3 The Relational Model

Alice: *What is a relation?*
Vittorio: *You studied that in math a long time ago.*
Sergio: *It is just a table.*
Riccardo: *But we have several ways of viewing it.*

A *database model* provides the means for specifying particular data structures, for constraining the data sets associated with these structures, and for manipulating the data. The specification of structure and constraints is done using a *data definition language* (DDL), and the specification of manipulation is done using a *data manipulation language* (DML). The most prominent structures that have been used for databases to date are graphs in the network, semantic, and object-oriented models; trees in the hierarchical model; and relations in the relational model.

DMLs provide two fundamental capabilities: *querying* to support the extraction of data from the current database; and *updating* to support the modification of the database state. There is a rich theory on the topic of querying relational databases that includes several languages based on widely different paradigms. This theory is the focus of Parts B, D, and E, and portions of Part F of this book. The theory of database updates has received considerably less attention and is touched on in Part F.

The term *relational model* is actually rather vague. As introduced in Codd's seminal article, this term refers to a specific data model with relations as data structures, an algebra for specifying queries, and no mechanisms for expressing updates or constraints. Subsequent articles by Codd introduced a second query language based on the predicate calculus of first-order logic, showed this to be equivalent to the algebra, and introduced the first integrity constraints for the relational model—namely, functional dependencies. Soon thereafter, researchers in database systems implemented languages based on the algebra and calculus, extended to include update operators and to include practically motivated features such as arithmetic operators, aggregate operators, and sorting capabilities. Researchers in database theory developed a number of variations on the algebra and calculus with varying expressive power and adapted the paradigm of logic programming to provide a third approach to querying relational databases. The story of integrity constraints for the relational model is similar: A rich theory of constraints has emerged, and two distinct but equivalent perspectives have been developed that encompass almost all of the constraints that have been investigated formally. The term *relational model* has thus come to refer to the broad class of database models that have relations as the data structure and that incorporate some or all of the query capabilities, update capabilities, and integrity constraints

mentioned earlier. In this book we are concerned primarily with the relational model in this broad sense.

Relations are simple data structures. As a result, it is easy to understand the conceptual underpinnings of the relational model, thus making relational databases accessible to a broad audience of end users. A second advantage of this simplicity is that clean yet powerful declarative languages can be used to manipulate relations. By *declarative*, we mean that a query/program is specified in a high-level manner and that an efficient execution of the program does not have to follow exactly its specification. Thus the important practical issues of compilation and optimization of queries had to be overcome to make relational databases a reality.

Because of its simplicity, the relational model has provided an excellent framework for the first generation of theoretical research into the properties of databases. Fundamental aspects of data manipulation and integrity constraints have been exposed and studied in a context in which the peculiarities of the data model itself have relatively little impact. This research provides a strong foundation for the study of other database models, first because many theoretical issues pertinent to other models can be addressed effectively within the relational model, and second because it provides a variety of tools, techniques, and research directions that can be used to understand the other models more deeply.

In this short chapter, we present formal definitions for the data structure of the relational model. Theoretical research on the model has grown out of three different perspectives, one corresponding most closely to the natural usage of relations in databases, another stemming from mathematical logic, and the third stemming from logic programming. Because each of these provides important intuitive and notational benefits, we introduce notation that encompasses the different but equivalent formulations reflecting each of them.

3.1 The Structure of the Relational Model

An example of a relational database is shown in Fig. 3.1[1]. Intuitively, the data is represented in tables in which each row gives data about a specific object or set of objects, and rows with uniform structure and intended meaning are grouped into tables. Updates consist of transformations of the tables by addition, removal, or modification of rows. Queries allow the extraction of information from the tables. A fundamental feature of virtually all relational query languages is that the result of a query is also a table or collection of tables.

We introduce now some informal terminology to provide the intuition behind the formal definitions that follow. Each table is called a relation and it has a name (e.g., *Movies*). The columns also have names, called attributes (e.g, *Title*). Each line in a table is a tuple (or record). The entries of tuples are taken from sets of constants, called domains, that include, for example, the sets of integers, strings, and Boolean values. Finally we distinguish between the database schema, which specifies the structure of the database; and the database instance, which specifies its actual content. This is analogous to the standard distinction between type and value found in programming languages (e.g., an

[1] *Pariscope* is a weekly publication that lists the cultural events occurring in Paris and environs.

