Lesson 3: An introduction to data modeling

3.1 Introduction: The importance of conceptual models

Before you sit down in front of the keyboard and start creating a database application, it is critical that you take a step back and consider your business problem—in this case, the kitchen supply scenario presented in Lesson 2— from a *conceptual* point of view. To facilitate this process, a number of **conceptual modeling** techniques have been developed by computer scientists, psychologists, and consultants.

For our purposes, we can think of a conceptual model as a *picture* of the information system we are going to build. To use an analogy, conceptual models are to information systems what blueprints are to buildings.

There are many different conceptual modeling techniques used in practice. Each technique uses a different set of symbols and may focus on a different part of the problem (e.g., data, processes, information flows, objects, and so on). Despite differences in notation and focus, however, the underlying rationale for conceptual modeling techniques is always the same: understand the problem before you start constructing a solution.

There are two important things to keep in mind when learning about and doing data modeling:

1. Data modeling is first and foremost *a tool for communication*. Their is no single "right" model. Instead, a valuable model highlights tricky issues, allows users, designers, and implementors to discuss the issues using the same vocabulary, and leads to better design decisions.
2. The modeling process is inherently iterative: you create a model, check its assumptions with users, make the necessary changes, and repeat the cycle until you are sure you understand the critical issues.

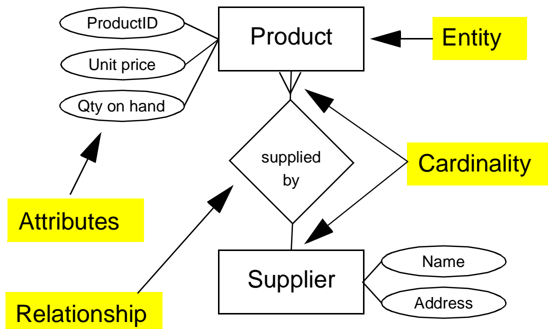
In this background lesson, you are going to use a **data modeling** technique—specifically, **Entity-Relationship Diagrams** (ERDs)—to model the business scenario from Lesson 2. The data model you create in this lesson will form the foundation of the database that you use throughout the remaining lessons.

© Michael Brydon (mjbrydon@sfu.ca)

3.1.1 What is data modeling?

A data model is a simply a diagram that describes the most important "things" in your business environment from a data-centric point of view. To illustrate, consider the simple ERD shown in Figure 3.1. The purpose of the diagram is to describe the relationship between the data stored about products and the data stored about the organizations that supply the products.

FIGURE 3.1: An ERD showing a relationship between products and suppliers.



3.1.1.1 Entities and attributes

The rectangles in Figure 3.1 are called **entity types** (typically shortened to "entities") and the ovals are called **attributes**. The entities are the "things" in the business environment about which we want to store data. The attributes provide us with a means of organizing and structuring the data. For example, we need to store certain information about the products that we sell, such as the typical selling price of the product ("Unit price") and the quantity of the product currently in inventory ("Qty on hand"). These pieces of data are attributes of the Product entity.

It is important to note that the precise manner in which data are *used* and *processed* within a particular business application is a separate issue from data modeling. For example, the data model says nothing about *how* the value of "Qty on hand" is changed over time. The focus in data modeling is on capturing data about the environment. You will learn how to change this data (e.g., process orders so that the inventory values are updated) once you have mastered the art of database design.

A data modeler assumes that if the right data is available, the other elements of the application will fall into place effortlessly and wonderfully. For now, this is a good working assumption.

3.1.1.2 Notation for relationships

In addition to entities and attributes, Figure 3.1 shows a **relationship** between the two entities using a line and a diamond. The relationship construct is used—not surprisingly—to indicate the existence or absence of a relationship between entities. A crow's foot at either end of a relationship line is used to denote the **cardinality** of the relationship.

For example, the crow's foot on the product side of the relationship in Figure 3.1 indicates that a particular supplier may provide your company with several different products, such as bowls, spatulas, wire whisks and so on. The *absence* of a crow's foot on the supplier side indicates that each product in your inventory is provided by a single supplier. Thus, the relationship in Figure 3.1 indicates that you always buy all your wire whisks from the same company.

3.1.1.3 Modeling assumptions

The relationship shown in Figure 3.1 is called **one-to-many**: each supplier supplies many products (where many means "any number including zero") but each product is supplied by one supplier (where "one" means "at most one").

The decision to use a one-to-many relationship reflects an assumption about the business environment in which your wholesale company operates. However, it is easy to imagine a different environment in which each product is supplied by *multiple* suppliers. For example, many suppliers may carry a particular brand of wire whisk. When you run out of whisks, it is up to you to decide where to place your order. In other words, it is possible that a **many-to-many** relationship exists between suppliers and products.

