, ShipDate)
Paint(<u>PaintID</u>, ColorList)
BikeTubes(<u>SerialNumber</u>, TubeID, Quantity)
TubeMaterial(<u>TubeID</u>, Type, Material, Description)
BikeParts(<u>SerialNumber</u>, ComponentID, SubstID, Quantity, DateInstalled, EmpInstalled)
Component(<u>ComponentID</u>, ProductNumber, BikeType, Category, Length, Height, Width,
Weight, ListPrice, Description, QOH, ManufacturerID)
PurchaseOrder(<u>PurchaseID</u>, PODate, EmployeeID, ManufacturerID,
ShipReceiveDate, ShippingCost, Discount)
PurchaseItem(<u>PurchaseID</u>, <u>ComponentID</u>, Quantity, PricePaid, ReceivedQuantity)
Employee(<u>EmployeeID</u>, FirstName, LastName)
Manufacturer(<u>ManufacturerID</u>, Name, Contact, Address, Phone,
CityID, ZipCode, CurrentBalance)
ManufacturerTransaction(<u>ManufacturerID</u>, TransactionDate, EmployeeID, Amount,
Description, Reference)
City(<u>CityID</u>, City, State, ZipCode, AreaCode)

PO	Manufacturer(ManufD, Name, Address, Phone, CityID, CurrentBalance)
Mfg	Manufacturer(ManufID, Name, Phone, Contact, BalanceDue)
Comp	Manufacturer(ManufID, Name, Phone, Contact, Address, ZIPCode, CityID)

Figure 3.62

Multiple versions of the Manufacturer table. Tables with the same key should be combined and reduced to one table. Moving Contact and ZipCode to the first table means the other two tables can be deleted. Do not be misled by the two names (CurrentBalance and BalanceDue) for the same column.

of complete nine-digit codes, the size is enormous, and the data tends to change. Companies that rely heavily on nine-digit mailings usually purchase verification software that contains authenticated databases to check their addresses and codes.

Look at the tables from Figures 3.54, 3.55, 3.59, and 3.60 again. Notice that similar tables are listed in each figure. In particular, look for the Manufacturer tables. Observe that the overlapping tables often contain different data from each form. In practice, particularly when there is a team of designers, similar columns might have different names, so be careful. The objective of this step is to combine the similar tables. The best way to start is to look for common keys. Tables that have the same key columns should be combined. For example, the Manufacturer variations are reproduced in Figure 3.61. The version from the PO table can be extended by adding the Contact and ZIPCode columns from the other variations.

After combining duplicate tables, you should have a single list of tables that contain all of the data from the forms. This list is shown in Figure 3.62. It is also a good idea at this point to double-check your work. In particular, verify that the keys are unique and that composite keys represent many-to-many relationships. Then verify the 3NF rules: Does each nonkey column depend on the whole key and nothing but the key? Also look for hidden dependencies that you might need to make explicit. Be sure that the tables can be linked back together through the data in the columns. You should be able to draw lines between all the tables. Now is a good time to draw a more complete class diagram. Each of the normalized tables becomes an entity. The relationships show how the tables are linked together. (See the Rolling Thunder database for the complete example.)

Finally, examine each table and decide whether you might want to collect additional data. For example, the Employee table would undoubtedly need more data, such as Address and DateHired. Similarly, you saw that the ManufacturerTransaction table could use a Reference column that will contain the PurchaseID when a transaction is automatically generated by the Purchase Order form. This column functions as an audit trail and makes it easier to trace accounting transactions back to the source. Some people might use date/time for the same purpose, but a round-off to seconds could cause problems.

Data Dictionary

How do you record the details for all of the columns and tables? In the process of collecting data and creating normalized tables, be sure to keep a data dictionary to record the data domains and various assumptions you make. A **data dictionary** or **data repository** consists of **metadata**, which is data that describes the data stored in the database. It typically lists all of the tables, columns, data domains,

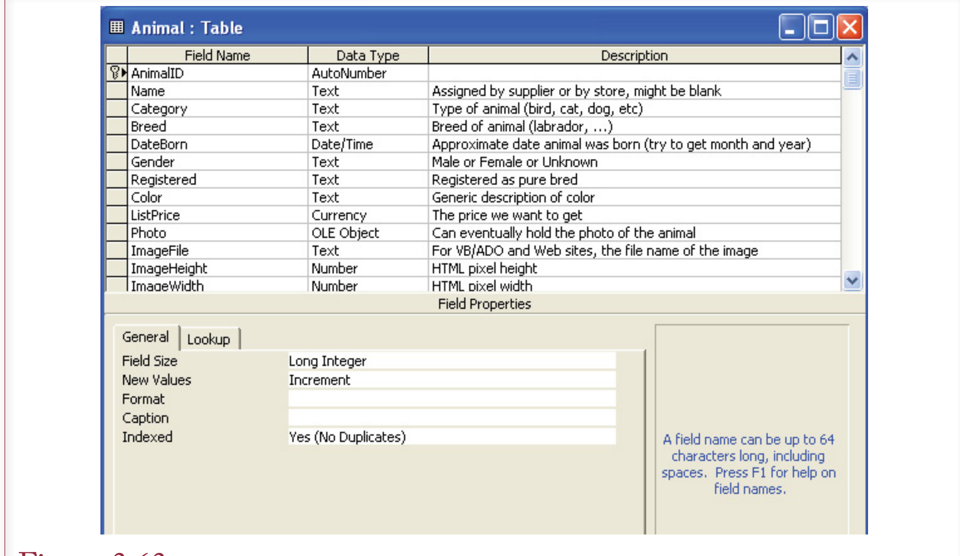


Figure 3.63

Table definition in Microsoft Access. Note the primary key indicator. Also note that text size limits and numeric subtypes are defined in the list at the bottom of the form.

and assumptions. It is possible to store this data in a notebook, but it is easier to organize if it is stored on a computer. Some designers create a separate database to track the underlying project data. Specialized computer tools known as computer-aided software engineering (CASE) tools help with software design. One of their strengths is the ability to create, store, and search a comprehensive data dictionary.

