Chapter 2

The Relational Model of Data

This chapter introduces the most important model of data: the two-dimensional table, or "relation." We begin with an overview of data models in general. We give the basic terminology for relations and show how the model can be used to represent typical forms of data. We then introduce a portion of the language SQL — that part used to declare relations and their structure. The chapter closes with an introduction to relational algebra. We see how this notation serves as both a query language — the aspect of a data model that enables us to ask questions about the data — and as a constraint language — the aspect of a data model that lets us restrict the data in the database in various ways.

2.1 An Overview of Data Models

The notion of a "data model" is one of the most fundamental in the study of database systems. In this brief summary of the concept, we define some basic terminology and mention the most important data models.

2.1.1 What is a Data Model?

A *data model* is a notation for describing data or information. The description generally consists of three parts:

1. *Structure of the data.* You may be familiar with tools in programming languages such as C or Java for describing the structure of the data used by a program: arrays and structures ("structs") or objects, for example. The data structures used to implement data in the computer are sometimes referred to, in discussions of database systems, as a *physical data model,* although in fact they are far removed from the gates and electrons that truly serve as the physical implementation of the data. In the database

world, data models are at a somewhat higher level than data structures, and are sometimes referred to as a *conceptual model* to emphasize the difference in level. We shall see examples shortly.

2. *Operations on the data.* In programming languages, operations on the data are generally anything that can be programmed. In database data models, there is usually a limited set of operations that can be performed. We are generally allowed to perform a limited set of *queries* (operations that retrieve information) and *modifications* (operations that change the database). This limitation is not a weakness, but a strength. By limiting operations, it is possible for programmers to describe database operations at a very high level, yet have the database management system implement the operations efficiently. In comparison, it is generally impossible to optimize programs in conventional languages like C, to the extent that an inefficient algorithm (e.g., bubblesort) is replaced by a more efficient one (e.g., quicksort).

3. *Constraints on the data.* Database data models usually have a way to describe limitations on what the data can be. These constraints can range from the simple (e.g., "a day of the week is an integer between 1 and 7" or "a movie has at most one title") to some very complex limitations that we shall discuss in Sections 7.4 and 7.5.

2.1.2   Important Data Models

Today, the two data models of preeminent importance for database systems are:

1. The relational model, including object-relational extensions.

2. The semistructured-data model, including XML and related standards.

The first, which is present in all commercial database management systems, is the subject of this chapter. The semistructured model, of which XML is the primary manifestation, is an added feature of most relational DBMS's, and appears in a number of other contexts as well. We turn to this data model starting in Chapter 11.

2.1.3   The Relational Model in Brief

The relational model is based on tables, of which Fig. 2.1 is an example. We shall discuss this model beginning in Section 2.2. This relation, or table, describes movies: their title, the year in which they were made, their length in minutes, and the genre of the movie. We show three particular movies, but you should imagine that there are many more rows to this table — one row for each movie ever made, perhaps.

The structure portion of the relational model might appear to resemble an array of structs in C, where the column headers are the field names, and each

title	year	length	genre
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Figure 2.1: An example relation

of the rows represent the values of one struct in the array. However, it must be emphasized that this physical implementation is only one possible way the table could be implemented in physical data structures. In fact, it is not the normal way to represent relations, and a large portion of the study of database systems addresses the right ways to implement such tables. Much of the distinction comes from the scale of relations — they are not normally implemented as main-memory structures, and their proper physical implementation must take into account the need to access relations of very large size that are resident on disk.

The operations normally associated with the relational model form the "relational algebra," which we discuss beginning in Section 2.4. These operations are table-oriented. As an example, we can ask for all those rows of a relation that have a certain value in a certain column. For example, we can ask of the table in Fig. 2.1 for all the rows where the genre is "comedy."

The constraint portion of the relational data model will be touched upon briefly in Section 2.5 and covered in more detail in Chapter 7. However, as a brief sample of what kinds of constraints are generally used, we could decide that there is a fixed list of genres for movies, and that the last column of every row must have a value that is on this list. Or we might decide (incorrectly, it turns out) that there could never be two movies with the same title, and constrain the table so that no two rows could have the same string in the first component.