If multiple supplier exist, attributes of the product, such as its price and product number may vary from supplier to supplier. In this situation, the data requirements of a many-to-many environment are slightly more complex than those of the one-to-many environment. If you design and implement your database around the one-to-many assumption but then discover that certain goods are supplied by multiple suppliers, much effort is going to be required to fix the problem.

Herein lies the point of drawing an ERD: The diagram makes your assumptions about the relationships within a particular business environment explicit *before* you start building things.

3.1.1.4 The role of the modeler

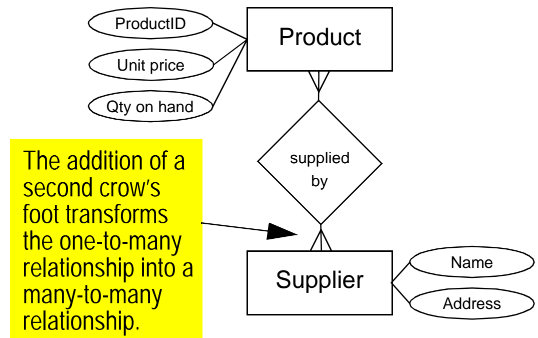
In the environment used in these tutorials, you are the user, the designer, and the implementor

of the system. In a more realistic environment, however, these roles are played by different individuals (or groups) with different backgrounds and priorities. For example, a common stereotype of implementors (programmers, database specialists, and so on) is that they seldom leave their cubicles to communicate with end-users of the software they are writing. Similarly, it is generally safe to assume that users have no interest in, or understanding of, low-level technical details (such as the cardinality of relationships on ERDs, mechanisms to enforce referential integrity, and so on). Thus, it is up to the **business analyst** to bridge the communication gap between the different groups involved in the construction, use, and administration of an information system.

As a business analyst (or more generally, a designer), it is critical that you walk through your conceptual models with users and make sure that your modeling assumptions are appropriate. In some cases, you may have to examine sample data from the existing computer-based or manual system to determine whether (for instance) there are any products that are supplied by multiple suppliers.

At the modeling stage, making changes such as converting a one-to-many relationship to a many-to-many relationship is trivial—all that is required is the addition of a crow's foot to one end of the relationship, as shown in Figure 3.2. In contrast, making the same change once you have implemented tables, built a user interface, and written code is a time-consuming and frustrating chore.

FIGURE 3.2: An ERD for an environment in which there is a many-to-many relationship between products and suppliers.



Generally, you can count on the 10× rule of thumb when building software: the cost of making a change increases by an order of magnitude for each stage of the systems development lifecycle that you complete.

3.1.2 Core modeling constructs and notation

Data modelers typically adopt a set of notational conventions so that their diagrams are consistent. For example, large IT organizations and consultancies typically adopt a **methodology**[1]—a set of tools and procedures for applying the tools that specifies the notation used within the organization. Enforcing standardization in this way facilitates teamwork on large projects. Similarly, if a **computer-aided software engineering** (CASE) tool is used for conceptual modeling and design, notational conventions are often enforced by the software.

What follows is a brief summary of the notational conventions that I use when drawing ERDs. Keep in mind, however, that ERDs are first and foremost a tool for communication between humans. As such, the precise notation you use is not particularly important as long as people can read and understand the diagrams. With experience, you will come to realize that differences in the shapes of the boxes and lines have little effect on the core concepts of data modeling.

[1] It can be argued that the term "method" is grammatically preferable. In Europe, for example, the term "method" tends to be favored.

3.1.2.1 Entities

Entities are drawn as rectangular boxes containing a noun in singular form, as shown in Figure 3.3.

FIGURE 3.3: An entity named "Customer".

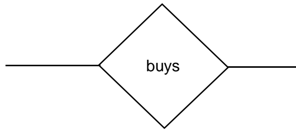
You will see later that each entity you draw ultimately becomes a table in your database. You might want to keep this transformation from entity to table in mind when selecting the names of your entities. For example, your entity names should be short but descriptive.

3.1.2.2 Relationships

A relationship between entities is drawn as a line bisected by a diamond. The diamond contains a verb (or short verb phrase) that describes the nature of the relationship between the entities, as shown in Figure 3.4.

Named relationships are used to make the ERDs more readable. However, unlike entity names, relationship names never show up in the final database. Consequently, it does not really matter how you label your relationships, as long

FIGURE 3.4: A relationship named “buys”.



as the labels make the diagram easier to interpret.

To illustrate, consider the relationship between products and suppliers shown in Figure 3.1. The relationship is described by the verb phrase “supplied by”. Although one could have opted for the shorter relationship name “has” instead, the resulting diagram (e.g., “Supplier has product”) would be more difficult for readers of the diagram to interpret.

3.1.2.3 Relationship direction

One issue that sometimes troubles neophyte data modelers is that the *direction* of the relationship is not made explicit on the diagram. Returning to Figure 3.1, it is obvious to me (since I drew the diagram) that the relationship should be read: “Product is *supplied by* supplier.” Reading the relationship in the other direction (“Supplier *is supplied* by product”) makes very little sense to anyone who is familiar with the particular problem domain.

