CHAPTER 4

Modeling and Designing Databases

When implementing a new database, it's easy to fall into the trap of trying to quickly get something up and running without dedicating adequate time and effort to the design. This carelessness frequently leads to costly redesigns and reimplementations down the track. Designing a database is similar to drafting the blueprints for a house; it's silly to start building without detailed plans. Importantly, good design allows you to extend the original building without having to pull everything down and start from scratch.

How Not to Develop a Database

Database design is probably not the most exciting task in the world, but it's still important. Before we describe how to go about the design process, let's look at an example of database design on the run.

Imagine we want to create a database to store student grades for a university computer science department. We could create a `Student_Grades` table to store grades for each student and each course. The table would have columns for the given names and the surname of each student as well as for each course they have taken, the course name, and the percentage result (shown as `Pctg`). We'd have a different row for each student for each of their courses:

```
+------------+---------+--------------------------+------+
| GivenNames | Surname | CourseName               | Pctg |
+------------+---------+--------------------------+------+
| John Paul  | Bloggs  | Web Database Applications |   72 |
| Sarah      | Doe     | Programming 1             |   87 |
| John Paul  | Bloggs  | Computing Mathematics     |   43 |
| John Paul  | Bloggs  | Computing Mathematics     |   65 |
| Sarah      | Doe     | Web Database Applications |   65 |
| Susan      | Smith   | Computing Mathematics     |   75 |
| Susan      | Smith   | Programming 1             |   55 |
| Susan      | Smith   | Computing Mathematics     |   80 |
+------------+---------+--------------------------+------+
```

This is nice and compact, and we can easily access grades for any student or any course. However, we could have more than one student called Susan Smith; in the sample data, there are two entries for Susan Smith and the Computing Mathematics course. Which Susan Smith got an 80? A common way to differentiate duplicate data entries is to assign a unique number to each entry. Here, we can assign a unique Student ID number to each student:

```
+------------+------------+---------+--------------------------+------+
| StudentID  | GivenNames | Surname | CourseName               | Pctg |
+------------+------------+---------+--------------------------+------+
| 12345678   | John Paul  | Bloggs  | Web Database Applications |   72 |
| 12345121   | Sarah      | Doe     | Programming 1            |   87 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics    |   43 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics    |   65 |
| 12345121   | Sarah      | Doe     | Web Database Applications |   65 |
| 12345876   | Susan      | Smith   | Computing Mathematics    |   75 |
| 12345876   | Susan      | Smith   | Programming 1            |   55 |
| 12345303   | Susan      | Smith   | Computing Mathematics    |   80 |
+------------+------------+---------+--------------------------+------+
```

So, the Susan Smith who got 80 is the one with the Student ID number 12345303.

There's another problem. In our table, John Paul Bloggs has failed the Computing Mathematics course once with 43 percent, and passed it with 65 percent in his second attempt. In a relational database, the rows form a set, and there is no implicit ordering between them; you might guess that the pass happened after the fail, but you can't actually be sure. There's no guarantee that the newer grade will appear after the older one, so we need to add information about *when* each grade was awarded, say by adding a year and semester (`Sem`):

```
+------------+------------+---------+--------------------------+------+-----+------+
| StudentID  | GivenNames | Surname | CourseName               | Year | Sem | Pctg |
+------------+------------+---------+--------------------------+------+-----+------+
| 12345678   | John Paul  | Bloggs  | Web Database Applications | 2004 |   2 |   72 |
| 12345121   | Sarah      | Doe     | Programming 1            | 2006 |   1 |   87 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics    | 2005 |   2 |   43 |
| 12345678   | John Paul  | Bloggs  | Computing Mathematics    | 2006 |   1 |   65 |
| 12345121   | Sarah      | Doe     | Web Database Applications | 2006 |   1 |   65 |
| 12345876   | Susan      | Smith   | Computing Mathematics    | 2005 |   1 |   75 |
| 12345876   | Susan      | Smith   | Programming 1            | 2005 |   2 |   55 |
| 12345303   | Susan      | Smith   | Computing Mathematics    | 2006 |   1 |   80 |
+------------+------------+---------+--------------------------+------+-----+------+
```

Notice that the `Student_Grades` table has become a bit bloated: the student ID, given names, and surname are repeated for every grade. We could split up the information and create a `Student_Details` table:

```
+------------+------------+---------+
| StudentID  | GivenNames | Surname |
+------------+------------+---------+
| 12345121   | Sarah      | Doe     |
| 12345303   | Susan      | Smith   |
| 12345678   | John Paul  | Bloggs  |
```

```
| 12345876   | Susan      | Smith   |
+------------+------------+---------+
```

and keep less information in the `Student_Grades` table:

```
+------------+--------------------------+------+-----+------+
| StudentID  | CourseName               | Year | Sem | Pctg |
+------------+--------------------------+------+-----+------+
| 12345678   | Web Database Applications | 2004 |   2 |   72 |
| 12345121   | Programming 1            | 2006 |   1 |   87 |
| 12345678   | Computing Mathematics    | 2005 |   2 |   43 |
| 12345678   | Computing Mathematics    | 2006 |   1 |   65 |
| 12345121   | Web Database Applications | 2006 |   1 |   65 |
| 12345876   | Computing Mathematics    | 2005 |   1 |   75 |
| 12345876   | Programming 1            | 2005 |   2 |   55 |
| 12345303   | Computing Mathematics    | 2006 |   1 |   80 |
+------------+--------------------------+------+-----+------+
```

To look up a student's grades, we'd need to first look up her Student ID from the `Student_Details` table and then read the grades for that Student ID from the `Student_Grades` table.

There are still issues we haven't considered. For example, should we keep information on a student's enrollment date, postal and email addresses, fees, or attendance? Should we store different types of postal address? How should we store addresses so that things don't break when a student changes his address?

Implementing a database in this way is problematic; we keep running into things we hadn't thought about and have to keep changing our database structure. Clearly, we can save a lot of reworking by carefully documenting the requirements and then working through them to develop a coherent design.

The Database Design Process

There are three major stages in database design, each producing a progressively lower-level description:

Requirements analysis
: First, we determine and write down what exactly the database is needed for, what data will be stored, and how the data items relate to each other. In practice, this might involve detailed study of the application requirements and talking to people in various roles that will interact with the database and application.

Conceptual design
: Once we know what the database requirements are, we distill them into a formal description of the database design. In this chapter, we'll see how to use modeling to produce the conceptual design.

Logical design
: Finally, we map the database design onto an actual database management system and database tables.

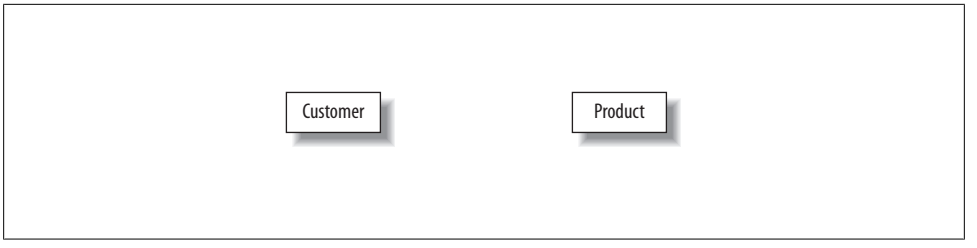

Figure 4-1. An entity set is represented by a named rectangle

At the end of the chapter, we'll look at how we can use the open source `MySQL Workbench` tool to automatically convert the conceptual design to a MySQL database schema.

The Entity Relationship Model

At a basic level, databases store information about distinct objects, or *entities*, and the associations, or *relationships*, between these entities. For example, a university database might store information about students, courses, and enrollment. A student and a course are entities, while an enrollment is a relationship between a student and a course. Similarly, an inventory and sales database might store information about products, customers, and sales. A product and a customer are entities, while a sale is a relationship between a customer and a product.

