# Chapter 2. Modeling and Designing Databases

When implementing a new database, it's easy to fall into the trap of quickly getting something up and running without dedicating adequate time and effort to the design. This carelessness frequently leads to costly redesigns and reimplementations down the road. Designing a database is like drafting the blueprints for a house; it's silly to start building without detailed plans. Notably, good design allows you to extend the original building without pulling everything down and starting from scratch. And as you will see, bad designs are directly related to poor database performance.

## How Not to Develop a Database

Database design is probably not the most exciting task in the world, but indeed it is becoming one of the most important ones. 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 and 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  | Data Science          |   72 |
| Sarah      | Doe     | Programming 1         |   87 |
| John Paul  | Bloggs  | Computing Mathematics |   43 |
| John Paul  | Bloggs  | Computing Mathematics |   65 |
| Sarah      | Doe     | Data Science          |   65 |
| Susan      | Smith   | Computing Mathematics |   75 |
```

```
| Susan      | Smith   | Programming 1         |   55 |
| Susan      | Smith   | Computing Mathematics |   80 |
+------------+---------+-----------------------+------+
```

The list is nice and compact, we can easily access grades for any student or any course, and it looks similar to a spreadsheet. However, we could have more than one student with the same name. For instance, there are two entries for Susan Smith and the Computing Mathematics course in the sample data. Which Susan Smith got 75% and which got 80%? A common way to differentiate duplicate data entries is to assign a unique number to each entry. Here, we can assign a unique `StudentID` number to each student:

```
+------------+------------+---------+-----------------
| StudentID  | GivenNames | Surname | CourseName
+------------+------------+---------+-----------------
| 12345678   | John Paul  | Bloggs  | Data Science
| 12345121   | Sarah      | Doe     | Programming 1
| 12345678   | John Paul  | Bloggs  | Computing Mathema
| 12345678   | John Paul  | Bloggs  | Computing Mathema
| 12345121   | Sarah      | Doe     | Data Science
| 12345876   | Susan      | Smith   | Computing Mathema
| 12345876   | Susan      | Smith   | Programming 1
| 12345303   | Susan      | Smith   | Computing Mathema
+------------+------------+---------+-----------------
```

Now we know which Susan Smith got 80%: the one with the student ID number 12345303.

There's another problem. In our table, John Paul Bloggs has two scores for the Computing Mathematics course: he failed it once with 43%, and then passed it with 65% on his second attempt. In a relational database, the rows form a set, and there is no implicit ordering between them. Looking at this table we might guess that the pass happened after the failure, but we can't 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 ( `Year` ) and semester ( `Sem` ):

```
+------------+------------+---------+-----------------
| StudentID  | GivenNames | Surname | CourseName
+------------+------------+---------+-----------------
| 12345678   | John Paul  | Bloggs  | Data Science
```

| 12345121 | Sarah | Doe | Programming 1
| 12345678 | John Paul | Bloggs | Computing Mathema
| 12345678 | John Paul | Bloggs | Computing Mathema
| 12345121 | Sarah | Doe | Data Science
| 12345876 | Susan | Smith | Computing Mathema
| 12345876 | Susan | Smith | Programming 1
| 12345303 | Susan | Smith | Computing Mathema

Notice that the `Student_Grades` table has become a bit bloated. We've repeated the student ID, given names, and surname for every year. 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 we could keep less information in the `Student_Grades` table:

| StudentID | CourseName | Year | Sem | Pct |
|---|---|---|---|---|
| 12345678 | Data Science | 2019 | 2 | 7 |
| 12345121 | Programming 1 | 2020 | 1 | 8 |
| 12345678 | Computing Mathematics | 2019 | 2 | 4 |
| 12345678 | Computing Mathematics | 2020 | 1 | 6 |
| 12345121 | Data Science | 2020 | 1 | 6 |
| 12345876 | Computing Mathematics | 2019 | 1 | 7 |
| 12345876 | Programming 1 | 2019 | 2 | 5 |
| 12345303 | Computing Mathematics | 2020 | 1 | 8 |

To look up a student's grades, we would need to first look up their 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, though. 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 addresses? How should we store addresses so that things don't break when students change their addresses?

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. We can save a lot of reworking by carefully documenting the requirements up front and then working through them to develop a coherent design.

# The Database Design Process

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

*Requirements analysis*

First, we determine and write down what we need from the database, what data we will store, and how the data items relate to each other. In practice, this might involve a 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 the database requirements, we distill them into a formal description of the database design. Later 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 existing database management system and database tables.

At the end of the chapter, we'll look at how we can use the open source MySQL Workbench tool to 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, whereas 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, and a sale is a relationship between a customer and a product. It is common to get confused between entities and relationships when you're starting out, and you may end up designing relationships as entities and vice versa. The best way to improve your database design skills is by practicing a lot.

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 in the database. We'll start by looking at how the ER modeling process works and then observe it in "Entity Relationship Modeling Examples" for three sample databases.

## Representing Entities

To help visualize the design, the ER modeling approach involves drawing an ER diagram. In the ER diagram, we represent an entity set by a rectangle containing the entity name. For our sales database example, our ER diagram would show the product and customer entity sets, as shown in Figure 2-1.


Figure 2-1. An entity set is represented by a named rectangle

We typically use the database to store specific characteristics, or *attributes*, of the entities. We could record the name, email address, postal address, and telephone number of each customer in a sales database. In a more elaborate customer relationship management (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.

We may form an attribute from smaller parts; for example, we compose a postal address from 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 can 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 differentiate between customers, but this could be an inadequate solution because several customers could have identical names. To tell them apart, we need an attribute (or a minimal combination of attributes) guaranteed to be unique to each customer. The identifying attribute or attributes form a unique key, and in this particular case, we call it a *primary key*.

In our example, we can assume that no two customers have the same email address, so the email address can be the primary key. However, when designing a database, we need to think carefully about the implications of our choices. For example, if we decide to identify customers by their email addresses, how will we handle a customer having multiple email addresses? Any applications we build to use this database might treat each email address as a separate person. It could be hard to adapt everything to allow people to have more than one. Using the email address as the key also means that every customer must have an email address; otherwise, we can't 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 could 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 will not 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 alternatives, or *candidate* keys, to be our main or *primary* key. We usually choose based on how confident we are that the attribute will be nonempty and unique for each entity and how small the key is (shorter keys are faster to maintain and to use to perform lookup operations).

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

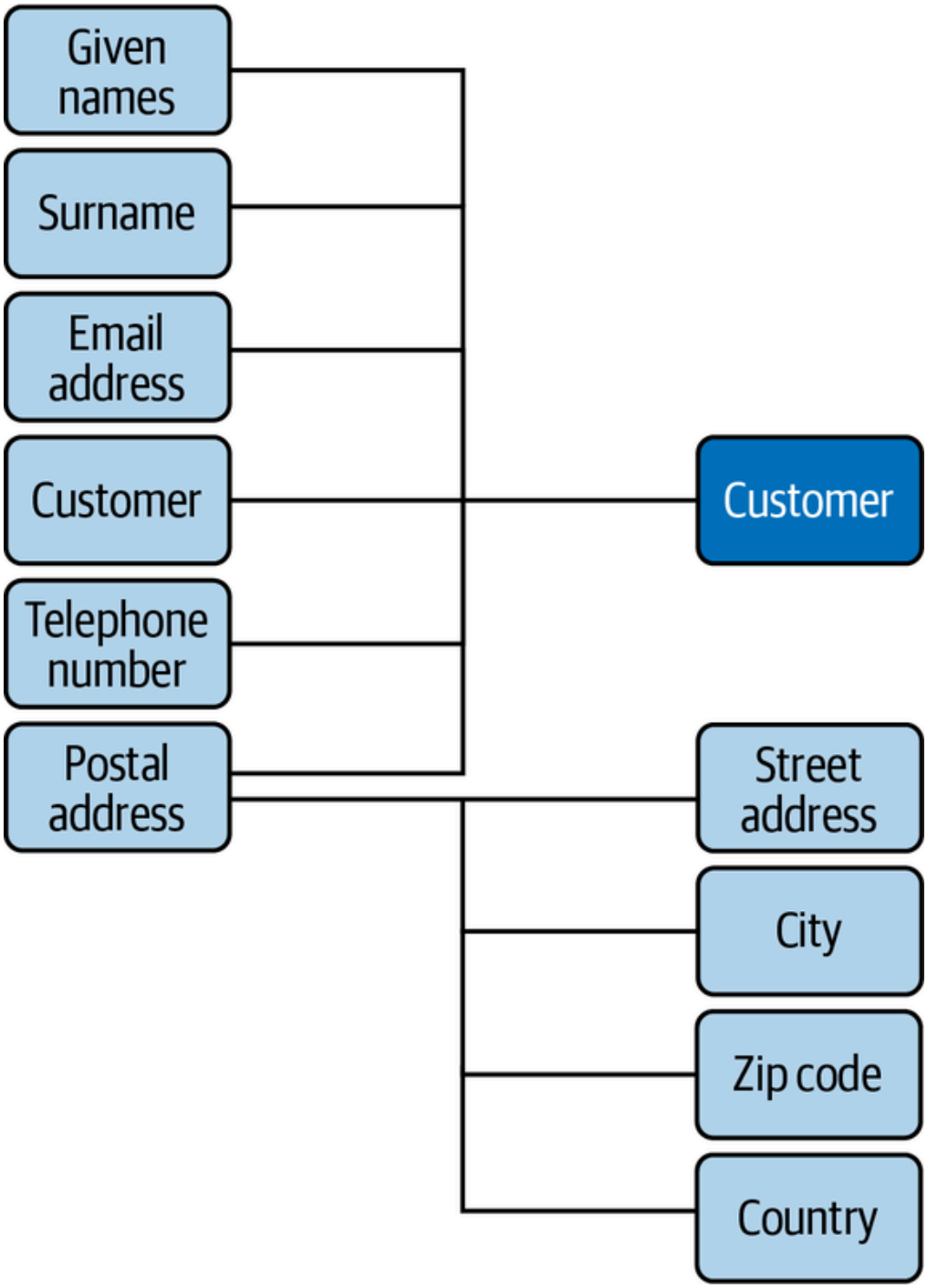


Figure 2-2. The ER diagram representation of the customer entity

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.