2.1.4 The Semistructured Model in Brief

Semistructured data resembles trees or graphs, rather than tables or arrays. The principal manifestation of this viewpoint today is XML, a way to represent data by hierarchically nested tagged elements. The tags, similar to those used in HTML, define the role played by different pieces of data, much as the column headers do in the relational model. For example, the same data as in Fig. 2.1 might appear in an XML "document" as in Fig. 2.2.

The operations on semistructured data usually involve following paths in the implied tree from an element to one or more of its nested subelements, then to subelements nested within those, and so on. For example, starting at the outer <Movies> element (the entire document in Fig. 2.2), we might move to each of its nested <Movie> elements, each delimited by the tag <Movie> and matching </Movie> tag, and from each <Movie> element to its nested <Genre>

```
<Movies>
    <Movie title="Gone With the Wind">
        <Year>1939</Year>
        <Length>231</Length>
        <Genre>drama</Genre>
    </Movie>
    <Movie title="Star Wars">
        <Year>1977</Year>
        <Length>124</Length>
        <Genre>sciFi</Genre>
    </Movie>
    <Movie title="Wayne's World">
        <Year>1992</Year>
        <Length>95</Length>
        <Genre>comedy</Genre>
    </Movie>
</Movies>
```

Figure 2.2: Movie data as XML

element, to see which movies belong to the "comedy" genre.

Constraints on the structure of data in this model often involve the data type of values associated with a tag. For instance, are the values associated with the `<Length>` tag integers or can they be arbitrary character strings? Other constraints determine which tags can appear nested within which other tags. For example, must each `<Movie>` element have a `<Length>` element nested within it? What other tags, besides those shown in Fig. 2.2 might be used within a `<Movie>` element? Can there be more than one genre for a movie? These and other matters will be taken up in Section 11.2.

2.1.5   Other Data Models

There are many other models that are, or have been, associated with DBMS's. A modern trend is to add object-oriented features to the relational model. There are two effects of object-orientation on relations:

1. Values can have structure, rather than being elementary types such as integer or strings, as they were in Fig. 2.1.

2. Relations can have associated methods.

In a sense, these extensions, called the *object-relational* model, are analogous to the way structs in C were extended to objects in C++. We shall introduce the object-relational model in Section 10.3.

There are even database models of the purely object-oriented kind. In these, the relation is no longer the principal data-structuring concept, but becomes only one option among many structures. We discuss an object-oriented database model in Section 4.9.

There are several other models that were used in some of the earlier DBMS's, but that have now fallen out of use. The *hierarchical model* was, like semistructured data, a tree-oriented model. Its drawback was that unlike more modern models, it really operated at the physical level, which made it impossible for programmers to write code at a conveniently high level. Another such model was the *network model*, which was a graph-oriented, physical-level model. In truth, both the hierarchical model and today's semistructured models, allow full graph structures, and do not limit us strictly to trees. However, the generality of graphs was built directly into the network model, rather than favoring trees as these other models do.

2.1.6 Comparison of Modeling Approaches

Even from our brief example, it appears that semistructured models have more flexibility than relations. This difference becomes even more apparent when we discuss, as we shall, how full graph structures are embedded into tree-like, semistructured models. Nevertheless, the relational model is still preferred in DBMS's, and we should understand why. A brief argument follows.

Because databases are large, efficiency of access to data and efficiency of modifications to that data are of great importance. Also very important is ease of use — the productivity of programmers who use the data. Surprisingly, both goals can be achieved with a model, particularly the relational model, that:

1. Provides a simple, limited approach to structuring data, yet is reasonably versatile, so anything can be modeled.

2. Provides a limited, yet useful, collection of operations on data.

Together, these limitations turn into features. They allow us to implement languages, such as SQL, that enable the programmer to express their wishes at a very high level. A few lines of SQL can do the work of thousands of lines of C, or hundreds of lines of the code that had to be written to access data under earlier models such as network or hierarchical. Yet the short SQL programs, because they use a strongly limited sets of operations, can be optimized to run as fast, or faster than the code written in alternative languages.