A popular approach to conceptual design uses the *Entity Relationship* (ER) model, which helps transform the requirements into a formal description of the entities and relationships that appear in the database. We'll start by looking at how the Entity Relationship modeling process itself works, then apply it in "Entity Relationship Modeling Examples" for three sample databases.

Representing Entities

To help visualize the design, the Entity Relationship Modeling approach involves drawing an Entity Relationship (ER) diagram. In the ER diagram, an *entity set* is represented by a rectangle containing the entity name. For our sales database example, the product and customer entity sets would be shown as in Figure 4-1.

We typically use the database to store certain characteristics, or *attributes*, of the entities. In a sales database, we could store the name, email address, postal address, and telephone number for each customer. In a more elaborate customer relationship managment (CRM) application, we could also store the names of the customer's spouse and children, the languages the customer speaks, the customer's history of interaction with our company, and so on. Attributes describe the entity they belong to.

An attribute may be formed from smaller parts; for example, a postal address is composed of a street number, city, ZIP code, and country. We classify attributes as *composite* if they're composed of smaller parts in this way, and as *simple* otherwise.

Some attributes can have multiple values for a given entity. For example, a customer could provide several telephone numbers, so the telephone number attribute is *multivalued*.

Attributes help distinguish one entity from other entities of the same type. We could use the name attribute to distinguish between customers, but this could be an inadequate solution because several customers could have identical names. To be able to tell them apart, we need an attribute (or a minimal combination of attributes) guaranteed to be unique to each individual customer. The identifying attribute or attributes form a *key*.

In our example, we can assume that no two customers have the same email address, so the email address can be the key. However, we need to think carefully about the implications of our choices. For example, if we decide to identify customers by their email address, it would be hard to allow a customer to have multiple email addresses. Any applications we build to use this database might treat each email address as a separate person, and it might be hard to adapt everything to allow people to have multiple email addresses. Using the email address as the key also means that every customer must have an email address; otherwise, we wouldn't be able to distinguish between customers who don't have one.

Looking at the other attributes for one that can serve as an alternative key, we see that while it's possible that two customers would have the same telephone number (and so we cannot use the telephone number as a key), it's likely that people who have the same telephone number never have the same name, so we can use the combination of the telephone number and the name as a composite key.

Clearly, there may be several possible keys that could be used to identify an entity; we choose one of the alternative, or *candidate*, keys to be our main, or *primary*, key. You usually make this choice based on how confident you are that the attribute will be nonempty and unique for each individual entity, and on how small the key is (shorter keys are faster to maintain and use).

In the ER diagram, attributes are represented as labeled ovals and are connected to their owning entity, as shown in Figure 4-2. Attributes comprising the primary key are shown underlined. The parts of any composite attributes are drawn connected to the oval of the composite attribute, and multivalued attributes are shown as double-lined ovals.

Attribute values are chosen from a *domain* of legal values; for example, we could specify that a customer's given names and surname attributes can each be a string of up to 100 characters, while a telephone number can be a string of up to 40 characters. Similarly, a product price could be a positive rational number.

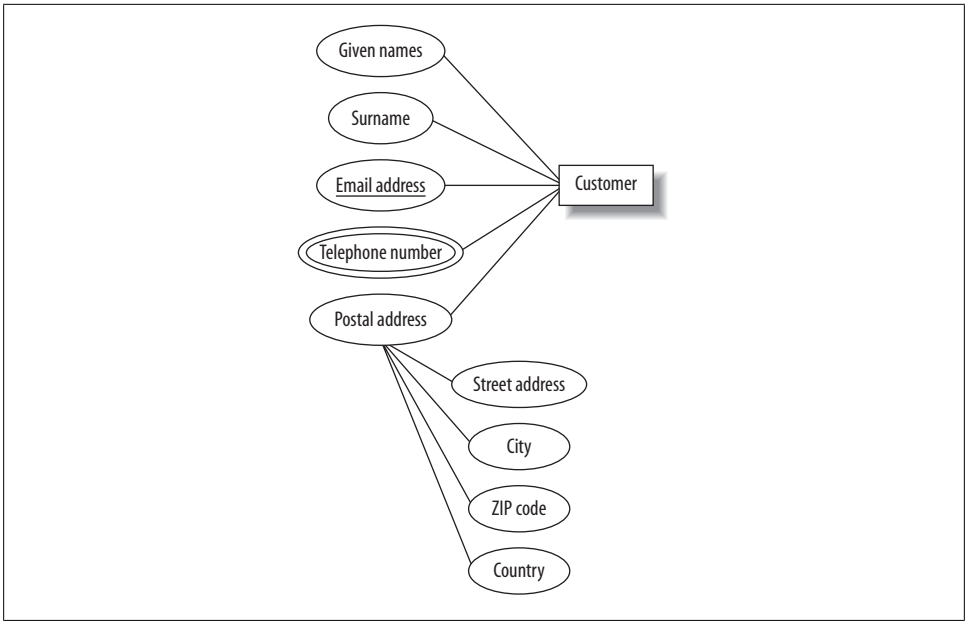

Figure 4-2. The ER diagram representation of the customer entity

Attributes can be empty; for example, some customers may not provide their telephone numbers. The primary key of an entity (including the components of a multiattribute primary key) must never be unknown (technically, it must be `NOT NULL`); for example, if it's possible for a customer to not provide an email address, we cannot use the email address as the key.

You should think carefully when classifying an attribute as multivalued: are all the values equivalent, or do they in fact represent different things? For example, when listing multiple telephone numbers for a customer, would they be more usefully labeled separately as the customer's business phone number, home phone number, cell phone number, and so on?

Let's look at another example. The sales database requirements may specify that a product has a name and a price. We can see that the product is an entity because it's a distinct object. However, the product's name and price aren't distinct objects; they're attributes that describe the product entity. Note that if we want to have different prices for different markets, then the price is no longer just related to the product entity, and we'd need to model it differently.

For some applications, no combination of attributes can uniquely identify an entity (or it would be too unwieldy to use a large composite key), so we create an artificial attribute that's defined to be unique and can therefore be used as a key: student numbers, Social Security numbers, driver's license numbers, and library card numbers are examples of unique attributes created for various applications. In our inventory and sales applica-

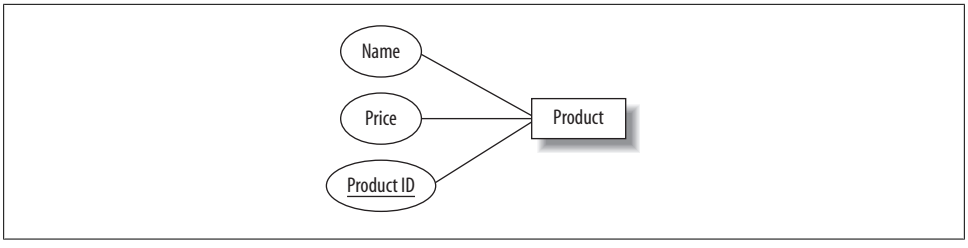

Figure 4-3. The ER diagram representation of the product entity

tion, it's possible that we could stock different products with the same name and price. For example, we could sell two models of "Four-port USB 2.0 Hub," both at $4.95 each. To distinguish between products, we can assign a unique product ID number to each item we stock; this would be the primary key. Each product entity would have name, price, and product ID attributes. This is shown in the ER diagram in Figure 4-3.

Representing Relationships

Entities can participate in relationships with other entities. For example, a customer can buy a product, a student can take a course, an artist can record an album, and so on.

Like entities, relationships can have attributes: we can define a sale to be a relationship between a customer entity (identified by the unique email address) and a given number of the product entity (identified by the unique product ID) that exists at a particular date and time (the timestamp).