Movies	*Title*	*Director*	*Actor*
	The Trouble with Harry	Hitchcock	Gwenn
	The Trouble with Harry	Hitchcock	Forsythe
	The Trouble with Harry	Hitchcock	MacLaine
	The Trouble with Harry	Hitchcock	Hitchcock
	...	...	...
	Cries and Whispers	Bergman	Andersson
	Cries and Whispers	Bergman	Sylwan
	Cries and Whispers	Bergman	Thulin
	Cries and Whispers	Bergman	Ullman

Location	*Theater*	*Address*	*Phone Number*
	Gaumont Opéra	31 bd. des Italiens	47 42 60 33
	Saint André des Arts	30 rue Saint André des Arts	43 26 48 18
	Le Champo	51 rue des Ecoles	43 54 51 60
	...	...	...
	Georges V	144 av. des Champs-Elysées	45 62 41 46
	Les 7 Montparnassiens	98 bd. du Montparnasse	43 20 32 20

Pariscope	*Theater*	*Title*	*Schedule*
	Gaumont Opéra	Cries and Whispers	20:30
	Saint André des Arts	The Trouble with Harry	20:15
	Georges V	Cries and Whispers	22:15
	...	...	...
	Les 7 Montparnassiens	Cries and Whispers	20:45

Figure 3.1: The **CINEMA** database

identifier X might have type *record* A : *int,* B : *bool endrecord* and value *record* A : *5*, B : *true endrecord*).

We now embark on the formal definitions. We assume that a countably infinite set **att** of *attributes* is fixed. For a technical reason that shall become apparent shortly, we assume that there is a total order $\leq_{\mathbf{att}}$ on **att**. When a set U of attributes is listed, it is assumed that the elements of U are written according to $\leq_{\mathbf{att}}$ unless otherwise specified.

For most of the theoretical development, it suffices to use the same domain of values for all of the attributes. Thus we now fix a countably infinite set **dom** (disjoint from **att**), called the underlying *domain*. A *constant* is an element of **dom**. When different attributes should have distinct domains, we assume a mapping *Dom* on **att**, where $Dom(A)$ is a set called the domain of A.

We assume a countably infinite set **relname** of relation names disjoint from the previous sets. In practice, the structure of a table is given by a relation name and a set of attributes. To simplify the notation in the theoretical treatment, we now associate a *sort* (i.e., finite set of attributes) to each relation name. (An analogous approach is usually taken in logic.) In particular, we assume that there is a function *sort* from **relname** to $\mathcal{P}^{\text{fin}}(\mathbf{att})$ (the *finitary powerset* of **att**; i.e., the family of finite subsets of **att**). It is assumed that *sort* has the property that for each (possibly empty) finite set U of attributes, $sort^{-1}(U)$ is infinite. This allows us to use as many relation names of a given sort as desired. The *sort* of a relation name is simply $sort(R)$. The *arity* of a relation name R is $arity(R) = |sort(R)|$.

A *relation schema* is now simply a relation name R. We sometimes write this as $R[U]$ to indicate that $sort(R) = U$, or $R[n]$, to indicate that $arity(R) = n$. A *database schema* is a nonempty finite set $\mathbf{R}$ of relation names. This might be written $\mathbf{R} = \{R_1[U_1], \ldots, R_n[U_n]\}$ to indicate the relation schemas in $\mathbf{R}$.

For example, the database schema **CINEMA** for the database shown in Fig. 3.1 is defined by

$$\mathbf{CINEMA} = \{Movies,\ Location,\ Pariscope\}$$

where relation names *Movies*, *Location*, and *Pariscope* have the following sorts:

$$\begin{aligned} sort(Movies) &= \{Title,\ Director,\ Actor\} \\ sort(Location) &= \{Theater,\ Address,\ Phone\ Number\} \\ sort(Pariscope) &= \{Theater,\ Title,\ Schedule\}. \end{aligned}$$

We often omit commas and set brackets in sets of attributes. For example, we may write

$$sort(Pariscope) = Theater\ Title\ Schedule.$$

The formalism that has emerged for the relational model is somewhat eclectic, because it is intimately connected with several other areas that have their own terminology, such as logic and logic programming. Because the slightly different formalisms are well entrenched, we do not attempt to replace them with a single, unified notation. Instead we will allow the coexistence of the different notations; the reader should have no difficulty dealing with the minor variations.

