

Pro SQL Server 2005 Database Design and Optimization



Louis Davidson
with Kevin Kline and Kurt Windisch

Pro SQL Server 2005 Database Design and Optimization

Copyright © 2006 by Louis Davidson, Kevin Kline, and Kurt Windisch

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 981-1-59059-529-9

ISBN-10 (pbk): 1-59059-529-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Matthew Moodie

Technical Reviewers: Dejan Sarka, Andrew Watt

Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade

Project Manager: Elizabeth Seymour

Copy Edit Manager: Nicole LeClerc

Copy Editors: Susannah Pfalzer, Nicole LeClerc

Assistant Production Director: Kari Brooks-Copony

Production Editor: Laura Esterman

Compositor: Lynn L'Heureux

Proofreader: Lori Bring

Indexer: Valerie Perry

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Source Code section.



Data Modeling

A picture is worth a thousand words.

—Royal Baking Powder advertisement¹

It's sad that the preceding quote came from a commercial advertisement for a cooking product, but it really did. Regardless of the source of the saying, though, I'm sure you can think of a number of instances in which it holds true. For example, consider a painting of any reasonable merit. With a little imagination, you can travel off into a dreamworld. And I know that when I see a picture of a new gadget, I start thinking of a thousand reasons why I must have it.

In this chapter, I will introduce the concept of *data modeling*, in which a “picture” will be produced that shows the objects involved in the database design and how they interrelate. But whereas the goal of a painting, for example, is to inspire, the purpose of the picture that will be produced throughout this chapter is to communicate a specific meaning.

In the next section, I'll provide some basic information about data modeling and introduce the modeling tool I prefer for data modeling: IDEF1X. I'll then cover how to use the IDEF1X methodology to model and document the following:

- Entities
- Attributes
- Relationships
- Descriptive information

Next, I'll introduce several other alternative modeling methodology styles, including Information Engineering and the Chen methodology. I'll also devote some time to a discussion of the diagramming capabilities built into SQL Server Management Studio.

Introduction to Data Modeling

Data modeling is a concept at the foundation of database design. In order to start designing databases, you need to be able to effectively communicate the design as well as make it easier to visualize. Most of the objects introduced in Chapter 1 have graphical representations that

1. Not Confucius!

make it easy to get an overview of a vast amount of database structure and metadata in a very small amount of space.

One common misconception about the data model is that it is solely about the graphical display. In fact, the model itself can exist without the graphical parts; it can consist of just textual information. Almost everything in the data model can be read in a manner that makes grammatical sense. The graphical nature is simply there to fulfill the baking powder prophecy—that a picture is worth a thousand words. It is a bit of a stretch, because as you will see, the data model will have lots of words on it!

Note There are many types of models or diagrams: process models, data flow diagrams, data models, sequence diagrams, and others. For the purpose of database design, however, I will focus only on data models.

Several popular modeling languages are available to use, and each is generally just as good as the others at the job of communicating a database design. When choosing my data modeling methodology, I looked for one that was easy to read and could display and store everything required to implement very complex systems. The modeling language I use is Integration Definition for Information Modeling (IDEF1X).

IDEF1X is based on Federal Information Processing Standards Publication 184, published September 21, 1993. To be fair, the other major methodology, Information Engineering, is pretty much just as good, but I like the way IDEF1X works, and it is based on a publicly available standard. IDEF1X was originally developed by the U.S. Air Force in 1985 to meet the following requirements:

- Support the development of data models.
- Be a language that is both easy to learn and robust.
- Be teachable.
- Be well tested and proven.
- Be suitable for automation.

Note At the time of this writing, the full specification for IDEF1X is available at <http://www.itl.nist.gov/fipspubs/idef1x.doc>. The exact URL of this specification is subject to change, but you can likely locate it by searching the <http://www.itl.nist.gov> site for “IDEF1X”.

While the selection of a data modeling methodology may be a personal choice, economics, company standards, or features usually influence tool choice. IDEF1X is implemented in many of the popular design tools, such as the following, which are just a few of the products available that claim to support IDEF1X (note that the URLs listed here were correct at the time of publication, but are subject to change in the future):

- AllFusion ERwin Data Modeler: <http://www3.ca.com/Solutions/Product.asp?ID=260>
- CASE Studio: <http://www.casestudio.com/enu/default.aspx>
- ER/Studio: <http://www.embarcadero.com/products/erstudio34>
- Visible Analyst DB Engineer: <http://www.visible.com/Products/Analyst/vadbengineer.htm>
- Visio Enterprise Edition: <http://www.microsoft.com/office/visio>

Let's next move on to practice modeling and documenting, starting with entities.

Entities

In the IDEF1X standard, *entities* (which, as discussed previously, are synonymous with tables) are modeled as rectangular boxes, as they are in most data modeling methodologies. Two types of entities can be modeled: *identifier-independent* and *identifier-dependent*, usually referred to as *independent* and *dependent*, respectively.

The difference between a dependent entity and an independent entity has to do with how the primary key of the entity is structured. The independent entity is so named because it has no primary key dependencies on any other entity or, to put it in other words, there are no foreign key columns from other entities in the primary key.

Chapter 1 introduced the term “foreign key,” and the IDEF1X specification introduces an additional term: *migrated*. The term “migrated” can be misleading, as the definition of *migrate* is “to move.” The primary key of one entity is not actually moving; rather, in this context the term refers to the primary key of one entity being copied as an attribute in a different entity, thus establishing a relationship between the two entities.

If the primary key of one entity is migrated into the primary key of another, it is considered dependent on the other entity, because one entity is dependent on the existence of the other to have meaning. If the attributes are migrated to the nonprimary key attributes, they are “independent” of any other entities. All attributes that are not migrated as foreign keys from other entities are referred to as *owned*, as they have their origins in the current entity.

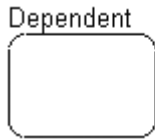
For example, consider an invoice that has one or more line items. The primary key of the invoice entity might be invoiceNumber. So, if the invoice has two line items, a reasonable choice for the primary key would be invoiceNumber and then lineNumber. Since the primary key contains invoiceNumber, it would be dependent upon the invoice entity. If you had an invoiceStatus entity, and it was related to invoice, it would be independent, as an invoice's existence is not really predicated on the existence of a status (even if a value for the invoiceStatus to invoice relationship is required).

An independent entity is drawn with square corners, as follows:

Independent



The dependent entity is the converse of the independent entity, as it will have the primary key of one or more entities migrated into its primary key. It is called “dependent” because its identifier depends on the existence of another entity. It is drawn with rounded-off corners, as follows:



Note The concept of dependent and independent entities lead us to a bit of a chicken and egg paradox. The dependent entity is dependent on a certain type of relationship. However, the introduction of entity creation can't wait until after the relationships are determined, since the relationships couldn't exist without entities. If this is the first time you've looked at data models, this chapter may require a reread to get the full picture, as the concept of independent and dependent objects are linked to relationships.

Entity Naming

One of the most important aspects of designing or implementing any system is how objects, variables, and so forth are named. If you have ever had to go back and work on code that you wrote months ago, you understand what I mean. For example, `@x` might seem like an OK variable name when you first write some code, and it certainly saves a lot of keystrokes versus typing `@holdEmployeeNameForCleaning`, but the latter is much easier to understand after a period of time has passed (for me, this period of time is approximately ten minutes, but your mileage may vary).

Naming database objects is no different, and actually, it is possibly more important to name database objects clearly than it is for other programming objects, as quite often your end users will get used to these names: the names given to entities will be translated into table names that will be used by programmers and users alike. The conceptual and logical model will be considered your primary schematic of the data in the database and should be a living document that you change before changing any implemented structures.

Most discussions on how objects should be named can get heated because there are several different “camps,” each with different ideas about how to name objects. The central issue is plural or singular. Both ways have merit, but one way has to be chosen. I choose to follow the IDEF1X standard, which says to use singular names. The name itself refers to an instance of what is being modeled, but some folks believe that the table name should name the set of rows. Is either way more correct? Not really—just pick one and stick with it. The most important thing is to be consistent and not let your style go all higgledy-piggledy as you go along. Even a bad set of naming standards is better than no standards at all.

In this book, I will follow these basic guidelines for naming entities:

- *Entity names should never be plural.* The primary reason for this is that the name should refer to an instance of the object being modeled rather than the collection. It is uncomfortable to say that you have an “automobiles row,” for example—you have an “automobile row.” If you had two of these, you would have two automobile rows.
- *The name given should directly correspond to the essence of what the entity is modeling.* For instance, if you are modeling a person, name the entity Person. If you are modeling an automobile, call it Automobile. Naming is not always this cut and dried, but it is wise to keep names simple and to the point. If you need to be more specific, that is fine, too. Just keep it succinct (unlike this explanation!).