Our database could then record each sale and tell us, for example, that at 3:13 p.m. on Wednesday, March 22, Ali Thomson bought one "Four-port USB 2.0 Hub," one "300 GB 16 MB Cache 7200 rpm SATA Serial ATA133 HDD Hard Disk," and two sets of "2000 Watt 5.1 Channel Sub-Woofer Speakers."

Different numbers of entities can appear on each side of a relationship. For example, each customer can buy any number of products, and each product can be bought by any number of customers. This is known as a *many-to-many* relationship. We can also have *one-to-many* relationships. For example, one person can have several credit cards, but each credit card belongs to just one person. Looking at it the other way, a *one-to-many* relationship becomes a *many-to-one* relationship; for example, many credit cards belong to a single person. Finally, the serial number on a car engine is an example of a *one-to-one* relationship; each engine has just one serial number, and each serial number belongs to just one engine. We often use the shorthand terms 1:1, 1:N, and M:N for one-to-one, one-to-many, and many-to-many relationships, respectively.

The number of entities on either side of a relationship (the *cardinality* of the relationship) define the *key constraints* of the relationship. It's important to think about the cardinality of relationships carefully. There are many relationships that may at first seem to be one-to-one, but turn out to be more complex. For example, people some-

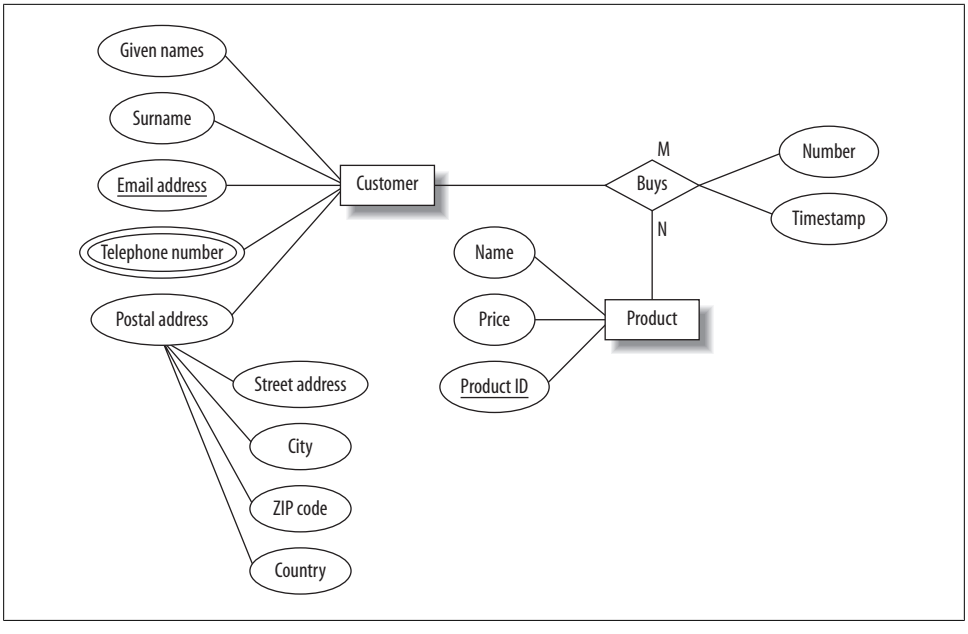

Figure 4-4. The ER diagram representation of the customer and product entities, and the sale relationship between them.

times change their names; in some applications, such as police databases, this is of particular interest, and so it may be necessary to model a many-to-many relationship between a person entity and a name entity. Redesigning a database can be time-consuming if you assume a relationship is simpler than it really is.

In an ER diagram, we represent a *relationship set* with a named diamond. The cardinality of the relationship is often indicated alongside the relationship diamond; this is the style we use in this book. (Another common style is to have an arrowhead on the line connecting the entity on the "1" side to the relationship diamond.) Figure 4-4 shows the relationship between the customer and product entities, along with the number and timestamp attributes of the sale relationship.

Partial and Total Participation

Relationships between entities can be optional or compulsory. In our example, we could decide that a person is considered to be a customer only if they have bought a product. On the other hand, we could say that a customer is a person whom we know about and whom we hope might buy something—that is, we can have people listed as customers in our database who never buy a product. In the first case, the `customer` entity has *total participation* in the bought relationship (all `customer`s have bought a product, and we can't have a `customer` who hasn't bought a product), while in the second case it has *partial participation* (a `customer` can buy a product). These are referred to as the

participation constraints of the relationship. In an ER diagram, we indicate total participation with a double line between the entity box and the relationship diamond.

Entity or Attribute?

From time to time, we encounter cases where we wonder whether an item should be an attribute or an entity on its own. For example, an email address could be modeled as an entity in its own right. When in doubt, consider these rules of thumb:

Is the item of direct interest to the database?
: Objects of direct interest should be entities, and information that describes them should be stored in attributes. Our inventory and sales database is really interested in customers, and not their email addresses, so the email address would be best modeled as an attribute of the `customer` entity.

Does the item have components of its own?
: If so, we must find a way of representing these components; a separate entity might be the best solution. In the student grades example at the start of the chapter, we stored the course name, year, and semester for each course that a student takes. It would be more compact to treat the course as a separate entity and to create a class ID number to identify each time a course is offered to students (the "offering").

Can the object have multiple instances?
: If so, we must find a way to store data on each instance. The cleanest way to do this is to represent the object as a separate entity. In our sales example, we must ask whether customers are allowed to have more than one email address; if they are, we should model the email address as a separate entity.

Is the object often nonexistent or unknown?
: If so, it is effectively an attribute of only some of the entities, and it would be better to model it as a separate entity rather than as an attribute that is often empty. Consider a simple example: to store student grades for different courses, we could have an attribute for the student's grade in every possible course; this is shown in Figure 4-5. Because most students will have grades for only a few of these courses, it's better to represent the grades as a separate entity set, as in Figure 4-6.

Entity or Relationship?

An easy way to decide whether an object should be an entity or a relationship is to map nouns in the requirements to entities, and to map the verbs to relations. For example, in the statement, "A degree program is made up of one or more courses," we can identify the entities "program" and "course," and the relationship "is made up of." Similarly, in the statement, "A student enrolls in one program," we can identify the entities "student" and "program," and the relationship "enrolls in." Of course, we can choose different terms for entities and relationships than those that appear in the relationships, but it's a good idea not to deviate too far from the naming conventions used in the

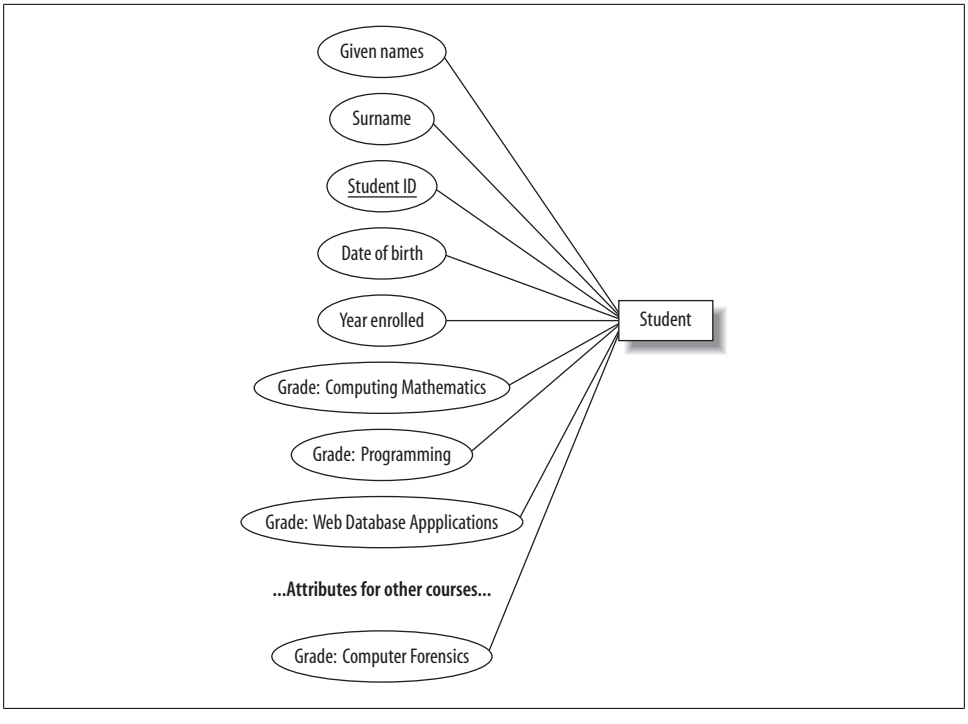

Figure 4-5. The ER diagram representation of student grades as attributes of the student entity