Generally, ERDs make certain assumptions about the reader’s knowledge of the underlying business domain.

A notational convention supported by some CASE tools is to require two names for each relationship: one that makes sense in one direction (e.g., “is supplied by”), and another that makes sense in the opposite direction (e.g., “supplies”). Although double-naming may make the diagram easier to read, it also adds clutter (twice as many labels) and imposes an additional burden on the modeler.

3.1.2.4 Cardinality

As discussed in Section 3.1.1.2, the cardinality of a relationship constrains the number of instances of one entity type that can be associated with a single instance of the other entity type.

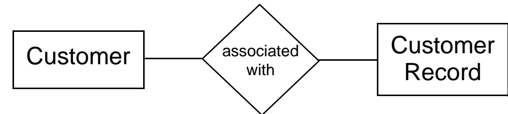
The cardinality of relationships has an important impact on number and structure of the tables in the database. Consequently, it is important to get the cardinality right on paper before starting the implementation.

There are three fundamental types of cardinality in ERDs:

- **One-to-many** — You have already seen an example of a one-to-many relationship in Figure 3.1. You will soon discover that one-to-many relationships are the bread and butter of relational databases.
- **One-to-one** — At this point in your data modeling career, you should avoid one-to-one relationships. To illustrate the basic issue, consider the ERD shown in Figure 3.5. Based on an existing paper-based system, the modeler has assumed that each customer is associated with one "customer record" (i.e., a paper form containing information about the customer, such as address, fax number, and so on). Clearly, each customer has only one customer record and each customer record belongs to a single customer. However, if we automate the system and get rid of the paper form, then there is no reason not to combine the Customer and Customer Record entities into a single entity called Customer.

In many cases, one-to-one relationships indicate a modeling error. When you have a one-to-one relationship such as the one shown in Figure 3.5, you should combine the two entities into a single entity.

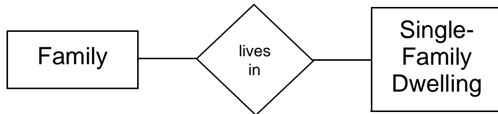
FIGURE 3.5: An incorrect one-to-one relationship



- **Many-to-many** — The world is full of many-to-many relationships. A well-used example is "Student takes course." Many-to-many relationships also arise when you consider the *history* of an entity. To illustrate, consider the ERD shown in Figure 3.6. At first glance, the relationship between Family and Single-Family Dwelling (SFD) might seem to be one-to-one since a particular family can only live in one SFD at a time and each SFD can (by definition) only contain a single family. However, it is possible for a family to live in different houses over time. Similarly, it is possible that many families inhabit a particular house over the years. Thus, if the concept of time is considered, the relationship becomes many-to-many.

We will discuss how you go about determining cardinality in subsequent sections. At this point, it is sufficient to recognize that there are two

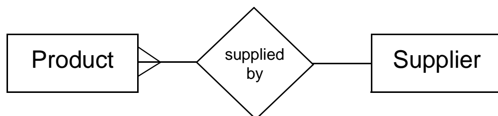
FIGURE 3.6: What is the cardinality of this relationship?



popular (and equivalent) approaches to denoting "one" and "many" on an ERD: the crow's foot notation you have already seen and the "1:N" notation.

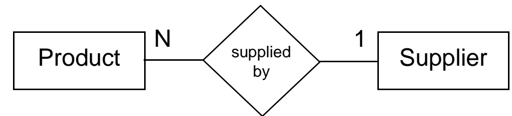
1. **Crow's foot notation** — In the crow's foot notation, three little lines (resembling a crow's foot) are used to indicate "many". Not surprisingly, the absence of a crow's foot indicates "one". Thus, the relationship in Figure 3.7 indicates that "each product is supplied by *at most* one supplier," whereas "each supplier may supply many products.".

FIGURE 3.7: A one-to-many relationship in crow's foot notation



2. **1:N notation** — In the 1:N notation, the symbol "N" (and/or "M") is used to indicate "many" whereas "1" is used to indicate "one". An example of the 1:N notation is shown in Figure 3.8.

FIGURE 3.8: A one-to-many relationship in 1:N notation



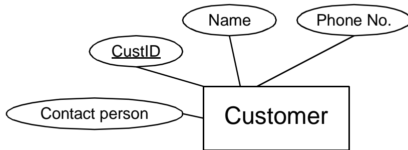
The ERD model also supports additional cardinality information in the form of cardinality constraints. To keep things simple, however, the discussion of cardinality constraints is deferred until Section 6.3.2.

3.1.2.5 Attributes

Attributes are properties or characteristics of a particularly entity about which we wish to collect and store data. In addition, there is typically one attribute that uniquely identifies particular instances of the entity. For example, each of your customers may have a unique customer ID. Such attributes are known as **key attributes**.

