The Database Relational Model

1. Outlines

- Relational model basics
- Integrity rules
- Rules about referenced rows
- Relational Algebra

2. History of the Relational Model

The relational data model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper (Codd, 1970), and it attracted immediate attention due to its simplicity and mathematical foundation. The model uses the concept of a *mathematical relation*—which looks somewhat like a table of values—as its basic building block, and has its theoretical basis in set theory and first-order predicate logic. In this chapter we discuss the basic characteristics of the model and its constraints.

The first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system on the MVS operating system by IBM and the Oracle DBMS. Since then, the model has been implemented in a large number of commercial systems, as well as a number of open source systems. Current popular commercial relational DBMSs (RDBMSs) include DB2 (from IBM), Oracle (from Oracle), Sybase DBMS (now from SAP), and SQLServer and Microsoft Access (from Microsoft). In addition, several open source systems, such as MySQL and PostgreSQL, are available.

3. Relational Model Terminology

The relational model represents the database as a collection of ***relations.*** Informally, each relation resembles a table of values or, to some extent, a *flat* file of records. It is called a **flat file** because each record has a simple linear or *flat* structure. For example, the database of files that was shown in Figure 1.2 is similar to the basic relational model representation. However, there are important differences between relations and files, as we shall soon see. When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row. For example, the first table of Figure 1.2 is called STUDENT because each row represents facts about a particular student entity. The column names—Name, Student_number, Class, and Major—specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a ***tuple,*** a column header is called an ***attribute,*** and the table is called a *relation.* The data type describing the types of values that can appear in each column is represented by a ***domain*** of possible values. We now define these terms—*domain, tuple, attribute,* and *relation*—formally.

4. Domains, Attributes, Tuples, and Relations

A **domain** *D* is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values. Some examples of domains follow:

- Usa_phone_numbers. The set of ten-digit phone numbers valid in the United States.
- Local_phone_numbers. The set of seven-digit phone numbers valid within a particular area code in the United States. The use of local phone numbers is quickly becoming obsolete, being replaced by standard ten-digit numbers.
- Social_security_numbers. The set of valid nine-digit Social Security numbers. (This is a unique identifier assigned to each person in the United States for employment, tax, and benefits purposes.)
- Names: The set of character strings that represent names of persons.
- Grade_point_averages. Possible values of computed grade point averages; each must be a real (floating-point) number between 0 and 4.
- Employee_ages. Possible ages of employees in a company; each must be an integer value between 15 and 80.
- Academic_department_names. The set of academic department names in a university, such as Computer Science, Economics, and Physics.
- Academic_department_codes. The set of academic department codes, such as 'CS', 'ECON', and 'PHYS'.

The preceding are called *logical* definitions of domains. A **data type** or **format** is also specified for each domain. For example, the data type for the domain Usa_phone_numbers can be declared as a character string of the form (*ddd*)*ddd*-*dddd*, where each *d* is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for Employee_ages is an integer number between 15 and 80. For Academic_department_names, the data type is the set of all character strings that represent valid department names. A domain is thus given a name, data type, and format. Additional information for interpreting the values of a domain can also be given; for example, a numeric domain such as Person_weights should have the units of measurement, such as pounds or kilograms.

A **relation schema**[2] R, denoted by $R(A_1, A_2, \ldots, A_n)$, is made up of a relation name R and a list of attributes, $A_1, A_2, \ldots, A_n$. Each **attribute** A_i is the name of a role played by some domain D in the relation schema R. D is called the **domain** of A_i and is denoted by **dom**(A_i). A relation schema is used to *describe* a relation; R is called the **name** of this relation. The ***degree*** (or **arity**) of a relation is the number of attributes n of its relation schema. A relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:

STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)

Using the data type of each attribute, the definition is sometimes written as:

STUDENT(Name: string, Ssn: string, Home_phone: string, Address: string, Office_phone: string, Age: integer, Gpa: real)

For this relation schema, STUDENT is the name of the relation, which has seven attributes. In the preceding definition, we showed assignment of generic types such as string or integer to the attributes. More precisely, we can specify the following previously defined domains for some of the attributes of the STUDENT relation:

dom(Name) = Names;