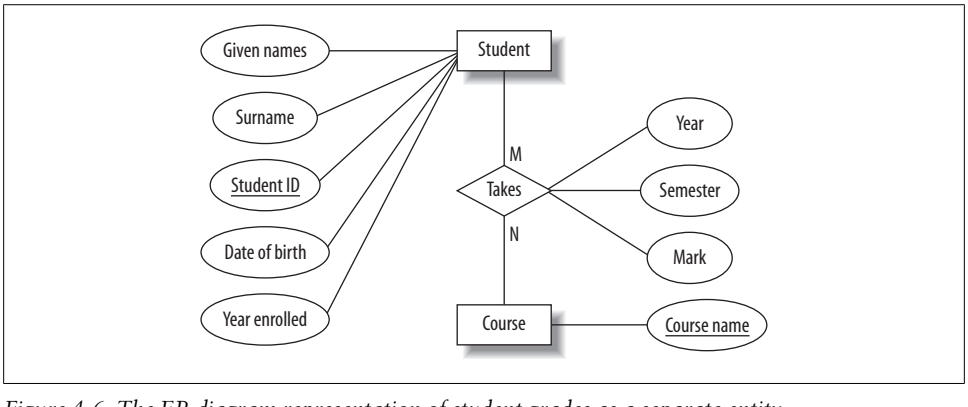

Figure 4-6. The ER diagram representation of student grades as a separate entity

requirements so that the design can be checked against the requirements. All else being equal, try to keep the design simple, and avoid introducing trivial entities where possible; i.e., there's no need to have a separate entity for the student's enrollment when we can model it as a relationship between the existing student and program entities.

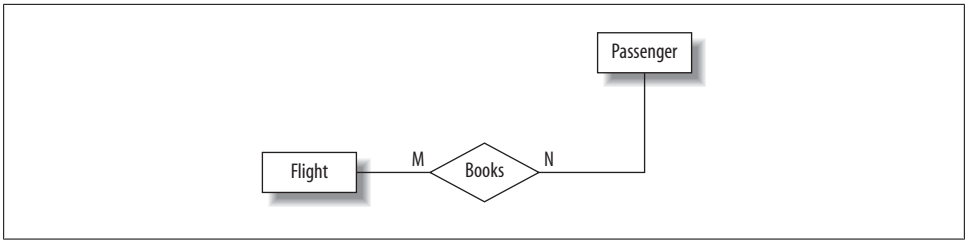

Figure 4-7. A passenger participates in an M:N relationship with flight

Intermediate Entities

It is often possible to conceptually simplify many-to-many relationships by replacing the many-to-many relationship with a new *intermediate entity* (sometimes called an *associate entity*) and connecting the original entities through a many-to-one and a one-to-many relationship.

Consider the statement: "A passenger can book a seat on a flight." This is a many-to-many relationship between the entities "passenger" and "flight." The related ER diagram fragment is shown in Figure 4-7.

However, let's look at this from both sides of the relationship:

- Any given flight can have many passengers with a booking.
- Any given passenger can have bookings on many flights.

Hence, we can consider the many-to-many relationship to be in fact two one-to-many relationships, one each way. This points us to the existence of a hidden intermediate entity, the booking, between the flight and the passenger entities. The requirement could be better worded as: "A passenger can make a booking for a seat on a flight." The related ER diagram fragment is shown in Figure 4-8.

Each passenger can be involved in multiple bookings, but each booking belongs to a single passenger, so the cardinality of this relationship is 1:N. Similarly, there can be many bookings for a given flight, but each booking is for a single flight, so this relationship also has cardinality 1:N. Since each booking must be associated with a particular passenger and flight, the booking entity participates totally in the relationships with these entities. This total participation could not be captured effectively in the representation in Figure 4-7. (We described partial and total participation earlier in "Partial and Total Participation.")

Weak and Strong Entities

Context is very important in our daily interactions; if we know the context, we can work with a much smaller amount of information. For example, we generally call family members by only their first name or nickname. Where ambiguity exists, we add further information such as the surname to clarify our intent. In database design, we can omit

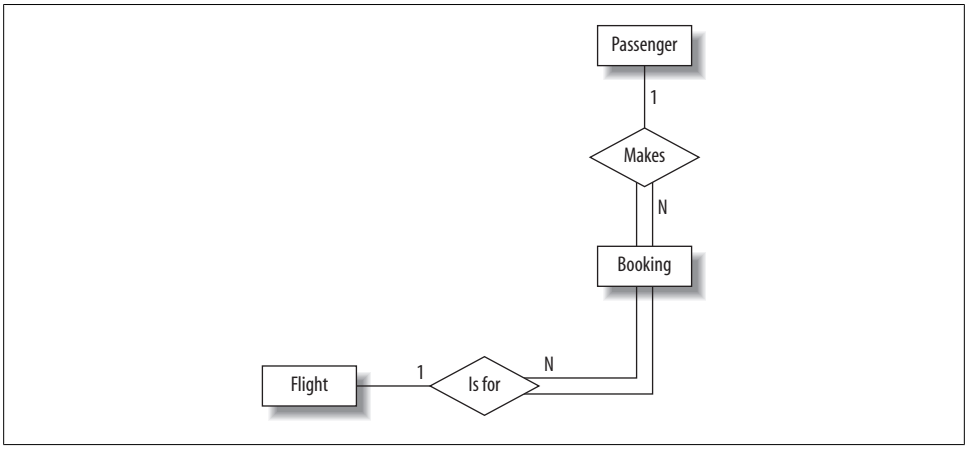

Figure 4-8. The intermediate booking entity between the passenger and flight entities

some key information for entities that are dependent on other entities. For example, if we wanted to store the names of our customers' children, we could create a child entity and store only enough key information to identify it in the context of its parent. We could simply list a child's first name on the assumption that a customer will never have several children with the same first name. Here, the child entity is a *weak* entity, and its relationship with the customer entity is called an *identifying relationship*. Weak entities participate totally in the identifying relationship, since they can't exist in the database independently of their owning entity.

In the ER diagram, we show weak entities and identifying relationships with double lines, and the partial key of a weak entity with a dashed underline, as in Figure 4-9. A weak entity is uniquely identified in the context of its regular (or *strong*) entity, and so the full key for a weak entity is the combination of its own (partial) key with the key of its owning entity. To uniquely identify a child in our example, we need the first name of the child and the email address of the child's parent.

Figure 4-10 shows a summary of the symbols we've explained for ER diagrams.

Entity Relationship Modeling Examples

Earlier in this chapter, we showed you how to design a database and understand an Entity Relationship (ER) diagram. This section explains the requirements for our three example databases—`music`, `university`, and `flight`—and shows you their Entity Relationship diagrams:

- The `music` database is designed to store details of a music collection, including the albums in the collection, the artists who made them, the tracks on the albums, and when each track was last played.

