Choosing a Primary Key

In the previous two chapters, I described how we need to choose a field or combination of fields to use as a primary key for a database table representing a class. The key fields will have unique values and so can be used to identify a particular row in a table. The primary key is also used to set up relationships between rows in different tables by way of a foreign key. Choosing a primary key is not always straightforward. For a person, combinations such as name and birth date are sometimes used as a key, but they cannot be guaranteed to be unique. You saw that introducing a customer number or using some sort of automatically generated ID number can ensure that we have a field that is unique for every row. Now let's have another look at this idea of ID numbers.

More About ID Numbers

A generated ID number does not solve all of our problems. If we have two rows in our table that are identical in every respect except for their ID, we are going to be in real trouble. Two John Smiths with the same birth date living at the same address are going to complicate matters whether they have different customer numbers or not. Are they the same person or are they different people? It would be intolerable if the only thing distinguishing one person from another was some generated ID. For one thing, who can ever remember all of their perhaps hundreds of different ID numbers? We always expect that a business will be able to find our customer number for us from information that differentiates us from everyone else.

So, does that mean that there will (or should) always be a possible key made up of some combination of the data kept about a customer? Probably yes. In that case, why do we need ID numbers? Wouldn't a primary key made up of all the fields in the table be OK?

One of the main reasons that ID numbers are necessary in many cases is that while there might always be some information that distinguishes one customer from another, it is likely that some of those values are constantly changing. If we decide that name, birth date, address, mother's maiden name, and so on will identify a customer, it is no use to us as a primary key in a table. Addresses are certain to change, names are likely to

change, and this is where we have a problem. We use the primary key in order to make connections to rows in different tables. For example, we would use the `Customer` table primary key as a foreign key in the `Order` table to identify which customer placed a specific order. If we have to put a combination of names and addresses into our `Order` table as a foreign key, I'm sure you can imagine the sorts of problems we are likely to encounter associating orders with particular customers when they move to a new address. An ID number, however, will be constant. One order might be associated with customer 3602 for example, the information in the `Customer` table about customer 3602 can change as much as it likes. Jane Green can move and remarry as much as she pleases, and we can still keep track of her orders through her constant customer number.

When storing information about people in a database, an ID number is almost always necessary. People are generally fairly resistant to being described by a number, and yet they are likely to have a different one for every business they deal with. Universal ID numbers are resisted by many civil liberties groups for privacy reasons, although in many countries social security, tax, or driver license numbers have almost become default universal IDs. Many web sites now use an email address as a form of identification—although this can cause problems when you share a personal email address with a spouse or partner. (I know this!)

While ID numbers are essential, there are still problems with them. One problem arises when a person is assigned two ID numbers for the same organization. Consider being admitted to a hospital. You are ill, and your friends are asked for your name and address and whether you have ever been admitted before. The name they give may be different from your exact name. They call you Rob Brown, but your real name is Jacob Robert Brown, and they don't know that you were once admitted as a child with tonsillitis. A new patient is therefore entered into the database with a new number. Now there are real problems: Rob Brown has two patient numbers and two rows in the patient table. Allergies may be associated with one patient number, and treatments with the other. Anecdotally, at various times the number of patients associated with New Zealand hospitals has been about 25% more than the total population!

This can happen just as easily when a student enrolls at a university. One year she pre-enrolls but then decides to take a year off traveling instead. She doesn't realize that she has been assigned a student number. The next year when she enrolls, she ticks the *new student* box and is given another number. Come graduation year, the student finds that some subjects have been credited to one number and some to the other, and neither has enough credits to graduate (this really does happen!).

There is not much that can be done about these problems other than to have very careful procedures at data entry times. Existing customers or clients with similar names need to be brought to the data entry operator's attention so that checks can be made. The process cannot be automated though, because sometimes two different people will have identical names and even birth dates.

Candidate Keys

In the previous chapter, we used functional dependencies to help us define what we meant by a key.

The key fields functionally determine all the other fields in a table.

This means that if we know the value of the key fields, we can locate a single row in our table, and then we can see the values of all the other fields. We also talked about fields that were not necessary to make a set of fields a possible key. For example, if we have `customerID` as a key in a `Customer` table, then by our definition the combination `customerID` and `customer_name` would also be a key. Clearly, `customer_name` is superfluous, and in Chapter 8 we discussed how this extra field would cause problems if we used it as part of a foreign key. The term *candidate key* is used to describe a key with no unnecessary fields.

A candidate key is a key where no subset of the fields is also a key.