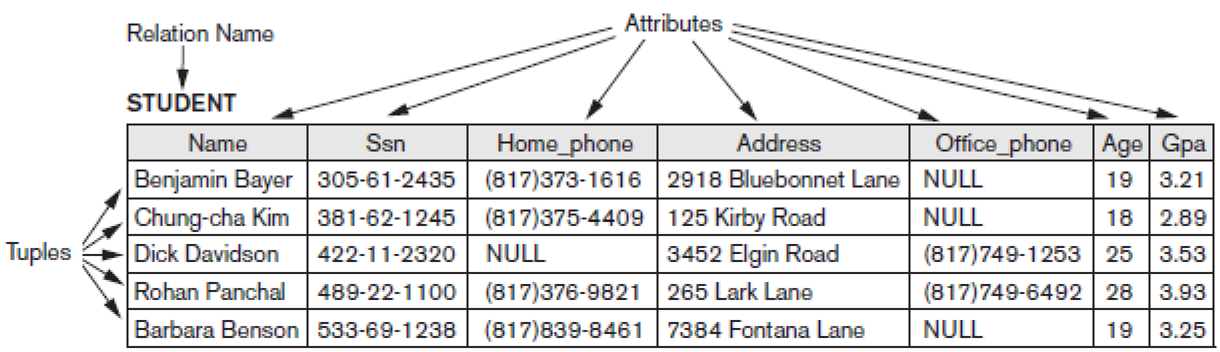
dom(Ssn) = Social_security_numbers;

dom(HomePhone) = USA_phone_numbers[3], dom(Office_phone) = USA_phone_numbers, and dom(Gpa) = Grade_point_averages.

It is also possible to refer to attributes of a relation schema by their position within the relation; thus, the second attribute of the STUDENT relation is Ssn, whereas the fourth attribute is Address.

A **relation** (or **relation state**)[4] r of the relation schema $R(A_1, A_2, \dots, A_n)$, also denoted by $r(R)$, is a set of n-tuples $r = \{t_1, t_2, \dots, t_m\}$. Each **$n$-tuple** t is an ordered list of n values $t = <v_1, v_2, \dots, v_n>$, where each value v_i, $1 \delta i \delta n$, is an element of dom (A_i) or is a special NULL value. (NULL values are discussed further below and in Section 5.1.2.) The ith value in tuple t, which corresponds to the attribute A_i, is referred to as $t[A_i]$ or $t.A_i$ (or $t[i]$ if we use the positional notation). The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used.

Figure 5.1 shows an example of a STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity (or object). We display the relation as a table, where each tuple is shown as a *row* and each attribute corresponds to a *column header* indicating a role or interpretation of the values in that column. *NULL values* represent attributes whose values are unknown or do not exist for some individual STUDENT tuple.



Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25

Figure 5.1
The attributes and tuples of a relation STUDENT.

The earlier definition of a relation can be *restated* more formally using set theory concepts as follows. A relation (or relation state) $r(R)$ is a **mathematical relation** of degree n on the domains dom(A_1), dom(A_2), … , dom(A_n), which is a **subset** of the **Cartesian product** (denoted by ×) of the domains that define R: $r(R)$ □ (dom(A_1) · dom(A_2) · . . . · (dom(A_n))

The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the total number of values, or ***cardinality***, in a domain D by $|D|$ (assuming that all domains are finite), the total number of tuples in the Cartesian product is $|dom(A_1)| \cdot |dom(A_2)| \cdot \dots \cdot |dom(A_n)|$

This product of cardinalities of all domains represents the total number of possible instances or tuples that can ever exist in any relation state $r(R)$. Of all these possible combinations, a relation state at a given time—the **current relation state**—reflects only the valid tuples that represent a particular state of the real world. In general, as the state of the real-world changes, so does the relation state, by being transformed into another relation state. However, the schema R is relatively static and changes *very* infrequently—for example, as a result of adding an attribute to represent new information that was not originally stored in the relation.

It is possible for several attributes to *have the same domain.* The attribute names indicate different **roles**, or interpretations, for the domain. For example, in the STUDENT relation, the same domain USA_phone_numbers play the role of Home_phone, referring to the *home phone of a student*, and the role of Office_phone, referring to the *office phone of the student*. A third possible attribute (not shown) with the same domain could be Mobile_phone.

5.1.2 Characteristics of Relations

The earlier definition of relations implies certain characteristics that make a relation different from a file or a table. We now discuss some of these characteristics. **Ordering of Tuples in a Relation.** A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. In other words, a relation is not sensitive to the ordering of tuples. However, in a file, records are physically stored on disk (or in memory), so there always is an order among the records. This ordering indicates first, second, *i*th, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.

Tuple ordering is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level. Many tuple orders can be specified on the same relation. For example, tuples in the STUDENT relation in Figure 5.1 could be ordered by values of Name, Ssn, Age, or some other attribute. The definition of a relation does not specify any order: There is *no preference* for one ordering over another. Hence, the relation displayed in Figure 5.2 is considered *identical* to the one shown in Figure 5.1. When a relation is implemented as a file or displayed as a table, a particular ordering may be specified on the records of the file or the rows of the table.

Ordering of Values within a Tuple and an Alternative Definition of a Relation.
According to the preceding definition of a relation, an n-tuple is an *ordered list* of n values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important. However, at a more abstract level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained.