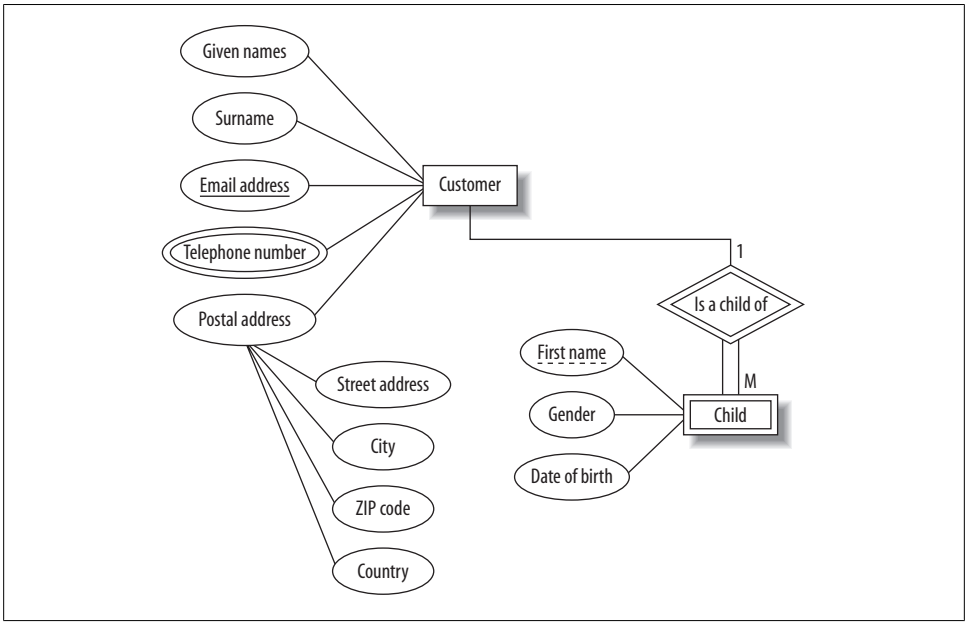

Figure 4-9. The ER diagram representation of a weak entity

- The `university` database captures the details of students, courses, and grades for a university.
- The `flight` database stores an airline timetable of flight routes, times, and the plane types.

The next section explains these databases, each with its ER diagram and an explanation of the motivation for its design. You'll find that understanding the ER diagrams and the explanations of the database designs is sufficient to work with the material in this chapter. We'll show you how to create the `music` database on your MySQL server in Chapter 5.

The Music Database

The `music` database stores details of a personal music library, and could be used to manage your MP3, CD, or vinyl collection. Because this database is for a personal collection, it's relatively simple and stores only the relationships between artists, albums, and tracks. It ignores the requirements of many music genres, making it most useful for storing popular music and less useful for storing jazz or classical music. (We discuss some shortcomings of these requirements at the end of the section in "What it doesn't do.")

We first draw up a clear list of requirements for our database:

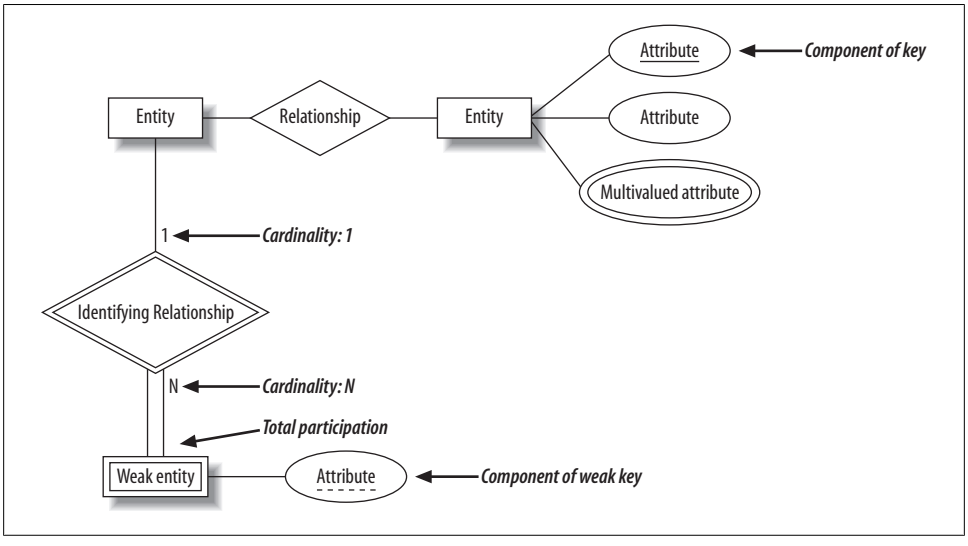

Figure 4-10. Quick summary of the ER diagram symbols

- The collection consists of albums.
- An album is made by exactly one artist.
- An artist makes one or more albums.
- An album contains one or more tracks
- Artists, albums, and tracks each have a name.
- Each track is on exactly one album.
- Each track has a time length, measured in seconds.
- When a track is played, the date and time the playback began (to the nearest second) should be recorded; this is used for reporting when a track was last played, as well as the number of times music by an artist, from an album, or a track has been played.

There's no requirement to capture composers, group members or sidemen, recording date or location, the source media, or any other details of artists, albums, or tracks.

The ER diagram derived from our requirements is shown in Figure 4-11. You'll notice that it consists of only one-to-many relationships: one artist can make many albums, one album can contain many tracks, and one track can be played many times. Conversely, each play is associated with one track, a track is on one album, and an album is by one artist. The attributes are straightforward: artists, albums, and tracks have names, as well as identifiers to uniquely identify each entity. The track entity has a time attribute to store the duration, and the played entity has a timestamp to store when the track was played.

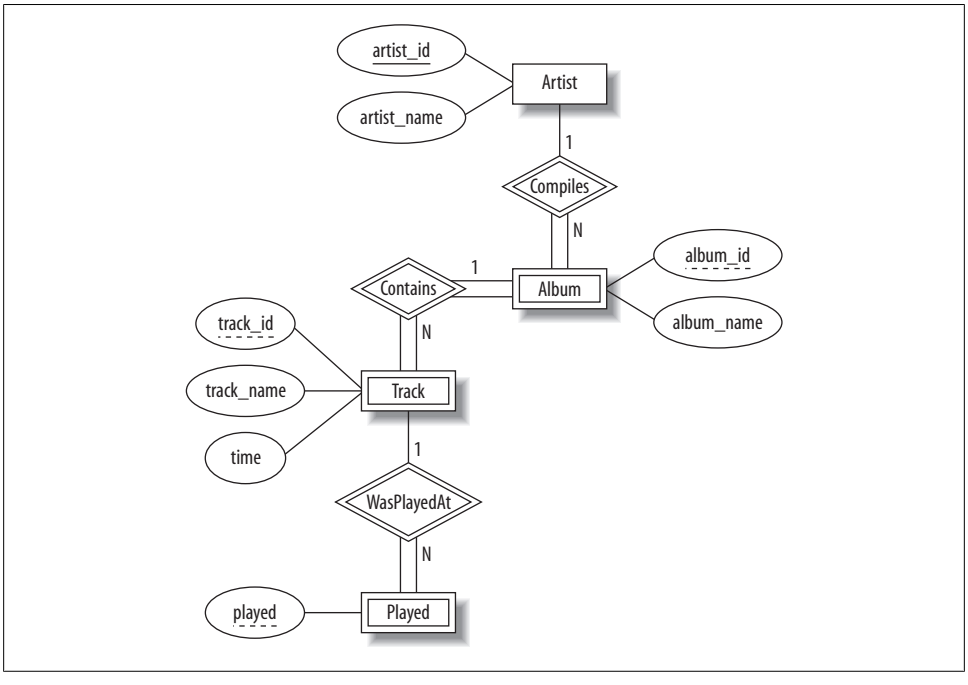

Figure 4-11. The ER diagram of the music database

The only strong entity in the database is `Artist`, which has an `artist_id` attribute that uniquely identifies it. Each `Album` entity is uniquely identified by its `album_id` combined with the `artist_id` of the corresponding `Artist` entity. A `Track` entity is similarly uniquely identified by its `track_id` combined with the related `album_id` and `artist_id` attributes. The `Played` entity is uniquely identified by a combination of its `played` time, and the related `track_id`, `album_id`, and `artist_id` attributes.

