Chapter 1

Introduction to Database Concepts

1.1 Databases and Database Systems

1.1.1 What Is a Database?

A *database* can be summarily described as a repository for data. This makes clear that building databases is really a continuation of a human activity that has existed since writing began; it can be applied to the result of any bookkeeping or recording activity that occurred long before the advent of the computer era. However, this description is too vague for some of our purposes, and we refine it as we go along.

The creation of a database is required by the operation of an enterprise. We use the term *enterprise* to designate a variety of endeavors that range from an airline, a bank, or a manufacturing company to a stamp collection or keeping track of people to whom you want to write New Year cards.

Throughout this book we use a running example that deals with the database of a small college. The college keeps track of its students, its instructors, the courses taught by the college, grades received by students, and the assignment of advisors to students, as well as other aspects of the activity of the institution that we discuss later. These data items constitute the *operational data* — that is, the data that the college needs to function. Operational data are built from various *input data* (application forms for students, registration forms, grade lists, schedules) and is used for generating *output data* (transcripts, registration records, administrative reports, etc.) Note that no computer is necessary for

using such a database; a college of the 1930's would have kept the same database in paper form. However, the existence of computers to store and manipulate the data does change user expectations: we expect to store more data and make more sophisticated use of these data.

1.1.2 Database Management Systems

A *database management system* (DBMS) is an aggregate of data, hardware, software, and users that helps an enterprise manage its operational data. The main function of a DBMS is to provide efficient and reliable methods of data retrieval to many users. If our college has 10,000 students each year and each student can have approximately 10 grade records per year, then over 10 years, the college will accumulate 1,000,000 grade records. It is not easy to extract records satisfying certain criteria from such a set, and by current standards, this set of records is quite small! Given the current concern for "grade inflation", a typical question that we may try to answer is determining the evolution of the grade averages in introductory programming courses over a 10-year period. Therefore, it is clear that efficient data retrieval is an essential function of database systems.

Most DBMSs deal with several users who try simultaneously to access several data items and, frequently, the same data item. For instance, suppose that we wish to introduce an automatic registration system for students. Students may register by using terminals or workstations. Of course, we assume that the database contains information that describes the capacity of the courses and the number of seats currently available. Suppose that several students wish to register for cs210 in the spring semester of 2003. Unfortunately, the capacity of the course is limited, and not all demands can be satisfied. If, say, only one seat remains available in that class, the database must handle these competing demands and allow only one registration to go through.

Database System Hardware

Database management systems are, in most cases, installed on general-purpose computers. Since the characteristics of the hardware have strongly influenced the development of DBMSs, we discuss some of the most important of these characteristics.

For our purposes, it is helpful to categorize computer memory into two classes: internal memory and external memory. Although some internal memory is permanent, such as ROM,[1] we are interested here only in memory that can be changed by programs. This memory is often known as RAM.[2] This memory is *volatile*, and any electrical interruption causes the loss of data.

By contrast, magnetic disks and tapes are common forms of external memory. They are *nonvolatile memory*, and they retain their content for practically

[1] ROM stands for Read Only Memory; it is memory that must be written using special equipment or special procedures, and for our purposes is considered unchangeable.

[2] RAM stands for Random Access Memory.

unlimited amounts of time. The physical characteristics of magnetic tapes force them to be accessed sequentially, making them useful for backup purposes, but not for quick access to specific data.

In examining the memory needs of a DBMS, we need to consider the following issues:

- Data of a DBMS must have a *persistent* character; in other words, data must remain available long after any program that is using it has completed its work. Also, data must remain intact even if the system breaks down.
- A DBMS must access data at a relatively high rate.
- Such a large quantity of data need to be stored that the storage medium must be low cost.

These requirements are satisfied at the present stage of technological development only by magnetic disks.

Database System Software

Users interact with database systems through *query languages*. The query language of a DBMS has two broad tasks: to define the data structures that serve as receptacles for the data of the database, and to allow the speedy retrieval and modification of data. Accordingly, we distinguish between two components of a query language: the *data definition component* and the *data manipulation component*.

The main tasks of data manipulation are *data retrieval* and *data update*. Data retrieval entails obtaining data stored in the database that satisfies a certain specification formulated by the user in a query. Data updates include data modification, deletion and insertion.

Programming in query languages of DBMSs is done differently from programming in higher-level programming languages. The typical program written in C, Pascal, or PL/1 directly implements an algorithm for solving a problem. A query written in a database query language merely states what the problem is and leaves the construction of the code that solves the problem to a special component of the DBMS software. This approach to programming is called *nonprocedural*.

A central task of DBMSs is *transaction management*. A *transaction* is a sequence of database operations (that usually consists of updates, with possible retrievals) that must be executed in its entirety or not at all. This property of transactions is known as *atomicity*. A typical example includes the transfer of funds between two account records A and B in the database of a bank. Such a banking operation should not modify the total amount of funds that the bank has in its accounts, which is a clear consistency requirement for the database. The transaction consists of the following sequence of operations:

1. Decrease the balance of account A by d dollars;
2. Increase the balance of account B by d dollars.

If only the first operation is executed, then d dollars will disappear from the funds deposited with the bank. If only the second is executed, then the total funds will increase by d dollars. In either case, the consistency of the database will be compromised. Thus, a transaction transforms one consistent database state into another consistent database state, a property of transactions known as *consistency*.

Typically, at any given moment in time a large number of transactions coexist in the database system. The transaction management component ensures that the execution of one transaction is not influenced by the execution of any other transaction. This is the *isolation* property of transactions. Finally, the effect of a transaction to the state of the database must by durable, that is, it must persist in the database after the execution of the transaction is completed. This is the *durability property* of transactions. Collectively, the four fundamental properties of transactions outlined above are known as the *ACID properties*, the acronym of **a**tomicity, **c**onsistency, **i**solation, and **d**urability.

DBMS software usually contains application development tools in addition to query languages. The role of these tools is to facilitate user interface development. They include forms systems, procedural and nonprocedural programming languages that integrate database querying with various user interfaces, etc.

The Users of a Database System

The community of users of a DBMS includes a variety of individuals and organizational entities. These users are classified based on their roles and interests in accessing and managing the databases.

Once a database is created, it is the job of the *database administrator* to make decisions about the nature of data to be stored in the database, the access policies to be enforced (who is going to access certain parts of the database), monitoring and tuning the performance of the database, etc.

At the other extremity of the user range, we have the *end users*. These users have limited access rights, and they need to have only minimal technical knowledge of the database. For instance, the end users of the database of the reservation system of an airline are travel and sales agents. The end users of a DBMS of a bank are bank tellers, users of the ATM machines, etc.

A particularly important category of users of DBMSs (on whom we focus in this book) consists of application programmers. Their role is to work within existing DBMS systems and, using a combination of the query languages and higher-level languages, to create various reports based on the data contained in the database. In some cases, they write more general programs that depend on these data.

1.2 The Architecture of Database Systems

The architecture of a DBMS can be examined from several angles: the functional architecture that identifies the main components of a DBMS, the application

architecture that focuses on application uses of DBMSs, and the logical architecture that describes various levels of data abstractions.

Functionally, a DBMS contains several main components shown in Figure 1.1:

- the *memory manager*;
- the *query processor*;
- the *transaction manager*.

The query processor converts a user query into instructions the DBMS can process efficiently, taking into account the current structure of the database (also referred as *metadata* — which means data about data).

The memory manager obtains data from the database that satisfies queries compiled by the query processor and manages the structures that contain data, according to the DDL directives.

Finally, the transaction manager ensures that the execution of possibly many transactions on the DBMS satisfies the ACID properties mentioned above and, also, provides facilities for recovery from system and media failures.

The standard application architecture of DBMSs is based on a *client/server* model. The client, which can be a user or an application, generates a query that is conveyed to the server. The server processes the query (a process that includes parsing, generation of optimized execution code, and execution) and returns an answer to the client. This architecture is known as *two-tier architecture*. In general, the number of clients may vary over time.

In large organizations, it is often necessary to create more layers of processing, with, say, a layer of software to concentrate the data activities of a branch office and organize the communication between the branch and the main data repository. This leads to what is called a *multi-tier architecture*. In this setting data are scattered among various data sources that could be DBMSs, file systems, etc. These constitute the lowest tier of the architecture, that is, the tier that is closest to the data. The highest tier consists of users that act through user interfaces and applications to obtain answers to queries. The intermediate tiers constitute the *middleware*, and their role is, in general, to serve as mediators between the highest and the lowest tiers. Middleware may be consist of web servers, data warehouses, and may be considerably complex. Multi-tier architecture is virtually a requirement for world wide web applications.

The logical architecture, also known as the ANSI/SPARC architecture, was elaborated at the beginning of the 1970s. It distinguishes three layers of data abstraction:

1. *The physical layer* contains specific and detailed information that describes how data are stored: addresses of various data components, lengths in bytes, etc. DBMSs aim to achieve *data independence*, which means that the database organization at the physical level should be indifferent to application programs.
2. *The logical layer* describes data in a manner that is similar to, say, definitions of structures in C. This layer has a conceptual character; it shields the user from the tedium of details contained by the physical layer, but is

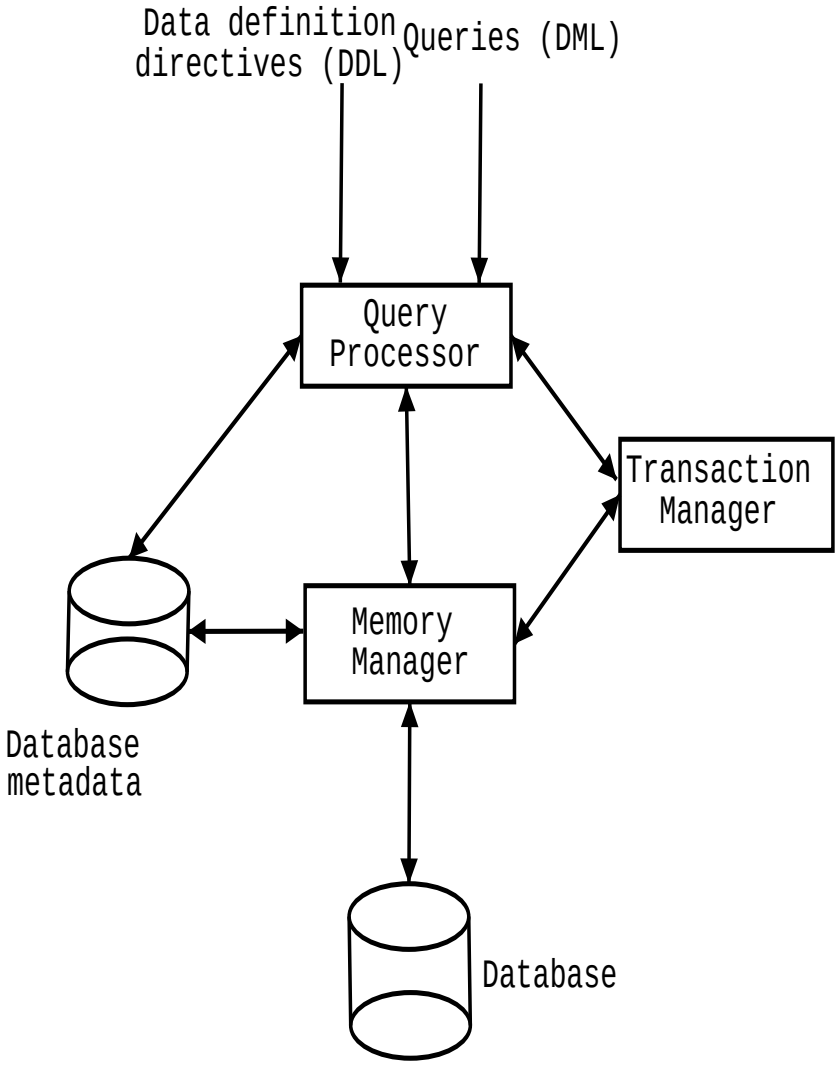

Figure 1.1: Functional Architecture of DBMSs

essential in formulating queries for the DMBS.
3. *The user layer* contains each user's perspective of the content of the database.

1.3 A Historical Perspective of Database Systems

The history of DBMSs begins in the late 1960s, when an IBM product named IMS (Information Management System) was launched. Data was structured hierarchically, in forests of trees of records, providing very fast access. A few years after IMS appeared, in 1971, the CODASYL Database Task Group proposed a new type of database models known today as the network model. The original report considered DBMSs as extensions of the COBOL language, and structured data contained by databases as graphs of records, consisting essentially of circular linked lists. The origins of the relational model, that is the mainstay of contemporary databases are in E. F. Codd's work in the early and mid 1970s. The development of relational database began in the late 1970s and early 1980s with an experimental relational database sytem at IBM called System R, a precursor of commercial IBM DBMSs, SQL/DS and DB2. A multitude of DBMSs emerged in the 1980s, such as ORACLE, INGRES, Rdb, etc. Relational technology evolved further in the 1990s with the addition of ideas and techniques inspired by object-oriented programming.

1.4 Bibliographical Comments

Codd's foundational work in relational databases is included in several articles [Codd, 1970; Codd, 1972a; Codd, 1972b; Codd, 1974], and [Codd, 1990].

Standard references in the database literature that contain extensive bibliographies are [Date, 2003; Elmasri and Navathe, 2006; Silberschatz *et al.*, 2005] and [Ullman, 1988a; Ullman, 1988b].

Chapter 2

The Entity–Relationship Model

The entity–relationship model (the E/R model) was developed by P. P. Chen and is an important tool for database design. After an introductory section, we define the main elements of the E/R model, and we discuss the use of the E/R model to facilitate the database design process.

2.1 The Main Concepts of the E/R Model

The E/R model uses the notions of *entity, relationship*, and *attribute*. These notions are quite intuitive. Informally, entities are objects that need to be represented in the database; relationships reflect interactions between entities; attributes are properties of entities and relationships.

For the present, the database of the college used for our running example reflects the following information:

- Students: any student who has ever registered at the college;
- Instructors: anyone who has ever taught at the college;
- Courses: any course ever taught at the college;
- Advising: which instructor currently advises which student, and
- Grades: the grade received by each student in each course, including the semester and the instructor.

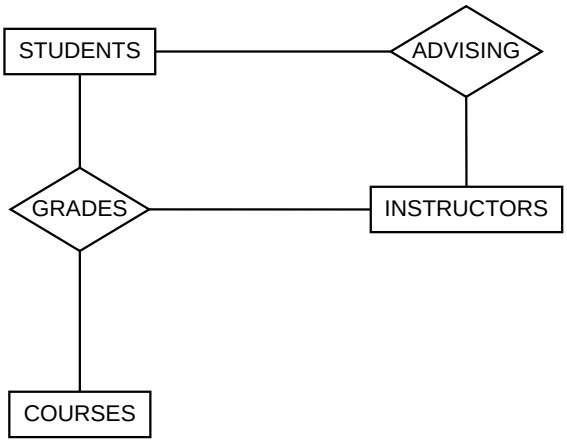

Figure 2.1: The E/R Diagram of the College Database

We stress that this example database is intentionally simplified; it is used here to illustrate certain ideas. To make it fully useful, we would need to include many more entities and relationships.

A single student is represented by an entity; a student's grade in a course is a single relationship between the student, the course, and the instructor. The fact that an instructor advises a student is represented by a relationship between them.

Individual entities and individual relationships are grouped into homogeneous sets of entities (STUDENTS, COURSES, and INSTRUCTORS) and homogeneous sets of relationships (ADVISING, GRADES). Thus, for example, STUDENTS represent all the student entities, and ADVISING, all the individual advising relationships. We refer to such sets as *entity sets* and *relationship sets*, respectively.