DBMS Table Definition

When the logical tables are defined and you know the domains for all of the columns, you can enter the tables into a DBMS. Most systems have a graphical interface that makes it easier to enter the table definitions. In some cases, however, you might have to use the SQL data definition commands described in Chapter 4. In both cases, the process is similar. Define the table name, enter the column names, select the data type for the column, and then identify the keys. Sometimes keys are defined by creating a separate index. Some systems enable you to create a description for each column and table. This description might contain instructions to users or it might be an extension of your data dictionary to help designers make changes in the future.

At this time, you should determine which keys you want to generate with an autonumber function. Similarly, identify any computed columns and specify the calculations needed for them. Some databases enable you to store these calculations within the database definition; others require that you write them into queries.

You can also set **default values** for each column to speed up data entry. In the video store example, you might set a default value for the base rental rate. Default values can be particularly useful for dates. Most systems enable you to set a default value for dates that automatically enters the current date. At this point you should also set validation rules to enforce data integrity. As soon as the tables are defined, you can set relationships. In Microsoft Access, go to the relationships

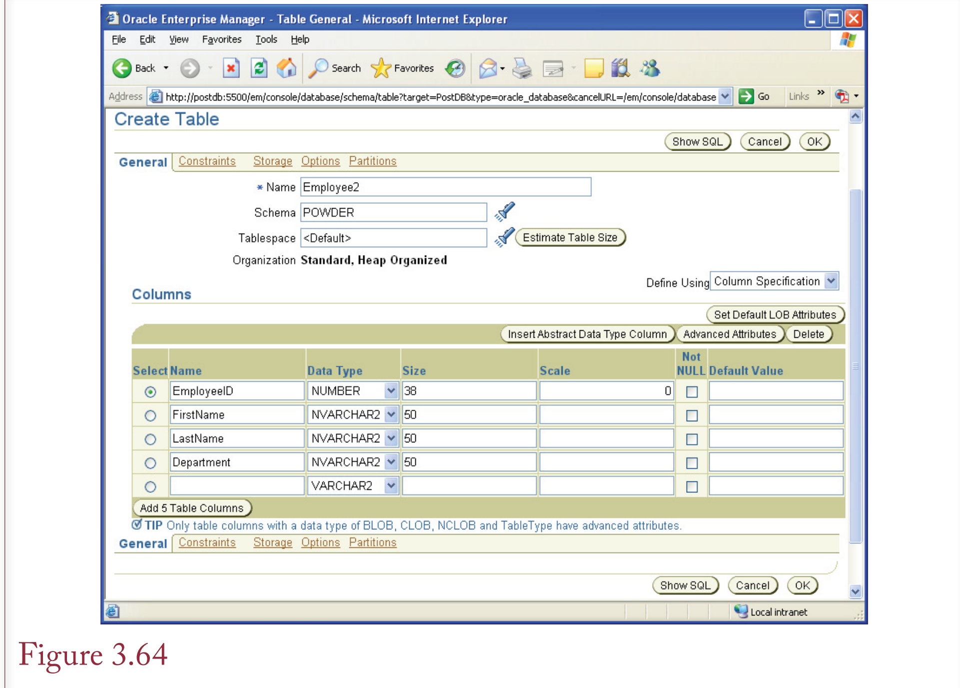


Figure 3.64

Oracle Schema Manager for creating tables. Primary keys and foreign keys are set in the Constraints tab. For primary key columns, be sure to also check that the values cannot be null.

screen, add all of the tables, then draw lines to show the connections (much like the class diagram). Be sure to check the boxes that specify "Enforce referential integrity," "Cascade on delete," and "Cascade on update." With SQL Server and Oracle, you specify referential integrity as constraints when you define the tables.

Figure 3.63 shows the form that Microsoft Access uses to define tables. Primary keys are set by selecting the appropriate rows and clicking an icon. Data type details such as character length and numeric subtype are set in the list at the bottom of the form. Changes to the table can be made at any time, but if data already exists in the table, you might lose some information if you select a smaller data type.

Figure 3.64 shows the form that can be used within Oracle to create tables—it is the Schema Manager. Primary and foreign keys can be set using the Constraints tab. Keep in mind that once you create a table in Oracle (and SQL Server), it can be difficult to change later. It is always possible to add new columns, but you might not be permitted to change the data type of an existing column or to delete columns.

If you are using Oracle version 8 or above, perform one additional step once the tables have been created by telling it to analyze the tables. For example, use the SQL Plus tool to issue commands for each table similar to: Analyze table Animal compute statistics. These commands tell Oracle to generate statistics for each table (notice the Statistics tab in the Schema Manager). Oracle uses these statistics to dramatically improve performance of queries.

