3

Relational Database Fundamentals

The previous chapters discussed databases in general terms. Chapter 1 explained the general goals of database design. Chapter 2 described some of the many kinds of databases that you might decide to use.

With this chapter the book starts to focus on a particular kind of database: the relational database. Relational databases are very powerful, are the most commonly used kind of database in computer applications today, and are the focus of the rest of this book.

Before you can start learning how to properly design a relational database, you must understand the basic concepts and terms that underlie relational databases.

This chapter provides an introduction to relational databases. It explains the major ideas and terms that you need to know before you can start designing and building relational databases.

In this chapter, you learn about relational database terms such as:

- ❑ Table and relation
- ❑ Record, row, and tuple
- ❑ Column, field, and attribute
- ❑ Constraint, key, and index

Finally, you learn about the operations that you can use to get data out of a relational database.

Relational Points of View

Relational databases play a critical role in many important (that is, money-related) computer applications. As is the case whenever enormous amounts of money are at stake, people have spent a huge amount of time and effort building, studying, and refining relational databases. Database researchers usually approach relational databases from one of three points of view.

The first group approaches the problem from a database-theoretical point of view. These people tend to think in terms of provability, mathematical set theory, and propositional logic. You'll see them at the local rave throwing around phrases such as *relational algebra*, *Cartesian product*, and *tuple relational calculus*. This approach is intellectually stimulating (and looks good on a resume) but can be a bit intimidating. These researchers focus on logical design and idealized database principles.

The second group approaches the matter from a less formal ''just build the database and get it done'' point of view. Their terminology tends to be less precise and rigorous but more intuitive. They tend to use terms that you may have heard before such as table, row, and column. These people focus on physical database design and pay more attention to concrete bits-and-bytes issues dealing with actually building a database and getting the most out of it.

The third group tends to think in terms of flat files and the underlying disk structure used to hold data. Though these people are probably in the minority these days, their terms file, record, and field snuck into database nomenclature and stuck. Many of those who still use these terms are programmers and other developers who look at the database from a consumer's ''how do I get my data out of it'' point of view.

These differing points of view have led to several different and potentially confusing ways to view relational databases. This can cause some serious confusion, particularly because the different groups have latched on to some of the same terms but used for different meanings. In fact, they sometimes use the term ''relation'' in very different ways (that are described later in this chapter).

This chapter loosely groups these terms into ''formal'' and ''informal'' categories, where the formal category includes the database theoretical terms and the informal category includes everything else.

This chapter starts with informal terms. Each section initially focuses on informal terms and concepts, and then explains how they fit together with their more formal equivalents.

Table, Rows, and Columns

Informally you can think of a relational database as a collection of *tables*, each containing *rows* and *columns*. At this level, it looks a lot like a workbook containing several worksheets (or spreadsheets), although a worksheet is much less constrained than a database table is. You can put just about anything in any cell in a worksheet. In contrast, every entry in a particular column of a table is expected to contain the same kind of data. For example, all of the cells in a particular column might contain phone numbers or last names.

Actually a poorly designed database application may allow the user to sneak some strange kinds of data into other fields. For example, if the database and user interface aren't designed properly, you might be able to enter a string such as ''none'' in a telephone number field. That's not the field's intent, however. In contrast, a spreadsheet's cells don't really care what you put in them.

The set of the values that are allowed for a column is called the column's *domain*. For example, a column's domain might be telephone numbers, bank account numbers, snowshoe sizes, or hang glider colors.

Domain is closely related to data type but it's not quite the same. A column's *data type* is the kind of data that the column can hold. The data types that you can use for a column depend on the particular database you are using but typical data types include integer, floating point number (a number with a decimal point), string, and date.

To see the difference between domain and data type, note that street address (323 Relational Rd) and jersey color (red) are both strings. However, the domain for the street address column is valid street addresses, whereas the domain for the jersey color column is colors (and possibly not even all colors if you only allow a few choices). (You can think of the data type as the highest level or most general possible domain. For example, an address or color domain is a more restrictive subset of the domain allowing all strings.)

The rows in a table correspond to column values that are related to each other according to the purpose of the table. For example, suppose you have a Competitors table that contains typical contact information for participants in your First (and probably Last) Annual Extreme Pyramid Sports Championship. This table includes columns to hold competitor name, address, event, blood type, and next of kin as shown in Figure 3-1. (Note that this is not a good database design. You'll see why in later chapters.)