Definition 2.1.1 An *n-ary relationship* is a relationship that involves n entities from n pairwise distinct sets of entities $E_1, \ldots, E_n$. □

We use the *entity/relationship diagram*, a graphical representation of the E/R model, where entity sets are represented by rectangles and sets of relationships by diamonds. (See Figure 2.1 for a representation of the entity/relationship diagram of the college database.)

An E/R diagram of a database can be viewed as a graph whose vertices are the sets of entities and the sets of relationships. An edge may exist *only between a set of relationships and a set of entities*. Also, every vertex must be joined by at least one edge to some other vertex of the graph; in other words, this graph must be connected. This is an expression of the fact that data contained in a database have an integrated character. This means that various parts of the database are logically related and data redundancies are minimized. An E/R design that results in a graph that is not connected indicates that we are dealing

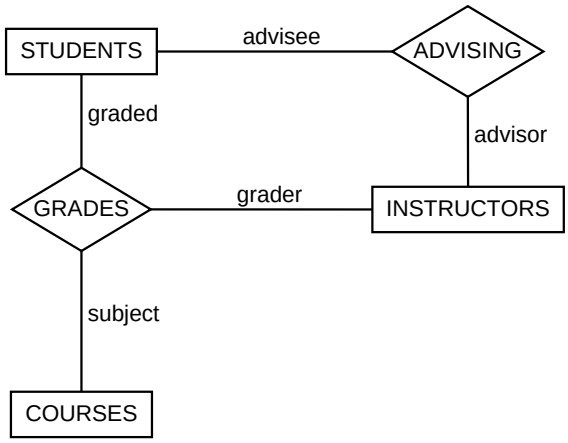

Figure 2.2: Roles of Entities in the College E/R Diagram

with more than one database.

The notion of *role* that we are about to introduce helps explain the significance of entities in relationships. Roles appear as labels of the edges of the E/R diagram.

Example 2.1.2 We consider the following roles in the college database:

Role	Relationship Set	Entity Set
advisee	ADVISING	STUDENTS
advisor	ADVISING	INSTRUCTORS
graded	GRADES	STUDENTS
grader	GRADES	INSTRUCTORS
subject	GRADES	COURSES

These role explain which entities are involved in the relationship and in which capacity: who is graded, who is the instructor who gave the grade, and in which course was the grade given. In Figure 2.2 we show the diagram from Figure 2.1 with the edges marked by the roles discussed above.

□

2.2 Attributes

Properties of entities and relationships are described by attributes. Each attribute A has an associated set of values, which we refer to as the *domain* of A and denote by $\mathrm{Dom}(A)$. The set of attributes of a set of entities E is denoted by $\mathbf{Attr}(E)$; similarly, the set of attributes of a set of relationships R is denoted by $\mathbf{Attr}(R)$.

Example 2.2.1 The set of entities STUDENTS of the college database has

the attributes *student identification number* (stno), *student name* (name), *street address* (addr), *city* (city), *state of residence* (state), *zip code* (zip).

The student Edwards P. David, who lives at 10 Red Rd. in Newton, MA, 02129, has been assigned ID number 1011. The value of his attributes are:

Attribute	Value
stno	'1011'
name	'Edwards P. David'
addr	'10 Red Rd.'
city	'Newton'
state	'MA'
zip	'02129'

□

We assume that domains of attributes consist of *atomic values*. This means that the elements of such domains must be "simple" values such as integers, dates, or strings of characters. Domains may not contain such values as sets, trees, relations, or any other complex objects. Simple values are those that are not further decomposed in working with them.

If e is an entity and A is an attribute of that entity, then we denote by $A(e)$ the value of the domain of A that the attribute associates with the entity e. Similarly, when r is a relationship, we denote the value associated by an attribute B to r as $B(r)$. For example, if s is a student entity, then the values associated to s are denoted by

$$\mathsf{stno}(s), \mathsf{name}(s), \mathsf{addr}(s), \mathsf{city}(s), \mathsf{state}(s), \mathsf{zip}(s).$$

A DBMS must *support* attribute domains. Such support includes validity checks and implementation of operations specific to the domains. For instance, whenever an assignment $A(e) = v$ is made, where e is an entity and A is an attribute of e, the DBMS should verify whether v belongs to $\mathrm{Dom}(A)$. Operations defined on specific domains include string concatenation for strings of characters, various computations involving dates, and arithmetic operations on numeric domains.

Dom(name) is the set of all possible names for students. However, such a definition is clearly impractical for a real database because it would make the support of such a domain an untenable task. Such support would imply that the DBMS must somehow store the list of all possible names that human beings may adopt. Only in this way would it be possible to check the validity of an assignment of a name. Thus, in practice, we define Dom(name) as the set of all strings of length less or equal to a certain length n. For the sake of this example, we adopt $n = 35$.

The set of all strings of characters of length k is denoted by $\mathrm{CHAR}(k)$. The set of all 4-bytes integers that is implemented on our system is denoted by INTEGERS. Similarly, we could consider the set of two-byte integers and denote this set with SMALLINT. Thus, in Figure 2.3, we use CHAR(35) as the domain for name, SMALLINT as domain for cr, and INTEGER for roomno.

Entity Set	Attribute	Domain	Description
STUDENTS	stno	CHAR(10)	college-assigned student ID number
	name	CHAR(35)	full name
	addr	CHAR(35)	street address
	city	CHAR(20)	home city
	state	CHAR(2)	home state
	zip	CHAR(10)	home zip
COURSES	cno	CHAR(5)	college-assigned course number
	cname	CHAR(30)	course title
	cr	SMALLINT	number of credits
	cap	INTEGER	maximum number of students
INSTRUCTORS	empno	CHAR(11)	college-assigned employee ID number
	name	CHAR(35)	full name
	rank	CHAR(12)	academic rank
	roomno	INTEGER	office number
	telno	CHAR(4)	office telephone number

Figure 2.3: Attributes of Sets of Entities

Relationship Set	Attribute	Domain
GRADES	stno	CHAR(10)
	empno	CHAR(11)
	cno	CHAR(5)
	sem	CHAR(6)
	year	INTEGER
	grade	INTEGER
ADVISING	stno	CHAR(10)
	empno	CHAR(11)

Figure 2.4: Attributes of Sets of Relationships

The attributes of the sets of entities considered in our current example (the college database) are summarized in Figure 2.3.

If several sets of entities that occur in the same context each have an attribute A, we qualify the attribute with the name of the entity set to be able to differentiate between these attributes. For example, because both STUDENTS and INSTRUCTORS have the attribute name, we use the qualified attributes STUDENTS.name and INSTRUCTORS.name.

Attributes of relationships may either be attributes of the entities they relate, or be new attributes, specific to the relationship. For instance, a grade involves a student, a course, and an instructor, and for these, we use attributes from the participating entities: stno, cno, and empno, respectively. In addition, we need to specify the semester and year when the grade was given as well as the grade itself. For these, we use new attributes: sem, year, and grade. Therefore, the set of relationships GRADES has the attributes stno (from STUDENTS), cno (from COURSES), and empno (from INSTRUCTORS), and also its own attributes sem, year, and grade. By contrast, the set of relationships ADVISING has only attributes gathered from the entities it relates stno (from STUDENTS) and empno (from INSTRUCTORS). The attributes of the sets of relationships GRADES and ADVISING are listed in Figure 2.4. Note that in our college, grades are integers (between 0 and 100) rather than letters.

It is a feature of the E/R model that the distinction between entities and relationships is intentionally vague. This allows different views of the constituents

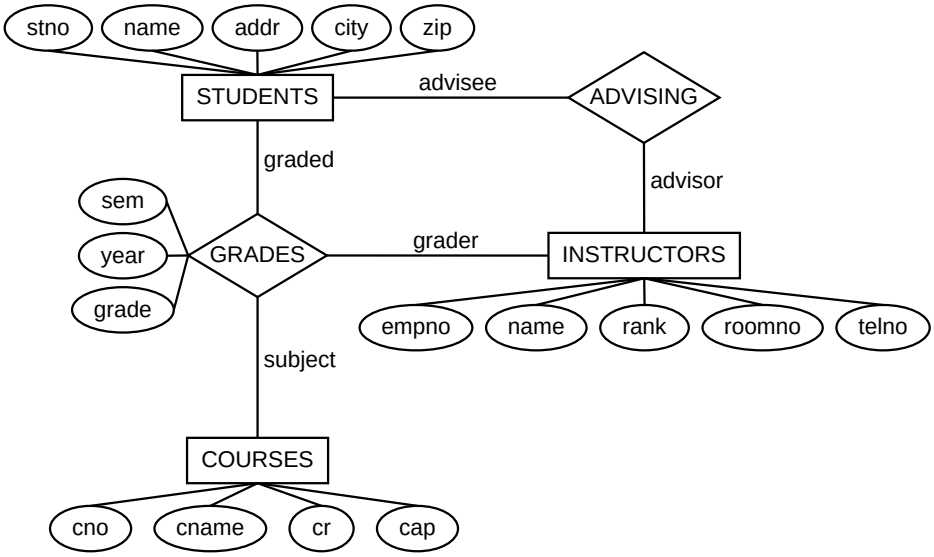

Figure 2.5: The E/R Diagram of the College Database

of the model to be adopted by different database designers. The distinction between entities and relationships is a decision of the model builder that reflects his or her understanding of the semantics of the model. In other words, an object is categorized as an entity or a relationship depending on a particular design choice at a given moment; this design decision could change if circumstances change. For instance, the E/R model of the college database regards GRADES as a set of relationships between STUDENTS, COURSES, and INSTRUCTORS. An alternative solution could involve regarding GRADES as a set of entities and then introducing sets of relationships linking GRADES with STUDENTS, COURSES, and INSTRUCTORS, etc.

Sometimes attributes are represented by circles linked to the rectangles or diamonds by undirected edges. However, to simplify the drawings, we list the attributes of sets of entities or relationships close to the graphical representations of those sets as in Figure 2.6.

2.3 Keys

In order to talk about a specific student, you have to be able to identify him. A common way to do this is to use his name, and generally, this works reasonably well. So, you can ask something like, "Where does Roland Novak live?" In database terminology, we are using the student's name as a "key", an attribute (or set of attributes) that uniquely identifies each student. So long as no two students have the same name, you can use the name attribute as a key.

What would happen, though, if there were two students named "Helen

Rivers"? Then, the question, "Where does Helen Rivers live?" could not be answered without additional information. The name attribute would no longer uniquely identify students, so it could not be used as a key for STUDENTS.

The college solves this problem in a common way: it assigns a unique identifier (corresponding to the stno attribute) to each student when he first enrolls. This identifier can then be used to specify a student unambiguously; i.e., it can be used as a key. If one Helen Rivers has ID 6568 and the other has ID 4140, then instead of talking about "Helen Rivers", leaving your listener wondering which one is meant, you can talk about "the student with ID number 6568". It's less natural in conversation, but it makes clear which student is meant. Avoiding ambiguity is especially important for computer programs, so having an attribute, or a set of attributes, that uniquely identifies each entity in a collection is generally a necessity for electronic databases.

We discuss the notion of *keys* for both sets of entities and sets of relationships. We begin with sets of entities.

Let E be a set of entities having $A_1, \ldots, A_n$ as its attributes. The set $\{A_1, \ldots, A_n\}$ is denoted by $A_1 \ldots A_n$. Unfortunately, this notation conflicts with standard mathematical notation; however, it has been consecrated by its use in databases, so we adhere to it when dealing with sets of attributes. Further, if H and L are two sets of attributes, their union is denoted by concatenation; namely, we write $HL = A_1 \ldots A_n B_1 \ldots B_m$ for $H \cup L$ if $H = A_1 \ldots A_n$ and $L = B_1 \ldots B_m$.

Definition 2.3.1 Let E be a set of entities such that $\mathbf{Attr}(E) = A_1 \ldots A_n$. A *key* of E is a nonempty subset L of $\mathbf{Attr}(E)$ such that the following conditions are satisfied:

1. For all entities, e, e' in E, if $A(e) = A(e')$ for every attribute A of L, then $e = e'$ (the *unique identification property of keys*).
2. No proper, nonempty subset of L has the unique identification property (*the minimality property of keys*).

□

Example 2.3.2 In the college database, the value of the attribute stno is sufficient to identify a student entity. Since the set *stno* has no proper, nonempty subsets, it clearly satisfies the minimality condition and, therefore, it is a key for the STUDENTS entity set. For our college, the entity set COURSES both cno and cname are keys. Note that this reflects a "business rule", namely that no two courses may have the same name, even if they are offered by different departments. □

Example 2.3.3 Consider the design of the database of the customers of a town library. We introduce the entity sets PATRONS and BOOKS and the set of relationships LOANS between BOOKS and PATRONS. The E/R diagram of this database is represented in Figure 2.6.

The inventory number invno is clearly a key for the set of entities BOOKS. If the library never buys more than one copy of any title, then the ISBN number, isbn, is another key, and so is the set of attributes author title publ

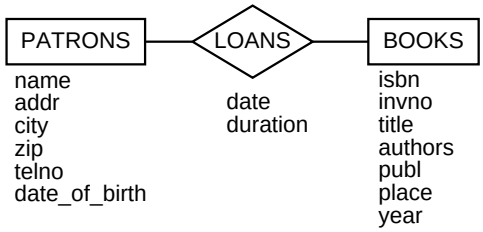

Figure 2.6: The E/R Diagram of the Town Library Database

year. For the PATRONS set of entities, it is easy to see that the sets $H =$ name telno date_of_birth and $L =$ name addr city date_of_birth are keys. Indeed, it is consistent with the usual interpretation of these attributes to assume that a reader can be uniquely identified by his name, his telephone number, and his date of birth. Note that the set H satisfies the minimality property. Assume, for example, that we drop the date_of_birth attribute. In this case, a father and a son who live in the same household and are both named "John Smith" cannot be distinguished through the values of the attributes name and telno. On the other hand, we may not drop the attribute telno because we can have two different readers with the same name and date of birth. Finally, we may not drop name from H because we could not distinguish between two individuals who live in the same household and have the same date of birth (for instance, between twins). Similar reasoning shows that L is also a key (see Exercise 10). □

Example 2.3.3 shows that it is possible to have several keys for a set of entities. One of these keys is chosen as the *primary key*; the remaining keys are *alternate keys*.

The primary key of a set of entities E is used by other constituents of the E/R model to refer to the entities of E.

As we now see, the definition of keys for sets of relationships is completely parallel to the definition of keys for sets of entities.

Definition 2.3.4 Let R be a set of relationships. A subset L of the set of attributes of R is a *key* of R if it satisfies the following conditions:

1. If $A(r) = A(r')$ for every attribute A of L, then $r = r'$ (*the unique identification property of relationships*).
2. No proper subset of L has the unique identification property (*the minimality property of keys of relationships*).

□

Note that the attributes that form a key of a set R of relationships are themselves either attributes of R or keys of the entities that participate in the relationships of R. The presence of the keys of the entities is necessary to indicate which entities actually participate in the relationships. There is no logical necessity that any particular key be chosen, but the reason for designating

one of the keys as the primary key is to make sure a single key is used to access entities of the corresponding set.