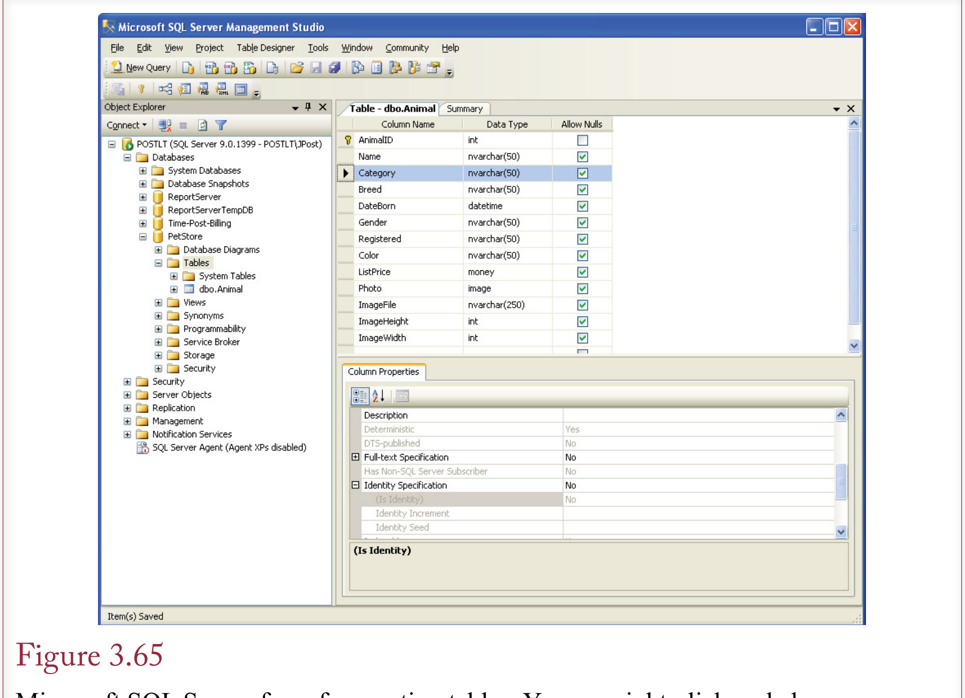


Figure 3.65

Microsoft SQL Server form for creating tables. You can right-click and choose a menu item to create relationships.

Figure 3.65 shows the forms that SQL Server uses to create and edit tables. You can right-click the design to set relationships and constraints. Relationships are defined as constraints and all constraints are stored as separate rules.

It is easy to see the similarities of the database design tools for the various products. Yet more important differences between the systems lie in the data types used within each system. Although some of the data type names appear similar, be particularly careful with Oracle databases. Oracle's underlying data types are different—particularly in dealing with numbers. Over time, the DBMS vendors have loosely adopted the SQL standard data types. In fact, you can use the generic names in almost any DBMS. However, each vendor adds additional types or interesting twists. Designers often work with the vendor's native data types instead of the generic names.

For both Oracle and SQL Server, the graphical forms seem easy to use; however, experienced developers almost always rely on direct SQL statements stored in a text file to create tables. Figure 3.66 gives an example for the Animal table. You simply create a list of all of the table creation statements and store them in a text file. This file is read by the database SQL processor to create the tables. The list has several advantages over the graphical approach: (1) It is easier to change the text file. (2) It is easier to re-create the tables on a different database or different computer. (3) Changing a table definition usually requires creating a new table, copying in the existing data, deleting the old table, and renaming the new one. With the text file, you can quickly define the new table and run the statement to create the table. (4) It is easier to specify the primary key and foreign key relationships in the text file. Most of the graphical approaches are cumbersome and hard

```
CREATE TABLE Animal
(
        AnimalID  INTEGER,
        Name                VARCHAR2(50),
        Category  VARCHAR2(50),
        Breed               VARCHAR2(50),
        DateBorn DATE,
        Gender              VARCHAR2(50)
                  CHECK (Gender='Male' Or Gender='Female'
                            Or Gender='Unknown' Or Gender Is Null)
        Registered          VARCHAR2(50),
        Color               VARCHAR2(50),
        ListPrice  NUMBER(38,4)
                  DEFAULT 0,
        Photo               LONG RAW,
        ImageFile VARCHAR2(250),
        ImageHeight         INTEGER,
        ImageWidth          INTEGER,
                  CONSTRAINT pk_Animal PRIMARY KEY (AnimalID)
                  CONSTRAINT fk_BreedAnimal FOREIGN KEY (Category, Breed)
                  REFERENCES Breed(Category, Breed)
                  ON DELETE CASCADE
        CONSTRAINT fk_CategoryAnimal FOREIGN KEY (Category)
                  REFERENCES Category(Category)
                  ON DELETE CASCADE
);
```

Figure 3.66

Oracle SQL Statements to create the Animal table. The statements for SQL Server are similar—just change the data types.

to read. Also, some versions of Oracle had stricter limits using the graphical interface that could be avoided by creating the SQL statements directly. (5) Because of the foreign key constraints, the order in which the tables are created is critical. You cannot refer to a table in the foreign key constraint unless that table already has been created. For example, the Category and Breed tables must be created before the Animal table. Keeping the table definitions in a text file means you only have to set up the sequence one time. If you are uncertain about the SQL syntax for creating a table, you can examine existing structure files, or you can use the Schema Manager to enter the basic information, then click the Show SQL button to cut and paste the underlying SQL code.

Note that the DB Design tool can automatically generate the CREATE TABLE commands using the data types for each of the major DBMSs. It even analyzes the foreign key references and generates the tables in the proper order. If you prefer to stick with the individual vendor tools, note that almost all of them (except Access) have an option to display the CREATE TABLE command defined by the visual tool. You can copy-and-paste this command into a separate text file for future reference. However, you will have to sort the command into the correct order yourself.

Data Volume and Usage

One more step is required when designing a database: estimating the size of the resulting database. The process is straightforward, but you have to ask a lot of questions of the users. When you design a database, it is important to estimate the overall size and usage of the database. These values enable you to estimate the hardware requirements and cost of the system. The first step is to estimate the size of the tables. Generally, you should investigate three situations: How big is the

Customer(C#, Name, Address, City, State, Zip)
Row: 4 + 15 + 25 + 20 + 2 + 10 = 76

Order(O#, C#, Odate)
Row: 4 + 4 + 8 = 16

OrderItem(O#, P#, Quantity, SalePrice)
Row: 4 + 4 + 4 + 8 = 20

$$Orders\ in\ 3\ yrs = 1000\ Customers * \frac{10\ Orders}{Customer} * 3\ yrs = 30{,}000$$

$$OrderLines = 30{,}000\ Orders * \frac{5\ Lines}{Order} = 150{,}000$$

Business rules

- Three year retention.
- 1000 customers.
- Average 10 orders per customer per year.
- Average 5 items per order.