With this definition, we see that the combination `customerID` and `customer_name` is not a candidate key, as the subset `customerID` is a key on its own. There may be more than one candidate key in a table. For example, in the `Customer` table, we may also store the customer's tax or social security number.

```
Customer(customerID, customer_name, address, phone, birth_date, tax_number)
```

Now we have two candidate keys: `customerID` and `tax_number`. Both will be unique for every record, and (so long as every customer is able and prepared to supply a tax number) either would be sufficient to uniquely identify a record. In a situation such as this, you choose one of the candidate keys as the primary key for the table. What are the considerations for choosing a primary key from among two or more candidates?

An ID Number or a Concatenated Key?

Let's take a fresh look at the problem from way back in Chapter 1 about insect data (Example 1-3). This was an environmental project where researchers regularly visited farms and took samples of insects from different fields. Because I want to use the word "field" in its database sense, I'm going to use the Australasian synonym for a field on a farm, *paddock*.

Let's build up the class diagram and the associated tables slowly. To start, we need to keep information about each farm as well as about the paddocks on that farm. A possible class diagram is shown in Figure 9-1.

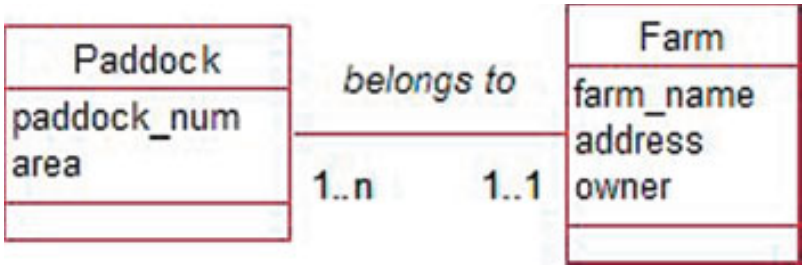


Figure 9-1. *Farms and paddocks*

Figure 9-1. *Farms and paddocks*

What will be a suitable primary key for the table representing the `Farm` class? Over time the name and owner may change, and in any case one person may own several farms, so the value of `owner` may not be unique. The farm is not going to shift, but the address may well change when roads are altered or boundaries change. An ID number seems the safest bet.

What about paddocks? Each farmer probably has some numbering system for his own paddocks. Just considering the two classes in Figure 9-1, we could therefore set up two tables:

```
Farm(farmID, farm_name, address, owner)
Paddock(paddock_num, area)
```

To represent the relationship between `Farm` and `Paddock`, we include the primary key from the `Farm` table as a foreign key field in the `Paddock` table: `Paddock(paddock_num, area, farm)`, where `farm` is a foreign key the value of which will be the farmID of the corresponding farm.

Now we have a decision to make. Is the paddock number a unique number over all paddocks, or is it just unique within a farm? The two possibilities are shown in Figure 9-2. In Figure 9-2a, the primary key would be `paddock_num`, and in Figure 9-2b, the primary key would be the combination (`farm`, `paddock_num`).

paddock_num	area	farm
336	30	18
337	23	18
345	25	17
346	25	17
347	35	17

a. Primary key paddock_num

farm	paddock_num	area
18	1	30
18	2	23
17	2	25
17	3	25
17	4	35

b. Primary key farm and paddock_num

Figure 9-2. *Simple and concatenated primary keys for the Paddock table*

In Figure 9-2a, we only need the one field as a primary key; however as the project grows, the numbers for a paddock will get large, and they don't mean very much. In the second option, the numbers for `paddock_num` are no longer unique (they restart from 1 for each farm), and we need two fields to identify a paddock. However, paddock (17, 2) means more to the owner of farm 17 than paddock 345. At this stage, the choice doesn't matter too much.

This relationship between farm and paddock (a 1–Many with a compulsory 1 end) is sometimes referred to as an *ownership* relation. The paddock *must* have an associated farm, or looking at it the other way around, the farm *owns* the paddock. When we get a long line of 1–Many ownership relationships, the issue of the size of the foreign key becomes more pressing. Consider some more of the insect data model as shown in Figure 9-3. The model describes a somewhat simplified version of the problem we considered before. Each visit has to be associated with a paddock, and each sample has to be associated with a particular visit.