Example 2.3.5 For instance, if we designate

$H =$ name telno date_of_birth

as the primary key for PATRONS and invno as primary key for BOOKS, we obtain the following primary key for LOANS:

$K =$ name telno date_of_birth invno date

To account for the possibility that a single patron borrows the same book repeatedly, thereby creating several loan relationships, the date attribute is necessary to distinguish among them. □

Definition 2.3.6 A *foreign key* for a set of relationships is a set of attributes that is a primary key of a set of entities that participates in the relationship set. □

Example 2.3.7 The set of attributes name telno date_of_birth is a foreign key for the set of relationships LOANS because it is a primary key of PATRONS. □

We conclude this initial presentation of keys by stressing that the identification of the primary key and of the alternate keys is a semantic statement: It reflects our understanding of the role played by various attributes in the real world. In other words, choosing the primary key from among the available keys is a choice of the designer.

2.4 Participation Constraints

The E/R model allows us to impose constraints on the number of relationships in which an entity is allowed to participate. Let R be a set of relationships between the sets of entities $E_1, \ldots, E_n$. The database satisfies the participation constraint (E_j, u, v, R) if every entity e in E_j participates in at least u relationships and no more than v relationships.

Example 2.4.1 Suppose, for instance, that the college requires that a student complete at least one course and no more than 45 courses (during the entire duration of his or her studies). This corresponds to a participation constraint

(STUDENTS, 1, 45, GRADES).

If every student must choose an advisor, and an instructor may not advise more than 7 students, we have the participation constraints

(STUDENTS, 1, 1, ADVISING)

and

(INSTRUCTORS, 0, 7, ADVISING)

□

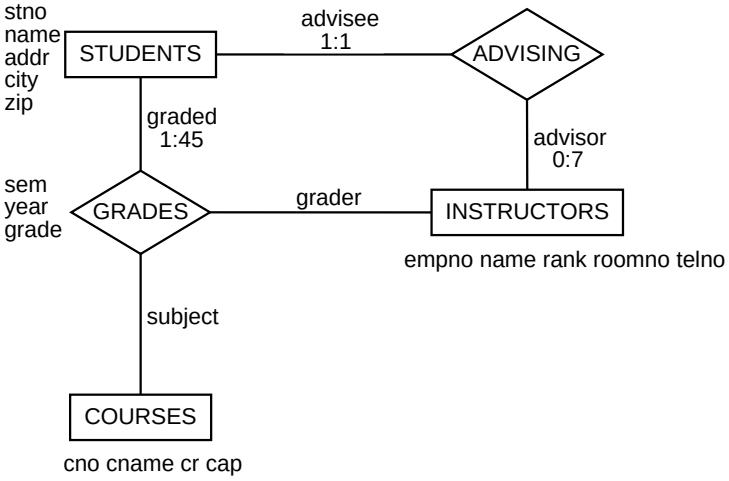

Figure 2.7: Participation Restrictions

If (E, u, v, R) is a participation constraint we may add $u : v$ to whatever other labels may be on the edge joining E to R. When there is no upper limit to the number of relationships in which an entity may participate, we write $u : +$.

Figure 2.7 reflects the roles and the participation constraints mentioned in Example 2.4.1.

Example 2.4.2 If a reader can have no more than 20 books on loan from the town library discussed in Example 2.3.3, then we impose the participation constraints

$$(\mathsf{PATRONS}, 0, 20, \mathsf{LOANS}) \text{ and } (\mathsf{BOOKS}, 0, 1, \mathsf{LOANS}).$$

The second restriction reflects the fact that a book is on loan to at most one patron. □

Let R be a set of binary relationships involving the sets of entities U and V. We single out several types of sets of binary relationships because they are popular in the business-oriented database literature. If every entity in U is related to exactly one entity in V, then we say that R is a set of *one-to-one* relationships. If an entity in U may be related to several entities of V, then R is a set of *one-to-many* relationships from U to V. If, on the other hand, many entities of U are related to a single entity in V, then R is a set of *many-to-one* relationships from U to V. And finally, if there are no such limitations between the entities of U and V, then R is a set of *many-to-many* relationships.

Example 2.4.3 The set of binary relationships LOANS between BOOKS and PATRONS considered in Example 2.4.2 is a one-to-many set of binary relationships. □

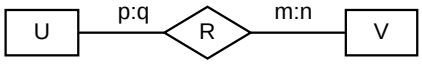

Figure 2.8: Binary Relationship with Participation Restrictions

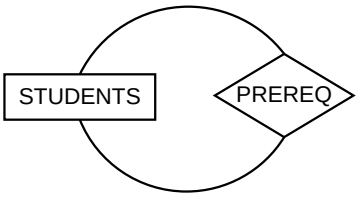

Figure 2.9: Recursive Set of Relationships

This terminology is limited to sets of binary relationships. We prefer to redefine these relationships using the participation constraints (U, p, q, R) and (V, m, n, R) that are imposed on the sets of entities by R (see Figure 2.8). This both makes the definitions very precise and generalizes the previous definitions to arbitrary relationships.

The set of relationships R from U to V is:

1. *one-to-one* if $p = 0, q = 1$ and $m = 0, n = 1$;
2. *one-to-many* if $p = 0, q > 1$ and $m = 0, n = 1$;
3. *many-to-one* if $p = 0, q = 1$ and $m = 0, n > 1$;
4. *many-to-many* if $p = 0, q > 1$ and $m = 0, n > 1$.

A recursive relationship is a binary relationship connecting a set of entities to itself.

Example 2.4.4 Suppose that we intend to incorporate in the college database information about prerequisites for courses. This can be accomplished by introducing the set of relationships PREREQ. If we assume that a course may have up to three prerequisites and place the appropriate participation constraint, then we obtain the E/R diagram shown in Figure 2.9.

□

2.5 Weak Entities

Suppose that we need to expand our database by adding information about student loans. This can be done, for instance, by adding a set of entities called LOANS. We assume that a student can have several loans (for the sake of this example, let us assume that a student can get up to 10 different loans). The existence of a loan entity in the E/R model of the college database is conditioned upon the existence of a student entity corresponding to the student to whom that loan was awarded. We refer to this type of dependency as an *existence dependency.*

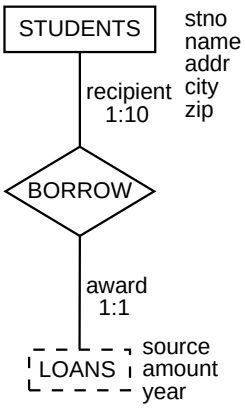

Figure 2.10: Representation of Weak Sets of Entities

The sets of entities STUDENTS and LOANS are related by the one-to-many sets of relationships BORROW.

If a student entity is deleted, the LOANS entities that depend on the student entity should also be removed. Note that the attributes of the LOANS entity set (source, amount, year) are not sufficient to identify an entity in this set. Indeed, if two students (say, the student whose student number is s_1 and the student whose student number is s_2) both got the "CALS" loan for 1993, valued at $1000, there is no way to distinguish between these entities using their own attributes. In other words, the set of entities LOANS does not have a key.

Definition 2.5.1 Let E, E' be sets of entities and let R be a set of relationships between E and E'. E is a *set of weak entities* if the following conditions are satisfied:

1. The set of entities E does not have a key, and
2. the participation constraint $(E, 1, k, R)$ is satisfied for some $k \geq 1$.

□

The second condition of Definition 2.5.1 states that no entity can exist in E unless it is involved in a relationship of R with an entity of E'. According to Definition 2.5.1, LOANS is a set of weak entities

Weak entity sets are represented in E/R diagrams by dashed boxes (see Figure 2.10).

Example 2.5.2 Consider a personnel database that contains a set of entities PERSINFO that contains personal information of the employees of a software company and a set of entities EMPHIST that contains employment history records of the employees. A set of recursive relationships REPORTING gives the reporting lines between employees; the set of entities EMPHIST is related to PERSINFO through the sets of relationships BELONGS_TO.

Note that the existence of an employment history entity in EMPHIST is

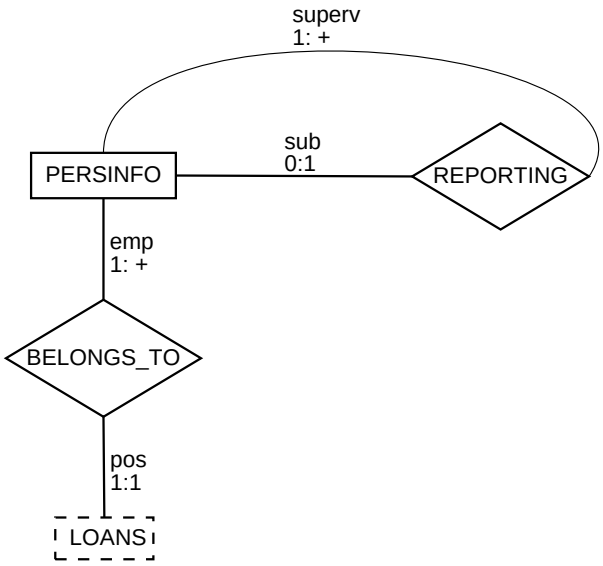

Figure 2.11: E/R Diagram of the EMPLOYEES database

conditioned upon the existence of a personal information entity in PERSINF. The E/R diagram of this database is shown in Figure 2.11. □

2.6 Is-a Relationships

We often need to work with subsets of sets of entities. Because the E/R model deals with sets of relationships, set inclusion must be expressed in these terms.

Let S, T be two sets of entities. We say that S **is-a** T if every entity of S is also an entity of T. In terms of the E/R model we have the set of relationships $R_{\textbf{is-a}}$. Pictorially, this is shown in Figure 2.12(a), where the representation for S is drawn below the one for T; we simplify this representation by replacing the diamond in this case by an arrow marked **is-a** directed from S to T as in Figure 2.12(b).

For example, foreign students are students, so we can use the notation FOREIGN_STUDENTS **is-a** STUDENTS.

Since every entity of S is also an entity of T the attributes of T are inherited by S. This property of the **is-a** relationships is known as the *descending inheritance property* of **is-a**.

Example 2.6.1 Consider UNDERGRADUATES and GRADUATES the sets of entities representing the undergraduate and the graduate students of the college. For undergraduate students we add sat as an extra attribute; for graduate students we add the attribute gre, which refers to the score obtained in the GRE examination. Both these sets of entities are linked to STUDENTS through the

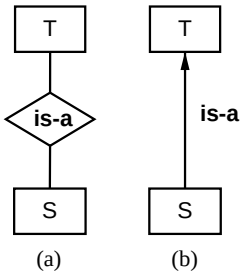

Figure 2.12: Representing an **is-a** Set of Relationships

is-a set of relationships (see Figure 2.13).

□

Example 2.6.2 Teaching assistants are both students and instructors, and therefore, the corresponding set of entities, TAs, inherits its attributes from both STUDENTS and INSTRUCTORS. □

This phenomenon described in Example 2.6.2 is called *multiple inheritance*, and certain precautions must be taken when it occurs. If S **is-a** U and S **is-a** V and both U and V have an attribute A, we must have $\mathrm{Dom}(U.A) = \mathrm{Dom}(V.A)$, because otherwise it would be impossible to have any meaning for the common restrictions of these attribute to S.

The **is-a** relation between sets of entities is transitive; that is, S **is-a** T and T **is-a** U imply S **is-a** U. To avoid redundancy in defining the **is-a** relation between entity sets (and, consequently, to eliminate redundancies involving the **is-a** relationships between entities), we assume that for no set of entities S do we have S **is-a** S.

The introduction of **is-a** relationships can be accomplished through two distinct processes, called *specialization* and *generalization*. Specialization makes a smaller set of entities by selecting entities from a set. Generalization makes a single set of entities by combining several sets whose attributes are those the original sets had in common.

Definition 2.6.3 A set of entities E' is derived from a set of entities through a *specialization* process if E' consists of all entities of E that satisfy a certain condition. □

If E' is obtained from E through specialization, then E' **is-a** E. In this case we may mark the arrow leading from E' to E by **is-a**(sp).

Example 2.6.4 The set TAs can be regarded as a specialization of both STUDENTS and INSTRUCTORS (see Figure 2.14). Therefore, entities of this set have all attributes applicable to INSTRUCTORS and STUDENTS and, in addi-

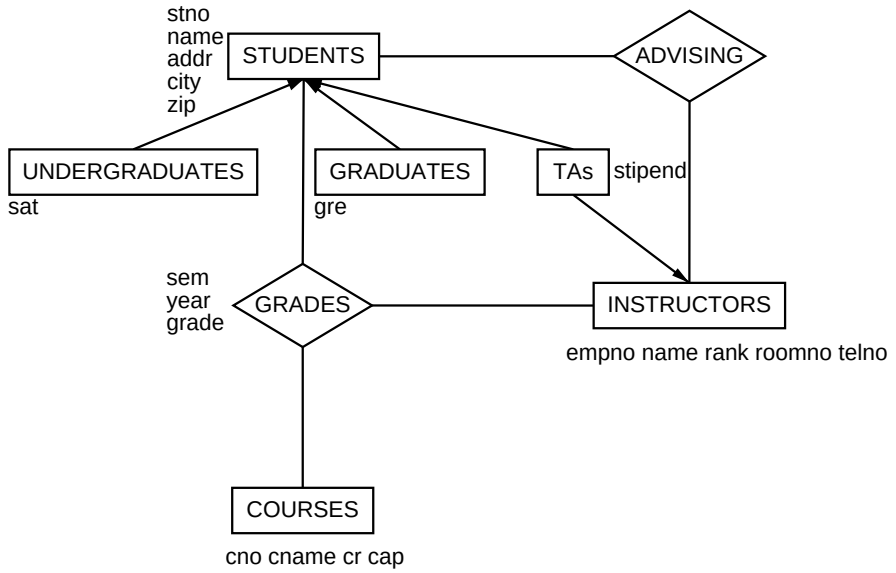

Figure 2.13: Representation of **is-a** Relationships

tion, their specific attribute stipend. □

Definition 2.6.5 Let $E_1, \ldots, E_n$ be n set of entities such that

1. no two distinct sets of entities E_i and E_j have an entity in common, and
2. there are some attributes that all entity sets have in common.

Let H be the set of attributes that all entities have in common..

The set of entities E is obtained by generalization from $E_1, \ldots, E_n$ if E consists of all entities that belong to one of the sets E_i and $\mathbf{Attr}(E) = H$. □

If E is obtained from $E_1, \ldots, E_n$ through generalization, we may mark the **is-a** arrows pointing from $E_1, \ldots, E_n$ to E by **is-a**(gen).

Example 2.6.6 Suppose that the construction of the college database begins from the existing sets of entities UNDERGRADUATES and GRADUATES. Then, the set of entities STUDENTS could have been obtained through generalization from the previous two sets (see Figure 2.14). □

The importance of the E/R methodology is that it allows the designer to organize and record his or her conception of the database. This enforces precision and facilitates unambiguous communication among all workers on a project.

Furthermore, it imposes discipline by requiring the designer to specify the entities and relationships, along with their attributes, and by insisting on a clear definition of the sets of relationships between sets of entities, rather than just assuming that they are somehow connected.