2.2 Basics of the Relational Model

The relational model gives us a single way to represent data: as a two-dimensional table called a *relation*. Figure 2.1, which we copy here as Fig. 2.3, is an example of a relation, which we shall call Movies. The rows each represent a

movie, and the columns each represent a property of movies. In this section, we shall introduce the most important terminology regarding relations, and illustrate them with the `Movies` relation.

title	*year*	*length*	*genre*
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Figure 2.3: The relation `Movies`

2.2.1 Attributes

The columns of a relation are named by *attributes*; in Fig. 2.3 the attributes are `title`, `year`, `length`, and `genre`. Attributes appear at the tops of the columns. Usually, an attribute describes the meaning of entries in the column below. For instance, the column with attribute `length` holds the length, in minutes, of each movie.

2.2.2 Schemas

The name of a relation and the set of attributes for a relation is called the *schema* for that relation. We show the schema for the relation with the relation name followed by a parenthesized list of its attributes. Thus, the schema for relation `Movies` of Fig. 2.3 is

$$\text{Movies(title, year, length, genre)}$$

The attributes in a relation schema are a set, not a list. However, in order to talk about relations we often must specify a "standard" order for the attributes. Thus, whenever we introduce a relation schema with a list of attributes, as above, we shall take this ordering to be the standard order whenever we display the relation or any of its rows.

In the relational model, a database consists of one or more relations. The set of schemas for the relations of a database is called a *relational database schema*, or just a *database schema*.

2.2.3 Tuples

The rows of a relation, other than the header row containing the attribute names, are called *tuples*. A tuple has one *component* for each attribute of the relation. For instance, the first of the three tuples in Fig. 2.3 has the four components Gone With the Wind, 1939, 231, and drama for attributes `title`, `year`, `length`, and `genre`, respectively. When we wish to write a tuple

---

Conventions for Relations and Attributes

We shall generally follow the convention that relation names begin with a capital letter, and attribute names begin with a lower-case letter. However, later in this book we shall talk of relations in the abstract, where the names of attributes do not matter. In that case, we shall use single capital letters for both relations and attributes, e.g., $R(A, B, C)$ for a generic relation with three attributes.

---

in isolation, not as part of a relation, we normally use commas to separate components, and we use parentheses to surround the tuple. For example,

$$(\text{Gone With the Wind, 1939, 231, drama})$$

is the first tuple of Fig. 2.3. Notice that when a tuple appears in isolation, the attributes do not appear, so some indication of the relation to which the tuple belongs must be given. We shall always use the order in which the attributes were listed in the relation schema.

2.2.4 Domains

The relational model requires that each component of each tuple be atomic; that is, it must be of some elementary type such as integer or string. It is not permitted for a value to be a record structure, set, list, array, or any other type that reasonably can have its values broken into smaller components.

It is further assumed that associated with each attribute of a relation is a *domain*, that is, a particular elementary type. The components of any tuple of the relation must have, in each component, a value that belongs to the domain of the corresponding column. For example, tuples of the Movies relation of Fig. 2.3 must have a first component that is a string, second and third components that are integers, and a fourth component whose value is a string.

It is possible to include the domain, or data type, for each attribute in a relation schema. We shall do so by appending a colon and a type after attributes. For example, we could represent the schema for the Movies relation as:

```
Movies(title:string, year:integer, length:integer, genre:string)
```

2.2.5 Equivalent Representations of a Relation

Relations are sets of tuples, not lists of tuples. Thus the order in which the tuples of a relation are presented is immaterial. For example, we can list the three tuples of Fig. 2.3 in any of their six possible orders, and the relation is "the same" as Fig. 2.3.

Moreover, we can reorder the attributes of the relation as we choose, without changing the relation. However, when we reorder the relation schema, we must be careful to remember that the attributes are column headers. Thus, when we change the order of the attributes, we also change the order of their columns. When the columns move, the components of tuples change their order as well. The result is that each tuple has its components permuted in the same way as the attributes are permuted.

For example, Fig. 2.4 shows one of the many relations that could be obtained from Fig. 2.3 by permuting rows and columns. These two relations are considered "the same." More precisely, these two tables are different presentations of the same relation.

year	*genre*	*title*	*length*
1977	sciFi	Star Wars	124
1992	comedy	Wayne's World	95
1939	drama	Gone With the Wind	231

Figure 2.4: Another presentation of the relation `Movies`

2.2.6   Relation Instances

A relation about movies is not static; rather, relations change over time. We expect to insert tuples for new movies, as these appear. We also expect changes to existing tuples if we get revised or corrected information about a movie, and perhaps deletion of tuples for movies that are expelled from the database for some reason.

