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.
* 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 musicdatabase on your MySQL server in [Chapter 5](https://www.oreilly.com/library/view/learning-mysql/0596008643/ch05.html).

The Music Database

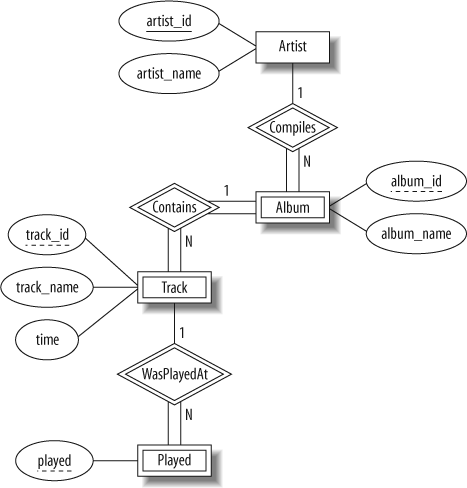
The music database stores details of a personal music library, and could be used to manage your MP3, CD, or vinyl collection. Because this database is for a personal collection, it’s relatively simple and stores only the relationships between artists, albums, and tracks. It ignores the requirements of 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](https://www.oreilly.com/library/view/learning-mysql/0596008643/ch04s04.html#BAS-SEC-MUSICNO).”)

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

* 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](https://www.oreilly.com/library/view/learning-mysql/0596008643/ch04s04.html#BAS-FIG-MUSICER). 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\_idattribute that uniquely identifies it. Each Album entity is uniquely identified by its album\_id combined with the artist\_id of the corresponding Artistentity. A Track entity is similarly uniquely identified by its track\_idcombined with the related album\_id and artist\_id attributes. The Playedentity 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.

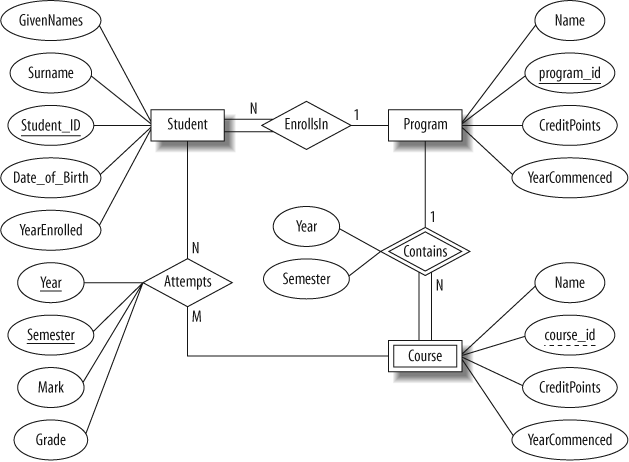
The University Database

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](https://www.oreilly.com/library/view/learning-mysql/0596008643/ch04s04.html#BAS-FIG-UNIER). Although it is compact, the diagram uses some advanced features, including relationships that have attributes and two many-to-many relationships.



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

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).
* 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 Attemptsrelationships; 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 Yearand 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.

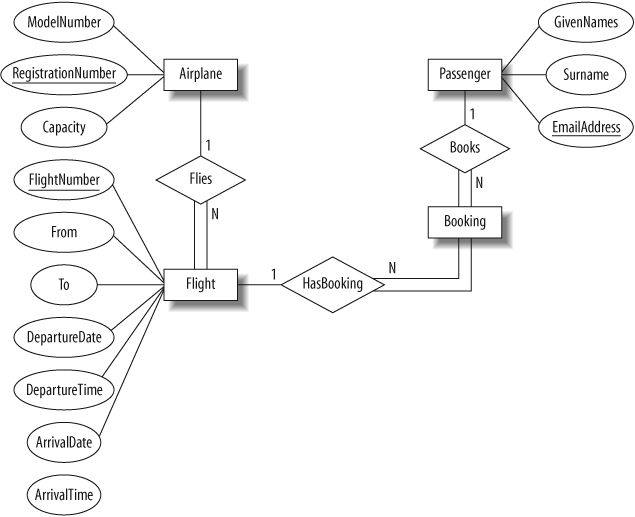
The Flight Database

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](https://www.oreilly.com/library/view/learning-mysql/0596008643/ch04s04.html#BAS-FIG-FLIGHT):



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

* 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](https://www.oreilly.com/library/view/learning-mysql/0596008643/ch04s03.html#SEC-ER-INTERMEDIATE),” we could specify an M:N Books relationship between the Passenger andFlight relationship, 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](https://www.oreilly.com/library/view/learning-mysql/0596008643/ch04s05.html).”

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.