8

Database Design

Once you install your DBMS software on your computer, it can be very tempting to just jump right into creating a database without much thought or planning. As with any software development, this kind of ad hoc approach works with only the simplest of problems. If you expect your database to support any kind of complexity, some planning and design will definitely save you time in the long run. You will need to take a look at what details are important to good database design.

Database Design Primer

Suppose you have a large collection of compact discs and you want to create a database to track them. The first step is to determine what the data that you are going to store is about. One good way to start is to think about why you want to store the data in the first place. In our case, we most likely want to be able to look up CDs by artist, title, and song. Since we want to look up those items, we know they must be included in the database. In addition, it is often useful to simply list items that should be tracked. One possible list might include: CD title, record label, band name, song title. As a starting point, we will store the data in the table shown in Table 2-1.

Table 2-1: A CD Database Made Up of a Single Table (continued)

Band Name	CD Title	Record Label	Songs
Stevie Wonder	Talking Book	Motown	You Are the Sunshine of My Life, Maybe Your Baby, Superstition, . . .
Miles Davis Quintet	Miles Smiles	Columbia	Orbits, Circle, . . .
Wayne Shorter	Speak No Evil	Blue Note	Witch Hunt, Fee-Fi-Fo-Fum
Herbie Hancock	Headhunters	Columbia	Chameleon, Watermelon

			Man, . . .
Herbie Hancock	Maiden Voyage	Blue Note	Maiden Voyage

(For brevity's sake, we have left out most of the songs.) At first glance, this table seems like it will meet our needs since we are storing all of the data we need. Upon closer inspection, however, we find several problems. Take the example of Herbie Hancock. "Band Name" is repeated twice: once for each CD. This repetition is a problem for several reasons. First, when entering data in the database, we end up typing the same name over and over. Second, and more important, if any of the data changes, we have to update it in multiple places. For example, what if "Herbie" were misspelled? We would have to update the data in each of the two rows. The same problem would occur if the name Herbie Hancock changes in the future (à la Jefferson Airplane or John Cougar). As we add more Herbie Hancock CDs to our collection, we add to the amount of effort required to maintain data consistency.

Another problem with the single CD table lies in the way it stores songs. We are storing them in the CD table as a list of songs in a single column. We will run into all sorts of problems if we want to use this data meaningfully. Imagine having to enter and maintain that list. And what if we want to store the length of the songs as well? What if we want to perform a search by song title? It quickly becomes clear that storing the songs in this fashion is undesirable.

This is where database design comes into play. One of the main purposes of database design is to eliminate redundancy from the database. To accomplish this task, we use a technique called *normalization*. Before we start with normalization, let's start with some fundamental relational database concepts. A data model is a diagram that illustrates your database design. It is made up of three main elements: entities, attributes, and relationships. For now, let's focus on entities and attributes; we will take a look at relationships later.

Database Entities

An *entity* is a thing or object of importance about which data must be captured. All "things" are not entities, only those things about which you need to capture information. Information about an entity is captured in the form of attributes and/or relationships. If something is a candidate for being an entity and it has no attributes or relationships, it is not really an entity. Database entities appear in a data model as a box with a title. The title is the name of the entity.

Entity Attributes

An *attribute* describes information about an entity that must be captured. Each entity has zero or more attributes that describe it, and each attribute describes exactly one entity. Each entity instance (row in the table) has exactly one value, possibly `NULL`, for each of its attributes. An attribute value can be numeric, a character string, date, time, or some other basic data value. In the first step of database design, logical data modeling, we do not worry about how the attributes will be stored.

NULL provides the basis for the problem of dealing with missing information. It is specifically used for the case in which you lack a certain piece of information. As an example, consider the situation where a CD does not list the song lengths of each of its tracks. Each song has a length, but you cannot tell from the case what that length is. You do not want to store the length as zero, since that would be incorrect. Instead, you store the length as NULL. If you are thinking you could store it as zero and use zero to mean "unknown", you are falling into one of the same traps that led to one of the Y2K problems. Not only did old systems store years as two digits, but they often gave a special meaning to 9-9-99.

Our example database refers to a number of things: the CD, the CD title, the band name, the songs, and the record label. Which of these are entities and which are attributes?

Data Model

Notice that we capture several pieces of data (CD title, band name, etc.) about each CD, and we absolutely cannot describe a CD without those items. CD is therefore one of those things we want to capture data about and is likely an entity. To start a data model, we will diagram it as an entity. Figure 2-1 shows our sole entity in a data model.