It is less common for the schema of a relation to change. However, there are situations where we might want to add or delete attributes. Schema changes, while possible in commercial database systems, can be very expensive, because each of perhaps millions of tuples needs to be rewritten to add or delete components. Also, if we add an attribute, it may be difficult or even impossible to generate appropriate values for the new component in the existing tuples.

We shall call a set of tuples for a given relation an *instance* of that relation. For example, the three tuples shown in Fig. 2.3 form an instance of relation `Movies`. Presumably, the relation `Movies` has changed over time and will continue to change over time. For instance, in 1990, `Movies` did not contain the tuple for `Wayne's World`. However, a conventional database system maintains only one version of any relation: the set of tuples that are in the relation "now." This instance of the relation is called the *current instance.*[1]

---

[1]Databases that maintain historical versions of data as it existed in past times are called *temporal databases.*

2.2.7 Keys of Relations

There are many constraints on relations that the relational model allows us to place on database schemas. We shall defer much of the discussion of constraints until Chapter 7. However, one kind of constraint is so fundamental that we shall introduce it here: *key* constraints. A set of attributes forms a *key* for a relation if we do not allow two tuples in a relation instance to have the same values in all the attributes of the key.

Example 2.1: We can declare that the relation `Movies` has a key consisting of the two attributes `title` and `year`. That is, we don't believe there could ever be two movies that had both the same title and the same year. Notice that `title` by itself does not form a key, since sometimes "remakes" of a movie appear. For example, there are three movies named *King Kong*, each made in a different year. It should also be obvious that `year` by itself is not a key, since there are usually many movies made in the same year. □

We indicate the attribute or attributes that form a key for a relation by underlining the key attribute(s). For instance, the `Movies` relation could have its schema written as:

<p align="center">Movies(<u>title</u>, <u>year</u>, length, genre)</p>

Remember that the statement that a set of attributes forms a key for a relation is a statement about all possible instances of the relation, not a statement about a single instance. For example, looking only at the tiny relation of Fig. 2.3, we might imagine that `genre` by itself forms a key, since we do not see two tuples that agree on the value of their `genre` components. However, we can easily imagine that if the relation instance contained more movies, there would be many dramas, many comedies, and so on. Thus, there would be distinct tuples that agreed on the `genre` component. As a consequence, it would be incorrect to assert that `genre` is a key for the relation `Movies`.

While we might be sure that `title` and `year` can serve as a key for `Movies`, many real-world databases use artificial keys, doubting that it is safe to make any assumption about the values of attributes outside their control. For example, companies generally assign employee ID's to all employees, and these ID's are carefully chosen to be unique numbers. One purpose of these ID's is to make sure that in the company database each employee can be distinguished from all others, even if there are several employees with the same name. Thus, the employee-ID attribute can serve as a key for a relation about employees.

In US corporations, it is normal for every employee to have a Social-Security number. If the database has an attribute that is the Social-Security number, then this attribute can also serve as a key for employees. Note that there is nothing wrong with there being several choices of key, as there would be for employees having both employee ID's and Social-Security numbers.

The idea of creating an attribute whose purpose is to serve as a key is quite widespread. In addition to employee ID's, we find student ID's to distinguish

students in a university. We find drivers' license numbers and automobile registration numbers to distinguish drivers and automobiles, respectively. You undoubtedly can find more examples of attributes created for the primary purpose of serving as keys.

```
Movies(
    title:string,
    year:integer,
    length:integer,
    genre:string,
    studioName:string,
    producerC#:integer
)
MovieStar(
    name:string,
    address:string,
    gender:char,
    birthdate:date
)
StarsIn(
    movieTitle:string,
    movieYear:integer,
    starName:string
)
MovieExec(
    name:string,
    address:string,
    cert#:integer,
    netWorth:integer
)
Studio(
    name:string,
    address:string,
    presC#:integer
)
```

Figure 2.5: Example database schema about movies

2.2.8   An Example Database Schema

We shall close this section with an example of a complete database schema. The topic is movies, and it builds on the relation Movies that has appeared so far in examples. The database schema is shown in Fig. 2.5. Here are the things we need to know to understand the intention of this schema.