Customer	76 * 1000	76,000
Order	16 * 30,000	480,000
OrderItem	20 * 150,000	3,000,000
Total		**3,556,000**

Figure 3.67

Estimating data volume. First estimate the size of each row, and then estimate the number of rows in the table. If there is a concatenated key, you will usually multiply an average value times the number of rows in a prior table, as in the calculation for OrderItem.

database now? How big will the database be in 2 or 3 years? and How big will the database be in 10 years?

Begin with the list of normalized tables. The process consists of estimating the average number of bytes in each row of the table and then estimating the number of rows in the table. Multiply the two numbers to get an estimate of the size of the table, and then add the table sizes to estimate the total database size. This number represents the minimum size of the database. Many databases will be three to five times larger than this base estimate. Some systems have more complex rules and estimation procedures. For example, Oracle provides a utility to help you estimate the storage required for the database. You still begin with the data types for each column and the approximate number of rows. The utility then uses internal rules about Oracle's procedures to help estimate the total storage space needed.

An example of estimating **data volume** is presented in Figure 3.67. Consider the Customer table. The database system sets aside a certain amount of storage space for each column of data. The amount used depends on the particular system, so consult the documentation for exact values. In the abbreviated Customer table, the identification number takes 4 bytes as a long integer, and you estimate that Names take an average of 15 characters. Other averages are displayed in the table. Better estimates could be obtained from statistical analysis of sample data. In any case the estimated size of one row of Customer data is 76 bytes. Evaluating the business provides an estimate of approximately 1,000 customers; hence, the Customer table would be approximately 76K bytes.

Estimating the size of the Order table follows a similar process, yielding an estimate of 16 bytes per row. Managers might know how many orders are placed in a given year. However, it might be easier to obtain the average number of orders placed by a given customer in 1 year. If that number is 10, then you could

expect 10,000 orders in a given year. Similarly, to get the number of rows in the OrderItem table, you need to know the average number of products ordered on one order form. If that number is 5, then you can expect to see 150,000 rows in the OrderItem table in 1 year.

The next step is to estimate the length of time data will be stored. Some companies plan to keep their data online for many years, whereas others follow a strict retention and removal policy. For legal purposes data must be maintained for a certain number of years, depending on its nature. Keep in mind that agencies such as the IRS also require that retrieval software (e.g., the DBMS) be available to reproduce the data.

In addition to the basic data storage, your database will also reserve space for indexes, log files, forms, programs, and backup data. Experience with a particular database system will provide a more specific estimate, but the final total will probably be three to five times the size of the base estimate.

The final number will give you some idea of the hardware needed to support the database. Although performance and prices continue to change, only small databases can be run effectively on personal computers. Larger databases can be moved to a file server on a local area network (LAN). The LAN provides access to the data by multiple users, but performance depends heavily on the size of the database, the characteristics of the DBMS, and the speed of the network. As the database size increases (hundreds or thousands of megabytes), it becomes necessary to move to a dedicated computer to handle the data. Very large databases (terabytes) need multiple computers and specialized disk drives to minimize capacity and performance bottlenecks. The data estimates do not have to be perfect, but they provide basic information that you can give to the planning committee to help allocate funds for development, hardware, software, and personnel.

While you are talking with the users about each table, you should ask them to identify some basic security information. You will eventually need to assign security access rights to each table. Chapter 10 presents the details, but for now you should find out which people use the table, and which people should be denied some privileges. For example, clerks who order merchandise should not be allowed to acknowledge receipt of that merchandise. Otherwise, an unethical clerk could order merchandise, record it as being received, and then steal it. Four basic operations can be granted to data: read it, change it, delete it, or add new data. You should keep a list of who may or may not access each table.

Summary

Database design relies on normalization, or the process of splitting data into tables. Ultimately, each table refers to a single entity or concept. Each table must have a primary key that uniquely identifies each row of data. To create the tables, you begin with a collection of data—generally derived from a user form or report. You reach 1NF by finding the repeating groups of data and putting them in a separate table. Next, you go through each of the intermediate tables and identify primary keys. You reach 2NF by checking each nonkey column and asking whether it depends on the whole key. If not, put the column into a new table along with the portion of the key that it does depend on. To reach 3NF, you check to see whether the nonkey column depends on anything that is not in the key. If so, pull out the column and the dependent column and put them into a new table. BCNF states that you cannot have hidden dependencies—all dependencies must be part of the primary key. 4NF looks at problems with keys, and states that you cannot have

two (or more) independent relationships within one table. In all cases, when you find incorrect dependencies or hidden dependencies, you solve the problem by splitting the tables and making the dependency explicit with a primary key.

Each form, report, or description that you collect from a user must be analyzed and a set of 4NF tables defined. For large projects several analysts may be given different forms, resulting in several lists of normalized tables. These tables must then be integrated into one standardized set of normalized data tables. Along the way you must specify the domain, or type of data, for each column. This final list of tables, with any comments, will be entered into the DMBS to start the database construction.

You should also collect estimates of data volume in terms of number of rows for each table. These numbers will enable you to estimate the average and maximum size of the database so that you can choose the proper hardware and software. You should also collect information on security conditions: Who owns the data? Who can have read access? Who can have write access? All of these conditions can be entered into the DBMS when you create the tables.

At this point, after you review your work, you can enter sample data to test your tables. When you are certain that the design is complete and accurate, you can begin building the application by constructing queries and creating forms and reports.