What it doesn't do

We've kept the `music` database simple because adding extra features doesn't help you learn anything new, it just makes the explanations longer. If you wanted to use the `music` database in practice, then you might consider adding the following features:

- Support for compilations or various-artists albums, where each track may be by a different artist and may then have its own associated album-like details such as a recording date and time. Under this model, the album would be a strong entity, with many-to-many relationships between artists and albums.
- Playlists, a user-controlled collection of tracks. For example, you might create a playlist of your favorite tracks from an artist.
- Track ratings, to record your opinion on how good a track is.

- Source details, such as when you bought an album, what media it came on, how much you paid, and so on.
- Album details, such as when and where it was recorded, the producer and label, the band members or sidemen who played on the album, and even its artwork.
- Smarter track management, such as modeling that allows the same track to appear on many albums.

The University Database

The `university` database stores details about university students, courses, the semester a student took a particular course (and his mark and grade if he completed it), and what degree program each student is enrolled in. The database is a long way from one that'd be suitable for a large tertiary institution, but it does illustrate relationships that are interesting to query, and it's easy to relate to when you're learning SQL. We explain the requirements next and discuss their shortcomings at the end of this section.

Consider the following requirements list:

- The university offers one or more programs.
- A program is made up of one or more courses.
- A student must enroll in a program.
- A student takes the courses that are part of her program.
- A program has a name, a program identifier, the total credit points required to graduate, and the year it commenced.
- A course has a name, a course identifier, a credit point value, and the year it commenced.
- Students have one or more given names, a surname, a student identifier, a date of birth, and the year they first enrolled. We can treat all given names as a single object —for example, "John Paul."
- When a student takes a course, the year and semester he attempted it are recorded. When he finishes the course, a grade (such as A or B) and a mark (such as 60 percent) are recorded.
- Each course in a program is sequenced into a year (for example, year 1) and a semester (for example, semester 1).

The ER diagram derived from our requirements is shown in Figure 4-12. Although it is compact, the diagram uses some advanced features, including relationships that have attributes and two many-to-many relationships.

In our design:

- `Student` is a strong entity, with an identifier, `student_id`, created to be the primary key used to distinguish between students (remember, we could have several students with the same name).

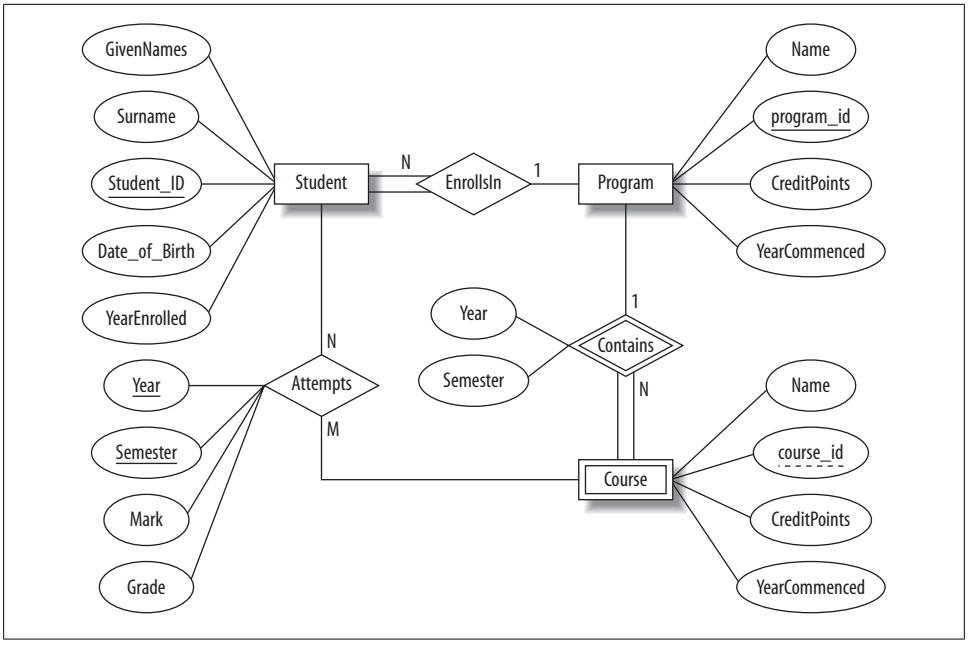

Figure 4-12. The ER diagram of the university database

- `Program` is a strong entity, with the identifier `program_id` as the primary key used to distinguish between programs.
- Each student must be enrolled in a program, so the `Student` entity participates totally in the many-to-one `EnrollsIn` relationship with `Program`. A program can exist without having any enrolled students, so it participates partially in this relationship.
- A `Course` has meaning only in the context of a `Program`, so it's a weak entity, with `course_id` as a weak key. This means that a `Course` is uniquely identified using its `course_id` and the `program_id` of its owning program.
- As a weak entity, `Course` participates totally in the many-to-one identifying relationship with its owning `Program`. This relationship has `Year` and `Semester` attributes that identify its sequence position.
- `Student` and `Course` are related through the many-to-many `Attempts` relationships; a course can exist without a student, and a student can be enrolled without attempting any courses, so the participation is not total.
- When a student attempts a course, there are attributes to capture the `Year` and `Semester`, and the `Mark` and `Grade`.

What it doesn't do

Our database design is rather simple, but this is because the requirements are simple. For a real university, many more aspects would need to be captured by the database. For example, the requirements don't mention anything about campus, study mode, course prerequisites, lecturers, timetabling details, address history, financials, or assessment details. The database also doesn't allow a student to be in more than one degree program, nor does it allow a course to appear as part of different programs.

The Flight Database

The `flight` database stores details about an airline's fleet, flights, and seat bookings. Again, it's a hugely simplified version of what a real airline would use, but the principles are the same.

Consider the following requirements list:

- The airline has one or more airplanes.
- An airplane has a model number, a unique registration number, and the capacity to take one or more passengers.
- An airplane flight has a unique flight number, a departure airport, a destination airport, a departure date and time, and an arrival date and time.
- Each flight is carried out by a single airplane.
- A passenger has given names, a surname, and a unique email address.
- A passenger can book a seat on a flight.

The ER diagram derived from our requirements is shown in Figure 4-13:

- An `Airplane` is uniquely identified by its `RegistrationNumber`, so we use this as the primary key.
- A `Flight` is uniquely identified by its `FlightNumber`, so we use the flight number as the primary key. The departure and destination airports are captured in the `From` and `To` attributes, and we have separate attributes for the departure and arrival date and time.
- Because no two passengers will share an email address, we can use the `EmailAddress` as the primary key for the `Passenger` entity.
- An airplane can be involved in any number of flights, while each flight uses exactly one airplane, so the `Flies` relationship between the `Airplane` and `Flight` relationships has cardinality 1:N; because a flight cannot exist without an airplane, the `Flight` entity participates totally in this relationship.
- A passenger can book any number of flights, while a flight can be booked by any number of passengers. As discussed earlier in "Intermediate Entities," we could specify an M:N `Books` relationship between the `Passenger` and `Flight` relationship,