Entity names frequently need to be made up of several words. During logical modeling, it is acceptable to include spaces, underscores, and other characters when multiple words are necessary in the name, but it is not required. For example, an entity that stores a person's addresses might be named Person Address, Person_Address or, using the style I have recently become accustomed to and the one I'll use in this book, PersonAddress. This type of naming is known as *Pascal case* or *mixed case*. (When you don't capitalize the first letter, but capitalize the first letter of the second word, this style is known as *camelCase*.) Just as in the plural/singular argument, I'm not going to come out and say which is “correct,” just that I'm going to follow a certain guideline to keep everything uniform.

Regardless of any style choices you make, no abbreviations should be used in the logical naming of entities. Every word should be fully spelled out, as abbreviations lower the value of the names as documentation and commonly cause confusion. Abbreviations may be necessary in the implemented model due to some naming standard that is forced upon you, and that will be your problem. If you do decide to use abbreviations in your names of any type, make sure that you have a standard in place to ensure the names use the same abbreviation every time. One of the primary reasons to avoid abbreviations is so you don't have Description, Descry, Desc, Descrip, and Descriptn all used for the same attribute. We'll delve into naming items in the implementation model further in Chapter 5.

A word of warning: You also don't want to go too far in the other direction with long, descriptive sentence names for an entity, such as `leftHandedMittensLostByKittensOnSaturdayAfternoons` (unless the entity will be quite different from `leftHandedMittensLostByKittensOnSundayMornings`), as this name will be painful to use in the logical and implementation models, and prove cumbersome if you need many different tables to represent multiple things that are just categorizations of the same item. A better entity and name might be simply `mitten`. Much of what is encoded in that name will likely be attributes of `mitten`: `mitten status`, `mitten hand`, `mitten used by`, and so forth. However, the whole question of what is an attribute or column actually falls more under the heading of normalization, which I'll discuss in detail in Chapter 4.

It is often the case that novice database designers elect to use a form of *Hungarian notation* and include prefixes and or suffixes in names—for example, `tblEmployee` or `tblCustomer`. The prefix is not needed and is considered a really bad practice. Using Hungarian notation is a good idea when writing functional code (like in Visual Basic or C#), since objects don't always have a very strict contextual meaning that can be seen immediately upon usage, especially if you are implementing one interface with many different types of objects. With database objects, however, it is rare that there is a question as to what each object is when it is used. It is very easy to query the system catalog to determine what the object is if it is not readily available. Not to go too far into implementation right now, but you can use the `sys.objects` catalog view to see the type of any object. For example, this query will list all of the different object types in the catalog:

```
SELECT distinct type_desc
FROM sys.objects
```

Here's the result:

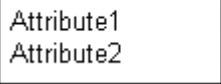
```
type_desc
-----
CHECK_CONSTRAINT
DEFAULT_CONSTRAINT
FOREIGN_KEY_CONSTRAINT
INTERNAL_TABLE
PRIMARY_KEY_CONSTRAINT
SERVICE_QUEUE
SQL_SCALAR_FUNCTION
SQL_STORED_PROCEDURE
SQL_TABLE_VALUED_FUNCTION
SQL_TRIGGER
SYNONYM
SYSTEM_TABLE
USER_TABLE
VIEW
```

We will use `sys.objects` more in Chapter 5 and beyond to view properties of objects that we create.

Attributes

All attributes in the entity must be uniquely named within it. They are represented by a list of names inside of the entity rectangle:

AttributeExample

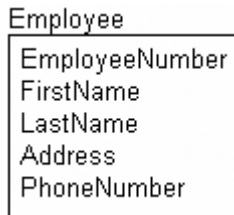


```
Attribute1
Attribute2
```

Note The preceding image shows a technically invalid entity, as there is no primary key defined (as required by IDEF1X). I'll cover the notation for keys in the following section.

At this point, you would simply enter all of the attributes that have been defined in the discovery phase. In practice, it is likely that you would have combined the process of discovering entities and attributes with the initial modeling phase. It will all depend on how well the tools you use work. Most data modeling tools cater for building models fast and storing a wealth of information to document their entities and attributes.

In the early stages of logical modeling, there can be quite a large difference between an attribute and what will be implemented as a column. As I will demonstrate in Chapter 4, the attributes will be transformed a great deal during the normalization process. For example, the attributes of an Employee entity may start out as follows:



However, during the normalization process, tables like this will often be broken down into many attributes (e.g., address might be broken into number, street name, city, state, zip code, etc.) and possibly many different entities.

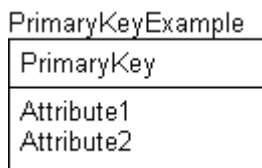
Note Attribute naming is one place where I tend to deviate from IDEF1X standard. The standard is that names are unique within a model. This tends to produce names that include the table name followed by the attribute name, which can result in unwieldy, long names.

Just as with entity names, there is no need to include Hungarian prefixes or suffixes in the attribute names now or in implementation names. The type of the attribute can be retrieved from the system catalog if there is any question about it.

Primary Key

As noted in the previous section, an IDEF1X entity must have a primary key. This is convenient for us, as in Chapter 1 an entity was defined such that each instance must be unique. The primary key may be a single attribute, or it may be a composite of multiple attributes. A value is required for every attribute in the key (logically speaking, no nulls are allowed in the primary key).

The primary key is denoted by placing attributes above a horizontal line through the entity rectangle. Note that no additional notation is required to indicate that the value is the primary key.



For example, consider the Employee entity from the previous section. The EmployeeNumber attribute is going to be unique, so this would be an acceptable primary key:

Employee	
EmployeeNumber	
FirstName	
LastName	
Address	
PhoneNumber	

In the early logical modeling phase, I generally do not like to spend time choosing the final primary key attribute(s). The main reason for this is to avoid worrying too much about what the key is going to be. I tend to add a meaningless primary key to migrate to other entities to help me see when there is any ownership. In the current example, `employeeNumber` clearly refers to an employee, but not every entity will be so clear. For example, consider an entity that models a product manufactured by a company. The company may identify the product by the type, style, size, and series:

Product	
Type	
Style	
Size	
Series	
ProductName	

The name may be a good key, and more than likely there is also a product code. Which is the best key—or which is even truly a key—may not become apparent until much later in the process. There are many ways to implement a good key, and the best way may not be instantly recognizable.

Instead of choosing a primary key during this part of the process, I add a surrogate value to the entity for identification purposes. I then model all candidate keys (or unique identifiers) as alternate keys. The result is that it is very clear in the logical model what entities are in an ownership role to other entities, since the key that is migrated contains the name of the modeled entity. I would model this entity as follows:

Product	
ProductId	
Type	
Style	
Size	
Series	
Name	

Note Using surrogate keys is certainly not a requirement in logical modeling; it is a personal preference that I have found a useful documentation method to keep models clean, and it corresponds to my method of implementation later. Not only is using a natural key as the primary key in the logical modeling phase reasonable, but also many architects find it preferable. Either method is perfectly acceptable.

Alternate Keys

As defined in Chapter 1, an *alternate key* is a set of one or more attributes whose uniqueness needs to be guaranteed over all of the instances of the entity. Alternate keys do not have a specific location like primary keys, and they are not migrated for any relationship. They are identified on the model in a very simple manner:

AlternateKeyExample

PrimaryKey
AlternateKey1 (AK1) AlternateKey2Attribute1 (AK2) AlternateKey2Attribute2 (AK2)

In this example, there are two alternate key *groups*: group AK1, which has one attribute as a member, and group AK2, which has two attributes. Thinking back to the product example, the two keys would then be modeled as follows:

Product

ProductId
Type (AK1) Style (AK1) Size (AK1) Series (AK1) Name (AK2)

One extension that ERwin adds to this notation is shown here:

AlternateKeyExample

PrimaryKey
AlternateKey1 (AK1.1) AlternateKey2Attribute1 (AK2.1) AlternateKey2Attribute2 (AK2.2)

A position number notation is tacked onto the name of the key (AK1 and AK2) to denote the position of the attribute in the key. In the logical model, technically the order of attributes in the key should not be considered and certainly should not be displayed. It really does not matter which attribute comes first in the key; all that really matters is that you make sure there are unique values across multiple attributes. When a key is implemented, the order of columns *will* become interesting for performance reasons, as SQL Server implements uniqueness with an index, but uniqueness will be served no matter what the order of the columns of the key is.

