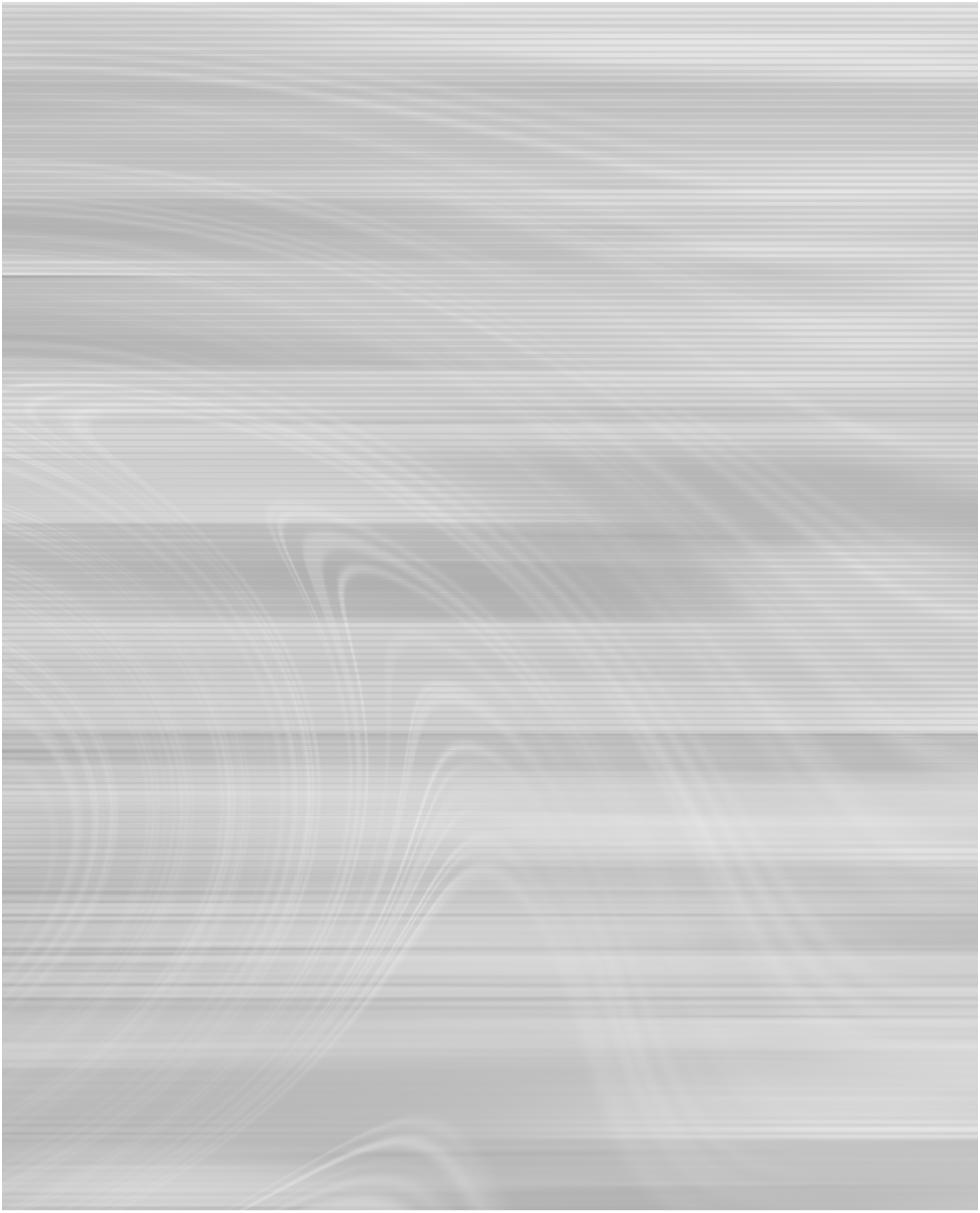
PART 8

Articles



ARTICLE 1

Designing Your Database Application

You don't have to go deeply into application and database design theory to build a solid foundation for your database project. You'll read about the fundamentals of application design in the next section of this article, and then you'll apply those fundamentals in the succeeding sections, "An Application Design Strategy" and "Data Analysis." The "Database Design Concepts" section teaches you a basic method for designing the tables you'll need for your application and for defining relationships between those tables.

Application Design Fundamentals

Methodologies for good computer application design were first devised in the 1960s by recognized industry consultants such as James Martin, Edward Yourdon, and Larry Constantine. At the dawn of modern computing, building an application or fixing a broken one was so expensive that the experts often advised spending 60 percent or more of the total project time getting the design right before writing a single line of code.

Today's application development technologies make building an application much cheaper and quicker. In fact, the pace of computing is several orders of magnitude faster than it was just a decade ago. An experienced user can sit down with Microsoft Office Access 2007 on a computer and build in an afternoon what took months to create on an early mainframe system (if it was even possible).

Today's technologies also give you the power to build very complex applications. But even with powerful tools, creating a database application (particularly a moderately complex one) without first spending some time determining what the application should do and how it should operate invites a lot of expensive time reworking the application. Even though it's easier than ever to go back and fix mistakes or to redesign "on the fly," if your application design is not well thought out it will be expensive and time-consuming later to track down any problems or to add new functionality.

The following is a brief overview of the typical steps involved in building a database application.

Step 1: Identifying Tasks

Before you start building an application, you should have some idea of what you want it to do. It is well worth your time to make a comprehensive list of all the major tasks you want to accomplish with the application—including those that you might not need right away but might want to implement in the future. By major tasks, we mean application functions that will ultimately be represented in a form or a report in your Access database. For example, "Enter customer orders" is a major task that you would accomplish by using a form created for that purpose, while "Calculate extended price" is most likely a subtask of "Enter customer orders" that you would accomplish by using the same form.

Step 2: Charting Task Flow

To be sure your application operates smoothly and logically, you should group the major tasks by topic and then order those tasks within groups on the basis of the sequence in which the tasks must be performed. For example, you should separate employee-related tasks and sales-related tasks into two topic groups. Within sales, an order must be entered into the system before you can print the order or examine commission totals.

You might discover that some tasks are related to more than one group or that completing a task in one group is a prerequisite to performing a task in another group. Grouping and charting the flow of tasks helps you discover a natural flow that you can ultimately reflect in the way you link the forms and reports in your finished application. Later in this article, you'll see how we laid out the tasks performed in one of the sample applications included with this book.

INSIDE OUT

Understanding the Work Process

When you're designing an application for someone else, these first two steps are absolutely the most important. Learning the work process of the business is critical to building an application that works correctly for the user. These first two steps help you understand how the business is run. Remember, your application is trying to make life easier for the users by automating some critical process that they're doing some other way.

If you do a lot of work for small businesses or small departments within larger businesses, walking the user through this process often helps them understand their own business, and often leads to new efficiencies even before you start to write a line of code!

Step 3: Identifying Data Elements

After you develop your task list, perhaps the most important design step is to list the bits of data—the data elements—required by each task and the changes that will be made to that data. A given task will require some input data (for example, a price to calculate an extended amount owed on an order); the task might also update the data. The task might delete some data elements (remove invoices paid, for example) or add new ones (insert new order details). Or the task might calculate some data and display it, but not save the data anywhere in the database.

Step 4: Organizing the Data

After you determine all the data elements you need for your application, you must organize the data elements by subject and then map the subjects into tables in your database. A subject is a person, place, thing, or action that you need to track in your application. Each subject normally requires several data elements—individual fields such as name or address—to fully define the subject. With a relational database system such as Access, you use a process called *normalization* to help you design the most efficient and most flexible way to store the data.