Thus there will be two forks in the road that lead to different but largely equivalent formulations of the relational model. The first fork in the road to defining the relational model is of a philosophical nature. Are the attribute names associated with different relation columns important?

3.2 Named versus Unnamed Perspectives

Under the *named* perspective, these attributes are viewed as an explicit part of a database schema and may be used (e.g., by query languages and dependencies). Under the *unnamed*

perspective, the specific attributes in the sort of a relation name are ignored, and only the arity of a relation schema is available (e.g., to query languages).

In the named perspective, it is natural to view tuples as functions. More precisely, a *tuple* over a (possibly empty) finite set U of attributes (or over a relation schema $R[U]$) is a total mapping u from U to **dom**. In this case, the *sort* of u is U, and it has *arity* $|U|$. Tuples may be written in a linear syntax using angle brackets—for example, $\langle A : 5,\ B : 3\rangle$. (In general, the order used in the linear syntax will correspond to $\leq_{\mathbf{att}}$, although that is not necessary.) The unique tuple over $\emptyset$ is denoted $\langle\rangle$.

Suppose that u is a tuple over U. As usual in mathematics, the value of u on an attribute A in U is denoted $u(A)$. This is extended so that for $V \subseteq U$, $u[V]$ denotes the tuple v over V such that $v(A) = u(A)$ for each $A \in V$ (i.e., $u[V] = u|_V$, the restriction of the function u to V).

With the unnamed perspective, it is more natural to view a tuple as an element of a Cartesian product. More precisely, a *tuple* is an ordered n-tuple ($n \geq 0$) of constants (i.e., an element of the Cartesian product $\mathbf{dom}^n$). The arity of a tuple is the number of coordinates that it has. Tuples in this context are also written with angle brackets (e.g., $\langle 5,\ 3\rangle$). The i^{th} coordinate of a tuple u is denoted $u(i)$. If relation name R has arity n, then a *tuple* over R is a tuple with arity $arity(R)$.

Because of the total order $\leq_{\mathbf{att}}$, there is a natural correspondence between the named and unnamed perspectives. A tuple $\langle A_1 : a_1, A_2 : a_2\rangle$ (defined as a function) can be viewed (assuming $A_1 \leq_{\mathbf{att}} A_2$) as an ordered tuple with $(A_1 : a_1)$ as a first component and $(A_2 : a_2)$ as a second one. Ignoring the names, this tuple may simply be viewed as the ordered tuple $\langle a_1, a_2\rangle$. Conversely, the ordered tuple $t = \langle a_1, a_2\rangle$ may be interpreted as a function over the set $\{1, 2\}$ of integers with $t(i) = a_i$ for each i. This correspondence will allow us to blur the distinction between the two perspectives and move freely from one to the other when convenient.

3.3 Conventional versus Logic Programming Perspectives

We now come to the second fork in the road to defining the relational model. This fork concerns how relation and database instances are viewed, and it is essentially independent of the perspective taken on tuples. Under the *conventional* perspective, a *relation* or *relation instance* of (or over) a relation schema $R[U]$ (or over a finite set U of attributes) is a (possibly empty) finite set I of tuples with sort U. In this case, I has *sort* U and *arity* $|U|$. Note that there are two instances over the empty set of attributes: $\{\}$ and $\{\langle\rangle\}$.

Continuing with the conventional perspective, a *database instance* of database schema $\mathbf{R}$ is a mapping $\mathbf{I}$ with domain $\mathbf{R}$, such that $\mathbf{I}(R)$ is a relation over R for each $R \in \mathbf{R}$.

The other perspective for defining instances stems from logic programming. This perspective is used primarily with the ordered-tuple perspective on tuples, and so we focus on that here. Let R be a relation with arity n. A *fact* over R is an expression of the form $R(a_1, \ldots, a_n)$, where $a_i \in \mathbf{dom}$ for $i \in [1, n]$. If $u = \langle a_1, \ldots, a_n\rangle$, we sometimes write $R(u)$ for $R(a_1, \ldots, a_n)$. Under the *logic-programming* perspective, a *relation* (*instance*) over R is a finite set of facts over R. For a database schema $\mathbf{R}$, a *database instance* is a finite set $\mathbf{I}$ that is the union of relation instances over R, for $R \in \mathbf{R}$. This perspective on

instances is convenient when working with languages stemming from logic programming, and it permits us to write database instances in a convenient linear form.