Attributes can be empty; for example, some customers may not provide their telephone numbers. However, the primary key of an entity (including the components of a multiattribute primary key) must never be unknown (technically, it must be `NOT NULL`). So, 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 will 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 application, 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 2-3.

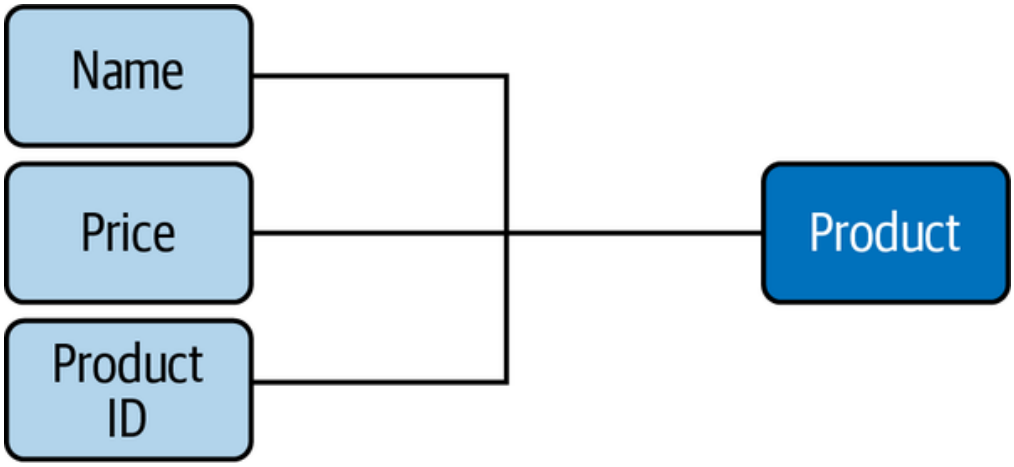


Figure 2-3. The ER diagram representation of the product entity

## 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 employee can have an address, 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, Marcos Albe bought one "Raspberry Pi 4," one "500 GB SSD M.2 NVMe," 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 use the shorthand terms *1:1*, *1:N*, and *M:N* for one-to-one, one-to-many, and many-to-many relationships.

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 sometimes 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 costly and 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 2-4 shows the relationship between the customer and product entities, along with the number and timestamp attributes of the sale relationship.

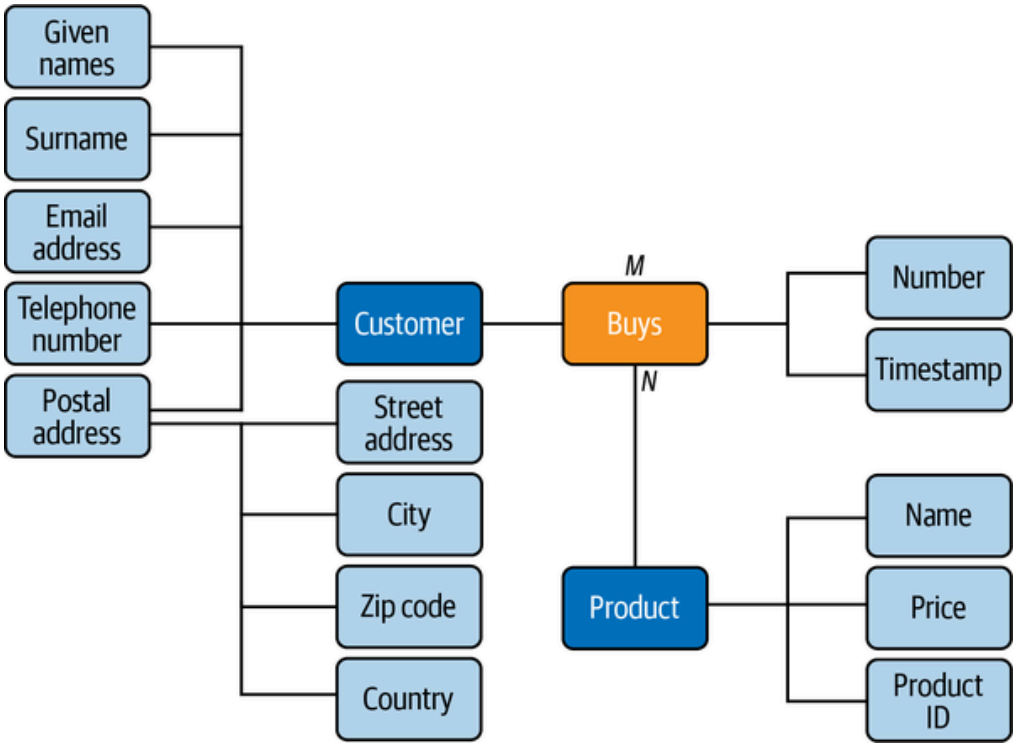


Figure 2-4. The ER diagram representation of the customer and product entities, and the sale relationship between them

## 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 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, 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, as shown in Figure 2-5. But 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 2-6.

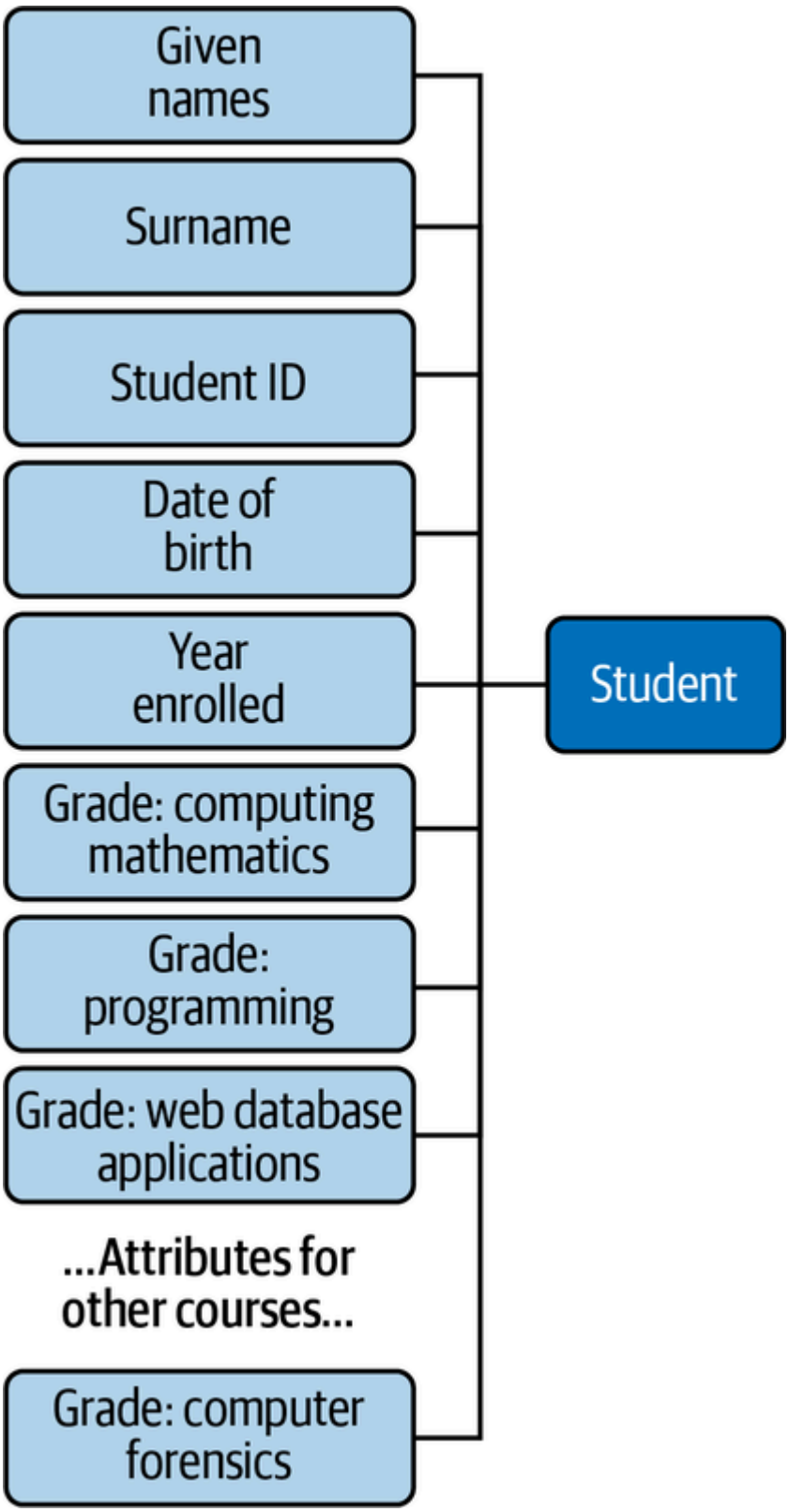