A Developer's View

Miranda learned that the class diagram is converted into a set of normalized tables. These tables are the foundation of the database application. Database design is crucial to developing your application. Engrave the basic normalization rule onto the back of your eyelids: Each nonkey column depends on the whole key and nothing but the key. Since the design depends on the business rules, make certain that you understand the rules. Listen carefully to the users. When in doubt, opt for flexibility. For your class projects, you should now be able to create the list of normalized tables. You should also be able to estimate the size of the database.

Key Terms

atomic
autonumber
Boyce-Codd normal form (BCNF)
cascading delete
composite keys
data dictionary
data integrity
data repository
data volume
default values
deletion anomaly
dependence
domain-key normal form (DKNF)
first normal form (1NF)
foreign key
fourth normal form (4NF)
globally-unique identifier (GUID)
hidden dependency
insertion anomaly
master-detail
metadata
pseudo column
referential integrity
repeating groups
second normal form (2NF)
surrogate keys
third normal form (3NF)

Review Questions

1. What is *dependency*?
2. What are the three main rules for normalization?
3. What problems do you encounter if data is not stored in normalized tables?

4. How are BCNF and 4NF different from 3NF?
5. What are the primary types of data that can be stored in a table?
6. Why is referential integrity important?

7. What complications are caused by setting referential integrity rules?
8. What problems do object-oriented designs cause in a relational database model and how do you compensate for them?
9. What elements do you look for when integrating views?
10. How do you estimate the potential size of a database?

Exercises

1. A local family has a large garden and regularly sells produce at the local Farmer's Market. Up to now the group has just picked items and sold them each week—basically tracking just the amount of money received. Now the family wants to track sales by types of items (potatoes, lettuce, tomatoes, carrots, and so on); both in terms of quantity sold and the amount of money received. They want to use the data to determine planting amounts for the coming year. The crops require about the same level of fertilizer and watering so profits are mainly determined by the yield and the price received. No one wants to create individual item receipts for each sale—that would take too much time, but they will use a tally sheet to record the number of items sold and the prices. Then enter the sales into a computer (or tablet) at the end of the day. Some items are sold by the unit (such as melons or lettuce--bunch) while others, such as carrots, are sold by the pound. The family starts out the day with a set price, but if items are not selling well and have a limited shelf life, the price is reduced. So the amount sold needs to be recorded at each price point.

Market Location Date Family member in charge	Weather comments Crowd comments Competition

Item	Quantity at Start	Sale Price	Quantity Sold	Unit/ Pounds	Comments
Carrots	*20*	*$1.00*	*5*	*Pounds*	*Few buyers*
Carrots		*0.50*	*10*	*Pounds*	
Tomatoes	*40*	*$2.00*	*20*	*Pounds*	
Tomatoes		*$1.50*	*10*	*Pounds*	
Melons	*10*	*$3.00*	*10*	*Units*	*Cantaloupe*

2. Your doctor told you that you need to get more exercise—particularly strength-based. So you decided to start lifting weights at the gym. To make it more interesting, you created an application to track your progress on each exercise. To make it easier to enter data, the application is based on the day and enables you to enter additional information such as health (*excellent, good, tired, weak, sick*), and general comments. For most weight-lifting exercises, you perform multiple lifts (repetitions) within a set, then change the weight for a new set and do another set of repetitions.

Date: Weight: Comments	Health: <pick> Total time:

<table>
<tr><th>Exercise</th><th>Weight/set</th><th># Reps</th><th>Comments</th></tr>
<tr><td rowspan="3"></td><td></td><td></td><td rowspan="3"></td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
<tr><td rowspan="3"></td><td></td><td></td><td rowspan="3"></td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
</table>

3. You have been hired by the student sports director to build a database for the intramural basketball leagues. The leagues have several teams each year. For the most part the teams get to select the level of competition: A, B, or C. There is more prestige to winning the A (or B) league so the better teams select into that league. There is also a league for teams composed of players all under 6 feet, and there is also a co-ed league that requires at least 2 women on the floor for each game. For the most part, players are assigned to a single team, but they do have the ability to switch to a different team if they desire. So players sign up for each team game but no one tracks points scored by each person. Only the total team points and win/loss are tracked.

Team Name League

Game Date, Time, Court playoff y/n
Referee, phone

Player	Student/faculty/staff	Height

Opponent

Points Opponent Points
Won/Loss

4. You recently added the twentieth device to your wireless network. Well, it feels like 20, but it might be 10 or even 30; you no longer remember which devices have wireless and when they were last updated. Partly to gain faster speeds (to handle all of the devices), you are planning to replace your main wireless router. Which means you will have to update all of the wireless devices. And then you will have to deal with all of your friends' wireless devices when they come by. You considered buying a commercial network management tool but that seems too expensive. Instead, you want to build a small database application to record basic information about the wireless devices and the IP addresses they are assigned so that you can quickly identify each device when you need it. All devices have a unique MAC (media access control) address which the network uses to identify the item. Sometimes you have to change the MAC address, such as with routers so the upstream device can recognize it (common with cable modems). A big problem with wireless is that older devices do not support newer standards and you have to decide if you want to lower your overall standards to support older devices, or run two or three different wireless systems. You also need to track the owner of the device to decide if you want to remove support for it. Most devices use dynamic assignment (DHCP) to obtain a local IP address so it can change over time. But sometimes you need to set a static address to a device to make it accessible (such as older printers). You only need the IP address once in a while but when you need it you want to record it in case you might need it again. The values you need the most often can eventually be made static.