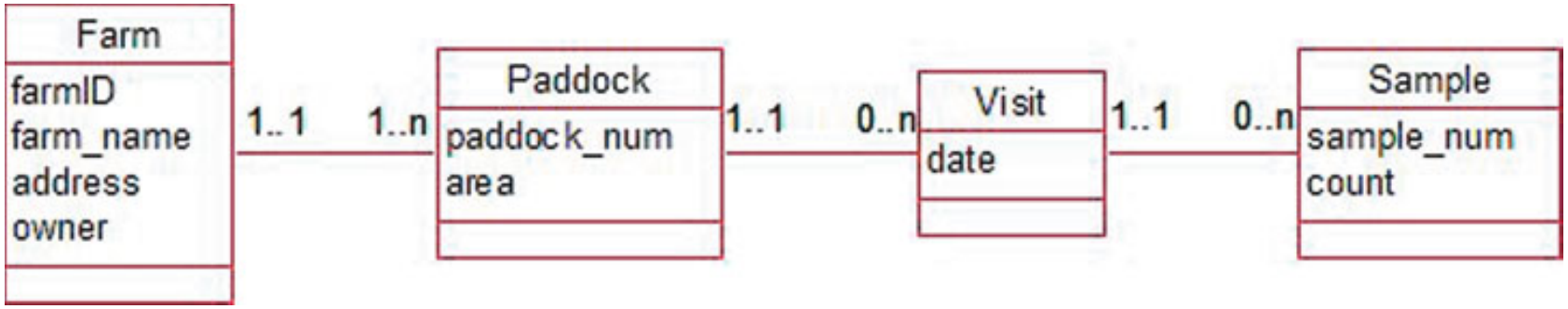


Figure 9-3. *Several 1–Many ownership relationships*

For each of the 1–Many relationships, we need to include the primary key from the 1 end as a foreign key in the Many end. Let's assume that a paddock can only be visited once on any given date. One possible set of tables for the preceding model could be as follows:

`Farm(farmID, farm_name, address, owner)`

`Paddock(farmID, paddock_num, area)`, with `farmID` being a foreign key referring to Farm

`Visit(date, farm, paddock)`, with `(farm, paddock)` being a foreign key referring to `Paddock`

`Sample(date, farm, paddock, sample_num,count)`, with `(date, farm, paddock)` being a foreign key referring to `Visit`

In this set of tables, we are assuming paddocks are numbered from 1 within each farm and samples are numbered from 1 within each visit. The `Visit` table doesn't need to have an ID because the combination (`date`, `farm`, `paddock`) is unique for this problem.

The `Sample` table is now looking quite cumbersome because the foreign key referring to the `Visit` table is a combination of three fields. This table is going to have the most rows eventually, so in addition to it just looking

ugly, there could also be a size consideration. Had we used the alternative in Figure 9-2a of a single key for Paddock, the foreign keys in the Visit and Sample tables would be a little smaller, but at the expense of having less–intuitive identifications for paddocks.

What other options do we have? Introducing a visitID makes some sense. Visits will probably be in a chronological order so that the ID number will mean something. Visit 458 will probably be the one that occurred after visit 457, whereas paddock 458 has no obvious relationship to paddock 457.

A happy compromise might be the following set of tables:

Farm(farmID, farm_name, address, owner)

Paddock(farmID, paddock_num, area) with farmID being a foreign key referring to Farm

Visit(visitID, date, farm, paddock) with (farm, paddock) being a foreign key referring to Paddock

Sample(visit, sample_num, count) with visit being a foreign key referring to Visit

The paddocks are numbered within farms, the visits are numbered chronologically, and the samples are numbered within a visit. All our introduced ID numbers therefore have some meaning, and the sample table is considerably smaller than in the previous design. In summary, choosing a primary key may not be straightforward. There are times when an automatically generated ID number will be necessary but won't solve all our problems. We might like to consider a primary key that is a concatenation of ID numbers (e.g., numbering paddocks within farms or samples within visits). There will always be a trade–off between concatenated ID numbers that might be more meaningful and having potentially cumbersome foreign keys in other related tables. There are always going to be alternative ways to choose a primary key, and as with most design issues, there is no hard–and–fast set of rules to say which choice is best.

Unique Constraints

Let's take another look at our `Visit` table, shown in Figure 9-4. We have two candidate keys: `visitID` and the combination (`date`, `farm`, `paddock`). For the reasons discussed in the previous section, we choose `visitID` as the primary key. Do we lose anything by making this choice?

visitID	date	farm	paddock
23	1/03/2011	18	1
24	1/03/2011	18	2
25	1/04/2011	17	3
26	1/04/2011	17	4

Figure 9-4. *Visit table with a generated visitID*

If the (`date`, `farm`, `paddock`) combination is not a primary key, we have lost the constraint that each row must have unique values for this combination of fields. This means we could mistakenly insert two rows for a visit to paddock 3, farm 17, on 1/04/2011. We still want to maintain the uniqueness of this combination, and we can do this by setting up a unique constraint.