See "Database Design Concepts" on page A16 for a simple method of creating a normalized design.

Step 5: Designing a Prototype and a User Interface

After you build the table structures needed to support your application, you can easily mock up the application flow in forms and tie the forms together using simple macros or Microsoft Visual Basic event procedures. You can build the actual forms and reports for your application "on screen," switching to Form view, Layout view, or Print Preview periodically to check your progress. If you're building the application to be used by someone else, you can easily demonstrate and get approval for the "look and feel" of your application before you spend time writing the complex code that's needed to actually accomplish the tasks. (Parts 3 and 7 of this book show you how to design and construct forms and reports for desktop applications and client/server (project) applications, respectively; Part 4 shows you how to use Visual Basic to link forms and reports to build an application.)

Step 6: Constructing the Application

For very simple applications, you might find that the prototype *is* the application. Most applications, however, will require that you write code to fully automate all the tasks you identified in your design. You'll probably also need to create certain navigation forms that facilitate moving from one task to another. For example, you might need to construct forms that provide the road map to your application. You might also need to build forms to gather user input to allow users to easily filter the data they want to use in a particular task. You might also want to build custom Ribbons for most, if not all, of the forms in the application.

Step 7: Testing, Reviewing, and Refining

As you complete various components of your application, you should test each option that you provide. When you automate your application using Visual Basic, you'll have many debugging tools at your disposal to verify correct application execution and to identify and fix errors.

INSIDE OUT

Get Feedback from Your Users

If at all possible, you should provide completed portions of your application to users so that they can test your code and provide feedback about the flow of the application. Despite your best efforts to identify tasks and lay out a smooth task flow, users will invariably think of new and better ways to approach a particular task after they've seen your application in action. Also, users often discover that some features they asked you to include are not so useful after all. Discovering a required change early in the implementation stage can save you a lot of time reworking things later.

The refinement and revision process continues even after the application is put into use. Most software developers recognize that after they've finished one "release," they often must make design changes and build enhancements. For major revisions, you should start over at Step 1 to assess the overall impact of the desired changes so that you can smoothly integrate them into your earlier work.

Typical Application Development Steps

Step 1: Identifying tasks

Step 2: Charting task flow

Step 3: Identifying data elements

Step 4: Organizing the data

Step 5: Designing a prototype and a user interface

Step 6: Constructing the application

Step 7: Testing, reviewing, and refining

An Application Design Strategy

The two major schools of thought on designing databases are *process-driven design* (also known as *top-down design*), which focuses on the functions or tasks you need to perform, and *data-driven design* (also known as *bottom-up design*), which concentrates on identifying and organizing all the bits of data you need. The method we describe here incorporates some of the best ideas from both philosophies.

The method we like to use starts by identifying and grouping tasks to decide whether you need only one database or more than one database. (This is a top-down approach.) As explained previously, databases should be organized around a group of related tasks, or functions. For each task, you choose the individual elements of data you need. Next you gather all the data fields for all related tasks and begin organizing them into subjects. (This is a bottom-up approach.) Each subject forms the foundation for the individual tables in your database. Finally, you apply the rules you will learn in the "Database Design Concepts" section of this article to create your tables.

Note

The examples in the rest of this article are based on the Conrad Systems Contacts sample database application on the companion CD. In the chapters in this book, you can learn how to build various parts of the application as you explore the architecture and features of Office Access 2007. Conrad Systems Contacts is not only a contacts management application (Companies, People, Events, and Reminders) but also an order-entry application (Products, Sales, and Invoices). As such, it is considerably more complex than the Northwind Traders application that is included with Access 2007. It also employs many techniques not found in the product documentation.

Analyzing the Tasks

Let's assume that you've been hired by the owner of Conrad Systems to build a Contacts and Sales Tracking database. The database application must allow the owner to enter companies or organizations, the people in these companies, and the various types of contacts a user within Conrad Systems made while marketing several software products. If the contact results in a sale, the application should track the sale and print invoices.

The first design step you should perform is to list all the major tasks that this database application must implement. A partial list might include the following:

- Enter company/organization data
- Enter person data
- Link persons with companies/organizations

Oh No! Not Another Order-Entry Example!

You might have noticed that when you study database design—whether in a seminar, by reading a book, or by examining sample databases—nearly all the examples (including the one presented here) seem to be order-entry applications. There are several good reasons why you encounter this sort of example over and over again.

- A large percentage of business-oriented database applications use the common order-entry model. If you build a database, it's likely to use this model.
- Using the order-entry model makes it easy to demonstrate good database design techniques.
- At the core of the model, you'll find a *many-to-many* relationship example. (An order might be for many products, and any one product can appear in many orders.) Many-to-many relationships are common to most database applications yet often trip up even the most seasoned computer user.

You might argue, "Wait a minute, I'm building a hospital patient tracking system, not an order-entry system!" Or perhaps you're creating a database to reserve rooms in corporate housing for employees visiting from out of town. (The Housing Reservations sample database that is included with this book does this.) Aren't you "selling" hospital beds to patients? Isn't reserving a room for an employee "selling" that room? If you look at your business applications from this viewpoint, you'll be able to compare your project to the order-entry example with ease. Even if you're writing a personal application to keep track of your wine collection, you're "selling" a rack position in your cellar to your latest bottle purchase, and you're probably also tracking the "supplier" of your purchases.

The concept of data subjects related to each other in a *many-to-many* fashion is important in all but the simplest of database applications. This type of data relationship can be found in nearly all business or personal database applications. For example, a particular patient might need many different medications, and any one medication is administered to many patients. A movie in your home collection has many starring actors, and any one actor appears in many movies. As you'll discover, a well-designed order-entry database contains several many-to-many relationships.

- Indicate the primary contact person for a company and the primary company for a person
- Enter product information
- Perform a company search
- Perform a person search
- Log a contact event with a person
- Sell a product during a contact event
- Create an invoice for products ordered

- Print an invoice
- Log contact events after the sale

Figure A1-1 shows a blank application design worksheet that you should fill out for each task.

APPLICATION DESIGN WORKSHEET #1 = TASKS			
Task Name:			
Brief Description:			
Related Tasks:			
Data Name	Usage	Description	Subject

Figure A1-1 You can use an application design worksheet to help you describe tasks.

Note

You can find the Application Design Worksheet #1 in the Documents subfolder of the files you install from the companion CD, in the ArticleA1-01.doc file. Worksheet #2 is in the ArticleA1-02.doc file.

Consider the task of logging a new contact event (such as a letter received). For this task, the user might need to search for the person or the person's company. If the search is by company, then the user should be able to look at a list of people who are contacts for that company and select the specific person. The user should then be able to directly enter the details about the letter received and schedule a follow-up if necessary. In this particular application, Conrad Systems also wants to be able to log a sale as a contact event and be able to easily specify the product sold as part of entering the event. The program must also automatically create the related product sale record for the contact when this happens.

Note

Some of the terminology we are using here might be a bit confusing. A "contact" might be either a person (the person contacted) or an event (the telephone call or letter or what have you). Throughout this book, we use *contact* to refer to the person and *contact event* to refer to the action.

Data or Information?

You need to understand the difference between data and information before you start building your data design. This bit of knowledge makes it easier for you to determine what you need to store in your database.

Data is the set of static values you store in the tables of the database, while *information* is data that is retrieved and organized in a way that is meaningful to the person viewing it. You *store* data and you *retrieve* information. The distinction is important because of the way that you construct a database application. You first determine the tasks that are necessary (what *information* you need to be able to retrieve), and then you determine what must be stored in the database to support those tasks (what *data* you need in order to construct and supply that information).