Figure 2-5. The ER diagram representation of student grades as attributes of the student entity

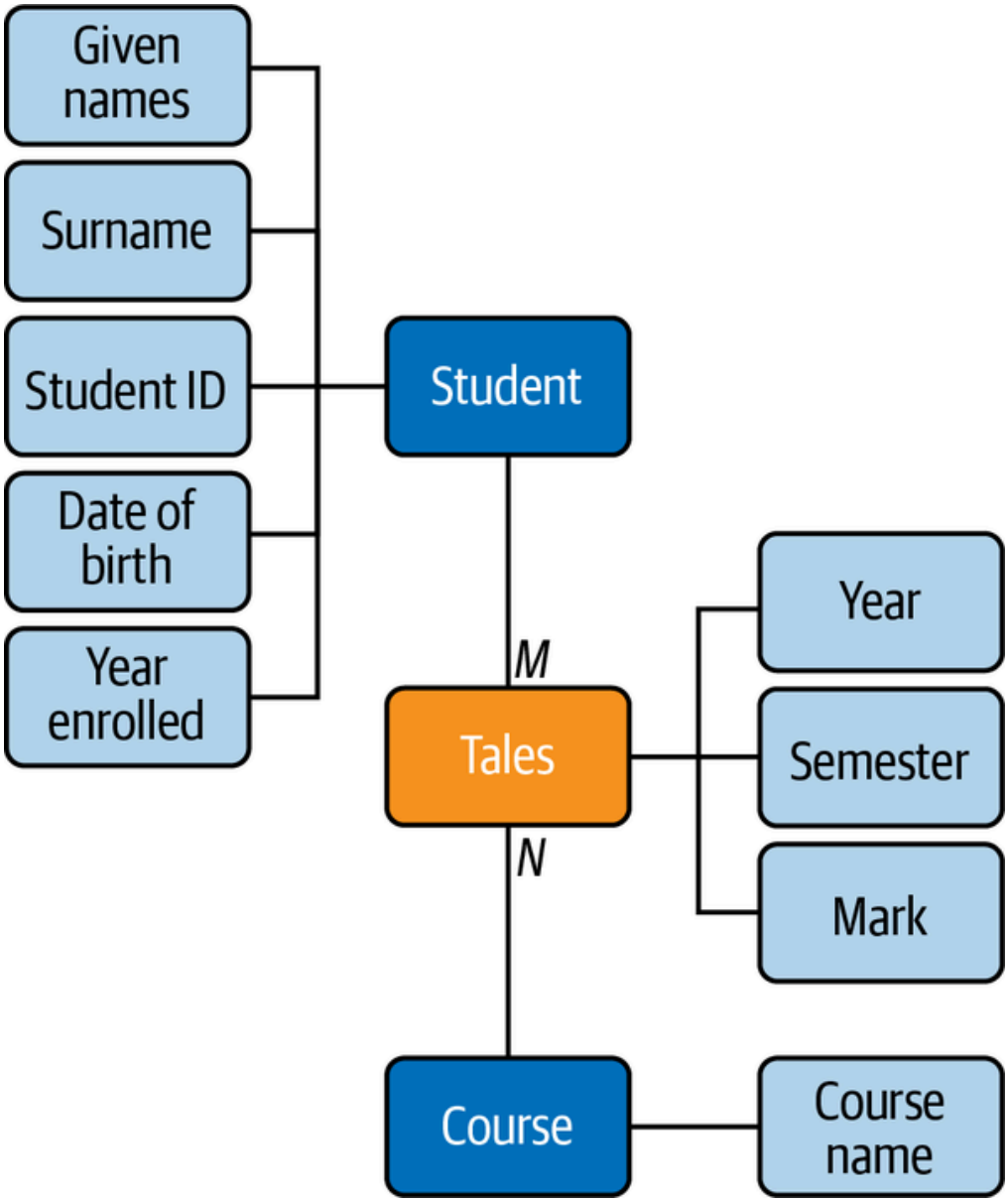


Figure 2-6. The ER diagram representation of student grades as a separate entity

## 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 map verbs to relationships. 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 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. That is, 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.

## Intermediate Entities

It is often possible to conceptually simplify a many-to-many relationship by replacing it 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 this 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 2-7.


Figure 2-7. A passenger participates in an M:N relationship with a flight

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 passenger entities. The requirement could be better worded as: “A passenger can make a booking for a seat on a flight.” The updated ER diagram fragment is shown in Figure 2-8.

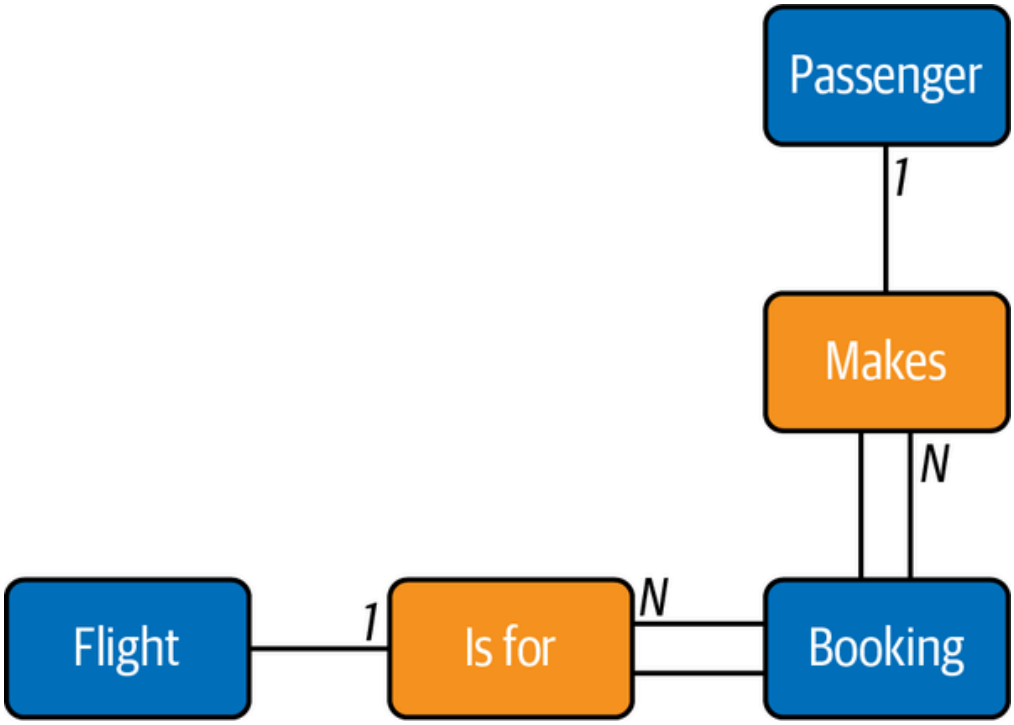


Figure 2-8. The intermediate booking entity between the passenger and flight entities

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 (as described in "Partial and Total Participation" on page 77). This total participation could not be captured effectively in the representation in Figure 2-7.

## 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 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 2-9. A weak entity is uniquely identified in the context of its owning (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 2-10 shows a summary of the symbols we've explained for ER diagrams.