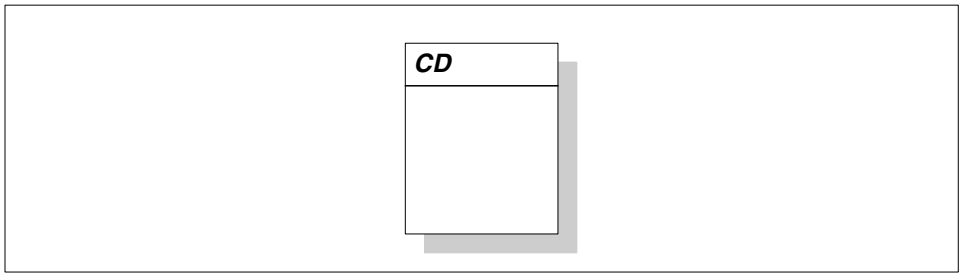

Figure 2-1: The CD entity in a data model

By common entity naming conventions, an entity name must be singular. We therefore call the table where we store CDs "CD" and not "CDs." We use this convention because each entity names an instance. For example, the "San Francisco 49ers" is an instance of "Football Team," not "Football Teams."

At first glance, it appears that the rest of the database describes a CD. This would indicate that they are attributes of CD. Figure 2-3 adds them to the CD entity in Figure 2-1. In a data model, attributes appear as names listed in their entity's box.

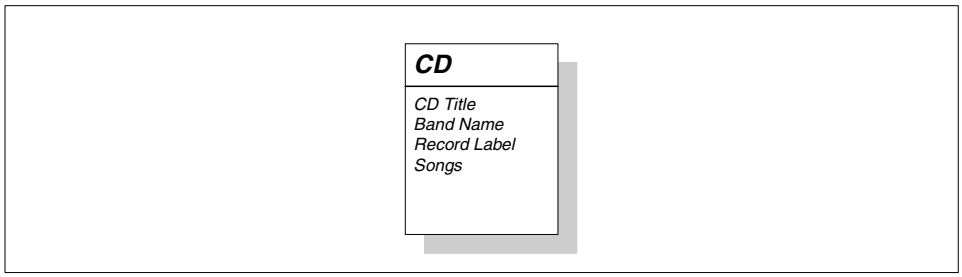

Figure 2-2: The CD entity with its attributes

This diagram is simple, but we are not done yet. In fact, we have only just begun. Earlier, we discussed how the purpose of data modeling is to eliminate redundancy using a technique called normalization. We have a nice diagram for our database, but we have not gotten rid of the redundancy as we set out to do. It is now time to normalize our database.

Normalization

E.F. Codd, then a researcher for IBM, first presented the concept of database normalization in several important papers written in the 1970s. The aim of normalization remains the same today: to eradicate certain undesirable characteristics from a database design. Specifically, the goal is to remove certain kinds of data redundancy and therefore avoid update anomalies. Update anomalies are difficulties with the insert, update, and delete operations on a database due to the data structure. Normalization additionally aids in the production of a design that is a high-quality representation of the real world; thus normalization increases the clarity of the data model.

As an example, say we misspelled "Herbie Hancock" in our database and we want to update it. We would have to visit each CD by Herbie Hancock and fix the artist's name. If the updates are controlled by an application which enables us to edit only one record at a time, we end up having to edit many rows. It would be much more desirable to have the name "Herbie Hancock" stored only once so we have to maintain it in just one place.

First Normal Form (1NF)

The general concept of normalization is broken up into several "normal forms." An entity is said to be in the first normal form when all attributes are single-valued. To apply the first normal form to an entity, we have to verify that each attribute in the entity has a single value for each instance of the entity. If any attribute has repeating values, it is not in 1NF.

A quick look back at our database reveals that we have repeating values in the Songs attribute, so the `CD` is clearly not in 1NF. To remedy this problem, an entity with repeating values indicates that we have missed at least one other entity. One way to discover other entities is to look at each attribute and ask the question "What thing does this describe?"

What does `Song` describe? It lists the songs on the CD. So `Song` is another "thing" that we capture data about and is probably an entity. We will add it to our diagram and give it a `Song Name` attribute. To complete the `Song` entity, we need to ask if there is more about a `Song` that we would like to capture. We identified earlier song length as something we might want to capture. Figure 2-3 shows the new data model.

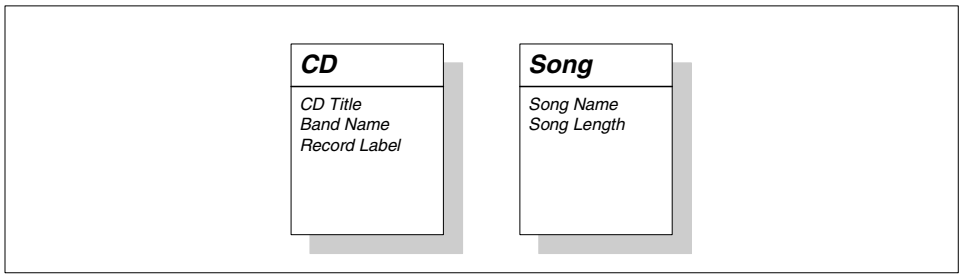

Figure 2-3: A data model with CD and Song entities

Now that the `Song Name` and `Song Length` are attributes in a `Song` entity, we have a data model with two entities in 1NF. None of their attributes contain multiple values. Unfortunately, we have not shown any way of relating a `CD` to a `Song`.