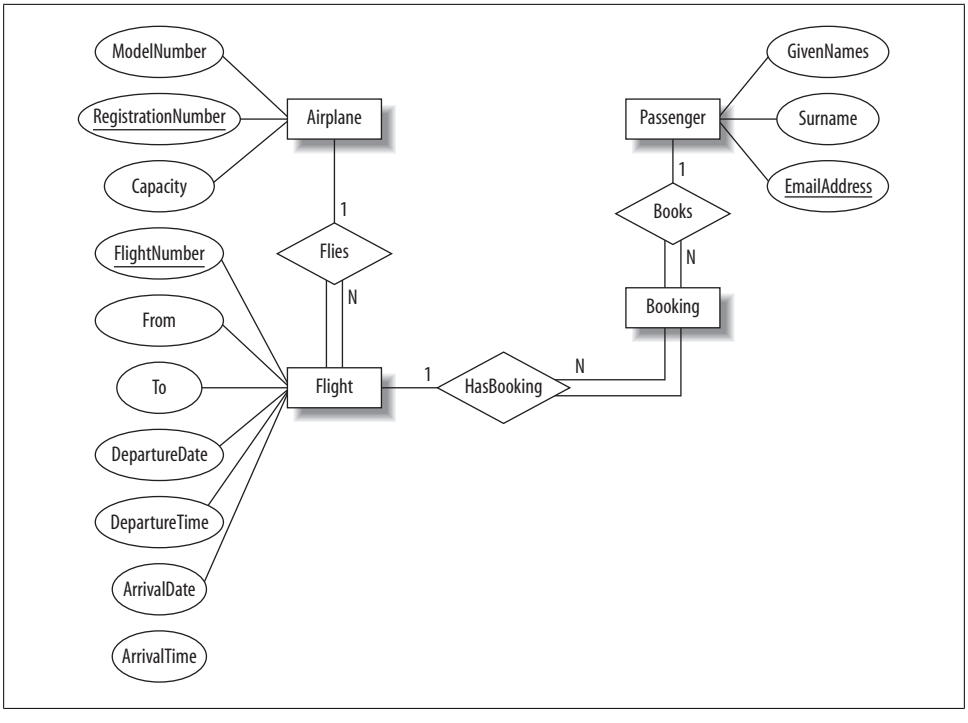

Figure 4-13. The ER diagram of the flight database

but considering the issue more carefully shows that there is a hidden entity here: the booking itself. We capture this by creating the intermediate entity `Booking` and 1:N relationships between it and the `Passenger` and `Flight` entities. Identifying such entities allows us to get a better picture of the requirements. Note that even if we didn't notice this hidden entity, it would come out as part of the ER-to-tables mapping process we'll describe next in "Using the Entity Relationship Model."

What it doesn't do

Again, this is a very simple flight database. There are no requirements to capture passenger details such as age, gender, or frequent-flier number.

We've treated the capacity of the airplane as an attribute of an individual airplane. If, instead, we assumed that the capacity is determined by the model number, we would have created a new `AirplaneModel` entity with the attributes `ModelNumber` and `Capacity`. The `Airplane` entity would then not have a `Capacity` attribute.

We've mapped a different flight number to each flight between two destinations. Airlines typically use a flight number to identify a given flight path and schedule, and they specify the date of the flight independently of the flight number. For example, there is one IR655 flight on April 1, another on April 2, and so on. Different airplanes can

operate on the same flight number over time; our model would need to be extended to support this.

The system also assumes that each leg of a multihop flight has a different `FlightNumber`. This means that a flight from Dubai to Christchurch via Singapore and Melbourne would need a different `FlightNumber` for the Dubai-Singapore, Singapore-Melbourne, and Melbourne-Christchurch legs.

Our database also has limited ability to describe airports. In practice, each airport has a name, such as "Melbourne Regional Airport," "Mehrabad," or "Tullamarine." The name can be used to differentiate between airports, but most passengers will just use the name of the town or city. This can lead to confusion, when, for example, a passenger could book a flight to Melbourne, Florida, USA, instead of Melbourne, Victoria, Australia. To avoid such problems, the International Air Transport Association (IATA) assigns a unique airport code to each airport; the airport code for Melbourne, Florida, USA is MLB, while the code for Melbourne, Victoria, Australia is MEL. If we were to model the airport as a separate entity, we could use the IATA-assigned airport code as the primary key. Incidentally, there's an alternative set of airport codes assigned by the International Civil Aviation Organization (ICAO); under this code, Melbourne, Florida is KMLB, and Melbourne, Australia is YMML.

Using the Entity Relationship Model

In this section, we'll look at the steps required to manually translate an ER model into database tables. We'll then perform these steps using the `music` database as an example. In "Using Tools for Database Design," we'll see how we can automate this process with the MySQL Workbench tool.

Mapping Entities and Relationships to Database Tables

When converting an ER model to a database schema, we work through each entity and then through each relationship according to the following rules to end up with a set of database tables.

Map the entities to database tables

For each strong entity, create a table comprising its attributes and designate the primary key. The parts of any composite attributes are also included here.

For each weak entity, create a table comprising its attributes and including the primary key of its owning entity. The primary key of the owning entity is known as a *foreign key* here, because it's a key not of this table, but of another table. The primary key of the table for the weak entity is the combination of the foreign key and the partial key of the weak entity. If the relationship with the owning entity has any attributes, add them to this table.

For each multivalued attribute of an entity, create a table comprising the entity's primary key and the attribute.

Map the relationships to database tables

For each one-to-one relationship between two entities, include the primary key of one entity as a foreign key in the table belonging to the other. If one entity participates totally in the relationship, place the foreign key in its table. If both participate totally in the relationship, consider merging them into a single table.

For each nonidentifying one-to-many relationship between two entities, include the primary key of the entity on the "1" side as a foreign key in the table for the entity on the "N" side. Add any attributes of the relationship in the table alongside the foreign key. Note that identifying one-to-many relationships (between a weak entity and its owning entity) are captured as part of the entity-mapping stage.

For each many-to-many relationship between two entities, create a new table containing the primary key of each entity as the primary key, and add any attributes of the relationship. This step helps to identify intermediate entities.

For each relationship involving more than two entities, create a table with the primary keys of all the participating entities, and add any attributes of the relationship.

Converting the Music Database ER Model to a Database Schema

Following the mapping rules as just described, we first map entities to database tables:

- For the strong entity `Artist`, we create the table `artist` comprising the attributes `artist_id` and `artist_name`, and designate `artist_id` as the primary key.
- For the weak entity `Album`, we create the table `album` comprising the attributes `album_id` and `album_name`, and include the primary key `artist_id` of the owning `Artist` entity as a foreign key. The primary key of the `album` table is the combination {`artist_id`, `album_id`}.
- For the weak entity `Track`, we create the table `track` comprising the attributes `track_id`, `track_name`, and `time`, and include the primary key {`artist_id`, `album_id`} of the owning `Album` entity as a foreign key. The primary key of the `track` table is the combination {`artist_id`, `album_id`, `track_id`}.
- For the weak entity `Played`, we create the table `played` comprising the attribute `played`, and include the primary key {`artist_id`, `album_id`, `track_id`} of the owning `Track` entity as a foreign key. The primary key of the `played` table is the combination {`artist_id`, `album_id`, `track_id`, `played`}.
- There are no multivalued attributes in our design, nor are there any nonweak relationships between our entities, so our mapping is complete here.

