

6.1 Introduction: Extending the scope of your system

In [Section 5.3.5](#), you created a new table called **Regions**, even though you do not have a Region entity on the entity relationship diagram (ERD) that you created in [Lesson 3](#).

The decision to add a **Regions** table was based on user interface criteria rather than data modeling criteria. Specifically, we decided to use region codes rather than the full region names to minimize the possibility of input errors. However, in order to remember what the one-letter region codes mean, we identified the requirement for a look-up table called **Regions**.

You will probably create a lot of look-up tables when you implement your databases. Look-up tables always have the same format:

- a short code or numeric ID that is appropriate for use as a value in other tables; and
- a description or name for the code that can be “looked up” so that users do not have to interpret (or even see) the short codes.



Some designers like to include look-up tables as entities on their ERDs. Doing so

maintains a consistent mapping between entities and tables. Personally, I find this practice clutters the ERDs and creates a lot of extra work for the person creating the diagram. Instead, I keep the “obvious” look-up tables off the ERDs, but add them to the database as required. Naturally, if you are using a CASE tool for physical design and to generate your database, you need to include all the entities on the diagram.

6.1.1 Death, taxes, and scope creep

Now that you have a table called **Regions** implemented, it might occur to someone (in this case, the someone is you wearing your user/manager hat) that it would be a good idea to store information about the sales representatives who are responsible for each region. In this way, you can use the order entry system to create reports showing performance-related information such as total sales for each sales rep, annual change in sales for each rep, and so on.

Adding new requirements and functionality to an information system once implementation is a common form of **scope creep**. In this case, you



want to expand the scope of the order entry system by including information about sales people. This is not an unreasonable demand. Since we already have the **Regions** table in place, it is a simple matter of associating each region with its assigned representative.



The next link in the scope creep chain of reasoning is: “Since we have employee information, we might as well add payroll and benefits information.” At some point, scope creep leads to a project that is too large and too complex to ever be completed. As a consequence, it is usually a good idea to resist scope creep and get a small system up and running before expanding it to address other problems and include “nice-to-have” features.

6.1.2 Leveraging existing data assets

To make the addition of sales rep information to the order entry system interesting, we are going to assume that you already have all your employee information in electronic form in an off-the-shelf payroll system that you bought a few years ago. Furthermore, we are going to assume that you do not know much about the payroll system, except that it contains information about all your employees (including sales reps) and that a great deal of effort is routinely expended to ensure that the

employee information in the payroll system is up to date. Given the turnover you experience with sales reps, you have no interest in storing and maintaining this data in two separate systems.

6.1.3 The relationship between customers, regions, and employees

Before we dive into implementation issues, it is worthwhile to return to our ERD and ensure we understand how salespeople fit into the grand scheme of operations in your kitchen supply company.

The current policy within the company is to assign each customer to one of your six sales regions (North, South, etc.). Each region can contain multiple customers, but customers never belong to more than one region.

Each sales region is the responsibility of a single sales rep. That is, if something is wrong in a region, there is a single employee who acts as the point of contact. In some cases, however, a single sales rep can be assigned responsibility for more than one region. You try to avoid this situation, but it arises from time to time when sales reps leave the company on short notice, go on maternity leave, and so on.

Later in this lesson, you will modify your ERD from [Lesson 3](#) to include these entities and relationships.



6.1.4 The infrastructure for one-to-many relationships

As discussed in [Section 3.1.2.4](#), one-to-many relationships are the bread and butter of relational databases. Not only do one-to-many relationships occur frequently in business contexts, but all many-to-many relationships must ultimately be decomposed into one-to-many relationships for implementation using a relational database system.



The relational database model cannot represent many-to-many relationships directly. Instead, you must transform each many-to-many relationship into an associative entity, as discussed in [Section 3.3.2.2](#).

The procedure for implementing a one-to-many relationship in a relational database is always the same: you take the primary key from the table on the “one” side of the relationship and include it in the table on the “many” side of the relationship.

To illustrate, recall the **Customers** and **Regions** tables you created in [Lesson 5](#): Look-up tables are always on the “one” side of a one-to-many relationship. In this case, each customer can belong to (at most) one region, but each region can contain many customers. To implement the

one-to-many relationship in the tables, we do the following:

1. Identify the primary key from the “one” side of the relationship—in this case, it is the **RegionCode** field in the **Regions** table.
2. Add a field with the same data type as the primary key to the table on the many side of the relationship—in this case, the **Customers** table.