For ERDs that are drawn manually, attributes are traditionally shown as ovals containing the name of the attribute. If the attribute is a key, it is typically underlined. A number of attributes for the Customer entity are shown in Figure 3.9.

FIGURE 3.9: A number of attributes of the Customer entity are shown in ovals.



Adding attributes to ERDs can result in very cluttered diagrams. Some CASE tools list the attributes inside the entity rectangle (see the discussion of CASE tools in Section 3.4.5). Another way to reduce clutter is to only show a handful of critical attributes on the diagram.

3.2 Learning objectives

- understand the core constructs of the entity-relationship model
- create an ERD based on your understanding of a business scenario
- use associative entities to add attributes to relationships
- gain some familiarity with the role of data modeling and CASE tools in the development process

3.3 Exercises

In the sections that follow, we step through the construction of an ERD for the kitchen supply scenario. By following along, you should gain a better understanding of the basic techniques involved in data modeling as well as some of the design pitfalls that should be avoided.

If this lesson was on how to play golf, you would not read it and then assume that you are a good golfer. Golf is a skill that requires both theoretical knowledge and hours of practice. Thus, the only way to become a good golfer is to acquire a solid understanding of the fundamentals and then go out and hit thousands of balls. The same principles apply to data modeling.

3.3.1 Starting simple

Let us begin with the simplest and most essential statement one can make about the

wholesaling environment: *customers buy products*. It is natural that you would want to both automate and informate (recall Section 2.4) this important business process.

3.3.1.1 Step one: identify the entities

Entities are physical things, organizations, roles and events about which we want to store information. In the wholesaling scenario, two entities are immediately obvious: Customer and Product. However, before we add the entities to our diagram, it is important that we have a firm understanding of what *exactly* these entities correspond to in the real world:

1. **Customers** — In the wholesaling environment, a customer is an organization, not a person. There may be a single person at the organization through whom we conduct our business. We will refer to this person as the "contact person" for the customer in order to maintain a clean distinction between people and organizations.
2. **Products** — It is not immediately clear whether the Product entity refers to a specific item or a class of similar items. For example, one of the products you sell is the "Fat Cat" mug. The Product ID of the mug is "88 4017" and it normally sells for $5.50. Note, however, that there are many *individual* "Fat Cat" mugs and each one is slightly different due to irregularities, variations in painting, and so on. In our case, there are advantages to ignoring the individuality of each mug and treating them all as a single group of interchangeable items. Thus, when we talk about "a product" or "a SKU", we are talking about an *entire class* of similar instances, not individual instances themselves.[1]

Having made these assumptions explicitly, we can now create our first ERD.

➡ Take out a piece of paper and a pencil.

Unless you have a special-purpose CASE tool, it is seldom worth the effort to draw the early drafts of your conceptual models on a computer.

ERDs typically require many modifications so you should not invest much time making your diagrams look nice. In fact, the diagram you are about to begin will

---

[1] In some environments, it may be necessary to treat products as individual items. For example, in the aerospace industry, there is a requirement to track individual parts by serial number in case a part fails. The requirement for "unit effectivity" necessitates a different set of assumptions about the Product entity and thus leads to a different database design.

end up the in recycle bin by the end of the lesson.

- Add the Customer and Product entities to your diagram, as shown in Figure 3.10.

FIGURE 3.10: Add the first two entities to your ERD.

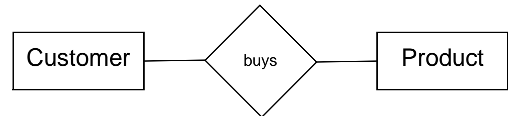
3.3.1.2 Step two: specify a relationship between the entities

We know that customers buy products and that products are bought by customers. It is a simple matter to create a relationship to communicate this fact.

- Add a relationship line between the Customer and Product entities.
- Label the relationship "buys", as shown in Figure 3.11.

Unlike flow charts, the arrangement of boxes and the direction of lines in an ERD have no significance—any arrangement that fits on the page is valid. Similarly, the relationship line does not denote any type of sequence or flow of information.

FIGURE 3.11: Add a relationship between the two entities.



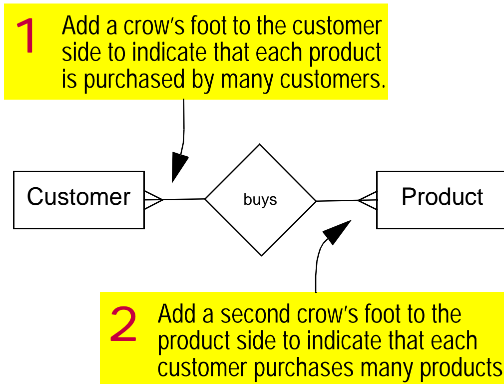
3.3.1.3 Step three: determine the cardinality of the relationship

Each customer can buy many products—indeed, that is the whole purpose of being in this line of business. To show this possibility on our ERD, we add a crow's foot to the Product side of the relationship line.