Name	Address	Event	Blood Type	NextOfKin
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Boarding	A+	Art Adventure
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Luge	A+	Art Adventure
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Camel Drafting	O–	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Pyramid Boarding	O–	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Sphinx Jumping	O–	Betty Bold
Cindy Copes	271 Sledding Hill, Ricky Ride CO 80281	Camel Drafting	AB–	John Finkle
Cindy Copes	271 Sledding Hill, Ricky Ride CO 80281	Sphinx Jumping	AB–	John Finkle
Dean Daring	73 Fighter Ave, New Plunge UT 78281	Pyramid Boarding	O+	Betty Dare
Dean Daring	73 Fighter Ave, New Plunge UT 78281	Pyramid Luge	O+	Betty Dare
Frank Fiercely	3872 Bother Blvd, Lost City HI 99182	Pyramid Luge	B+	Fred Farce
Frank Fiercely	3872 Bother Blvd, Lost City HI 99182	Sphinx Jumping	B+	Fred Farce
George Forman	73 Fighter Ave, New Plunge UT 78281	Sphinx Jumping	O+	George Forman
George Forman	73 Fighter Ave, New Plunge UT 78281	Pyramid Luge	O+	George Forman
Gina Gruff	1 Skatepark Ln, Forever KS 72071	Camel Drafting	A+	Gill Gruff
Gina Gruff	1 Skatepark Ln, Forever KS 72071	Pyramid Boarding	A+	Gill Gruff

Figure 3-1

A particular row in the table holds all of the values for a given competitor. For example, the values in the first row (Alice Adventure, 6543 Flak Ter, Runner AZ 82018, Pyramid Boarding, A+, Art Adventure) all apply to the competitor Alice Adventure.

Back in olden times when database developers worked with primitive tools by candlelight, everyone lived much closer to nature. In this case that means they needed to work more closely with the underlying file system. It was common to store data in ''flat'' files without any indexes, search tools, or other fancy modern luxuries. A file would hold the related information that you might represent at a higher level as a table. The file was divided into chunks called *records* that each had the same size and that each corresponded to a row in a table. The records were divided into fixed-length *fields* that corresponded to the columns in a table.

For example, if you think of the table shown in Figure 3-1 as a flat file, the first row corresponds to a record in the file. Each record contains Name, Address, Event, and other fields to hold the data.

Though relatively few people still work with flat files at this level, the terms file, record, and field are still with us and are often used in database documentation and discussions.

Relations, Attributes, and Tuples

The values in a row are *related* by the fact that they apply to a particular person. Because of this fact, the formal term for a table is a *relation*. This can cause some confusion because the word ''relation'' is also used informally to describe a relationship between two tables. This use is described in the section ''Foreign Key Constraints'' later in this chapter.

The formal term for a column is an *attribute* or *data element*. For example, in the Competitors relation shown in Figure 3-1, Name, Address, BloodType, and NextOfKin are the attributes of each of the people represented. You can think of this as in: ''each person in the relation has a Name attribute.''

The formal term for a row is a *tuple* (rhymes with ''scruple''). This almost makes sense if you think of a two-attribute relation as holding data pairs, a three-attribute relation as holding value triples, and a four-attribute relation as holding data quadruples. Beyond four items, mathematicians would say 5-tuple, 6-tuple, and so forth, hence the name tuple.

Don't confuse the formal term *relation* (meaning table) with the more general and less precise use of the term that means ''related to'' as in ''these fields form a relation between these two tables'' (or ''that psycho is no relation of mine''). Similarly, don't confuse the formal term *attribute* with the less precise use that means ''feature of'' as in ''this field has the 'required' attribute'' (or ''don't attribute that comment to me!''). I doubt you'll confuse the term *tuple* with anything — it's probably confusing enough all by itself.

Theoretically a relation does not impose any ordering on the tuples that it contains nor does it give an ordering to its attributes. Generally the orderings don't matter to mathematical database theory. In practice, however, database applications usually sort the records selected from a table in some manner to make it easier for the user to understand the results. It's also a lot easier to write the program (and for the user to understand) if the order of the fields remains constant, so database products typically return fields in the order in which they were created in the table unless told otherwise.

