Garcia-Molina, Hector et al

Chapter 2. The Relational Model of Data

ISBN: 0131354280

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

```
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,

(`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:

`Movies(title, year, length, genre)`

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.

acct	*type*	*balance*
12345	checking	10000
23456	checking	2000
34567	savings	5

The relation Accounts

firstName	*lastName*	*id*	*account*
Robbie	Banks	901-333	12345
Lena	Hand	805-222	12345
Lena	Hand	805-222	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 make 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`.

b) A suitable schema for relation `Laptop`.

c) A suitable schema for relation `Printer`.

d) A suitable schema for relation `PC`.

e) An alteration to your `Printer` schema from (d) to delete the attribute `color`.

f) An alteration to your `Laptop` schema from (c) to add the attribute `od` (optical-disk type, e.g., cd or dvd). Let the default value for this attribute be `'none'` if the laptop does not have an optical disk.

Exercise 2.3.2: This exercise introduces another running example, concerning World War II capital ships. It involves the following relations:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

Ships are built in "classes" from the same design, and the class is usually named for the first ship of that class. The relation `Classes` records the name of the class, the type (`'bb'` for battleship or `'bc'` for battlecruiser), the country that built the ship, the number of main guns, the bore (diameter of the gun barrel, in inches) of the main guns, and the displacement (weight, in tons). Relation `Ships` records the name of the ship, the name of its class, and the year in which the ship was launched. Relation `Battles` gives the name and date of battles involving these ships, and relation `Outcomes` gives the result (sunk, damaged, or ok) for each ship in each battle.

Write the following declarations:

a) A suitable schema for relation `Ships`.

b) A suitable schema for relation `Outcomes`.

c) A suitable schema for relation `Classes`.

d) A suitable schema for relation `Battles`.

e) An alteration to your `Classes` relation from (a) to delete the attribute `bore`.

f) An alteration to your `Ships` relation from (b) to include the attribute `yard` giving the shipyard where the ship was built.

2.4 An Algebraic Query Language

In this section, we introduce the data-manipulation aspect of the relational model. Recall that a data model is not just structure; it needs a way to query the data and to modify the data. To begin our study of operations on relations, we shall learn about a special algebra, called *relational algebra*, that consists of some simple but powerful ways to construct new relations from given relations. When the given relations are stored data, then the constructed relations can be answers to queries about this data.

Relational algebra is not used today as a query language in commercial DBMS's, although some of the early prototypes did use this algebra directly. Rather, the "real" query language, SQL, incorporates relational algebra at its center, and many SQL programs are really "syntactically sugared" expressions of relational algebra. Further, when a DBMS processes queries, the first thing that happens to a SQL query is that it gets translated into relational algebra or a very similar internal representation. Thus, there are several good reasons to start out learning this algebra.

2.4.1 Why Do We Need a Special Query Language?

Before introducing the operations of relational algebra, one should ask why, or whether, we need a new kind of programming languages for databases. Won't conventional languages like C or Java suffice to ask and answer any computable question about relations? After all, we can represent a tuple of a relation by a struct (in C) or an object (in Java), and we can represent relations by arrays of these elements.

The surprising answer is that relational algebra is useful because it is *less* powerful than C or Java. That is, there are computations one can perform in any conventional language that one cannot perform in relational algebra. An example is: determine whether the number of tuples in a relation is even or odd. By limiting what we can say or do in our query language, we get two huge rewards — ease of programming and the ability of the compiler to produce highly optimized code — that we discussed in Section 2.1.6.

2.4.2 What is an Algebra?

An algebra, in general, consists of operators and atomic operands. For instance, in the algebra of arithmetic, the atomic operands are variables like x and constants like 15. The operators are the usual arithmetic ones: addition, subtraction, multiplication, and division. Any algebra allows us to build *expressions* by applying operators to atomic operands and/or other expressions of the algebra. Usually, parentheses are needed to group operators and their operands. For instance, in arithmetic we have expressions such as $(x+y)*z$ or $\big((x+7)/(y-3)\big)+x$.

Relational algebra is another example of an algebra. Its atomic operands are:

1. Variables that stand for relations.
2. Constants, which are finite relations.

We shall next see the operators of relational algebra.

2.4.3 Overview of Relational Algebra

The operations of the traditional relational algebra fall into four broad classes:

a) The usual set operations — union, intersection, and difference — applied to relations.

b) Operations that remove parts of a relation: "selection" eliminates some rows (tuples), and "projection" eliminates some columns.

c) Operations that combine the tuples of two relations, including "Cartesian product," which pairs the tuples of two relations in all possible ways, and various kinds of "join" operations, which selectively pair tuples from two relations.

d) An operation called "renaming" that does not affect the tuples of a relation, but changes the relation schema, i.e., the names of the attributes and/or the name of the relation itself.

We generally shall refer to expressions of relational algebra as *queries*.

2.4.4 Set Operations on Relations

The three most common operations on sets are union, intersection, and difference. We assume the reader is familiar with these operations, which are defined as follows on arbitrary sets R and S:

- $R \cup S$, the *union* of R and S, is the set of elements that are in R or S or both. An element appears only once in the union even if it is present in both R and S.
- $R \cap S$, the *intersection* of R and S, is the set of elements that are in both R and S.
- $R - S$, the *difference* of R and S, is the set of elements that are in R but not in S. Note that $R - S$ is different from $S - R$; the latter is the set of elements that are in S but not in R.

When we apply these operations to relations, we need to put some conditions on R and S:

1. R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be the same in R and S.

2. Before we compute the set-theoretic union, intersection, or difference of sets of tuples, the columns of R and S must be ordered so that the order of attributes is the same for both relations.

Sometimes we would like to take the union, intersection, or difference of relations that have the same number of attributes, with corresponding domains, but that use different names for their attributes. If so, we may use the renaming operator to be discussed in Section 2.4.11 to change the schema of one or both relations and give them the same set of attributes.

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Relation R