Similarly, each product can be purchased by many customers. For example, a number of our customers may chose to stock the "Fat Cat" mug (keeping in mind that the product refers to a style of mug, not an individual mug). As a result, a crow's foot is added to the Customer side of the relationship.

- Designate the "buys" relationship as many-to-many using the crow's foot notation, as shown in Figure 3.12.

FIGURE 3.12: Indicate the many-to-many cardinality of the relationship.



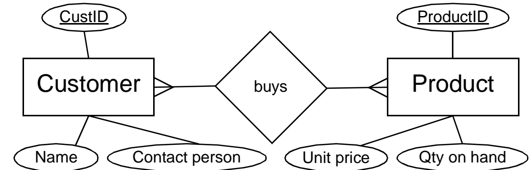
3.3.1.4 Step four: identify a few important attributes

There is a need to find a balance between the descriptiveness of the ERD and the ease with which others can decipher it. As such, I prefer to show only a handful of attributes on the ERDs.

➡ Add a small number of important attributes to your diagram, as shown in Figure 3.13.

By adding attributes such as "Qty on hand" to the Product entity, it is clear that the entity refers to a class of products, not an individual

FIGURE 3.13: An initial ERD for the kitchen supply environment.



product. In contrast, if "Serial number" were added to the diagram as an attribute instead of "Qty on hand", the reader of the ERD would come to a different conclusion about the meaning of the Product entity.

3.3.2 Dealing with many-to-many relationships

Although the diagram in Figure 3.13 is technically correct, it is missing a great deal of information about how a purchase transaction occurs in reality.

To illustrate, consider the attribute "Unit price" belonging to the Product entity. Unit price contains the *default* selling price of the product. To understand why it is the default price, consider the case of a "Fat Cat" mug that typically sells for $5.50. What if there is a

particular mug that has a minor flaw? Although the mug can still be sold, the customer may expect a discounted price to compensate for the flaw. The question is therefore: Where on the diagram do we indicate the *actual* selling price of a particular mug?

A second example is the purchase quantity. What if the customer purchases a dozen "Fat Cat" mugs? Where is this information recorded? The "Qty ordered" attribute does not belong to the Customer entity because customers order many products besides mugs. Similarly, "Qty ordered" does not belong to the Product entity because different customers may order different quantities.

This is a problem that typically arises in many-to-many relationships: There are certain important attributes that do not seem to belong to either of the entities participating in the relationship. The solution is to assign the attributes to the relationship itself.

3.3.2.1 Attributes of relationships

The issues surrounding price and order quantity arise because the attributes belong to the *interaction* of the entities in the many-to-many relationship. Thus, the price of a flawed mug is an attribute of a particular product being sold to a particular customer on a particular day.

To summarize, there are a number of attributes that should be attached to the "buys" relationship in Figure 3.13:

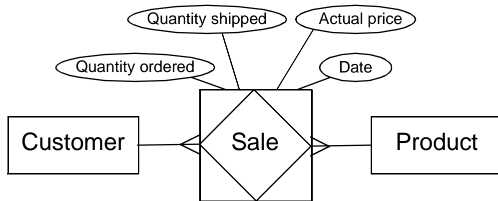
- **Date** — the date on which the purchase is made;
- **Actual price** — the price at which the item (or multiple items within the same class of products) are actually sold to the customer;
- **Quantity ordered** — the number of items with a certain product ID requested by the customer; and,
- **Quantity shipped** — the actual number of items shipped to the customer.

3.3.2.2 Associative entities

Given the number and importance of the attributes attached to the "buys" relationship, it makes sense to treat the relationship as an entity in its own right. To transform a relationship into an entity on an ERD, we use a special symbol called an **associative entity**. The notation for an associative entity is a relationship diamond nested inside of an entity rectangle, as shown in Figure 3.14.

To transform your many-to-many relationship (without attributes) into an associative entity (with attributes), do the following:

FIGURE 3.14: Transform a many-to-many relationship into an associative entity.



➡ Draw a rectangle around the “buys” relationship.

➡ Replace the relationship name “buys” with an appropriate noun, for example “Sale”.

Remember, entities—including associative entities—are named with nouns.

➡ Decompose the many-to-many relationship into two one-to-many relationships.

➡ Add the attributes to the associative entity, as shown in Figure 3.14.

Although Sale is now treated as an entity, there is no requirement to add relationship diamonds between Product and Sale or Customer and Sale. An associative entity serves as an entity *and* a relationship at the same time.

The meaning of the Sale associative entity in Figure 3.14 is the following: Each customer can be involved in many sales transactions, but each individual sales transaction involves only one customer. Similarly, each product can be involved in many sales transactions, but each sales transaction involves only one type of product.