The Unique Identifier

Before discussing relationships, we need to impose one more rule on entities. Each entity must have a unique identifier—we'll call it the ID. An *ID* is an attribute of an entity that meets the following rules:

- It is unique across all instances of the entity.
- It has a non-`NULL` value for each instance of the entity, for the entire lifetime of the instance.
- It has a value that never changes for the entire lifetime of the instance.

The ID is very important because it gives us a way to know which instance of an entity we are dealing with. Identifier selection is critical because it is also used to model relationships. If, after you've selected an ID for an entity, you find that it doesn't meet one of the above rules, this could affect your entire data model.

Novice data modelers often make the mistake of choosing attributes that should not be identifiers and making them identifiers. If, for example, you have a `Person` entity, it might be tempting to use the `Name` attribute as the identifier because all people have a name and that name never changes. But what if a person marries? What if the person decides to legally change his name? What if you misspelled the name when you first entered it? If any of these events causes a name change, the third rule of identifiers is violated. Worse, is a name really ever unique? Unless you can guarantee with 100% certainty that the `Name` is unique, you will be violating the first rule. Finally, you do know that all `Person` instances have non-`NULL` names. But are you certain that you will always know the name of a `Person` when you first enter information about them in the database? Depending on your application processes, you may not know the name of a `Person` when a record is first created. The lesson to be learned is that there are many problems with taking a non-identifying attribute and making it one.

The solution to the identifier problem is to invent an identifying attribute that has no other meaning except to serve as an identifying attribute. Because this attribute is invented and completely unrelated to the entity, we have full control over it and guarantee that it meets the rules of unique identifiers. Figure 2-4 adds invented ID attributes to each of our entities. A unique identifier is diagrammed as an underlined attribute.

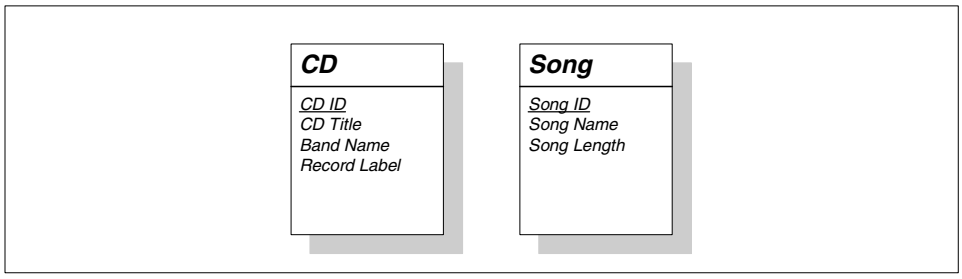

Figure 2-4: The CD and Song entities with their unique identifiers

Relationships

The identifiers in our entities enable us to model their relationships. A relationship describes a binary association between two entities. A relationship may also exist between an entity and itself. Such a relationship is called a *recursive relationship*. Each entity within a relationship describes and is described by the other. Each side of the relationship has two components: a name and a degree.

Each side of the relationship has a name that describes the relationship. Take two hypothetical entities, an `Employee` and a `Department`. One possible relationship between the two is that an `Employee` is "assigned to" a `Department`. That `Department` is "responsible for" an `Employee`. The `Employee` side of the relationship is thus named "assigned to" and the `Department` side "responsible for."

Degree, also referred to as cardinality, states how many instances of the describing entity must describe one instance of the described entity. Degree is expressed using two different values: "one and only one" (1) and "one or many" (M). An employee is assigned to one department at a time, so `Employee` has a one and only one relationship with `Department`. In the other direction, a department is responsible for many employees. We therefore say `Department` has a "one or many" relationship with `Employee`. As a result a `Department` could have exactly one `Employee`.

It is sometimes helpful to express a relationship verbally. One way of doing this is to plug the various components of a direction of the relationship into this formula:

> *entity1* has [one and only one | one or many] *entity2*

Using this formula, `Employee` and `Department` would be expressed like so:

> Each `Employee` must be assigned to one and only one `Department`.
> Each `Department` may be responsible for one or many `Employees`.

We can use this formula to describe the entities in our data model. A `CD` contains one or many `Songs` and a `Song` is contained on one and only one `CD`. In our data model, this relationship can be shown by drawing a line between the two entities. Degree is expressed with a straight line for "one and only one" relationships or "crows feet" for "one or many" relationships. Figure 2-5 illustrates these conventions.