An **alternative definition** of a relation can be given, making the ordering of values in a tuple *unnecessary.* In this definition, a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a *set* of attributes (instead of an ordered list of attributes), and a relation state $r(R)$ is a finite set of mappings $r = \{t_1, t_2, \ldots, t_m\}$, where each tuple t_i is a **mapping** from R to D, and D is the **union** (denoted by $*$) of the attribute domains; that is, $D = \text{dom}(A_1) * \text{dom}(A_2) * \ldots * \text{dom}(A_n)$. In this definition, $t[A_i]$ must be in $\text{dom}(A_i)$ for $1 \,\delta\, i \,\delta\, n$ for each mapping t in r. Each mapping t_i is called a tuple.

According to this definition of tuple as a mapping, a ***tuple*** can be considered as a **set** of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from dom(A_i). The ordering of attributes is *not* important, because the *attribute name* appears with its *value*. By this definition, the two tuples shown in Figure 5.3 are identical. This makes sense at an abstract level, since there really is no reason to prefer having one attribute value appear before another in a tuple. When the attribute name and value are included together in a tuple, it is known as **self-describing data**, because the description of each value (attribute name) is included in the tuple.

Figure 5.2
The relation STUDENT from Figure 5.1 with a different order of tuples.

STUDENT

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21

t = < (Name, Dick Davidson),(Ssn, 422-11-2320),(Home_phone, NULL),(Address, 3452 Elgin Road),
(Office_phone, (817)749-1253),(Age, 25),(Gpa, 3.53)>

t = < (Address, 3452 Elgin Road),(Name, Dick Davidson),(Ssn, 422-11-2320),(Age, 25),
(Office_phone, (817)749-1253),(Gpa, 3.53),(Home_phone, NULL)>

Figure 5.3
Two identical tuples when the order of attributes and values is not part of relation definition.

We will mostly use the **first definition** of relation, where the attributes are *ordered* in the relation schema and the values within tuples *are similarly ordered,* because it simplifies much of the notation. However, the alternative definition given here is more general.[5] **Values and NULLs in the Tuples.** Each value in a tuple is an ***atomic*** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes (see Chapter 3) are not allowed. This model is sometimes called the **flat relational model**. Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption.[6] Hence, multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes in the basic relational model.[7] An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases. For example, in Figure 5.1, some STUDENT tuples have NULL for their office phones because they do not have an office (that is, office phone *does not apply* to these students). Another student has a NULL for home phone, presumably because either he does not have a home phone or he has one but we do not know it (value is *unknown*). In general, we can have several meanings for NULL values, such as ***value unknown,*** **value** exists but is ***not available***, or ***attribute does not apply*** to this tuple *(*also known as ***value undefined****)*. An example of the last type of NULL will occur if we add an attribute Visa_status to the STUDENT relation that applies only to tuples representing foreign students. It is possible to devise different codes for different meanings of NULL values. Incorporating different types of NULL values into relational model operations has proven difficult and is outside the scope of our presentation.

The exact meaning of a NULL value governs how it fares during arithmetic aggregations or comparisons with other values. For example, a comparison of two NULL values leads to ambiguities—if both Customer A and B have NULL addresses, it *does not mean* they have the same address. During database design, it is best to avoid NULL values as much as possible. We will discuss this further in Chapters 7 and 8 in the context of operations and queries, and in Chapter 14 in the context of database design and normalization.

Interpretation (Meaning) of a Relation. The relation schema can be interpreted as a declaration or a type of **assertion**. For example, the schema of the STUDENT relation of Figure 5.1 asserts that, in general, a student entity has a Name, Ssn, Home_phone, Address, Office_phone, Age, and Gpa. Each tuple in the relation can then be interpreted as a ***fact*** or a particular instance of the assertion. For example, the first tuple in Figure 5.1 asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on.

Notice that some relations may represent facts about ***entities,*** whereas other relations may represent facts about ***relationships****.* For example, a relation schema MAJORS (Student_ssn, Department_code) asserts that students major in academic disciplines. A tuple in this relation relates a student to his or her major discipline.

Hence, the relational model represents facts about both entities and relationships *uniformly* as relations. This sometimes compromises understandability because one has to guess whether a relation represents an entity type or a relationship type. We introduced the entity–relationship (ER) model in detail in Chapter 3, where the entity and relationship concepts were described in detail. The mapping procedures in Chapter 9 show how different constructs of the ER/EER conceptual data models (see Part 2) get converted to relations.

An alternative interpretation of a relation schema is as a **predicate**; in this case, the values in each tuple are interpreted as values that *satisfy* the predicate. For example, the predicate STUDENT (Name, Ssn, …) is true for the five tuples in relation STUDENT of Figure 5.1. These tuples represent five different propositions or facts in the real world. This interpretation is quite useful in the context of logical programming languages, such as Prolog, because it allows the relational model to be used within these languages (see Section 26.5). An assumption called **the closed world assumption** states that the only true facts in the universe are those present within the extension (state) of the relation(s). Any other combination of values makes the predicate false. This interpretation is useful when we consider queries on relations based on relational calculus in Section 8.6.