The two perspectives provide alternative ways of describing essentially the same data. For instance, assuming that $sort(R) = AB$ and $sort(S) = A$, we have the following four representations of the same database:

Named and Conventional
$I(R) = \{f_1, f_2, f_3\}$

$$f_1(A) = a \quad f_1(B) = b$$
$$f_2(A) = c \quad f_2(B) = b$$
$$f_3(A) = a \quad f_3(A) = a$$

$I(S) = \{g\}$

$$g(A) = d$$

Unnamed and Conventional
$I(R) = \{\langle a, b\rangle, \langle c, b\rangle, \langle a, a\rangle\}$
$I(S) = \{\langle d\rangle\}$

Named and Logic Programming
$\{R(A : a, B : b),\ R(A : c, B : b),\ R(A : a, B : a),\ S(A : d)\}$

Unnamed and Logic Programming
$\{R(a, b),\ R(c, b),\ R(a, a),\ S(d)\}$.

Because relations can be viewed as sets, it is natural to consider, given relations of the same sort, the standard set operations *union* ($\cup$), *intersection* ($\cap$), and *difference* ($-$) and the standard set comparators $\subset$, $\subseteq$, $=$, and $\neq$. With the logic-programming perspective on instances, we may also use these operations and comparators on database instances.

Essentially all topics in the theory of relational database can be studied using a fixed choice for the two forks. However, there are some cases in which one perspective is much more natural than the other or is technically much more convenient. For example, in a context in which there is more than one relation, the named perspective permits easy and natural specification of correspondences between columns of different relations whereas the unnamed perspective does not. As will be seen in Chapter 4, this leads to different but equivalent sets of natural primitive algebra operators for the two perspectives. A related example concerns those topics that involve the association of distinct domains to different relation columns; again the named perspective is more convenient. In addition, although relational dependency theory can be developed for the unnamed perspective, the motivation is much more natural when presented in the named perspective. Thus during the course of this book the choice of perspective during a particular discussion will be motivated primarily by the intuitive or technical convenience offered by one or the other.

In this book, we will need an infinite set **var** of *variables* that will be used to range over elements of **dom**. We generalize the notion of tuple to permit variables in coordinate positions: a *free tuple* over U or $R[U]$ is (under the named perspective) a function u from U to $\mathbf{var} \cup \mathbf{dom}$. An *atom* over R is an expression $R(e_1, \ldots, e_n)$, where $n = arity(R)$ and

e_i is *term* (i.e., $e_i \in \mathbf{var} \cup \mathbf{dom}$ for each $i \in [1, n]$). Following the terminology of logic and logic programming, we sometimes refer to a fact as a *ground* atom.

3.4 Notation

We generally use the following symbols, possibly with subscripts:

Constants	a, b, c
Variables	x, y
Sets of variables	X, Y
Terms	e
Attributes	A, B, C
Sets of attributes	U, V, W
Relation names (schemas)	R, S; $R[U]$, $S[V]$
Database schemas	$\mathbf{R}, \mathbf{S}$
Tuples	t, s
Free tuples	u, v, w
Facts	$R(a_1, \ldots, a_n)$, $R(t)$
Atoms	$R(e_1, \ldots, e_n)$, $R(u)$
Relation instances	I, J
Database instances	$\mathbf{I}, \mathbf{J}$

Bibliographic Notes

The relational model is founded on mathematical logic (in particular, predicate calculus). It is one of the rare cases in which substantial theoretical development preceded the implementation of systems. The first proposal to use predicate calculus as a query language can be traced back to Kuhns [Kuh67]. The relational model itself was introduced by Codd [Cod70]. There are numerous commercial database systems based on the relational model. They include IBM's DBZ, [A$^+$76], INGRES [SWKH76], and ORACLE [Ora89], Informix, and Sybase.

Other data models have been proposed and implemented besides the relational model. The most prominent ones preceding the relational model are the hierarchical and network models. These and other models are described in the books [Nij76, TL82]. More recently, various models extending the relational model have been proposed. They include semantic models (see the survey [HK87]) and object-oriented models (see the position paper [ABD$^+$89]). In this book we focus primarily on the relational model in a broad sense. Some formal aspects of other models are considered in Part F.