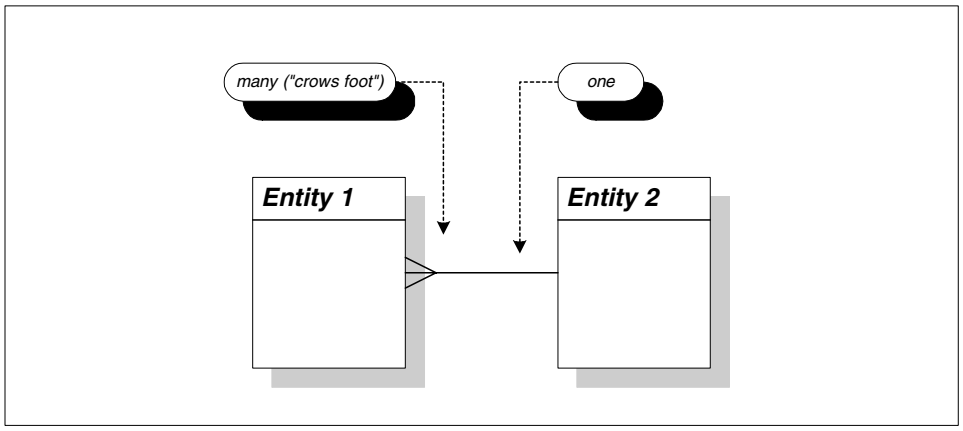

Figure 2-5: Anatomy of a relationship

How does this apply to the relationship between `Song` and `CD`? In reality, a `Song` can be contained on many `CD`s, but we ignore this for the purposes of this example. Figure 2-6 shows the data model with the relationships in place.

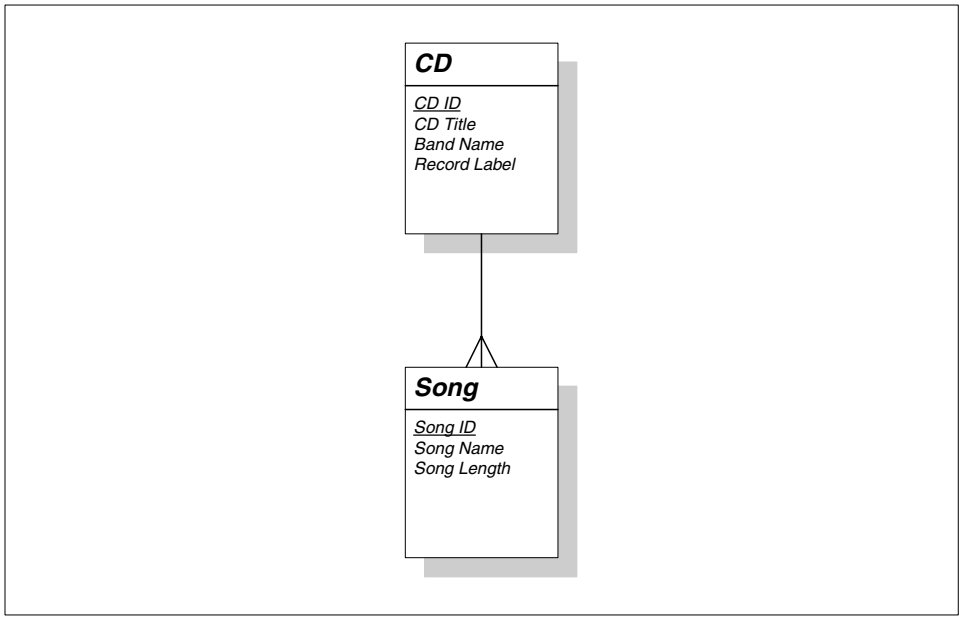

Figure 2-6: CD-Song relationship

With these relationships firmly in place, we can go back to the normalization process and improve upon the design. So far, we have normalized repeating song values into a new entity, Song, and modeled the relationship between it and the `CD` entity.

Second Normal Form (2NF)

An entity is said to be in the second normal form if it is already in 1NF and all non-identifying attributes are dependent on the entity's entire unique identifier. If any attribute is not

dependent entirely on the entity's unique identifier, that attribute has been misplaced and must be removed. Normalize these attributes either by finding the entity where it belongs or by creating an additional entity where the attribute should be placed.

In our example, "Herbie Hancock" is the `Band Name` for two different CDs. This fact illustrates that `Band Name` is not entirely dependent on `CD ID`. This duplication is a problem because if, for example, we had misspelled "Herbie Hancock," we would have to update the value in multiple places. We thus have a sign that `Band Name` should be part of a new entity with some relationship to `CD`. As before, we resolve this problem by asking the question: "What does a band name describe"? It describes a band, or more generally, an artist. Artist is yet another thing we are capturing data about and is therefore probably an entity. We will add it to our diagram with `Band Name` as an attribute. Since all artists may not be bands, we will rename the attribute `Artist Name`. Figure 2-7 shows the new state of the model.