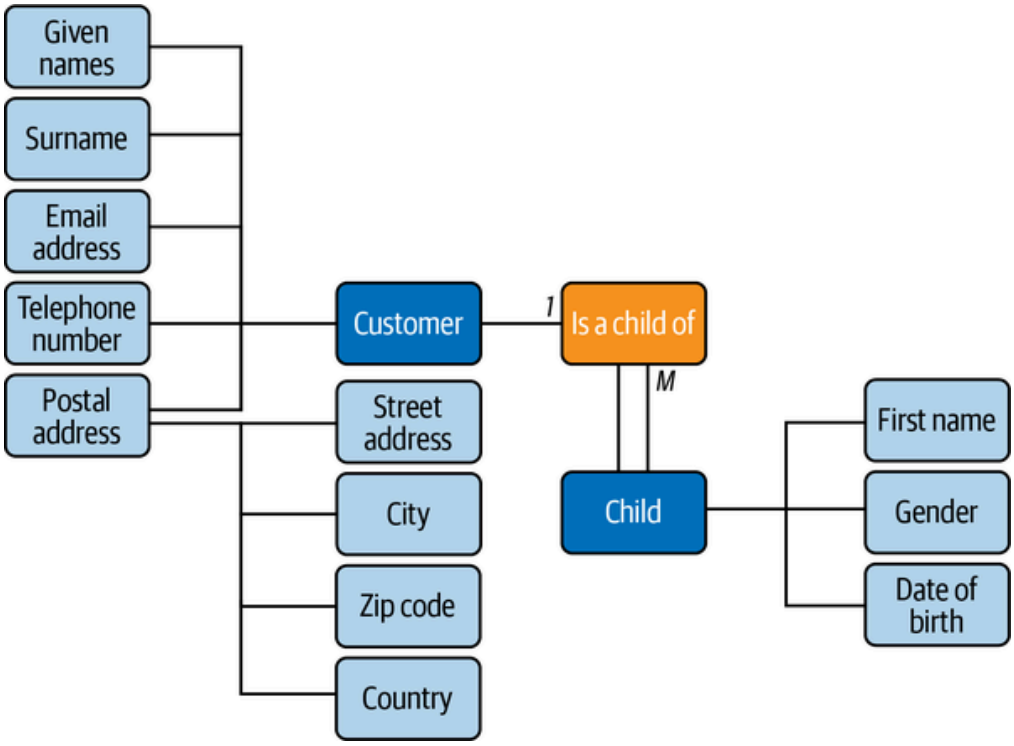


Figure 2-9. The ER diagram representation of a weak entity

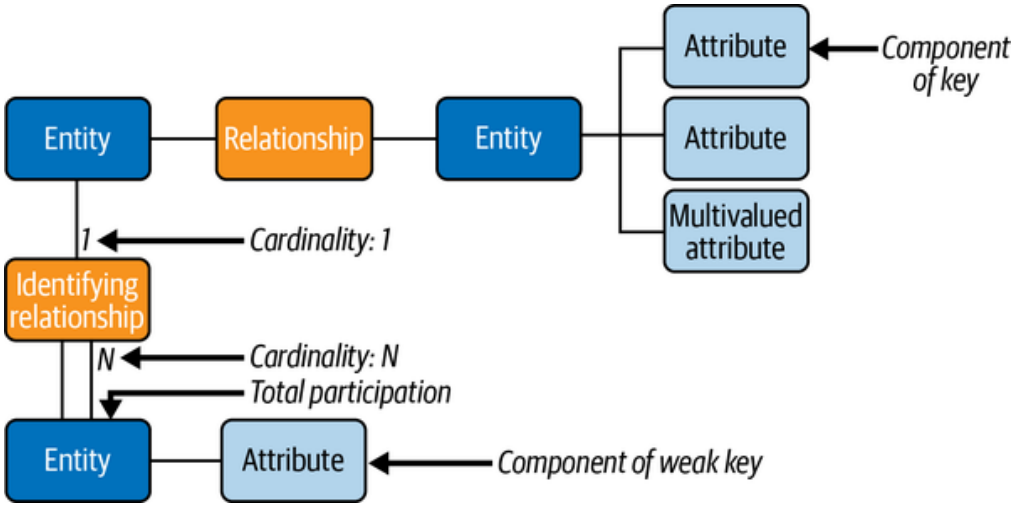


Figure 2-10. A summary of the ER diagram symbols

# Database Normalization

Database normalization is an important concept when designing the relational data structure. Dr. Edgar F. Codd, the inventor of the relational database model, proposed the normal forms in the early ’70s, and these are still widely used by the industry nowadays. Even with the advent of the NoSQL databases, there is no evidence in the short or medium term that relational databases will disappear or that the normal forms will fall into disuse.

The main objective of the normal forms is to reduce data redundancy and improve data integrity. Normalization also facilitates the process of redesigning and extending the database structure.

Officially, there are six normal forms, but most database architects deal only with the first three forms. That is because the normalization process is progressive, and we cannot achieve a higher level of database normalization unless the previous levels have been satisfied. Using all six norms constricts the database model too much, however, and in general, they become very complex to implement.

In real workloads, usually there are performance issues. This is one reason for extract, transform, load (*ETL*) jobs to exist: they denormalize the data to process it.

Let’s take a look at the first three normal forms:

*The first normal form (1NF) has the following goals*

- Eliminate repeating groups in individual tables.
- Create a separate table for each set of related data.
- Identify each set of related data with a primary key.

> If a relation contains composite or multivalued attributes, it violates the first normal form. Conversely, a relation is in first normal form if it does not contain any composite or multivalued attributes. So, a relation is in first normal form if every attribute in that relation has a single value of the appropriate type.

*The goals of second normal form (2NF) are*

- Create separate tables for sets of values that apply to multiple records.
- Relate these tables with a foreign key.

Records should not depend on anything other than a table's primary key (a compound key, if necessary).

*Third normal form (3NF) adds one more goal*

- Eliminate fields that do not depend on the key.

Values in a record that are not part of that record's key do not belong in the table. In general, any time the contents of a group of fields may apply to more than a single record in the table, you should consider placing those fields in a separate table.

Table 2-1 lists the normal forms, from the least normalized to the most normalized. The unnormalized form (UNF) is a database model that does not meet any of the database normalization conditions. Other normalization forms exist, but they are beyond the scope of this discussion.

Table 2-1. The normal forms (from least to most normalized)

| | **UNF (1970)** | **1NF (1970)** | **2NF (1971)** | **3NF (1971)** |
|---|---|---|---|---|
| Primary key (no duplicate tuples) | Maybe | Yes | Yes | Yes |
| No repeating groups | No | Yes | Yes | Yes |
| Atomic columns (cells have single value) | No | Yes | Yes | Yes |
| Every nontrivial functional dependency either does not begin with a proper subset of a candidate key or ends with a prime attribute (no partial functional dependencies of nonprime attributes on candidate keys) | No | No | Yes | Yes |
| Every nontrivial functional | No | No | No | Yes |

| | UNF (1970) | 1NF (1970) | 2NF (1971) | 3NF (1971) |
|---|---|---|---|---|
| dependency begins with a superkey or ends with a prime attribute (no transitive functional dependencies of nonprime attributes on candidate keys) | | | | |
| Every nontrivial functional dependency either begins with a superkey or ends with an elementary prime attribute | No | No | No | No |
| Every nontrivial functional dependency begins with a superkey | No | No | No | No |
| Every nontrivial multivalued | No | No | No | No |

| | UNF (1970) | 1NF (1970) | 2NF (1971) | 3NF (1971) |
|---|---|---|---|---|
| dependency begins with a superkey | | | | |
| Every join dependency has a superkey component | No | No | No | No |
| Every join dependency has only superkey components | No | No | No | No |
| Every constraint is a consequence of domain constraints and key constraints | No | No | No | No |
| Every join dependency is trivial | No | No | No | No |

# Normalizing an Example Table

To make these concepts clearer let’s walk through an example of normalizing a fictional student table.

We'll start with the unnormalized table:

| Student# | Advisor | Adv-Room | Class1 | Class2 | Class |
|---|---|---|---|---|---|
| 1022 | Jones | 412 | 101-07 | 143-01 | 159-0 |
| 4123 | Smith | 216 | 201-01 | 211-02 | 214-0 |

## First Normal Form: No Repeating Groups

Tables should have only a single field for each attribute. Since one student has several classes, these classes should be listed in a separate table. The fields `Class1`, `Class2`, and `Class3` in our unnormalized table are indications of design trouble.

Spreadsheets often have multiple fields for the same attribute (e.g., `address1`, `address2`, `address3`), but tables should not. Here's another way to look at this problem: with a one-to-many relationship, don't put the one side and the many side in the same table. Instead, create another table in first normal form by eliminating the repeating group—for example, with `Class#`, as shown here:

| Student# | Advisor | Adv-Room | Class# |
|---|---|---|---|
| 1022 | Jones | 412 | 101-07 |
| 1022 | Jones | 412 | 143-01 |
| 1022 | Jones | 412 | 159-02 |
| 4123 | Smith | 216 | 201-01 |
| 4123 | Smith | 216 | 211-02 |
| 4123 | Smith | 216 | 214-01 |

## Second Normal Form: Eliminate Redundant Data

Note the multiple `Class#` values for each `Student#` value in the previous table. `Class#` is not functionally dependent on `Student#` (the primary key), so this relationship is not in second normal form.