Keys

Relational database terminology includes an abundance of different flavors of keys. In the loosest sense, a key is a combination of one or more columns that you use to find rows in a table. For example, a Customers table might use CustomerID to find customers. If you know a customer's ID, you can quickly find that customer's record in the table. (In fact, many ID numbers, such as employee IDs, student IDs, driver's licenses, and so forth, are invented just to make searching in database tables easier. My library card certainly doesn't include a 10-character ID number for my convenience.)

The more formal relational vocabulary includes several other more precise definitions of keys.

In general, a key is a set of one or more columns in the table that have certain properties. A *compound key* or *composite key* is a key that includes more than one column. For example, you might use the combination of FirstName and LastName to look up customers.

A *superkey* is a set of one or more columns in a table for which no two rows can have the exact same values. For example, in the Competitors table shown in Figure 3-1, the Name, Address, and Event columns together form a superkey because no two rows have exactly the same Name, Address, and Event values. Because superkeys define fields that must be unique within a table, they are sometimes called *unique keys*.

Because no two rows in the table have the same values for a superkey, a superkey can uniquely identify a particular row in the table. In other words, a program could use a superkey to find any particular record.

A *candidate key* is a minimal superkey. That means if you remove any of the columns from the superkey, it won't be a superkey anymore.

For example, you already know that Name/Address/Event is a superkey for the Competitors table. If you remove Event from the superkey, Name/Address is not a superkey because everyone in the table is participating in multiple events so they have more than one record with the same name and address.

If you remove Name, Address/Event is not a superkey because Dean Daring and his roommate George Foreman share the same address and are both signed up for Pyramid Luge. (They also have the same blood type. They became friends and decided to become roommates when Dean donated blood for George after a particularly flamboyant skateboarding accident.)

Finally if you remove Address, Name/Event is still a superkey. That means Name/Address/Event is not a candidate key because it is not minimal. However, Name/Event is a candidate key because no two rows have the same Name/Event values and you can easily see neither Name nor Event is a superkey, so the pair is minimal.

You could still have a problem if one of George's other brothers, who are all named George, moves in. If they compete in the same event, you won't be able to tell them apart. Perhaps we should add a CompetitorId column to the table after all.

Note that there may be more than one superkey or candidate key in a table. In Figure 3-1, Event/NextOfKin also forms a candidate key because no two rows have the same Event and NextOfKin values. (That would probably not be the most natural way to look up rows, however. ''Yes sir, I can look up your record if you give me your event and next of kin.'')

A *unique key* is a superkey that is used to uniquely identify the rows in a table. The difference between a unique key and any other candidate key is in how it is used. A candidate key *could* be used to identify rows if you wanted it to, but a unique key *is* used to constrain the data. In this example, if you make Name/Event be a unique key, the database will not allow you to add two rows with the same Name and Event values. A unique key is an implementation issue, not a more theoretical concept like a candidate key is.

A *primary key* is a superkey that is actually used to uniquely identify or find the rows in a table. A table can have only one primary key (hence the name ''primary''). Again, this is more of an implementation issue than a theoretical concern. Database products generally take special action to make finding records based on their primary keys faster than finding records based on other keys.

Some databases allow alternate key fields to have missing values, whereas all of the fields in a primary key are required. For example, the Competitors table might have Name/Address/Event as a unique key and Name/Event as a primary key. Then it could contain a record with Name and Event but no Address value. (Although that would be a bit strange. We might want to require that all of the fields have a value.)

An *alternate key* is a candidate key that is not the primary key. Some also call this a *secondary key*, although others use the term secondary key to mean any set of fields used to locate records even if the fields don't define unique values.

That's a lot of keys to try to remember! The following list briefly summarizes the different flavors:

- **Compound key or composite key:** A key that includes more than one field.
- **Superkey:** A set of columns for which no two rows can have the exact same values.
- **Candidate key:** A minimal superkey.
- **Unique key:** A superkey used to require uniqueness by the database.
- **Primary key:** A unique key that is used to quickly locate records by the database.
- **Alternate key:** A candidate key that is not the primary key.
- **Secondary key:** A key used to look up records but that may not guarantee uniqueness.