Listing 9-1 shows the SQL to create the `Visit` table with a unique constraint to ensure that the combination (`date`, `farm`, `paddock`) is not duplicated in the table.

Listing 9-1. *SQL to Create the Visit Table with a Unique Constraint*

```
CREATE TABLE Visit (
visitID INT PRIMARY KEY,
date DATE,
farm INT,
paddock INT ,
FOREIGN KEY (farm, paddock) REFERENCES Paddock
UNIQUE (date, farm, paddock) )
```

Unique constraints are also a way to enforce a 1–1 relationship between tables. Consider the class diagram for sports teams in Figure 9-5, where each team has a member as its captain.

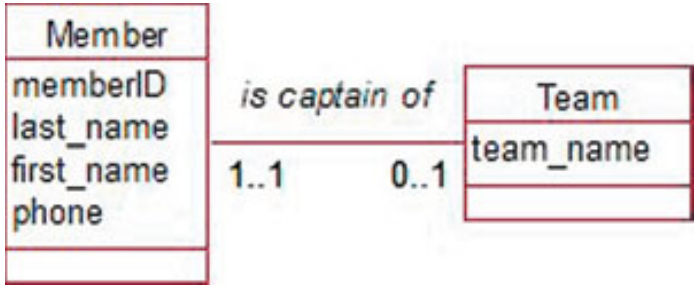


Figure 9-5. *A 1–1 relationship between Team and Member*

When we set up the classes in Figure 9-5 in a relational database, the 1–1 relationship will be represented by a foreign key in the Team table as shown in Figure 9-6.

team_name	captain
SeniorA	203
SeniorB	156
Wed Social	203

Figure 9-6. *Team table*

Each team can have only one captain (because we only have one `captain` field); however, we have yet to discuss a way of ensuring that each member can captain only one team as required by the 1–1 relationship. In Figure 9-6, note that member 203 is the captain of more than one team. We can prevent this from happening by adding a unique constraint on the `captain` field in the `Team` table. This will prevent a value being entered into the `captain` field in more than one row. The SQL to create the `Team` table with a unique constraint on the `captain` field is shown in Listing 9-2.

Listing 9-2. *Ensuring a 1–1 Captain Relationship Between Member and Team*

```
CREATE TABLE Team (
team_name VARCHAR(20) PRIMARY KEY,
captain INT UNIQUE FOREIGN KEY REFERENCES Member)
```

Unique constraints are able to help us with a couple of design issues: enforcing a 1–1 relationship and maintaining uniqueness for a candidate key that has not been chosen as a primary key.

Using Constraints Instead of Category Classes

Much of our discussion about classes and their corresponding tables in a relational database has involved introducing new classes and tables in order to keep data accurate and consistent. Now we are going to have a look at when you might decide not to add additional classes and why. Let's think about members of a club and their membership type (e.g., Senior, Junior, or Social). If we include membership type as a field in the `Member` table, we can have problems with consistency as can be shown by the misspelling in the first row of the table in Figure 9-7.

memberID	last_name	first_name	type
156	Jones	Graeme	senor
187	Green	Chris	Junior
203	Wang	James	Senior

Figure 9-7. *Keeping membership type as a field in the Member table*

If we are interested in creating reports that group all the members of different types (e.g., all the Seniors, and all the Juniors, etc.), we are going to run into trouble with the table in Figure 9-7 where we have different spelling of the types. Our solution in previous chapters was to create an additional class (table) to keep the different membership types and set up a 1–Many relationship as shown in Figure 9-8.

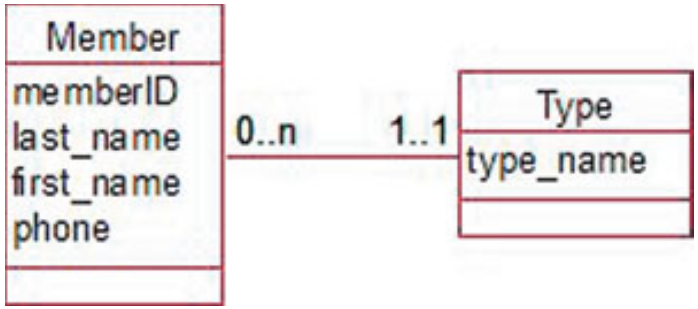


Figure 9-8. *Representing membership type with a class*

We can now have objects of the Type class or rows in a Type table to represent each of our types: Junior, Senior, and so on. This ensures that we have consistency in naming the different types. Take a look at the tables in Figure 9-9. The Type table seems a bit superfluous.

memberID	last_name	first_name	type
156	Jones	Graeme	Senior
187	Green	Chris	Junior
203	Wang	James	Senior

Member

type_name
Senior
Junior
Social

Type

Figure 9-9. *Membership type is a separate table*

All the additional Type table is achieving is to ensure the consistency of the entries in the `type` field of the `Member` table. We can achieve the same thing by putting a check constraint on the `type` field. We discussed constraints on fields in Chapter 7. Figure 9-10 shows how easily this can be done in a product like Access, and Listing 9-3 shows the equivalent SQL.