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Relation S

Figure 2.12: Two relations

Example 2.8: Suppose we have the two relations R and S, whose schemas are both that of relation MovieStar Section 2.2.8. Current instances of R and S are shown in Fig. 2.12. Then the union $R \cup S$ is

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Note that the two tuples for Carrie Fisher from the two relations appear only once in the result.

The intersection $R \cap S$ is

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

Now, only the Carrie Fisher tuple appears, because only it is in both relations.

The difference $R - S$ is

name	*address*	*gender*	*birthdate*
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

That is, the Fisher and Hamill tuples appear in R and thus are candidates for $R - S$. However, the Fisher tuple also appears in S and so is not in $R - S$. □

2.4.5 Projection

The *projection* operator is used to produce from a relation R a new relation that has only some of R's columns. The value of expression $\pi_{A_1,A_2,\ldots,A_n}(R)$ is a relation that has only the columns for attributes $A_1, A_2, \ldots, A_n$ of R. The schema for the resulting value is the set of attributes $\{A_1, A_2, \ldots, A_n\}$, which we conventionally show in the order listed.

title	*year*	*length*	*genre*	*studioName*	*producerC#*
Star Wars	1977	124	sciFi	Fox	12345
Galaxy Quest	1999	104	comedy	DreamWorks	67890
Wayne's World	1992	95	comedy	Paramount	99999

Figure 2.13: The relation Movies

Example 2.9: Consider the relation Movies with the relation schema described in Section 2.2.8. An instance of this relation is shown in Fig. 2.13. We can project this relation onto the first three attributes with the expression:

$$\pi_{title,year,length}(\texttt{Movies})$$

The resulting relation is

title	*year*	*length*
Star Wars	1977	124
Galaxy Quest	1999	104
Wayne's World	1992	95

As another example, we can project onto the attribute **genre** with the expression $\pi_{genre}(\texttt{Movies})$. The result is the single-column relation

genre
sciFi
comedy

Notice that there are only two tuples in the resulting relation, since the last two tuples of Fig. 2.13 have the same value in their component for attribute **genre**, and in the relational algebra of sets, duplicate tuples are always eliminated. □

A Note About Data Quality :-)

While we have endeavored to make example data as accurate as possible, we have used bogus values for addresses and other personal information about movie stars, in order to protect the privacy of members of the acting profession, many of whom are shy individuals who shun publicity.

2.4.6 Selection

The *selection* operator, applied to a relation R, produces a new relation with a subset of R's tuples. The tuples in the resulting relation are those that satisfy some condition C that involves the attributes of R. We denote this operation $\sigma_C(R)$. The schema for the resulting relation is the same as R's schema, and we conventionally show the attributes in the same order as we use for R.

C is a conditional expression of the type with which we are familiar from conventional programming languages; for example, conditional expressions follow the keyword `if` in programming languages such as C or Java. The only difference is that the operands in condition C are either constants or attributes of R. We apply C to each tuple t of R by substituting, for each attribute A appearing in condition C, the component of t for attribute A. If after substituting for each attribute of C the condition C is true, then t is one of the tuples that appear in the result of $\sigma_C(R)$; otherwise t is not in the result.

Example 2.10: Let the relation `Movies` be as in Fig. 2.13. Then the value of expression $\sigma_{length \geq 100}(\texttt{Movies})$ is

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	sciFi	Fox	12345
Galaxy Quest	1999	104	comedy	DreamWorks	67890

The first tuple satisfies the condition $length \geq 100$ because when we substitute for *length* the value 124 found in the component of the first tuple for attribute `length`, the condition becomes $124 \geq 100$. The latter condition is true, so we accept the first tuple. The same argument explains why the second tuple of Fig. 2.13 is in the result.

The third tuple has a `length` component 95. Thus, when we substitute for *length* we get the condition $95 \geq 100$, which is false. Hence the last tuple of Fig. 2.13 is not in the result. □

Example 2.11: Suppose we want the set of tuples in the relation `Movies` that represent Fox movies at least 100 minutes long. We can get these tuples with a more complicated condition, involving the `AND` of two subconditions. The expression is

$$\sigma_{length \geq 100 \texttt{ AND } studioName = \texttt{'Fox'}}(\texttt{Movies})$$

The tuple

title	*year*	*length*	*inColor*	*studioName*	*producerC#*
Star Wars	1977	124	true	Fox	12345

is the only one in the resulting relation. □

2.4.7 Cartesian Product

The *Cartesian product* (or *cross-product*, or just *product*) of two sets R and S is the set of pairs that can be formed by choosing the first element of the pair to be any element of R and the second any element of S. This product is denoted $R \times S$. When R and S are relations, the product is essentially the same. However, since the members of R and S are tuples, usually consisting of more than one component, the result of pairing a tuple from R with a tuple from S is a longer tuple, with one component for each of the components of the constituent tuples. By convention, the components from R (the left operand) precede the components from S in the attribute order for the result.

The relation schema for the resulting relation is the union of the schemas for R and S. However, if R and S should happen to have some attributes in common, then we need to invent new names for at least one of each pair of identical attributes. To disambiguate an attribute A that is in the schemas of both R and S, we use $R.A$ for the attribute from R and $S.A$ for the attribute from S.

Example 2.12: For conciseness, let us use an abstract example that illustrates the product operation. Let relations R and S have the schemas and tuples shown in Fig. 2.14(a) and (b). Then the product $R \times S$ consists of the six tuples shown in Fig. 2.14(c). Note how we have paired each of the two tuples of R with each of the three tuples of S. Since B is an attribute of both schemas, we have used $R.B$ and $S.B$ in the schema for $R \times S$. The other attributes are unambiguous, and their names appear in the resulting schema unchanged. □

2.4.8 Natural Joins

More often than we want to take the product of two relations, we find a need to *join* them by pairing only those tuples that match in some way. The simplest sort of match is the *natural join* of two relations R and S, denoted $R \bowtie S$, in which we pair only those tuples from R and S that agree in whatever attributes are common to the schemas of R and S. More precisely, let $A_1, A_2, \ldots, A_n$ be all the attributes that are in both the schema of R and the schema of S. Then a tuple r from R and a tuple s from S are successfully paired if and only if r and s agree on each of the attributes $A_1, A_2, \ldots, A_n$.

If the tuples r and s are successfully paired in the join $R \bowtie S$, then the result of the pairing is a tuple, called the *joined tuple*, with one component for each of the attributes in the union of the schemas of R and S. The joined tuple

A	B
1	2
3	4

(a) Relation R

B	C	D
2	5	6
4	7	8
9	10	11

(b) Relation S