The following two tables demonstrate the conversion to second normal form. We now have a `Students` table:

| Student# | Advisor | Adv-Room |
|---|---|---|
| 1022 | Jones | 412 |

```
4123        Smith     216
```

and a `Registration` table:

```
Student#  Class#
1022      101-07
1022      143-01
1022      159-02
4123      201-01
4123      211-02
4123      214-01
```

### Third Normal Form: Eliminate Data Not Dependent on Key

In the previous example, `Adv-Room` (the advisor's office number) is functionally dependent on the `Advisor` attribute. The solution is to move that attribute from the `Students` table to a `Faculty` table, as shown next.

The `Students` table now looks like this:

```
Student#  Advisor
1022      Jones
4123      Smith
```

And here's the `Faculty` table:

```
Name  Room  Dept
Jones 412 42
Smith 216 42
```

## Entity Relationship Modeling Examples

In the previous sections, we walked though hypothetical examples to help you understand the basics of database design, ER diagrams, and normalization. Now we're going to look at some ER examples from sample databases available for MySQL. To visualize the ER diagrams, we are going to use *MySQL Workbench*.

MySQL Workbench uses a physical ER representation. Physical ER diagram models are more granular, showing the processes necessary to add information to a database. Rather than using symbols, we use tables in the ER diagram, making it closer to the real database. MySQL Workbench goes one step further and uses *enhanced entity-relationship (EER) diagrams*. EER diagrams are an expanded version of ER diagrams.

We won't go into all the details, but the main advantage of an EER diagram is that it provides all the elements of an ER diagram while adding support for:

- Attribute and relationship inheritance
- Category or union types
- Specialization and generalization
- Subclasses and superclasses

Let's start with the process to download the sample databases and visualize their EER diagrams in MySQL Workbench.

The first one we'll use is the `sakila` database. Development of this database began in 2005. Early designs were based on the database used in the Dell whitepaper "Three Approaches to MySQL Applications on Dell PowerEdge Servers", which was designed to represent an online DVD store. Similarly, the `sakila` sample database is designed to represent a DVD rental store, and it borrows film and actor names from the Dell sample database. You can use the following commands to import the `sakila` database to your MySQL instance:

```
# wget https://downloads.mysql.com/docs/sakila-db.tar.g
# tar -xvf sakila-db.tar.gz
# mysql -uroot -pmsandbox < sakila-db/sakila-schema.sql
# mysql -uroot -pmsandbox < sakila-db/sakila-data.sql
```

`sakila` also provides the EER model, in the *sakila.mwb* file. You can open the file with MySQL Workbench, as shown in Figure 2-11.

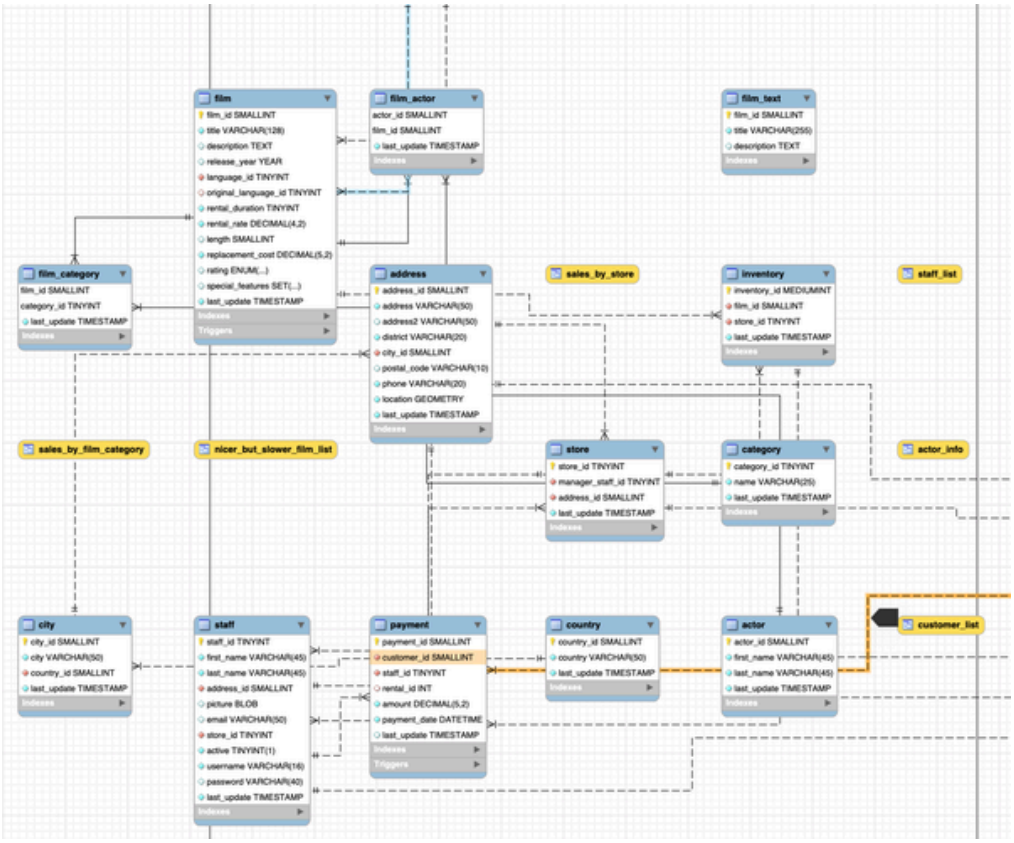


Figure 2-11. The `sakila` database EER model; note the physical representation of the entities instead of using symbols

Next is the `world` database, which uses sample data from Statistics Finland.

The following commands will import the `world` database to your MySQL instance:

```
# wget https://downloads.mysql.com/docs/world-db.tar.gz
# tar -xvf world-db.tar.gz
# mysql -uroot -plearning_mysql < world-db/world.sql
```

The `world` database does not come with an EER file as `sakila` does, but you can create the EER model from the database using MySQL Workbench. To do this, select Reverse Engineer from the Database menu, as in Figure 2-12.

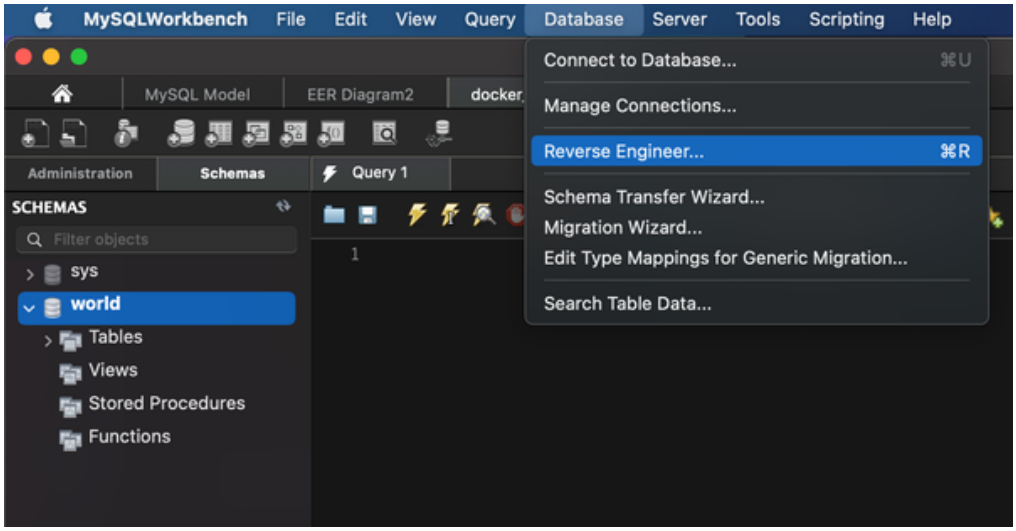


Figure 2-12. Reverse engineering from the `world` database

Workbench will connect to the database (if not connected already) and prompt you to choose the schema you want to reverse engineer, as shown in Figure 2-13.

Click Continue, and then click Execute on the next screen, shown in Figure 2-14.