3.3.2.3 Illustration

To better understand how an associative entity works, it is worthwhile to jump ahead a bit and consider what the data might look like. In Figure 3.15, sample data for the Customer, Product, and Sale entities are shown. There are a couple of interesting things to notice about the sample data:

1. Each entity contains information relevant to that entity only. For example, Customer only contains information about customers; Product only contains information about products, and so on.
2. Each row in the Sale entity shows the details of a single sales transaction. Each sales transaction consists of a particular product being sold to a particular customer. The transaction-specific information (such

FIGURE 3.15: Data showing the role of an associative entity

Customer

Customer ID	Customer name
1	Sam's Stock Pot
2	Loonie Mart #107
3	Rosch Dry Goods Inc.
4	Gadgets "R" Us
5	The Chef's Assistant

Each customer can participate in many sales transactions.

Product

Product ID	Description	Unit price	Quantity on hand
51 5012	Water jug, s.s. w/ice guar	$23.50	36
57 3826	Spatula, 6" "Cuisipro"	$4.00	65
57 3828	Spatula, 8" "Cuisipro"	$4.25	20
57 4966	Mixing bowl, 16 qt.	$12.50	7
57 551	S.S. salad server set	$3.15	32
71 12101	S.S. soup ladle	$5.25	49
71 12110	S.S. skimmer	$5.00	32
71 12111	S.S. sauce ladle	$5.25	9
71 12114	S.S. grave ladle with spou	$4.75	56
74 4042	Snail plate w/white handle	$3.15	36
74 4321	Pastry brush, 1"	$4.00	32
74 4539	Meat tenderizing hammer	$2.50	12
74 6083	Spring form pan, 9" non st	$7.50	8

Each product can participate in many sales transactions.

Sale

Customer ID	Product ID	Actual Price	Qty Ordered	Qty Shipped
3	51 5012	$23.50	12	10
1	51 5012	$22.80	1	1
3	57 3828	$4.25	48	48
2	71 12101	$5.25	12	12
3	71 12101	$5.25	24	0

as the actual selling price and quantity ordered) are attributes of the Sale entity.

3. By using the data for the Sale entity, it is possible to determine which products have been purchased by a particular customer. Similarly, it is possible to determine which customers have purchased a particular product.

4. Only the minimum amount of information required to identify the customer and product is included in the Sale associative entity. For example, neither the name of the customer or the description of the product appear in Sale since this information can easily be found elsewhere using the values of Customer ID and Product ID respectively. In the context of

the Sale entity, the Customer ID and Product ID attributes are called **foreign keys** (foreign keys are so important that Lesson 6 is devoted to the topic).

3.3.3 Revising the ERD

There is an important problem with the ERD as its now stands. The constraint that each sales transaction involves only a single product appears to be at odds with the reality of the business situation described in Lesson 2. For example, the “sales transactions” with which we are most familiar—the customer orders you receive by fax— typically request many different products: a dozen “Fat Cat” mugs, two dozen spatulas, some wire whisks, and so on. The mismatch between the diagram and the business environment means that the ERD must be revised.

3.3.3.1 Identifying the problem

The problem with the ERD in Figure 3.14 is that ignores the *technology* (broadly speaking) used by customers to place orders. Specifically, customers normally wait until they need enough stock to make an order worthwhile. In addition, factors such as minimum order values and shipping costs favor the *batching* of small, single-product orders into large, multi-product orders.

By taking the technology used for ordering into account, it becomes clear that we have failed to model an important **event entity**: the arrival of an order.

Be careful—not all pieces of paper in the existing business process are automatically event entities. For example, the invoices that we send to our customers are more properly thought of as *reports* (which can be generated from the information already contained in other entities). With practice, the distinction between entities and non-entities will become clear.

3.3.3.2 Adding the new entity

In this section, you are going to modify your ERD to include an Order entity.

- Create a new ERD consisting of entities for customers, orders and products.

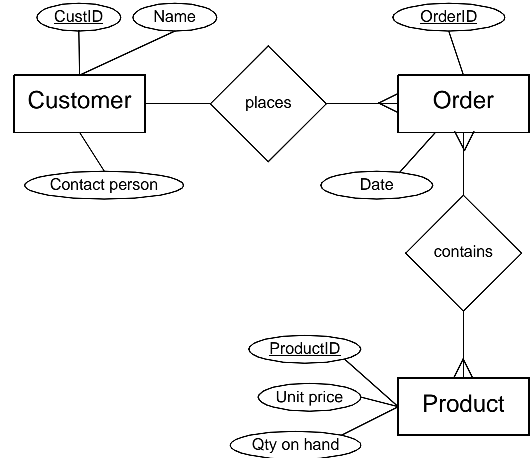
Here is where a CASE tool pays off: you can delete entities or move entities around the screen and the relationships and connector lines follow automatically.

- Create relationships to reflect the fact that customers place orders and orders consist of products.