Foreign Keys

As I've alluded to, foreign keys are also referred to as migrated attributes. They are primary keys from one entity that serve as a reference to an instance in another entity. They are again a result of relationships (which we'll look at later in the chapter). They are indicated much like alternate keys by adding the letters "FK" after the foreign key:

ForeignKeyExample

PrimaryKey
ForeignKey (FK)

For example, consider an entity that is modeling a music album:

Album
AlbumId
Name (AK1)
ArtistId (FK)(AK1)
PublisherId (FK)(AK1)
CatalogNumber (AK2)

The artistId and publisherId represent migrated foreign keys from the artist entity and the publisher entity. We'll revisit this example in the "Relationships" section later in the chapter.

One tricky thing about this example is that the diagram doesn't show what entity the key is migrated from. This can tend to make things a little messy, depending on how you choose your primary keys. This lack of clarity about what table a foreign key migrates from is a limitation of most modeling methodologies, as it would be unnecessarily confusing if the name of entity where the key came from was displayed, for a couple of reasons:

- There is no limit (nor should there be) on how far a key will migrate from its original owner entity (the entity where the key value was not a migrated foreign key reference).
- It is not unreasonable that the same attribute might migrate from two separate entities, especially early in the logical design process.

One of the reasons for the primary key scheme I will employ in logical models is to add a key named <entityName>Id as the identifier for entities. The name of the entity is easily identi-

fiable, it lets us easily know where the original source of the attribute is, and we can see the attribute migrated from entity to entity.

Domains

The term “domain” is regrettably used in two very similar contexts in the database world. In Chapter 1, a domain referred to a set of valid values for an attribute. In IDEF1X, you can define named, reusable specifications known as domains, for example:

- `String`: A character string
- `SocialSecurityNumber`: A character value with a format of ###-##-####
- `PositiveInteger`: An integer value with an implied domain of 0 to `max(integer value)`
- `Truth`: A five-character value with a domain of ('FALSE', 'TRUE')

Domains in the specification not only allow us to define the valid values that can be stored in an attribute, but also provide a form of inheritance in the datatype definitions. *Subclasses* can then be defined of the domains that inherit the settings from the base domain. It is a good practice to build domains for any attributes that get used regularly, as well as domains that are base templates for infrequently used attributes. For example, you might have a character type domain where you specify a basic length, like 60. Then you may specify common domains like *name* and *description* to use in many entities. For these, you should choose a reasonable length for the values, plus you should require that the data cannot be an empty string.

Regardless of whether or not you are using a tool, it is useful to define common domains that you use for specific types of things (kind of like applying a common pattern to solve a common problem). For example, a person's first name might be a domain. The reason that this is so cool is that you don't have to think “Hmm, how long to make a person's name?” more than once. After you make a decision, you just use what you have used before.

In logical modeling, you'll likely want to keep a few bits of information, such as the general type of the attribute: character, numeric, logical, or even binary data. It's also important to document in some form the legal values for an attribute that is classified as being of the domain type. This is generally done using some pseudocode or in textual manner, either in your modeling tool or even in a spreadsheet.

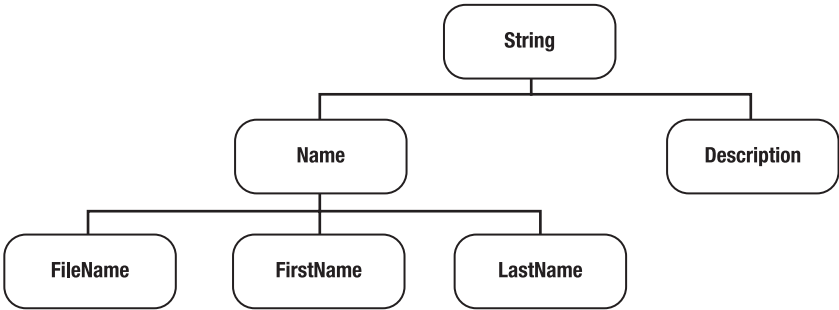
It is extremely important to keep these domains as implementation-independent datatype descriptions. For example, you might specify a domain of `GloballyUniqueIdentifier`, a value that will be unique no matter where it is generated. In SQL Server, a unique identifier could be used (GUID value) to implement this domain. In another operating system (created by a company other than Microsoft) where there is not exactly the same mechanism, it might be implemented differently; the point is that it is a value that is statistically guaranteed to be unique every time the value is generated.

When you start implementation modeling, you will use the same domains to assign the implementation properties. This is the real value in using domains. By creating reusable template attributes that will also be used when you start creating columns, you'll spend less effort and time building simple entities, which make up the bulk of your work. Doing so also provides a way for you to enforce companywide standards, by reusing the same domains on all corporate models (predicated, of course, on you being diligent with your data model over time!).

Later on, implementation details such as datatype, constraints, and so forth will be chosen, just to name a few of the more basic properties that may be inherited. Since it is very

likely that you will have fewer domains than implemented attributes, the double benefit of speedy and consistent model assembly is achieved. However, it is not reasonable or even useful to employ the inheritance mechanisms when building tables by hand. Implementation of domains is way too much trouble to do without a tool.

As an example of a domain hierarchy, consider this set of character string domains:



Here, `String` is the base domain from which you can then inherit `Name` and `Description`. `FileName`, `FirstName`, and `LastName` are inherited from `Name`. During logical modeling, this might seem like a lot of work for nothing, because most of these domains will share some basic details, such as not allowing NULLs or blank data. However, `FileName` may be optional, whereas `LastName` might be mandatory. It is important to set up domains for as many distinct attribute types as possible, in case rules or datatypes are discovered that are common to any domains that already exist.

Domains are a nice feature of IDEF1X. They provide an easy method of building standard attribute types, reducing both the length of time required for repeating common attribute types and the number of errors that occur in doing so. Specific tools implement domains with the ability to define and inherit more properties throughout the domain chain to make creating databases easier. During logical modeling, domains can optionally be shown to the right of the attribute name in the entity:

DomainExample

AttributeName: DomainName
AttributeName2: DomainName2

So if I have an entity that holds domain values for describing a type of person, I might model it as follows:

Person

PersonId: SurrogateKey
Description: Description
FirstName: PersonFirstName
LastName: PersonLastName

To model this example, I defined four domains:

- `SurrogateKey`: The surrogate key value. (Implementation is not implied by building a domain, so later this can be implemented in any manner.) I could also choose to use a natural key.
- `Description`: The same type of domain as the name domain, except to hold a description (can be 60 characters maximum).
- `PersonFirstName`: A person's first name (30 characters maximum).
- `PersonLastName`: A person's last name (50 characters maximum).

The choice of the length of name is an interesting one. I searched on Google for “person first name varchar” and found lots of different possibilities: 10, 35, unlimited, 25, 20, 15—all on the first page of the search! Just as you should use a consistent naming standard, you should use standard lengths every time like data is represented, so when you hit implementation the likelihood that two columns storing like data will have different definitions is minimized.

Naming

Attribute naming is a bit more interesting than entity naming. I stated earlier that my preference is to use singular, not plural, entity names. The same issues that apply in entity naming are technically true for attribute naming (and no one disagrees with this!). However, until the logical model is completed, the model may still have attribute names that are plural. Leaving a name plural can be a good reminder that you expect multiple values. For example, consider a Person entity with a Children attribute identified. The Person entity would identify a single person, and the Children attribute would identify sons and daughters of that person.

The naming standard I follow is very simple:

- Generally, it is not necessary to repeat the entity name in the attribute name, though it is common to do so with a surrogate key, since it is specific for that table. The entity name is implied by the attribute's inclusion in the entity.
- The chosen attribute name should reflect precisely what is contained in the attribute and how it relates to the entity.
- As with entities, no abbreviations are to be used in attribute names. Every word should be spelled out in its entirety. If for some reason an abbreviation must be used (e.g., due to naming standard currently in use), then a method should be put into place to make sure the abbreviation is used consistently, as discussed earlier in the chapter. For example, if your organization has a ZRF “thing” that is commonly used and referred to in general conversation as a ZRE, you might use this abbreviation. In general, however, I recommend avoiding abbreviations in all naming unless the client is insistent.
- The name should contain absolutely no information other than that necessary to explain the meaning of the attribute. This means no Hungarian notation of the type of data it represents (e.g., `LastNameString`) or prefix notation to tell you that it is in fact an attribute.

Note Attribute names in the finalized logical and implementation models will not be plural, but we'll work this out in Chapter 4 when normalizing the model. At this point it is not a big deal at all.

Consistency is the key to proper naming, so if you or your organization does not have a standard naming policy, it's worthwhile to develop one. My naming philosophy is to keep it simple and readable, and to avoid all but universally standard corporate abbreviations. This standard will be followed from logical modeling into the implementation phase. Whatever your standard is, establishing a pattern of naming will make your models easy to follow, both for yourself and for your programmers and users. Any standard is better than no standard.