The **is-a** relationship imposes hierarchy on the sets of entities. Seeing which sets are generated by specialization and which by generalization helps expose

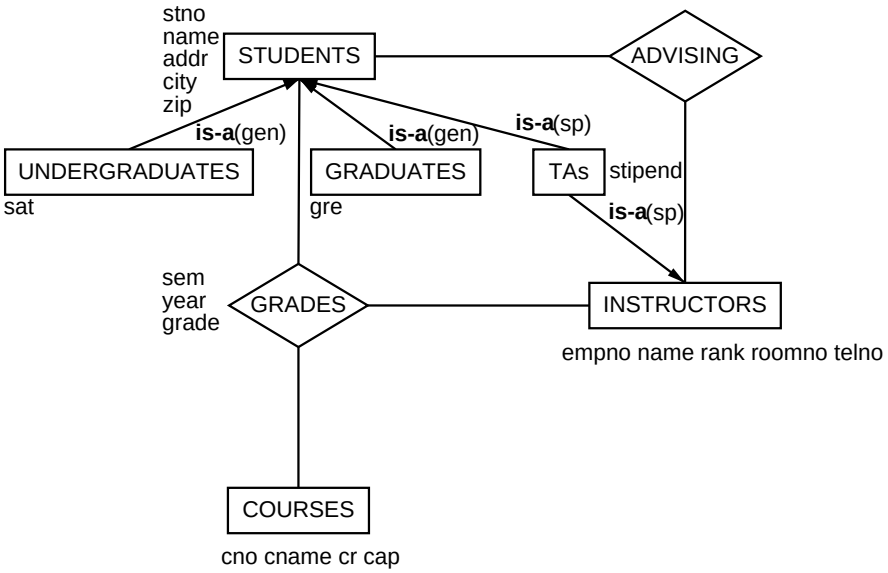

Figure 2.14: Specialization and Generalization

the underlying logic as seen by the designer.

In short, we use the E/R technique as a first step in a database project because it allows us to think about and modify the database before we commit to the relational design, which we discuss in the next chapter.

2.7 Exercises

1. Consider the following alternative designs for the college database:
   (a) Make GRADES a set of entities, and consider binary sets of relationships between GRADES and each of the sets of entities STUDENTS, COURSES, and INSTRUCTORS.
   (b) Replace the set of relationship GRADES with two binary sets of relationships: One such set should relate STUDENTS with COURSES and reflect the results obtained by students in the courses; another one should relate COURSES with INSTRUCTORS and reflect the teaching assignment of the instructors.

   Explain the advantages and disadvantages of these design choices.
2. Consider a database that has a set of entities CUSTOMERS that consists of all the customers of a natural gas distribution company. Suppose that this database also records the meter readings for each customer. Each meter reading has the date of the reading and the number read from the meter. Bills are generated after six consecutive readings.
   (a) Can you consider the readings to be weak entities?

(b) Draw the E/R diagram for this database; identify relevant participation constraints.

3. The data for our college will grow quite large. One of the techniques for dealing with the explosion of data is to remove those items that are not used. Describe how you would augment the college database to include information about when something was accessed (either read or written). To what will you attach this information? How? How detailed will the information you attach be? Why? What would you suggest be done with this information once it is available?

4. Design an E/R model for the patient population of a small medical office. The database must reflect patients, office visits, prescriptions, bills and payments. Explain your choice of attributes, relationships, and constraints.

5. Design an E/R model for the database of a bank. The database must reflect customers, branch offices, accounts, and tellers. If you like, you can include other features, such as deposits, withdrawals, charges, interest, transfers between accounts, etc. Explain your choice of attributes, relationships, and constraints.

6. Design an E/R model for the database of a car-rental business. Specify entities, relationships, attributes, keys, and cardinality constraints for this database and explain your design choices. Be sure to include such objects like vehicles, renters, rental locations, etc.

7. A small manufacturing company needs a database for keeping track of its inventory of parts and supplies. Parts have part numbers, names, type, physical characteristics, etc. Some parts used by the company are installed during the fabrication process as components of other parts that enter the device produced by the company. Parts are stored at several manufacturing facilities. The database must contain information about the vendors and must keep track of the orders placed with vendors. Use the E/R technique to design the database.

8. Let A, B, C, and D, be four entity sets linked by **is-a** relationships as shown in Figure 2.15. What is wrong with the choice of these relationships?

9. Let E_1, E_2 be two sets of entities.

   (a) Assume that E is a nonempty set of entities that is a specialization of both E_1 and E_2. Can you construct the generalization of the sets E_1 and E_2?

   (b) Suppose that E' is a generalization of E_1 and E_2. Can you construct a set of entities E'' that is a common specialization of E_1 and E_2? What can you say about $|E''|$?

10. Explain why the set L in example 2.3.3 is a key. Are there reasons for prefering H as the primary key and L as an alternate key?

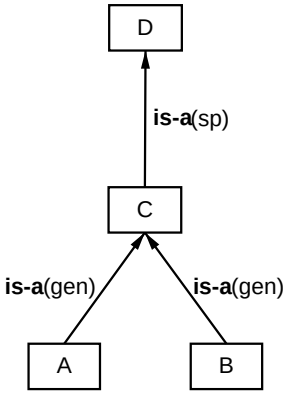

Figure 2.15: Hypothetical Use of Specialization and Generalization

2.8 Bibliographical Comments

The E/R model was introduced by P. P. Chen in his article [Chen, 1976]. Other important references on this topic are [Teorey, 1990; Elmasri and Navathe, 2006].

Chapter 3

The Relational Model

3.1 Introduction

The relational model is the mainstay of contemporary databases. This chapter presents the fundamental ideas of this model, which focus on data organization and retrieval capabilities.

Informally, the *relational model* consists of:

- A class of data structures referred to as *tables*.
- A collection of methods for building new tables starting from an initial collection of tables; we refer to these methods as *relational algebra operations*.
- A collection of *constraints* imposed on the data contained in tables.

3.2 Tables — The Main Data Structure of the Relational Model

The relational model revolves around a fundamental data structure called a *table*, which is a formalization of the intuitive notion of a table. For example, the schedule of a small college may look like:

SCHEDULE

dow	cno	roomno	time
'Mon'	'cs110'	84	5:00 p.m.
'Mon'	'cs450'	62	7:00 p.m.
'Wed'	'cs110'	65	10:00 a.m.
'Wed'	'cs310'	63	12:00 p.m.
'Thu'	'cs210'	63	2:00 p.m.
'Thu'	'cs450'	65	3:00 p.m.
'Thu'	'cs240'	84	5:00 p.m.
'Fri'	'cs310'	63	5:00 p.m.

When contemplating a table we distinguish three main components: the *name* of the table, in our case SCHEDULE, the *heading* of the table, with one entry for each column, in our case dow, cno, roomno, and time and the content of the table, i.e., the list of 8 rows specified above.

The members of the heading are referred to as *attributes*. In keeping with the practice of databases, if the heading H of the table consists of the attributes $A_1, \ldots, A_n$, then we write H as a string rather than a set, $H = A_1 \cdots A_n$.

Each attribute A has a special set that is attached to it called the *domain* of A that is denoted $\mathrm{Dom}(A)$. This domain comprises the set of values of the attribute; i.e., a value may occur in a column labeled by A only if it belongs to the set $\mathrm{Dom}(A)$. For example, in the table SCHEDULE considered above the domain of the attribute dow (for "day of the week") is the set that consists of the strings:

'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'

We need a name for each table, so we can refer to it. In principle, names are simply arbitrary strings of symbols chosen from some fixed alphabet (for example the modern Roman alphabet), augmented with certain special symbols, such as "_", "[", "]", "∪", etc., that we introduce from time to time as needed. However, it is always a good idea to choose names that are meaningful to the reader. Similar comments apply to the names of attributes. In most database systems, these names are case insensitive.

When speaking informally, we use descriptions based on the visual layout of a table T when it is displayed. So, a tuple t of T is called a *row* of T. Also, the set of values that occur under an attribute may be referred to as a *column* of T.

The term "relational model" reflects that fact that, from a mathematical point of view, the content of a table is what is known in mathematics as a *relation*. To introduce the notion of relation we need to define the Cartesian product of sets (sometimes called a *cross product*), a fundamental set operation.

Definition 3.2.1 Let $D_1, \ldots, D_n$ be n sets. The *Cartesian product of the sequence of sets* $D_1, \ldots, D_n$ is the set that consists of all sequences of the form $(d_1, \ldots, d_n)$, where $d_i \in D_i$ for $1 \leq i \leq n$.

We denote the Cartesian product of $D_1, \ldots, D_n$ by $D_1 \times \cdots \times D_n$.

A sequence that belongs to $D_1 \times \cdots \times D_n$ is referred to as an *n-tuple*, or simply as *tuple*, when n is clear from context. When n has small values we use special terms such as *pair* for $n = 2$, *triple* for $n = 3$, etc. □

The Cartesian product can generate rather large sets starting from sets that have a modest size. For example if D_1, D_2, D_3 are three sets having 1000 elements each, then $D_1 \times D_2 \times D_3$ contains 1,000,000,000 elements.

Example 3.2.2 Consider the domains of the attributes dow, cno, roomno and time:

$$\begin{aligned} \text{Dom}(\mathsf{dow}) &= \{\text{'Mon'}, \text{'Tue'}, \text{'Wed'}, \text{'Thu'}, \text{'Fri'}, \text{'Sat'}, \text{'Sun'}\} \\ \text{Dom}(\mathsf{cno}) &= \{\text{'cs110'}, \text{'cs210'}, \text{'cs240'}, \text{'cs310'}, \text{'cs450'}\} \\ \text{Dom}(\mathsf{roomno}) &= \{62, 63, 65, 84\} \\ \text{Dom}(\mathsf{time}) &= \{\text{8:00a.m.}, \ldots, \text{7:00p.m.}\} \end{aligned}$$

The Cartesian product of these sets:

$$\text{Dom}(\mathsf{dow}) \times \text{Dom}(\mathsf{cno}) \times \text{Dom}(\mathsf{roomno}) \times \text{Dom}(\mathsf{time})$$

consists of $7 \cdot 5 \cdot 4 \cdot 12 = 1680$ quadruples. □

If $H = A_1 \cdots A_n$ is a sequence of attributes, we refer to the Cartesian product $\text{Dom}(A_1) \times \cdots \times \text{Dom}(A_n)$ as the *set of H-tuples*. We will denote this set by $\mathbf{tupl}(H)$. Thus, $\mathbf{tupl}(\mathsf{dow\ cno\ roomno\ time})$ consists of 1680 quadruples.

Definition 3.2.3 A *relation* on the sets $D_1, \ldots, D_n$ is a subset ρ of the Cartesian product $D_1 \times \cdots \times D_n$. □

There is no requirement that the n sets be distinct. Many common relations are defined on $D_1 \times D_2$, where $D_1 = D_2 = D$; i.e., they are defined on $D \times D$. Perhaps the most common example of this is the equality relation, consisting of all pairs (a, a) for a in D.

Example 3.2.4 Consider the set $D = \{1, 2, 3, 4, 5, 6\}$ and the Cartesian product $D \times D$, which has 36 pairs. Certain of these pairs (a, b) have the property that a is less than b, i.e., that they satisfy the relation $a < b$. With a little bit of counting, we see that there are 15 such pairs.

One way to characterize this set is to describe it is operationally. We could say that if a and b are in D, then $a < b$ if there is some number k in D such that $a + k = b$. This has the advantage of being concise.

However, there is another way to describe $<$ on this set: we could list out all 15 pairs (a, b) of $D \times D$ such that $a < b$. If we do this in a vertical list, we get (in no particular order)

$(3, 4)$
$(1, 2)$
$(2, 6)$
$(2, 5)$
$(1, 6)$
$(2, 3)$
$(1, 3)$
$(2, 4)$
$(1, 5)$
$(5, 6)$
$(3, 6)$
$(4, 5)$
$(4, 6)$
$(3, 5)$
$(1, 4)$

With a little reformatting, to remove all those parentheses and commas, this same list of pairs becomes a table of two columns and 15 rows, where each row lists two elements of D, such that the first is less than the second. Furthermore, just as in the list above, all pairs with the first element less than the second occur as some row in this table.

3	4
1	2
2	6
2	5
1	6
2	3
1	3
2	4
1	5
5	6
3	6
4	5
4	6
3	5
1	4

Thus, we have a table that lists precisely the pairs of $D \times D$ that comprise the $<$ relation. In just this same manner, we can list out all the tuples of any relation defined on finite sets as rows of a table. It is this correspondence between tables and relations that is at the heart of the name "relational model." ∎

Example 3.2.5 Using the sets

$$\mathrm{Dom}(\mathsf{dow}), \mathrm{Dom}(\mathsf{cno}), \mathrm{Dom}(\mathsf{roomno}), \mathrm{Dom}(\mathsf{time})$$

we can define course schedules as relations. One possible course schedule is the relation ρ that consists of the following 8 quadruples:

('Mon', 'cs110', 84, '5:00 p.m.'), ('Mon', 'cs450', 62, '7:00 p.m.'),
('Wed', 'cs110', 65, '10:00 a.m.'), ('Wed', 'cs310', 63, '12:00 p.m.'),
('Thu', 'cs210', 63, '2:00 p.m.'), ('Thu', 'cs450', 65, '3:00 p.m.'),
('Thu', 'cs240', 84, '5:00 p.m.'), ('Fri', 'cs310', 63, '5:00 p.m.')

□

If $D_1, D_2, \ldots, D_n$ are n sets with $k_1, \ldots, k_n$ elements, respectively, then there are $2^{k_1 k_2 \cdots k_n}$ relations that can be defined on these sets. The number of relations that can be defined on relatively small sets can be astronomical. For example, if each of D_1, D_2 and D_3 has ten elements, then there are 2^{1000} relations that can be defined on D_1, D_2, D_3.

It is clear now that the content of a table T having the heading $H = A_1 \cdots A_n$ is a relation that consists of tuples from $\mathbf{tupl}(H)$, and this is the essential part of the table.

During the life of a database, the constituent tables of the database may change through insertions or deletions of tuples, or changes to existing tuples. Thus, at any given moment we may see a different picture of the database, which suggests the need of introducing the the notion of relational database instance.

Definition 3.2.6 Let $H_1, \ldots, H_n$ be n sets of attributes. An *relational database instance* is a finite collection of tables $T_1, \ldots, T_n$ that have the headings $H_1, \ldots, H_n$, respectively, such that all names of the tables are distinct. □

Definition 3.2.7 The tables T and S are *compatible* if they have the same headings. □

Implicit in the definition of tables is the fact that *tables do not contain duplicate tuples*. This is not a realistic assumption, and we shall remove it later, during the study of SQL, the standard query language for relational databases.

The same relational attribute may occur in several tables of a relational database. Therefore, it is important to be able to differentiate between attributes that originate from different tables; we accomplish this using the following notion.

Definition 3.2.8 Let T be a table. A *qualified attribute* is an attribute of the form $T.A$. For every qualified attribute of the form $T.A$, $\mathrm{Dom}(T.A)$ is the same as $\mathrm{Dom}(A)$. □

Example 3.2.9 The qualified attributes of the table SCHEDULE are

SCHEDULE.dow, SCHEDULE.cno, SCHEDULE.roomno, SCHEDULE.time

□

3.2.1 Projections

For a tuple t of a table T having the heading H we may wish to consider only some of the attributes of t while ignoring others. If L is the set of attributes we are interested in, then $t[L]$ is the corresponding tuple, referred to as the projection of t on L.

Example 3.2.10 Let $H =$ dow cno roomno time be the set of attributes that is the heading of the table SCHEDULE introduced above. The tuple

$$t = (\text{'Mon'}, \text{'cs110'}, 84, \text{'5:00 p.m.'})$$

can be restricted to any of the sixteen subsets of the set H. For example, the restriction of t to the set $L =$ dow roomno is the tuple $t[L] = (\text{'Mon'}, 84)$, and the restriction of t to the set $K =$ cno room time is $(\text{'cs110'}, 84, \text{'5:00 p.m.'})$

The restriction of t to H is, of course, t itself. Also, the restriction of t to the empty set is $t[\emptyset] = ()$, that is, the empty sequence. □

By extension, the table T itself can be projected onto L, giving a new table named $T[L]$, with heading L, consisting of all tuples of the form $t[L]$, where t is a tuple in T; i.e., the rows of $T[L]$ are obtained by projecting the rows of T on L. Projecting a table often creates duplicate rows, but within the context of the relational model, which is based on sets, only one copy of each row appears in the table, as shown in the second projection of Example 3.2.11.