- Add cardinality symbols to reflect the fact that each customer can place many orders, but each order belongs to a single customer.
- Add cardinality symbols to reflect the fact that each order can contain many products and each product may be contained in many orders.
- Add a handful of attributes to the diagram to help clarify the meaning of the entities.

The resulting ERD is shown in Figure 3.16.

FIGURE 3.16: Revise the ERD to include an Order entity.



3.3.3.3 Creating OrderDetails

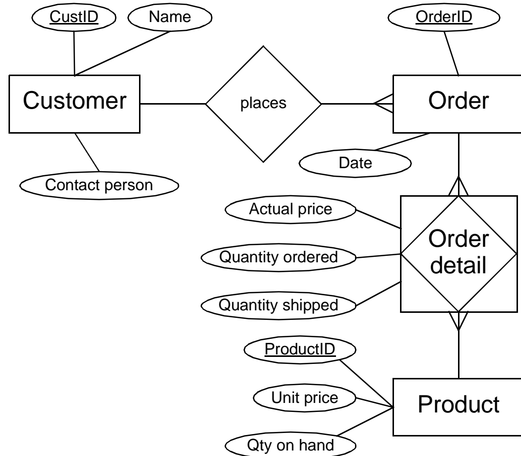
Although Figure 3.16 is a great improvement over our previous ERD, much of the same information missing from Figure 3.13—such as actual price and quantity ordered—is missing from the new ERD. As a consequence, we must transform the “contains” relationship into an associative entity with its own attributes.

- Transform the “contains” relationship into an associative entity using the procedure described in Section 3.3.2.2.

To remain consistent with MICROSOFT’s sample databases, I recommend using the name “Order Detail” for the associative entity. Alternatives include “Line Item” and “Order Item”.

The resulting ERD is shown in Figure 3.17. To help understand the relationship between the Order entity and the Order Detail associative entity, look at the orders included in the project package. Each order has header information (such as customer, order date, and so on) and multiple order details. Each detail has information such as quantity ordered, quantity shipped, and price.

FIGURE 3.17: Create an associative entity to model individual order details.



3.4 Discussion

3.4.1 Logical versus physical models

A distinction is typically made between **logical** and **physical** data models:

- **Logical data models** — logical models capture general information about entities and relationships and are used for communication with business users.
- **Physical data models** — physical models serve as a precise specification for the implemented system. As a consequence, the models must take into account the technology used to store the data. For example, a given logical data model translates into very different physical data models depending on whether the target technology is a file-based system, a relational database, or an object-oriented database.

Normally, you start with a high-level logical model and refine with the help of users over several iterations. Once you are happy with the logical model, you transform it into a physical model and hand it to a **database administrator** (DBA) for implementation as a database.

For relational databases, the translation process from logical to physical is relatively straightforward and involves the following steps:

1. **Decompose all many-to-many relationships** — Since the relational database model does not support many-to-many relationships, you must replace all many-to-many relationships with associative entities as described in Section 3.3.2.2.

2. **Add attributes** — The data model should be "fully attributed" before handing it over to a DBA.
3. **Identify primary keys** — Each entity requires an attribute that uniquely identifies instances.
4. **Add foreign keys** — In relational databases, relationships between entities are implemented using foreign keys.

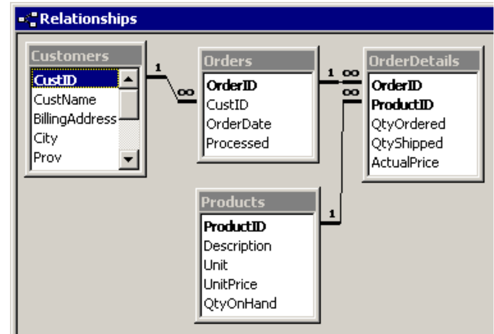
At this point, it is not critical that you understand each of these steps (you will get lots of practice in subsequent lessons). What is important is that you understand that there is a clear progression from high-level, graphical, conceptual models to low-level database schemas.

3.4.2 ERDs versus the ACCESS relationship window

If you have used the relationship window in MICROSOFT ACCESS, you know that the relationship diagrams used by ACCESS resemble ERDs. The primary differences between the two are

- The relationships between tables are not named in ACCESS.
- Many-to-many relationships are not supported. That is, the relationships window permits physical data models only.
- ACCESS uses the symbols "1" and "∞" instead of "1" and "N", as shown in Figure 3.18.

FIGURE 3.18: The relationship window in ACCESS uses a notation similar to that used in ERDs.



The ACCESS relationship window supports physical data modeling. One-to-many relationships are denoted using "1" and "∞".