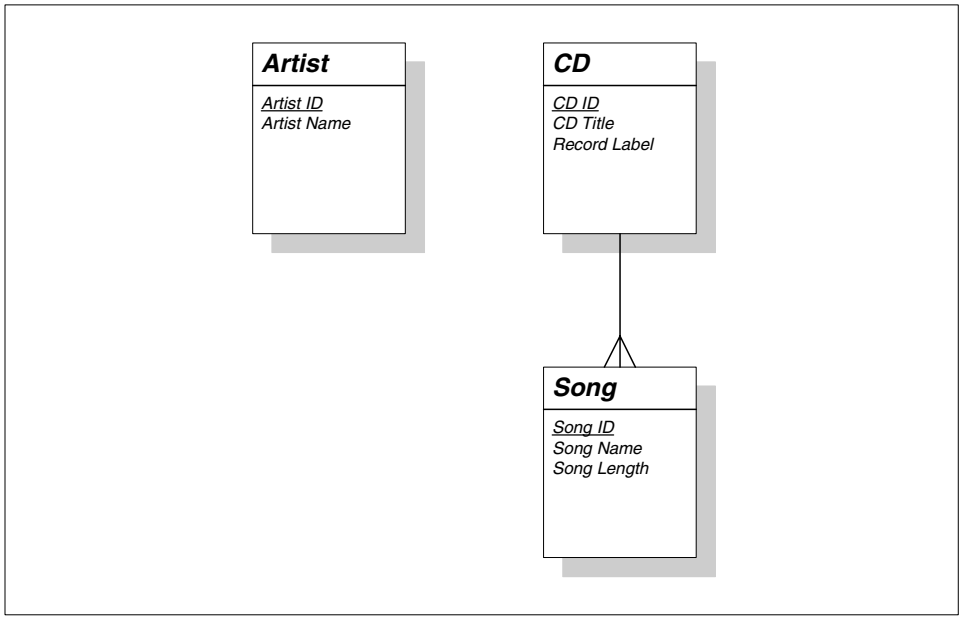

Figure 2-7: The data model with the new Artist entity

Of course, the relationships for the new Artist table are missing. We know that each `Artist` has one or many `CD`s. Each `CD` could have one or many `Artists`. We model this in Figure 2-8.

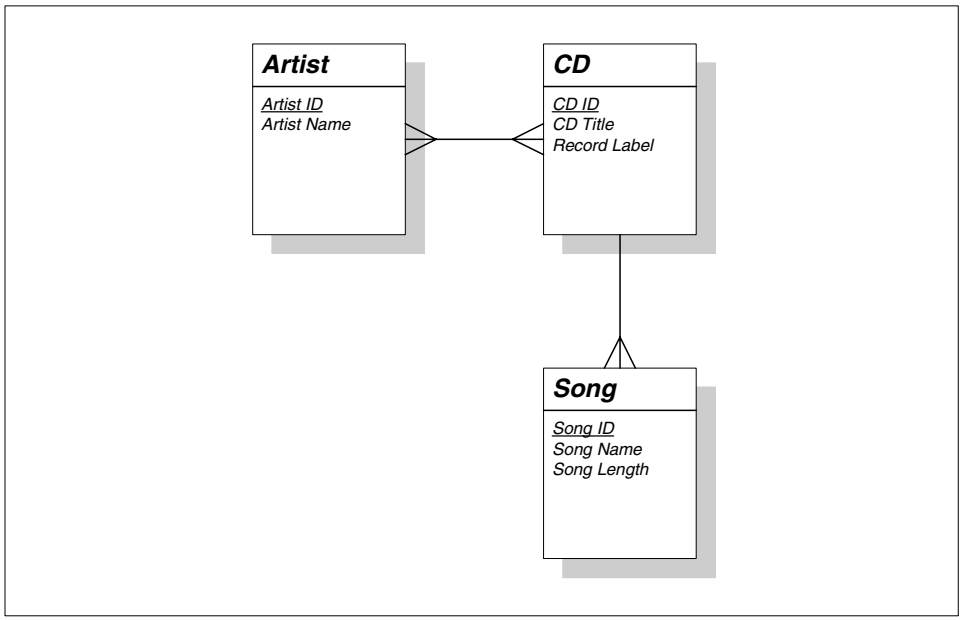

Figure 2-8: The Artist relationships in the data model

We originally had the `Band Name` attribute in the `CD` entity. It thus seemed natural to make `Artist` directly related to `CD`. But is this really correct? On closer inspection, it would seem that there should be a direct relationship between an `Artist` and a `Song`. Each `Artist` has one or more `Songs`. Each `Song` is performed by one and only one `Artist`. The true relationship appears in Figure 2-9.

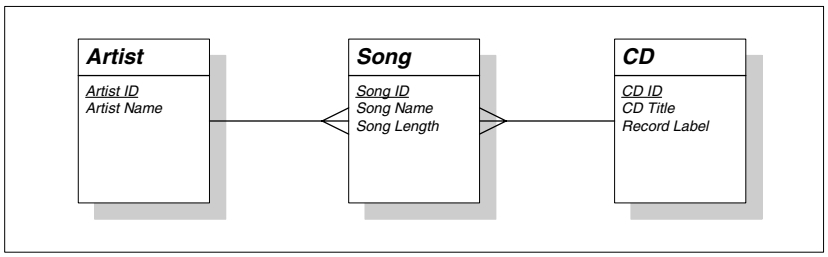

Figure 2-18: The real relationship between Artist and the rest of our data model

Not only does this make more sense than a relationship between `Artist` and `CD`, but it also addresses the issue of compilation `CDs`.

Kinds of Relationships

When modeling the relationship between entities, it is important to determine both directions of the relationship. After both sides of the relationship have been determined, we end up with three main kinds of relationships. If both sides of the relationship have a degree of one and only one, the relationship is called a “one-to-one” or “1-to-1” relationship. As we will find out later, one-to-one relationships are rare. We do not have one in our data model.