A	$R.B$	$S.B$	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

(c) Result $R \times S$

Figure 2.14: Two relations and their Cartesian product

agrees with tuple r in each attribute in the schema of R, and it agrees with s in each attribute in the schema of S. Since r and s are successfully paired, the joined tuple is able to agree with both these tuples on the attributes they have in common. The construction of the joined tuple is suggested by Fig. 2.15. However, the order of the attributes need not be that convenient; the attributes of R and S can appear in any order.

Example 2.13: The natural join of the relations R and S from Fig. 2.14(a) and (b) is

A	B	C	D
1	2	5	6
3	4	7	8

The only attribute common to R and S is B. Thus, to pair successfully, tuples need only to agree in their B components. If so, the resulting tuple has components for attributes A (from R), B (from either R or S), C (from S), and D (from S).

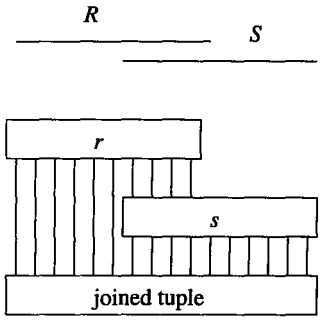

Figure 2.15: Joining tuples

In this example, the first tuple of R successfully pairs with only the first tuple of S; they share the value 2 on their common attribute B. This pairing yields the first tuple of the result: $(1,2,5,6)$. The second tuple of R pairs successfully only with the second tuple of S, and the pairing yields $(3,4,7,8)$. Note that the third tuple of S does not pair with any tuple of R and thus has no effect on the result of $R \bowtie S$. A tuple that fails to pair with any tuple of the other relation in a join is said to be a *dangling tuple.* □

Example 2.14: The previous example does not illustrate all the possibilities inherent in the natural join operator. For example, no tuple paired successfully with more than one tuple, and there was only one attribute in common to the two relation schemas. In Fig. 2.16 we see two other relations, U and V, that share two attributes between their schemas: B and C. We also show an instance in which one tuple joins with several tuples.

For tuples to pair successfully, they must agree in both the B and C components. Thus, the first tuple of U joins with the first two tuples of V, while the second and third tuples of U join with the third tuple of V. The result of these four pairings is shown in Fig. 2.16(c). □

2.4.9 Theta-Joins

The natural join forces us to pair tuples using one specific condition. While this way, equating shared attributes, is the most common basis on which relations are joined, it is sometimes desirable to pair tuples from two relations on some other basis. For that purpose, we have a related notation called the *theta-join.* Historically, the "theta" refers to an arbitrary condition, which we shall represent by C rather than θ.

The notation for a theta-join of relations R and S based on condition C is $R \bowtie_C S$. The result of this operation is constructed as follows:

1. Take the product of R and S.

2. Select from the product only those tuples that satisfy the condition C.

A	B	C
1	2	3
6	7	8
9	7	8

(a) Relation U

B	C	D
2	3	4
2	3	5
7	8	10

(b) Relation V

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

(c) Result $U \bowtie V$

Figure 2.16: Natural join of relations

As with the product operation, the schema for the result is the union of the schemas of R and S, with "R." or "S." prefixed to attributes if necessary to indicate from which schema the attribute came.

Example 2.15: Consider the operation $U \bowtie_{A<D} V$, where U and V are the relations from Fig. 2.16(a) and (b). We must consider all nine pairs of tuples, one from each relation, and see whether the A component from the U-tuple is less than the D component of the V-tuple. The first tuple of U, with an A component of 1, successfully pairs with each of the tuples from V. However, the second and third tuples from U, with A components of 6 and 9, respectively, pair successfully with only the last tuple of V. Thus, the result has only five tuples, constructed from the five successful pairings. This relation is shown in Fig. 2.17. □

Notice that the schema for the result in Fig. 2.17 consists of all six attributes, with U and V prefixed to their respective occurrences of attributes B and C to distinguish them. Thus, the theta-join contrasts with natural join, since in the latter common attributes are merged into one copy. Of course it makes sense to

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

Figure 2.17: Result of $U \bowtie_{A<D} V$

do so in the case of the natural join, since tuples don't pair unless they agree in their common attributes. In the case of a theta-join, there is no guarantee that compared attributes will agree in the result, since they may not be compared with $=$.

Example 2.16: Here is a theta-join on the same relations U and V that has a more complex condition:

$$U \bowtie_{A<D \text{ AND } U.B \neq V.B} V$$

That is, we require for successful pairing not only that the A component of the U-tuple be less than the D component of the V-tuple, but that the two tuples disagree on their respective B components. The tuple

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	7	8	10

is the only one to satisfy both conditions, so this relation is the result of the theta-join above. □

2.4.10 Combining Operations to Form Queries

If all we could do was to write single operations on one or two relations as queries, then relational algebra would not be nearly as useful as it is. However, relational algebra, like all algebras, allows us to form expressions of arbitrary complexity by applying operations to the result of other operations.

One can construct expressions of relational algebra by applying operators to subexpressions, using parentheses when necessary to indicate grouping of operands. It is also possible to represent expressions as expression trees; the latter often are easier for us to read, although they are less convenient as a machine-readable notation.

Example 2.17: Suppose we want to know, from our running `Movies` relation, "What are the titles and years of movies made by Fox that are at least 100 minutes long?" One way to compute the answer to this query is:

1. Select those `Movies` tuples that have *length* ≥ 100.

2. Select those `Movies` tuples that have *studioName* = `'Fox'`.

3. Compute the intersection of (1) and (2).

4. Project the relation from (3) onto attributes `title` and `year`.

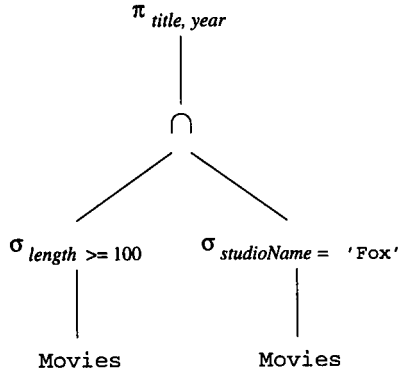

Figure 2.18: Expression tree for a relational algebra expression

In Fig. 2.18 we see the above steps represented as an expression tree. Expression trees are evaluated bottom-up by applying the operator at an interior node to the arguments, which are the results of its children. By proceeding bottom-up, we know that the arguments will be available when we need them. The two selection nodes correspond to steps (1) and (2). The intersection node corresponds to step (3), and the projection node is step (4).

Alternatively, we could represent the same expression in a conventional, linear notation, with parentheses. The formula

$$\pi_{title,year}\Big(\sigma_{length\geq 100}(\texttt{Movies}) \cap \sigma_{studioName=\texttt{'Fox'}}(\texttt{Movies})\Big)$$

represents the same expression.

Incidentally, there is often more than one relational algebra expression that represents the same computation. For instance, the above query could also be written by replacing the intersection by logical `AND` within a single selection operation. That is,

$$\pi_{title,year}\Big(\sigma_{length\geq 100 \texttt{ AND } studioName=\texttt{'Fox'}}(\texttt{Movies})\Big)$$

is an equivalent form of the query. □

Equivalent Expressions and Query Optimization