Relationships

Up to this point, the constructs we have looked at have been pretty much the same across all data modeling methodologies. Entities are always signified by rectangles, and attributes are quite often words within the rectangles. In IDEF1X, every relationship is denoted by a line drawn between two entities, with a solid circle at one end of that line.

This is where things start to get confusing when it comes to the other methodologies, as each approaches relationships a bit differently. I favor IDEF1X for the way it represents relationships. To make the concept of relationships clear, I need to go back to the terms “parent” and “child.” Consider the following definitions from the IDEF1X specification's glossary:²

- Entity, Child: The entity in a specific connection relationship whose instances can be related to zero or one instance of the other entity (parent entity)
- Entity, Parent: An entity in a specific connection relationship whose instances can be related to a number of instances of another entity (child entity)
- Relationship: An association between two entities or between instances of the same entity

In the following image, the primary key of the parent is migrated to the child. This is how to denote a foreign key on a model.



There are several different types of relationships that we'll examine in this section:

- *Identifying*, where the primary key of one table is migrated to the primary key of another. The child will be a dependent entity.
- *Nonidentifying*, where the primary key of one table is migrated to the nonprimary key attributes of another. The child will be an independent entity as long as no identifying relationships exist.

2. These are remarkably lucid definitions to have been taken straight from a government specification!

- *Optional identifying*, when the nonidentifying relationship does not require a child value.
- *Recursive*, when a table is related to itself.
- *Subtype* or *categorization*, which is a one-to-one relationship used to let one entity extend another.
- *Many-to-many*, where an instance of an entity can be related to many in another, and in turn many instances of the second entity can be related to multiples in the other.

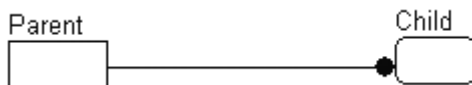
We'll also cover the *cardinality* of the relationship (how many of the parent relate to how many of the child), *role names* (changing the name of a key in a relationship), and *verb phrases* (the name of the relationship).

Relationships are a key topic in database design, but not a completely simple one. A lot of information is related using a few dots and lines.

Tip All of the relationships (except the last one) discussed in this section are of the one-to-many variety, which encompasses *one-to-zero*, *one*, *many*, or perhaps *exactly-n* relationships. Technically, it is more accurately *one-to-(from M to N)*, as this enables specification of the *many* in very precise (or very loose) terms as the situation dictates. However, the more standard term is “one-to-many,” and I will not try to make an already confusing term more so.

Identifying Relationship

The *identifying relationship* indicates that the migrated primary key attribute is migrated to the primary key of the child. It is drawn as follows:



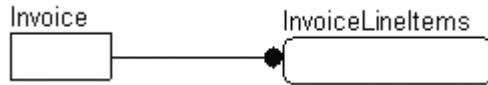
In the following example, you can see that the ParentId attribute is a foreign key in the Child entity, from the Parent entity.



The child entity in the relationship is drawn as a rounded-off rectangle, which as you learned earlier in this chapter means it is a dependent entity. It is called an identifying relationship because it will need to have to have a parent instance in order to be able to identify a child instance record. The essence (defined as “the intrinsic or indispensable properties that serve to characterize or identify something”) of the child instance is defined by the existence of a parent.

Another way to look at this is that generally the child in an identifying relationship is a part of the parent. Without the existence of the parent, the child would make no sense.

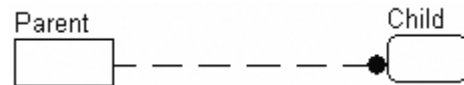
A common example is an invoice and the line items being charged to the customer on the invoice:



Without the existence of the invoice, the line items would have no reason to exist. It can also be said that the line items are identified as being part of the parent.

Nonidentifying Relationship

The *nonidentifying relationship* indicates that the primary key attribute is not migrated to the primary key of the child. It is denoted by a dashed line between the entities:



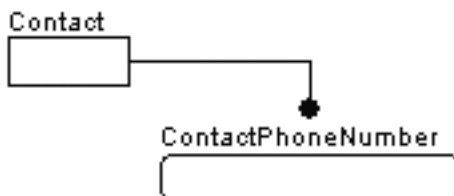
Nonidentifying relationships are used more frequently than identifying relationships. Whereas the identifying relationship indicated that the child was an essential part of the parent entity, the nonidentifying relationship indicates that the child represents an attribute of the parent.

Taking again the example of an invoice, consider the vendor of the products that have been sold and documented as such in the line items. The product vendor does not define the existence of a line item, because with or without knowing the vendor, the line item still makes sense.

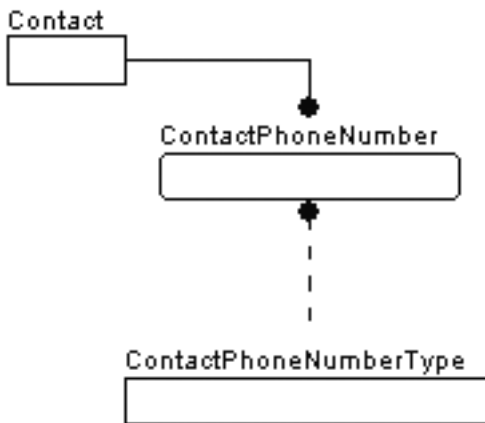
The difference between identifying and nonidentifying relationships can be tricky. If the parent entity defines the need for the existence of the child (as stated in the previous section), then use an identifying relationship. If, on the other hand, the relationship defines one of the child's attributes, then use a nonidentifying relationship.

Consider the following examples:

- *Identifying*: You have an entity that stores a contact and an entity that stores the contact's telephone number. The Contact defines the phone number, and without the contact, there would be no need for the ContactPhoneNumber.



- *Nonidentifying*: Consider the entities that were defined for the identifying relationship, along with an additional entity called `ContactPhoneNumberType`. This entity is related to the `ContactPhoneNumber` entity, but in a nonidentifying way, and defines a set of possible phone number types (Voice, Fax, etc.) that a `ContactPhoneNumber` might be. The type of phone number does not identify the phone number; it simply classifies it.



The `ContactPhoneNumberType` entity is commonly known as a *domain entity* or *domain table*, or sometimes as a *lookup table*. Rather than having a fixed domain for an attribute, an entity is designed that allows programmatic changes to the domain with no recoding of constraints or client code. As an added bonus, you can add columns to define, describe, and extend the domain values to implement business rules. It also allows the client user to build lists for users to choose values with very little programming.

Optional Identifying Relationship

While every nonidentifying relationship defines the domain of an attribute of the child table, sometimes when the row is created it's not necessary that the values are selected. For example, consider a database where you model houses, like for a neighborhood. Every house would have a color, a style, and so forth. However, not every house would have an alarm company, a mortgage holder, and so on. The relationship between the alarm company and bank would be optional in this case, while the color and style relationships would be mandatory.

Note Here I assume you would need a table for color and style. Whether or not this is required is something that will be discussed in the next couple of chapters. (Hint: You usually will.)

The difference in the implemented table will be whether or not the child table's foreign key will allow nulls. If a value is required, then it is considered *mandatory*. If a value of the migrated key can be null, then it is considered *optional*.

The optional case is signified by an open diamond at the opposite end of the dashed line from the black circle, as shown here:

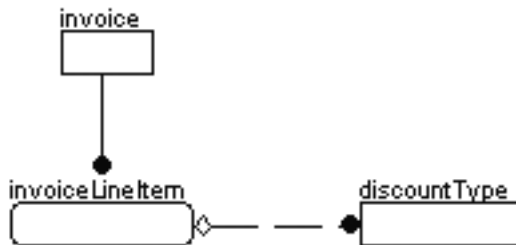


In the mandatory case, the relationship is drawn as before, without the diamond. Note that an optional relationship may have a cardinality of zero, while a mandatory relationship must have a cardinality of one or greater. (The concept of cardinality will be discussed further in the next section.)

So why would you make a relationship optional? Consider once again the nonidentifying relationship between the invoice line item and the product vendor. The vendor in this case may be required or not required as the business rules dictate. If it is not required, you should make the relationship optional.

Note You might be wondering why there is not an optional identifying relationship. This is due to the fact that you may not have any optional attributes in a primary key, which is true in relational theory and for SQL Server 2005.

For a one-to-many, optional relationship, consider the following:



The `invoiceLineItem` entity is where items are placed onto an invoice to receive payment. The user may sometimes apply a standard discount amount to the line item. The relationship then from the `invoiceLineItem` to the `discountType` entity is an optional one, as no discount may have been applied to the line item.