If one of the sides has a degree of "one or many" and the other side has a degree of "one and only one," the relationship is a "one-to-many" or "1-to-M" relationship. All of the relationships in our current data model are one-to-many relationships. This is to be expected since one-to-many relationships are the most common.

The final kind of relationship is where both sides of the relationship are "one or many" relationships. These kinds of relationships are called "many-to-many" or "M-to-M" relationships. In an earlier version of our data model, the `Artist-CD` relationship was a many-to-many relationship.

Refining Relationships

As we noted earlier, one-to-one relationships are quite rare. In fact, if you encounter one during your data modeling, you should take a closer look at your design. A one-to-one relationship may imply that two entities are really the same entity. If they do turn out to be the same entity, they should be folded into a single entity.

Many-to-many relationships are more common than one-to-one relationships. In these relationships, there is often some data we want to capture about the relationship. For example, take a look at the earlier version of our data model in Figure 2-15 that had the many-to-many relationship between `Artist` and `CD`. What data might we want to capture about that relationship? An `Artist` has a relationship with a `CD` because an `Artist` has one or more `Songs` on that `CD`. The data model in Figure 2-17 is actually another representation of this many-to-many relationship.

All many-to-many relationships should be resolved using the following technique:

1. Create a new entity (sometimes referred to as a *junction entity*). Name it appropriately. If you cannot think of an appropriate name for the junction entity, name it by combining the names of the two related entities (e.g., `ArtistCD`). In our data model, `Song` is a junction entity for the `Artist-CD` relationship.
2. Relate the new entity to the two original entities. Each of the original entities should have a one-to-many relationship with the junction entity.
3. If the new entity does not have an obvious unique identifier, inherit the identifying attributes from the original entities into the junction entity and make them together the unique identifier for the new entity.

In almost all cases, you will find additional attributes that belong in the new junction entity. If not, the many-to-many relationship still needs to be resolved, otherwise you will have a problem translating your data model into a physical schema.

More 2NF

Our data model is still not in 2NF. The value of the `Record Label` attribute has only one value for each `CD`, but we see the same `Record Label` in multiple `CDs`. This situation is similar to the one we saw with `Band Name`. As with `Band Name`, this duplication indicates that `Record Label` should be part of its own entity. Each `Record Label` releases one or many `CDs`. Each `CD` is released by one and only one `Record Label`. Figure 2-10 models this relationship.

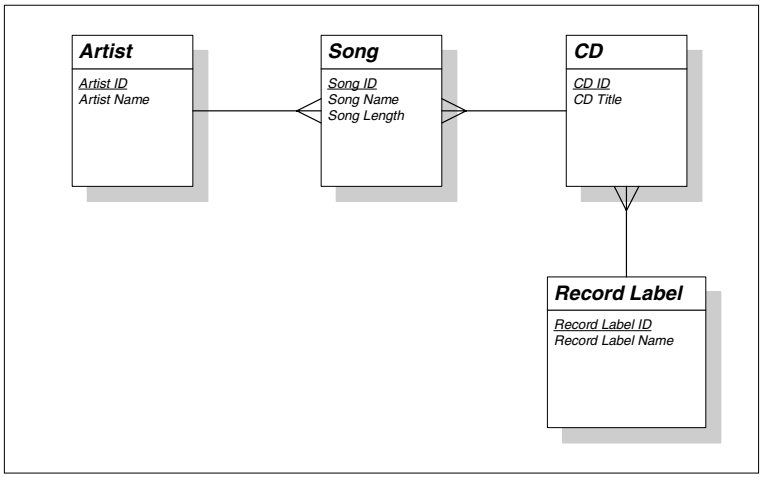

Figure 2-10: Our data model in the second normal form

Third Normal Form (3NF)

An entity is said to be in the third normal form if it is already in 2NF and no non-identifying attributes are dependent on any other non-identifying attributes. Attributes that are dependent on other non-identifying attributes are normalized by moving both the dependent attribute and the attribute on which it is dependent into a new entity.

If we wanted to track `Record Label` address information, we would have a problem for 3NF. The `Record Label` entity with address data would have `State Name` and `State Abbreviation` attributes. Though we really do not need this information to track `CD` data, we will add it to our data model for the sake of our example. Figure 2-11 shows address data in the `Record Label` entity.