<table>
<tr><td>Device Name MAC Address
Category (router, cable modem, bridge, PC, phone, …)

Year
Highest wireless standard (b, g, n, ac, …)
Max bit rate (56, 100, 150, 300, 600, 900, 1000, …)
Highest encryption (DES, AES, WPA2, …)

Upstream network device:</td><td>Owner

Date updated
Source of update</td></tr>
</table>

Date	IP Address	DHCP/Static	Comments

5. A friend of yours is starting a business to build semi-customized cases for cell phones and laptops. The focus of the case is in the design—colors, logos, artwork and so on. She plans to buy relatively plain cases and then paint them with various designs. She can even take photographs from customers and incorporate those into the artwork. Existing stores and Web sites focus on finding a case to fit a particular device, but she thinks the process should work the other direction. Customers will pick the design and artwork and then specify the device. Cases for popular devices will be supported automatically and kept in stock, but other devices will take time to customize. The device aspect ratio is a particular problem because the artwork is relatively easy to scale up or down in size but not if the ratio of width to height changes too far. Then it has to be redrawn. And Apple devices tend to use the older 4:3 aspect ratio compared to other companies that use the newer HD 16:9 ratio.

Design Name
Description
Category
Aspect Ratio

Base price

Basic colors

File

Date created
Date modified
Date stopped

Artist
Phone
Commission rate

Standard Stock Devices

Device Name	Price	Quantity on Hand	Device type	Height	Width

Customer name
Phone
Address
City, State, ZIP

Sale Date
Shipment date

Item	Phone/Device	Sale Price	Quantity	Stock/Custom

Payment Method

Subtotal
Tax
Shipping
Total

6. A friend of yours is starting a "vintage" clothing shop. She will buy older clothes (lightly used) from people, clean and mend them if necessary, and then sell them in her store. Some older fashion items often become popular years later and "fashionistas" mix and match items to achieve their own style. But as the inventory grows, your friend needs a better way to track which items are popular to help figure out what price she should pay for clothing and when prices on existing items should be changed. Occasionally, a good customer will ask her to keep an eye out for a special item—particularly in terms of sizes. So she also needs to keep a list of search items. Currently, she tracks items by placing a tag on them when she buys them. The tag includes a basic description of the item, where and when she bought it, and eventually, she adds data to the tag that says when it was sold, the sale price and then the price/cost she paid for the item.

<table>
<tr><td>Item Description
Category
Men/Women/Child
Base color
Size</td><td>Designer/Manufacturer
Mfg. Original Price
History if known</td></tr>
<tr><td>Purchase Location
Purchase Date
Amount (coded on tag)
Condition</td><td>Repairs made:</td></tr>
<tr><td>Sale Date
Customer Name
E-mail
Phone</td><td>Sale Price</td></tr>
</table>

<table>
<tr><td>Search Item Description
Designer
Color Desired Size
Category
Estimated Price
Substitutions (such as color)</td><td>Date Start
Date Needed
Date Found

Location
Price Paid
Condition</td></tr>
<tr><td colspan="2">Customer Name
Phone
E-mail

Sale Date
Sale Price</td></tr>
</table>

7. You have volunteered to work for a local politician who is a friend of the family. She wants a system to track information about contacts with voters. Most voters in the district do not contact the office, but she wants to be sure to track information on those who do make contact. It is particularly important to track those who have strong opinions on various legislation. It is also important to track those who need rides to the polls each year or help with absentee ballots. And it is absolutely critical to track the requests for money sent to each person and the amount received each time. Note that houses are assigned to districts and regions; but those designations can change over time. Donors are often organized into groups where one person (a consolidator) collects money from other donors. Over time, voters and donors are assigned various tags (such as whale, whiner, or various issues) which are used to target future letters to each person.

Voter Last Name, First Name, Gender Phone Address, City, State, ZIP District #, Region	Party (repub/democ/ind) Probability of voting way we want:

Contact from voter

Date/Time	Method	Reason	Importance	Staff member Phone Vol/Paid

Fund-raising campaign
Campaign year

Event/Mailing/Request

Date	Event/Mailing	Primary topic	Location	Target

Donor Name E-mail Consolidator Labels/Tags	Amount	Date	Event	Comments

Sally's Pet Store

8. Define the tables needed to extend the Pet Store database to handle genealogy records for the animals.
9. Define the tables needed to extend the Pet Store database to handle health and veterinary records for the animals.
10. Sally wants to add payroll and monthly employee evaluation information to the database. Define the tables needed.
11. Sally wants to add pet grooming services. Define the tables necessary to schedule appointments, assuming two workers will be dedicated to this area.

Rolling Thunder Bicycles

12. Using the class diagram, identify five business rules that are described by the table definitions and table relationships (similar to the ListPrice rules described by the Sale example).

13. The company wishes to add more data for human resources, such as tax withholding, benefits selected, and benefit payments by the employees and by the company. Research common methods of handling this type of data and define the required tables.

Corner Med

14. Physicians and medical administrators are often interested in a hierarchical classification of illnesses and diagnoses. Some of the hierarchy is built into the ICD codes, but the managers and physicians want to be able to create reports that roll up the weekly diagnoses into specific categories. They also want the ability to define new categories. Essentially, the physician administrators will create a medical category and list the various conditions that apply to the category. For example, Broken Bones could be a general category, and specific fractures (e.g., S62.2 Fracture of first metacarpal bone using ICD10) would comprise the list of conditions. Define the table(s) needed to handle this summarization data. Optional: What if the physicians what to create multiple levels within the summary data? For example, Family Practice could be a parent category to Childhood Diseases, Accidents, Minor Illnesses, and Checkups. Each of these could have subcategories.

15. The physicians would like to add another step in the patient examination process. They want more complete records and the ability to handle cases where the diagnosis is not immediately available. Specifically, the physicians want to record the symptoms described by the patient at each visit. This record would also include the severity of the symptom and whether it was observed by the physician (or nurse). At each visit the patient's weight, blood pressure, and heart rate are also recorded (children are also measured for height). Along the same lines, they want to record any tests taken and the results of the tests. The tests can include simple physical tests such as reflexes as well as chemical tests. Define the tables and modify the class diagram to handle this additional data.