For most optional relationships like this, there is another possible solution, which can be modeled as required, and in the implementation a row can be added to the `discountType` table that indicates “none.”

An example of a mandatory relationship could be genre to movie in a movie rental shop’s database:



The relationship is genre <classifies> movie, where the genre entity represents the one and movie represents the many in the one-to-many relationship. Every movie being rented must have a genre, so that it can be organized in the inventory and then placed on the appropriate rental shelf.

Cardinality

The cardinality of the relationship denotes the number of child instances that can be inserted for each parent of that relationship. The following table shows the six possible cardinalities that relationships can take on (see Figures 2-1 through 2-6). The cardinality indicators are applicable to either mandatory or optional relationships.

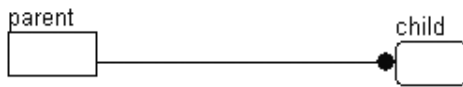


Figure 2-1. One-to-zero or more



Figure 2-2. One-to-one or more (at least one)



Figure 2-3. One-to-zero or one (no more than one)

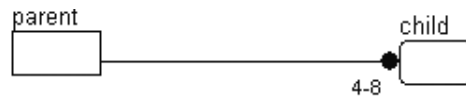


Figure 2-4. One-to-some fixed range (in this case, between 4 and 8 inclusive)



Figure 2-5. One-to-exactly N (in this case, 5, meaning each parent must have five children)

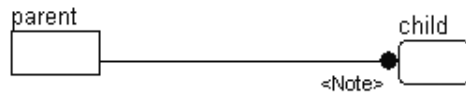
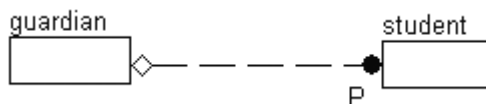


Figure 2-6. Specialized note describing the cardinality

For example, a possible use for the one to one-or-more might be to represent the relationship between a guardian and a student in a school database:



This is a good example of a zero-or-one to one-or-more relationship, and a fairly complex one at that. It says that for a guardian record to exist, a student must exist, but a student record need not have a guardian for us to wish to store the guardian's data. Note that I did not limit the number of guardians in the example, since it is not clear at this point if there is a limit.

Next, let's consider the case of a club that has members with certain positions that they should or could fill, as shown in Figures 2-7 through 2-9.

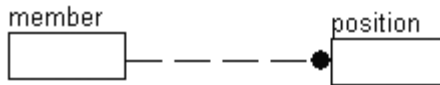


Figure 2-7. Example A

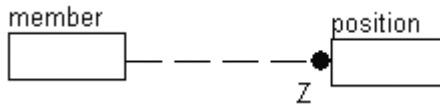


Figure 2-8. Example B

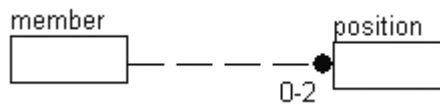


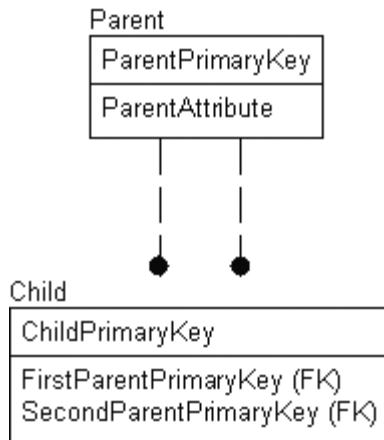
Figure 2-9. Example C

Figure 2-7 shows that a member can take as many positions as there are possible. Figure 2-8 shows that a member can serve in no position or one position, but no more. Finally, Figure 2-9 shows that a member can serve in zero, one, or two positions. They all look about the same, but the Z or 0-2 is important in signifying the cardinality.

Note I considered including examples of each of these cardinality types, but in most cases they were too difficult or too silly.

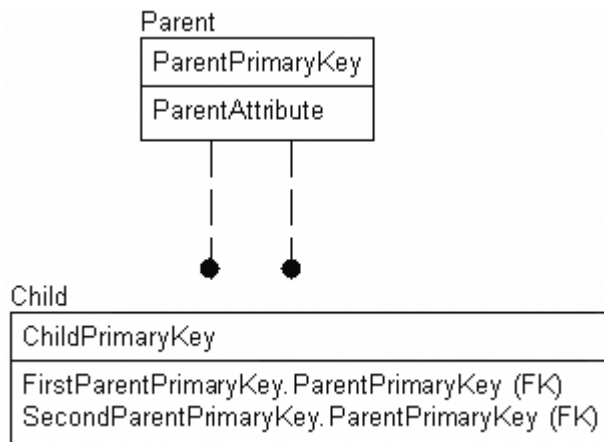
Role Names

A *role name* is an alternative name you can give an attribute when it is used as a foreign key. The purpose of a role name is to clarify the usage of a migrated key, either because the parent entity is very generic and a more specific name is needed or because the same entity has multiple relationships. As attribute names must be unique, it is often necessary to assign different names for the child foreign key references. Consider this example:

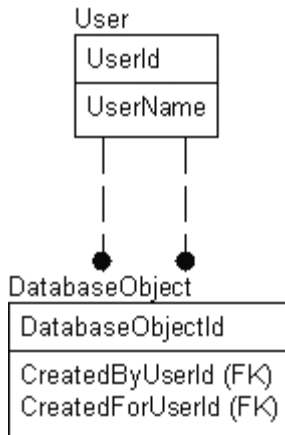


In this diagram, there are two relationships from the **Parent** entity to the **Child** entity, and the migrated attributes have been named as **FirstParentPrimaryKey** and **SecondParentPrimaryKey**.

In diagrams, you can indicate the original name of the migrated attribute after the role name, separated by a period (.), as follows:



As an example, say you have a **User** entity, and you want to store the name or ID of the user who created a **DatabaseObject** entity. It would then end up as follows:



Note that there are two relationships to the **DatabaseObject** entity from the **User** entity. It is not clear from the diagram which foreign key goes to which relationship.

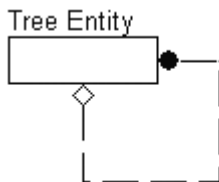
Other Types of One-to-N Relationships

There are a few other, less important relationship types that are not employed nearly as much as the previously mentioned ones. However, it's extremely important that you understand them, as they are a bit tricky and provide advanced solutions to problems that are not easily solved.

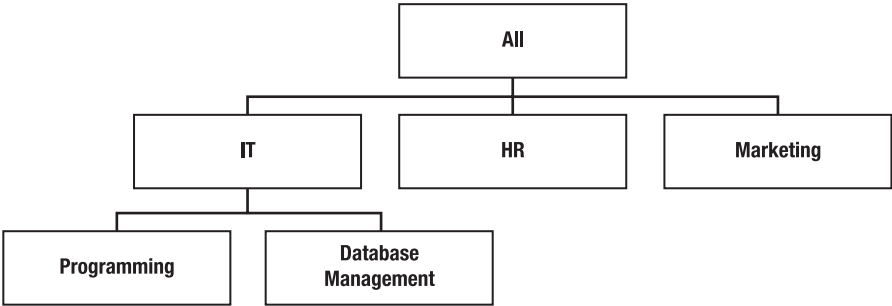
- *Recursive*: Used to model hierarchies in data
- *Subtypes*: Used to model a limited type of inheritance that allows specific types of entities to be built to extend general entities

Recursive

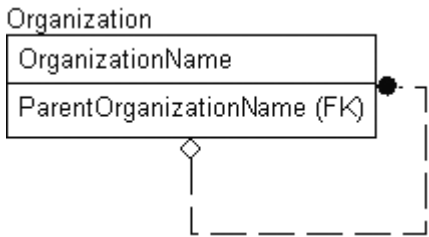
One of the more difficult—but most important—relationships to implement is the *recursive relationship*, also known as a *self-join*, *hierarchical*, *self-referencing*, or *self-relationship*. This is modeled by drawing a nonidentifying relationship not to a different entity, but to the same entity. The migrated key of the relationship is given a role name. (I generally use a naming convention of adding “parent” to the front of the attribute name, but this is not a necessity.)