Example 3.2.11 The projection of the table SCHEDULE on the set of attributes dow cno is

SCHEDULE[dow cno]

dow	cno
'Mon'	'cs110'
'Mon'	'cs450'
'Wed'	'cs110'
'Wed'	'cs310'
'Thu'	'cs210'
'Thu'	'cs450'
'Thu'	'cs240'
'Fri'	'cs310'

The projection of SCHEDULE on the attribute dow gives the following table.

SCHEDULE[dow]

dow
'Mon'
'Wed'
'Thu'
'Fri'

□

3.3 Transforming an E/R Design into a Relational Design

The design of a database formulated in the E/R model can be naturally translated into the relational model. We show how to translate both sets of entities and sets of relationships into tables.

From time to time, it is necessary to assume that a set of entities or a set of relationships has a primary key. For any that does not, we can induce a key by arbitrarily assigning a unique identifier to each element. In "real world" examples, this is generally accomplished by picking a sequence and assigning the next unused element to each entity as it enters the system. This can easily be seen to be a key that we may designate to be the primary key.

For example, whenever a new patron applies for a card at the library, the library may assign a new, distinct number to the patron; this set of numbers could be the primary key for the entity set PATRONS. Similarly, each time a book is loaned out, a new loan number could be assigned, and this set of numbers could be the primary key for the set of relationships LOANS. Note, however, that if we introduce these identifiers we in fact have made a small change to the original in that we have actually added a new attribute to PATRONS and to LOANS.

Consider a set of entities named E that has the set of attributes $H = A_1 \dots A_n$. Its translation is a table named E that has the heading $A_1 \dots A_n$. For each entity e, we include in the table a tuple t_e by that has the components $A_1(e), \dots, A_n(e)$.

In other words, for every entity e there is a tuple in the table E consisting of the tuple $(A_1(e), \dots, A_n(e))$. For instance, if e is an entity that represents a student (that is $e \in$ STUDENTS) and

stno(e)	=	'2415'
name(e)	=	'Grogan A. Mary'
addr(e)	=	'8 Walnut St.'
city(e)	=	'Malden'
state(e)	=	'MA'
zip(e)	=	'02148',

then e is represented in the table named STUDENTS by the row:

('2415', 'Grogan A. Mary', '8 Walnut St.', 'Malden', 'MA', '02148').

The set of entities STUDENTS is translated into the table named STUDENTS shown in Figure 3.1.

While in the E/R model we dealt with two types of basic constituents, entity sets and relationship sets, in the relational model, we deal only with tables, and we use these to represent both sets of entities and sets of relationships. Thus, it is necessary to reformulate the definition of keys in this new setting. The conditions imposed on keys are obvious translations of the conditions formulated in Definition 2.3.1.

Definition 3.3.1 Let T be a table that has the heading H. A set of attributes K is a *key* for T if $K \subseteq H$ and the following conditions are satisfied:

1. For all tuples u, v of the table, if $u[K] = v[K]$, then $u = v$ (*unique identification property*).
2. There is no proper subset L of K that has the unique identification property (*minimality property*).

□

STUDENTS

stno	name	addr	city	state	zip
1011	Edwards P. David	10 Red Rd.	Newton	MA	02159
2415	Grogan A. Mary	8 Walnut St.	Malden	MA	02148
2661	Mixon Leatha	100 School St.	Brookline	MA	02146
2890	McLane Sandy	30 Cass Rd.	Boston	MA	02122
3442	Novak Roland	42 Beacon St.	Nashua	NH	03060
3566	Pierce Richard	70 Park St.	Brookline	MA	02146
4022	Prior Lorraine	8 Beacon St.	Boston	MA	02125
5544	Rawlings Jerry	15 Pleasant Dr.	Boston	MA	02115
5571	Lewis Jerry	1 Main Rd.	Providence	RI	02904

INSTRUCTORS

empno	name	rank	roomno	telno
019	Evans Robert	Professor	82	7122
023	Exxon George	Professor	90	9101
056	Sawyer Kathy	Assoc. Prof.	91	5110
126	Davis William	Assoc. Prof.	72	5411
234	Will Samuel	Assist. Prof.	90	7024

COURSES

cno	cname	cr	cap
cs110	Introduction to Computing	4	120
cs210	Computer Programming	4	100
cs240	Computer Architecture	3	100
cs310	Data Structures	3	60
cs350	Higher Level Languages	3	50
cs410	Software Engineering	3	40
cs460	Graphics	3	30

GRADES

stno	empno	cno	sem	year	grade
1011	019	cs110	Fall	2001	40
2661	019	cs110	Fall	2001	80
3566	019	cs110	Fall	2001	95
5544	019	cs110	Fall	2001	100
1011	023	cs110	Spring	2002	75
4022	023	cs110	Spring	2002	60
3566	019	cs240	Spring	2002	100
5571	019	cs240	Spring	2002	50
2415	019	cs240	Spring	2002	100
3442	234	cs410	Spring	2002	60
5571	234	cs410	Spring	2002	80
1011	019	cs210	Fall	2002	90
2661	019	cs210	Fall	2002	70
3566	019	cs210	Fall	2002	90
5571	019	cs210	Spring	2003	85
4022	019	cs210	Spring	2003	70
5544	056	cs240	Spring	2003	70
1011	056	cs240	Spring	2003	90
4022	056	cs240	Spring	2003	80
2661	234	cs310	Spring	2003	100
4022	234	cs310	Spring	2003	75

ADVISING

stno	empno
1011	019
2415	019
2661	023
2890	023
3442	056
3566	126
4022	234
5544	023
5571	234

Figure 3.1: An Instance of the College Database

If several keys exist for a table, one of them is designated as the *primary key* of the table; the remaining keys are *alternate keys*. The main role of the primary key of a table T is to serve as a reference for the tuples of T that can be used by other tables that refer to these tuples.

Example 3.3.2 The table that results from the translation of the set of entities PATRONS introduced in Example 2.3.3 has the keys

$$K = \text{name telno date\_of\_birth}$$

and

$$L = \text{name address city date\_of\_birth.}$$

If we consider K to be the primary key, then L is an alternate key.

As a practical matter, if K_1 and K_2 are both keys, where K_1 has fewer attributes than K_2, we would prefer K_1 as the primary key. □

Translating sets of relationships is a little more intricate than translating tables. Let R be a set of relationships that relates the set of entities $E_1, \ldots, E_n$. Suppose that every set E_i has its own primary key K_i for $1 \leq i \leq n$ and that no two such keys have an attribute in common. We exclude, for the moment, the **is-a** relationship and the dependency relationship that relates sets of weak entities to sets of regular entities. When translating relationships, the entities involved are represented by their primary key values.

If the set of attributes of R itself is $B_1, \ldots, B_k$, then a relationship r of R relates the entities $e_1, \ldots, e_n$ with some values, say $b_1, \ldots, b_k$. In making the translation of this particular relationship, each entity e_i is represented by its primary key, $e_i[K_i]$, which may comprise several values, $e^i_1, \ldots, e^i_{m^i}$. To simplify, we will write $\vec{e}_i$ for the primary key of e_i. The value of the relationship itself is represented by the value b_j of each attribute B_j. So, r can be translated to a tuple $w_r = (\vec{e}_1, \ldots, \vec{e}_n, b_1, \ldots, b_k)$. In other words, the translation W_R of the set of relationships R is defined on the set of all attributes that appear in the primary keys of the entities, $K_1, \ldots, K_n$, as well as attributes $B_1, \ldots, B_k$; and the tuple w_r is put together from the values of the primary keys of the participating entities and the values of the attributes of the relationship r.

Example 3.3.3 Consider, for example, the relationship g that belongs to the set of relationships GRADES, that relates STUDENTS, COURSES, and INSTRUCTORS. Further, assume that this relationship involves the student whose student number is '1011', the instructor whose employee number is '019', and the course whose number is 'cs110'. Further, assume that

$$\begin{aligned} \text{sem}(g) &= \text{'Fall'} \\ \text{year}(g) &= \text{'2001'} \\ \text{grade}(g) &= 40. \end{aligned}$$

Then, the relationship g will be represented by the tuple

$$w_g = (\text{'1011'}, \text{'019'}, \text{'cs110'}, \text{'Fall'}, \text{'2001'}, 40).$$

□

Formally, w_r is given by:

$$w_r(A) = \begin{cases} A(e_i) & \text{if } A \text{ is in } K_i \text{ for some } i, 1 \leq i \leq n \\ A(r) & \text{if } A \text{ is in } \{B_1, \ldots, B_k\}. \end{cases}$$

In turn, the set R is translated into a table named R whose heading contains $B_1, \ldots, B_k$ as well as all attributes that occur in a key $K_1, \ldots, K_n$. The content of this table consists of all tuples of the form w_r for each relationship r.

The collection of tables shown in Figure 3.1 represents an instance of the college database obtained by the transformation of the E/R design shown in Figure 2.5.

If E is a set of weak entities linked by a dependency relationship R to a set of entities E', then we map both the set of entities E and the set of relationships R to a single table T defined as follows. If K is the primary key of the table T' that represents the set of entities E', we define H to be the set of attributes that includes the attributes of E and the attributes of K. The content of the table T consists of those tuples t in $\mathbf{tupl}(H)$ such that there exists an entity e' in E' and a weak entity e in E such that

$$t(A) = \begin{cases} A(e) & \text{if } A \text{ is an attribute of } E \\ A(e') & \text{if } A \text{ belongs to } K. \end{cases}$$

Example 3.3.4 Consider the set of weak entities LOANS dependent on the set STUDENTS. Assuming that the primary key of STUDENTS is stno, both the relationship GRANTS and the weak set of entities LOANS are translated into the table named LOANS:

LOANS

stno	source	amount	year
1011	CALS	1000	2002
1011	Stafford	1200	2003
3566	Stafford	1000	2002
3566	CALS	1200	2003
3566	Gulf Bank	2000	2003

□

Example 3.3.5 In Example 2.5.2 we discussed the E/R design of a personnel database. Recall that we had the set of entities PERSINFO and the set of weak entities EMPHIST linked to PERSINFO through the set of relationships BELONGS_TO. In addition, we had the set of relationships REPORTING. These components of the E/R model are translated into three tables represented in Figure 3.2. □

The translation of a set of entities involved in an **is-a** relationship depends on the nature of the relationship (generalization or specialization).

Suppose that a set of entities is obtained by generalization from the collection of sets of entities $E_1, \ldots, E_n$, such that no two distinct sets of entities E_i and E_j have an entity in common. In addition we assume that there are attributes that are shared by all sets $E_1, \ldots, E_n$ and we denote the set of all such attributes by H.

PERSINFO

empno	ssn	name	address	city	zip	state
1000	'340-90-5512'	'Natalia Martins'	'110 Beacon St.'	'Boston'	'02125'	'MA'
1005	'125-91-5172'	'Laura Schwartz'	'40 Tremont St.'	'Newton'	'02661'	'MA'
1010	'016-70-0033'	'John Soriano'	'10 Whittier Rd.'	'Lexington'	'02118'	'MA'
1015	'417-52-5751'	'Kendall MacRae'	'4 Maynard Dr.'	'Cambridge'	'02169'	'MA'
1020	'311-90-6688'	'Rachel Anderson'	'55 Columbus St.'	'Boston'	'02123'	'MA'
1025	'671-27-5577'	'Richard Laughlin'	'37 Greenough St.'	'Somerville'	'02060'	'MA'
1030	'508-56-7700'	'Danielle Craig'	'72 Dove Rd.'	'Boston'	'02225'	'MA'
1035	'870-50-5528'	'Abby Walsh'	'717 Park St.'	'Roxbury'	'02331'	'MA'
1040	'644-21-0887'	'Bailey Burns'	'35 White Pl.'	'Cambridge'	'02169'	'MA'

EMPHIST

empno	position	dept	appt_date	term_date	salary
1000	'President'	null	'1-oct-1999'	null	150000
1005	'Vice-President'	'DB'	'12-oct-1999'	null	120000
1010	'Vice-President'	'WWW'	'1-jan-2000'	null	120000
1015	'Senior Engineer'	'DB'	'25-oct-1999'	null	100000
1020	'Engineer'	'DB'	'1-nov-1999'	null	70000
1025	'Programmer'	'DB'	'10-mar-2000'	null	70000
1030	'Senior Engineer'	'WWW'	'10-jan-2000'	null	90000
1035	'Programmer'	'WWW'	'20-feb-2000'	null	75000
1040	'Programmer'	'WWW'	'1-mar-2000'	null	70000

REPORTING

empno	superv
1000	null
1005	1000
1010	1000
1015	1005
1020	1005
1025	1005
1030	1010
1035	1010
1040	1010

Figure 3.2: An Instance of the Employee Database

If E_i is translated into a table T_i, having the heading H_i for $1 \leq i \leq n$, then E is represented by the table T that contains every projections of every tuple of T_i on the set H.

Example 3.3.6 Assume that we want to form a table named STUDENTS from the tables named UNDERGRADUATES and GRADUATES in the college database. If these tables have the form

UNDERGRADUATES

stno	name	addr	city	state	zip	major
1011	Edwards P. David	10 Red Rd.	Newton	MA	02159	CS
2415	Grogan A. Mary	8 Walnut St.	Malden	MA	02148	BIO
2661	Mixon Leatha	100 School St.	Brookline	MA	02146	MATH
2890	McLane Sandy	30 Cass Rd.	Boston	MA	02122	CS
3442	Novak Roland	42 Beacon St.	Nashua	NH	03060	CHEM

GRADUATES

stno	name	addr	city	state	zip	qualdate
3566	Pierce Richard	70 Park St.	Brookline	MA	02146	2/1/92
4022	Prior Lorraine	8 Beacon St.	Boston	MA	02125	11/5/93
5544	Rawlings Jerry	15 Pleasant Dr.	Boston	MA	02115	2/1/92
5571	Lewis Jerry	1 Main Rd.	Providence	RI	02904	11/5/93

then the table that represents the set of entities STUDENTS obtained by generalization from UNDERGRADUATES and GRADUATES is the one shown in Figure 3.1. □

If the set of entities E' is obtained by specialization from the set of entities E, the heading of the table that represents E' must include the attributes of E plus the extra attributes that are specific to E' whenever such attributes exist (see Figure 3.3).

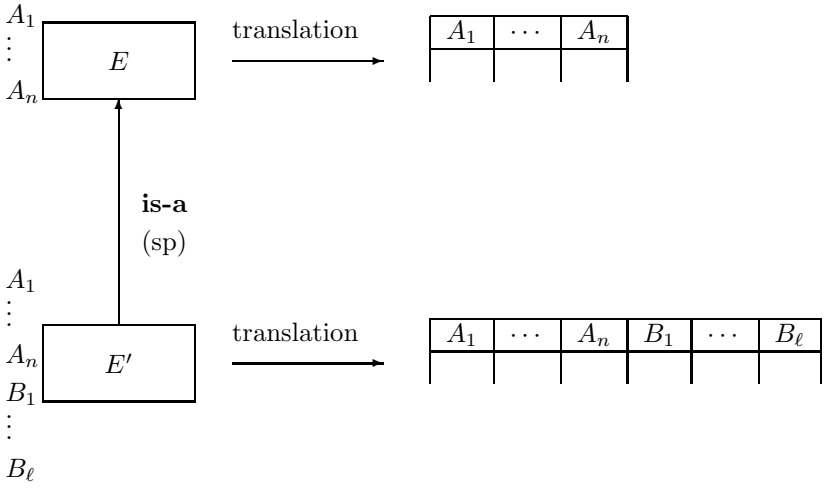

Figure 3.3: Translation of Specialization

Example 3.3.7 The heading of the table that represents the set of entities TA consists of the attributes stno, name, addr, city, state, zip, empno, rank, roomno, telno, stipend. The extension of the table that results from the translation of TA consists of the translation of all entities that belong to both STUDENTS and INSTRUCTORS. □

3.4 Entity and Referential Integrity

If student course registrations are recorded using the structure of this database, a tuple must be inserted into the table GRADES. Naturally, at the beginning of the semester there is no way to enter a numerical grade; we need a special value to enter in the field grade of the table GRADES that indicates that the grade component of the tuple is not yet determined. Such a value is called a *null value*. We represent this value by **null**.