Whenever you refer to or work with the structure of your database or the items stored in the tables, queries, macros, or code, you're dealing with data. Likewise, whenever you refer to or work with query records, filters, forms, or reports, you're dealing with information. The process of designing a database and its application becomes clearer once you understand this distinction. Unfortunately, these two terms are ones that folks in the computer industry have used interchangeably. But armed with this new knowledge, you're ready to tackle data design.

Selecting the Data

After you identify all the tasks, you must list the data items you need in order to perform each task. On the task worksheet, you enter a name for each data item, a usage code, and a brief description. In the Usage column, you enter one or more usage codes—I, O, U, D, and C—which stand for input, output, update, delete, and calculate. A data item is an *input* for a task if you need to read it from the database (but not update it) to perform the task. For example, a contact person name and address are some of the inputs needed to create a contact event. Likewise, data is an *output* for a task if it is new data that you enter as you perform the task or that the task calculates and stores based on the input data. For example, the payment due date of an invoice is an output; quantity sold and the selling price for a product in a new order are outputs as well.

You *update* data in a task if you read data from the database, change it, and write it back. A task such as recording a company's change of address would input the old address, update it, and write the new one back to the database. As you might guess, a task *deletes* data when it removes the data from the database. In the Contacts database, you might have a task to remove a product from the list of products owned by a contact person if that person decides to return the product. Finally, *calculated* data creates new values from input data to be displayed or printed but not written back to the database.

In the Subject column of the task worksheet, you enter the name of the subject to which you think each data element belongs. For example, an address might belong to a Contact. A completed application design worksheet for the Enter a Contact Event task might look like the one shown in Figure A1-2.

You might be wondering why we appear to have duplicate data here—ContactEventRequiresFollowUp and ContactFollowUp or ContactEventFollowUpDays and ContactFollowUpDate. The two ContactEvent data elements define the default actions that should occur for a particular type of event, and the two Contact fields are items that should be calculated by the application whenever the user chooses an event that requires a follow-up. The latter is something we call *point-in-time* data, which we discuss later in this article. You might not be able to spot this sort of subtle distinction as you first start to document your tasks, but you'll sort it out later in the design process as you finalize your table design following the rules we list in "Normalization Is the Solution."

Organizing Tasks

You should use task worksheets as a guide in laying out an initial structure for your application. Part of the planning you do on these worksheets is to consider usage—whether a piece of data might be needed as input, for updating, or as output of a given task.

Wherever you have something that is required as input, you should have a *precedent* task that creates that data item as output. For example, for the worksheet shown in Figure A1-2, you must gather company, contact, and product data before you can record a contact event. Similarly, you need to create the contact event type data in some other task before you can use that data in this task. Therefore, you should have a task for gathering basic company data, a task for entering basic contact person data, a task for creating product data, and a task for defining contact event types. It's useful to lay out all your defined tasks in a relationship diagram. You can see relationships among the tasks

in the Conrad Systems Contacts database in Figure A1-3. When one task is optionally precedent to another task, the two tasks are linked with dashed lines. For example, you do not have to define all products before you define simple contact event types. You can create an event for a contact (but you can't sell a product in that event) before you define the default company for a contact.

APPLICATION DESIGN WORKSHEET #1 = TASKS			
Task Name:	Enter a contact event		
Brief Description:	Search for contact person Add event to person		
Related Tasks:	Company add / edit, Contact person add / edit Contact event type add / edit, Contact product add / edit* Product add / edit		
Data Name	**Usage**	**Description**	**Subject**
ContactID	I, O	ID of the contact for the event	Contacts
ContactDateTime	O	Date and time of the contact event	ContactEvents
ContactEventTypeID	I, O	ID of the type of contact event	ContactEventTypes
ContactEventTypeDesc	I	Description of the contact type	ContactEventTypes
ContactEventRequires-FollowUp	I	Follow-up flag	ContactEventTypes
ContactEventFollowUp-Days	I	Default number of days in future for follow-up	ContactEventTypes
ContactEventProduct-Sold	I	Flag indicating a product sale event	ContactEventTypes
ContactEventProductID	I	Unique ID of the product sold	Products
ContactNotes	O	Notes about the contact event	ContactEvents
ContactFollowUp	O	Flag indicating follow-up required	ContactEvents
ContactFollowUpDate	O	Date the follow-up should occur	ContactEvents
*Additional items if ContactEventProduct-Sold is true.			
CompanyID	I, O	ID of the default company for this contact person	Companies
ProductID	I, O	ID of the product sold	Products
DateSold	O	Date the product was sold	ContactProducts
SoldPrice	O	Price charged for the product	ContactProducts

Figure A1-2 A completed worksheet for the Enter a Contact Event task might look like this.

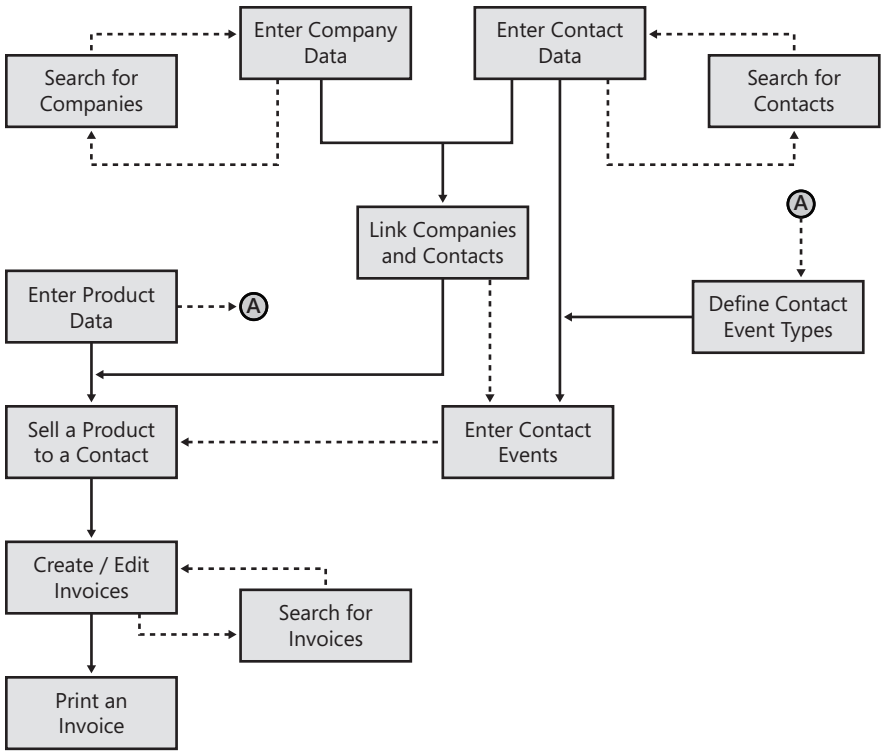


Figure A1-3 This diagram shows the relationships among tasks in the Conrad Systems Contacts database.

Data Analysis

Now you're ready to begin a more thorough analysis of your data and to organize the individual elements into data subjects. These subjects become candidates for tables in your database design.

Choosing the Database Subjects

If you've been careful in identifying the subject for each data item you need, the next step is very easy. You create another worksheet, similar to the worksheet shown in Figure A1-4, to help you collect all the data items that belong to each subject. In the top part of the worksheet, you list the related subjects that appear in any given task and indicate the kind of relationship.