Movies

This relation is an extension of the example relation we have been discussing so far. Remember that its key is `title` and `year` together. We have added two new attributes; `studioName` tells us the studio that owns the movie, and `producerC#` is an integer that represents the producer of the movie in a way that we shall discuss when we talk about the relation `MovieExec` below.

MovieStar

This relation tells us something about stars. The key is `name`, the name of the movie star. It is not usual to assume names of persons are unique and therefore suitable as a key. However, movie stars are different; one would never take a name that some other movie star had used. Thus, we shall use the convenient fiction that movie-star names are unique. A more conventional approach would be to invent a serial number of some sort, like social-security numbers, so that we could assign each individual a unique number and use that attribute as the key. We take that approach for movie executives, as we shall see. Another interesting point about the `MovieStar` relation is that we see two new data types. The gender can be a single character, M or F. Also, birthdate is of type "date," which might be a character string of a special form.

StarsIn

This relation connects movies to the stars of that movie, and likewise connects a star to the movies in which they appeared. Notice that movies are represented by the key for `Movies` — the title and year — although we have chosen different attribute names to emphasize that attributes `movieTitle` and `movieYear` represent the movie. Likewise, stars are represented by the key for `MovieStar`, with the attribute called `starName`. Finally, notice that all three attributes are necessary to form a key. It is perfectly reasonable to suppose that relation `StarsIn` could have two distinct tuples that agree in any two of the three attributes. For instance, a star might appear in two movies in one year, giving rise to two tuples that agreed in `movieYear` and `starName`, but disagreed in `movieTitle`.

MovieExec

This relation tells us about movie executives. It contains their name, address, and networth as data about the executive. However, for a key we have invented "certificate numbers" for all movie executives, including producers (as appear in the relation `Movies`) and studio presidents (as appear in the relation `Studio`, below). These are integers; a different one is assigned to each executive.

acctNo	type	balance
12345	savings	12000
23456	checking	1000
34567	savings	25

The relation Accounts

firstName	lastName	idNo	account
Robbie	Banks	901-222	12345
Lena	Hand	805-333	12345
Lena	Hand	805-333	23456

The relation Customers

Figure 2.6: Two relations of a banking database

Studio

This relation tells about movie studios. We rely on no two studios having the same name, and therefore use name as the key. The other attributes are the address of the studio and the certificate number for the president of the studio. We assume that the studio president is surely a movie executive and therefore appears in MovieExec.

2.2.9   Exercises for Section 2.2

Exercise 2.2.1: In Fig. 2.6 are instances of two relations that might constitute part of a banking database. Indicate the following:

a) The attributes of each relation.

b) The tuples of each relation.

c) The components of one tuple from each relation.

d) The relation schema for each relation.

e) The database schema.

f) A suitable domain for each attribute.

g) Another equivalent way to present each relation.

Exercise 2.2.2: In Section 2.2.7 we suggested that there are many examples of attributes that are created for the purpose of serving as keys of relations. Give some additional examples.

!! **Exercise 2.2.3:** How many different ways (considering orders of tuples and attributes) are there to represent a relation instance if that instance has:

a) Three attributes and three tuples, like the relation Accounts of Fig. 2.6?

b) Four attributes and five tuples?

c) n attributes and m tuples?

2.3 Defining a Relation Schema in SQL

SQL (pronounced "sequel") is the principal language used to describe and manipulate relational databases. There is a current standard for SQL, called SQL-99. Most commercial database management systems implement something similar, but not identical to, the standard. There are two aspects to SQL:

1. The *Data-Definition* sublanguage for declaring database schemas and

2. The *Data-Manipulation* sublanguage for *querying* (asking questions about) databases and for modifying the database.

The distinction between these two sublanguages is found in most languages; e.g., C or Java have portions that declare data and other portions that are executable code. These correspond to data-definition and data-manipulation, respectively.

In this section we shall begin a discussion of the data-definition portion of SQL. There is more on the subject in Chapter 7, especially the matter of constraints on data. The data-manipulation portion is covered extensively in Chapter 6.