All database systems have a query-answering system, and many of them are based on a language that is similar in expressive power to relational algebra. Thus, the query asked by a user may have many *equivalent expressions* (expressions that produce the same answer whenever they are given the same relations as operands), and some of these may be much more quickly evaluated. An important job of the query "optimizer" discussed briefly in Section 1.2.5 is to replace one expression of relational algebra by an equivalent expression that is more efficiently evaluated.

2.4.11 Naming and Renaming

In order to control the names of the attributes used for relations that are constructed by applying relational-algebra operations, it is often convenient to use an operator that explicitly renames relations. We shall use the operator $\rho_{S(A_1,A_2,\ldots,A_n)}(R)$ to rename a relation R. The resulting relation has exactly the same tuples as R, but the name of the relation is S. Moreover, the attributes of the result relation S are named $A_1, A_2, \ldots, A_n$, in order from the left. If we only want to change the name of the relation to S and leave the attributes as they are in R, we can just say $\rho_S(R)$.

Example 2.18: In Example 2.12 we took the product of two relations R and S from Fig. 2.14(a) and (b) and used the convention that when an attribute appears in both operands, it is renamed by prefixing the relation name to it. Suppose, however, that we do not wish to call the two versions of B by names $R.B$ and $S.B$; rather we want to continue to use the name B for the attribute that comes from R, and we want to use X as the name of the attribute B coming from S. We can rename the attributes of S so the first is called X. The result of the expression $\rho_{S(X,C,D)}(S)$ is a relation named S that looks just like the relation S from Fig. 2.14, but its first column has attribute X instead of B.

A	B	X	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Figure 2.19: $R \times \rho_{S(X,C,D)}(S)$

When we take the product of R with this new relation, there is no conflict of names among the attributes, so no further renaming is done. That is, the result of the expression $R \times \rho_{S(X,C,D)}(S)$ is the relation $R \times S$ from Fig. 2.14(c), except that the five columns are labeled A, B, X, C, and D, from the left. This relation is shown in Fig. 2.19.

As an alternative, we could take the product without renaming, as we did in Example 2.12, and then rename the result. The expression

$$\rho_{RS(A,B,X,C,D)}(R \times S)$$

yields the same relation as in Fig. 2.19, with the same set of attributes. But this relation has a name, RS, while the result relation in Fig. 2.19 has no name. □

2.4.12 Relationships Among Operations

Some of the operations that we have described in Section 2.4 can be expressed in terms of other relational-algebra operations. For example, intersection can be expressed in terms of set difference:

$$R \cap S = R - (R - S)$$

That is, if R and S are any two relations with the same schema, the intersection of R and S can be computed by first subtracting S from R to form a relation T consisting of all those tuples in R but not S. We then subtract T from R, leaving only those tuples of R that are also in S.

The two forms of join are also expressible in terms of other operations. Theta-join can be expressed by product and selection:

$$R \bowtie_C S = \sigma_C(R \times S)$$

The natural join of R and S can be expressed by starting with the product $R \times S$. We then apply the selection operator with a condition C of the form

$$R.A_1 = S.A_1 \text{ AND } R.A_2 = S.A_2 \text{ AND} \cdots \text{AND } R.A_n = S.A_n$$

where $A_1, A_2, \ldots, A_n$ are all the attributes appearing in the schemas of both R and S. Finally, we must project out one copy of each of the equated attributes. Let L be the list of attributes in the schema of R followed by those attributes in the schema of S that are not also in the schema of R. Then

$$R \bowtie S = \pi_L\Big(\sigma_C(R \times S)\Big)$$

Example 2.19: The natural join of the relations U and V from Fig. 2.16 can be written in terms of product, selection, and projection as:

$$\pi_{A,U.B,U.C,D}\Big(\sigma_{U.B=V.B \text{ AND } U.C=V.C}(U \times V)\Big)$$

That is, we take the product $U \times V$. Then we select for equality between each pair of attributes with the same name — B and C in this example. Finally, we project onto all the attributes except one of the B's and one of the C's; we have chosen to eliminate the attributes of V whose names also appear in the schema of U.

For another example, the theta-join of Example 2.16 can be written

$$\sigma_{A<D \text{ AND } U.B \neq V.B}(U \times V)$$

That is, we take the product of the relations U and V and then apply the condition that appeared in the theta-join. □

The rewriting rules mentioned in this section are the only "redundancies" among the operations that we have introduced. The six remaining operations — union, difference, selection, projection, product, and renaming — form an independent set, none of which can be written in terms of the other five.

2.4.13 A Linear Notation for Algebraic Expressions

In Section 2.4.10 we used an expression tree to represent a complex expression of relational algebra. An alternative is to invent names for the temporary relations that correspond to the interior nodes of the tree and write a sequence of assignments that create a value for each. The order of the assignments is flexible, as long as the children of a node N have had their values created before we attempt to create the value for N itself.

The notation we shall use for assignment statements is:

1. A relation name and parenthesized list of attributes for that relation. The name `Answer` will be used conventionally for the result of the final step; i.e., the name of the relation at the root of the expression tree.

2. The assignment symbol `:=`.

3. Any algebraic expression on the right. We can choose to use only one operator per assignment, in which case each interior node of the tree gets its own assignment statement. However, it is also permissible to combine several algebraic operations in one right side, if it is convenient to do so.

Example 2.20: Consider the tree of Fig. 2.18. One possible sequence of assignments to evaluate this expression is:

$$\begin{aligned}
&\texttt{R(t,y,l,i,s,p)} := \sigma_{length \geq 100}(\texttt{Movies}) \\
&\texttt{S(t,y,l,i,s,p)} := \sigma_{studioName='\texttt{Fox}'}(\texttt{Movies}) \\
&\texttt{T(t,y,l,i,s,p)} := \texttt{R} \cap \texttt{S} \\
&\texttt{Answer(title, year)} := \pi_{t,y}(\texttt{T})
\end{aligned}$$

The first step computes the relation of the interior node labeled $\sigma_{length \geq 100}$ in Fig. 2.18, and the second step computes the node labeled $\sigma_{studioName='Fox'}$. Notice that we get renaming "for free," since we can use any attributes and relation name we wish for the left side of an assignment. The last two steps compute the intersection and the projection in the obvious way.

It is also permissible to combine some of the steps. For instance, we could combine the last two steps and write:

$$\texttt{R(t,y,l,i,s,p)} := \sigma_{length \geq 100}(\texttt{Movies})$$
$$\texttt{S(t,y,l,i,s,p)} := \sigma_{studioName='Fox'}(\texttt{Movies})$$
$$\texttt{Answer(title, year)} := \pi_{t,y}(\texttt{R} \cap \texttt{S})$$

We could even substitute for R and S in the last line and write the entire expression in one line. □

2.4.14 Exercises for Section 2.4

Exercise 2.4.1: This exercise builds upon the products schema of Exercise 2.3.1. Recall that 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)
```