The field **Customers.RegionCode** is called a **foreign key**. It is “foreign” in the sense that it is the primary key of the **Regions** table, not the **Customers** table.



It is common practice to express table names and field names using “dot” notation: **<table name>.<field name>**. In this way, it is possible to distinguish between fields with the same name in more than one table (e.g., **Regions.RegionCode** and **Customers.RegionCode**).

The operation of the foreign key is shown in [Figure 6.1](#): If we know that SAM’S STOCK POT has been assigned to the region with **RegionCode = “C”**, then it is a simple matter to go to the **Regions** table and look up the corresponding record. The values of **RegionCode** are guaranteed to be unique in the **Regions** table so there can be no confusion or



ambiguity regarding the region to which SAM'S STOCK POT has been assigned.

FIGURE 6.1: The **RegionCode** field in the **Customers** table is a foreign key. It permits us to associate a region with every customer.

Customer ID	Customer name	RegionCode	Contact person
▶ 1	Sam's Stock Pot	C	Sam Wong
2	Loonie Mart #107	R	Bill Williams
3	Rosch Dry Goods Inc.	K	Alice McRorie
4	Gadgets "R" Us	C	Leslie Cranfield, Innes
5	The Chef's Assistant	N	Andre Oulett
* (AutoNumber)			

RegionCode	Region
C	Central
E	East
K	Key Accounts
N	North
S	South
▶ W	West
*	

As you will see in the lessons on relationships ([Lesson 7](#)) and join queries ([Lesson 10](#)), the look-up procedure described above is automated by the DBMS. That is, you never have to manually make the connection between the foreign key in one table and the primary key in another. This is one of the core strengths of the relational database model.

6.2 Learning objectives

- understand how to implement one-to-many relationships using foreign keys
- update your ERD to include cardinality constraints

- modify an existing database to include foreign keys
- understand the concept of a weak entity

6.3 Exercises

6.3.1 Adding new entities

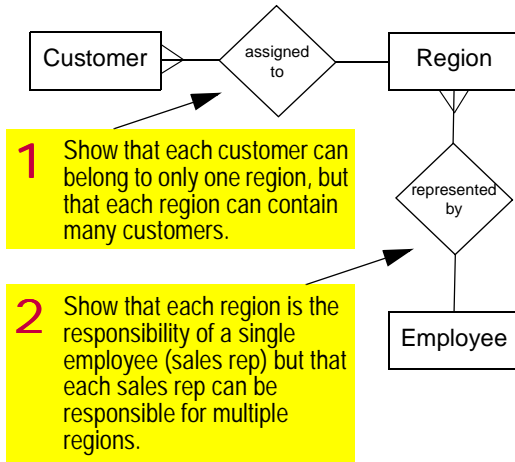
The first step in adding employee information to the database is to update your ERD.

- ➔ Add two new entities to your ERD: Region and Employee.



- ➔ Create the one-to-many relationships shown in Figure 6.2.

FIGURE 6.2: Update your ERD to show the relationship between Customer, Region, and Employee.



The cardinality of the relationships in Figure 6.2 follow directly from the company policies outlined in Section 6.1.3. For example, the constraint that each region is represented by at most one sales rep is an idiosyncratic feature of this particular company, not an immutable law of the

universe. Another company might organize its sales function differently and therefore require a different set of modeling assumptions.

6.3.2 Cardinality constraints

You may be wondering about the choice of the name “Employee” instead of “Sales Rep” in Figure 6.2. You know that your payroll system contains information about all your employees. As such, you can assume that the Employee entity has already been implemented within the organization. It makes sense to reuse this entity and retain any existing naming conventions rather than introduce an entirely new entity called “Sales Rep”.

Using the Employee entity leads to a different problem: not all Employee entities participate in a relationship with the Regions entity. That is, only employees belonging to the sales function of the firm are assigned sales regions. Unfortunately, the ERD in Figure 6.2 does make the distinction between salespeople and other employees explicit.