APPLICATION DESIGN WORKSHEET #2 = SUBJECTS			
Subject Name:			
Brief Description:			
Related Subjects:	Name	Relationship	
Data Name	Data Type	Description	Validation Rule

Figure A1-4 This application design worksheet will help to identify related subjects.

If there are potentially many instances of the related subject for one instance of the current subject (for example, many contacts within a company), enter Many in the Relationship column. If there is potentially only one instance of the related subject to one instance of the current subject (for example, one and only one contact refers a company), enter One in the Relationship column. For details about relationship types, see "Efficient Relationships Are the Result" on page A27.

It's important to understand these relationships because they have a significant effect on the database structure and on how you work with two related subject tables in Access. If you take care in filling out and revising your worksheets, you can ultimately use each worksheet to create a table in Access. You'll learn more about these relationships later in this article.

You can see a completed worksheet for the Companies subject in Figure A1-5.

APPLICATION DESIGN WORKSHEET #2 = SUBJECTS			
Subject Name:	Companies		
Brief Description:	Information about companies / organizations to which contact persons are related.		
Related Subjects:	**Name**	**Relationship**	
	CompanyContacts	Many	
	Invoices	Many	
	Contacts	One (contact referring this Company)	
Data Name	**Data Type**	**Description**	**Validation Rule**
CompanyID	Autonumber	Company identifier	Required (P Key)
CompanyName	Text (50)	Name of the company or organization	Is Not Null
Department	Text (50)	Optional department name	
Address	Text (255)	Street address	
City	Text (50)	City	
County	Text (50)	County	
StateOrProvince	Text (20)	State or province	
PostalCode	Text (20)	Postal code	00000\-9999
Country	Text (50)	Country	
PhoneNumber	Text (30)	Phone	!\(999") "000\-0000
FaxNumber	Text (30)	Phone	!\(999") "000\-0000
WebSite	Hyperlink	Website address	
ReferredBy	Number, Long	Contact who referred this company / organization.	RI rule – child of Contacts

Figure A1-5 Here is a completed worksheet for the Companies subject.

As you copy each data item to the subject worksheet, you designate the data type (Text, Number, Currency, Memo, and so on) and the data length in the Data Type column. You can enter a short descriptive phrase for each data item in the Description column. When you create your table from the worksheet, the description is the default information that Access will display on the status bar at the bottom of the screen whenever the field is selected on a datasheet or in a form or a report.

Finally, in the Validation Rule column, you should make a note of any validation rules or input mask restrictions that always apply to the data field. Later, you can define these rules in Access, and Access will check each time you create new data to ensure that you haven't violated any of the rules. Validating data can be especially important when you create a database application for other people to use.

Mapping Subjects to Your Database

After you fill out all of the subject worksheets, each worksheet becomes a candidate to be a table in your database. For each table, you must confirm that all the data you need is included. You should also be sure that you don't include any unnecessary data.

For example, if any customers need more than one line for an address, you should consider adding a second data field. If you expect to have more than one type of product category (in Conrad Systems' case, they sell Single, Multi-User, and Remote versions of their software as well as support for each), you should create a separate worksheet for product categories that you'll use to define a table that contains records for each product type. In the next section, you'll learn how to use four simple rules to create a flexible and logical set of tables from your subject worksheets.

Database Design Concepts

When using a relational database system such as Access 2007, you should begin by designing each database around a specific set of tasks or functions. For example, you might design one database for customers and orders that contains data about each customer, the products available for sale, the orders for each customer, and the product sales history. You might have another database that handles human resources for your company. It would contain all relevant data about the employees and their dependents, such as names, job titles, employment histories, departmental assignments, insurance information, and the like.

INSIDE OUT

Review Your Work

If you have filled out the subject worksheets for your application before you start this process, it's a good idea to go back and make any necessary corrections to those worksheets as you follow the rules in this section to refine your table structure. At the end of the process, each subject worksheet should map to exactly one table.

At this point, you face your biggest design challenge: How do you organize data within each task-oriented database so that you take advantage of the relational capabilities of Access and avoid inefficiency and waste? If you followed the steps outlined earlier in this article for analyzing application tasks and identifying database subjects, you're well on your way to creating a logical, flexible, and usable database design. But what if you just dove in and started laying out your tables without first analyzing tasks and subjects? The rest of this article shows you how to apply some rules to transform a makeshift database design into one that is robust and efficient.

Waste Is the Problem

A *table* stores the data you need for the tasks you want to perform. A table is made up of columns, or *fields*, each of which contains a specific kind of data (such as a customer name or a credit rating), and rows, or *records*, that collect all the data about a particular person, place, or thing. You can see this organization in the Companies table in the Conrad Systems Contacts database, as shown in Figure A1-6.

tblCompanies

Company	Company / Organization	Departme	Billing Address	City	County	State/Provir	Postal
1	Contoso, Ltd		7890 Lincoln Avenue	Youngsville	Warren	PA	16371
2	Fourth Coffee		7890 3rd Street	Kalamazoo	Kalamazoo	MI	49004
3	Coho Vineyard		6789 4th Street	Grosse Point	Wayne	MI	48236
4	Southridge Video		1234 1st Avenue	Warriors Mai	Huntingdon	PA	16877
5	Consolidated Messenger		4567 Lincoln Drive	Yorktown	York	VA	23692
6	Baldwin Museum of Science		7890 Central Drive	Matoaka	Mercer	WV	24736
7	Fabrikam, Inc.		4567 2nd Street	Easton	Fresno	CA	93706
8	Alpine Ski House		3456 3rd Boulevard	Pingree Grov	Kane	IL	60140
9	School of Fine Art		5678 3rd Avenue	Elias	Jackson	KY	40486
10	City Power & Light		5678 Willow Avenue	Conway	Franklin	MA	01341
11	Blue Yonder Airlines		5678 Central Avenue	Riverside	Cass	ND	58078
12	Margie's Travel		4567 3rd Avenue	Grandy	Currituck	NC	27939
13	Lucerne Publishing		6789 1st Parkway	Irrigon	Morrow	OR	97844
14	Humongous Insurance		6789 Willow Parkway	Flushing	Queens	NY	11351
15	Viescas Consulting, Inc.	Managemer	379 Amherst St PMB 2	Nashua	Hillsborough	NH	03063
16	Conrad Systems Development	Managemer		Bend	Deschutes	OR	97701
*	{New}						

Record: 1 of 16 No Filter Search

Figure A1-6 The Companies table in Datasheet view is an example of how data is organized in a table.

> **Note**
>
> The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

For the purposes of this design exercise, let's say you want to build a new database (named Contacts) for tracking contacts, contact events, and products sold during contact events without the benefit of first analyzing the tasks and subjects you'll need. You might be tempted to put all the data about the task you want to do—keeping track

of customers and their contacts with you and the products they might buy during a contact—in a single Contact Events table, whose fields are represented in Figure A1-7.