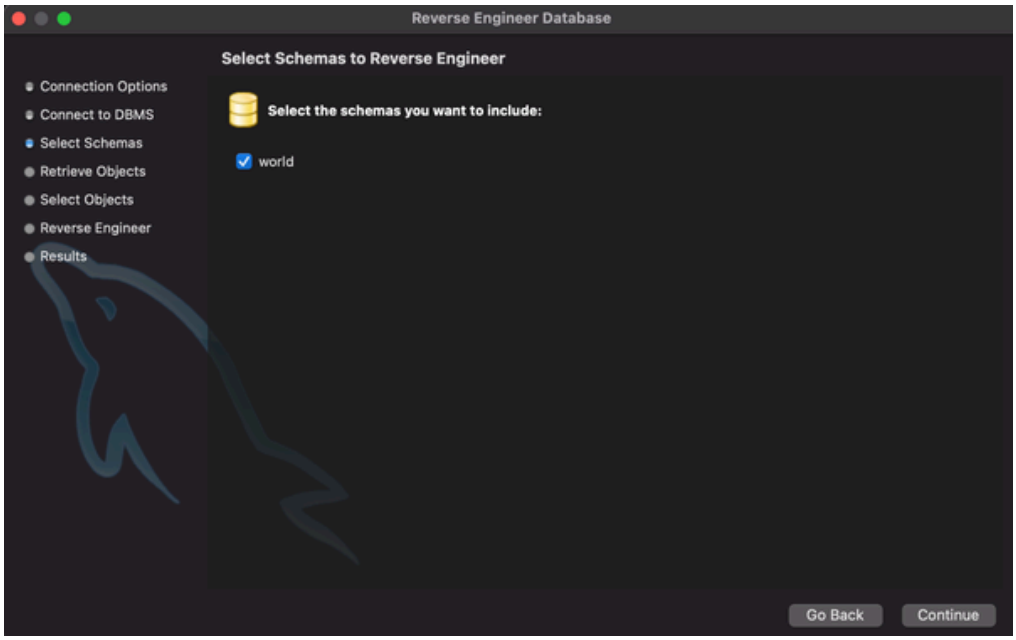


Figure 2-13. Choosing the schema

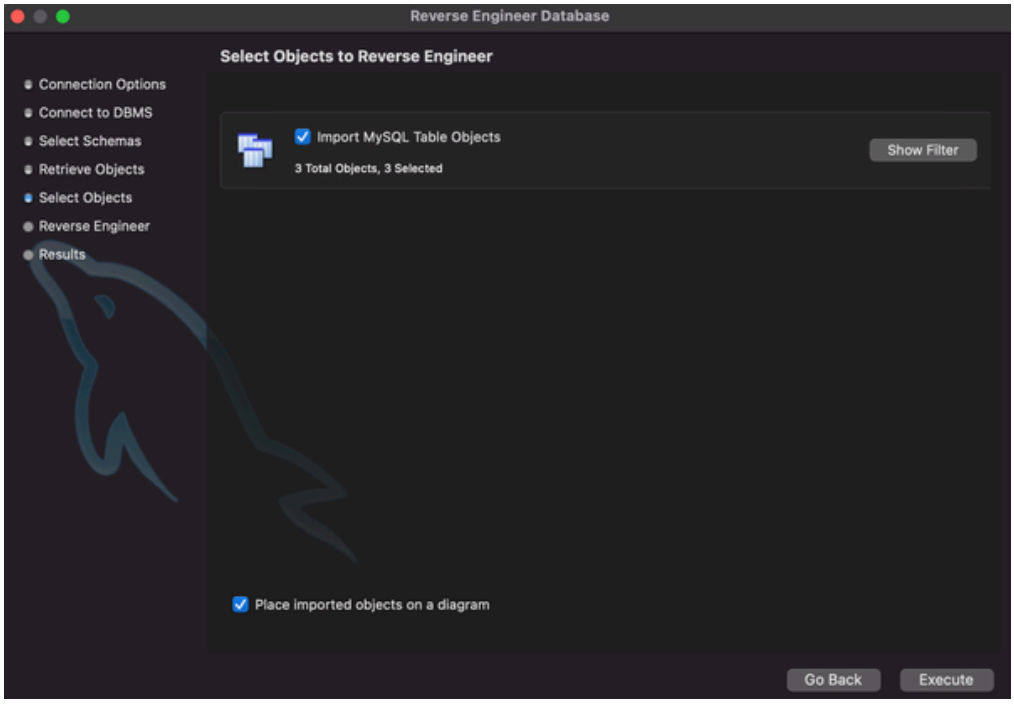


Figure 2-14. Click *Execute* to start the reverse-engineering process

This produces the ER model for the `world` database, shown in Figure 2-15.

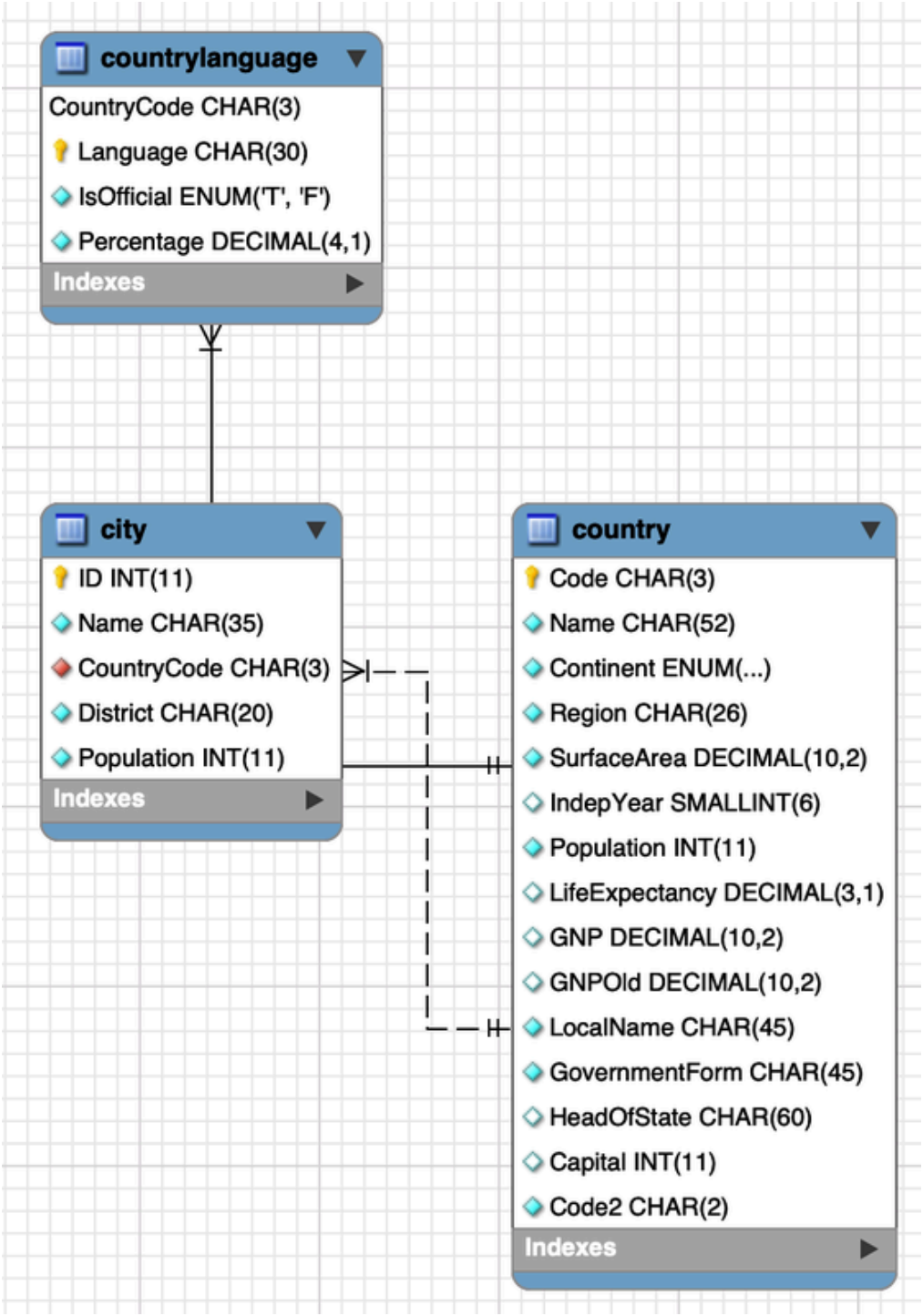


Figure 2-15. The ER model for the `world` database

The last database you'll import is the `employees` database. Fusheng Wang and Carlo Zaniolo created the original data at Siemens Corporate Research. Giuseppe Maxia made the relational schema, and Patrick Crews exported the data in relational format.

To import the database, first you need to clone the Git repository:

```
# git clone https://github.com/datacharmer/test_db.git
# cd test_db
# cat employees.sql | mysql -uroot -psekret
```

Then you can use the reverse engineering procedure in MySQL Workbench again to create the ER model for the `employees` database, as shown in Figure 2-16.