One last kind of key is the *foreign key*. A foreign key is used as a constraint rather than to find records in a table, so it is described a bit later in the section ''Constraints.''

Indexes

An index is a database structure that makes it quicker and easier to find records based on the values in one or more fields. Indexes are not the same as keys, although the two are related closely enough that many developers confuse the two and use the terms interchangeably.

For example, suppose you have a Customers table that holds customer information: name, address, phone number, Swiss bank account number, and so forth. The table also contains a CustomerId field that it uses as its primary key.

Unfortunately customers usually don't remember their customer IDs, so you need to be able to look them up by name or phone number. If you make Name and PhoneNumber be two different keys, you can quickly locate a customer's record in three ways: by customer ID, by name, and by phone number.

Relational databases also make it easy to look up records based on non-indexed fields, although it may take a while. If the customer only remembers his address and not his customer ID or name, you can search for the address even if it that field isn't part of an index. It may just take a long time. Of course if the customer cannot remember his name, he's got bigger problems.

Building and maintaining an index takes the database some extra time, so you shouldn't make indexes gratuitously. Place indexes on the fields that you are most likely to need to search and don't bother indexing fields such as apartment number that you are unlikely to need to search.

Constraints

As you might guess from the name, a *constraint* places restrictions on the data allowed in a table. In formal database theory, constraints are not considered part of the database. However, in practice constraints play such a critical role in managing the data properly that they are informally considered part of the database. (Besides, the database product enforces them!)

The following sections describe some of the kinds of constraints that you can place on the fields in a table.

Basic Constraints

Relational databases let you specify some simple basic constraints on a particular field. For example, you can make a field required. The special value *null* represents an empty value. For example, suppose you don't know a customer's income. You can place the value null in the Income field to indicate that you don't know the correct value. This is different from placing 0 in the field, which would indicate that the customer doesn't have any income.

Making a field required means it cannot hold a null value, so this is also called a *not null* constraint.

The database will also prevent a field from holding a value that does not match its data type. For example, you cannot put a 20-character string in a 10-character field. Similarly, you cannot store the value ''twelve'' in a field that holds integers.

These types of constraints restrict the values that you can enter into a field. They help define the field's domain so they are called *domain constraints*. Some database products allow you to define more complex domain constraints, often by using check constraints.

Check Constraints

A *check constraint* is a more complicated type of restriction that evaluates a Boolean expression to see if certain data should be allowed. If the expression evaluates to true, the data is allowed.

A *field-level* check constraint validates a single column. For example, in a SalesPeople table you could place the constraint `Salary > 0` on the Salary field to mean that the field's value must be positive.

A *table-level* check constraint can examine more than one of a record's fields to see if the data is valid. For example, the constraint `(Salary > 0) OR (Commission > 0)` requires that each SalesPeople record have a positive salary or a positive commission (or both).

Primary Key Constraints

By definition, no two records can have identical values for the fields that define the table's primary key. That greatly constrains the data.

In more formal terms, this type of constraint is called *entity integrity*. It simply means that no two records are exact duplicates (which is true if the fields in their primary keys are not duplicates) and that all of the fields that make up the primary key have non-null values.

Unique Constraints

A *unique constraint* requires that the values in one or more fields be unique. Note that it only makes sense to place a uniqueness constraint on a superkey. Recall that a superkey is a group of one or more fields that cannot contain duplicate values. It wouldn't make sense to place a uniqueness constraint on fields that can validly contain duplicated values. For example, it would be silly to place a uniqueness constraint on a Gender field.

Foreign Key Constraints

A *foreign key* is not quite the same kind of key defined previously. Instead of defining fields that you use to locate records, a foreign key refers to a key in another (foreign) table. The database uses it to locate records in the other table but you don't. Because it defines a reference from one table to another, this kind of constraint is also called a *referential integrity constraint*.

A foreign key constraint requires that a record's values in one or more fields in one table (the referencing table) must match the values in another table (the foreign or referenced table). The fields in the referenced table must form a candidate key in that table. Usually they are that table's primary key, and most database products try to use the foreign table's primary key by default when you make a foreign key constraint.

For a simple example, suppose you want to validate the entries in the Competitors table's Event field so the minimum wage interns manning the phones cannot assign anyone to an event that doesn't exist.