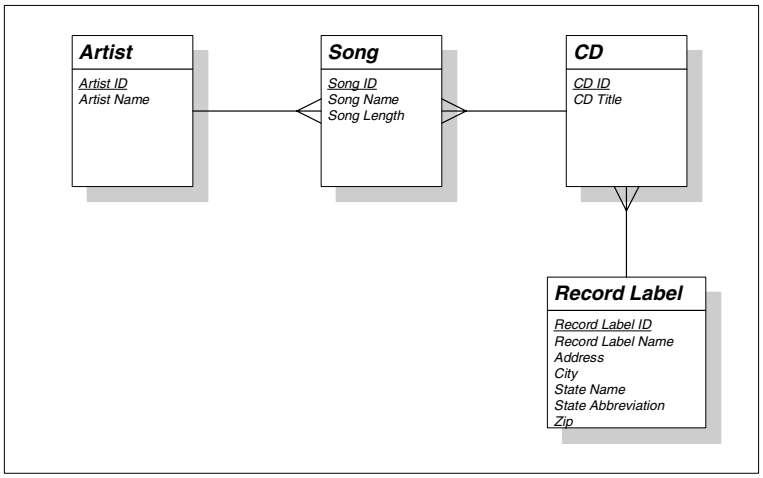

Figure 2-11: Record Label address information in our CD database

The values of `State Name` and `State Abbreviation` would conform to 1NF because they have only one value per record in the `Record Label` entity. The problem here is that `State Name` and `State Abbreviation` are dependent on each other.

In other words, if we change the `State Abbreviation` for a particular `Record Label`—from MN to CA—we also have to change the `State Name`—from Minnesota to California. We would normalize this by creating a `State` entity with `State Name` and `State Abbreviation` attributes. Figure 2-12 shows how to relate this new entity to the `Record Label` entity.

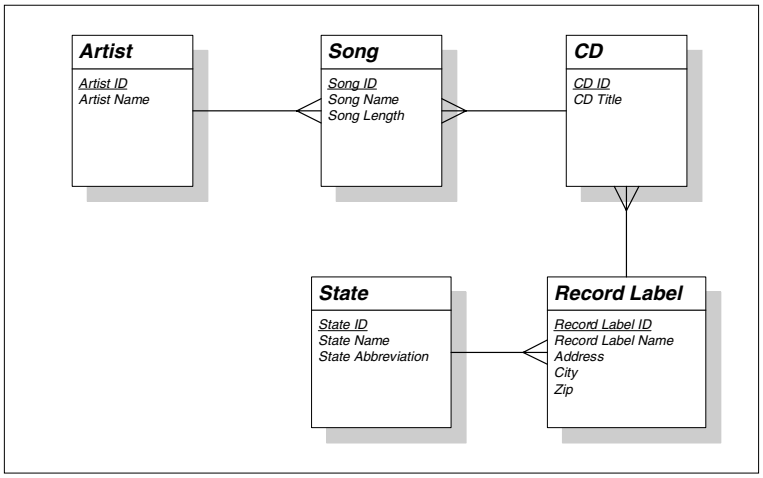

Figure 2-12: Our data model in the third normal form

Now that we are in 3NF, we can say that our data model is normalized. There are other normal forms which have some value from a database design standpoint, but these are beyond the scope of this book. For most design purposes, the third normal form is sufficient to guarantee a proper design.

A Logical Data Modeling Methodology

We now have a completed logical data model. Let's review the process we went through to get here.

1. Identify and model the entities.
2. Identify and model the relationships between the entities.
3. Identify and model the attributes.
4. Identify unique identifiers for each entity.
5. Normalize.

In practice, the process is rarely so linear. As shown in the example, it is often tempting and appropriate to jump around between entities, relationships, attributes, and unique identifiers. It is not as important that you follow a strict process as it is that you discover and capture all of the information necessary to correctly model the system.

The data model we created in this chapter is quite simple. We covered an approach to creating such a model which is in-line with the type and complexity of databases you are likely to encounter in developing MySQL or mSQL databases. We did not cover a whole

host of design techniques and concepts that are not so important to small-scale database design, but these can be found in any text dedicated to database design.

Physical Database Design

What was the point in creating the logical data model? You want to create a database to store data about CDs. The data model is only an intermediate step along the way. Ultimately, you would like to end up with a MySQL database where you can store data. How do you get there? Physical database design translates your logical data model into a set of SQL statements that define your MySQL database.

Since MySQL is a relational database systems, it is relatively easy to translate from a logical data model, such as the one we described earlier, into a physical MySQL database. Here are the rules for translation:

1. Entities become tables in the physical database.
2. Attributes become columns in the physical database. Choose an appropriate datatype for each of the columns.
3. Unique identifiers become columns that are not allowed to have NULLs. These are called *primary keys* in the physical database. You may also choose to create a unique index on the identifiers to enforce uniqueness.
4. Relationships are modeled as *foreign keys*.

Tables and columns

If we apply the first three rules to our data model—minus the Record Label address information—we will end up with the physical database described in Table 2-2. This does not include the foreign keys yet.