Some sample data for the relation `Product` is shown in Fig. 2.20. Sample data for the other three relations is shown in Fig. 2.21. Manufacturers and model numbers have been "sanitized," but the data is typical of products on sale at the beginning of 2007.

Write expressions of relational algebra to answer the following queries. You may use the linear notation of Section 2.4.13 if you wish. For the data of Figs. 2.20 and 2.21, show the result of your query. However, your answer should work for arbitrary data, not just the data of these figures.

a) Find those manufacturers that sell printers, but not PC's.

b) What PC models have a speed of at least 2.50?

c) Which manufacturers make laptops with a hard disk of at least 120GB?

d) Find the model number and price of all products (of any type) made by manufacturer C.

e) Find the model numbers of all black-and-white laser printers.

! f) Find those hard-disk sizes that occur in two or more PC's.

maker	*model*	*type*
A	1001	pc
A	1002	pc
A	1003	pc
A	2004	laptop
A	2005	laptop
A	2006	laptop
B	1004	pc
B	1005	pc
B	1006	pc
B	2007	laptop
C	1007	pc
D	1008	pc
D	1009	pc
D	1010	pc
D	3004	printer
D	3005	printer
E	1011	pc
E	1012	pc
E	1013	pc
E	2001	laptop
E	2002	laptop
E	2003	laptop
E	3001	printer
E	3002	printer
E	3003	printer
F	2008	laptop
F	2009	laptop
G	2010	laptop
H	3006	printer
H	3007	printer

Figure 2.20: Sample data for Product

model	*speed*	*ram*	*hd*	*price*
1001	2.66	1024	250	2114
1002	2.10	512	250	995
1003	1.42	512	80	478
1004	2.80	1024	250	649
1005	3.20	512	250	630
1006	3.20	1024	320	1049
1007	2.20	1024	200	510
1008	2.20	2048	250	770
1009	2.00	1024	250	650
1010	2.80	2048	300	770
1011	1.86	2048	160	959
1012	2.80	1024	160	649
1013	3.06	512	80	529

(a) Sample data for relation PC

model	*speed*	*ram*	*hd*	*screen*	*price*
2001	2.00	2048	240	20.1	3673
2002	1.73	1024	80	17.0	949
2003	1.80	512	60	15.4	549
2004	2.00	512	60	13.3	1150
2005	2.16	1024	120	17.0	2500
2006	2.00	2048	80	15.4	1700
2007	1.83	1024	120	13.3	1429
2008	1.60	1024	100	15.4	900
2009	1.60	512	80	14.1	680
2010	2.00	2048	160	15.4	2300

(b) Sample data for relation Laptop

model	*color*	*type*	*price*
3001	true	ink-jet	99
3002	false	laser	239
3003	true	laser	899
3004	true	ink-jet	120
3005	false	laser	120
3006	true	ink-jet	100
3007	true	laser	200

(c) Sample data for relation Printer

Figure 2.21: Sample data for relations of Exercise 2.4.1

! g) Find those pairs of PC models that have both the same speed and RAM. A pair should be listed only once; e.g., list (i, j) but not (j, i).

!! h) Find those manufacturers of at least two different computers (PC's or laptops) with speeds of at least 2.20.

!! k) Find the manufacturers who sell exactly three different models of PC.

!! j) Find the manufacturer(s) of the computer (PC or laptop) with the highest available speed.

!! k) Find the manufacturers of PC's with at least three different speeds.

Exercise 2.4.2: Draw expression trees for each of your expressions of Exercise 2.4.1.

Exercise 2.4.3: This exercise builds upon Exercise 2.3.2 concerning World War II capital ships. Recall it involves the following relations:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

Figures 2.22 and 2.23 give some sample data for these four relations.[4] Note that, unlike the data for Exercise 2.4.1, there are some "dangling tuples" in this data, e.g., ships mentioned in `Outcomes` that are not mentioned in `Ships`.