The recursive relationship is useful for creating tree structures, as in the following organizational chart:



To explain this concept fully, I will show the data that would be stored to implement this hierarchy:

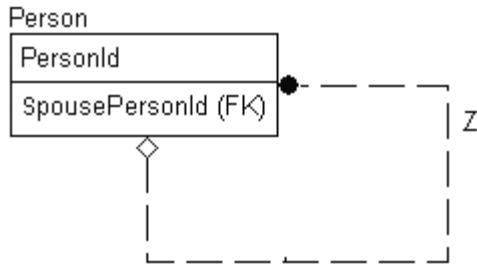


Here is the sample data for this table:

organizationName	parentOrganizationName
-----	-----
All	
IT	All
HR	All
Marketing	All
Programming	IT
Database Management	IT

The organizational chart can now be traversed by starting at All and getting the children to ALL, for example: IT. Then you get the children of those values, like for IT one of the values is Programming.

As a final example, consider the case of a Person entity. If you wanted to associate a person with a single other person as the first person's spouse, you might design the following:

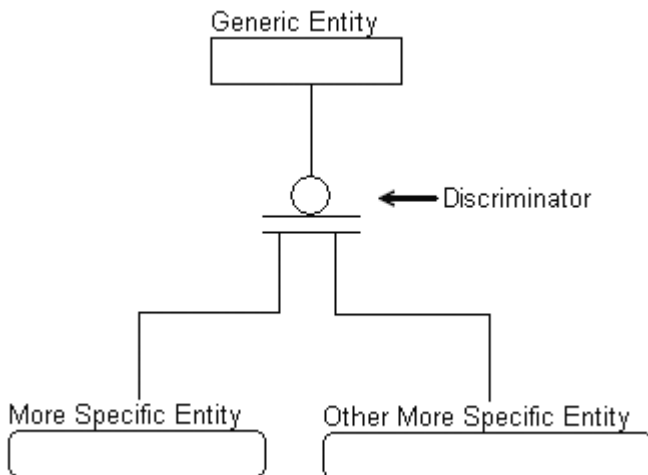


Notice that this is a one-to-zero or one relationship, since (in most places) a person may have no more than a single spouse, but need not have one. If you require one person to be related as a child to two parents, an associative entity is required to link two people together.

Note As an aside, a tree where each instance can have only one parent (but unlimited children) is called a *binary tree*. If you model the relationship as a many-to-many using an associative entity, allowing an instance to have more than one parent, this implements a *graph*.

Subtypes

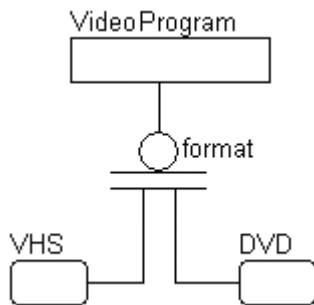
Subtypes (also referred to as *categorization relationships*) are another special type of one-to-zero or -one relationship used to indicate whether one entity is a specific type of a generic entity. Note also that there are no black dots at either end of the lines; the specific entities are drawn with rounded corners, signifying that they are indeed dependent on the generic entity.



There are three distinct parts of the subtype relationship:

- *Generic entity*: This entity contains all of the attributes common to all of the subtyped entities.
- *Discriminator*: This attribute acts as a switch to determine the entity where the additional, more specific information is stored.
- *Specific entity*: This is the place where the specific information is stored, based on the discriminator.

For example, let's look at a video library. If you wanted to store information about each of the videos that you owned, regardless of format, you might build a categorization relationship like the following:



In this manner, you might represent each video's price, title, actors, length, and possibly description of the content in the Video entity, and then, based on format, which is the discriminator, you might store the information that is specific to VHS or DVD in its own separate entity (e.g., special features and menus for DVDs, long or slow play for VHS tapes, etc.).

Tip The types of relationships in this example are what I referred to earlier as “is-a” relationships: a VHS is a video, and a DVD is also a video.

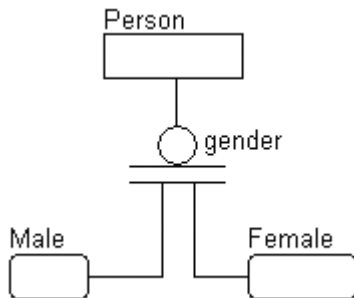
There are two distinct category types: *complete* and *incomplete*. The complete set of categories is modeled with a double line on the discriminator, and the incomplete set is modeled with a single line (see Figure 2-10).



Figure 2-10. Complete (left) and incomplete (right) sets of categories

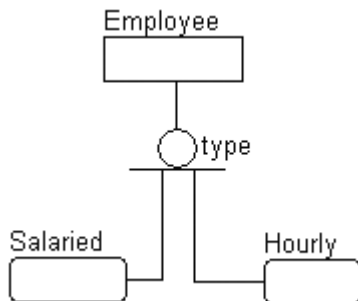
The primary difference between the complete and incomplete categories is that in the complete categorization relationship, each generic instance must have one specific instance, whereas in the incomplete case this is not necessarily true. An instance of the generic entity can be associated with an instance of only one of the category entities in the cluster, and each instance of a category entity is associated with exactly one instance of the generic entity. In other words, overlapping subentities are not allowed.

For example, you might have a complete set of categories like this:



This relationship is read as follows: “A Person *must* be either Male or Female.” This is certainly a complete category. This is not to say that you know the gender of every person in every instance of all entities. Rather, it simply means that if the instance has been categorized, any person must fall in one of the two buckets (male or female).

However, consider the following incomplete set of categories:



This is an incomplete subtype, because employees are either salaried or hourly, but there may be other categories, such as contract workers. You may not need to store any additional information about them, though, so there is no need to implement the specific entity. This relationship is read as follows: “An Employee can be either Salaried or Hourly or other.”

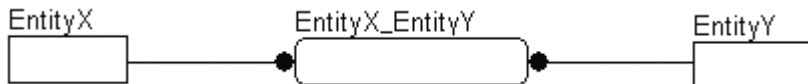
Many-to-Many Relationship

The many-to-many relationship is also known as the *nonspecific relationship*, which is actually a better name, but far less well known. It is common to have many-to-many relationships in the data model. In fact, the closer you get to your final database model, the more you'll find that quite a few of your relationships will be of the many-to-many type.

These relationships are modeled by a line with a solid black dot at either end:



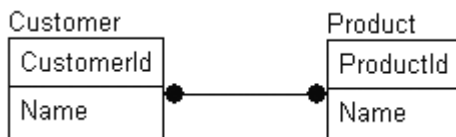
There is one real problem with modeling a many-to-many relationship: it is often necessary to have more information about the relationship than that simply many EntityX instances are connected to many EntityY instances. So the relationship is usually modeled as follows:



Here, the intermediate EntityX_EntityY entity is known as an *associative entity* or *resolution entity*. In modeling, I will often stick with the former representation when I haven't identified any extended attributes to describe the relationship and the latter representation when I need to add additional information to the model.

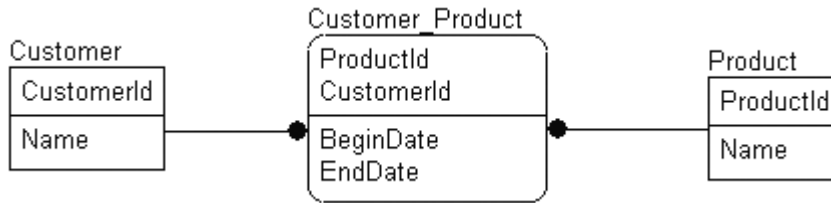
Tip I should also note that you can't implement a many-to-many relationship in the relationship model without using a table for the resolution. This is because there is no way to migrate keys both ways. You will notice when you use a many-to-many relationship that no key is migrated from either table, so there would be no data to substantiate the relationship. In the database, you are required to implement all many-to-many relationships using a resolution entity.

To clarify the concept, let's look at the following example:



Here I have set up a relationship where many customers are related to many products. This is a common situation, as in most cases companies don't create specific products for specific customers; rather, any customer can purchase any of the company's products. At this point in the modeling, it is reasonable to use the many-to-many representation. Note that I am generalizing the customer-to-product relationship. It is not uncommon to have a company build specific products for only one customer to purchase.

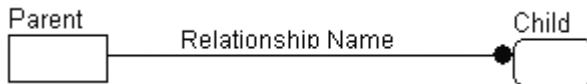
Consider, however, the case where the Customer need only be related to a Product for a certain period of time. To implement this, you can use the following representation:



In fact, almost all of the many-to-many relationships tend to require some additional information like this to make them complete. It is not uncommon to have no many-to-many relationships modeled with the black circle on both ends of a model, so you will need to look for entities modeled like this to be able to discern them.