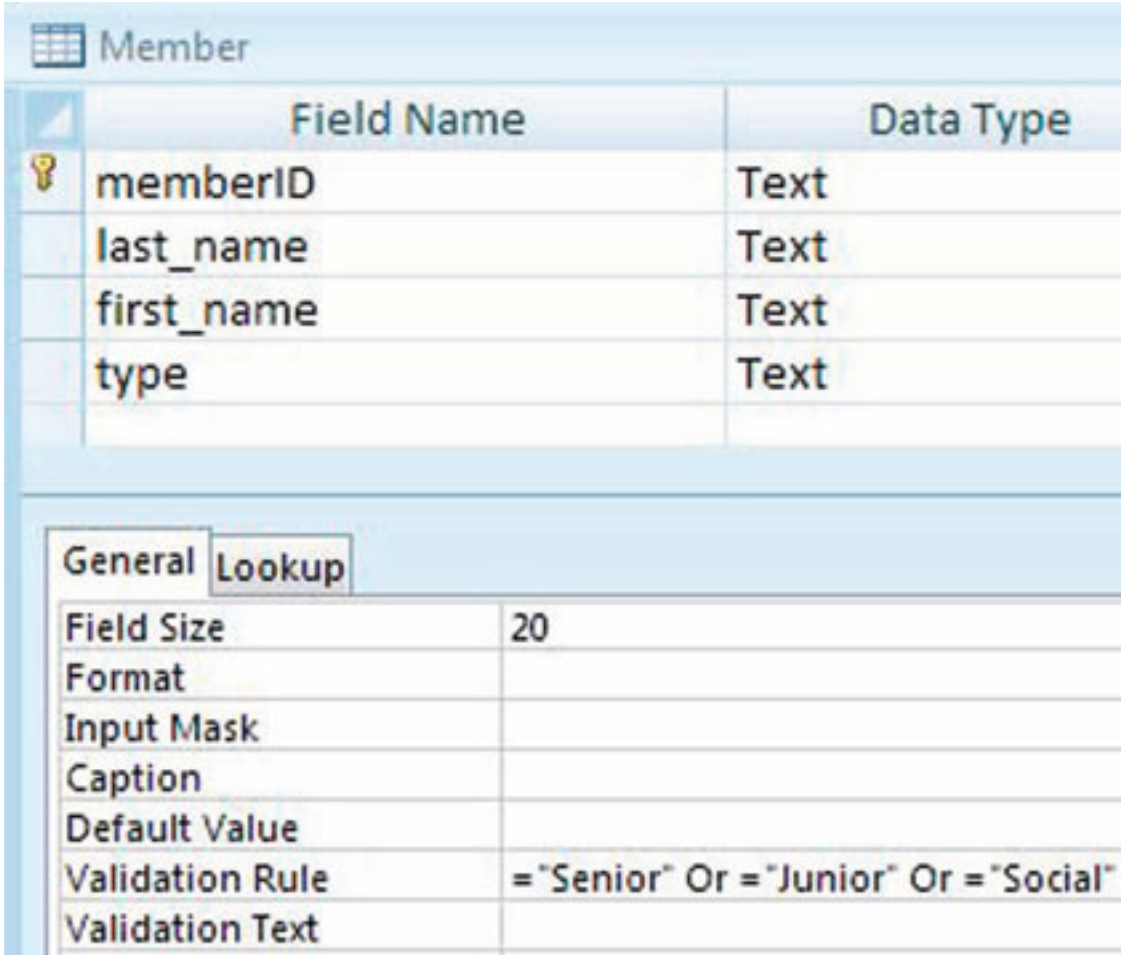


Figure 9-10. *Membership type with a constraint*

Listing 9-3. *Placing a Constraint on a Field*

```
CREATE TABLE Member (
memberID INT PRIMARY KEY,
last_name VARCHAR(20),
first_name VARCHAR(20),
type VARCHAR(20) CHECK type IN (‘Senior’, ‘Junior’, ‘Social’))
```

Which should we prefer: a table with a constraint on a field (Figure 9-10) or one with a reference to another table (Figure 9-9)? In Figure 9-10, we have a constraint built into the design of the table. If additional membership types are added at a later date, the definition of the constraint would have to be changed. This is something that needs to be done by a system manager or at least someone trusted to alter the design. On the positive side, we have one fewer table in our database.