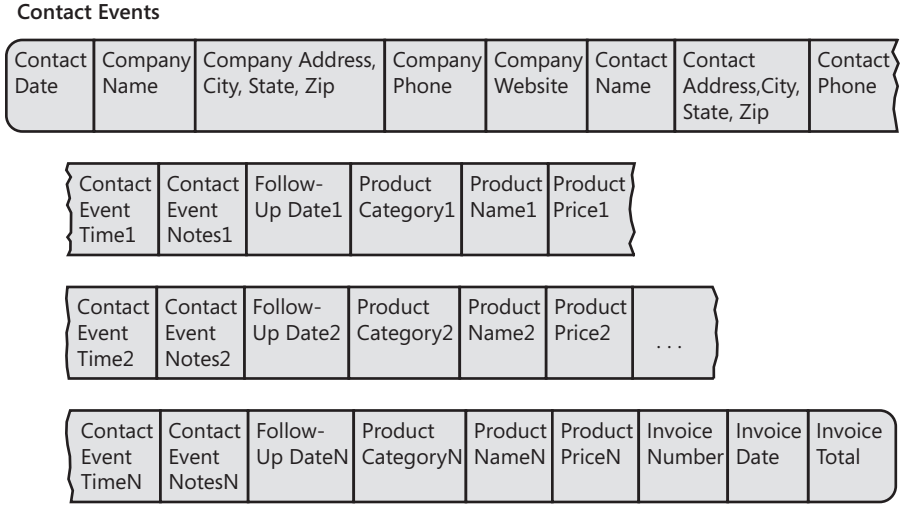


Figure A1-7 This design for the Contacts database uses a single Contact Events table.

There are many problems with this technique. For example:

- Every day that a contact calls you, you have to duplicate the Company Name, Company Address, Contact Name, and Contact Address fields in another record for the new contact event. Repeatedly storing the same name and address in your database wastes a lot of space—and you can easily make mistakes if you have to enter basic information about a contact more than once.
- You have no way of predicting how many contact events you'll have in a given day or how many products might be ordered. If you keep track of each day's contact events in a single record, you have to guess the largest number of individual events and products and leave space for Event Time 1, Event Time 2, Event Time 3, Product Name 1, Product Name 2, and so on, all the way to the maximum number. Again, you're wasting valuable space in your database. If you guess wrong, you'll have to change your design just to accommodate a day when a contact calls you (or you call them) more times than you have allocated in your record. And later, if you want to find out what products were sold to which contacts, you'll have to search each Product Name field in every record.
- You have to waste space in the database storing data that can easily be calculated when it's time to print a report. For example, you'll certainly want to calculate the total invoice amount, but you do not need to keep the result in a field.
- Designing one complex field to contain all the parts of simple data items (for example, lumping together Street Address, City, State, and Postal Code) makes it difficult to search or sort on part of the data. In this example, it would be

impossible to sort on company or contact postal code because that piece of information might appear anywhere within the more complex single address fields.

Normalization Is the Solution

You can minimize the kinds of problems just noted (although it might not always be desirable to eliminate all duplicate values), by using a process called *normalization* to organize data fields into a group of tables. The mathematical theory behind normalization is rigorous and complex, but the tests you can apply to determine whether you have a design that makes sense and that is easy to use are quite simple—and can be stated as rules.

Field Uniqueness

Because wasted space is one of the biggest problems with an unnormalized table design, it makes sense to remove redundant fields from a table. So the first rule is about field uniqueness.

Rule 1: Each field in a table should represent a unique type of information. This means that you should break up complex compound fields and get rid of the repeating groups of information. In this example, you should separate the complex address fields into simple fields and new tables designed to eliminate the repeating contact event and product information. When you create separate tables for the repeating data, you include some "key" information from the main table to create a link between the new tables and the original one. One possible result might look like the diagram in Figure A1-8.

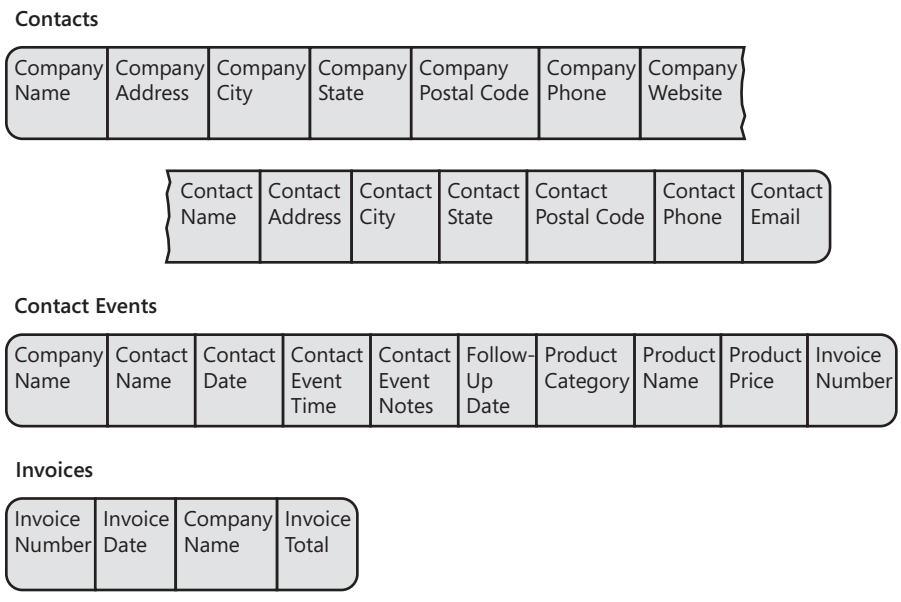


Figure A1-8 This design for the Contacts database eliminates redundant fields.

These tables are much simpler because you can store one record per contact event. Also, you don't have to reserve room in your records to hold a large number of events per day per contact. All the lengthy address information is now in a separate table so that you don't have to repeat it for each event. Because an invoice might cover multiple products purchased, there's also a separate table for that. Notice that the Contact Events table includes certain key information to link it to the Contacts table (Company Name and Contact Name) and to the Invoices table (Invoice Number).

Searching and sorting the information will now also be easier. You can sort the Contacts records on postal code or do a search on the separate city and state fields. Can you spot a field that we failed to break up into separate elements? If your answer is the Contact Name field, you're correct! As you'll see in the final solution, we need to break this field into at least separate First Name and Last Name fields.

The duplicate data problem is now somewhat worse because you are repeating the Company Name and Contact Name fields in each Contact Events record. This duplicate data is necessary, however, to maintain the links between the tables. The potentially long Product Name field is also redundant in the Contact Events table—when you sell the same product more than once, the Product Name will appear in multiple rows. (Maybe Products should have a separate table?) What happens if you misspell a product name in one of the rows? Will you be able to find all contacts who bought the same product? You can solve this problem by following the second rule.

Primary Keys

In a good relational database design, each record in any table must be uniquely identified. That is, some field (or combination of fields) in the table must yield a unique value for each record in the table. This unique identifier is called the *primary key*.

Rule 2: Each table must have a unique identifier, or primary key, that is made up of one or more fields in the table. Whenever possible, you should use the simplest data that naturally provides unique values. You should always be able to find a field or some combination of fields whose values are unique across all rows. (In relational design terminology, these are called *candidate keys*.) You should consider the simplest combination of fields as the best candidate to be your primary key. However, in the case of the Contacts table as currently designed in Figure A1-8, you would probably need a combination of Company Name, Contact Name, and perhaps one of the contact address or city fields to guarantee uniqueness. When this happens, it is preferable to generate an artificial unique ID field to use as the primary key (Contact ID). However, you might want to add code in your final application that checks for a potential duplicate name (another record previously saved that has the same name as the new record about to be saved) and warns the user before inserting a new unique record. Access provides a handy data type called *AutoNumber* to make it easy to create a unique ID field like Contact ID. You can learn more about the AutoNumber data type in Chapter 4, "Creating Your Database and Tables."

After we assign Contact ID as the primary key of the Contacts table, it becomes much easier to link a contact with a contact event by substituting Contact ID for the Company Name and Contact Name fields in the Contact Events table. Although Contact ID in the

Contact Events table perhaps looks like duplicate information, it's really the link that you can use to associate or *relate* the rows from the two tables. Relational databases are equipped to support this design technique by giving you powerful tools to bring related information back together easily. You can learn more about these tools in Chapter 8, "Creating and Working with Simple Queries."