Write expressions of relational algebra to answer the following queries. You may use the linear notation of Section 2.4.13 if you wish. For the data of Figs. 2.22 and 2.23, show the result of your query. However, your answer should work for arbitrary data, not just the data of these figures.

a) Find the ships launched prior to 1917.

b) Find the ships sunk in the battle of Surigao Strait.

c) The treaty of Washington in 1921 prohibited capital ships heavier than 35,000 tons. List the ships that violated the treaty of Washington.

d) List the name, displacement, and number of guns of the ships engaged in the battle of North Cape.

e) List all the capital ships mentioned in the database. (Remember that all these ships may not appear in the `Ships` relation.)

f) Give the class names and countries of the classes that carried guns of at least 16-inch bore.

[4]Source: J. N. Westwood, *Fighting Ships of World War II*, Follett Publishing, Chicago, 1975 and R. C. Stern, *US Battleships in Action*, Squadron/Signal Publications, Carrollton, TX, 1980.

class	*type*	*country*	*numGuns*	*bore*	*displacement*
Bismarck	bb	Germany	8	15	42000
Iowa	bb	USA	9	16	46000
Kongo	bc	Japan	8	14	32000
South Dakota	bb	USA	9	16	37000
Renown	bc	Gt. Britain	6	15	32000
Revenge	bb	Gt. Britain	8	15	29000
Mississippi	bb	USA	12	14	33000
Yamato	bb	Japan	9	18	65000

(a) Sample data for relation Classes

name	*date*
Denmark Strait	5/24-27/41
Guadalcanal	11/15/42
North Cape	12/26/43
Surigao Strait	10/25/44

(b) Sample data for relation Battles

ship	*battle*	*result*
Arizona	Pearl Harbor	sunk
Bismarck	Denmark Strait	sunk
California	Surigao Strait	ok
Duke of York	North Cape	ok
Fuso	Surigao Strait	sunk
Hood	Denmark Strait	sunk
King George V	Denmark Strait	ok
Kirishima	Guadalcanal	sunk
Prince of Wales	Denmark Strait	damaged
Rodney	Denmark Strait	ok
Scharnhorst	North Cape	sunk
South Dakota	Guadalcanal	damaged
Tennessee	Surigao Strait	ok
Washington	Guadalcanal	ok
West Virginia	Surigao Strait	ok
Yamashiro	Surigao Strait	sunk

(c) Sample data for relation Outcomes

Figure 2.22: Data for Exercise 2.4.3

name	*class*	*launched*
Alabama	South Dakota	1942
Haruna	Kongo	1915
Hiei	Kongo	1914
Idaho	Mississippi	1919
Iowa	Iowa	1943
Kirishima	Kongo	1915
Kongo	Kongo	1913
Missouri	Iowa	1944
Musashi	Yamato	1942
New Jersey	Iowa	1943
New Mexico	Mississippi	1918
Ramillies	Revenge	1917
Renown	Renown	1916
Repulse	Renown	1916
Resolution	Revenge	1916
Revenge	Revenge	1916
Royal Oak	Revenge	1916
Royal Sovereign	Revenge	1916
South Dakota	South Dakota	1942
Wisconsin	Iowa	1944
Yamato	Yamato	1941

Figure 2.23: Sample data for relation Ships

! g) Find those countries that had both battleships and battlecruisers.

! h) Find those ships that "lived to fight another day"; they were damaged in one battle, but later fought in another.

! i) Find the classes that had only one ship as a member of that class.

Exercise 2.4.4: Draw expression trees for each of your expressions of Exercise 2.4.3.

Exercise 2.4.5: What is the difference between the natural join $R \bowtie S$ and the theta-join $R \bowtie_C S$ where the condition C is that $R.A = S.A$ for each attribute A appearing in the schemas of both R and S?

! **Exercise 2.4.6:** An operator on relations is said to be *monotone* if whenever we add a tuple to one of its arguments, the result contains all the tuples that it contained before adding the tuple, plus perhaps more tuples. Which of the operators described in this section are monotone? For each, either explain why it is monotone or give an example showing it is not.

! **Exercise 2.4.7:** Suppose relations R and S have n tuples and m tuples, respectively. Give the minimum and maximum numbers of tuples that the results of the following expressions can have.

a) $R \cup S$.

b) $R \bowtie S$.

c) $\sigma_C(R) \times S$, for some condition C.

d) $\pi_L(R) - S$, for some list of attributes L.

! **Exercise 2.4.8:** The *semijoin* of relations R and S, written $R \ltimes S$, is the set of tuples t in R such that there is at least one tuple in S that agrees with t in all attributes that R and S have in common. Give three different expressions of relational algebra that are equivalent to $R \ltimes S$.

! **Exercise 2.4.9:** The *antisemijoin* $R \overline{\ltimes} S$ is the set of tuples t in R that do *not* agree with any tuple of S in the attributes common to R and S. Give an expression of relational algebra equivalent to $R \overline{\ltimes} S$.

!! **Exercise 2.4.10:** Let R be a relation with schema

$$(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$$

and let S be a relation with schema $(B_1, B_2, \ldots, B_m)$; that is, the attributes of S are a subset of the attributes of R. The *quotient* of R and S, denoted $R \div S$, is the set of tuples t over attributes $A_1, A_2, \ldots, A_n$ (i.e., the attributes of R that are not attributes of S) such that for every tuple s in S, the tuple ts, consisting of the components of t for $A_1, A_2, \ldots, A_n$ and the components of s for $B_1, B_2, \ldots, B_m$, is a member of R. Give an expression of relational algebra, using the operators we have defined previously in this section, that is equivalent to $R \div S$.

2.5 Constraints on Relations

We now take up the third important aspect of a data model: the ability to restrict the data that may be stored in a database. So far, we have seen only one kind of constraint, the requirement that an attribute or attributes form a key (Section 2.3.6). These and many other kinds of constraints can be expressed in relational algebra. In this section, we show how to express both key constraints and "referential-integrity" constraints; the latter require that a value appearing in one column of one relation also appear in some other column of the same or a different relation. In Chapter 7, we see how SQL database systems can enforce the same sorts of constraints as we can express in relational algebra.