In Figure 9-9, we have the additional complexity of an extra table. However, if another membership type is required, it can be added simply as a new row in the Type table. This is just a data entry job and doesn't involve any change to the design of the database. If the types are going to be fairly constant, the constraint is simpler, whereas the reference to another table makes it easy for a user to add different types.

There is one case in which the extra table will always be the appropriate choice. This is when there are (or may later be) some additional attributes belonging to the Type class. For example, if we wish to keep a fee for each different membership type, the only way we can do this is via a Type class as shown in Figure 9-11.

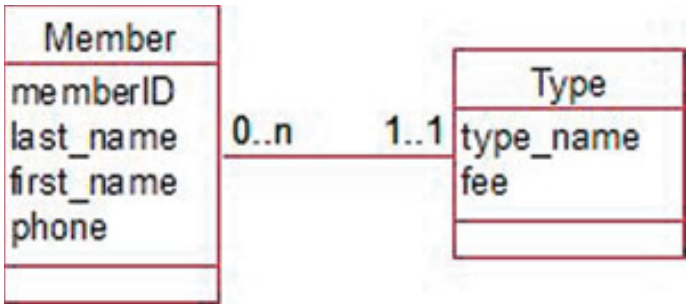


Figure 9-11. *An extra class is needed if there are additional attributes.*

In summary, when we have a piece of data that acts like a category (e.g., membership type), we sometimes have a choice as to whether we store this as a simple field in a table and keep the values consistent by way of a constraint or validation rule, or we have a separate table of categories to which we refer. If the number of categories is likely to increase, the second option is better, as this then becomes a simple matter of adding additional rows to the category table rather than changing the constraint on the parent table. If there are or are likely to be other attributes associated with the category, the additional table is the only option. If neither of these situations applies, it is worth thinking about whether a simple field with a constraint may be more appropriate.

Deleting Referenced Records

You have seen how we can use foreign keys to represent relationships between two tables. Have another look at our model of teams and members in Figure 9-5. We can represent the relationship *is captain of* with a foreign key (`captain`) in the `Team` table, as shown in Figure 9-12.

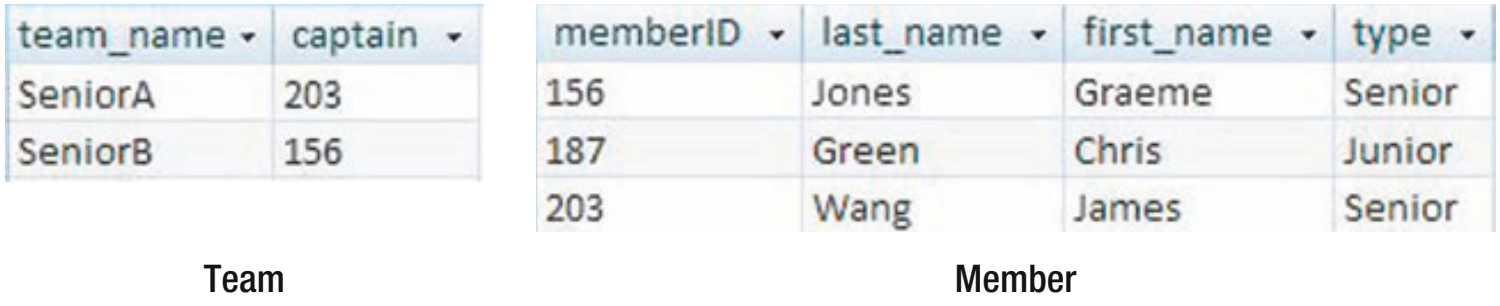


Figure 9-12. *Teams and members*

A foreign key ensures that we have referential integrity. Recall from Chapter 7 that referential integrity prevents us from having a value in the foreign key field `captain` if the value does not exist in the primary key field `memberID` in the `Member` table. This ensures that all of our captains are members for which we have already recorded names and other details. Unlike a primary key, a foreign key field is not necessarily mandatory, and the captain field may be empty (i.e., referential integrity does not make it necessary for every team to have a captain) We can of course impose that extra constraint if we want to, by specifying that the `captain` field must be `NOT NULL`

So far we have only looked at referential integrity from the point of view of adding new teams and captains. However, we also have the situation of deleting members from the `Member` table. If we attempt to delete member 156, for example, we shall have a problem with the referential integrity in the `Team` table. The captain of SeniorB will no longer exist in the `Member` table.