6.3.2.1 Mandatory versus optional participation

In the Entity-Relationship model, a construct called **cardinality constraints** can be used to differentiate between “mandatory” and “optional” participation in a relationship. To

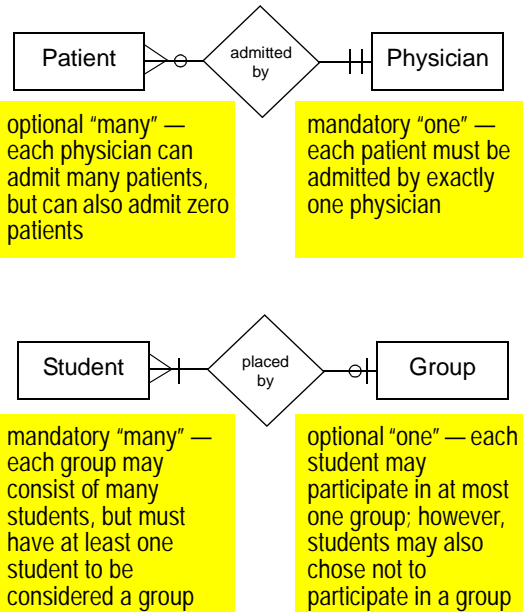
this point, all we have said about the relationship between the Employee and Region entities is that each employee can represent “many” regions. However, we may want to use ERD notation to indicate that “many” includes zero—that is, it is okay for an employee (e.g., the warehouse manager) to have no sales region.


There are other situations, however, in which we want to indicate mandatory participation in a relationship. For example, in a hospital environment, each patient is admitted by exactly one physician (that is, one physician takes initial responsibility for the patient). It is never the case the more than one physician admits a patient and it is certainly never the case that a patient is admitted without being assigned to a physician.

6.3.2.2 ERD notation

One notational convention for indicating cardinality constraints on ERDs is shown in Figure 6.3 (there are many other notations, but the underlying concept is always the same). Although including the little lines and circles on the relationships makes the ERDs more complex and harder to explain to users and managers, the designation of a relationship as “mandatory” or “optional” can have important consequences during implementation.

FIGURE 6.3: Examples of “optional” or “mandatory” participation in relationships.



 When you give your physical ERDs to implementors to build, you should make sure that all your design decisions are explicit. For example, you do not want a contract database developer working on a



hospital information system deciding on her own whether it is okay for a patient to be admitted without a doctor's order.

6.3.2.3 Adding cardinality constraints

- ➔ Use the notation shown in [Figure 6.3](#) to show that each region must have exactly one employee representing it, but that all employees do not necessarily participate in a sales rep role.

6.3.3 Adding foreign keys

It might not be immediately obvious, but the status of the Region entity has changed. Initially, information about regions was implemented as a simple look-up table and thus the Region entity was not included as a *real* entity on the ERD. But now that the link between employees and customers is being made, Region is a *bona fide* entity—that is, it corresponds to something identifiable and important in the domain we are modeling.

Fortunately, the foreign key infrastructure between the **Customers** and **Regions** tables is already in place because the **Customers** table already contains a the **RegionCode** foreign key. However, the same is not true of the foreign key infrastructure between the **Regions** and **Employees** table. Since **Regions** is on the many side of the relationship, it requires a foreign

key. But what is the primary key of the **Employees** table?

To make things interesting, we are going to assume that the primary key of the **Employees** table in the payroll application is the combination of the employee's first names (a field we know is called **emp_fname**) and last name (a field we know is called **emp_lname**). Of course, we recognize that this particular combination of fields is a poor choice for a concatenated primary key because it is (a) not guaranteed to be unique, and is (b) long and unruly.

Despite the obvious shortcomings of our primary key, we will stick with it for now and make changes as we learn more about the structure of the payroll data in [Lesson 8](#).

- ➔ Open the **Regions** table in design mode and add two fields: **emp_fname** and **emp_lname**.
- ➔ Set the data type of the fields to text and the length to some suitably large value (e.g., 100).



At this point, we do not know enough about the structure of the two fields in the **Employees** database to match the data type and lengths exactly. Thus, we will make educated guesses for now and make modifications as required.



➔ Since the use of `emp_fname` and `emp_lname` as a foreign key is provisional, do not spend any time specifying optional field properties.

➔ Save and close the `Regions` table.

You now have the foreign key infrastructure for creating relationships between customers, regions and employees. In [Lesson 7](#), you will learn how to make the relationships explicit in ACCESS.