5.1.3 Relational Model Notation

We will use the following notation in our presentation:

- A relation schema R of degree n is denoted by $R(A_1, A_2, \ldots, A_n)$.
- The uppercase letters Q, R, S denote relation names.

- The lowercase letters q, r, s denote relation states.
- The letters t, u, v denote tuples.
- In general, the name of a relation schema such as STUDENT also indicates the current set of tuples in that relation—the *current relation state*—whereas STUDENT(Name, Ssn, …) refers *only* to the relation schema.
- An attribute A can be qualified with the relation name R to which it belongs by using the dot notation $R.A$—for example, STUDENT.Name or STUDENT.Age. This is because the same name may be used for two attributes in different relations. However, all attribute names *in a particular relation* must be distinct.
- An n-tuple t in a relation $r(R)$ is denoted by $t = <v_1, v_2, \ldots, v_n>$, where v_i is the value corresponding to attribute A_i. The following notation refers to **component values** of tuples:
  - Both $t[A_i]$ and $t.A_i$ (and sometimes $t[i]$) refer to the value v_i in t for attribute A_i.
  - Both $t[A_u, A_w, \ldots, A_z]$ and $t.(A_u, A_w, \ldots, A_z)$, where $A_u, A_w, \ldots, A_z$ is a list of attributes from R, refer to the subtuple of values $<v_u, v_w, \ldots, v_z>$ from t corresponding to the attributes specified in the list.

As an example, consider the tuple t = <'Barbara Benson', '533-69-1238', '(817)839-8461', '7384 Fontana Lane', NULL, 19, 3.25> from the STUDENT relation in Figure 5.1; we have t[Name] = <'Barbara Benson'>, and t[Ssn, Gpa, Age] = <'533-69-1238', 3.25, 19>.

5.2 Relational Model Constraints and Relational Database Schemas

So far, we have discussed the characteristics of single relations. In a relational database, there will typically be many relations, and the tuples in those relations are usually related in various ways. The state of the whole database will correspond to the states of all its relations at a particular point in time. There are generally many restrictions or **constraints** on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents, as we discussed in Section 1.6.8. In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints. Constraints on databases can generally be divided into three main categories:

a) Constraints that are inherent in the data model. We call these **inherent model-based constraints** or **implicit constraints**.

b) Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL (data definition language, see Section 2.3.1). We call these **schema-based constraints** or **explicit constraints**.

c) Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs or in some other way. We call these **application-based** or **semantic constraints** or **business rules**.

The characteristics of relations that we discussed in Section 5.1.2 are the inheren constraints of the relational model and belong to the first category. For example, the constraint that a relation cannot have duplicate tuples is an inherent constraint. The constraints we discuss in this section are of the second category, namely, constraints that can be expressed in the schema of the relational model via the DDL. Constraints in the third category are more general, relate to the meaning as well as behavior of attributes, and are difficult to express and enforce within the data model, so they are usually checked within the application programs that perform database updates. In some cases, these constraints can be specified as **assertions** in SQL (see Chapter 7).

Another important category of constraints is *data dependencies*, which include *functional dependencies* and *multivalued dependencies*. They are used mainly for testing the "goodness" of the design of a relational database and are utilized in a process called *normalization*, which is discussed in Chapters 14 and 15. The schema-based constraints include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

5.2.1 Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute *A* must be an atomic value from the domain dom(*A*). We have already discussed the ways in which domains can be specified in Section 5.1.1. The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double-precision float).

Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and other special data types. Domains can also be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed. Rather than describe these in detail here, we discuss the data types offered by the SQL relational standard in Section 6.1.

5.2.2 Key Constraints and Constraints on NULL Values

In the formal relational model, a *relation* is defined as a *set of tuples.* By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema *R* with the property that no two tuples in any relation state *r* of *R* should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples t_1 and t_2 in a relation state *r* of *R*, we have the constraint that:

$t_1[\text{SK}] \quad \bar{t}_2[\text{SK}]$

Any such set of attributes SK is called a **superkey** of the relation schema *R*. A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state *r* of *R* can have the same value for SK. Every relation has at least one default superkey— the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key,* which has no redundancy. A **key** *k* of a relation schema *R* is a superkey of *R* with the additional property that removing any attribute *A* from *K* leaves a set of attributes *K*2 that is not a superkey of *R* any more. Hence, a key satisfies two properties:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This *uniqueness* property also applies to a superkey.
2. It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold. This *minimality* property is required for a key but is optional for a superkey.