A common mistake would be to create another ID field to uniquely identify the rows in Contact Events. Now with the addition of Contact ID, it's easy to see that the combination of Contact ID, Contact Date, and Contact Event Time are most likely unique to each row, so we should use the this combination of elements as a natural primary key. For the new Invoices table, the choice is simple. Invoice Number might be an AutoNumber ID field, but it is probably a unique number entered by the user when creating a new invoice record. Some companies like to use a year prefix combined with a unique sequence number within the year as an invoice number. The Invoice Number is still in the Contact Events table to identify which products were billed on what invoice number. You can see the result of adding primary keys in Figure A1-9.

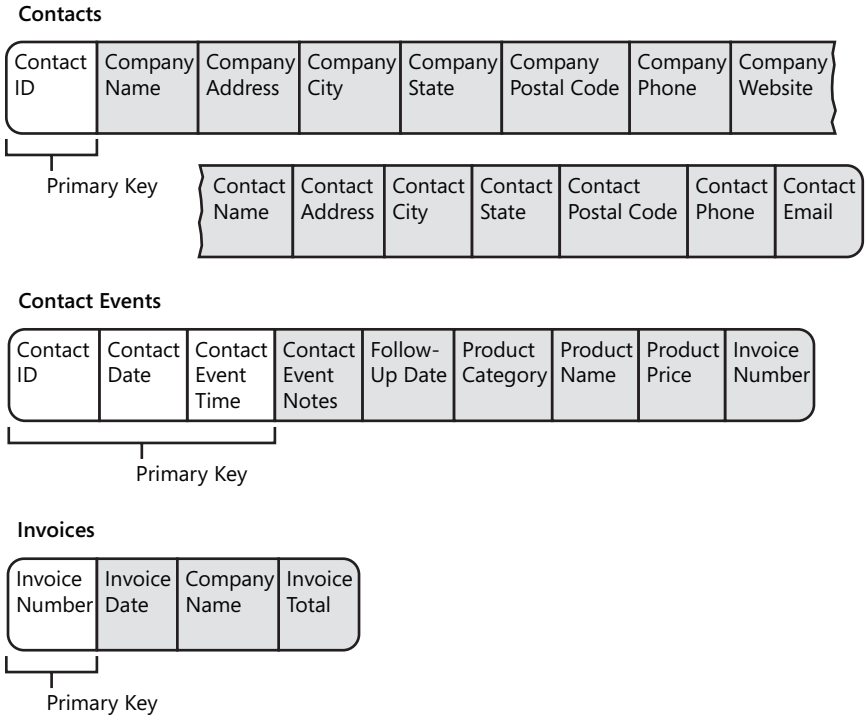


Figure A1-9 The Conrad Systems Contacts database tables now have primary keys defined.

Functional Dependence

Defining a primary key helps you better identify the true subject of the table. Now, you can check to see whether you included all the data relevant to the subject of the table

and whether each of the fields in the table describes an attribute of the subject (and not some other subject). In relational terminology, you should check to see whether each field is *functionally dependent* on the primary key that defines the subject of the table.

Rule 3: For each unique primary key value, the values in the data columns must be relevant to, and must completely describe, the subject of the table. This rule works in two ways. First, you shouldn't have any data in a table that is not relevant to the subject (as defined by the primary key) of the table. Second, the data in the table should completely describe the subject.

Let's start by looking at the Contacts table as defined in Figure A1-9. The subject of this table is the people who are our contacts. We certainly need to know the company or organization with which a person is associated. What if a person has more than one such association? For example, a person might work for a company but also be a member of one or more professional organizations. We certainly do not want to repeat the contact name and personal address information multiple times for each different association. Is the company information in a contact row unique to the individual defined by that row? Probably not. Even if we're certain that a person is associated with only one company or organization, we'll have to duplicate the company information in multiple rows when a company has more than one person associated with it.

The solution is to identify companies (organizations) as a separate subject with its own unique identifier. If a person is related to one and only one company, we can place a linking copy of the Company ID in the Contacts table. In this case, let's assume that a person can be related to more than one company or organization. A company has many persons, and a person might belong to many companies or organizations. In relational terminology, this is called a *many-to-many relationship*, which you can read more about later in this article. To define this in our table design, we need a *linking table* that stores the multiple relationships of the companies and people—a table called Company Contacts. While we're at it, let's refine the Contact Name field by splitting it into separate First Name and Last Name fields (so we can sort and search by just the last name), and let's complete the Company Contacts table by adding an indicator field that defines which company is the primary one for the contact.

Now, we should turn our attention to the Contact Events table. In the table shown in Figure A1-9, we have not only information about the event but also information about a product that might be sold during the event. In fact, the user of this database might make many calls or mail out many brochures or letters before actually selling a product. The product information isn't fully *functionally dependent* on the subject of this table, so it needs to be in a separate subject table. In fact, a product is not going to be purchased by an individual contact—it will be bought by the contact's primary company or organization.

So, we also need to create a separate Contact Products table to store the products a contact might purchase after dozens of contacts. This table should have all the information relevant to a company purchasing a product for an employee, but nothing extra. This

moves the extra product information from the old Contact Events table and makes the fields in that table relevant only to events and nothing else.

Finally, we should completely define the Invoices subject by adding other relevant information such as the purchasing company's purchase order number, the date the invoice payment is due, and an indicator field to mark when the invoice is paid. You can see the result of applying the rule in this step in Figure A1-10.

Companies

Company ID	Company Name	Company Address	Company City	Company State	Company Postal Code	Company Phone	Company Website

Company Contacts

Company ID	Contact ID	Position	Primary for Contact

Contacts

Contact ID	Contact Last Name	Contact First Name	Contact Address	Contact City	Contact State	Contact Postal Code	Contact Phone	Contact Email

Contact Events

Contact ID	Contact Date	Contact Event Time	Contact Event Notes	Follow-Up Date

Contact Products

Company ID	Contact ID	Product Name	Product Category	Date Sold	Product Price	Invoice Number

Invoices

Invoice Number	Invoice Date	Company ID	PO Number	Invoice Due	Invoice Paid	Invoice Total

Figure A1-10 Creating additional subject tables in the Conrad Systems Contacts database ensures that all fields in a table are functionally dependent on the primary key of the table.

Field Independence

The last rule checks to see whether you'll have any problems when you make changes to the data in your tables.

Rule 4: You must be able to make a change to the data in any field (other than to a field in the primary key) without affecting the data in any other field. Take a look again at the Contact Products table in Figure A1-10. As we applied the second and third rules, we left product information with the Contact Products information because it seems reasonable that you need to know about the product sold to a contact. Note that if you need to correct the spelling of a product name, you can do so without affecting any other fields in that record. If you misspelled the same product name for many contact products, however, you might have to change many records. Also, if you entered the wrong product (for example, an order is actually for a Single-User edition, not a Multi-User edition), you can't change the product name without also changing that record's category and pricing information.

The Product Category, Product Name, and Product Price fields are not independent of one another. In fact, Product Category and Product Price are functionally dependent on Product Name. (See Rule 3.) Although it wasn't obvious at first, Product Name describes another subject that is different from the subject of contact products. You can see how carefully applying this fourth rule helps you identify changes that you perhaps should have made when applying earlier rules. This situation calls for another table in your design: a separate Products table, as shown in Figure A1-11.

Now, if you misspell a product name, you can simply change the product name in the Products table. Also, instead of using the Product Name field (which might be 40 or 50 characters long) as the primary key for the Products table, you can create a shorter Product ID field (perhaps a five-digit number) to minimize the size of the relational data you need in the Contact Products table.

