

*Persistence Models and Techniques  
for Java Database Programming*



Java™  
Database  
Best Practices

O'REILLY®

*George Reese*

---

# Java™ Database Best Practices

## Related titles from O'Reilly

Ant: The Definitive Guide

Building Java™ Enterprise Applications

Database Programming with  
JDBC and Java™

Developing JavaBeans™

Enterprise JavaBeans™

J2ME in a Nutshell

Java™ 2D Graphics

Java™ and SOAP

Java™ & XML

Java™ and XML Data Binding

Java™ and XSLT

Java™ Cookbook

Java™ Cryptography

Java™ Data Objects

Java™ Distributed Computing

Java™ Enterprise in a Nutshell

Java™ Examples in a Nutshell

Java™ Foundation Classes in a Nutshell

Java™ I/O

Java™ in a Nutshell

Java™ Internationalization

Java™ Message Service

Java™ Network Programming

Java™ NIO

Java™ Performance Tuning

Java™ Programming with Oracle SQLJ

Java™ Security

JavaServer™ Pages

Java™ Servlet Programming

Java™ Swing

Java™ Threads

Java™ Web Services

JXTA in a Nutshell

Learning Java™

Mac OS X for Java™ Geeks

NetBeans: The Definitive Guide

Programming Jakarta Struts

---

# Java™ Database Best Practices

*George Reese*

**O'REILLY®**

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

## CHAPTER 2

---

# Relational Data Architecture

*Good sense is the most evenly shared thing in the world, for each of us thinks that he is so well endowed with it that even those who are the hardest to please in all other respects are not in the habit of wanting more than they have. It is unlikely that everyone is mistaken in this. It indicates rather that the capacity to judge correctly and to distinguish true from false, which is properly what one calls common sense or reason, is naturally equal in all men, and consequently the diversity in our opinions does not spring from some of us being more able to reason than others, but only from our conducting our thoughts along different lines and not examining the same things.*

—René Descartes  
*Discourse on the Method*

Database programming begins with the database. A well-performing, scalable database application depends heavily on proper database design. Just about every time I have encountered a problematic database application, a large part of the problem sat in the underlying data model. Before you worry too much about writing Java code, it is important to lay the proper foundation for that Java code in the database.

Relational data architecture is the discipline of structuring databases to serve application needs while remaining scalable to future demands and usage patterns. It is a complex discipline well beyond the scope of any single chapter. We will focus instead on the core data architecture needs of Java applications—from basic data normalization to object-relational mapping.

Though knowledge of SQL (Structured Query Language) is not a requirement for this chapter, I use it to illustrate some concepts. I provide a SQL tutorial in the tutorial section of the book should you want to dive into SQL now. You will definitely need it as we get further into database programming.

# Relational Concepts

Before we approach the details of relational data architecture, it helps to establish a base understanding of relational concepts. If you are an experienced database programmer, you will probably want to move on to the next section on normalization. In this section, we will review the key concepts behind relational databases critical to an in-depth understanding of relational data architecture.

## Databases and Database Engines

Developers new to database programming often run into problems understanding just what a *database* is. In some contexts, it represents a collection of data like the music library. In other contexts, however, it may refer to the software that supports that collection, a process instance of the software, or even the server machine on which the process is running.

Technically speaking, a database is really the collection of related data and the relationships supporting the data. The database software—a.k.a the database management system (DBMS)—is the software, such as Oracle, Sybase, MySQL, and DB2, that is used to store that data. A database engine, in turn, is a process instance of the software accessing your database. Finally, the database server is the computer on which the database engine is running.

In the industry, this distinction is often understood from context. I will therefore continue to use the term “database” interchangeably to refer to any of these definitions. It is important, however, to database programming to understand this breakdown.

## The Relational Model

A database is any collection of related data. The files on your hard drive and the piles of paper on your desk all count as databases. What distinguishes a relational database from other kinds of databases is the mechanism by which the database is organized—the way the data is modeled. A relational database is a collection of data organized *in accordance with the relational model* to suit a specific purpose.

Relational principles are based on the mathematical concepts developed by Dr. E. F. Codd that dictate how data can be structured to define data relationships in an efficient manner. The focus of the relational model is thus the data relationships. In short, by organizing your data according to the relational model as opposed to the hierarchical principles of your filesystem or the random mess of your desktop, you can find your data at a later date much easier than you would have had you stored it some other way.

A relationship in relational parlance is a table with columns and rows.\* A row in the database represents an instance of the relation. Conceptually, you can picture a table as a spreadsheet. Rows in the spreadsheet are analogous to rows in a table, and the spreadsheet columns are analogous to table attributes. The job of the relational data architect is to fit the data for a specific problem domain into this relational model.

## Other Data Models

The relational model is not the only data model. Prior to the widespread acceptance of the relational model, two other models ruled data storage:

- The hierarchical model
- The network model

Though systems still exist based on these models, they are not nearly as common as they once were. A directory service like ActiveDirectory or OpenLDAP is where you are most likely to engage in new hierarchical development.

Another model—the object model—is slowly coming into favor for limited problem domains. As its name implies, it is a data model based on object-oriented concepts. Because Java is an object-oriented programming language, it actually maps best to the object model. However, it is not as widespread as the relational model and is definitely not proven to support systems on the scale of the relational model.

## Entities

The relational model is one of many ways of modeling data from the real world. The modeling process starts with the identification of the things in the real world that you are modeling. These real world things are called *entities*. If you were creating a database to catalog your music library, the entities would be things like compact disc, song, band, record label, and so on. Entities do not need to be tangible things; they can also be conceptual things like a genre or a concert.

**BEST PRACTICE** Capture the “things” in your problem domain as relational entities.

An entity is described by its *attributes*. Back to the example of a music library, a compact disc has attributes like its title and the year in which it was made. The individual values behind each attribute are what the database engine stores. Each row describes a distinct *instance* of the entity. A given instance can have only a single value for each attribute.

\* You will sometimes see a row referred to as a *tuple*—especially in more theoretical discussions of relational theory. Columns are often referred to as attributes or fields.

Table 2-1 describes the attributes for a CD entity and lists instances of that entity.

Table 2-1. A list of compact discs in a music library

| Artist                 | Title                                                  | Category    | Year |
|------------------------|--------------------------------------------------------|-------------|------|
| The Cure               | <i>Pornography</i>                                     | Alternative | 1983 |
| Garbage                | <i>Garbage</i>                                         | Grunge      | 1995 |
| Hole                   | <i>Live Through This</i>                               | Grunge      | 1994 |
| The Mighty Lemon Drops | <i>World Without End</i>                               | Alternative | 1988 |
| Nine Inch Nails        | <i>The Downward Spiral</i>                             | Industrial  | 1994 |
| Public Image Limited   | <i>Compact Disc</i>                                    | Alternative | 1986 |
| Ramones                | <i>Mania</i>                                           | Punk        | 1988 |
| The Sex Pistols        | <i>Never Mind the Bollocks, Here's the Sex Pistols</i> | Punk        | 1977 |
| Skinny Puppy           | <i>Last Rights</i>                                     | Industrial  | 1992 |
| Wire                   | <i>A Bell Is a Cup Until It Is Struck</i>              | Alternative | 1989 |

You could, of course, store this entire list in a spreadsheet. If you wanted to find data based on complex criteria, however, the spreadsheet would present problems. If, for example, you were having a “Johnny Rotten Night” party featuring music from the punk rocker, how would you create this list? You would probably go through each row in the spreadsheet and highlight the compact discs from Johnny Rotten’s bands.

Using the data in Table 2-1, you would have to hope that you had in mind an accurate recollection of which bands he belonged to. To avoid taxing your memory, you could create another spreadsheet listing bands and their members. Of course, you would then have to meticulously check each band in the CD spreadsheet against its member information in the spreadsheet of musicians.

## Constraints

What constitutes identity for a compact disc? In other words, when you look at a list of compact discs, how do you know that two items in the list are actually the same compact disc? On the face of it, the disc title seems as if it might be a good candidate. Unfortunately, different bands can have albums with the same title. In fact, you probably use a combination of the artist name and disc title to distinguish among different discs.

The artist and title in our CD entity are considered identifying attributes because they identify individual CD instances. In creating the table to support the CD entity, you tell the database about the identifying attributes by placing a *constraint* on the database in the form of a unique index or primary key. Constraints are limitations you place on your data that are enforced by the DBMS. In the case of unique indexes (primary keys are a special kind of unique index), the DBMS will prevent the insertion of two

rows with the same values for the entity's identifying attributes. The DBMS would prevent, for example, the insertion of another row with values of 'Ramones' and 'Mania' for the artist and title values in a CD table having artist and title as a unique index. It won't matter if the values for all of the other columns differ.

**BEST PRACTICE** Use constraints to help enforce the data integrity of your system.

Constraints like unique indexes help the DBMS help you maintain the overall data integrity of your database. Another kind of constraint is formally known as an *attribute domain*. You probably know the domain as its data type. Choosing data types and indexes along with the process of normalization are the most critical design decisions in relational data architecture.