Hence, a key is a superkey but not vice versa. A superkey may be a key (if it is minimal) or may not be a key (if it is not minimal). Consider the STUDENT relation of Figure 5.1. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn.[8] Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey. However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey. In general, any superkey formed from a single attribute is also a key. A key with multiple attributes must require *all* its attributes together to have the uniqueness property.

The value of a key attribute can be used to identify uniquely each tuple in the relation. For example, the Ssn value 305-61-2435 identifies uniquely the tuple corresponding to Benjamin Bayer in the STUDENT relation. Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on *every* valid relation state of the schema. A key is determined from the meaning of the attributes, and the property is *time-invariant:* It must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the STUDENT relation in Figure 5.1 as a key because it is possible that two students with identical names will exist at some point in a valid state.[9] In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key**. For example, the CAR relation in Figure 5.4 has two candidate keys: License_number and Engine_serial_number. It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined, as shown in Figure 5.4. Notice that when a relation schema has several candidate keys, the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. The other candidate keys are designated as **unique keys** and are not underlined.

CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 5.4
The CAR relation, with two candidate keys: License_number and Engine_serial_number.

Another constraint on attributes specifies whether NULL values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

5.2.3 Relational Databases and Relational Database Schemas

The definitions and constraints we have discussed so far apply to single relations and their attributes. A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section, we define a relational database and a relational database schema.

A **relational database schema** S is a set of relation schemas $S = \{R_1, R_2, \ldots, R_m\}$ and a set of **integrity constraints** IC. A **relational database state**[10] DB of S is a set of relation states DB = $\{r_1, r_2, \ldots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC. Figure 5.5 shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. In each relation schema, the underlined attribute represents the primary key. Figure 5.6 shows a relational database state corresponding to the COMPANY schema. We will use this schema and database state in this chapter and in Chapters 4 through 6 for developing

sample queries in different relational languages. (The data shown here is expanded and available for loading as a populated database from the Companion Website for the text, and can be used for the hands-on project exercises at the end of the chapters.) When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is called **not valid**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state.**

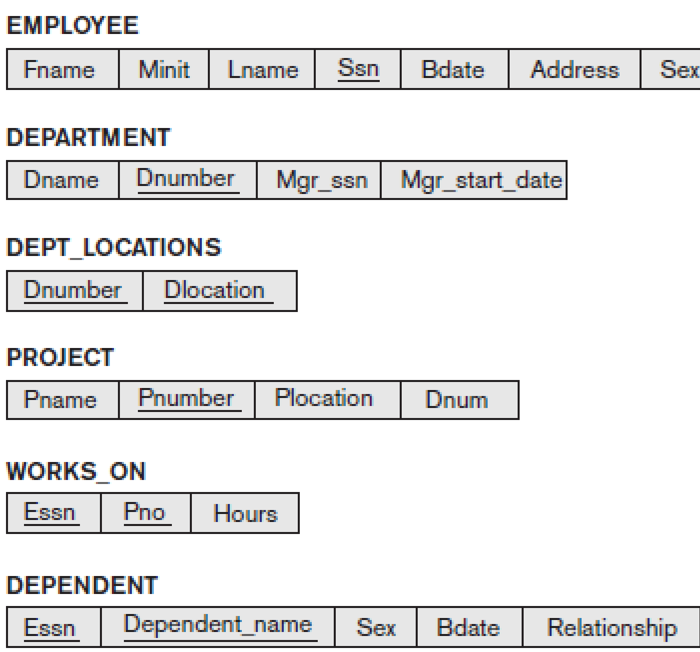


Figure 5.5
Schema diagram for the COMPANY relational database schema.

In Figure 5.5, the Dnumber attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called Dno in EMPLOYEE and Dnum in PROJECT. Attributes that represent the same real-world concept may or may not have identical names in different relations. Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name Name for both Pname of PROJECT and Dname of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different realworld concepts—project names and department names.

In some early versions of the relational model, an assumption was made that the same real-world concept, when represented by an attribute, would have *identical* attribute names in all relations. This creates problems when the same real-world concept is used in different roles (meanings) in the same relation. For example, the concept of Social Security number appears twice in the EMPLOYEE relation of Figure 5.5: once in the role of the employee's SSN, and once in the role of the supervisor's SSN. We are required to give them distinct attribute names—Ssn and Super_ssn, respectively—because they appear in the same relation and in order to distinguish their meaning.

Each relational DBMS must have a data definition language (DDL) for defining a relational database schema. Current relational DBMSs are mostly using SQL for this purpose. We present the SQL DDL in Sections 6.1 and 6.2.

Figure 5.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Integrity constraints are specified on a database schema and are expected to hold on *every valid database state* of that schema. In addition to domain, key, and NOT NULL constraints, two other types of constraints are considered part of the relational model: entity integrity and referential integrity.