A null value can have a significant semantic content: It may indicate that a component of a tuple is not defined yet (as is the case with the previous example), or that a certain attribute is inapplicable to a tuple, or that the value of the component of the tuple is unknown. Unfortunately, it is not always possible to tell which of these three intrepretations is intended whan a **null** value is encountered. This can lead to some serious problems in practical situations.

Example 3.4.1 Suppose that we need to expand the table STUDENTS by adding the relational attributes SAT and GRE. The first attribute is applicable to undergraduates, while the second can be applied only to graduate students. Therefore, every tuple that represents an undergraduate student has a null component for the GRE attribute, and every tuple that represents a graduate student has a null component for the SAT attribute. □

Null values cannot be allowed to occur as tuple components corresponding to the attributes of the primary key of a table, regardless of the semantic content of a null value. Sometimes the primary key is used to compute the memory address of a tuple. Therefore, the presence of null components in a tuple would jeopardize the role of the primary key in the physical placement of the tuples in memory. Also, such null values would interfere with the role of the primary key of "representing" the tuple in its relationships with other data in the database. This general requirement for relational databases is known as the *entity integrity rule*.

Recall that every table that represents a set of relationships R contains references to the sets of entities involved $E_1, \ldots, E_n$. These references take the form of the primary keys of $E_1, \ldots, E_n$. For instance, the table GRADES contains the attributes stno, empno, and cno, which are primary keys for STUDENTS, INSTRUCTORS, and COURSES, respectively. It is natural to assume that the student number stno component of a tuple of the table GRADES refers to the student number component of a tuple that is actually in the table STUDENTS, which is the place where student records are kept. This requirement (and similar requirements involving references to the tables COURSES and INSTRUCTORS) is formalized by the notion of *referential integrity*.

To define the concept of referential integrity, we need to introduce the notion of a *foreign key*.

Definition 3.4.2 An *S-foreign key* for a table T with heading H is a set of attributes L included in H that is the primary key for some other table S of the relational database. We omit the mention of S when it is clear from context, and we refer to a S-foreign key of a table T simply as a foreign key. □

Although a foreign key in a table T must be a primary key of some table S in the database, it may or may not be a part of the primary key of T. Suppose, for instance, that the college database contains a table named ROOMS that lists all the rooms of the campus. If the primary key of this table is roomno, then this attribute is a ROOMS-foreign key for the INSTRUCTORS table.

The relational model has the following fundamental rule.

Referential Integrity Rule: If L is an S-foreign key for a table T, only the following two cases may occur for each tuple t of T:

1. Either all components of $t[L]$ are **null**, or
2. there is a tuple s in S such that $t[L] = s[L]$.

This rule says that if a relationship refers to a row that could be in another table, S, then that row must be present in S.

Example 3.4.3 Since roomno is a foreign key for INSTRUCTORS, any non-

null value that occurs under this attribute in the table INSTRUCTORS must appear in the table ROOMS. This corresponds to the real-world constraint that either an instructor has no office, in which case the roomno-component is **null**, or the instructor's office in one of the rooms of the college. □

Of course, if an S-foreign key is a part of the primary key of a table T (as is the case with stno for GRADES, for example), then **null** values are not permitted in T under the attributes of the S-foreign key.

3.5 Metadata

Metadata is a term that refers to data that describes other data. In the context of the relational model, metadata are data that describe the tables and their attributes.

The relational model allows a relational database to contain tables that describe the database itself. These tables are known as *catalog tables*, and they constitute the *data catalog* or the *data dictionary* of the database.

Typically, the catalog tables of a database include a table that describes the names, owners, and some parameters of the headings of the data tables of the database. The owner of a table is relevant in multi-user relational database systems, where some users are permitted only limited access to tables they do not own.

For example, a catalog table named SYSCATALOG that describes the tables of the college database might look like:

SYSCATALOG

owner	tname	dbspacename
dsim	courses	system
dsim	students	system
dsim	instructors	system
dsim	grades	system
dsim	advising	system
sys	syscolumns	system
⋮	⋮	⋮

In the table SYSCATALOG the attribute owner describes the creator of the table; this coincides, in general, with the owner of that table. The attribute tname gives the name of the table, while dbspacename indicates the memory area (also known as the table space) where the table was placed.

Note that the above table mentions the table SYSCOLUMNS (recall that table names are case insensitive). SYSCOLUMNS describes various attributes and domains that occur in the user's tables. For example, for the college database, the table may look like:

SYSCOLUMNS

owner	cname	tname	coltype	nulls	length	in_pr_key
dsim	cno	courses	char	N	5	Y
dsim	cname	courses	char	Y	20	N
dsim	cr	courses	smallint	Y	2	N
dsim	cap	courses	integer	Y	4	N
dsim	stno	grades	char	N	10	Y
dsim	empno	grades	char	N	11	N
dsim	cno	grades	char	N	5	Y
dsim	sem	grades	char	N	6	Y
dsim	year	grades	integer	N	4	Y
dsim	grade	grades	integer	Y	4	N
⋮	⋮	⋮	⋮	⋮	⋮	⋮

The attributes cname and tname give the name of the column (attribute) and the name of table where the attribute occurs. The nature of the domain (character or numeric) is given by the attribute coltype and the size in bytes of the values of the domain is given by the attribute length. The attribute nulls specifies whether or not **null** values are allowed. Finally, the attribute in_pr_key indicates whether the attribute belongs to the primary key of the table tname.

The access to and the presentation of metadata is highly dependent on the specific database system. We examine the approach taken by ORACLE in section 5.24.

The relational model currently dominates all database systems, and it is likely to continue to do so for quite some time. Researchers are continually producing enhancements to the model, adding, e.g., object-oriented and web-centered features. Some of these features are already implemented in contemporary database systems, as we will see when we discuss ORACLE in detail.

3.6 Exercises

1. Convert the alternative E/R models for the college database discussed in Exercise 1 of Chapter 2 to a relational design.
2. Convert the E/R design of the database of the customers of the natural gas distribution company to a relational design. Specify the keys of each relation.
3. Suppose that the set of entities E' is obtained by specialization from the set of entities E and that
$$\begin{aligned} \tau &= (T, A_1 \dots A_n, \rho), \\ \tau' &= (T', A_1 \dots A_n B_1 \dots B_\ell, \rho') \end{aligned}$$
are the tables that result from the translation of ρ and ρ', respectively. Show that if e is an entity from $E - E'$ and t is the tuple that results from the translation of e, then $t \in \rho - \rho'[A_1 \dots A_n]$.

3.7 Bibliographical Comments

The relational model was introduced by E. F. Codd in [Codd, 1970]. A revised and extended version of the relational model is discussed in [Codd, 1990]. Interesting reflections on the relational model can be found in [Date and Darwen, 1993] and in [Date, 1990].

Chapter 4

Data Retrieval in the Relational Model

4.1 Introduction

Tables are more than simply places to store data. The real interest in tables is in how they are used. To obtain information from a database, a user formulates a question known as a "query." For example, if we wanted to construct an honor roll for the college for Fall 2002, we could examine the GRADES table and select all students whose grades are above some threshold, say 90. Note that the result can again be stored in a table. In this case, every tuple in the resultant table actually appears in the original table. However, if we wanted to know the names of the students in this table, we cannot find it out directly, as students are represented only by their student numbers in the GRADES table. We have to add some information from the STUDENTS table to find their names. The result can again be stored in a table, which we can call HONOR_ROLL.

In general, the method of working with relational databases is to modify and combine tables using specific techniques. These techniques have been studied and, of course, have names. For example, the method above that generates the sub-table of GRADES is an example of a "selection." This table can be thought of as an "intermediate result" along the path of obtaining HONOR_ROLL. The method of combining this intermediate result with STUDENTS is known as "joining." These and various other methods are what we study under the name "relational algebra."

Relational algebra is thus a collection of methods for building new tables starting from existing ones. These methods are referred to as "operations"

on the tables. The interest in relational algebra is clear: Because a relational database instance is a finite set of tables, and the answer to a query is again a table, we need methods for constructing the tables corresponding to our queries.

Traditionally, relational algebra defines the minimal retrieval capabilities of a relational database system. Thus, any system that purports to be a relational database management system must provide retrieval capabilities that are at least as powerful as the operations of relational algebra.

We introduce the operations of relational algebra one by one. For each, we specify how the operation acts on the contents of the tables involved. However, tables comprise more than just their contents, so to make the specification of an operation complete, we must also specify the heading of the resultant table and its name.

4.1.1 Renaming of Tables and Attributes

In building new tables, sometimes we need to create copies of existing tables. Such a copy has the same extension (that is, contains the same tuples) as the original table; the new copy must have a different name. In addition, for technical reasons, attributes of the new table may be different, provided each has the same domain as the corresponding attribute of the original table.

Definition 4.1.1 Let T be a table having the heading $A_1 \cdots A_n$. The table T' is obtained from T by *renaming* if $T' \neq T$, the heading of T' is $B_1 \cdots B_n$, where $\operatorname{Dom} B_i = \operatorname{Dom} A_i$, for $1 \leq i \leq n$ and the tables T and T' have the same content.

We denote that T' was obtained from T through renaming by writing

$$T'(B_1, \ldots, B_n) := T(A_1, \ldots, A_n).$$

If $B_1 = A_1, \ldots, B_n = A_n$, then we may write $T' := T$. In this case we refer to T' as an *alias* of T. □

Example 4.1.2 Suppose that we need an alias of the COURSES table. If we write

SUBJECTS := COURSES,

then we create the table:

SUBJECTS

cno	cname	cr	cap
cs110	Introduction to Computing	4	120
cs210	Computer Programming	4	100
cs240	Computer Architecture	3	100
cs310	Data Structures	3	60
cs350	Higher Level Languages	3	50
cs410	Software Engineering	3	40
cs460	Graphics	3	30

□

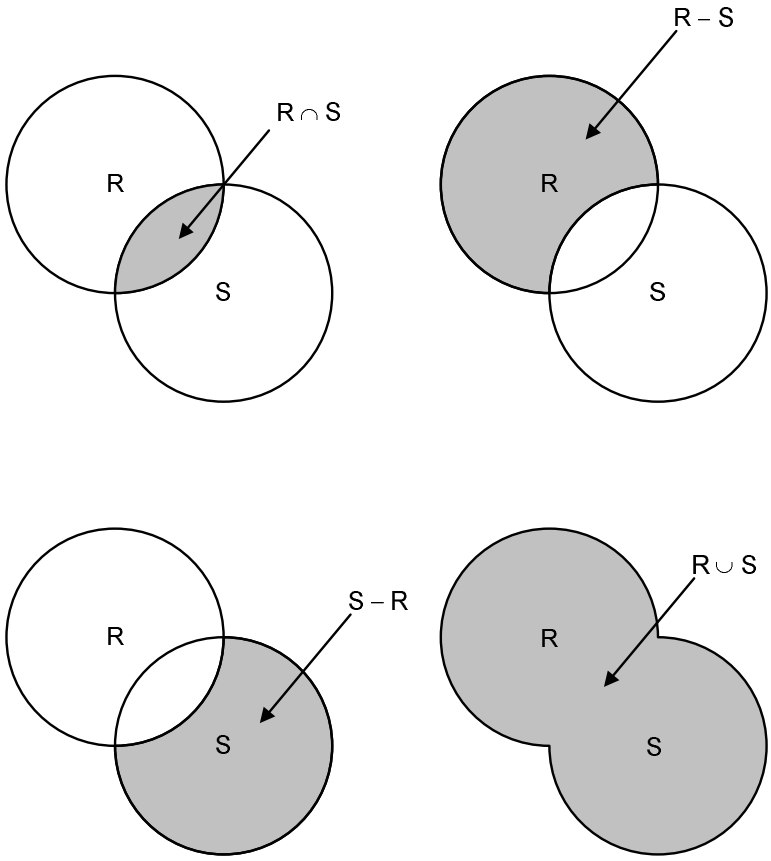

Figure 4.1: The Venn diagrams of set-theoretical operations

4.1.2 Set-Theoretical Operations

Since a table is essentially a *set of tuples*, it is natural to consider operations similar to the usual set-theoretical operations.

The basic set-theoretical operations: union, intersection, and difference are represented in Figure 4.1 using the well-known method of Venn diagrams.

In Figure 4.1 we show the intersection $R \cap S$, the difference $R - S$, the difference $S - R$, and the union $R \cup S$ of the sets R and S.

Unlike the set-theoretical case where the union, intersection, or difference of any two sets exists, in relational algebra only certain tables may be involved in these operations. The following definition introduces the necessary restriction, that of being compatible.

Definition 4.1.3 Let T_1, T_2 be two tables. The tables T_1, T_2 are *compatible* if they have the same headings.

If R_1, R_2 are extensions of two compatible tables T_1, T_2, respectively we say that they are *compatible relations*. Otherwise, we say that the relations are *incompatible*. □

Example 4.1.4 The tables STUDENTS and INSTRUCTORS are incompatible because

$$\textit{heading}(\text{STUDENTS}) = \text{stno name addr city state zip}$$
$$\textit{heading}(\text{INSTRUCTORS}) = \text{empno name rank roomno telno.}$$

It is not enough for the tables to have attributes in common; *equality* of the sets of attributes is required for compatibility.

Now consider a table that contains courses offered by the college under a continuing education program. Some of these courses are the same as the regular courses; others are offered only by this program.

CED_COURSES

cno	cname	cr	cap
cs105	Computer Literacy	2	150
cs110	Introduction to Computing	4	120
cs199	Survey of Programming	3	120

The tables COURSES and CED_COURSES are clearly compatible. □

Definition 4.1.5 Let T_1, T_2 be two compatible tables.

The *union of* T_1 *and* T_2 is the table $(T_1 \cup T_2)$ that contains the tuples that belong to T_1 *or* T_2.

The *intersection of* T_1 *and* T_2 is the table $(T_1 \cap T_2)$ that contains the tuples that belong to both T_1 *and* T_2

The *difference of* T_1 *and* T_2 is the table $(T_1 - T_2)$ that contains those rows that belong to T_1 but *do not* belong to T_2. □

Note that the names of the tables that we define here have the form $(T_1 \,\texttt{oper}\, T_2)$. By this we mean that the name of the new table is a string that is the concatenation of the left parenthesis "(", the name T_1, the symbol "`oper`", the name T_2, and the right parenthesis ")", where the symbol "`oper`" can be "$\cup$", "$\cap$", or "$-$". Observe that these symbols must be added to the alphabet we use to name tables. When there is no ambiguity, we simplify our notation by omitting parentheses; e.g., we write $T_1 \,\texttt{oper}\, T_2$ for $(T_1 \,\texttt{oper}\, T_2)$.