2.3.1 Relations in SQL

SQL makes a distinction between three kinds of relations:

1. Stored relations, which are called *tables*. These are the kind of relation we deal with ordinarily — a relation that exists in the database and that can be modified by changing its tuples, as well as queried.

2. *Views*, which are relations defined by a computation. These relations are not stored, but are constructed, in whole or in part, when needed. They are the subject of Section 8.1.

3. Temporary tables, which are constructed by the SQL language processor when it performs its job of executing queries and data modifications. These relations are then thrown away and not stored.

In this section, we shall learn how to declare tables. We do not treat the declaration and definition of views here, and temporary tables are never declared. The SQL CREATE TABLE statement declares the schema for a stored relation. It gives a name for the table, its attributes, and their data types. It also allows us to declare a key, or even several keys, for a relation. There are many other features to the CREATE TABLE statement, including many forms of constraints that can be declared, and the declaration of *indexes* (data structures that speed up many operations on the table) but we shall leave those for the appropriate time.

2.3.2 Data Types

To begin, let us introduce the primitive data types that are supported by SQL systems. All attributes must have a data type.

1. Character strings of fixed or varying length. The type CHAR(n) denotes a fixed-length string of up to n characters. VARCHAR(n) also denotes a string of up to n characters. The difference is implementation-dependent; typically CHAR implies that short strings are padded to make n characters, while VARCHAR implies that an endmarker or string-length is used. SQL permits reasonable coercions between values of character-string types. Normally, a string is padded by trailing blanks if it becomes the value of a component that is a fixed-length string of greater length. For example, the string 'foo',[2] if it became the value of a component for an attribute of type CHAR(5), would assume the value 'foo ' (with two blanks following the second o).

2. Bit strings of fixed or varying length. These strings are analogous to fixed and varying-length character strings, but their values are strings of bits rather than characters. The type BIT(n) denotes bit strings of length n, while BIT VARYING(n) denotes bit strings of length up to n.

3. The type BOOLEAN denotes an attribute whose value is logical. The possible values of such an attribute are TRUE, FALSE, and — although it would surprise George Boole — UNKNOWN.

4. The type INT or INTEGER (these names are synonyms) denotes typical integer values. The type SHORTINT also denotes integers, but the number of bits permitted may be less, depending on the implementation (as with the types int and short int in C).

---

[2] Notice that in SQL, strings are surrounded by single-quotes, not double-quotes as in many other programming languages.

Dates and Times in SQL

Different SQL implementations may provide many different representations for dates and times, but the following is the SQL standard representation. A date value is the keyword DATE followed by a quoted string of a special form. For example, DATE '1948-05-14' follows the required form. The first four characters are digits representing the year. Then come a hyphen and two digits representing the month. Finally there is another hyphen and two digits representing the day. Note that single-digit months and days are padded with a leading 0.

A time value is the keyword TIME and a quoted string. This string has two digits for the hour, on the military (24-hour) clock. Then come a colon, two digits for the minute, another colon, and two digits for the second. If fractions of a second are desired, we may continue with a decimal point and as many significant digits as we like. For instance, TIME '15:00:02.5' represents the time at which all students will have left a class that ends at 3 PM: two and a half seconds past three o'clock.

---

5. Floating-point numbers can be represented in a variety of ways. We may use the type FLOAT or REAL (these are synonyms) for typical floating-point numbers. A higher precision can be obtained with the type DOUBLE PRECISION; again the distinction between these types is as in C. SQL also has types that are real numbers with a fixed decimal point. For example, DECIMAL(n,d) allows values that consist of n decimal digits, with the decimal point assumed to be d positions from the right. Thus, 0123.45 is a possible value of type DECIMAL(6,2). NUMERIC is almost a synonym for DECIMAL, although there are possible implementation-dependent differences.

6. Dates and times can be represented by the data types DATE and TIME, respectively (see the box on "Dates and Times in SQL"). These values are essentially character strings of a special form. We may, in fact, coerce dates and times to string types, and we may do the reverse if the string "makes sense" as a date or time.

2.3.3 Simple Table Declarations

The simplest form of declaration of a relation schema consists of the keywords CREATE TABLE followed by the name of the relation and a parenthesized, comma-separated list of the attribute names and their types.