2.5.1 Relational Algebra as a Constraint Language

There are two ways in which we can use expressions of relational algebra to express constraints.

1. If R is an expression of relational algebra, then $R = \emptyset$ is a constraint that says "The value of R must be empty," or equivalently "There are no tuples in the result of R."

2. If R and S are expressions of relational algebra, then $R \subseteq S$ is a constraint that says "Every tuple in the result of R must also be in the result of S." Of course the result of S may contain additional tuples not produced by R.

These ways of expressing constraints are actually equivalent in what they can express, but sometimes one or the other is clearer or more succinct. That is, the constraint $R \subseteq S$ could just as well have been written $R - S = \emptyset$. To see why, notice that if every tuple in R is also in S, then surely $R - S$ is empty. Conversely, if $R - S$ contains no tuples, then every tuple in R must be in S (or else it would be in $R - S$).

On the other hand, a constraint of the first form, $R = \emptyset$, could just as well have been written $R \subseteq \emptyset$. Technically, $\emptyset$ is not an expression of relational algebra, but since there are expressions that evaluate to $\emptyset$, such as $R - R$, there is no harm in using $\emptyset$ as a relational-algebra expression.

In the following sections, we shall see how to express significant constraints in one of these two styles. As we shall see in Chapter 7, it is the first style — equal-to-the-emptyset — that is most commonly used in SQL programming. However, as shown above, we are free to think in terms of set-containment if we wish and later convert our constraint to the equal-to-the-emptyset style.

2.5.2 Referential Integrity Constraints

A common kind of constraint, called a *referential integrity constraint*, asserts that a value appearing in one context also appears in another, related context. For example, in our movies database, should we see a `StarsIn` tuple that has person p in the `starName` component, we would expect that p appears as the name of some star in the `MovieStar` relation. If not, then we would question whether the listed "star" really was a star.

In general, if we have any value v as the component in attribute A of some tuple in one relation R, then because of our design intentions we may expect that v will appear in a particular component (say for attribute B) of some tuple of another relation S. We can express this integrity constraint in relational algebra as $\pi_A(R) \subseteq \pi_B(S)$, or equivalently, $\pi_A(R) - \pi_B(S) = \emptyset$.

Example 2.21: Consider the two relations from our running movie database:

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

We might reasonably assume that the producer of every movie would have to appear in the MovieExec relation. If not, there is something wrong, and we would at least want a system implementing a relational database to inform us that we had a movie with a producer of which the database had no knowledge.

To be more precise, the producerC# component of each Movies tuple must also appear in the cert# component of some MovieExec tuple. Since executives are uniquely identified by their certificate numbers, we would thus be assured that the movie's producer is found among the movie executives. We can express this constraint by the set-containment

$$\pi_{producerC\#}(\texttt{Movies}) \subseteq \pi_{cert\#}(\texttt{MovieExec})$$

The value of the expression on the left is the set of all certificate numbers appearing in producerC# components of Movies tuples. Likewise, the expression on the right's value is the set of all certificates in the cert# component of MovieExec tuples. Our constraint says that every certificate in the former set must also be in the latter set. □

Example 2.22: We can similarly express a referential integrity constraint where the "value" involved is represented by more than one attribute. For instance, we may want to assert that any movie mentioned in the relation

```
StarsIn(movieTitle, movieYear, starName)
```

also appears in the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

Movies are represented in both relations by title-year pairs, because we agreed that one of these attributes alone was not sufficient to identify a movie. The constraint

$$\pi_{movieTitle,\ movieYear}(\texttt{StarsIn}) \subseteq \pi_{title,\ year}(\texttt{Movies})$$

expresses this referential integrity constraint by comparing the title-year pairs produced by projecting both relations onto the appropriate lists of components. □

2.5.3 Key Constraints

The same constraint notation allows us to express far more than referential integrity. Here, we shall see how we can express algebraically the constraint that a certain attribute or set of attributes is a key for a relation.

Example 2.23: Recall that name is the key for relation

```
MovieStar(name, address, gender, birthdate)
```

That is, no two tuples agree on the name component. We shall express algebraically one of several implications of this constraint: that if two tuples agree on name, then they must also agree on address. Note that in fact these "two" tuples, which agree on the key name, must be the same tuple and therefore certainly agree in all attributes.

The idea is that if we construct all pairs of MovieStar tuples (t_1, t_2), we must not find a pair that agree in the name component and disagree in the address component. To construct the pairs we use a Cartesian product, and to search for pairs that violate the condition we use a selection. We then assert the constraint by equating the result to $\emptyset$.

To begin, since we are taking the product of a relation with itself, we need to rename at least one copy, in order to have names for the attributes of the product. For succinctness, let us use two new names, MS1 and MS2, to refer to the MovieStar relation. Then the requirement can be expressed by the algebraic constraint:

$$\sigma_{MS1.name=MS2.name \text{ AND } MS1.address \neq MS2.address}(\texttt{MS1} \times \texttt{MS2}) = \emptyset$$

In the above, MS1 in the product MS1 × MS2 is shorthand for the renaming:

$$\rho_{MS1(name,address,gender,birthdate)}(\texttt{MovieStar})$$

and MS2 is a similar renaming of MovieStar. □

2.5.4 Additional Constraint Examples

There are many other kinds of constraints that we can express in relational algebra and that are useful for restricting database contents. A large family of constraints involve the permitted values in a context. For example, the fact that each attribute has a type constrains the values of that attribute. Often the constraint is quite straightforward, such as "integers only" or "character strings of length up to 30." Other times we want the values that may appear in an attribute to be restricted to a small enumerated set of values. Other times, there are complex limitations on the values that may appear. We shall give two examples, one of a simple *domain constraint* for an attribute, and the second a more complicated restriction.