By an abuse of notation, renaming can also be used as an assignment to store results obtained using relational algebra operations. Thus, we may write, for instance, $T'(B_1, B_2, B_3) := T \,\texttt{oper}\, S$, where the tables T, S have the attributes A_1A_2 and A_2A_3, respectively. This means that after performing the `oper` operation, we rename the attributes A_1, A_2, A_3 to B_1, B_2, B_3, and we rename the resulting table T'.

As we introduce the operations of relational algebra, we use relational algebra expressions informally to construct the names of the tables that we are about to define.

Example 4.1.6 Consider the tables COURSES and CED_COURSES introduced in Example 4.1.4. If we need to determine all the courses offered by either the

regular program or the continuing education division, then we compute the table (COURSES ∪ CED_COURSES):

(COURSES ∪ CED_COURSES)

cno	cname	cr	cap
cs105	Computer Literacy	2	150
cs110	Introduction to Computing	4	120
cs199	Survey of Programming	3	120
cs210	Computer Programming	4	100
cs240	Computer Architecture	3	100
cs310	Data Structures	3	60
cs350	Higher Level Languages	3	50
cs410	Software Engineering	3	40
cs460	Graphics	3	30

Courses offered under both the regular and the extension program are computed in the table (COURSES ∩ CED_COURSES):

(COURSES ∩ CED_COURSES)

cno	cname	cr	cap
cs110	Introduction to Computing	4	120

Finally, (COURSES − CED_COURSES) contains courses offered by the regular program but not by the continuing education division.

(COURSES − CED_COURSES)

cno	cname	cr	cap
cs210	Computer Programming	4	100
cs240	Computer Architecture	3	100
cs310	Data Structures	3	60
cs350	Higher Level Languages	3	50
cs410	Software Engineering	3	40
cs460	Graphics	3	30

□

Definition 4.1.7 Let T and S be two distinct tables. The *product of T and S* is the table named $(T \times S)$ whose heading is $T.A_1 \dots T.A_n S.B_1 \dots S.B_k$ and which contains all tuples of the form

$$(u_1, \dots, u_n, v_1, \dots, v_k),$$

for every tuple $(u_1, \dots, u_n)$ of T and every $(v_1, \dots, v_k)$ of S. □

Example 4.1.8 The product of the tables

T

A	B	C
a_1	b_1	c_1
a_2	b_2	c_4
a_3	b_1	c_1

S

D	E
d_1	e_1
d_2	e_1

is the table

$(T \times S)$

$T.A$	$T.B$	$T.C$	$S.D$	$S.E$
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_4	d_1	e_1
a_3	b_1	c_1	d_1	e_1
a_1	b_1	c_1	d_2	e_1
a_2	b_2	c_4	d_2	e_1
a_3	b_1	c_1	d_2	e_1

□

In short, the product contains all possible combinations of the rows of the original tables. So, we see that the product operation can create huge tables starting from tables of modest size; for instance, the product of three tables of 1000 rows apiece yields a table with one billion tuples.

Note that the definition of the product of tables prevents us from considering the product of a table with itself. Indeed, if we were to try to construct the product $T \times T$, where the attributes of the new table would be $T.A_1, \ldots, T.A_n, T.A_1, \ldots, T.A_n$. This contradicts the requirement that all attributes of a table be distinct. To get around this restriction we create an alias T' by writing $T' := T$; then, we can compute $(T \times T')$, which has $T.A_1, \ldots, T.A_n$, $T'.A_1, \ldots, T'.A_n$ as its attributes.

Example 4.1.18 shows a query that requires this kind of special handling.

4.1.3 Selection

Selection is a unary operation (that is, an operation that applies to one table) that allows us to select tuples that satisfy specified conditions. For instance, using selection, we can extract the tuples that refer to all students who live in Massachusetts from the STUDENTS table. To begin, we formalize the notion of a condition.

Definition 4.1.9 Let H be a set of attributes. An *atomic condition on* H has the form $A\,\texttt{oper}\,a$ or $A\,\texttt{oper}\,B$, where A, B are attributes of H that have the same domain, `oper` is one of $=, !=, <, >, \leq$, or $\geq$, and a is a value from the domain of A. □

As is common in query languages, we use != to represent $\neq$, because $\neq$ is not part of the ASCII character set and does not appear on most keyboards.

Example 4.1.10 Consider the table ITEMS that is a part of the database of a department store and lists items sold by the store. We assume that the heading consists of the following attributes:

$H =$ itno iname dept cost retprice date.

The significance of the attributes of ITEMS is summarized below:

Attribute	Meaning
itno	item number
iname	item name
dept	store department
cost	wholesale price
retprice	retail price
date	date when the retail price was set

The following constructions

$$\begin{array}{l}\mathsf{dept} = \text{'Sport'}\\ \mathsf{cost} > \mathsf{retprice}\\ \mathsf{cost} <= 1.25\end{array}$$

are atomic conditions on the attributes of ITEMS. Note that we use quotation marks for the value 'Sport', because it is a part of a string domain, but there are no quotation marks around 1.25, because this value belongs to a numerical domain. □

Starting from these atomic condtions, we can build more complicated conditions using **and**, **or**, and **not**. So, if we want to list the sports items that sell for under $ 1.25, we can use the condition $\mathsf{dept} = \text{'Sport'}$ **and** $\mathsf{cost} <= 1.25$. This method of building conditions is known as "recursive", and we use it in the following definition.

Definition 4.1.11 *Conditions on a set of attributes* H are defined recursively as follows:

1. Every atomic condition on H is a condition on H.
2. If $\mathcal{C}_1, \mathcal{C}_2$ are conditions on H, then

$$(\mathcal{C}_1 \textbf{ or } \mathcal{C}_2), (\mathcal{C}_1 \textbf{ and } \mathcal{C}_2), (\textbf{not } \mathcal{C}_1)$$

are conditions on H.

□

It is common practice to omit parentheses when the expression is unambiguous. This depends on a hierarchy of operations, where **not** has the higher priority, and by **and** and **or** are at the same, lower priority. Successive operations at the same priority are associated from left-to-right. So, $\mathcal{C}_1$ **and** $\mathcal{C}_2$ **or** $\mathcal{C}_3$ **and** $\mathcal{C}_4$ is to be interpreted as $(((\mathcal{C}_1 \textbf{ and } \mathcal{C}_2) \textbf{ or } \mathcal{C}_3) \textbf{ and } \mathcal{C}_4)$.

Next, we define what it means for a tuple of a table T to satisfy a condition.

Definition 4.1.12 A tuple t satisfies an atomic condition on H, $A \,\mathtt{oper}\, a$ if $t[A] \,\mathtt{oper}\, a$; t satisfies the atomic condition $A \,\mathtt{oper}\, B$ if $t[A] \,\mathtt{oper}\, t[B]$.

A tuple t satisfies the condition $(\mathcal{C}_1 \textbf{ and } \mathcal{C}_2)$ if it satisfies both $\mathcal{C}_1$ and $\mathcal{C}_2$; t satisfies the condition $(\mathcal{C}_1 \textbf{ or } \mathcal{C}_2)$ if it satisfies at least one of $\mathcal{C}_1$ and $\mathcal{C}_2$. Finally, t satisfies $(\textbf{not } \mathcal{C}_1)$ if it fails to satisfy $\mathcal{C}_1$. □

To introduce the selection operation we add to the alphabet $\mathcal{A}$ the symbols **or**, **and** and **not**; also, we add the relational attributes and the members of their domains. Observe that a relational attribute that is written using several letters (such as stno) is considered in this context to be a single symbol rather than a sequence of several letters.

Definition 4.1.13 Let T be a table, and let $\mathcal{C}$ be a condition on H. The *table obtained by* $\mathcal{C}$*-selection* is the table $(T \textbf{ where } \mathcal{C})$ having the same heading H as T, where the content of $(T \textbf{ where } \mathcal{C})$ consists of all tuples of T that satisfy the condition $\mathcal{C}$. □

The next example shows how selection can be used to extract data from the college database. Sometimes we show the table resulting from the operation (or the succession of operations) that we intend to illustrate. In all such cases, we assume that the college database is in the state shown in Figure 3.1.

Example 4.1.14 To retrieve all students who live in Boston or in Brookline, we write:

T1 := (STUDENTS **where**(city = 'Boston' **or** city = 'Brookline'))

The corresponding table is:

T1

stno	name	addr	city	state	zip
2661	Mixon Leatha	100 School St.	Brookline	MA	02146
2890	McLane Sandy	30 Cass Rd.	Boston	MA	02122
3566	Pierce Richard	70 Park St.	Brookline	MA	02146
4022	Prior Lorraine	8 Beacon St.	Boston	MA	02125
5544	Rawlings Jerry	15 Pleasant Dr.	Boston	MA	02115

□

Example 4.1.15 Let us find the list of grades given in CS110 during the spring semester of 2002. This can be done by applying the following selection operation:

T := (GRADES **where**cno = 'CS110' **and**
sem = 'Spring' **and** year = 2002).

This selection gives the table:

T

stno	empno	cno	sem	year	grade
1011	023	cs110	Spring	2002	75
4022	023	cs110	Spring	2002	60

□

We conclude the definition of selection with the observation that selection extracts "horizontal" slices from a table. The next operation extracts vertical slices from tables.

4.1.4 Projection

Recall that we introduced the projection of tables in Section 3.2. In this section we re-examine this notion as a relational algebra operation.

A table may contain many attributes, but for any particular query, only some of these may be relevant; projection allows us to chose these.

Example 4.1.16 Suppose that we wish to produce a list of instructors' names and the room numbers of their offices. This can be accomplished by projection:

OFFICE_LIST := INSTRUCTORS[name roomno]

and we obtain the table:

OFFICE_LIST

name	roomno
Evans Robert	82
Exxon George	90
Sawyer Kathy	91
Davis William	72
Will Samuel	90

□

Example 4.1.17 Projection and selection may be combined, provided the projection does not eliminate the attributes used in the selection. Consider, for ex-

ample, the task of determining the grades of the student whose student number is 1011. The table T created by

T:=(GRADES **where**sno='1011')[grade]

is

T

grade
40
75
90

Observe that duplicates are dropped through projection. Indeed, instead of two grades of 90, the table shows only one. This happens because, as we pointed out above, tables do not contain duplicate entries. □

Example 4.1.18 Suppose that we need to find all pairs of instructors' names for instructors who share the same office. Of course, we need to compare the office of every instructor with the office of every other instructor; we output the names of instructors who have the same office. This query requires that we form the product of the table INSTRUCTORS with an alias I of this table, as follows:

I := INSTRUCTORS

PROD := (INSTRUCTORS × I)

Next, we extract the pairs of instructors who have equal values for roomno. This is accomplished using the selection:

(PROD **where**INSTRUCTORS.roomno = I.roomno).

Note that this is not an entirely satisfactory solution. Indeed, we have no interest in knowing that an instructor is in the same room as himself or herself; and, once we know that instructor i_1 is in the same room as instructor i_2 it is clear that i_2 is in the same room as i_1. To eliminate this type of redundancy from the answer we use a more restrictive selection:

(PROD **where** INSTRUCTORS.roomno = I.roomno
and INSTRUCTORS.empno < I.empno)

Finally, we extract the names of the instructors involved in the pairs retrieved above:

(PROD **where** INSTRUCTORS.roomno = I.roomno
and INSTRUCTORS.empno < I.empno)
[INSTRUCTORS.name, I.name]

□

4.1.5 The Join Operation

The join operation is important for answering queries that combine data that reside in several tables. To define the join operation between two tables, we first introduce the join between two tuples.

Definition 4.1.19 Let T_1, T_2 be two tables that have the headings

$$A_1 \cdots A_m B_1 \cdots B_n \text{ and } B_1 \cdots B_n C_1 \cdots C_p,$$

respectively. (In other words, assume that the two tables that have only the attributes $B_1, \ldots, B_n$ in common.)

The tuples t_1 in T_1 and t_2 in T_2 are *joinable* if

$$t_1[B_1 \cdots B_n] = t_2[B_1 \cdots B_n].$$

If t_1 and t_2 are joinable tuples, their *join* is a tuple t defined on

$$A_1 \ldots A_m B_1 \ldots B_n C_1 \ldots C_p$$

such that

$$t[A_1 \ldots A_m B_1 \ldots B_n] = t_1[A_1 \ldots A_m B_1 \ldots B_n],$$

and

$$t[B_1 \ldots B_n C_1 \ldots C_p] = t_2[B_1 \ldots B_n C_1 \ldots C_p].$$

The join of t_1 and t_2 is denoted by $t_1 \bowtie t_2$. ∎

Note that in the above definition, if D is one of the attributes $B_1, \ldots, B_n$ shared by the two tables, then $t_1[D] = t_2[D]$ because t_1, t_2 are joinable, so $t[D]$ can be defined correctly to be either $t_1[D]$ or $t_2[D]$.

Example 4.1.20 Let T_1, T_2 be the tables given by:

T_1

	A	B	D
t_1	a_2	b_1	d_1
t_2	a_1	b_2	d_4
t_3	a_3	b_1	d_1
t_4	a_3	b_1	d_2
t_5	a_1	b_3	d_3

T_2

	B	C	D
u_1	b_1	c_1	d_1
u_2	b_2	c_2	d_4
u_3	b_3	c_2	d_1
u_4	b_2	c_1	d_2

The tuples t_1 and u_1 are joinable because $t_1[BD] = u_1[BD] = (b_1\ d_1)$; similarly, t_2 is joinable with u_2, t_3 is joinable with u_1, and t_4 and t_5 are not joinable with any tuple of S.

We have

$$\begin{aligned} t_1 \bowtie u_1 &= (a_2, b_1, d_1, c_1) \\ t_2 \bowtie u_2 &= (a_1, b_2, d_4, c_2) \\ t_3 \bowtie u_1 &= (a_3, b_1, d_1, c_1) \end{aligned}$$

∎

Definition 4.1.21 Suppose that $B_1, \ldots, B_n$ are the attributes that two tables T_1, T_2 have in common.

The *natural join of* T_1 *and* T_2, or simply the *join*, is the table named $(T_1 \bowtie T_2)$ having the heading $A_1 \ldots A_m B_1 \ldots B_n C_1 \ldots C_p$ that contains of all tuples $t_1 \bowtie t_2$ such that t_1 is in T_1 and t_2 is in T_2, and t_1 is joinable with t_2. ∎

Note that if $n = 0$ (that is, if the tables T_1, T_2 have no attributes in common), then the joinability condition is satisfied by every tuple t_1 of T_1 and t_2 of T_2. In this special case, the tables $T_1 \bowtie T_2$ and $T_1 \times T_2$ are virtually identical: they have the same rows but different names and headings.