5.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation. For example, in Figure 5.6, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

To define *referential integrity* more formally, first we define the concept of a *foreign key.* The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2. A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that **references** relation R_2 if it satisfies the following rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R_2; the attributes FK are said to **reference** or **refer to** the relation R_2.
2. A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ *or is NULL*. In the former case, we have $t_1[\text{FK}] = t_2[\text{PK}]$, and we say that the tuple t_1 **references** or **refers to** the tuple t_2.

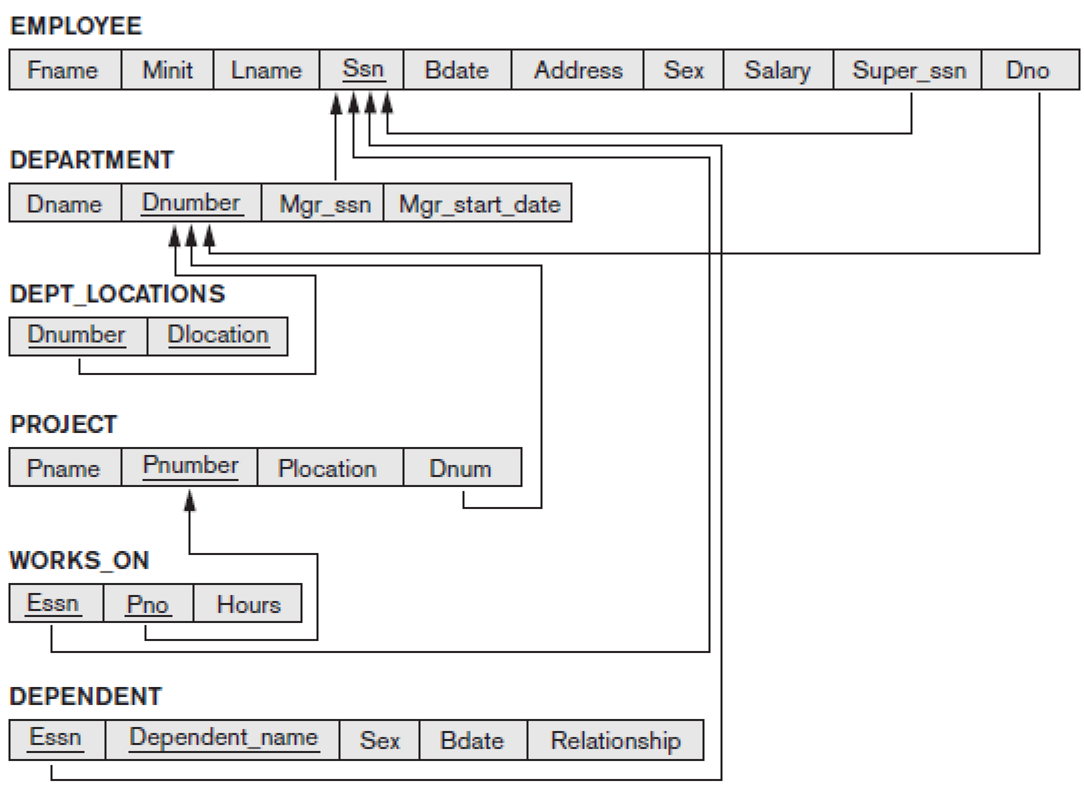
In this definition, R_1 is called the **referencing relation** and R_2 is the **referenced relation**. If these two conditions hold, a **referential integrity constraint** from R_1 to R_2 is said to hold. In a database of many relations, there are usually many referential integrity constraints. To specify these constraints, first we must have a clear understanding of the meaning or role that each attribute or set of attributes plays in the various relation schemas of the database. Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas. For example, consider the database shown in Figure 5.6. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation. This means that a value of Dno in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT—the Dnumber attribute—in some tuple t_2 of the DEPARTMENT relation, or the value of Dno *can be NULL* if the employee does not belong to a department or will be assigned to a department later. For example, in Figure 5.6 the tuple for employee 'John Smith' references the tuple for the 'Research' department, indicating that 'John Smith' works for this department. Notice that a foreign key can *refer to its own relation.* For example, the attribute Super_ssn in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, Super_ssn is a foreign key that references the EMPLOYEE relation itself. In Figure 5.6 the tuple for employee 'John

Smith' references the tuple for employee 'Franklin Wong,' indicating that 'Franklin Wong' is the supervisor of 'John Smith'.

We can *diagrammatically display referential integrity constraints* by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. Figure 5.7 shows the schema in Figure 5.5 with the referential integrity constraints displayed in this manner.

All integrity constraints should be specified on the relational database schema (that is, specified as part of its definition) if we want the DBMS to enforce these constraints on the database states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. In SQL, the CREATE TABLE statement of the SQL DDL allows the definition of primary key, unique key, NOT NULL, entity integrity, and referential integrity constraints, among other constraints (see Sections 6.1 and 6.2) .