16. In 2006, Benjamin Brewer, M.D., a practicing physician, listed common statistics for a medical office ["A Doctor Faces Tough Decision to Stop Taking New Patients," The Wall Street Journal, February 7, 2006]. Read the article and use the numbers to estimate the size of the database after 1 year and 5 years of operation. Some basic data from the article: 3 physicians, 2,500 patients, 90 patients per week in office visits per physician. But, read the article to gain perspective on the situation. It is available in your library.

Web Site References

http://ibmdatamag.com/	IBM Data Magazine
http://www.for.gov.bc.ca/his/datadmin/	Canadian Ministry of Forests data administration site, with useful information on data administration and design. Start with the development standards.
http://support.microsoft.com/kb/100139/en-us	Introduction to normalization.
http://www.phpbuilder.com/columns/barry20000731.php3	Normalization examples.

Additional Reading

Date, C. J., *An Introduction to Database Systems*, 8th ed. Reading: Addison-Wesley, 2003. [A classic higher-level textbook that covers many details of normalization and databases.]

Fagin, R. "Multivalued dependencies and a new normal form for relational databases," *ACM Transactions on Database Systems*, 2 no. 3 (September 1977), pp. 262-278. [A classic paper in the development of normal forms.]

Fagin, R. "A Normal Form for Relational Databases That Is Based on Domains and Keys," *ACM Transactions on Database Systems*, 6 no. 3 (September 1981), pp 387-415. [The paper that initially described domain-key normal form.]

Kent, W. "A simple guide to five normal forms in relational database theory," *Communications of the ACM*, 26 no. 2 (February 1983), 120-125. [A nice presentation of normalization with examples.]

Rivero, L., J. Doorn, and V. Ferraggine, "Elicitation and Conversion of Hidden Objects and Restrictions in a Database Schema, *Proceedings of the 2002 ACM Symposium on Applied Computing*, 2002, 463-469. [Good discussion of referential integrity issues and problems with weak designs heavily dependent on surrogate ID columns.]

Wu, M.S., "The Practical Need for Fourth Normal Form," *Proceedings of the Twenty-third SIGCSE Technical Symposium on Computer Science Education*, 1992, 19-23. [A small study showing that fourth normal form violations are common in business applications.]

Appendix: Formal Definitions of Normalization

One of the strengths of the relational database model is that it was developed from the mathematical foundations of set theory. Although it is not necessary to know the formal definitions, sometimes they make it easier to understand the process. For a more detailed description of the normal forms and the complications, you should read C. J. Date's advanced textbook. Keep in mind that the formal definitions use specific terms. Figure 3.1A lists the major terms and their common interpretation. Although the formal terms are more accurate, few people have a common understanding of the terms, so in most conversations, it is easier to use the informal terms.

Initial Definitions

A relation is a set of attributes with data that changes over time. Each attribute has a corresponding domain and refers to some real-world characteristic. The formal definitions refer to subsets of attributes, which are collections of the columns. The data value returned within tuples for a specified subset of attributes X is denoted t[X].

The essence of normalization is to recognize that a set of attributes has some real-world relationships. The goal is to accurately portray these relationship constraints. These semantic constraints are known as *functional dependencies (FD)*.

Definition: Functional Dependency and Determinant

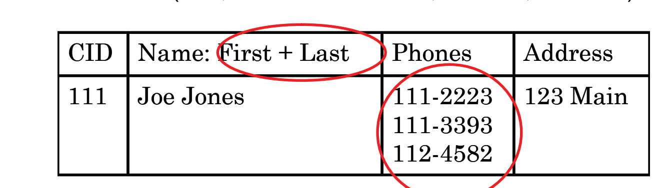
Where X and Y are subsets of attributes, a functional dependency is denoted as $X \rightarrow Y$, read as (X implies Y or X determines Y) and holds when any rows of data that have identical values for the X attributes always have identical values for the Y attributes. That is, for tuples t1 and t2 of R, if t1[X] = t2[X], then t1[Y] = t2[Y]. In an FD, X is also known as a determinant, because given the dependency, once you are given the values for the X attributes, it determines the resulting values for the Y attributes.

Figure 3.1A

Terminology. The formal terms are more accurate and defined mathematically, but difficult for developers and users to understand.

Formal	Definition	Informal
Relation	A set of attributes with data that changes over time. Often denoted R.	Table
Attribute	Characteristic with a real-world domain. Subsets of attributes are multiple columns, often denoted X or Y.	Column
Tuple	The data values returned for specific attribute sets are often denoted as t[X].	Row of data
Schema	Collection of tables and constraints and relationships.	
Functional dependency	$X \rightarrow Y$	Business rule dependency

Customer(CID, Name: First + Last, Phones, Address)



CID	Name: First + Last	Phones	Address
111	Joe Jones	111-2223 111-3393 112-4582	123 Main

Figure 3.2A

Nonatomic attributes. This table is not in first normal form because the Name attribute is a composite of two elementary attributes, and the phone attribute is being used to handle multiple values.

Definition: Keys

A key is a set of attributes K such that, where U is the set of all attributes in the relation,

1. There is a functional dependency K → U.
2. If K' is a subset of K, then there is no FD K' → U.

That is, a set of key attributes K functionally determines all other attributes in the relation, and it is the smallest set of attributes that will do so (there is no smaller subset of K that determines the other attributes).

Primary keys are important in relational databases because they are used to identify rows of data. Sometimes multiple attribute sets could be used to form different keys, so they are sometimes referred to as *candidate keys*.

Normal Form Definitions

The definition of first normal form is closely tied to the definition of an atomic attribute, so both need to be defined at the same time.