You don't have to use consistent names across all tables; for example, you could have a column `musician` in the `album` table that contains the artist ID that you call

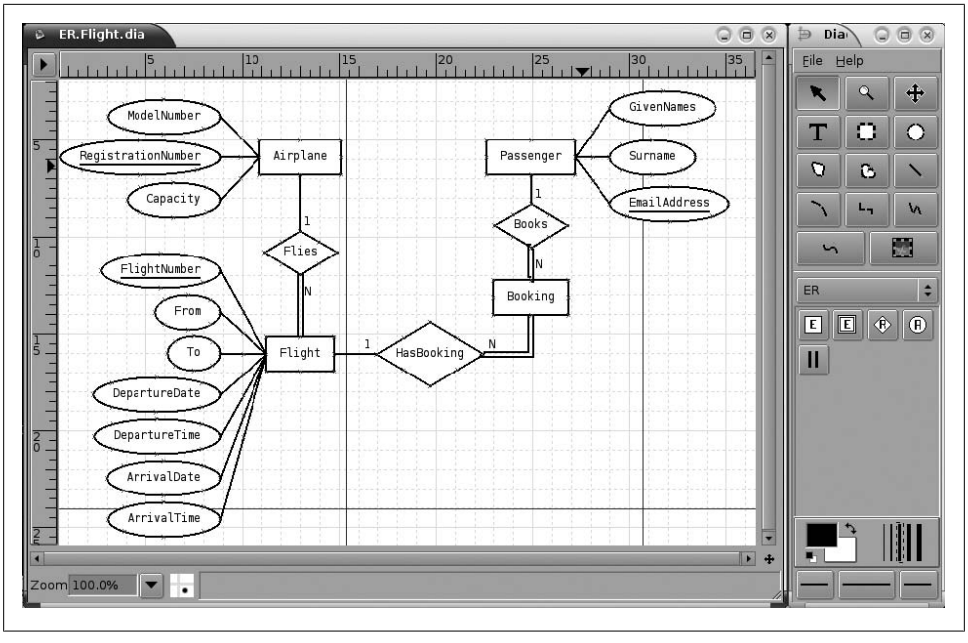

Figure 4-14. Using the Dia program to draw an ER diagram

`artist_id` in the `artist` table. Obviously, it's much better to use a consistent naming convention to avoid confusion. Some designers put `fk_` in front of columns that contain foreign keys; for example, in the `album` table, we could store the artist ID in the `fk_artist_id` column. We don't use this convention in this book.

Using Tools for Database Design

It's a good idea to use a tool to draw your ER diagrams; this way, you can easily edit diagrams as you refine your designs, and the final diagram is clear and unambiguous. There are a large number of programs that can be used for this purpose. A good free tool that is available for both Linux and Windows is Dia; you can download the latest version from *http://www.gnome.org/projects/dia*. Mac OS X users can use the OmniGraffle program that comes bundled with the operating system. Windows users can also use Microsoft Visio.

A screenshot of Dia is shown in Figure 4-14. When you open the program, you should first select the ER "sheet" of shapes from the drop-down list in the middle of the control window (where the ER label appears at the right of the figure) and then select from the entity and relation shapes.

You can assign properties to shapes by double-clicking on them. For example, you can mark an attribute as being a key or a weak key, and you can mark an entity's participation in a relation as being total or partial.

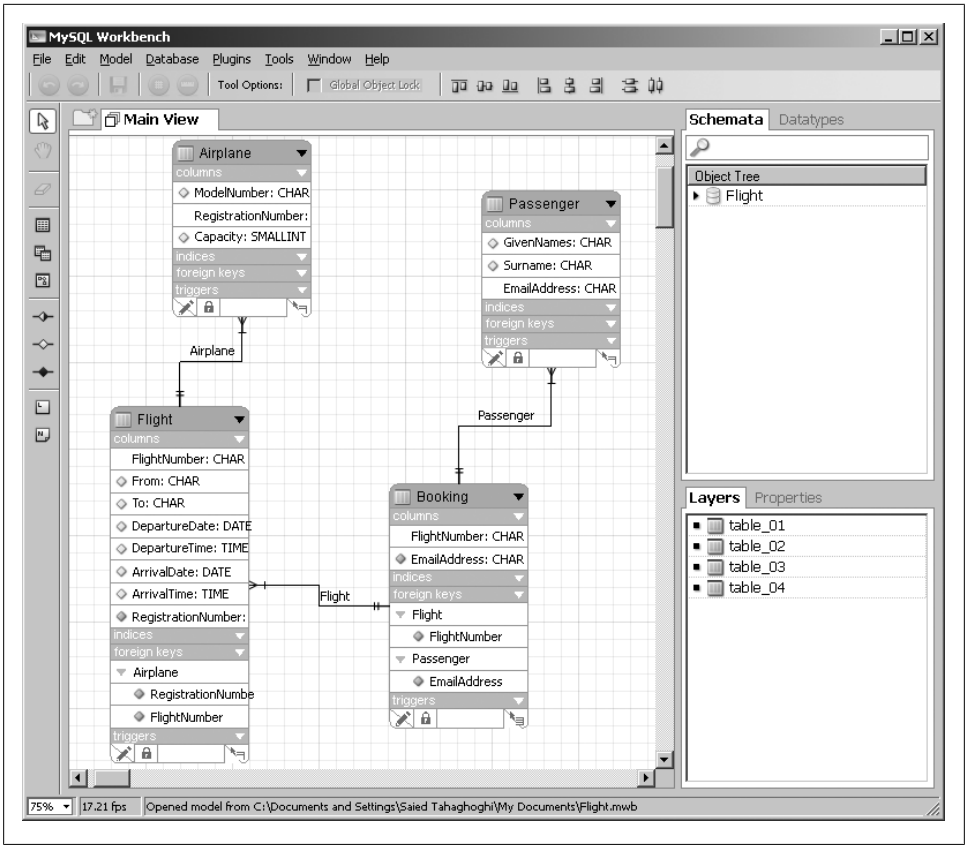

Figure 4-15. A screenshot of the MySQL Workbench program to design the Flight database

The open source MySQL Workbench program is a very powerful visual database design tool available as part of the MySQL GUI Tools Bundle from the MySQL AB downloads page at *http://dev.mysql.com/downloads*.

Figure 4-15 shows a screenshot of using MySQL Workbench to design the `flight` database. You can select tables and relations from the toolbar icons on the left of the screen, and double-click on each object to set properties such as attributes and relationship cardinality.

A very useful feature of MySQL Workbench is that it can export your design as SQL statements ready to use on a MySQL database. Even better, it can connect to a MySQL database to export a design directly. You can also reverse-engineer an ER model from an existing database, edit the model, and then export the modified design back to the MySQL database. Note that this program is currently in the beta testing phase, so you should use it with care.

Resources

To learn more about database fundamentals, including ER modeling, we recommend the following books:

- *Database Management Systems* by Raghu Ramakrishnan and Johannes Gehrke (McGraw-Hill).
- *Fundamentals of Database Systems* by Ramez Elmasri and Shamkant B. Navathe (Addison-Wesley).
- *Database Systems: A Practical Approach to Design, Implementation and Management* by Thomas M. Connolly and Carolyn E. Begg (Addison-Wesley).

Exercises

1. When would you use a weak entity?
2. Is it better to use entities instead of attributes?
3. Alter and extend the `music` database ER model so that it can store compilations, where a *compilation* is an album that contains tracks by two or more different artists.
4. Create an ER diagram for an online media store using the following requirements:
   - There are two types of product: music CDs and video DVDs.
   - Customers can buy any number of each product.
   - For each CD, store the title, the artist's name, the label (publisher), and the price. Also store the number, title, and length (in seconds) of each track on the CD.
   - For each video DVD, store the title the studio name, and the price.

   Tables 4-1 and 4-2 contain some sample data to help you better understand the requirements.

Table 4-1. Video DVDs

Title	Studio	Price
Leon—The Professional	Sony Pictures	$21.99
Chicken Run	Dreamworks Video	$19.99

Table 4-2. Music CDs

Title	Artist	Label	Price
Come Away With Me	Norah Jones	Blue Note Records	$11.99
Feels Like Home	Norah Jones	Blue Note Records	$11.99
The Joshua Tree	U2	Island	$10.99
Brothers in Arms	Dire Straits	Vertigo	$9.99

Table 4-3 contains a sample list of music CD track titles and length in seconds for the CD with the title "Come Away With Me" by the artist Norah Jones.

Table 4-3. Tracks

Number	Name	Length
1	Don't Know Why	186
2	Seven Years	145
3	Cold, Cold Heart	218
4	Feelin' the Same Way	177
5	Come Away with Me	198
6	Shoot the Moon	236
7	Turn Me On	154
8	Lonestar	186
9	I've Got to See You Again	253
10	Painter Song	162
11	One Flight Down	185
12	Nightingale	252
13	The Long Day Is Over	164
14	The Nearness of You	187