Verb Phrases (Relationship Names)

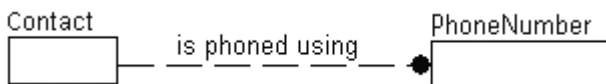
Relationships are given names, called *verb phrases*, to make the relationship between a parent and child entity a readable sentence and to incorporate the entity names and the relationship cardinality. The name is usually expressed from parent to child, but it can be expressed in the other direction, or even in both directions. The verb phrase is located on the model somewhere close to the line that forms the relationship:



The relationship should be named such that it fits into the following general structure for reading the entire relationship:

parent cardinality – parent entity name – relationship name – child cardinality – child entity name

For example, the following relationship



would be read as

One contact is phoned using zero, one, or more phoneNumbers(s).

Of course, the sentence may or may not make perfect grammatical sense, as this one brings up the question of how a contact is phoned using zero phone numbers. If presenting this phrase to a nontechnical person, it would make more sense to read it as follows:

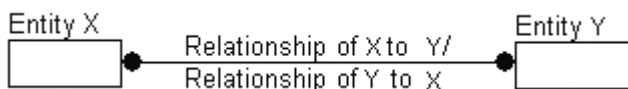
One contact can have either no phone number or one or more phoneNumbers.

You can then name the relationship from child to parent. Note that when naming in this direction, you are in the context of zero or one phone number to one and only one contact.

Zero or one phoneNumber(s) may be used to phone exactly one contact.

Since this is going from many to one, it is assumed that the parent in the relationship will have one related value, and since you are reading in the context of the existence of the child, you can also assume that there is zero or one child record to consider in the sentence.

For the many-to-many relationship, the scheme is pretty much the same. As both entities are parents in this kind of relationship, you read the verb phrase written above the line from left to right and from right to left for the verb phrase written below it.



Tip Getting people to take the time to define verb phrases can be troublesome, because they are not actually used in the implementation of the database. However, they make for great documentation, giving the reader a good idea of why the relationship exists and what it means.

Descriptive Information

We have drawn entities, assigned attributes and domains to them, and set up relationships between them, but this is not quite the end of the road yet. I have discussed naming entities, attributes, and the relationships, but even with well-formed names, there will still likely be confusion as to what exactly an attribute is used for.

We also need to add comments to the pictures in the model. When sharing the model, comments will let the eventual reader—and even yourself—know what you originally had in mind. Remember that not everyone who views the models will be on the same technical level: some will be nonrelational programmers, or indeed users or (nontechnical) product managers who have no modeling experience.

Descriptive information need not be in any special format. It simply needs to be detailed, up to date, and capable of answering as many questions as can be anticipated. Each bit of descriptive information should be stored in a manner that makes it easy for users to quickly connect it to the part of the model where it was used, and it should be stored either in a document or as metadata in a modeling tool.

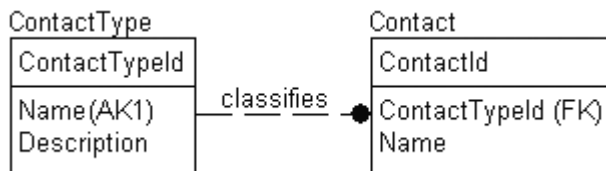
You should start creating this descriptive text by asking questions such as the following:

- What is the object supposed to represent?
- How will the object be used?
- Who might use the object?

- What are the future plans for the object?
- What constraints are not specifically implied by the model?

The scope of the descriptions should not extend past the object or entities that are affected. For example, the entity description should refer only to the entity, and not any related entities, relationships, or even attributes unless necessary. An attribute definition should only speak to the single attribute and where its values might come from.

Maintaining good descriptive information is equivalent to putting decent comments in code. As the eventual database that you are modeling is usually the central part of any computer system, comments at this level are more important than any others. For example, say the following two entities have been modeled:



The very basic set of descriptive information in Tables 2-1 and 2-2 could be stored to describe the attributes created.

Table 2-1. Entities

Contact		Persons That Can Be Contacted to Do Business With	
Attributes		Description	
ContactId		Surrogate key	
ContactTypeId		Primary key pointer to a contactType	
Name		The full name of a contact	

ContactType		Domain of Different Contact Types	
Attributes		Description	
ContactTypeId		Surrogate key	
Name		The name that the contact type will be uniquely known as	
Description		The description of exactly how the contact should be used as	

Table 2-2. Relationships

Parent Entity Name	Phrase	Child Entity Name	Definition
ContactType	Classifies	Contact	Contact type classification. Was required by specifications.

Alternative Modeling Methodologies

In this section, I will briefly describe a few of the other modeling methodologies that you will likely run into frequently when designing databases for SQL Server 2005. You will see a lot of similarities among them—for example, most every methodology uses a rectangle to represent a table, and a line to indicate a relationship. You will also see some big differences among them, such as how the cardinality and direction of a relationship is indicated. Where IDEF1X uses a filled circle on the child end and an optional diamond on the other, one of the most popular methodologies uses multiple lines on one end and several dashes to indicate the same things.

All of the examples in this book will be done in IDEF1X, but knowing about the other methodologies may be helpful when you are surfing around the Internet, looking for sample diagrams to help you design the database you are working on. I will briefly discuss the following:

- *Information Engineering (IE)*: The other main methodology, commonly referred to as the *crow's feet* method
- *Chen Entity Relationship Model (ERD)*: The methodology used mostly in academic texts
- *Management Studio database diagrams*: The database object viewer that can be used to view the database as a diagram right in Management Studio





Information Engineering

The *Information Engineering* (IE) methodology is well known and widely used. It does a very good job of displaying the necessary information.

Tables in this method are basically the same as in IDEF1X, and they are denoted as rectangles. According to the IE standard, attributes are not shown on the model, but most models show them the same as in IDEF1X—as a list, although the primary key denoted by underlining the attributes, rather than the position in the table. (I have seen other ways of denoting the primary key, as well as alternate/foreign keys, but they are all very clear.) Where things get very different using IE is when dealing with relationships.

Just like in IDEF1X, IE has a set of symbols that have to be understood to indicate the cardinality and ownership of the data in the relationships. By varying the basic symbols at the end of the line, you can arrive at all of the various possibilities for relationships. Table 2-3 shows the different symbols that can be employed to build relationship representations.

Table 2-3. *Information Engineering Symbols*

Symbol	Relationship Type	Description
	Many	The entity on the end with the crow's foot denotes that there can be greater than one value related to the other entity.
	Optional	This symbol indicates that there does not have to be a related instance on this end of the relationship for one to exist on the other. This relationship has been described as zero-or-more as opposed to one-or-more.
	Identifying	The key of the entity on the other end of the relationship is migrated to this entity.
	Nonrequired	A set of dashed lines on one end of the relationship line indicates that the migrated key may be null.

Figures 2-11 through 2-14 show some examples of relationships in IE.

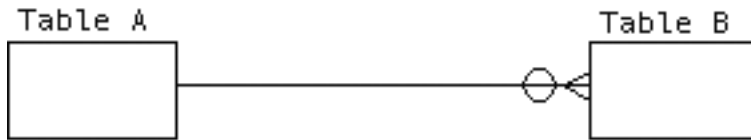


Figure 2-11. *One-to-many nonidentifying mandatory relationship*



Figure 2-12. *One-to-many identifying mandatory relationship*

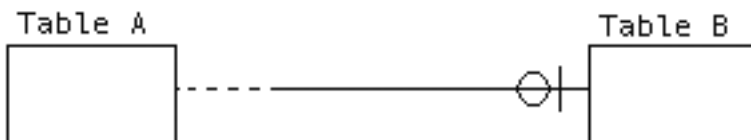


Figure 2-13. *One-to-one identifying optional relationship*

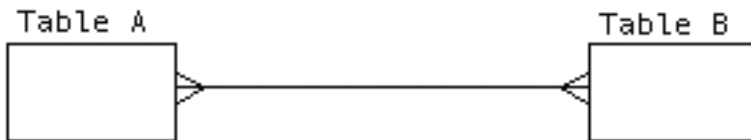


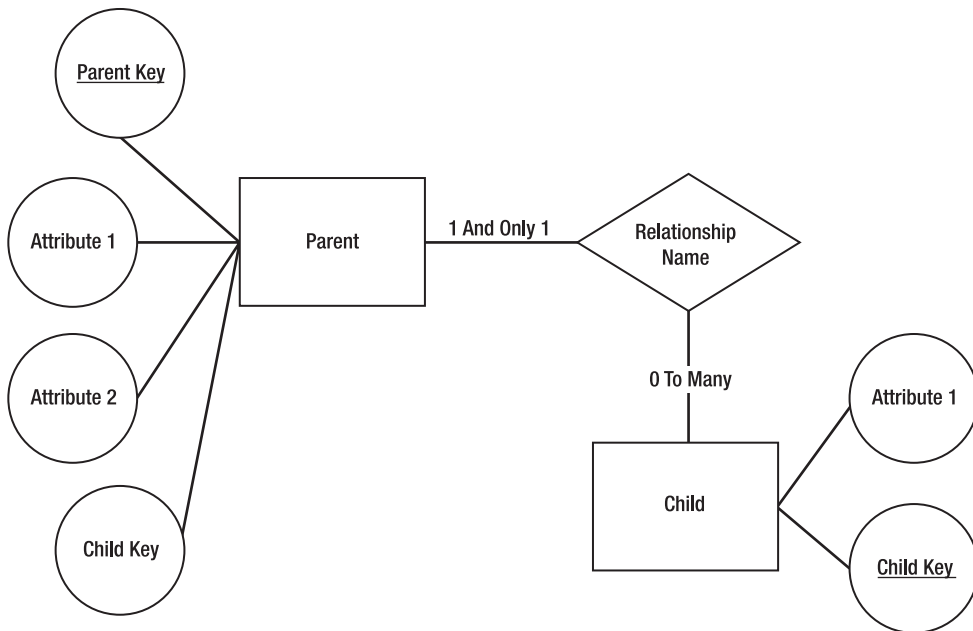
Figure 2-14. *Many-to-many relationship*