Note also that we removed the Invoice Total field from the Invoices table because any change to a price in Contact Products would make this value incorrect. The database is not going to maintain this calculated value for you, so you would have to write extra code in your application to recalculate and update the value each time a contact ordered another product. It's a simple matter to build a query to sum the product prices for the records related to an invoice to calculate the total owed. (See Chapter 7, "Creating and Working with simple Queries," and Chapter 8 for details.) You can also calculate the total invoice value when the invoice is complete—perhaps as part of the report that prints the invoice.

An alternative (but less rigorous) way to check for field independence is to see whether you have the same data repeated in your records. In the previous design, whenever you created a sale for a particular product during a contact event, you had to enter the product's name, category, and price in the record. With a separate Products table, if you need to correct a product name spelling or change a list price or product category, you have to make the change only in one field of one record in the Products table. If you entered the wrong product in a contact product record, you have to change only the Product ID to fix the problem.

Companies

Company ID	Company Name	Company Address	Company City	Company State	Company Postal Code	Company Phone	Company Website

Company Contacts

Company ID	Contact ID	Position	Primary for Contact

Contacts

Contact ID	Contact Last Name	Contact First Name	Contact Address	Contact City	Contact State	Contact Postal Code	Contact Phone	Contact Email

Contact Events

Contact ID	Contact Date	Contact Event Time	Contact Event Notes	Follow-Up Date

Contact Products

Company ID	Contact ID	Product ID	Date Sold	Product Sold Price	Invoice Number

Products

Product ID	Product Category	Product Name	Product Price

Invoices

Invoice Number	Invoice Date	Company ID	PO Number	Invoice Due	Invoice Paid

Figure A1-11 This design for the Conrad Systems Contacts database follows all the design rules.

Note that we added a new field, a separate Product Sold Price in the Contact Products table. Why not link to the new Products table to find out the price? Why isn't this duplicate data that violates Rule 1? This is an example of why it is very important to understand how the business runs. In this case, Conrad Systems sometimes offers a discount

off "list price" to a company that purchases multiple copies of Conrad Systems' products. The price in the Contact Products table is the actual sales price that the user enters when the company buys the product. You can learn more about the concept of such point-in-time data later in this article.

The actual Conrad Systems Contacts sample database includes 10 tables, which are all shown in the Relationships window in Figure A1-12. Notice that we created additional fields in each table to fully describe the subject of each table and we added other tables to support some of the other tasks identified earlier in this section. For example, many fields were added to both the Companies and Contacts tables to fully capture all the pertinent information about those subjects. There are also three *lookup tables* to help ensure accurate data entry and to provide additional information about the nature of some classification codes.

A lookup table helps you restrict the list of values that are valid for a field in a main table, and they might also contain additional fields that help further define the meaning of each value in the list. You can learn more about defining lookup properties in Chapter 4.

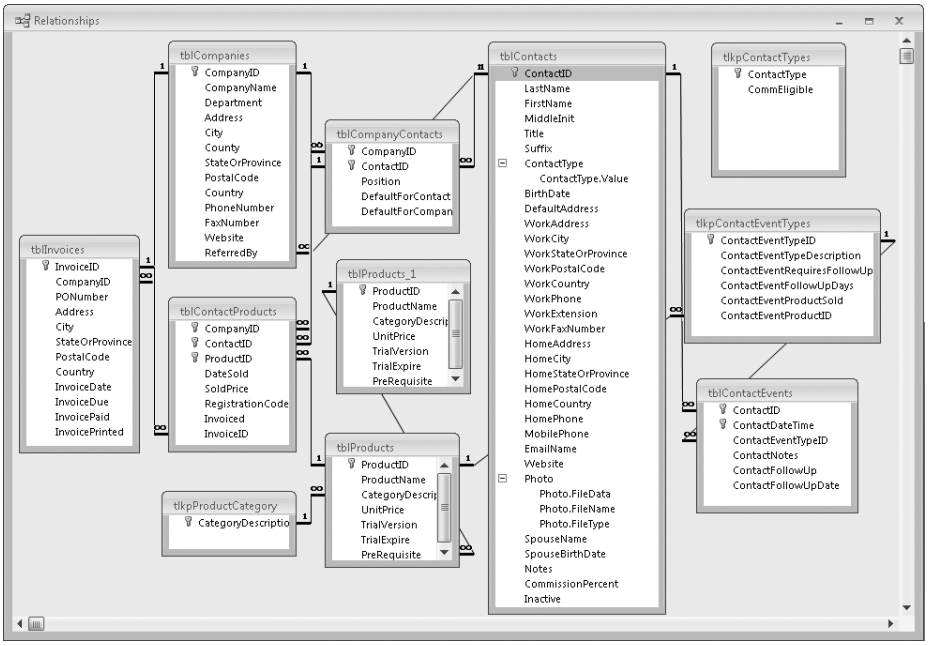


Figure A1-12 The tables in the Conrad Systems Contacts sample database are shown in the Relationships window.

The Four Rules of Good Table Design

Rule 1: Each field in a table should represent a unique type of information.

Rule 2: Each table must have a unique identifier, or primary key, that is made up of one or more fields in the table.

Rule 3: For each unique primary key value, the values in the data columns must be relevant to, and must completely describe, the subject of the table.

Rule 4: You must be able to make a change to the data in any field (other than to a field in the primary key) without affecting the data in any other field.

Efficient Relationships Are the Result

When you apply good design techniques, you end up with a database that efficiently links your data. You probably noticed that when you normalize your data as recommended, you tend to get many separate tables. Before relational databases were invented, you had to either compromise your design or manually keep track of the relationships between files or tables. For example, you had to put company data in your contacts and invoices tables or write your program to first open and read a record from one table and then search for the matching record in the related table. Relational databases solve these problems. With a good design you don't have to worry about how to bring the data together when you need it.

Foreign Keys

You might have noticed as you followed the Conrad Systems Contacts design example that each time we created a new table, we left behind in the other table a field that could link the two, such as the Company ID, Contact ID, and Product ID fields in the Contact Products table. The Invoice Number field in Contact Products is also a link to the Invoices table. These "linking" fields are called *foreign keys*.

In a well-designed database, foreign keys result in efficiency. You keep track of related foreign keys as you lay out your database design. When you define your tables in Access, you link primary keys to foreign keys to tell Access how to join the data when you need to retrieve information from more than one table. You can also ask Access to maintain the integrity of your table relationships—for example, Access will ensure that you don't create a contact event for a contact that doesn't exist. When you ask Access to maintain this *referential integrity*, Access automatically creates indexes for you. Indexes help Access find data more quickly when you're searching, filtering, or linking data.

For details about referential integrity and defining indexes, see Chapter 4.

One-to-Many and One-to-One Relationships

In most cases, the relationship between any two tables is one-to-many. That is, for any one record in the first table, there are many related records in the second table; but for any record in the second table, there is exactly one matching record in the first table. You can see several instances of this type of relationship in the design of the Conrad Systems Contacts database. For example, each company might have several invoices, but a single invoice record applies to only one company.

Occasionally, you might want to break down a table further because you use some of the data in the table infrequently or because some of the data in the table is highly sensitive and should not be available to everyone. For example, you might want to keep track of certain company data for marketing purposes, but you don't need access to that data all the time. Or you might have data about credit ratings that should be accessible only to authorized people. In either case, you can create a separate table that also has a primary key of Company ID. The relationship between the original Companies table and the Company Info or Company Credit table is one-to-one. That is, for each record in the first table, there is exactly one record in the second table.