## Indexes

An *index* is a constraint that tells the DBMS about how you wish to search for instances of an entity. The relational model provides for three main kinds of indexes:

### *Index*

An index in the generic sense is a simple tool that tells the DBMS what kind of searches you intend to perform. With this information, the DBMS can organize information to make the searches go quickly. A very crude way to think of an index is as a Java `HashMap` in which the key is your index attribute and the values are arrays of matching rows.

### *Unique index*

A unique index is an index whose values are guaranteed to be unique. In other words, instead of an array of matching rows, this index is like a `HashMap` that returns a single value for its key. The index created earlier for the artist and title columns in the CD table is an example of a unique index.

### *Primary key*

A primary key is a special unique index that acts as the main identifier for the row. A table can have any number of unique indexes, but it can have only one primary key.

We can examine the impact of indexes by creating the CD entity as a table in a MySQL database and using a special SQL command called the `EXPLAIN` command. The SQL to create the CD table looks like this:

```
CREATE TABLE CD (  
  artist      VARCHAR(50) NOT NULL,  
  title       VARCHAR(100) NOT NULL,  
  category    VARCHAR(20),  
  year        INT  
);
```

The EXPLAIN command tells you what the database will do when trying to run a query. In this case, we want to look at what happens when we are looking for a specific compact disc:

```
mysql> EXPLAIN SELECT * FROM CD
-> WHERE artist = 'The Cure' AND title = 'Pornography';
+-----+-----+-----+-----+-----+-----+-----+-----+
| table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| CD    | ALL  | NULL          | NULL | NULL    | NULL | 10 | where used |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

The important information in this output for now is to look at the number of rows. Given the data in Table 2-1, we have 10 rows in the table. The results of this command tell us that MySQL will have to examine all 10 rows in the table to complete this query. If we add a unique index, however, things look much better:

```
mysql> ALTER TABLE CD ADD UNIQUE INDEX ( artist, title );
Query OK, 10 rows affected (0.20 sec)
Records: 10 Duplicates: 0 Warnings: 0
mysql> EXPLAIN SELECT * FROM CD
-> WHERE artist = 'The Cure' AND title = 'Pornography';
+-----+-----+-----+-----+-----+-----+-----+-----+
| table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+
| CD    | const | artist        | artist | 150    | const,const | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
mysql>
```

The same query can now be executed simply by examining a single row.

**BEST PRACTICE** Make indexes for attributes you intend to search against.

Unfortunately, the artist and title probably make a poor unique index. First of all, there is no guarantee that a band will actually choose distinct names for its albums. Worse, in some circumstances, bands have chosen to have the same album carry different names. Public Image Limited’s *Compact Disc* is an example of such an album. The cassette version of the album is called *Cassette*.

Even if artist and title were solid identifying attributes, they still make for a poor primary key. A primary key must meet the following requirements:

- It can never be NULL.
- It must be unique across all entity instances.
- The primary key value must be known when the instance is created.

In addition to these requirements, good primary keys have the following characteristics:

- The primary key should never change value.
- The primary key attributes should have no meaning except to uniquely identify the entity instance.

It is very common for people to find attributes inherent in an entity and chose one or more of those identifying attributes as a primary key. Perhaps the best example of this practice is the use of an email address as a primary key. Email addresses, however, can and do change. A change to a primary key attribute can cause an instance to become inaccessible to anyone with old information about the instance. In plain English, it can break your application.

Another example of a common primary key with meaning is a U.S. Social Security number. It is supposed to be unique. It is never supposed to change. You, however, have no control over its uniqueness or whether it changes. As it turns out, sometimes the uniqueness of Social Security numbers is violated. In addition, they do sometimes change. Furthermore, in many cases, the law restricts your ability to share this information. It is therefore best to choose a primary key with no external meaning; you will control exactly how it is used and have the full power to enforce its uniqueness and immutability.

**BEST PRACTICE** Never use meaningful attributes or attributes whose values can change as primary keys.

The solution is to create a new attribute to serve as the primary identifier for instances of an entity. For the CD table, we will call this new attribute the `cdID`. The SQL to create the table then looks like this:

```
CREATE TABLE CD (  
    cdID          INT          NOT NULL,  
    artist        VARCHAR(50)  NOT NULL,  
    title         VARCHAR(100) NOT NULL,  
    category      VARCHAR(20),  
    year          INT,  
    PRIMARY KEY ( cdID ),  
    INDEX ( artist, title ),  
    INDEX ( category ),  
    INDEX ( year )  
);
```



You may have noted that my naming style does not redundantly name columns like `title cdTitle`. Yet I chose to name the primary key for the CD table `cdID` instead of `id`. This choice basically makes the use of data modeling tools a lot simpler. In short, data modeling tools look for natural joins—joins between two tables when the common columns share the same name, data type, and value. I discuss natural joins in more detail in Chapter 10.

**BEST PRACTICE** Include the table name in the primary key name to assist data modeling tools.

Ideally, you always search on unique indexes. In the real world, however, you will select on attributes like the year or genre that are not unique. You can still help the database organize the underlying data storage by creating plain indexes. In general, you want any attribute you commonly search on to be indexed. An index does, however, come with some downsides:

- Indexes are stored apart from the table data. Every index thus adds to the disk space requirements of the database.
- Every change to the table requires every index to be updated to reflect the changes.

In other words, if you have a table on which you perform a significant number of write operations, you want to minimize your indexes to those attributes that appear frequently in queries.

Finally, as you have already seen, you can have indexes—including primary keys—that are formed out of any number of identifying columns so long as those columns together sufficiently identify a single entity instance. It is always a good idea, however, to build primary keys out of the minimal number of columns possible.

**BEST PRACTICE** Use the smallest number of columns possible in your primary keys.

## Domains

The proper choice of data type is another critical aspect of relational data architecture. It constrains the kind of data that can be stored for a given attribute. By creating an email attribute as a text value, you prevent people from storing numbers in the field. A time-oriented domain like a SQL DATE enables you to perform time arithmetic on date values.

The domains that exist in a relational database depend on the DBMS of choice. Those that support the SQL specification generally support a core set of data types. Just about every database engine comes with its own, proprietary data types. When modeling a system, you should use SQL-standard data types.

Primary keys deserve special consideration when you are putting domain constraints on an entity. Because they are the primary mechanism for getting access to an entity instance, it is important that the database is able to do quick matches against primary key values. In general, numeric types form the best primary keys. I recommend the use of 64-bit, sequentially generated integers for primary key columns. The only exception is for lookup tables.

A lookup table is a small table with a known, finite set of data like a table containing a list of states or, with respect to the music library example, a set of genres. In the

case of lookup tables, they more often than not have codes against which you will do most lookups. For example, you will almost always retrieve the state of Maine from a State table by its abbreviation ME. It therefore makes more sense to use fixed character data types like SQL's CHAR for primary keys in lookup tables. The length of these fixed character values should be no more than a few characters.

|                                                                                                  |
|--------------------------------------------------------------------------------------------------|
| <b>BEST PRACTICE</b> Use fixed character data types like CHAR for primary keys in lookup tables. |
|--------------------------------------------------------------------------------------------------|

The data types for other kinds of attributes vary with the diversity in the kinds of data you will want to store in your databases. These days, many databases even support the creation of user-defined data types. These pseudo-object data types prove particularly useful in the development of Java database applications.

## Relationships

The creation of relationships among the entities in the database lies at its heart. These relationships enable you to easily answer the question, “On what compact discs in my library did Johnny Rotten play?” Unlike other models, the relational model does not create hard links between two entities. In the hierarchical model, a hard relationship exists between a parent entity and its child entities. The relational model, on the other hand, creates relationships by matching a primary key attribute in one entity to a *foreign key* attribute in another entity.

The relational model supports three kinds of entity relationships:

- One-to-one
- One-to-many
- Many-to-many

With any of these relationships, one side of the relationship may be optional. An optional relationship allows the foreign key to contain NULL values to indicate the relationship does not exist for that row.

### One-to-one relationships

The one-to-one relationship is the most rare relationship in the relational model. A one-to-one relationship says that for every instance of entity A, there is a corresponding instance of entity B. It is so rare that its appearance in a data model should be met with skepticism as it generally indicates a design flaw. You indicate a one-to-one relationship in the same way you indicate a one-to-many relationship.

|                                                                                                                                                       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>BEST PRACTICE</b> Recheck your design whenever you encounter one-to-one relationships, as they are often indicators of problematic design choices. |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|

## One-to-many relationships

A one-to-many relationship means that for every instance of entity A, there can be multiple instances of entity B. As Figure 2-1 shows, the “many” side of the relationship houses the foreign key that points to the primary key of the “one” side of the relationship.

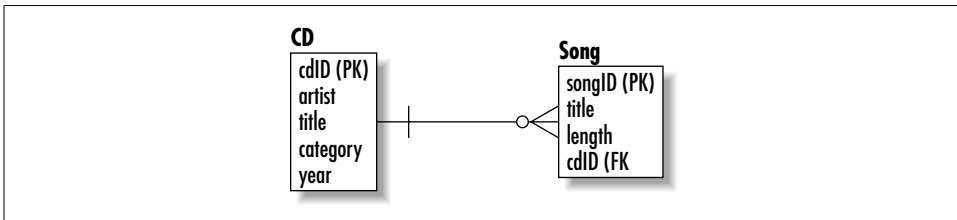


Figure 2-1. A One-to-Many Relationship

Table 2-2 lists data from a Song table whose rows are dependent on rows in the CD table.

Table 2-2. The Song entity with a foreign key from the CD entity

| Attribute | Domain       | Notes       | NULL? |
|-----------|--------------|-------------|-------|
| songID    | INT          | PRIMARY KEY | No    |
| cdID      | INT          | FOREIGN KEY | No    |
| title     | VARCHAR(100) |             | No    |
| length    | INT          |             | No    |

Under this design, one compact disc is associated with many songs. The placement of cdID into the Song table as a foreign key indicates the dependency on a row of the CD table. In databases that manage foreign key constraints, this dependency will prevent the insertion of songs into the Song table that do not already have a corresponding CD. Similarly, the deletion of a disc will cause the deletion of its associated songs. You should note, however, that not all database engines support foreign key constraints. Of those that do support them, you often have the option of turning them on or off.



Why would you want foreign key constraints off? Many application environments—particularly multitier distributed object systems—prefer to manage dependencies in the object layer instead of the database. It is generally a trade-off between a combination of speed with object purity and guaranteed data integrity. When foreign key constraints are not checked in the database, updates occur more quickly. Furthermore, you do not end up with a situation in which objects exist in the middle tier that have been automatically deleted by the database. On the other hand, if your middle-tier logic is not sound, your application can damage the data integrity without proper foreign key constraints.

You now have a proper relationship between compact discs and their songs. To ask which songs are on a particular compact disc, you need to ask the Song table which songs have the disc's cdID. Assuming you are looking for all songs from the disc *Garbage* (cdID 2), the SQL to find the songs looks like this:

```
SELECT songID, title FROM Song WHERE cdID = 2;
```

More powerfully, however, you can ask for all songs from a compact disc by the disc title:

```
SELECT Song.songID, Song.title
FROM Song, CD
WHERE CD.title = 'Last Rights'
AND CD.cdID = Song.cdID;
```

The last part of the query where the cdID was compared in both tables is called a *join*. A join is where the implicit relationship between two tables becomes explicit.

### Many-to-many relationships

A many-to-many relationship allows an instance of entity A to be associated with multiple instances of entity B and an instance of entity B to be associated with multiple instances of entity A. These relationships require the creation of a special table to manage the relationship. You may hear these tables referred to by any number of names: composite entities, join tables, cross-reference tables, and so forth. This extra table creates the relationship by having the primary keys of each table in the relationship as foreign keys. It then uses the combination of foreign keys as its own compound primary key. If, for example, we had an Artist table in our music library, we indicate a many-to-many relationship between an Artist and a CD through an ArtistCD join table. Table 2-3 shows this special table.

Table 2-3. The ArtistCD table creates a many-to-many relationship between Artist and CD

| Attribute | Domain | Notes                    | NULL? |
|-----------|--------|--------------------------|-------|
| cdID      | INT    | FOREIGN KEY, PRIMARY KEY | No    |
| artistID  | INT    | FOREIGN KEY, PRIMARY KEY | No    |

You can now ask for all of the compact discs by Garbage:

```
SELECT CD.cdID, CD.title
FROM CD, ArtistCD, Artist
WHERE ArtistCD.cdID = CD.cdID
AND ArtistCD.artistID = Artist.artistID
AND Artist.name = 'Garbage';
```

|                                                                           |
|---------------------------------------------------------------------------|
| <b>BEST PRACTICE</b> Use join tables to model many-to-many relationships. |
|---------------------------------------------------------------------------|

Another useful aspect of join tables is that you can use them to contain information about a relationship. If, for example, you wanted to track guest artists on albums, where would you store that information? It really is not an attribute of an artist or a compact disc. It is instead an attribute of the relationship between the two entities. To capture this information, you would therefore add a column to ArtistCD called guest. Finding which compact discs on which Sting appeared as a guest artist would then be as simple as:

```
SELECT CD.cdID, CD.title
FROM CD, ArtistCD, Artist
WHERE ArtistCD.cdID = CD.cdID
AND ArtistCD.artistID = Artist.artistID
AND Artist.name = 'Sting'
AND ArtistCD.guest = 'Y';
```

## NULL

NULL is a special value in relational databases that indicates the absence of a value. If you have a pet store site that gathers information on your users, for example, you may track the number of pets your users have. Without the concept of NULL, you have no proper way to indicate that you do not know how many pets a user has. Applications commonly resort to nonsense values (like -1) or unlikely values (like 9999) as a substitute for NULL.

|                                                                       |
|-----------------------------------------------------------------------|
| <b>BEST PRACTICE</b> Use NULL to represent unknown or missing values. |
|-----------------------------------------------------------------------|

Though the basic concept of NULL is pretty straightforward, beginning database programmers often have trouble figuring out how NULL works in database operations. A basic example would come about by adding a new column to our Song table that is a rating. It can be NULL since it is unlikely anyone wants to rate every single song in their library. The following SQL may not do what you think:

```
SELECT songID, title FROM Song WHERE rating = NULL;
```

No matter what data is in your database, this query will always return zero rows. Relational logic is not Boolean; it is three-value logic: true, false, and unknown. Most NULL comparisons therefore result in NULL since a NULL comparison is indeterminate under three-value logic. SQL provides special mechanisms to test for NULL in the form of IS NULL and IS NOT NULL so that it is possible to ask for the unrated songs:

```
SELECT songID, title FROM Song WHERE rating IS NULL;
```

## Modeling

Throughout this book, I will be using industry-standard diagrams to illustrate designs. A critical part of relational data architecture is understanding a special kind

of diagram called an entity relationship diagram, or ERD. An ERD graphically captures the entities in your problem domain and illustrates the relationships among them. Figure 2-2 is the ERD of the music library database.

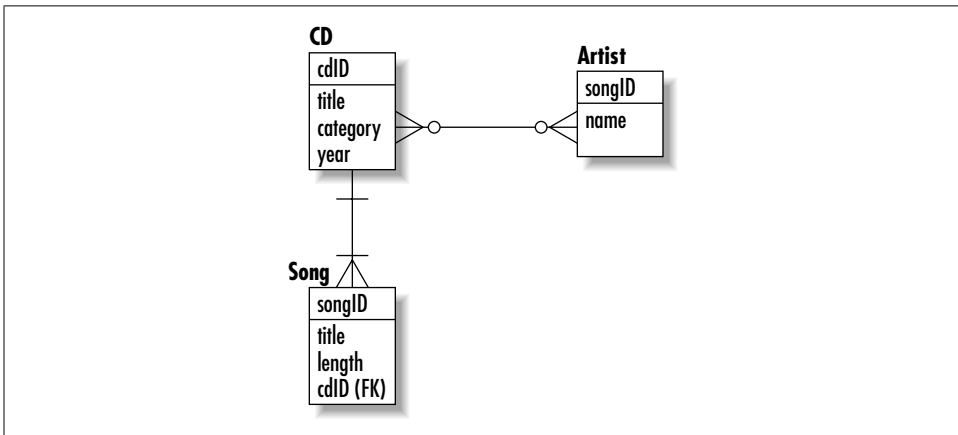


Figure 2-2. The ERD for the music library

There are in fact several forms of ERDs. In the style I use in this book, each entity is indicated by a box with the name of the entity at the top. A line separates the name of the entity from its attributes inside the box. Primary key attributes have “PK” after them, and foreign key attributes have “FK” after them.

The lines between entities indicate a relationship. At each end of the relationship are symbols that indicate what type of relationship it is and whether it is optional or mandatory. Table 2-4 describes these symbols.

Table 2-4. Symbols for an ERD

| Symbol | Description                                                           |
|--------|-----------------------------------------------------------------------|
|        | The many side of a mandatory one-to-many or many-to-many relationship |
|        | The one side of a mandatory one-to-one or one-to-many relationship    |
|        | The many side of an optional one-to-many or many-to-many relationship |
|        | The one side of an optional one-to-one or one-to-many relationship    |

Our ERD therefore says the following things:

- One compact disc contains one or more songs.
- One song appears on exactly one compact disc.
- One compact disc features one or more artists.
- One artist is featured on one or more compact discs.
- An artist can optionally be part of one or more artists (bands).

This ERD is a *logical* representation of the music library. The entities in a logical model are not tables. First of all, you probably noticed there is no composite entity handling the relationship between an artist and a compact disc—I have drawn the relation directly as a many-to-many relationship. Furthermore, all of the entity names and attributes are in plain English. Finally, no foreign keys are shown.

**BEST PRACTICE** Develop an ERD to model your problem before you create the database.

The physical data model transforms the logical data model into the tables that will be created in the working database. A data architect works with the logical data model while DBAs (database administrators) and developers work with the physical data model. You translate the logical data model into a physical one by adding join tables, turning domains into database-specific data types, and using table and column names appropriate to your DBMS. Figure 2-3 shows the physical data model for the music library as it would be created in MySQL.

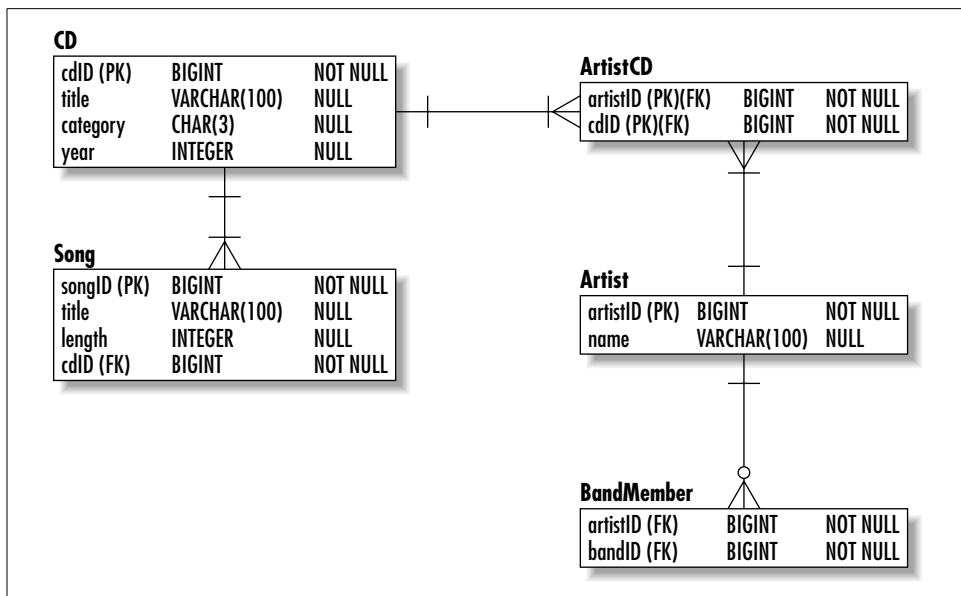


Figure 2-3. The physical data model for the music library

## Normalization

When beginning the development of a data architecture for a project, you first want to capture all the entities in your problem domain and the attributes associated with those entities. Depending on your software engineering processes, these entities may be driven by your object model or your object model may be driven by the logical ERD. Either way, you should not initially concern yourself in any way with issues like performance, scalability, or flexibility—the task is to model the problem domain properly.

Unlike other areas of software architecture, relational data architecture provides a very formal process for optimizing your model for efficient resource usage, scalability, and flexibility. This formal process is known as *normalization*. Normalization seeks to achieve the following goals:

*Remove redundant data*

A fully normalized database repeats nothing other than foreign keys. Removal of redundant data guarantees that you are storing the minimum data necessary to model your domain and protects the integrity of your data by requiring just a single point of maintenance for any piece of information.

*Protect the relational model*

The process of normalization forces you to examine all aspects of your data model to make certain that you are not violating any of the basic principles of the relational model (e.g., all attributes must be single-valued; only the table name, column name, and primary key value should be needed to identify a row; etc.).

*Improve scalability and flexibility*

A normalized database guarantees the ability of the data model to evolve with even the most drastic of changes in the problem domain with a minimal impact on the applications it supports.

As I noted earlier, normalization is a formal process. It defines very specific criteria for your data model that it breaks out into *normal forms*. The normal forms establish a stringent and objective set of rules to which a data model must adhere. Each one builds on requirements of the previous, as is shown in Figure 2-4, and improves upon the overall design of the model.

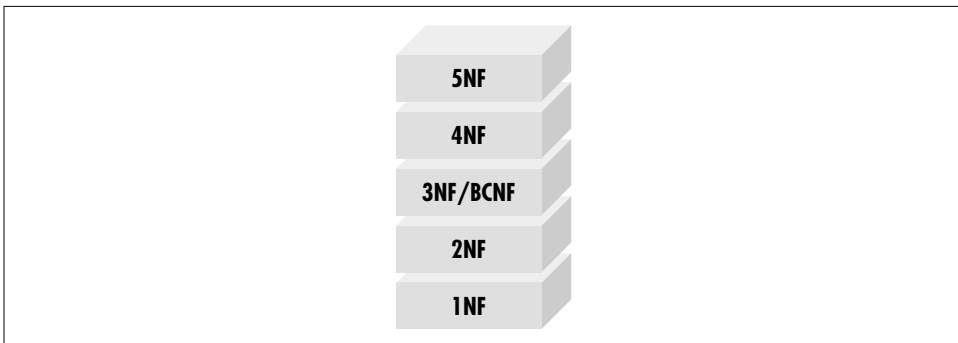


Figure 2-4. The six normal forms build on top of one another

Before a data model can be said to be in a certain normal form, it must meet all of the requirements of that normal form and any lesser normal forms. The second normal form, for example, necessitates that a data model meet the requirements for both the first and second normal forms.

No matter what your problem domain, you will want to normalize your data model at least to the third normal form. For most simple problem domains, the third

normal form is good enough. Deeper normal forms represent specific data modeling issues that do not apply to most data models. Most data models in the third normal form are therefore already in the fifth normal form. If you have a very complex system, you should go ahead and verify that it is in at least the fourth normal form. Formally normalizing your data model to the fifth normal form should be left for very specific problem domains. I will dive into the details of each of these normal forms later in the chapter.

**BEST PRACTICE** Very complex systems should be normalized to the fourth normal form.

In addition to the six normal forms noted here, a seventh normal form called the domain/key normal form (DKNF) exists. The rule for DKNF is that every logical restriction on attribute values results from the definition of keys and domains. In theory, a table in DKNF cannot contain anomalies. If nothing about DKNF makes any sense to you, don't worry about it—no process exists to prove a table is in DKNF and therefore it is not used in real world modeling.

## Before Normalization

Before you begin the process of normalization, you should already have a logical ERD describing your problem domain. This logical ERD should describe all of the entities that make up the problem domain, their major attributes, and their relationships. For the purposes of this section, I will be referring to a data model to support a web site for film fans. It specifically stores information about films and enables people to browse the films in the database based on that information. Figure 2-5 shows the data model before normalization.

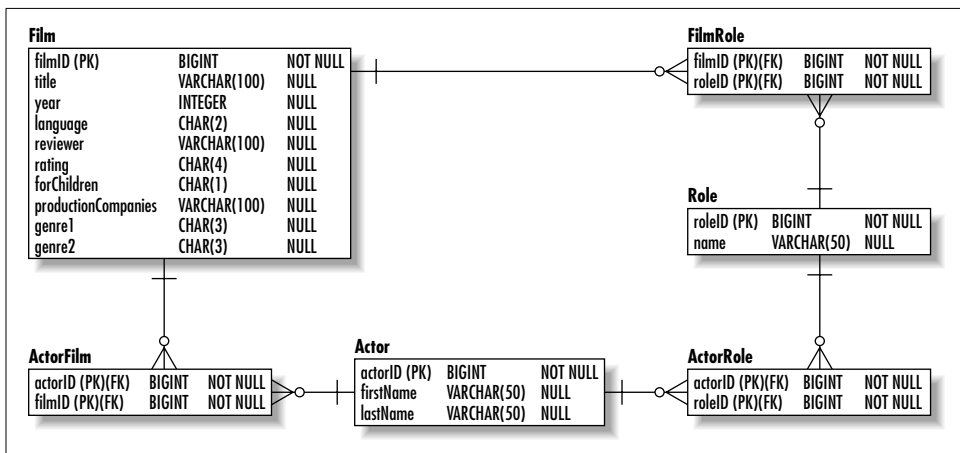


Figure 2-5. The raw film site data model

Because we want this web site to be accessible to all North American visitors, it has translations into Spanish and French. These translations are supported through a duplication of the logical data model into three separate physical databases.

## Basic Normalization

Basic normalization addresses the design demands common to any relational database. As a data architect, you will always want to carry your design through to the third normal form.

### First normal form

A table is in the first normal form (1NF) when all attributes are single-valued. This requirement is not simply a good design requirement; it is a fundamental requirement of the relational model. At its simplest, it means that only a single value may exist at the intersection of a column and a row.

The film database has three different violations of 1NF:

- The `productionCompanies` attribute in the `Film` table is multivalued.
- The `genre1` and `genre2` columns in effect represent a multivalued attribute.
- The duplication of the database for multilingual content also represents turning all values into multivalued attributes.

The problem with the first violation is that it makes the database very inflexible. For one thing, searching for films by a specific production company is difficult. You cannot use a simple equality check like:

```
SELECT filmID, title
FROM Film
WHERE productionCompanies = 'Imaginary Productions';
```

That column, after all, contains a comma-separated list of companies. Instead, you need a much less efficient query like:

```
SELECT filmID, title
FROM Film
WHERE productionCompanies LIKE '%Imaginary Productions%';
```

The solution to the problem of multivalued attributes is to create a new entity to support that attribute. In the case of the film database, we should create a `ProductionCompany` table with foreign key references to the primary key in the `Film` table. We now have a one-to-many relationship between films and their production companies.

Another, less obvious multivalued attribute is the genre support for films. Because of a need to support the classification of films like *Blazing Saddles* that fall into two genres, we have in our data model two genre columns. This approach has several problems associated with it.

The problems are:

- It limits the assignment of genres to films to two genres.
- Searching for a film by genre becomes a complex operation.
- Space is wasted for any film with a single genre.

The solution, again, is the creation of a new entity to support the multiple values. In this case, we will create a lookup table for genres and a many-to-many relationship between `Film` and `Genre`.

The final problem with the raw data model is the fact that the entire database is duplicated for every language we want to support. In order to add a new language, we need to create a new duplicate of that database and replace its text values with translations for the target language. The application then needs to be configured to use that database as a data source for the new language.

For this problem, we need to create translation entities for each of the text attributes in the database. Adding support for a new language means nothing more than adding new rows to each translation table.

Figure 2-6 contains the film database in 1NF.

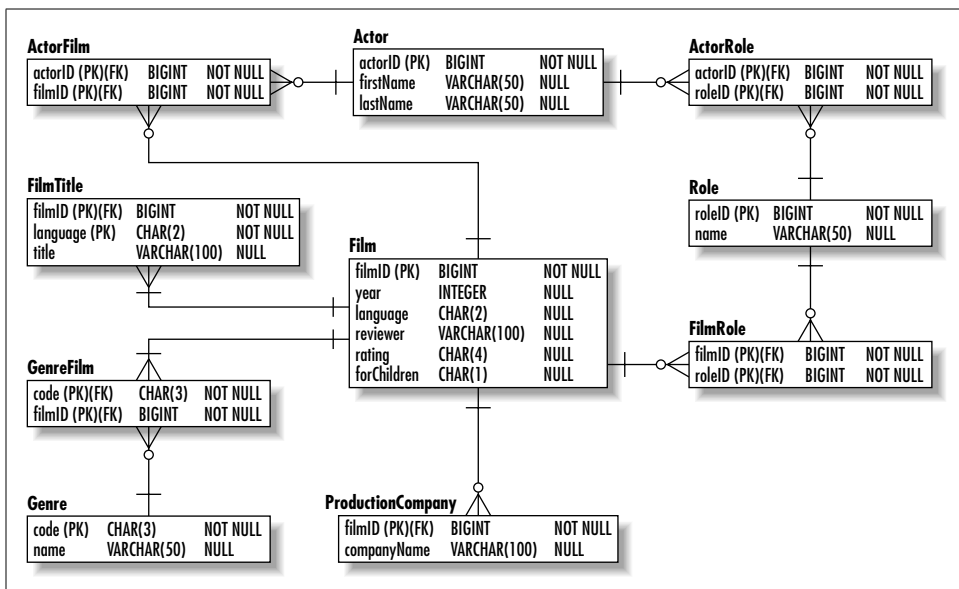


Figure 2-6. The film database in 1NF

## Second normal form

A table is in the second normal form (2NF) when it is in 1NF and all non-key attributes are *functionally dependent* on the table's entire primary key. Functional

dependency means that an attribute is determined by another attribute. In the case of `filmID` and `title`, the `title` is functionally dependent on the `filmID` because which film the row represents determines what its title is. On the other hand, the title of the film does not necessarily indicate which film you are dealing with.

When an attribute is not dependent on the entire primary key of the table it is in, it has likely been placed in the wrong table. Our data model has this problem in the `reviewer` attribute of the `Film` entity. The purpose of this attribute is to capture the name of the person who initially reviewed the film for the site. The `reviewer` attribute, however, does not depend on the `filmID`—the reviewer exists independent of the film.

The existence of attributes that violate 2NF causes database anomalies. A database anomaly is an error or inconsistency that occurs when some event takes place. Specifically, there are:

#### *Insertion anomalies*

An insertion anomaly occurs when you are forced to know information about an entity instance that may not yet be knowable in order to create an instance of another entity. In the case of `reviewer`, a person cannot be a reviewer until he has reviewed a film. More to the point, we cannot capture any information about our reviewers until they have reviewed a film.

#### *Deletion anomalies*

A deletion anomaly occurs when a delete causes data that is not related to the instance being deleted to be removed from the database. In our existing model, removing a film may remove the reviewer from the database.

#### *Update anomalies*

An update anomaly occurs when the same data must be changed in more than one location to preserve database integrity. If a reviewer has a name change, our data model requires the change be made to each film reviewed and every other place in the database with that reviewer's name.

Again, the solution to this normalization problem is the creation of a new entity to remove the nondependent attribute. This entity, `Reviewer`, contains the name of the reviewer and is related to many films.

Figure 2-7 shows our data model in 2NF.

### **Third normal form**

A table is in the third normal form (3NF) when it is in 2NF and no *transitive dependencies* exist. A transitive dependency occurs when a functional dependency is inherited through some other identifying attribute. In our data model, the `forChildren` attribute depends on the `rating` attribute, which in turn depends on the `filmID`. Because the `forChildren` attribute has no direct dependency on the `filmID`, it is thus transitively dependent on the `filmID`.

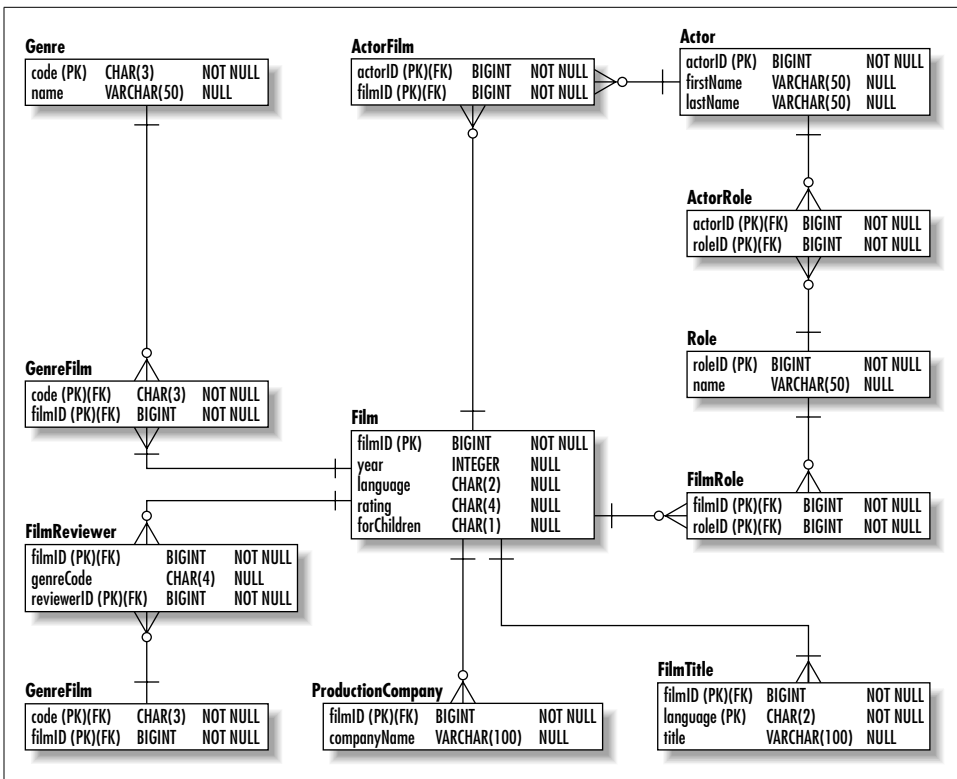


Figure 2-7. The film database in 2NF

Violation of 3NF causes database anomalies. First of all, if the MPAA changes which ratings are suitable for children, you will need to update every instance of the *Film* entity to reflect that change. An insertion anomaly also exists in that any new ratings for children will not be reflected in existing films. Of course, the insertion anomaly is not a huge problem for this database since films rarely change ratings. Finally, deletion of the only row with a rating associated with being for children causes us to lose all information about it being for children.

**BEST PRACTICE** Normalize your data model minimally to the third normal form.

To fix this problem, you need to move the transitively dependent value into a table that provides functional dependency. In other words, move the *forChildren* attribute into the *Rating* table as shown in Figure 2-8.

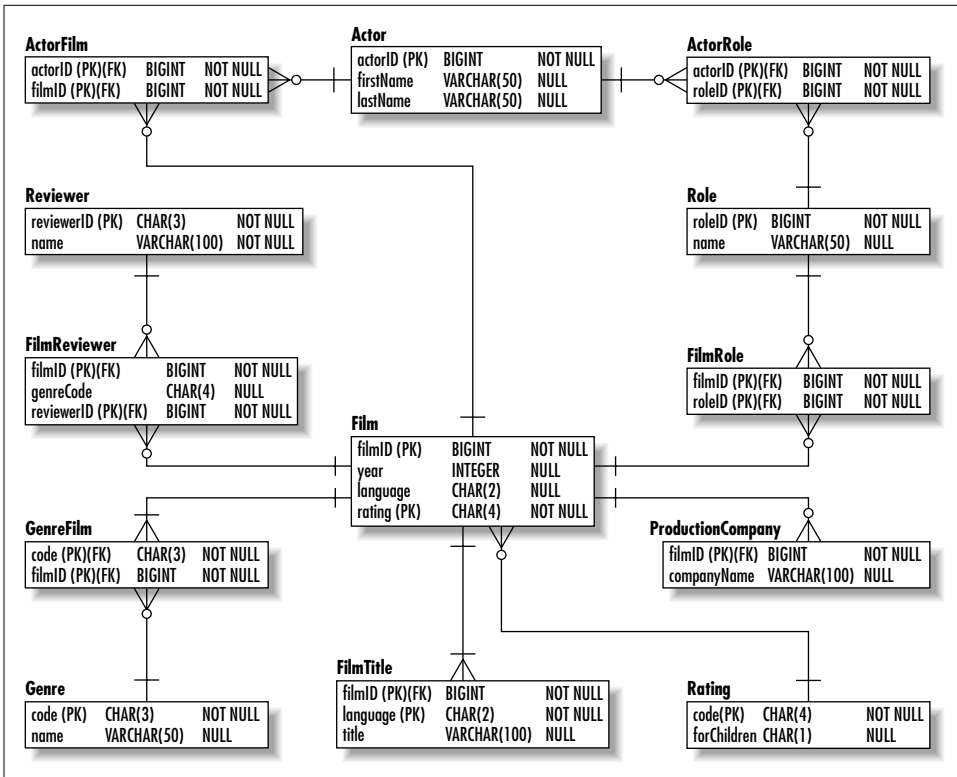


Figure 2-8. The film database in 3NF

## Specialized Normalization

Having your database in 3NF is generally good enough to guarantee your system is free of the most common anomalies. The other forms of normalization handle special situations. In fact, if your database is not subject to the special considerations of Boyce-Codd normal form or fourth normal form, your database is automatically in 4NF. The fifth normal form is impossible to verify without computer-aided modeling tools and is rarely worth seeking.

### Boyce-Codd normal form

A table is in Boyce-Codd normal form (BCNF) when every determinant is a candidate key. A *candidate key* is a set of attributes that could potentially serve as a primary key. BCNF is essentially a more generalized form of 3NF. It specifically addresses issues that arise in tables with one or more of the following characteristics:

- Multiple candidate keys
- Composite candidate keys
- Overlapping candidate keys

Our data model contains no relations to which BCNF applies. To illustrate BCNF, consider a table that contains three or more columns with a couple of the combinations capable of uniquely identifying a row. An example might be a *Showing* table that represents when a real estate agent shows a house to a client. The table has the structure shown in Table 2-5.

Table 2-5. The structure of a table meeting BCNF

| Attribute  | Domain       | Notes                    | NULL? |
|------------|--------------|--------------------------|-------|
| propertyID | BIGINT       | PRIMARY KEY, FOREIGN KEY | No    |
| agentID    | BIGINT       | PRIMARY KEY, FOREIGN KEY | No    |
| timeslot   | INT          | PRIMARY KEY, FOREIGN KEY | No    |
| buyerID    | BIGINT       | FOREIGN KEY              | No    |
| notes      | VARCHAR(255) |                          |       |

For the sake of this example, assume that a buyer gets only one chance to view a property. Furthermore, only one agent can show a property to one buyer in a given time slot. In that case, it is possible for notes to be determined by either of the following combinations:

- propertyID, agentID, timeslot
- propertyID, agentID, buyerID

You could choose either of the two combinations. BCNF simply states that as long as every column that determines notes is a candidate key, the table is in BCNF.

#### Fourth normal form

A table is in the fourth normal form when it is in BCNF and all multivalued dependencies are also functional dependencies. The problem here with the current model is the *FilmReviewer* table. It ties film reviewers with the films and genres they review. Table 2-6 shows some sample data from the table.