To do this with a foreign key, create a new table named Events that has a single column called Event. Make this the new table's primary key and make records that list the possible events: Pyramid Boarding, Pyramid Luge, Camel Drafting, and Sphinx Jumping.

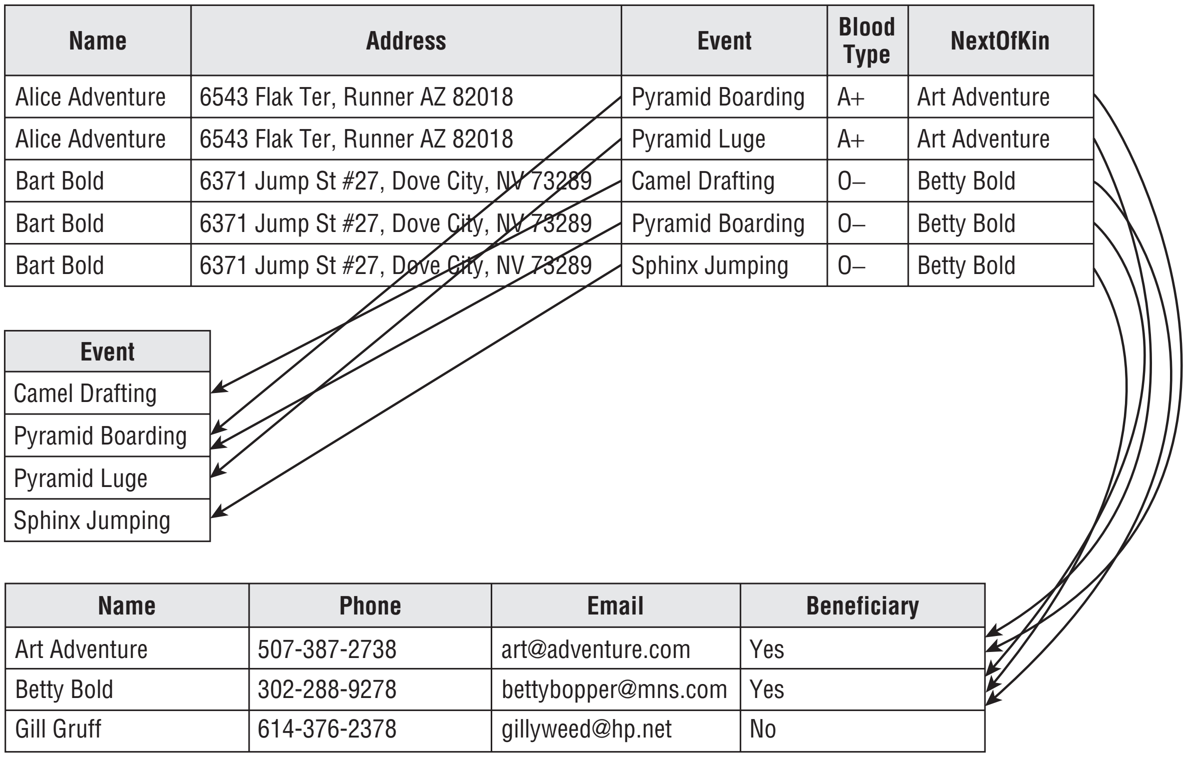
Next, make a foreign key that relates the Competitors table's Event field with the Events table's Event field. Now whenever someone adds a new record to the Competitors table, the foreign key constraint will require that the new record's Event value be listed in the Events table.

The database will also ensure that no one modifies a Competitors record to change the Event value to something that is not in the Events table.

Finally, the database will take special action if you try to delete a record in the Events table if its value is being used by a Competitors record. Depending on the type of database and how you have the relationship configured, the database will either refuse to remove the Events record or it will automatically delete all of the Competitors records that use it.

This example uses the Events table as a lookup table for the Competitors table. Another common use for foreign key constraints is to make sure related records always go together. For example, you could build a NextOfKin table that contains information about the competitors' next of kin (name, phone number, email address, beneficiary status, and so forth). Then you could make a foreign key constraint to ensure that every Competitor record's NextOfKin value is contained in the Name fields in some NextOfKin table record. That way you know that you can always contact the next of kin for anyone in the Competitors table.