I have never felt that this notation was as clean as IDEF1X, but it does a very good job and is likely to be used in some of the documents that you will come across in your work as a data architect. IE is also not always fully implemented in tools; however, the circle and the crow's feet are generally implemented properly.

Tip You can find more details about the Information Engineering methodology in the book *Information Engineering, Books 1, 2, and 3* by James Martin (Prentice Hall, 1990).

Chen ERD

The Chen Entity Relationship Model (ERD) methodology is quite a bit different from IDEF1X, but it's pretty self-explanatory. You will seldom see this methodology anywhere other than in academic texts, but since quite a few of these types of diagrams are on the Internet, it's good to understand the basics of the methodology. Here's a very simple Chen ERD diagram:



Each entity is again a rectangle; however, the attributes are not shown in the entity but are instead attached to the entity in circles. The primary key either is not denoted or, in some variations, is underlined. The relationship is denoted by a rhombus, or diamond shape.

The cardinality for a relationship is denoted in text. In the example, it is *1 and Only 1 Parent rows <relationship name> 0 to Many Child rows*.

The primary reason for including the Chen ERD format is for contrast. Several other modeling methodologies—for example, Object Role Modeling (ORM) and Bachman—implement attributes in this style, where they are not displayed in the rectangle.

While I understand the logic behind this approach (entities and attributes are separate things), I have found that models I have seen using the format with attributes attached to the entity like this seemed overly cluttered, even for very small diagrams. The methodology does, however, do an admirable job with the logical model and does not overrely on an arcane symbology to describe cardinality.

Note You can find further details on the Chen ERD methodology in the paper “The Entity Relationship Model - Toward a Unified View of Data” by Peter Chen (<http://www.cobase.cs.ucla.edu/pub/dblp/html/db/journals/tods/Chen76.html>).

Note While I am not saying that such a tool does not exist, I personally have not seen the Chen ERD methodology implemented in a mainstream database design tool other than some versions of Microsoft Visio. Many of the diagrams you will find on the Internet will be in this style, however, so it is interesting to understand at least the basics of the Chen ERD methodology.

Management Studio Database Diagrams

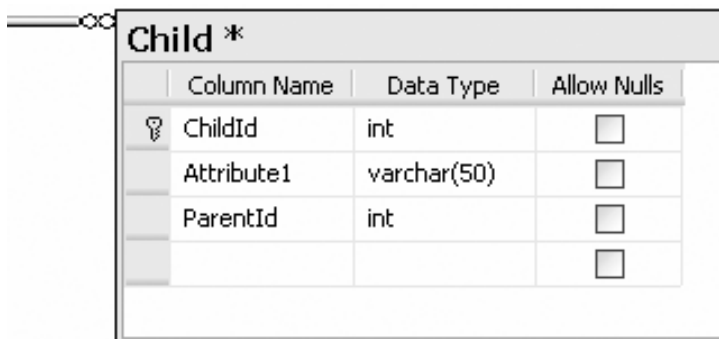
The database diagramming capability built into SQL Server 2005 is not really a good modeling tool, in my opinion. It can be a useful tool to view the structure of an existing database (a picture really *is* worth a thousand words!), but because it works directly against the implemented tables, it is not overly useful for design but only for the final implementation. You could do this design on an empty database, but it is seriously too clunky of a tool for design.

The following is an example of a one-to-many relationship in Management Studio:



The primary keys are identified by the little key in an attribute. The relationship is denoted by the line between the entities, with the one end having a key and the many end having an infinity sign.

You can display the entities in several formats by just showing the names of the entities or by showing all of the attributes with datatypes, for example:



While the database diagram tool does have its place, it isn't a full-featured data modeling tool and shouldn't be used as such if you can avoid it. I am including coverage of the SQL Server modeling methodology here because it's included in SQL Server and in some situations it's the

best tool you may have access to. It does give access to all implementation-specific features of SQL Server, including the ability to annotate your tables and columns with descriptive information. Unfortunately, if you decide to implement a relationship in a trigger, it will not know that the trigger exists. (I cover triggers in Chapter 6, so if you have no idea what a trigger is right now, don't worry.)

In most cases, the SQL Server tool isn't the optimal way to see actual relationship information that is designed into the database, but it does offer a serviceable look at the database structure when needed.

Note In the Business Intelligence tools for SQL Server 2005, there is also another tool that resembles a data model in the Data Source view. It is used to build a view of a set of tables, views, and (not implemented) queries for building reports from. It is pretty much self-explanatory as a tool, but it uses an arrow on the parent end of the relation line to indicate where a primary key came from. This tool is not pertinent to the task of building or changing a database, but I felt I should at least mention it briefly, as it does look very much like a data modeling tool.

Best Practices

The following are some basic best practices that can be very useful to follow when doing data modeling:

- *Entity names:* There are two ways you can go with these: plural or singular. I feel that names should be singular (meaning that the name of the table describes a single instance, or row, of the entity, much like an OO object name describes the instance of an object, not a group of them), but many other highly regarded data architects and authors feel that the table name refers to the set of rows and should be plural. Whichever way you decide to go, it's most important that you are consistent. Anyone reading your model shouldn't have to guess why some entity names are plural and others aren't.
- *Attribute names:* It's generally not necessary to repeat the entity name in the attribute name, except for the primary key. The entity name is implied by the attribute's inclusion in the entity. The chosen attribute name should reflect precisely what is contained in the attribute and how it relates to the entity. And as with entities, no abbreviations are to be used in the logical naming of attributes; every word should be spelled out in its entirety. If any abbreviation is to be used, due to some naming standard currently in place, for example, then a method should be put into place to make sure the abbreviation is used consistently, as discussed earlier in the chapter.
- *Relationships:* Name relationships with verb phrases, which make the relationship between a parent and child entity a readable sentence. The sentence expresses the relationship using the entity names and the relationship cardinality. The relationship sentence is a very powerful tool for communicating the purpose of the relationships with nontechnical members of the project team (e.g., customer representatives).

- *Domains*: Define domains for your attributes, implementing type inheritance wherever possible to take advantage of domains that are similar. Using domains gives you a set of standard templates to use when building databases that ensures consistency across your database and, if used extensively, all of your databases.
- *Objects*: Define every object so it is clear what you had in mind when you created a given object. This is a tremendously valuable practice to get into, as it will pay off later when questions are asked about the objects, and it will serve as documentation to provide to other programmers and/or users.

Summary

In this chapter I presented the basic process of graphically documenting the objects that were introduced in the first chapter. I focused heavily on the IDEF1X modeling methodology, taking a detailed look at the symbology that will be used through database designs. The base set of symbols outlined here will enable us to fully model logical databases (and later physical databases) in great detail.

The primary tool of a database designer is the data model. It's such a great tool because it can show the details not only of single tables at a time, but the relationships between several entities at a time. Of course it is not the only way to document a database; each of the following is useful, but not nearly as useful as a full-featured data model:

- Often a product that features a database as the central focus will include a document that lists all tables, datatypes, and relationships.
- Every good DBA has a script of the database saved somewhere for re-creating the database.
- SQL Server's metadata includes ways to add properties to the database to describe the objects.

I also briefly outlined the other types of model methodologies, such as Information Engineering, Chen ERD, and Microsoft Management Studio.

Now that we've considered the symbology required to model the database, I'll use data models throughout the book to describe the entities in the conceptual model (Chapter 3), the logical model (Chapter 4), and throughout the implementation presented in the rest of the book.