Table 2-6. Data in *FilmReviewer*

| filmID | reviewerID | genreCode |
|--------|------------|-----------|
| 101    | 1          | ACT       |
| 101    | 1          | SCI       |
| 102    | 2          | DRA       |
| 102    | 2          | COM       |
| 103    | 1          | ACT       |
| 103    | 1          | SCI       |

The full set of columns forms the primary key for this table. It is thus normalized to BCNF. Unfortunately, it still contains redundant data. The redundancy is caused by

multivalued dependencies. Specifically, reviewerID determines the values of filmID and genreCode independently. In the relations we have seen so far, the determinant establishes the full set of values that together form the instance.

We can fix this problem by splitting genreCode's dependence into one table and reviewerID's dependence into another. For example, we can create a ReviewGenre table that captures the genres the reviewer specializes in. We can similarly create a ReviewerFilm table that contains the film reviews. Figure 2-9 shows the resulting data model.

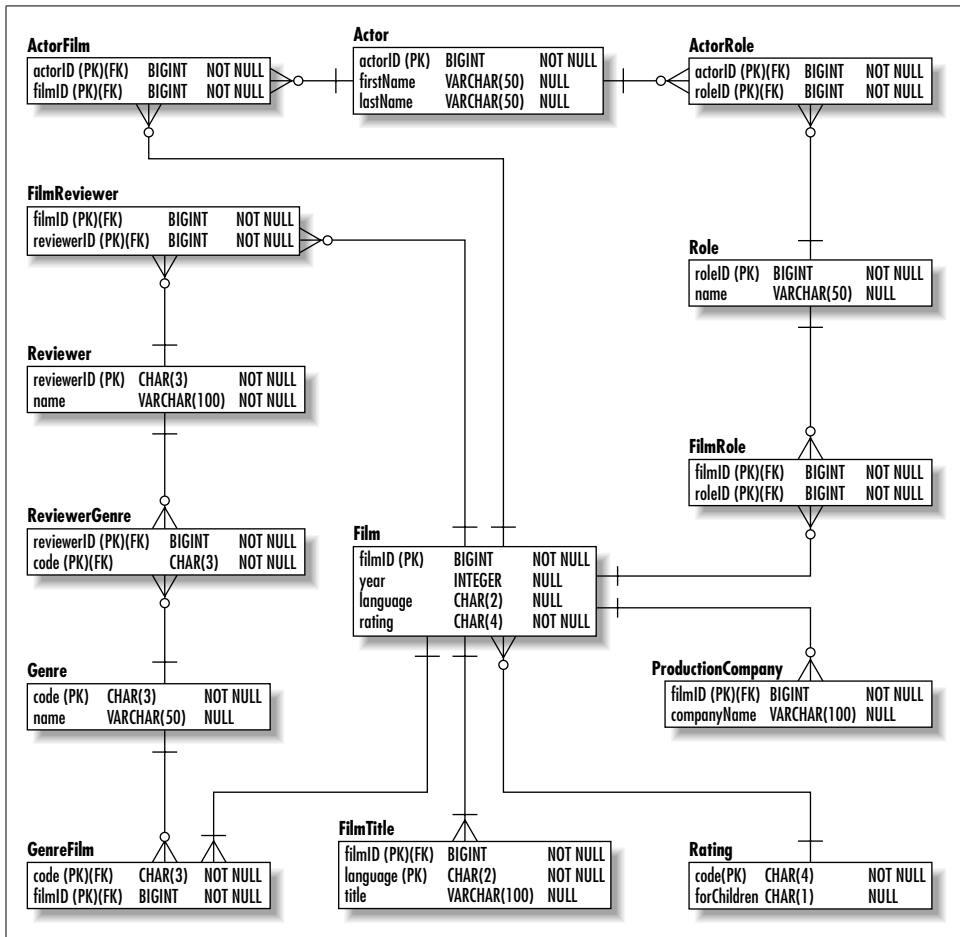


Figure 2-9. The film database in 4NF

Spending time normalizing to the fourth normal form is useful only for data models that have a lot of complex join tables. If you have just a few such tables, you are probably already in 4NF anyway. If you are not, the anomaly is almost certainly of no consequence to your system.

## Fifth normal form

A table is in the fifth normal form (5NF) if it is in 4NF and cannot have lossless decomposition into any number of smaller tables. It is actually very hard to tell when a table is truly in 5NF—just about any table in 4NF is also in 5NF. It can occur in situations in which you have a many-to-many-to-many relationship as exists with `Film`, `Actor`, and `Role`. Given certain data, the database can end up making claims that simply are not true.

For simplicity's sake, assume that the database has a single actor in it who has appeared in two separate films playing two separate roles. The joined information from these tables looks like the data in Table 2-7.

Table 2-7. Joining actor, film, and role

| actorID | filmID | roleName      | Description                                                                  |
|---------|--------|---------------|------------------------------------------------------------------------------|
| 1       | 101    | The president | Stanley Anderson (1) played the president in <i>Armageddon</i> (101).        |
| 1       | 102    | Edwin Sneller | Stanley Anderson (1) played Edwin Sneller in <i>The Pelican Brief</i> (102). |

So far, this structure should seem quite normal to you. The three entities have three corresponding join tables `ActorFilm`, `FilmRole`, and `ActorRole` to help manage the relationships. The problem arises when you insert particular data, such as adding Robert Culp who also played the president, but in the movie *The Pelican Brief*. In short, we add one row to `Actor`, one row to `ActorFilm`, one row to `ActorRow`, and one row to `FilmRole`. No rows are added to `Role` or `Film`. The join suddenly ends up with both true claims and some utterly false ones as Table 2-8 shows.

Table 2-8. The false claims (in italic) of a database not in 5NF

| actorID | filmID | roleName             | Description                                                                  |
|---------|--------|----------------------|------------------------------------------------------------------------------|
| 1       | 101    | The president        | Stanley Anderson (1) played the president in <i>Armageddon</i> (101).        |
| 1       | 102    | Edwin Sneller        | Stanley Anderson (1) played Edwin Sneller in <i>The Pelican Brief</i> (102). |
| 2       | 102    | The president        | Robert Culp (2) played the president in <i>The Pelican Brief</i> (102).      |
| 1       | 102    | <i>The president</i> | <i>Stanley Anderson (1) played the president in The Pelican Brief (102).</i> |
| 2       | 101    | <i>The president</i> | <i>Robert Culp (2) played the president in Armageddon (101).</i>             |

The important thing to note about the database is that there is nothing wrong with the data in the tables. The only thing wrong is what the relationships among the tables imply given a very specific data set.

By now, you have probably guessed that the solution is to create another entity to manage this trinary relationship. The `Appearance` table in the fully normalized Figure 2-10 manages this solution.

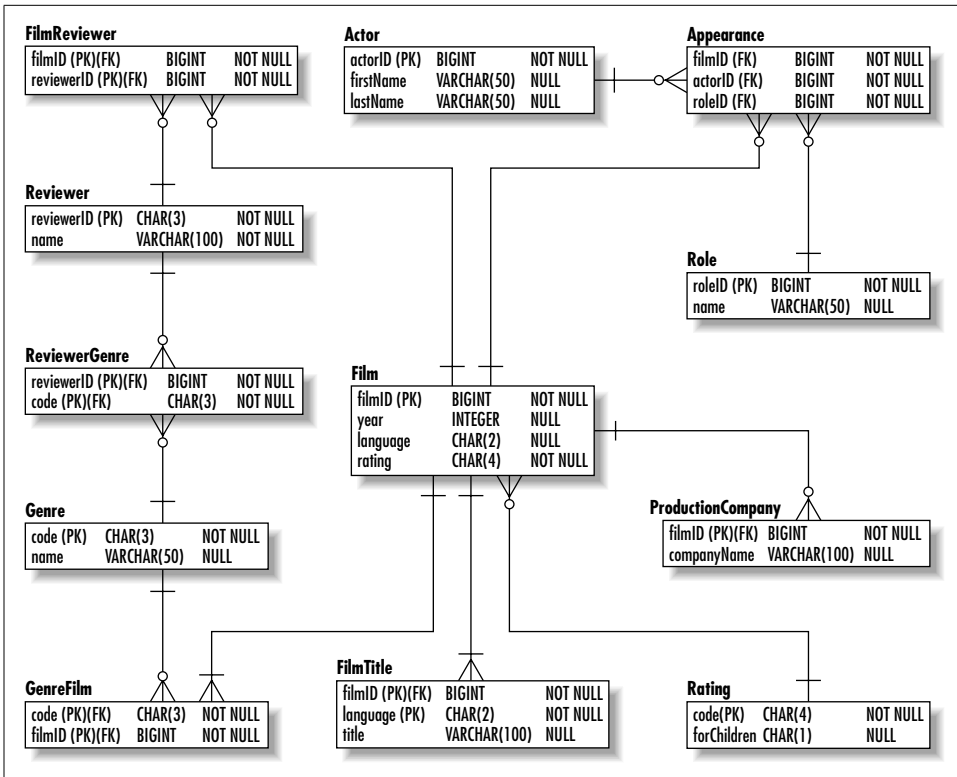


Figure 2-10. The film database in 5NF

## Denormalization

Denormalization is the process of consciously removing entities created through the normalization process. An unnormalized database is *not* a denormalized database. A database can be denormalized only after it has been sufficiently normalized, and solid justifications need to exist to support every act of denormalization.

Nevertheless, fully normalized databases can require complex programming and generally require more joins than their unnormalized or denormalized counterparts. Joins are resource-intensive operations; thus, the more joins, the more time a query will take.

To deal with queries that take too long or are too complex to be maintainable, a database architect denormalizes the database. As we have seen from the process of normalization, each lower normal form introduces database anomalies that can compromise the integrity, maintainability, and extensibility of the database. Denormalization is thus a reasoned trade-off between query complexity/performance and system integrity, maintenance, and extensibility.

## The Perils of Denormalization

Denormalization must be approached with caution. In general, a table should have proven it requires denormalization in testing or even in production before you actually denormalize it. Data architects very commonly denormalize based on hunches about performance or experience with similar applications in the past—a practice that leads down the path to a poorly designed database.

Denormalization can in some circumstances incur performance penalties. More important, however, most of the time you do not see the kinds of performance improvements from denormalization that actually make a difference. When you denormalize without concrete performance benchmarks backing the denormalization, you end up:

- Denormalizing tables without appreciable performance improvement
- Denormalizing again later, after you have done performance testing

The result is a database that looks more unnormalized than denormalized. The best rule of thumb is to prove the database needs denormalization and document that need for the people who will be maintaining the database. Subsequently, you should prove that your denormalization actually improves performance and back out the changes if they fail to address the performance concerns.

In most cases, you can deal with complexity simply by creating views that hide the complexity. Performance is thus the general driver of denormalization. To determine whether denormalization makes sense, I recommend Craig Mullins’s simple guidelines posted in an online article for *The Data Administration Newsletter* in an article called “Denormalization Guidelines” (<http://www.tdan.com/i001fe02.htm>):

- Can you achieve performance goals without denormalization?
- Will the system still fail to achieve performance goals with denormalization?
- Will the system be less reliable as a result of denormalization?

If you answer “yes” to any of these questions, you should not use denormalization as your performance tuning tool.

**BEST PRACTICE** Denormalize only when you have concrete proof that denormalization will boost performance.

The most common temptation to denormalize comes from queries that require joins to retrieve a single value. Any query pulling a film’s suitability for children along with the film from the database would fall into this category. For example:

```
SELECT Film.title, Film.language, Film.year, Rating.forChildren
FROM Film, Rating
WHERE Film.filmID = 2
AND Film.rating = Rating.code;
```

Denormalization would move the rating code and suitability for children back into the `Film` table. Wouldn't the query perform much better without that join? Actually, it probably would *not* perform *noticeably* better—the join is done using a unique index (`Rating.code`). Denormalization, however, would incur all of the anomalies that led us to normalize the table in the first place.

A better candidate for normalization might be pulling a state name into an `Address` table along with the state code used in the join. If most queries actually want the state name and the query would definitely benefit from avoiding the join to the `State` table, it can make sense to add an extra column to `Address` for `stateName`. You do not, however, remove the `State` table. This denormalization works—assuming real performance benefits are achieved for the application—because the state name is a candidate key for the `State` table. Though `stateName` would technically be a transitive dependency in the `Address` table (and thus violate 3NF), its status as a candidate key for `State` makes it almost a functional dependency and consequently almost acceptable to put into the `Address` table.

A common situation in which performance does truly become a problem is reporting. For reporting, database normalization is just one of many factors that lead to performance degradation. Because complex reports generally eat server resources regardless of normalization issues, it is generally a bad idea to empower users to execute complex reports against live tables. Instead, you can denormalize by replicating the data into special tables designed to support reporting needs. To create a table for reporting on westerns, we might create a `WesternReport` table that looks like the table in Table 2-9.

*Table 2-9. A table for reporting on westerns*

| Attribute   | Domain       | Notes       | NULL? |
|-------------|--------------|-------------|-------|
| filmID      | BIGINT       | PRIMARY KEY | No    |
| title       | VARCHAR(100) |             | No    |
| rating      | CHAR(5)      |             | Yes   |
| forChildren | CHAR(1)      | DEFAULT 'N' | No    |
| otherGenres | VARCHAR(255) |             | Yes   |
| directors   | VARCHAR(255) |             | Yes   |
| actors      | VARCHAR(255) |             | Yes   |
| ranking     | INT          |             | No    |
| year        | INT          |             | No    |

Reporting on all of the westerns from 1992 would look like this:

```
SELECT * FROM WesternReport
WHERE year = 1992;
```

The alternative is to have users constantly executing the following query against the tables that actually maintain your data:

```
SELECT Film.filmID, Film.title, Film.rating,
IFNULL(Rating.forChildren, 'N'), Film.ranking, Film.year
```

```
FROM Film, FilmGenre
LEFT OUTER JOIN Rating ON Film.rating = Rating.code

WHERE FilmGenre.code = 'WES'
AND year = 1992
AND Film.filmID = FilmGenre.filmID;
```

Use follow-up queries to get other genres, directors, and actors associated with the film.

## Object-Relational Mapping

You now have your data structured for optimal performance and extensibility in your database. To make use of that data, you need to pull it into applications—in our case, Java applications—that manipulate the data. Java is an object-oriented programming language. In other words, it models its problem domain using object-oriented principles. In general, object-oriented principles can be summed up as:

### *Encapsulation*

Encapsulation is the hiding of the data and behavior of a thing behind a limited and well-described interface. In Java terms, the limited and well-described interface is the set of your public methods and attributes.

### *Abstraction*

Abstraction is the modeling of only the essential characteristics of a thing and ignoring or hiding the details of its nonessential characteristics. A Java interface is an example of an abstraction.

### *Polymorphism*

Polymorphism means that a single interface can be used for a generic class of actions rather than a single specific action. The `equals()` method in `Object` is an example of a polymorphic interface because it means different specific things in different classes even though it generally means testing for equality.

### *Inheritance*

Inheritance is the ability for one thing to take on the behavior and characteristics of another. Java supports inheritances through extending classes.

Though a relational database is a model of a problem domain, it is a different kind of model. Your Java application models behavior and uses data to support that behavior. The database, however, models the data in your problem domain and its relationships. Java application logic is inefficient at determining what actors who have played the president during their career have appeared in films together. Similarly, a database is a poor tool for determining pricing rules for a set of products.

When a Java application needs to save its state to some sort of data storage, it is said to require persistence. Often, complex Java applications persist against a relational database. The use of a relational database for persistence has several advantages:

- Relational databases are efficient at storing data for later retrieval using complex criteria. You cannot search on your stored objects nearly as efficiently if they are serialized to a filesystem or stored somewhere in XML.

- Java’s JDBC API is simple to learn. Other persistence mechanisms tend to be much harder. Java’s file access APIs, for example, are painful to write cross-platform code with.
- Most people have easy access to a relational database. MySQL and PostgreSQL are freely available to those with limited budgets, and most organizations already have a huge investment in enterprise database engines like Oracle and DB2.

When you attempt to persist your Java objects to a relational database, however, you run into the problem of the object-relational mismatch. The most basic question facing the object-oriented developer using a relational database is how to map relational data into objects. Your immediate thought might be to simply map object attributes to entity attributes. Though this approach creates a solid starting point, it does not create the perfect mapping for two reasons:

- Unlike relational attributes, object attributes are multivalued. An object stores within itself attributes with multiple simple values as well as direct references to groups of complex objects.
- The relational model has no natural way of modeling inheritance relationships.

**BEST PRACTICE** Normalize data models based on object-relational mapping just as you would normalized any other data model: to 3NF or 4NF.

Figure 2-11 contains a sample class diagram for a system that should persist against a relational database. In the class diagram, you are modeling a person who can play many roles—including the roles of employee and customer. Each employee has many addresses and phone numbers.

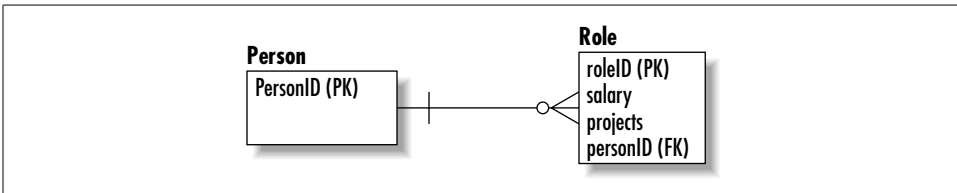


Figure 2-11. A simple class diagram for a persistent system

## Inheritance Mapping

The biggest difficulty in object-relational mapping arises in inheritance mapping—the mapping of class inheritance hierarchies into a relational database. In our class diagram, this problem appears in the structure related to roles. Do you need separate entities for Role, Employee, and Customer? Or does it make more sense to have Employee and Customer entities? Or perhaps just a Role entity will suffice?

Depending on the nature of your class diagram, all three solutions can work. Figure 2-12 shows three possible data models for supporting the class diagram from Figure 2-11.

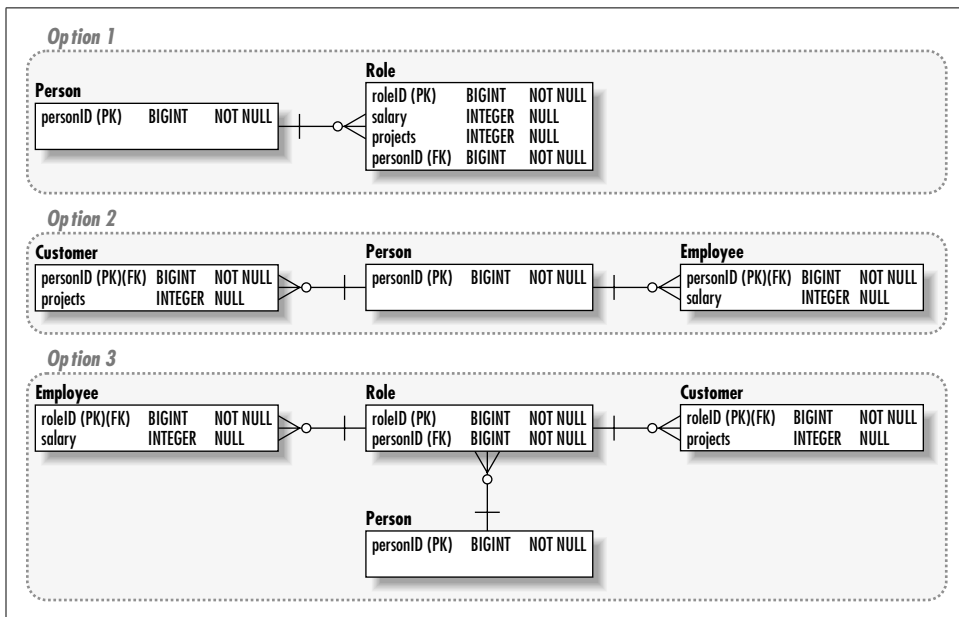


Figure 2-12. Three possible data models for roles

As you can see from all three data models, the relational model’s inability to support abstraction does some serious damage to the application. While the class diagram enables the addition of new roles to the system without impacting the Person class, you cannot accomplish the same flexibility in the data model. All three data models demand some understanding of all possible roles available.

The simplest data model—the one with only a Role entity—must contain all data associated with all possible roles. This approach to object-relational inheritance mapping when the implementation classes contain mostly the same data and the differences among the roles from a data perspective can be summarized through a role type.

On the other hand, if the commonalities in the different roles are not interesting from a data perspective, you can solve the problem with the second data model. In other words, model each role with its own entity and ignore the entire Role abstraction in the database. This approach fails if you need to ask questions like “What roles does this person have?” without necessarily knowing what possible roles can exist.

The third approach is a sometimes unwieldy compromise between the first and second approaches. When you need to deal with the roles both abstractly and concretely, you have to create a relationally artificial structure to model those nuances. In this data model, the entities for the concrete classes—Employee and Customer—

borrow their primary keys from the Role entity. Only the Role entity contains the foreign key of Person with whom the role is associated. You can now deal with the roles in an abstract sense by joining with the Role entity, or you can get specific information about individual roles through a complex join.

In short, the proper way of modeling an inheritance relationship depends heavily on what sort of queries you intend to use to retrieve that data. When in doubt, refer to this set of rules to help you in inheritance mapping:

*Model only the superclass*

- If your queries rely heavily on data associated with the superclass and
- If your queries do not rely on data associated with the subclasses and
- If the subclasses do not contain a significant number of distinct attributes

*Model only the subclasses*

- If your queries rely heavily on data associated with the subclasses and
- If your queries do not rely on data associated with the superclass

*Model the superclass and its subclasses*

For all other circumstances.

**BEST PRACTICE** When modeling OO (object-oriented) inheritance, consider whether your database will be used by non-OO systems. If it is being used by non-OO systems, focus more on the data model and less on mapping the OO concepts to the relational world.

## Multivalued Attributes

Collections such as arrays, HashMaps, and ArrayLists present another problem to object-relational mapping. In relational terms, these kinds of object attributes represent multivalued attributes. The solution to this problem starts with the same solution for multivalued attributes in your relational model: create entities to support the multivalued attributes.

That approach is simple enough for collections of objects like the Phone and Address classes in our class diagram. You create a Phone entity with a foreign key of personID and a primary key of personID and type and you are done. The mapping becomes tricky with attributes like int arrays or String collections.

**BEST PRACTICE** If your database engine supports SQL3 data types, map multivalued attributes to the SQL ARRAY type.

In our class diagram, we store a list of favorite colors as a String[ ]. The solution again is to handle this mapping in the same way you would handle the normalization of an entity with multivalued attributes: create a new entity. In this case, we would create a FavoriteColor table with personID and color as a primary key.