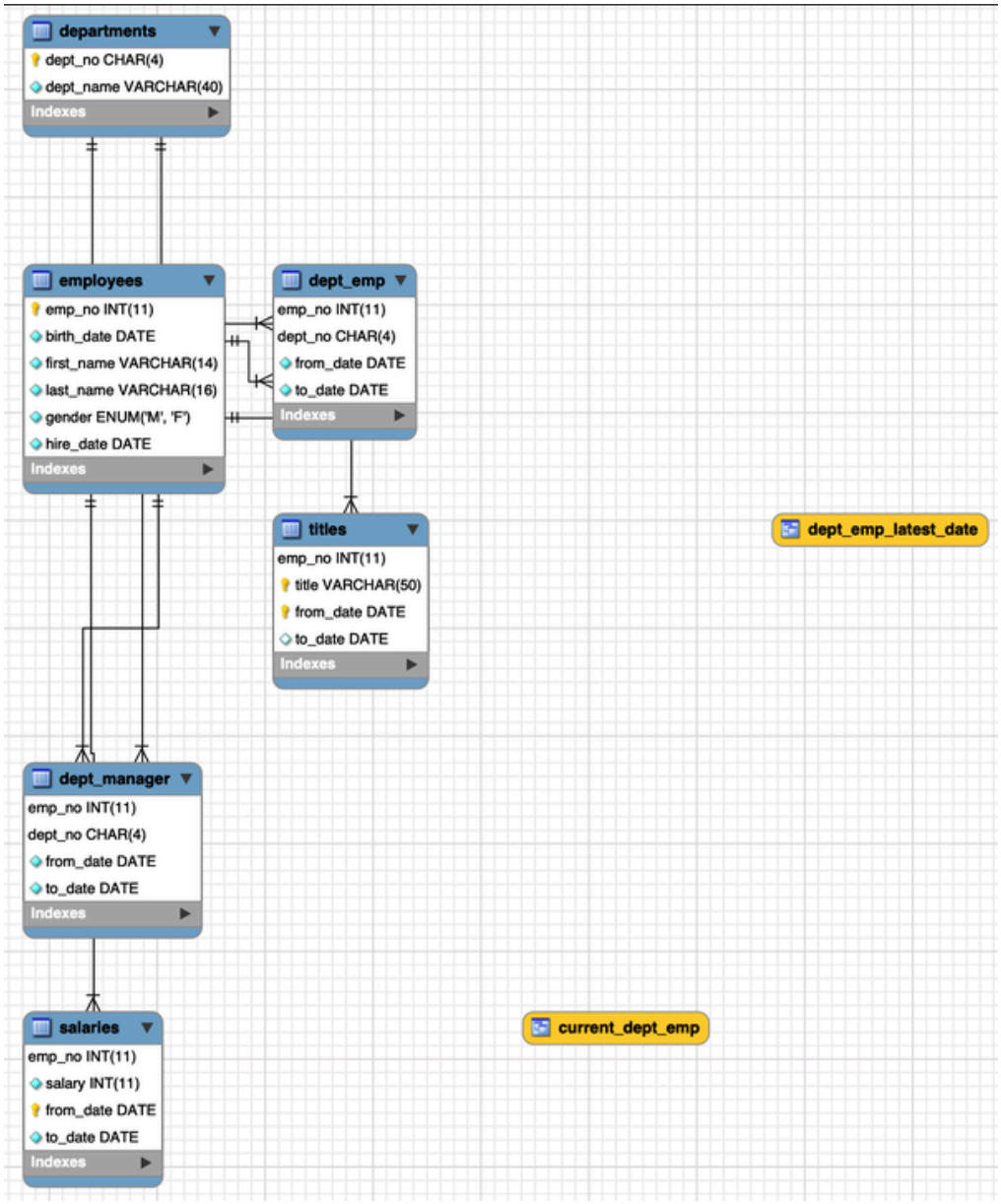


Figure 2-16. The ER model for the `employees` database

It is important that you carefully review the ER models shown here so you understand the relationships between entities and their attributes. Once the concepts are solidified, start practicing. You will see how to do that in the next section. We'll show you how to create a database on your MySQL server in Chapter 4.

# Using the Entity Relationship Model

This section looks at the steps required to create an ER model and deploy it into database tables. We saw previously that MySQL Workbench lets us

reverse engineer an existing database. But how do we model a new database and deploy it? 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 rules discussed in the following sections 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 owning entity's primary key is a foreign key here because it's a key not of this table but another table. The table's primary key 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 entity's multivalued attribute, create a table comprising the entity's primary key and the attribute.

### Map the relationships to database tables

Each one-to-one relationship between two entities includes 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 entity's primary key 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 each entity's primary key 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 relationship attributes.

## Creating a Bank Database ER Model

We've discussed database models for student grades and customer information, plus the three open source EERs available for MySQL. Now let's see how we could model a bank database. We've collected all the requisites from the stakeholders and defined our requirements for the online banking system, and we've decided we need to have the following entities:

- Employees
- Branches
- Customers
- Accounts

Now, following the mapping rules as just described, we are going to create the tables and attributes for each table. We established primary keys to ensure every table has a unique identifier column for its records. Next, we need to define the relationships between the tables.

### Many to many relationships (N:M)

We've established this type of relationship between branches and employees, and between accounts and customers. An employee can work for any number of branches, and a branch could have any number of employees. Similarly, a customer could have many accounts, and an account could be a joint account held by more than two customers.

To model these relationships, we need two more intermediate entities. We create them as follows:

- account_customers
- branch_employees

The account_customers and branch_employees entities will be the bridges between account and customer entities and branch and employee entities, respectively. We are converting the N:M relationship into two 1:N relationships. You will see how the design looks in the next section.

## One to many relationship (1:N)

This type of relationship exists between branches and accounts and between customers and account_customers. This brings up the concept of the *nonidentifying relationship*. For example, in the `accounts` table, the `branch_id` field is not part of the primary key (one reason for this is that you can move your bank account to another branch). It is common nowadays to keep a surrogate key as the primary key in each table; therefore, a genuine identifying relationship where the foreign key is also part of the primary key in a data model is rare.

Because we're creating a physical EER model, we are also going to define the primary keys. It is common and recommended to use auto-incrementing unsigned fields for the primary key.

Figure 2-17 shows the final representation of the bank model.

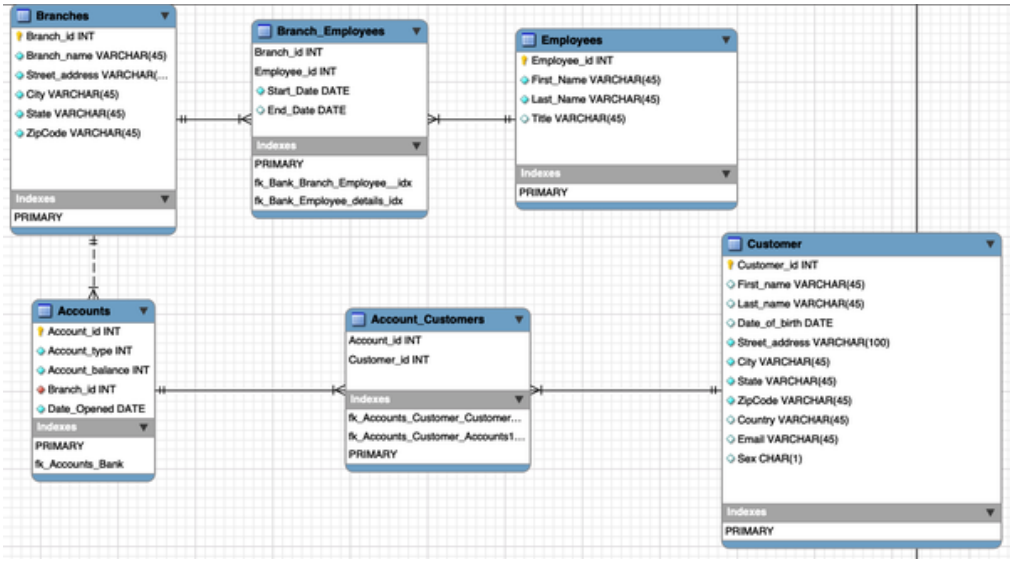


Figure 2-17. The EER model for the `bank` database

Note that there are items we haven't considered for this model. For example, our model does not support a customer with multiple addresses (say, a work address and a home address). We did this intentionally to emphasize the importance of collecting the requisites prior to database deployment.

You can download the model from the book's GitHub repository. The file is *bank_model.mwb*.

## Converting the EER to a MySQL Database Using Workbench

It's a good idea to use a tool to draw your ER diagrams; this way, you can easily edit and redefine them until the final diagrams are clear and unambiguous. Once you're comfortable with the model, you can deploy it. MySQL Workbench allows the conversion of the EER model into data definition language (DDL) statements to create a MySQL database, using the Forward Engineer option in the database menu (Figure 2-18).