There are three ways to deal with this situation. Database software products vary in their ability to provide each of these options, but all will provide the first, as follows:

Disallow delete: You cannot delete a row that is being referenced. For example, the deletion of member 156 will not be allowed while it is being referenced by the `Team` table. If we want to delete member 156, we will first have to remove the reference to him in the `Team` table and then delete him from the `Member` table.

Nullify delete: If member 156 is deleted, the field that is referencing it, `captain`, will be nullified (made empty). This essentially is saying that if a captain of a team leaves the club, that team has no captain—which is probably quite sensible in this situation.

Cascade delete: If a row is deleted, all the rows referencing it will be deleted also (and the rows referencing them, and on and on). In this case, deleting member 156 would mean that the team SeniorB would be deleted. This is clearly not desirable.

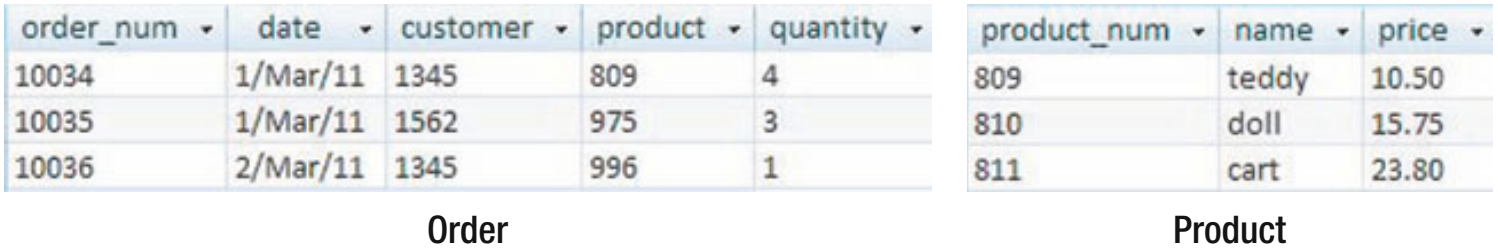
When we set up a field as a foreign key, we can specify what should occur when there is an attempt to delete the row to which it refers. Listing 9-4 shows the SQL statement for specifying a "nullify delete" for the foreign key `captain` when we create the `Team` table. If you do not specify an option, the default is generally "disallow delete."

Listing 9-4. *Specifying a Deletion Option on a Foreign Key*

```
CREATE TABLE Team (
team_name VARCHAR(20),
captain INT FOREIGN KEY REFERENCES Member ON DELETE NULLIFY )
```

Depending on the particular problem, we can choose the deletion option that is most appropriate. For the team and member situation, a "nullify delete" seems sensible for the foreign key `captain`. We want to be able to delete members, and it makes sense that if member 156 leaves, there will be a vacancy for the captain of the SeniorB team. Our model as it stands in Figure 9-5 doesn't allow this, however. It says every team must have a captain. Maybe it is worth reconsidering this. While in the normal course of events we expect all our teams to have captains, we are going to come across cases where people unexpectedly leave or resign. What do we want to happen to the data we are keeping in these cases? If we insist that every team has a captain (by making that field required), we will have to find a new captain before we can delete the old captain from our membership list. Perhaps this is what we want to do, or maybe that will be too restrictive. We talked in previous chapters about the consequences of making fields required. Thinking about deletions from the database may make us reconsider the relationship *is captain of* and whether or not it should be optional.

Let's consider a different situation, of orders and products, as in Figure 9-13.



order_num	date	customer	product	quantity
10034	1/Mar/11	1345	809	4
10035	1/Mar/11	1562	975	3
10036	2/Mar/11	1345	996	1

Order

product_num	name	price
809	teddy	10.50
810	doll	15.75
811	cart	23.80

Product

Figure 9-13. *Orders and products: What happens if we delete a product?*

What happens if we no longer stock product 809? If we delete this row in the `Product` table, our referential integrity will be compromised as order number 10034 refers to it. What are our choices? A "nullify delete" means having nothing in the foreign key `product` field in the `Order` table. This makes no sense. We would have that there was once an order for four of some product—but we don't know what that product was and we have no way of finding the price. Clearly, this is not going to be useful. A "cascade delete" would mean that all the orders for product 809 would be deleted. This doesn't seem sensible either, as a business is going to need to keep track of all its orders to determine profits, tax, and so on. Our only choice in this case, then, is the "disallow delete" option. If there is an order for the product, we can't delete that product from the `Product` table.