Example 2.24: Suppose we wish to specify that the only legal values for the gender attribute of MovieStar are 'F' and 'M'. We can express this constraint algebraically by:

$$\sigma_{gender \neq \text{'F'} \text{ AND } gender \neq \text{'M'}}(\texttt{MovieStar}) = \emptyset$$

That is, the set of tuples in MovieStar whose gender component is equal to neither 'F' nor 'M' is empty. □

Example 2.25: Suppose we wish to require that one must have a net worth of at least \$10,000,000 to be the president of a movie studio. We can express this constraint algebraically as follows. First, we need to theta-join the two relations

```
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

using the condition that presC# from Studio and cert# from MovieExec are equal. That join combines pairs of tuples consisting of a studio and an executive, such that the executive is the president of the studio. If we select from this relation those tuples where the net worth is less than ten million, we have a set that, according to our constraint, must be empty. Thus, we may express the constraint as:

$$\sigma_{netWorth<10000000}(\texttt{Studio} \bowtie_{presC\#=cert\#} \texttt{MovieExec}) = \emptyset$$

An alternative way to express the same constraint is to compare the set of certificates that represent studio presidents with the set of certificates that represent executives with a net worth of at least \$10,000,000; the former must be a subset of the latter. The containment

$$\pi_{presC\#}(\texttt{Studio}) \subseteq \pi_{cert\#}\Big(\sigma_{netWorth\geq 10000000}(\texttt{MovieExec})\Big)$$

expresses the above idea. □

2.5.5 Exercises for Section 2.5

Exercise 2.5.1: Express the following constraints about the relations of Exercise 2.3.1, reproduced here:

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

You may write your constraints either as containments or by equating an expression to the empty set. For the data of Exercise 2.4.1, indicate any violations to your constraints.

a) A PC with a processor speed less than 3.00 must not sell for more than \$800.

b) A laptop with a screen size less than 15.4 inches must have at least a 120 gigabyte hard disk or sell for less than \$1000.

! c) No manufacturer of PC's may also make printers.

! d) If a laptop has a larger main memory than a PC, then the laptop must also have a higher price than the PC.

!! e) A manufacturer of a PC must also make a laptop with at least as great a processor speed.

Exercise 2.5.2: Express the following constraints in relational algebra. The constraints are based on the relations of Exercise 2.3.2:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

You may write your constraints either as containments or by equating an expression to the empty set. For the data of Exercise 2.4.3, indicate any violations to your constraints.

a) No class of ships may have guns with larger than 18-inch bore.

b) If a class of ships has more than 10 guns, then their bore must be no larger than 15 inches.

! c) No class may have more than 3 ships.

! d) No country may have both battleships and battlecruisers.

!! e) No ship with more than 10 guns may be in a battle with a ship having fewer than 9 guns that was sunk.

! **Exercise 2.5.3:** Suppose R and S are two relations. Let C be the referential integrity constraint that says: whenever R has a tuple with some values $v_1, v_2, \ldots, v_n$ in particular attributes $A_1, A_2, \ldots, A_n$, there must be a tuple of S that has the same values $v_1, v_2, \ldots, v_n$ in particular attributes $B_1, B_2, \ldots, B_n$. Show how to express constraint C in relational algebra.

! **Exercise 2.5.4:** Another algebraic way to express a constraint is $E_1 = E_2$, where both E_1 and E_2 are relational-algebra expressions. Can this form of constraint express more than the two forms we discussed in this section?

2.6 Summary of Chapter 2

✦ *Data Models*: A data model is a notation for describing the structure of the data in a database, along with the constraints on that data. The data model also normally provides a notation for describing operations on that data: queries and data modifications.

- *Relational Model*: Relations are tables representing information. Columns are headed by attributes; each attribute has an associated domain, or data type. Rows are called tuples, and a tuple has one component for each attribute of the relation.
- *Schemas*: A relation name, together with the attributes of that relation and their types, form the relation schema. A collection of relation schemas forms a database schema. Particular data for a relation or collection of relations is called an instance of that relation schema or database schema.
- *Keys*: An important type of constraint on relations is the assertion that an attribute or set of attributes forms a key for the relation. No two tuples of a relation can agree on all attributes of the key, although they can agree on some of the key attributes.
- *Semistructured Data Model*: In this model, data is organized in a tree or graph structure. XML is an important example of a semistructured data model.
- *SQL*: The language SQL is the principal query language for relational database systems. The current standard is called SQL-99. Commercial systems generally vary from this standard but adhere to much of it.
- *Data Definition*: SQL has statements to declare elements of a database schema. The `CREATE TABLE` statement allows us to declare the schema for stored relations (called tables), specifying the attributes, their types, default values, and keys.
- *Altering Schemas*: We can change parts of the database schema with an `ALTER` statement. These changes include adding and removing attributes from relation schemas and changing the default value associated with an attribute. We may also use a `DROP` statement to completely eliminate relations or other schema elements.
- *Relational Algebra*: This algebra underlies most query languages for the relational model. Its principal operators are union, intersection, difference, selection, projection, Cartesian product, natural join, theta-join, and renaming.
- *Selection and Projection*: The selection operator produces a result consisting of all tuples of the argument relation that satisfy the selection condition. Projection removes undesired columns from the argument relation to produce the result.
- *Joins*: We join two relations by comparing tuples, one from each relation. In a natural join, we splice together those pairs of tuples that agree on all attributes common to the two relations. In a theta-join, pairs of tuples are concatenated if they meet a selection condition associated with the theta-join.

- ✦ *Constraints in Relational Algebra*: Many common kinds of constraints can be expressed as the containment of one relational algebra expression in another, or as the equality of a relational algebra expression to the empty set.

2.7 References for Chapter 2

The classic paper by Codd on the relational model is [1]. This paper introduces relational algebra, as well. The use of relational algebra to describe constraints is from [2]. References for SQL are given in the bibliographic notes for Chapter 6.

The semistructured data model is from [3]. XML is a standard developed by the World-Wide-Web Consortium. The home page for information about XML is [4].

1. E. F. Codd, "A relational model for large shared data banks," *Comm. ACM* **13**:6, pp. 377–387, 1970.

2. J.-M. Nicolas, "Logic for improving integrity checking in relational databases," *Acta Informatica* **18**:3, pp. 227–253, 1982.

3. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object exchange across heterogeneous information sources," *IEEE Intl. Conf. on Data Engineering*, pp. 251–260, March 1995.

4. World-Wide-Web Consortium, `http://www.w3.org/XML/`