Figure 3-2 shows the Competitors, Events, and NextOfKin tables with lines showing the relationships among their related fields.



Name	Address	Event	Blood Type	NextOfKin
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Boarding	A+	Art Adventure
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Luge	A+	Art Adventure
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Camel Drafting	O–	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Pyramid Boarding	O–	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Sphinx Jumping	O–	Betty Bold

Event
Camel Drafting
Pyramid Boarding
Pyramid Luge
Sphinx Jumping

Name	Phone	Email	Beneficiary
Art Adventure	507-387-2738	art@adventure.com	Yes
Betty Bold	302-288-9278	bettybopper@mns.com	Yes
Gill Gruff	614-376-2378	gillyweed@hp.net	No

Figure 3-2

Foreign keys define associations between tables that are sometimes called *relations*, *relationships*, or *links* between the tables. The fact that the formal database vocabulary uses the word *relation* to mean table sometimes leads to confusion. Fortunately, the formal and informal database people usually get invited to different parties so the terms usually don't collide in the same conversation.

Database Operations

The final topic in this chapter covers database operations. (I'll save the rest so I have something for the rest of the book.)

Eight operations were originally defined for relational databases and they form the core of modern database operations. The following list describes those original operations:

- **Selection:** This selects some or all of the records in a table. For example, you might want to select only the Competitors records where Event is Pyramid Luge so you can know who to expect for that event (and how many ambulances to have standing by).
- **Projection:** This drops columns from a table (or selection). For example, when you make your list of Pyramid Luge competitors you may only want to list their names and not their addresses, blood types, events (which you know is Pyramid Luge anyway), or next of kin.

- **Union:** This combines tables with similar columns and removes duplicates. For example, suppose you have another table named FormerCompetitors that contains data for people who participated in previous years' competitions. Some of these people are competing this year and some are not. You could use the union operator to build a list of everyone in either table. (Note that the operation would remove duplicates, but for these tables you would still get the same person several times with different events.)
- **Intersection:** This finds the records that are the same in two tables. The intersection of the FormerCompetitors and Competitors tables would list those few who competed in previous years and who survived to compete again this year (the slow learners).
- **Difference:** This selects the records in one table that are not in a second table. For example, the difference between FormerCompetitors and Competitors would give you a list of those who competed in previous years but who are not competing this year (so you can email them and ask them what the problem is).
- **Cartesian Product:** This creates a new table containing every record in a first table combined with every record in a second table. For example, if one table contains values 1, 2, 3 and a second table contains values A, B, C, then their Cartesian product contains the values 1/A, 1/B, 1/C, 2/A, 2/B, 2/C, 3/A, 3/B, and 3/C.
- **Join:** This is similar to a Cartesian product except records in one table are paired only with those in the second table if they meet some condition. For example, you might join the Competitors records with the NextOfKin records where a Competitors record's NextOfKin value matches the NextOfKin record's Name value. In this example, that gives you a list of the competitors together with their corresponding next of kin data.
- **Divide:** This operation is the opposite of the Cartesian product. It uses one table to partition the records in another table. It finds all of the field values in one table that are associated with every value in another table. For example, if the first table contains the values 1/A, 1/B, 1/C, 2/A, 2/B, 2/C, 3/A, 3/B, and 3/C and a second table contains the values 1, 2, 3, then the first divided by the second gives A, B, C. (Don't worry, I think it's pretty weird and confusing, too, so it won't be on the final exam. Probably.)

The workhorse operation of the relational database is the join, often combined with selection and projection. For example, you could *join* Competitors records with NextOfKin records that have the correct name. Next you could *project* to select only the competitors' names, the next of kin names, and the next of kin phone numbers. You could then *select* only Bart Bold's records. Finally, you could *select* for unique records so the result would contain only a single record containing the values Bart Bold, Betty Bold, 302-288-9278.

The following SQL query produces this result:

```
SELECT DISTINCT Competitors.Name, NextOfKin.Name, Phone
FROM Competitors, NextOfKin
WHERE Competitors.NextOfKin = NextOfKin.Name
  AND Competitors.Name = 'Bart Bold'
```

The `SELECT` clause performs selection, the `FROM` clause tells which tables to join, the first part of the `WHERE` clause (`Competitors.NextOfKin = NextOfKin.Name`) gives the join condition, the second part of the `WHERE` clause (`Competitors.Name = 'Bart Bold'`) selects only Bart's records, and the `DISTINCT` keyword selects unique results.