Example 2.2: The relation Movies with the schema given in Fig. 2.5 can be declared as in Fig. 2.7. The title is declared as a string of (up to) 100 characters.

```
CREATE TABLE Movies (
    title       CHAR(100),
    year        INT,
    length      INT,
    genre       CHAR(10),
    studioName  CHAR(30),
    producerC#  INT
);
```

Figure 2.7: SQL declaration of the table `Movies`

The year and length attributes are each integers, and the genre is a string of (up to) 10 characters. The decision to allow up to 100 characters for a title is arbitrary, but we don't want to limit the lengths of titles too strongly, or long titles would be truncated to fit. We have assumed that 10 characters are enough to represent a genre of movie; again, that is an arbitrary choice, one we could regret if we had a genre with a long name. Likewise, we have chosen 30 characters as sufficient for the studio name. The certificate number for the producer of the movie is another integer.   □

Example 2.3: Figure 2.8 is a SQL declaration of the relation `MovieStar` from Fig. 2.5. It illustrates some new options for data types. The name of this table is `MovieStar`, and it has four attributes. The first two attributes, `name` and `address`, have each been declared to be character strings. However, with the name, we have made the decision to use a fixed-length string of 30 characters, padding a name out with blanks at the end if necessary and truncating a name to 30 characters if it is longer. In contrast, we have declared addresses to be variable-length character strings of up to 255 characters.[3] It is not clear that these two choices are the best possible, but we use them to illustrate the two major kinds of string data types.

```
CREATE TABLE MovieStar (
    name     CHAR(30),
    address  VARCHAR(255),
    gender   CHAR(1),
    birthdate DATE
);
```

Figure 2.8: Declaring the relation schema for the `MovieStar` relation

---

[3]The number 255 is not the result of some weird notion of what typical addresses look like. A single byte can store integers between 0 and 255, so it is possible to represent a varying-length character string of up to 255 bytes by a single byte for the count of characters plus the bytes to store the string itself. Commercial systems generally support longer varying-length strings, however.

The `gender` attribute has values that are a single letter, M or F. Thus, we can safely use a single character as the type of this attribute. Finally, the `birthdate` attribute naturally deserves the data type `DATE`. □

2.3.4 Modifying Relation Schemas

We now know how to declare a table. But what if we need to change the schema of the table after it has been in use for a long time and has many tuples in its current instance? We can remove the entire table, including all of its current tuples, or we could change the schema by adding or deleting attributes.

We can delete a relation R by the SQL statement:

```
DROP TABLE R;
```

Relation R is no longer part of the database schema, and we can no longer access any of its tuples.

More frequently than we would drop a relation that is part of a long-lived database, we may need to modify the schema of an existing relation. These modifications are done by a statement that begins with the keywords `ALTER TABLE` and the name of the relation. We then have several options, the most important of which are

1. `ADD` followed by an attribute name and its data type.

2. `DROP` followed by an attribute name.

Example 2.4: Thus, for instance, we could modify the `MovieStar` relation by adding an attribute `phone` with:

```
ALTER TABLE MovieStar ADD phone CHAR(16);
```

As a result, the `MovieStar` schema now has five attributes: the four mentioned in Fig. 2.8 and the attribute `phone`, which is a fixed-length string of 16 bytes. In the actual relation, tuples would all have components for `phone`, but we know of no phone numbers to put there. Thus, the value of each of these components is set to the special *null value*, `NULL`. In Section 2.3.5, we shall see how it is possible to choose another "default" value to be used instead of `NULL` for unknown values.

As another example, the `ALTER TABLE` statement:

```
ALTER TABLE MovieStar DROP birthdate;
```

deletes the `birthdate` attribute. As a result, the schema for `MovieStar` no longer has that attribute, and all tuples of the current `MovieStar` instance have the component for `birthdate` deleted. □

2.3.5   Default Values

When we create or modify tuples, we sometimes do not have values for all components. For instance, we mentioned in Example 2.4 that when we add a column to a relation schema, the existing tuples do not have a known value, and it was suggested that NULL could be used in place of a "real" value. However, there are times when we would prefer to use another choice of *default* value, the value that appears in a column if no other value is known.