Creating Table Links

The last step in designing your database is to create the links between your tables. For each subject, identify those for which you wrote *Many* under Relationship on the worksheet. Be sure that the corresponding relationship for the other table is *One*. If you see *Many* in both places, you must create a separate *linking table* to handle the relationship. (Access won't let you define a many-to-many relationship directly between two tables.) In the example of the Add/Edit a Contact task, a contact might be associated with many companies or organizations, and a company most likely has many contacts. The Company Contacts table in the Conrad Systems Contacts database is a linking table that clears up this many-to-many relationship between companies and contacts. Contact Products is another table that works as an intersection table because it has a one-to-many relationship with both Contacts and Products. (A contact might purchase several products, and a product is most likely owned by many contacts.)

After you straighten out the many-to-many relationships and create additional subject worksheets to reflect the linking tables, you need to create the links between subjects. To complete the links, you should place a copy of the primary key from the *one* subject into a field in the *many* subject. For example, by looking at the worksheet for Companies shown in Figure A1-5, you can surmise that the primary key for the Companies subject, Company ID, also needs to be a field in the Company Contacts and Invoices subjects.

When to Break the Rules

As a starting point, for every application that you build, you should always analyze the tasks you need to perform, decide on the data required to support those tasks, and create a well-designed (also known as *normalized*) database table structure. After you have

a design that follows all the rules, you might discover changes that you need to make either to follow specific business rules or to make your application more responsive to the needs of your users. In every case for which you decide to "break the rules," you should know the specific reason for doing so, document your actions, and be prepared to add procedures to your application to manage the impact of those changes. The following sections discuss some of the reasons why you might need to break the rules.

Improving Performance of Critical Tasks

The majority of cases for breaking the rules involve manipulating the design to achieve better performance for certain critical tasks. For example, although modern relational database systems (like Access) do a good job of linking many related tables to perform complex tasks, you might encounter situations in which the performance of a multiple-table link (also called a *joined query*—see Chapter 8 for details) is not fast enough. Sometimes if you *denormalize* selected portions of the design, you can achieve the required performance. For example, instead of building a separate table of product category codes that requires a link, you might place the category descriptions directly in the products table. If you choose to do this, you will need to add procedures to the forms you provide to enter these categories to make sure that any similar descriptions aren't duplicate entries. We chose to do this in the Conrad Systems Contacts database, and we solved the problem by using a combo box that allows the user to choose a value only from a validated list in another table. You'll learn more about working with combo box controls in Chapter 12, "Building a Form."

Another case for breaking the rules is the selective inclusion of calculated values in your database. For example, if a critical management report needs the calculated totals for all orders in a month, but the data is retrieved too slowly when calculating the detailed values for thousands of product purchase records per order and thousands of orders, you might want to add a field for order total in the Orders or Invoices table. Of course, this also means adding procedures to your order-entry forms to ensure that any change in an order detail record is reflected immediately in the calculated order total. Your application will spend a few extra fractions of a second processing each order so that month-end totals can be obtained quickly.

Capturing Point-in-Time Data

Sometimes you need to break the rules to follow known business rules. In the previous design exercise, we considered removing the Price field from the Contact Products table because it duplicated the price information in the Products table. However, if your business rules say that the price of a product can change over time, you might need to include the price in your order details to record the price at the *point in time* that the order was placed. If your business rules dictate this sort of change, you should add procedures to your application to automatically copy the "current" price to any new order detail row.

You can see another case in the Conrad Systems Contacts database. Some of the billing address information in the Invoices table looks like it duplicates information in the Companies table. If you examine the way the database works, you'll find some code

that copies the company information to the invoice information when you create a new invoice. Again, this address information in the Invoice is point-in-time data. It is the address that was current at the time the invoice was created. The company address might change later, but we will always know where we mailed a particular invoice.

Note

You can find the Housing Reservations sample application on the companion CD.

There's yet another example in the Housing Reservations database. In this database, the user creates room reservation requests that indicate an employee needs a specific type of room over a range of dates. Some of this request information gets copied to the actual reservation record at the time the housing manager confirms the reservation. It is also company policy to honor the quoted rate at the time the reservation is made so that the manager who approves the reservation knows exactly what will be charged. (Likewise, if this were a commercial hotel, you would expect to pay the rate quoted at the time of the reservation, not the current rate at the time you check in three months later!) If you look at the database design for the Housing Reservations database, shown in Figure A1-13, you'll see what looks like duplicate information in the Reservation Requests and the Reservations tables. In this case, check-in and check-out information is copied from Reservation Requests to Reservations when a reservation is confirmed. Likewise, the daily and weekly rates that are current at the time the reservation is made are copied to the reservation by code in the application.

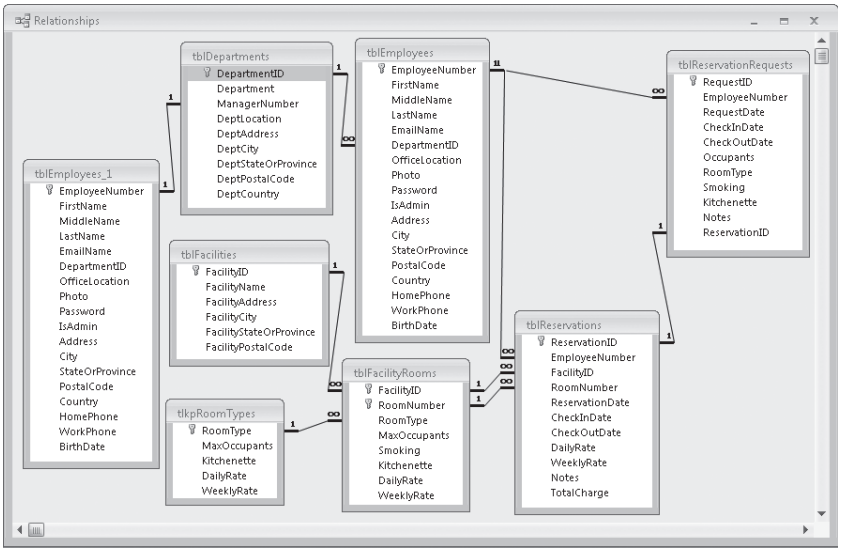


Figure A1-13 The design for the Housing Reservations database includes duplicate point-in-time pricing information in the Reservations table.

Note also that there's a Total Charge field in the record that must be calculated by code within the application. The application spends a little computing time for each change to the records in the table to save processing time in reports that might need to work with hundreds of rows. If you look behind the Reservations form in the Housing Reservations database, you'll find lots of code to accomplish both the rate copy and the total calculation.

Creating Report Snapshot Data

One additional case for breaking the rules involves accumulating data for reporting. As you can see if you study the examples in Chapter 16, "Advanced Report Design," the queries required to collect data for a complex report can be quite involved. If you have a lot of data required for your report, running the query could take an unacceptably long time, particularly if you need to run several large reports from the same complex collection of data. In this case, it's acceptable to create temporary but "rule-breaking" tables that you load once with the results of a complex query in order to run your reports. We call these tables "snapshots" because they capture the results of a complex reporting query for a single moment in time. You can look in Chapter 9, "Modifying Data with Action Queries," for some ideas about how to build action queries that save a complex data result to a temporary table. If you use the resulting "snapshot" data from these tables, you can run several complex reports without having to run long and complex queries more than once. Chapter 4 shows you how to create a new database and tables, and Chapter 5, "Modifying Your Table Design," shows you how to make changes later if you discover that you need to modify your design.