The results of these operations are table-like objects that aren't permanently stored in the database. They have rows and columns so they look like tables, but their values are generated on the fly when the database operations are executed. These result objects are called *views*. Because they are often generated by SQL queries, they are also called *query results*. Because they look like tables that are generated as needed, they are sometimes called *virtual tables*.

Chapter 20 has more to say about relational database operations as they are implemented in practice.

Summary

Before you can start designing and building relational databases, you need to understand some of the basics. This chapter provided an introduction to relational databases and their terminology.

In this chapter you learned about:

- ❑ Formal relational database terms such as relation, attribute, and tuple.
- ❑ Informal terms such as table, row, record, column, and field.
- ❑ Several kinds of keys including superkeys, candidate keys, and primary keys.
- ❑ Different kinds of constraints that you can place on columns or tables.
- ❑ Operations defined for relational databases.

The following chapters change the book's focus from general database concepts and terminology to design techniques. They describe the tasks you must perform to design a database from scratch. Chapter 4 starts the process by explaining how to gather user requirements so the database you design has a good chance of actually satisfying the users' needs.

Before you move on to Chapter 4, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

Exercises

1. What does the following check constraint on the SalesPeople table mean?

```
((Salary > 0) AND (Commission = 0)) OR ((Salary = 0) AND (Commission > 0))
```

2. In Figure 3-3, draw lines connecting the corresponding terms.

Attribute	Row	File
Relation	Column	Relationship
Foreign Key	Table	Virtual Table
Tuple	Foreign Key	Record
View	Query Result	Field

Figure 3-3

For questions 3 through 6, suppose you're a numismatist and you want to track your progress in collecting the 50 state quarters created by the United States Mint. You start with the following table and plan to add more data later (after you take out the trash and finish painting your lead miniatures).

State	Abbr	Title	Engraver	Year	Got
Arizona	AZ	Grand Canyon State	Joseph Menna	2008	No
Idaho	ID	Esto Perpetua	Norm Nemeth	2007	No
Iowa	IA	Foundation in Education	John Mercanti	2004	Yes
Michigan	MI	Great Lakes State	Donna Weaver	2004	Yes
Montana	MT	Big Sky Country	Don Everhart	2007	No
Nebraska	NE	Chimney Rock	Charles Vickers	2006	Yes
Oklahoma	OK	Scissortail Flycatcher	Phebe Hemphill	2008	No
Oregon	OR	Crater Lake	Charles Vickers	2005	Yes

3. Is State/Abbr/Title a superkey? Why or why not?
4. Is Engraver/Year/Got a superkey? Why or why not?
5. What are all of the candidate keys for this table?
6. What are the domains of each of the table's columns?

For questions 7 through 10, suppose you are building a dorm room database. Consider the following table. For obscure historical reasons, all of the rooms in the building have even numbers. The Phone field refers to the number of the phone in the room. Rooms that have no phone cost less but students in those rooms are required to have a cell phone (so you can call them and nag if they miss too many classes).

Room	FirstName	LastName	Phone	CellPhone
100	John	Smith	Null	202-837-2897
100	Mark	Garcia	Null	504-298-0281
102	Anne	Johansson	202-237-2102	Null
102	Sally	Helper	202-237-2102	Null
104	John	Smith	202-237-1278	720-387-3928
106	Anne	Uumellmahaye	Null	504-298-0281
106	Wendy	Garcia	Null	202-839-3920
202	Mike	Hfuhruhurr	202-237-7364	Null
202	Jose	Johansson	202-237-7364	202-839-3920

7. If you don't allow two people with the same name to share a room (due to administrative whimsy), what are all of the possible candidate keys for this table?
8. If you do allow two people with the same name to share a room, what are all of the possible candidate keys for this table?
9. What field-level check constraints could you put on this table's fields? Don't worry about the syntax for performing the checks, just define them.
10. What table-level check constraints could you put on this table's fields? Don't worry about the syntax for performing the checks, just define them.