It is important that we keep discontinued products in the table, but we will want to be able to distinguish them from current products. For a case like this, we might decide to add an additional field to our `Product` table (say `current`) to distinguish current products from discontinued ones. We have a new problem now. How do we prevent new orders from being entered for discontinued products? This is starting to get outside the scope of this book. Many database applications allow you to put additional constraints on a table by way of *triggers*. A trigger is a procedure that is fired by a change to a table (e.g., adding or updating a row) and will carry out specified actions. In this case, the trigger would check whether a potential new row in the `Order` table was for a

discontinued product, and if so prevent that row from being permanently added to the table. Constraints such as only allowing orders for current products can also be implemented through the interface of the database, and we will look at the benefits and drawbacks of this in Chapter 11.

When might a "cascade delete" be a good choice? It is a fairly brutal solution, and you should be very careful about setting it up. If we have enrollments for a subject, and then that subject is cancelled, it is perhaps reasonable to expect that all the enrollments for it should be deleted also. We have to be careful, though, that there aren't historical enrollments from previous years. Deleting information does not happen as often as you might expect. Products and subjects may be discontinued, but if we have historical orders or enrollments for subjects which have since been discontinued, we need to keep the information. In these cases, the "disallow delete" option is the best bet. When customers and orders do outlive their usefulness, it is more usual to archive the important, or summarized, information and store it elsewhere rather than deleting it entirely. Use the "cascade delete" option with caution.

Having said all that, we might think that the safest option is to never delete anything. It is possible to set up tables so that no rows can ever be deleted. However, while this might seem like a good idea, we will always need to delete records that have been entered into a table by mistake. Say we accidentally enter the same customer twice (with a different customer number), for example. We need to get that extra record out of the table as quickly as possible before it causes us all sorts of problems.

Summary

In this chapter, we have looked at some issues involved with choosing appropriate primary keys and for ensuring referential integrity is maintained when we update the data in our tables. Some of the important points to remember are the following:

- We often need to introduce a generated ID number to ensure we have a field, with stable and unique values, that we can use as a primary key. This is particularly true for people, where identifying information such as names and addresses are likely to change.
- Be aware that mistakes in data entry means it is possible to have a person in your database twice with two different ID numbers. Try to avoid this!
- Where a primary key is made up of several concatenated fields, it is worth considering a generated ID number to reduce the size of the foreign keys referencing the table.
- Where a generated ID has been introduced, constraints should be used to retain the uniqueness of the combinations of fields that have been replaced as a primary key.
- Unique constraints can be used to enforce a 1–1 relationship.
- A constraint on the value of a field may be more appropriate than a relationship to another (very simple) table.
- You have three options when you wish to delete a row that it is being referenced by a foreign key:
    - Disallow the deletion.
    - Make the field referencing the deleted row `NULL` (“nullify delete”).
    - Remove all rows that reference the deleted row (“cascade delete”).

Exercise 9-1

The ever–useful “customer orders product” example again—there is always something new to discover. Design the table that will represent the `Order` class in Figure 9-14. Consider constraints, primary and foreign keys, and updating rules.

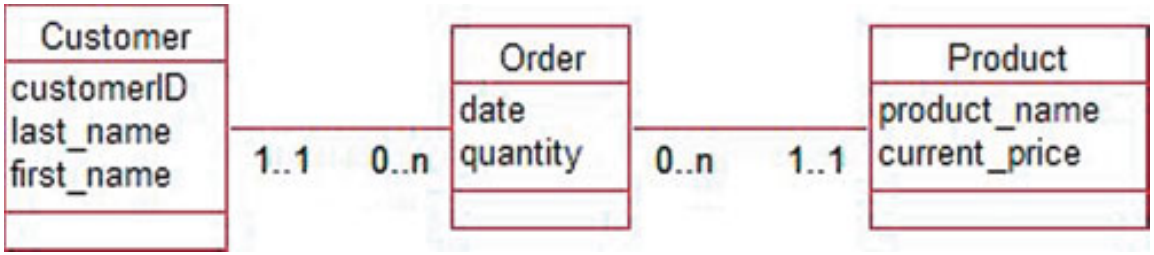


Figure 9-14. *Model for customers placing orders*

Exercise 9-2

A car sales yard needs to keep information about makes and models of cars that are available, and also the registrations of the individual cars they have in stock. For example, Ford Siestas are available in sedans and hatchbacks, and they currently have a blue sedan with registration TC545 in stock. For some models you might be able to choose automatic or manual transmission and some come with different capacities (e.g., 1.5 l or 2.0 l). Think about the options available for setting up tables for this situation.