Table 2-2: Physical Table Definitions for the CD Database

Table	Column	Datatype	Notes
CD	CD_ID	INT	primary key
	CD_TITLE	VARCHAR(50)	
ARTIST	ARTIST_ID	INT	primary key
	ARTIST_NAME	VARCHAR(50)	
SONG	SONG_ID	INT	primary key
	SONG_NAME	VARCHAR(50)	
	SONG_LENGTH	TIME	
RECORD_LABEL	RECORD_LABEL_ID	INT	primary key
	RECORD_LABEL_NAME	VARCHAR(50)	

The first thing you may notice is that all of the spaces are gone from the entity names in our physical schema. This is because these names need to translate into SQL calls to create these tables. Table names should thus conform to SQL naming rules. Another thing to notice is that we made all primary keys of type `INT`. Because these attributes are complete inventions on our part, they can be of any index-able datatype.1 The fact that they are of type `INT` here is almost purely arbitrary. It is *almost* arbitrary because it is actually faster to search on numeric fields in many database engines and hence numeric fields make good primary keys. However, we could have chosen `CHAR` as the type for the primary key fields and everything would work just fine. The bottom line is that this choice should be driven by your criteria for choosing identifiers.

CD_TITLE, ARTIST_NAME, SONG_NAME and RECORD_LABEL_NAME are set to be of type `VARCHAR` with a length of 50. The length has been chosen arbitrarily for the sake of this example. In reality, you should do some analysis of sample data to determine the length of text fields. If you set them too short, you may end up with a database that is not able to capture all the data you need to store.

SONG_LENGTH is set to type `TIME` which can store elapsed time.

Foreign Keys

We now have a starting point for a physical schema. We haven't yet translated the relationships into the physical data model. As we discussed earlier, once you have refined your data model, you should have all 1-to-1 and 1-to-M relationships—the M-to-M relationships were resolved via junction tables. We model relationships by adding a foreign key to one of the tables involved in the relationship. A foreign key is the unique identifier or primary key of the table on the other side of the relationship.

The most common relationship is the 1-to-M relationship. This relationship is mapped by placing the primary key from the "one" side of the relationship into the table on the "many" side. In our example, this rule means that we need to do the following:

- Place a `RECORD_LABEL_ID` column in the `CD` table.
- Place a `CD_ID` column in the `SONG` table.
- Place an `ARTIST_ID` column in the `SONG` table.

Table 2-3 shows the new schema.

Table 2-3: The Physical Data Model for the CD Database (continued)

Table	Column	Datatype	Notes

1 Later in this book, we will cover the datatypes supported by MySQL and mSQL. Each database engine has different rules about which datatypes can be indexible. Neither database, for example, allows indices to be created on whole `TEXT` fields. It would therefore be inappropriate to have a primary key column be of type `TEXT`.

CD	CD_ID	INT	primary key
	CD_TITLE	VARCHAR(50)	
	RECORD_LABEL_ID	INT	foreign key
ARTIST	ARTIST_ID	INT	primary key
	ARTIST_ID	VARCHAR(50)	
SONG	SONG_ID	INT	primary key
	SONG_NAME	VARCHAR(50)	
	SONG_LENGTH	TIME	
	CD_ID	INT	foreign key
	ARTIST_ID	INT	foreign key
RECORD_LABEL	RECORD_LABEL_ID	INT	primary key
	RECORD_LABEL_NAME	VARCHAR(50)	

We do not have any 1-to-1 relationships in this data model. If we did have such a relationship, it should be mapped by picking one of the tables and giving it a foreign key column that matches the primary key from the other table. In theory, it does not matter which table you choose, but practical considerations may dictate which column makes the most sense as a foreign key. Another way to handle 1-to-1 relationships is to simply combine both entities into a single table. In that case, you have to pick a primary key from one of the tables to be the primary key of the combined table.

We now have a complete physical database schema ready to go. The last remaining task is to translate that schema into SQL. For each table in the schema, you write one `CREATE TABLE` statement. Typically, you will choose to create unique indices on the primary keys to enforce uniqueness.

Example 2-1 is an example SQL script for creating the example database in MySQL.

Example 2-1: An Example Script for Creating the CD Database in MySQL

```
CREATE TABLE CD (CD_ID INT NOT NULL,
                 RECORD_LABEL_ID INT,
                 CD_TITLE TEXT,
                 PRIMARY KEY (CD_ID))

CREATE TABLE ARTIST (ARTIST_ID INT NOT NULL,
                     ARTIST_NAME  TEXT,
                     PRIMARY KEY (ARTIST_ID))

CREATE TABLE SONG (SONG_ID INT NOT NULL,
                   SONG_NAME TEXT,
                   SONG_LENGTH TEXT,
                   CD_ID INT,
                   ARTIST_ID INT,
                   PRIMARY KEY (SONG_ID))
```



```
CREATE TABLE RECORD_LABEL (RECORD_LABEL_ID INT NOT NULL,
                           RECORD_LABEL_NAME TEXT,
                           PRIMARY KEY(RECORD_LABEL_ID))
```

Note that the "FOREIGN KEY" reference is not used in the script. This is because MySQL does not support FOREIGN KEY constraints. MySQL will allow you to embed them in your CREATE TABLE statements but they will be ignored for the purposes of enforcing them.

Data models are meant to be database independent. You can therefore take the techniques and the data model we have generated in this chapter and apply them not only to MySQL, but to Oracle, Sybase or any other relational database engine. In the following chapters, we will discuss the details of how you can use your new database design knowledge to build applications.