Figure 5.7
Referential integrity constraints displayed on the COMPANY relational database schema.



5.2.5 Other Types of Constraints

The preceding integrity constraints are included in the data definition language because they occur in most database applications. Another class of general constraints, sometimes called *semantic integrity constraints,* are not part of the DD and have to be specified and enforced in a different way. Examples of such constraints are *the salary of an employee should not exceed the salary of the employee's supervisor* and *the maximum number of hours an employee can work on all projects per week is 56*. Such constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose **constraint specification language**. Mechanisms called **triggers** and **assertions** can be used in SQL, through the CREATE ASSERTION and CREATE TRIGGER statements, to specify some of these constraints (see Chapter 7). It is more common to check for these types of constraints within the application programs than to use constraint specification languages because the latter are sometimes difficult and complex to use, as we discuss in Section 26.1.

The types of constraints we discussed so far may be called **state constraints** because they define the constraints that a *valid state* of the database must satisfy. Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database.[11] An example of a transition constraint is: "the salary of an employee can only increase." Such constraints are typically enforced by the application programs or specified using active rules and triggers, as we discuss in Section 26.1.

5. 5.3 Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates.* The relational algebra operations, which can be used to specify **retrievals**, are discussed in detail in Chapter 8. A relational algebra expression forms a new relation after applying a number of algebraic operators to an existing set of relations; its main use is for querying a database to retrieve information. The user formulates a query that specifies the data of interest, and a new relation is formed by applying relational operators to retrieve this data. The **result relation** becomes the answer to (or result of) the user's query. Chapter 8 also introduces the language called relational calculus, which is used to define a query declaratively without giving a specific order of operations.

In this section, we concentrate on the database **modification** or **update** operations. There are three basic operations that can change the states of relations in the database:

Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records, respectively. **Insert** is used to insert one or more new tuples in a relation, **Delete** is used to delete tuples, and **Update** (or **Modify**) is used to change the values of some attributes in existing tuples. Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this section we discuss the types of constraints that may be violated by each of these operations and the types of actions that may be taken if an operation causes a violation. We use the database shown in Figure 5.6 for examples and discuss only domain constraints, key constraints, entity integrity constraints, and the referential integrity constraints shown in Figure 5.7. For each type of operation, we give some examples and discuss any constraints that each operation may violate.

5.3.1 The Insert Operation

The **Insert** operation provides a list of attribute values for a new tuple *t* that is to be inserted into a relation *R*. Insert can violate any of the four types of constraints. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type. Key constraints can be violated if a key value in the new tuple *t* already exists in another tuple in the relation *r*(*R*). Entity integrity can be violated if any part of the primary key of the new tuple *t* is NULL. Referential integrity can be violated if the value of any foreign key in *t* refers to a tuple that does not exist in the referenced relation. Here are some examples to illustrate this discussion.

- *Operation*: Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, NULL, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, NULL, 4> into EMPLOYEE.

  Result: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.

- *Operation*: Insert <‘Alicia’, ‘J’, ‘Zelaya’, ‘999887777’, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, ‘987654321’, 4> into EMPLOYEE.

  Result: This insertion violates the key constraint because another tuple with the same Ssn value already exists in the EMPLOYEE relation, and so it is rejected.

- *Operation*: Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, ‘677678989’, ‘1960-04-05’, ‘6357 Windswept, Katy, TX’, F, 28000, ‘987654321’, 7> into EMPLOYEE.

  Result: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.

- *Operation*: Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, ‘677678989’, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, NULL, 4> into EMPLOYEE.

  Result: This insertion satisfies all constraints, so it is acceptable.

If an insertion violates one or more constraints, the default option is to *reject the insertion.* In this case, it would be useful if the DBMS could provide a reason to the user as to why the insertion was rejected. Another option is to attempt to *correct the reason for rejecting the insertion,* but this is *typically not used for violations caused by Insert*; rather, it is used more often in correcting violations for Delete and Update. In the first operation, the DBMS could ask the user to provide a value for Ssn, and could then accept the insertion if a valid Ssn value is provided. In operation 3, the DBMS could either ask the user to change the value of Dno to some valid value (or set it to NULL), or it could ask the user to insert a DEPARTMENT tuple with Dnumber = 7 and could accept the original insertion only after such an operation was accepted. Notice that in the latter case the insertion violation can **cascade** back to the EMPLOYEE relation if the user attempts to insert a tuple for department 7 with a value for Mgr_ssn that does not exist in the EMPLOYEE relation.

5.3.2 The Delete Operation

The **Delete** operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.

- *Operation*: Delete the WORKS_ON tuple with Essn = ‘999887777’ and Pno = 10.

  Result: This deletion is acceptable and deletes exactly one tuple.

- *Operation*: Delete the EMPLOYEE tuple with Ssn = ‘999887777’.