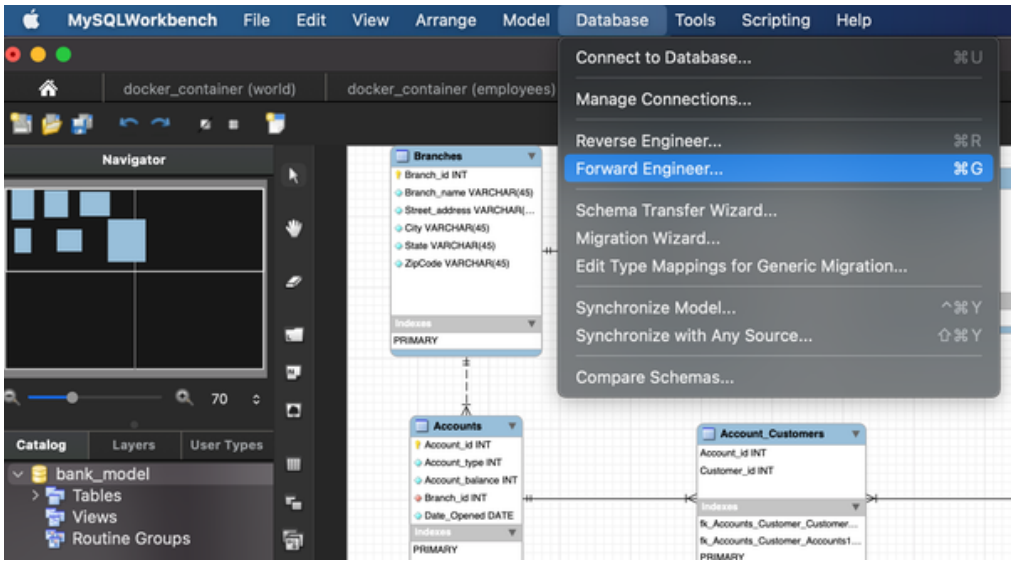


Figure 2-18. Forward Engineering a database in MySQL Workbench

You'll need to enter the credentials to connect to the database, and after that MySQL Workbench will present some options. For this model, we are going to use the standard options as shown in Figure 2-19, with all but the last option unchecked.

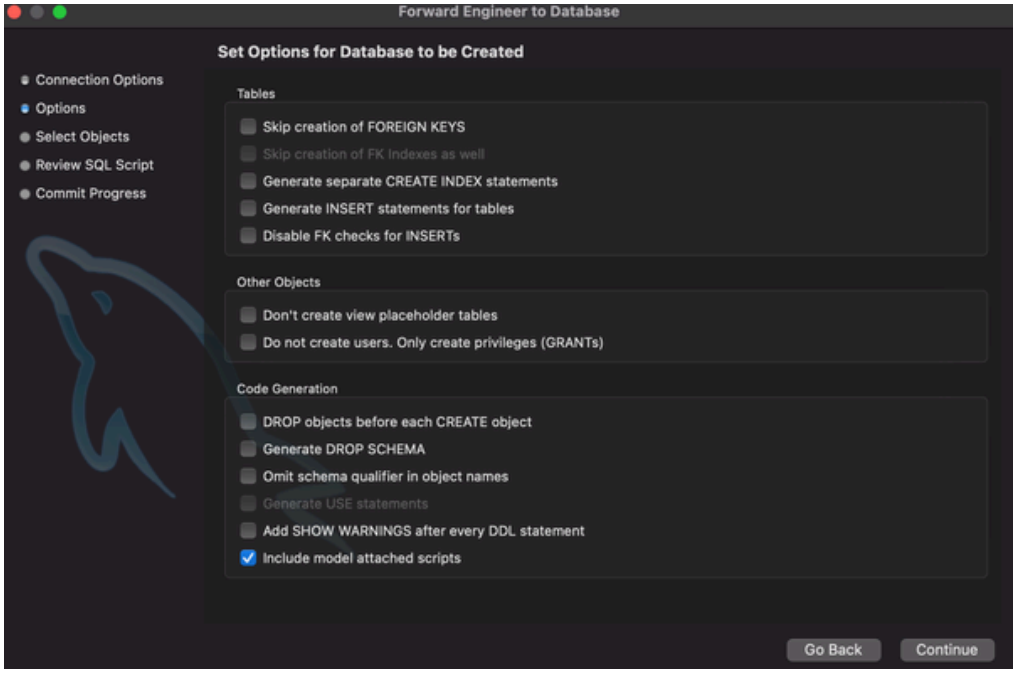


Figure 2-19. Database creation options

The next screen will ask which elements of the model we want to generate. Since we do not have anything special like triggers, stored procedures, users, and so on, we will only create the table objects and their relationships; the rest of the options are unchecked.

MySQL Workbench will then present us with the SQL script that will be executed to create the database from our model, as shown in Figure 2-20.

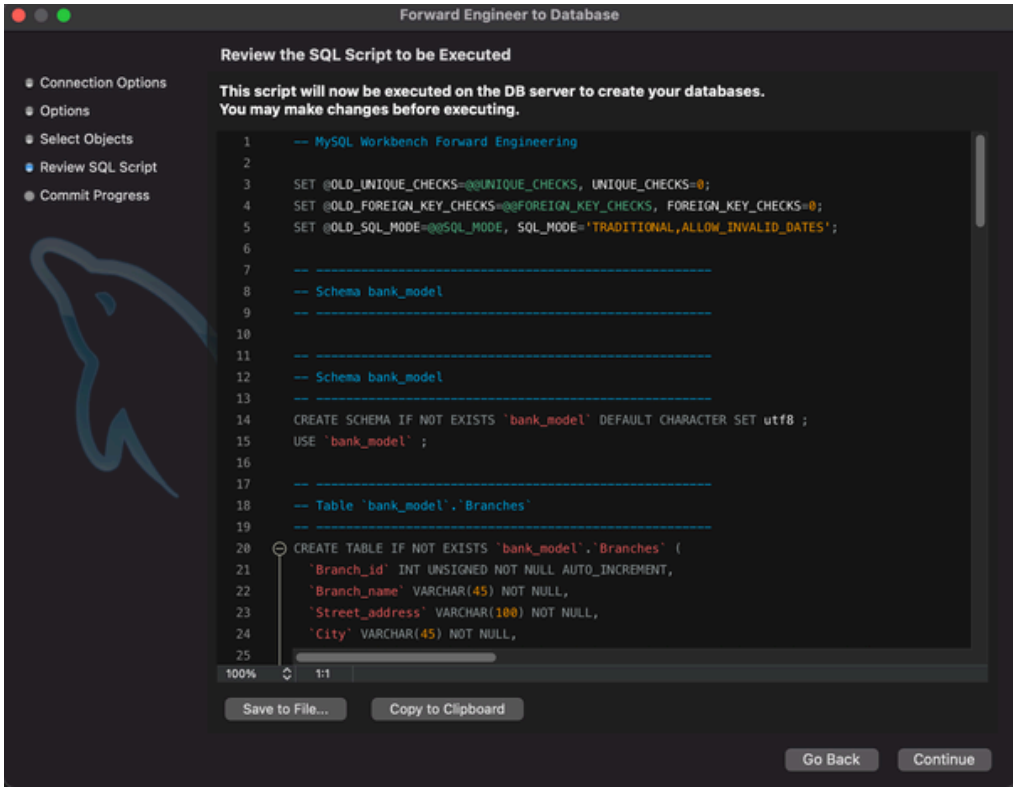


Figure 2-20. The script generated to create the database

When we click Continue, MySQL Workbench will execute the statements on our MySQL server, as shown in Figure 2-21.

We cover the details of the statements in this script in “Creating Tables”.

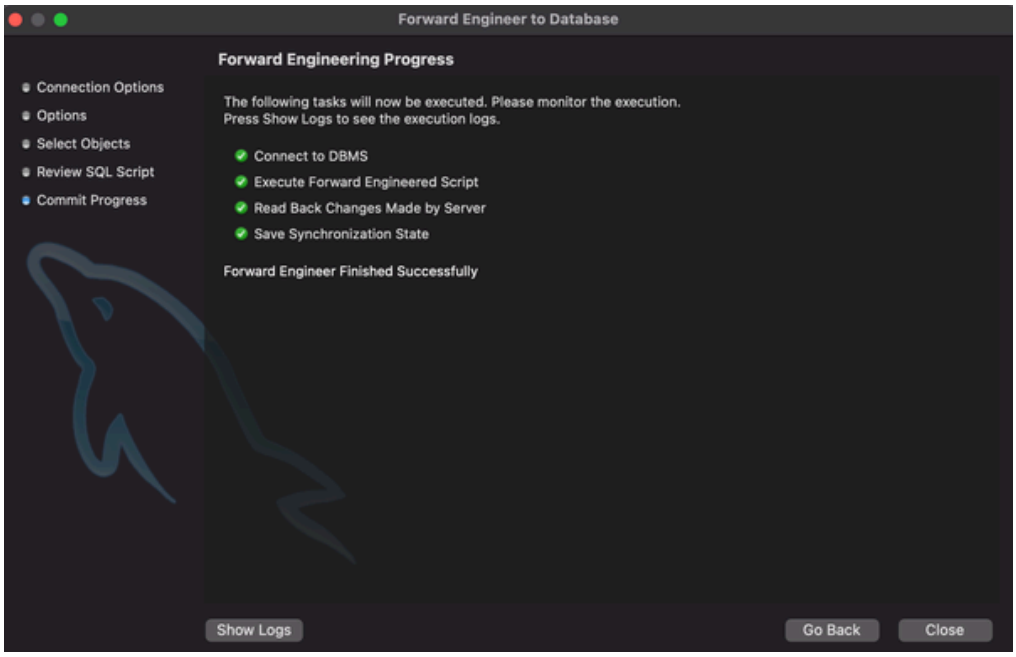


Figure 2-21. MySQL Workbench starts running the script