3.4.3 Why do I need to know about data modeling?

Clearly, you do not *need* to know the intricacies of data modeling to start using a database package such as MICROSOFT ACCESS. In fact, MICROSOFT has gone through great lengths

between the release of ACCESS version 2.0 and ACCESS 2000 to make the product more accessible to data modeling neophytes. For example, there is a table analyzer, a table design wizard, and all sorts of other aids intended to automate the database design process. In my view, there are three problems with MICROSOFT's "dumbing-down" strategy:

1. No wizard or add-in tool is going to change the fact that database management systems (DBMSs) are specialized software packages that presuppose an enormous amount of prior knowledge. Even the error messages generated by ACCESS (as you will soon discover) can only be understood if you have a firm grasp on the theoretical fundamentals of the relational database model. For example, what do you do when you accidentally violate a "referential integrity constraint"? What is a "primary key"? What is an "ambiguous outer join"?
2. Trends in application development increasingly emphasize data and de-emphasize programming. Thus, a solid understanding of your data is critical. Put another way, just about anyone can build a reasonably sophisticated system if the underlying database is well designed (indeed, by doing these tutorials, you will see just how far you can go without writing a single line of programming code). In contrast, if the database design is poor, you will have to be a programming wizard just to create the illusion that the application works.
3. CASE tools from vendors such as ORACLE, COMPUTER ASSOCIATES, VISIO (now part of MICROSOFT), and many others translate ERDs directly into database tables. Thus, if you know how to draw diagrams similar to the one shown in Figure 3.17, you know how to design databases.

In short, the last thing you want to do is rely on wizards to shelter you from the database design process. Instead, you want to get in at the nitty-gritty level and understand the trade-offs between various designs. Once the design is complete you can use wizards and shortcuts for everything else.

3.4.4 How do I learn about data modeling?

One important problem that I perceive as a university professor is that despite the importance of data modeling, it is very difficult to find good practical training as a data modeler.

In the standard computer science database course, we tend to focus on theoretical issues such as relational algebra, set theory, indexing, normalization, and so on. Although such knowledge is certainly important for DBAs and

other technical professionals, it provides little guidance when we are faced with real-world modeling problems.

Conversely, the introductory information systems (IS) courses that we offer in business schools provide only cursory treatment of data modeling and database design. I suppose the rationale is that it should be possible to hire a computer science graduate to do the data modeling!

3.4.5 CASE tools and the design process

Computer-aided software engineering (CASE) tools are software packages that simplify the process of creating conceptual models. In addition, some CASE packages are more than just drawing tools: they translate the data models into database tables, programming code templates, and so on.

To illustrate, consider the diagram in Figure 3.19 which was drawn using the “database modeling tool” in VISIO ENTERPRISE. The database modeling tool allows a designer to start with a a physical ERD-like diagram and add implementation-level metadata (data about data). For example, in Figure 3.20, the physical-level properties of the `ActualPrice` attribute are specified in a dialog box.

Once the physical-level metadata has been added to the model, many CASE tools can generate table schemas for the target database. For example, ORACLE DESIGNER can generate structured query language (SQL) data definition commands for ORACLE databases. VISIO ENTERPRISE can generate SQL or create the tables in various database packages (including ACCESS) directly.

Given CASE tools with this type of functionality, it is clear that the most important skill for database designers is not a complete knowledge of SQL syntax. Instead, it is the ability to analyze a real-world problem and create a data model that accurately and elegantly captures the critical elements of the problem.

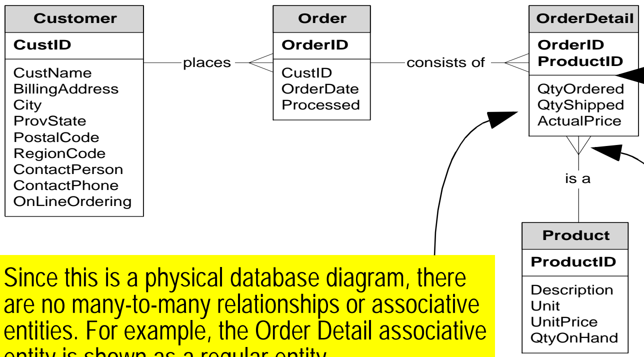
3.5 Application to the project

➡ Complete your own ERD for the order entry scenario. Remember that at this point, the scope of the project is very limited. It will be expanded somewhat in subsequent lessons.

HINT: If you get stuck, you can refer to the VISIO diagram in Figure 3.19. Keep in mind, however, that Figure 3.19 is a *physical* ERD; you should be working with *logical* ERDs at this stage of the development process.

FIGURE 3.19: An physical database model for the kitchen supply environment created using a CASE tool.

VISIO ENTERPRISE can be used to draw various styles of logical and physical data models. This is an example of a physical model.



Attributes are shown within the body of the entity and primary keys are shown in bold.

Since this is a physical database diagram, there are no many-to-many relationships or associative entities. For example, the Order Detail associative entity is shown as a regular entity.

Crow's foot notation is used indicate the cardinality of the relationships.

FIGURE 3.20: A CASE tool can be used to add implementation details to a graphical model.