Result: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.

- *Operation*: Delete the EMPLOYEE tuple with Ssn = '333445555'.

Result: This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT, WORKS_ON, and DEPENDENT relations.

Several options are available if a deletion operation causes a violation. The first option, called **restrict,** is to *reject the deletion.* The second option, called **cascade,** is to *attempt to cascade (or propagate) the deletion* by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with Essn = '999887777'. A third option, called **set null** or **set default,** is to *modify the referencing attribute values* that cause the violation; each such value is either set to NULL or changed to reference another default valid tuple. Notice that if a referencing attribute that causes a violation is *part of the primary key,* it *cannot* be set to NULL; otherwise, it would violate entity integrity. Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, the DBMS may automatically delete all tuples from WORKS_ON and DEPENDENT with Essn = '333445555'. Tuples in EMPLOYEE with Super_ssn = '333445555' and the tuple in DEPARTMENT with Mgr_ssn = '333445555' can have their Super_ssn and Mgr_ssn values changed to other valid values or to NULL. Although it may make sense to delete automatically the WORKS_ON and DEPENDENT tuples that refer to an EMPLOYEE tuple, it may not make sense to delete other EMPLOYEE tuples or a DEPARTMENT tuple.

In general, when a referential integrity constraint is specified in the DDL, the DBMS will allow the database designer to *specify which of the options* applies in case of a violation of the constraint. We discuss how to specify these options in the SQL DDL in Chapter 6.

5.3.3 The Update Operation

The **Update** (or **Modify**) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation *R*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Here are some examples.

- *Operation*: Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.

Result: Acceptable.

- *Operation*: Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1.

Result: Acceptable.

- *Operation*: Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7.

Result: Unacceptable, because it violates referential integrity.

- *Operation*: Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654321'.

Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn.

Updating an attribute that is *neither part of a primary key nor part of a foreign key* usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Modifying a primary key value is similar to deleting one tuple and inserting another in its place because we use the primary key to identify tuples. Hence, the issues discussed earlier in both Sections 5.3.1 (Insert) and 5.3.2 (Delete) come into play. If a foreign key attribute is modified, the DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is set to NULL). Similar options exist to deal with referential integrity violations caused by Update as those options discussed for the Delete operation.

In fact, when a referential integrity constraint is specified in the DDL, the DBMS will allow the user to choose separate options to deal with a violation caused by Delete and a violation caused by Update (see Section 6.2).

5.3.4 The Transaction Concept

A database application program running against a relational database typically executes one or more *transactions*. A **transaction** is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database. At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema. A single transaction may involve any number of retrieval operations (to be discussed as part of relational algebra and calculus in Chapter 8, and as a part of the language SQL in Chapters 6 and 7) and any number of update operations. These retrievals and updates will together form an atomic unit of work against the database. For example, a transaction to apply a bank withdrawal will typically read the user account record, check if there is a sufficient balance, and then update the record by the withdrawal amount. A large number of commercial applications running against relational databases in **online transaction processing** (**OLTP**) systems are executing transactions at rates that reach several hundred per second. Transaction processing concepts, concurrent execution of transactions, and recovery from failures will be discussed in Chapters 20 to 22.

6. 5.4 Summary

In this chapter we presented the modeling concepts, data structures, and constraints provided by the relational model of data. We started by introducing the concepts of domains, attributes, and tuples. Then, we defined a relation schema as a list of attributes that describe the structure of a relation. A relation, or relation state, is a set of tuples that conforms to the schema.

Several characteristics differentiate relations from ordinary tables or files. The first is that a relation is not sensitive to the ordering of tuples. The second involves the ordering of attributes in a relation schema and the corresponding ordering of values within a tuple. We gave an alternative definition of relation that does not require ordering of attributes, but we continued to use the first definition, which requires attributes and tuple values to be ordered, for convenience. Then, we discussed values in tuples and introduced NULL values to represent missing or unknown information.

We emphasized that NULL values should be avoided as much as possible. We classified database constraints into inherent model-based constraints, explicit schema-based constraints, and semantic constraints or business rules. Then, we discussed the schema constraints pertaining to the relational model, starting with domain constraints, then key constraints (including the concepts of superkey, key, and primary key), and the NOT NULL constraint on attributes. We defined relational databases and relational database schemas. Additional relational constraints include the entity integrity constraint, which prohibits primary key attributes from being NULL. We described the interrelation referential integrity constraint, which is used to maintain consistency of references among tuples from various relations.

The modification operations on the relational model are Insert, Delete, and Update. Each operation may violate certain types of constraints (refer to Section 5.3). Whenever an operation is applied, the resulting database state must be a valid state.

Finally, we introduced the concept of a transaction, which is important in relational DBMSs because it allows the grouping of several database operations into a single atomic action on the database.