6.4 Discussion

6.4.1 Concatenated keys

In this lesson, you added a **concatenated foreign key** in the `Regions` table. A concatenated foreign key is identical to a single-field foreign key except that multiple fields are used to make the link between the “one” and “many” sides of the relationship.

To illustrate why a concatenated key is used, consider the Region and Employee entity: If the employee in charge of the East region is Bill Williams, then the field `emp_fname` is probably insufficient as a foreign key since there may be more than one employee named “Bill” in the organization. However, at this point in time, it turns out that the combination of `emp_fname` + `emp_lname` (e.g., “Bill Williams”) is unique in

the `Employees` table. Of course, if you hire Bill’s son, Bill Jr. to work in the warehouse, your system will break and need to be fixed.



You should take some care when selecting the primary keys for you table. If existing fields cannot be guaranteed to be unique in the long term, you should opt for a surrogate key such as an AutoNumber.

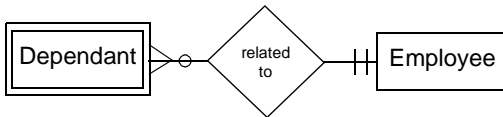
6.4.2 Weak entities

Consider the relationship between the Dependant and Employee entities shown in [Figure 6.4](#). Knowing who an employee’s dependants are is important for benefits, such as health and dental insurance. However, if a particular employee leaves the company, then the Dependant entities associated with that employee cease to be of interest to the organization (from an information system perspective).

In this example, Dependent is called a **weak entity**. It is *weak* in the sense that each instance of the entity relies on an instance of another entity (in this case, Employee) for existence in the database. If the *strong* instance is removed from the database, then all corresponding instances of the weak entity should also be removed from the database.



FIGURE 6.4: An ERD for employee benefits.
Note that **Dependant** is a weak entity.



Weak entities are often designated with a double rectangle.

There is one feature of weak entities that often causes confusion when determining foreign keys: the foreign key of a weak entity is also *part of its primary key*.

To illustrate, assume the table schema for the **Dependants** table is as follows:

Dependants(emp_id, index, name, relationship, date_of_birth, ...)



The standard way to write a table schema is **Table name**(*key field*₁, *key field*₂, ..., *field*_{n-1}, *field*_n). The field or fields that make up the primary key are typically underlined.

To make the table schema more concrete, assume that we have the following information about the dependents of two employees,

Russell Plevy (**emp_id** = 8) and Vivian Peng (**emp_id** = 2):

TABLE 6.1:

emp_id	index	name	relationship
8	1	Alica Marie Plevy-Jones	wife
8	2	Russell James Plevy	child
8	3	Susan Ann Plevy	child
2	1	Martin Alec Peng	husband

The combination of Russell's employee ID and the index number 2 provides a convenient means of uniquely identifying Russell Jr. Thus, **emp_id** is half of the table's concatenated primary key in addition to being the foreign key that links the **Dependants** and **Employees** table.

6.5 Application to the project

Based on what you have learned in this lesson and [Lesson 5](#), you are now ready to add some more tables to your database:

- ➔ Add an **Orders** table to store the details of customer orders that you receive.



- ➔ Ensure you use an AutoNumber for the **Orders.OrderID** field. In this way, Access will automatically generate a unique **OrderID** every time you create a new order.
- ➔ Create an **OrderDetails** table that uses **OrderID** as a foreign key (each order detail belongs to exactly one order, but each order can have many order details).



Since it is a foreign key, each value of **OrderDetails.OrderID** must refer to an existing value of **Orders.OrderID**. Thus, you cannot set **OrderDetails.OrderID** as an AutoNumber because doing so would create a new value of **OrderID** each time a new *order detail* is added. This is not what you want.

HINT: For a particular order, each product should appear only once as an order detail. Thus, rather than having one order detail for six “Fat Cat” mugs and a separate order detail for a dozen more “Fat Cat” mugs a few lines down in the same order, a single order detail for 18 should be used to consolidate the two. Given this policy, it is possible to set the primary key for the **OrderDetails** table to **OrderID + ProductID**.



Like all associative entities, Order Detail is also a weak entity because it relies on both the Order and Products entities for existence. As such, it is natural that **OrderDetails.OrderID** and **OrderDetails.ProductID** be foreign keys in addition to being the table’s primary key.