In general, any place we declare an attribute and its data type, we may add the keyword DEFAULT and an appropriate value. That value is either NULL or a constant. Certain other values that are provided by the system, such as the current time, may also be options.

Example 2.5: Let us consider Example 2.3. We might wish to use the character ? as the default for an unknown gender, and we might also wish to use the earliest possible date, DATE '0000-00-00' for an unknown birthdate. We could replace the declarations of gender and birthdate in Fig. 2.8 by:

```
gender CHAR(1) DEFAULT '?',
birthdate DATE DEFAULT DATE '0000-00-00'
```

As another example, we could have declared the default value for new attribute phone to be 'unlisted' when we added this attribute in Example 2.4. In that case,

```
ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT 'unlisted';
```

would be the appropriate ALTER TABLE statement.   □

2.3.6   Declaring Keys

There are two ways to declare an attribute or set of attributes to be a key in the CREATE TABLE statement that defines a stored relation.

1. We may declare one attribute to be a key when that attribute is listed in the relation schema.

2. We may add to the list of items declared in the schema (which so far have only been attributes) an additional declaration that says a particular attribute or set of attributes forms the key.

If the key consists of more than one attribute, we have to use method (2). If the key is a single attribute, either method may be used.

There are two declarations that may be used to indicate keyness:

a) PRIMARY KEY, or

b) UNIQUE.

The effect of declaring a set of attributes S to be a key for relation R either using PRIMARY KEY or UNIQUE is the following:

- Two tuples in R cannot agree on all of the attributes in set S, unless one of them is NULL. Any attempt to insert or update a tuple that violates this rule causes the DBMS to reject the action that caused the violation.

In addition, if PRIMARY KEY is used, then attributes in S are not allowed to have NULL as a value for their components. Again, any attempt to violate this rule is rejected by the system. NULL is permitted if the set S is declared UNIQUE, however. A DBMS may make other distinctions between the two terms, if it wishes.

Example 2.6: Let us reconsider the schema for relation MovieStar. Since no star would use the name of another star, we shall assume that name by itself forms a key for this relation. Thus, we can add this fact to the line declaring name. Figure 2.9 is a revision of Fig. 2.8 that reflects this change. We could also substitute UNIQUE for PRIMARY KEY in this declaration. If we did so, then two or more tuples could have NULL as the value of name, but there could be no other duplicate values for this attribute.

```
CREATE TABLE MovieStar (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE
);
```

Figure 2.9: Making name the key

Alternatively, we can use a separate definition of the key. The resulting schema declaration would look like Fig. 2.10. Again, UNIQUE could replace PRIMARY KEY. □

```
CREATE TABLE MovieStar (
    name CHAR(30),
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE,
    PRIMARY KEY (name)
);
```

Figure 2.10: A separate declaration of the key

Example 2.7: In Example 2.6, the form of either Fig. 2.9 or Fig. 2.10 is acceptable, because the key is a single attribute. However, in a situation where the key has more than one attribute, we must use the style of Fig. 2.10. For instance, the relation Movie, whose key is the pair of attributes title and year, must be declared as in Fig. 2.11. However, as usual, UNIQUE is an option to replace PRIMARY KEY. □

```
CREATE TABLE Movies (
    title       CHAR(100),
    year        INT,
    length      INT,
    genre       CHAR(10),
    studioName  CHAR(30),
    producerC#  INT,
    PRIMARY KEY (title, year)
);
```

Figure 2.11: Making title and year be the key of Movies

2.3.7   Exercises for Section 2.3

Exercise 2.3.1: In this exercise we introduce one of our running examples of a relational database schema. The database schema consists of four relations, whose schemas are:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

The Product relation gives the manufacturer, model number and type (PC, laptop, or printer) of various products. We assume for convenience that model numbers are unique over all manufacturers and product types; that assumption is not realistic, and a real database would include a code for the manufacturer as part of the model number. The PC relation gives for each model number that is a PC the speed (of the processor, in gigahertz), the amount of RAM (in megabytes), the size of the hard disk (in gigabytes), and the price. The Laptop relation is similar, except that the screen size (in inches) is also included. The Printer relation records for each printer model whether the printer produces color output (true, if so), the process type (laser or ink-jet, typically), and the price.

    Write the following declarations:

  a) A suitable schema for relation Product.