Definition: First Normal Form (1NF)

A relation is in first normal form if and only if all of its attributes are atomic.

Definition: Atomic Attributes

Atomic attributes are single valued, which means they cannot be composite, multi-valued, or nested relations.

Essentially, a 1NF relation is a table with simple cells under each attribute column. You are not allowed to play tricks and try to squeeze extra data, other relationships, or multiple columns into one column. Figure 3.2A provides an example of a table that is not in first normal form because it has two attributes that are not atomic.

Second normal form is defined in terms of primary keys and functional dependency.

Definition: Second Normal Form (2NF)

A relation is in second normal form if it is in first normal form and each nonkey attribute is fully functionally dependent on the primary key. That is, $K \rightarrow A_i$ for each nonkey attribute A_i. Consequently, there is no subset K' such that $K' \rightarrow A_i$ for any attribute.

OrderProduct(OrderID, ProductID, Quantity, Description)

OrderID	ProductID	Quantity	Description
32	15	1	Blue Hose
32	16	2	Pliers
33	15	1	Blue Hose

Figure 3.3A

Not full dependency. The product description depends on just the ProductID and not the full key {OrderID, ProductID}, so this relation is not in second normal form.

This definition corresponds closely to the simpler version presented in the chapter that each nonkey column depends on the entire key, not just a portion of the key. Figure 3.3A shows an example of a relation that is not in second normal form.

The formal definition of third normal form is a little harder to comprehend because it relies on a new concept: transitive dependency.

Definition: Transitive Dependency

Given functional dependencies $X \rightarrow Y$ and $Y \rightarrow Z$, the transitive dependency $X \rightarrow Z$ must also hold.

The concept of transitivity should be familiar from basic algebra. The fact that it holds true arises from the set-theory foundations. To understand the definition, remember that functional dependency represents business semantic relationships. Consider the relationship between OrderID, CustomerID, and customer Name attributes. The business rule that there is only one customer per order translates to a functional dependency OrderID → CustomerID. Once you know the OrderID value you always know the CustomerID value. Likewise, the key relationship between CustomerID and other attributes such as Name means there is a functional dependency CustomerID → Name. Applying transitivity, once you know the OrderID value, you can obtain the CustomerID value, and in turn learn the value of the customer Name.

Figure 3.4A

Transitive dependency. The customer Name and Phone attributes transitively depend on the CustomerID, so this relation is not in third normal form.

Order(OrderID, OrderDate, CustomerID, Name, Phone)

OrderID	OrderDate	CustomerID	Name	Phone
32	5/5/2004	1	Jones	222-3333
33	5/5/2004	2	Hong	444-8888
34	5/6/2004	1	Jones	222-3333

Employees can have many specialties, and many employees can be within a specialty. Employees can have many managers, but a manager can have only one specialty: Mgr → Specialty

EmpSpecMgr(EID, Specialty, ManagerID)

EID	Speciality	ManagerID
32	Drill	1
33	Weld	2
34	Drill	1

FD ManagerID → Specialty is not currently a key.

Figure 3.5A

Boyce-Codd normal form. Notice that there is a functional dependency from ManagerID to Specialty. Because this FD is not a candidate key in the relation, it is hidden, and this relation is not in BCNF.

Definition: Third Normal Form (3NF)

A relation is in third normal form if and only if it is in second normal form and no nonkey attributes are transitively dependent on the primary key. That is, given second normal form: $K \rightarrow A_i$ for each attribute A_i, there is no subset of attributes X such that $K \rightarrow X \rightarrow A_i$.

In simpler terms, each non-key attribute depends on the entire key (K), and not on some intermediate attribute (X). Figure 3.4A shows a common business example of a relation that is not in third normal form, because customer attributes depend transitively on the CustomerID.

As discussed in Chapter 3, Boyce-Codd normal form is a little harder to follow. It represents the same basic issue: removing a hidden dependency as seen by the formal definition.

Definition: Boyce-Codd Normal Form (BCNF)

A relation is in Boyce-Codd normal form if and only if it is in third normal form and every determinant is a candidate key. That is, if there is an FD $X \rightarrow Y$, then X must be the primary key (or equivalent to the primary key). In simpler terms: there cannot be a hidden dependency, where hidden means it is not part of the primary key.

As shown in the example in Figure 3.5A, consider the situation where employees can have many specialties, there are many employees for each specialty, and an employee can have many managers, but each manager is manager for only one specialty. This functional dependency (MangerID → Specialty) is not a key within the relation EmpSpecMgr(EID, Specialty, ManagerID), so the relation is not in BCNF. It has to be decomposed to create new relations ManagerSpecialty(ManagerID, Specialty), and EmployeeManager(EmployeeID, ManagerID) that explicitly have each functional dependency as keys.

Fourth normal form is slightly tricky but easy to apply once you understand it. The definition is closely tied to the definition of a multi-valued dependency.

Definition: Multi-Valued Dependency (MVD)

A multi-valued dependency (MVD) exists when there are at least three attributes in a relation (A, B, and C; which could be sets of attributes), and one attribute A determines the other two (B and C), but the two dependencies are independent of each other. That is, A → B and A → C, but B and C are not functionally dependent on each other.

For example, employees can have many specialties and be assigned many tools, but tools and specialties are not directly related to each other.

Definition: Fourth Normal Form (4NF)

A relation is in fourth normal form if and only if it is in Boyce-Codd normal form and there are no multi-valued dependencies. That is, all attributes of the relation are functionally dependent on A.

In the multi-valued dependency example for employee specialties and tools, the relation EmpSpecTools(EID, Specialty, ToolID) is not in fourth normal form, because of the two functional dependencies: EID → Specialty; and EID → ToolID. Solving the problem results in two simpler relations: EmployeeSpecialty(EID, Specialty) and EmployeeTools(EID, ToolID).