Example 4.1.22 The join $T_1 \bowtie T_2$ of the tables considered in Example 4.1.20 is the table

$T_1 \bowtie T_2$

A	B	D	C
a_2	b_1	d_1	c_1
a_1	b_2	d_4	c_2
a_3	b_1	d_1	c_1

□

Example 4.1.23 Suppose that we need to find the names of all instructors who have taught cs110. Initially, we extract all grade records involving cs110 using a selection operation:

$$T_1 := (\mathsf{GRADES}\ \mathbf{where} \mathsf{cno} = \text{'cs110'}).$$

Then, by joining T_1 with the table INSTRUCTORS we extract the records of instructors who teach this course:

$$T_2 := (T_1 \bowtie \mathsf{INSTRUCTORS}).$$

Finally, a projection on name yields the answer to the query:

$$\mathsf{ANS} := T_2[\mathsf{name}].$$

□

Example 4.1.24 To find the names of all instructors who have ever taught any four-credit course, we can compute the join:

$$T_1 := ((\mathsf{COURSES} \bowtie \mathsf{GRADES}) \bowtie \mathsf{INSTRUCTORS}).$$

Then, by applying a selection we extract records corresponding to four-credit courses:

$$T_2 := (T_1\ \mathbf{where}\ \mathsf{cr} = 4).$$

The names of instructors are thus obtained by projection:

$$\mathsf{ANS} := T_2[\mathsf{name}].$$

□

An interesting variant of the previous example is given below:

Example 4.1.25 Let us determine the names of all instructors who have taught any student who lives in Brookline. Observe that join cannot be used because computing the join

$$((\mathsf{STUDENTS} \bowtie \mathsf{GRADES}) \bowtie \mathsf{INSTRUCTORS})$$

would require the name of the student to be identical with the name of the instructor (which is, of course, not what is required by this query). Instead, we

can use the product of tables and enforce the "limited joining" through selection:

$$\begin{aligned} T_1 &:= (\mathsf{STUDENTS} \times \mathsf{GRADES} \times \mathsf{INSTRUCTORS}) \\ T_2 &:= T_1 \,\textbf{where}\, \mathsf{STUDENTS.stno} = \mathsf{GRADES.stno}\,\textbf{and} \\ &\quad\ \mathsf{GRADES.empno} = \mathsf{INSTRUCTORS.empno}\,\textbf{and} \\ &\quad\ \mathsf{STUDENTS.city} = \text{'Brookline'} \end{aligned}$$

Then, by projection, we extract the name of the instructors involved:

$$\mathsf{ANS} := T_2[\mathsf{INSTRUCTORS.name}].$$

□

Join can be used to express other operations. Note, for instance, that if T and T' are two compatible tables, then $T \bowtie T'$ has the same rows as $T \cap T'$. Indeed, since the two tables have all their attributes in common, two tuples t in T and t' in T' are joinable only if they are equal on all attributes, that is, if they are the same.

4.1.6 Division

Definition 4.1.26 Let T_1, T_2 be two tables such that the heading of T_1 is $A_1 \dots A_n B_1 \dots B_k$ and the heading of T_2 is $B_1 \dots B_k$. The table obtained by *division* of T_1 by T_2 is the table $T_1 \div T_2$ that has the heading $A_1 \dots A_n$ and contains those tuples t in $\mathbf{tupl}(A_1 \dots A_n)$ such that $t \bowtie t_2$ is a tuple in T_1 for *every* tuple t_2 of T_2. □

In other words, the content of the table obtained by dividing T_1 by T_2, $T_1 \div T_2$, consists of each tuple from $\mathbf{tupl}(A_1 \dots A_n)$ which, when concatenated with *every* tuple of T_2, yields a tuple of T_1.

We stress that, in order for two tables T_1 and T_2 to be involved in a division, the heading of T_2 must be included in the heading of T_1.

Example 4.1.27 Suppose that we need to determine the courses taught by all full professors. We can solve this query by first determining the employee numbers (empno) for all full professors:

$$T_1 := (\mathsf{INSTRUCTORS}\,\textbf{where}\,\mathsf{rank} = \text{'Professor'})[\mathsf{empno}].$$

This generates the table:

T_1

empno
019
023

Then, using projection, we discard all attributes from GRADES with the exception of cno and empno:

$$T_2 := \mathsf{GRADES}[\mathsf{cno}, \mathsf{empno}],$$

which results in

T_2

empno	cno
019	cs110
023	cs110
019	cs240
234	cs410
019	cs210
056	cs240
234	cs310

Finally, by applying division, we extract the course numbers of courses that are taught by all full professors:

$$\mathsf{ANS} := (T_2 \div T_1),$$

that is,

ANS

cno
cs110

It is essential to project GRADES on cno empno; otherwise, if we divide GRADES by T_1, a tuple $t = (s, c, m, y, g)$ is placed into $GRADES \div T_1$ only if the student s has taken the course c during the semester m of the year y and has obtained the grade g from *all* full professors. Extracting the course number afterwards does not help at all, since this requirement is both impossible to satisfy and has nothing to do with our query. □

4.2 The Basic Operations of Relational Algebra

So far, we have introduced nine operations: renaming, union, intersection, difference, product, selection, projection, join, and division. Now, we show that certain operation can be expressed in terms of other operations. Our goal is to build a list of "*basic operations*" that have the same computational capabilities as the full set of operations previously introduced. In other words, for any table created using the full set of operations, we can build a a table that has the same content using the set of basic operations. Consequently, a relational database systems need to implement only the basic operations and indeed, this is what most of them do.

By convention, unary operations of relational algebra — that is, selection and projection — have higher priority than the remaining binary operations. Thus, for example, T_1 `oper` T_2 **where** $\mathcal{C}$ means T_1 `oper`(T_2 **where** $\mathcal{C}$).

Let T_1 and T_2 be two compatible tables. It is easy to see that $T_1 \cap T_2$ has the same content as $T_1 - (T_1 - T_2)$. Thus intersection can be accomplished using difference.

To see how the join operation can be expressed using the operations of renaming, product, selection, and projection consider the following example.

Example 4.2.1 The tables T_1, T_2 introduced in Example 4.1.20, have the headings ABD and BCD, respectively. The table $T_3 := T_1 \times T_2$ is

T_3

$T_1.A$	$T_1.B$	$T_1.D$	$T_2.B$	$T_2.C$	$T_2.D$
a_2	b_1	d_1	b_1	c_1	d_1
a_1	b_2	d_4	b_2	c_2	d_4
a_3	b_1	d_1	b_1	c_1	d_1

Then, we eliminate duplicate columns and rename the attributes in

$$T_4(A, B, D, C) := T_3[T_1.A, T_1.B, T_2.C, T_2.D].$$

T_4

A	B	D	C
a_2	b_1	d_1	c_1
a_1	b_2	d_4	c_2
a_3	b_1	d_1	c_1

The table T_4 contains exactly the same tuples as the join $T_1 \bowtie T_2$. □

Example 4.2.2 The query considered in Example 4.1.24 (where we use join to find the names of instructors who have taught any four-credit course) can now be solved using product, selection, projection, and renaming by the following computation:

$$\begin{aligned} T_1 &= (\mathsf{COURSES} \times \mathsf{GRADES} \times \mathsf{INSTRUCTORS}) \\ T_2 &= (T_1 \textbf{ where } \mathsf{COURSES.cr} = 4 \textbf{ and} \\ &\quad \mathsf{COURSES.cno} = \mathsf{GRADES.cno} \textbf{ and} \\ &\quad \mathsf{GRADES.empno} = \mathsf{INSTRUCTORS.empno}) \\ T_3(\mathsf{name}) &= T_2[\mathsf{INSTRUCTORS.name}] \end{aligned}$$

□

The division operation can be expressed using renaming, product, selection, projection, and difference. We illustrate how this can be accomplished by offering an alternative solution to the query in Example 4.1.27, listing the courses taught by every full professor.

Example 4.2.3 Instead of directly finding the courses taught by all full professors, we initially determine the courses that do not satisfy this condition. In other words, in the first phase of the solution, we determine those courses that are not taught by every full professor. Then, in the second phase, we eliminate from the table GRADES[cno] (the list of courses that are actually taught) the courses retrieved in the first phase.

We begin by forming a table T_3 containing all pairs of course numbers and employee numbers for full professors. This is accomplished by:

$$\begin{aligned} T_1 &:= (\mathsf{INSTRUCTORS} \textbf{ where} \\ &\quad rank = \text{'Professor'})[\mathsf{empno}] \\ T_2 &:= \mathsf{GRADES}[\mathsf{cno}] \\ T_3(cno, empno) &:= (T_2 \times T_1) \end{aligned}$$

The renaming of last step is required to replace the qualified attributes with unqualified ones; i.e., we must consider all possible combinations (pairs) of professors and courses, not just the courses that the various professors taught.

Next, by computing

$$T_4 := (T_3 - \mathsf{GRADES}[\mathsf{cno}, \mathsf{empno}]),$$

we retain a pair (c, e) in T_4 (where c is a course number and e is an employee number) only if there is a full professor (whose employee number is e) who did not teach the course that has course number c. Therefore, a course number occurs in $T_5 := T_4[\mathsf{cno}]$ only if there is a full professor who did not teach that course. Consequently, courses taught by all full professors are the ones that do not appear in T_5; in other words, these courses can be found in the table $T_6 = (\mathsf{COURSES}[\mathsf{cno}] - T_5)$.

Starting from the database instance from Figure 3.1 we obtain the table T_1 that contains all employee numbers for full professors:

T_1

empno
019
023

$T_2 = \mathsf{GRADES}[\mathsf{cno}]$ contains all course numbers that are currently taught:

GRADES[cno]

cno
cs110
cs210
cs240
cs310
cs410

T_3 gives all possible pairs of course numbers and employee numbers for full professors:

T_3

cno	empno
cs110	019
cs110	023
cs210	019
cs210	023
cs240	019
cs240	023
cs310	019
cs310	023
cs410	019
cs410	023

The relation $T_4 := (T_3 - \mathsf{GRADES}[\mathsf{cno}, \mathsf{empno}])$ is

T_4

cno	empno
cs210	023
cs240	023
cs310	019
cs310	023
cs410	019
cs410	023

This means that the courses not taught by every full professor are:

T_5

cno
cs210
cs240
cs310
cs410

Finally, the result of the computation is:

T_6

cno
cs110

□

The same series of steps can always be used to calculate the division operation.

The arguments just presented show that only six of the nine operations of relational algebra are required: renaming, union, difference, product, selection, and projection.

It is natural to ask whether we can eliminate any of these remaining operations and still retain the full computational power of relational algebra. We show, however, the set of six operations just mentioned is *minimal*. In other words, if we discard any of these six operations, the remaining five are unable to do the job of the discarded operation.

The following observation shows that the union operation cannot be discarded.

Consider the one-attribute, one-tuple tables:

T_1

A
a_1

and

T_2

A
a_2

If we assume $a_1 \neq a_2$, then the table $T_1 \cup T_2$ is given by

$(T_1 \cup T_2)$

A
a_1
a_2

Note that if we apply the operations of difference, product, selection, projection, and renaming to tables that consist of at most one tuple, then the result may contain at most one tuple; therefore, any computation that makes use only of these operations is not capable of computing a target that contains more than

one tuple, and therefore is unable to produce a table that has the same content as $T_1 \cup T_2$. ∎

The product operation cannot be eliminated from the set of basic operations. Indeed, suppose that a database consists of two tables T and S that have the one-attribute headings A and B, respectively. Observe that no computation that uses renaming, union, difference, selection, and projection is capable of computing the table $(T_1 \times T_2)$. Indeed, any table that is obtained through such a computation may have only one attribute and, therefore, it cannot compute $T_1 \times T_2$, which has two attributes.

Slightly more complicated examples show that the difference, selection, and projection operations are all essential.

4.3 Other Relational Algebra Operations

We consider now three operations related to the natural join. In a join operation tuples may be combined only if they have equal values on all columns they share and they must have such values in *all* such columns. This can be awkward in many situations; the operation we are about to introduce allows for more flexibility.

Definition 4.3.1 Let T and T' be two tables such that their headings H, H', respectively, have no common attributes.

Suppose that $A_1, \ldots, A_n$ are attributes of H and $B_1, \ldots, B_n$ are attributes of H' such that $\mathrm{Dom}\, A_i = \mathrm{Dom}\, B_i$ for $1 \leq i \leq n$, and let θ_i be one of $\{=, !=, <, \leq, >, \geq\}$ for $1 \leq i \leq n$. Here, we use $!=$ to denote inequality.
If $\theta = (\theta_1, \ldots, \theta_n)$, the *$\theta$-join of τ and τ'* is the table having the name $T \bowtie_{A_1\theta_1B_1,\ldots,A_n\theta_nB_n} T'$, the heading HH' and the content $\rho_{A_1\theta_1B_1,\ldots,A_n\theta_nB_n}$, where $\rho_{A_1\theta_1B_1,\ldots,A_n\theta_nB_n}$ consists of those tuples $u \in \mathbf{tupl}(HH')$ for which there is $t \in \rho$ and $t \in \rho'$ such that

$$u[A] = \begin{cases} t[A] & \text{if } A \in H \\ t'[A] & \text{if } A \in H', \end{cases}$$

and $t[A_i]\theta_i t'[B_i]$ for $1 \leq i \leq n$.

If θ_i is equality for all i, $1 \leq i \leq n$, then we refer to the table

$$T \bowtie_{A_1\theta_1B_1,\ldots,A_n\theta_nB_n} T'$$

as the *equijoin* of τ and τ'. ∎

Example 4.3.2 Suppose we need to determine the pairs of student names and instructor names such that the instructor is not an advisor for the student. In order to deal with the requirement that the tables involved in a θ-join have disjoint headings we create the tables:

ADVISING1(stno, empno1) := ADVISING,

and

INSTRUCTORS1(empno,name1) := INSTRUCTORS[empno,name].

Since every student has one advisor it suffices to compute the θ-join:

$$\mathsf{T} := (\mathsf{ADVISING1} \bowtie_{\mathsf{empno1!=empno}} \mathsf{INSTRUCTORS1})$$

Then, using natural join and projection we extract the answer:

$$\mathsf{ANS} := (\mathsf{STUDENTS} \bowtie \mathsf{T})[\mathsf{name,name1}].$$

□

The semijoin $\ltimes$ is another operation related to the join operation.

Definition 4.3.3 Let T_1, T_2 be two tables having the headings H_1, H_2 and the contents ρ_1, ρ_2, respectively. Their *semijoin* is the table named $T_1 \ltimes T_2$ that has the heading H_1 and the content $\rho_1 \ltimes \rho_2$, where $\rho_1 \ltimes \rho_2 = (\rho_1 \bowtie \rho_2)[H_1]$. □

Example 4.3.4 Let

$\tau_1 = (T_1, ABD, \rho_1)$,

$\tau_2 = (T_2, BCD, \rho_2)$

be the tables considered in Example 4.1.20. The semijoin $\tau_1 \ltimes \tau_2$ is the table

$(T_1 \ltimes T_2)$

A	B	D
a_2	b_1	d_1
a_1	b_2	d_4
a_3	b_1	d_1

The semijoin $\tau_2 \ltimes \tau_1$ is:

$(T_2 \ltimes T_1)$

B	D	C
b_1	d_1	c_1
b_2	d_4	c_2

□

Clearly, we have in general $\rho_1 \ltimes \rho_2 \neq \rho_2 \ltimes \rho_1$.

The join operation is linked to semijoin by the identities:

$$\rho_1 \bowtie \rho_2 = \rho_1 \bowtie (\rho_2 \ltimes \rho_1) = \rho_2 \bowtie (\rho_1 \ltimes \rho_2).$$

The semijoin of table τ_1 with table τ_2 computes that part of table τ_1 that consists of the tuples of τ_1 that are joinable with tuples of τ_2; in other words, it computes the "useful" part of τ_1 for the join with τ_2. This operation is very important for distributed databases. In such databases various tables (or even portions of tables) may reside at different computing sites, and it is often important to minimize the amount of data traffic through the network that connects these sites. Suppose, for example that τ_1 is a very large table stored at site S_1, τ_2 is a relatively small table stored at site S_2 and $\tau_1 \bowtie \tau_2$ is needed at site S_2 (see Figure 4.2).

Suppose that the tuples of T_1 and T_2 have approximatively the same size. Also, assume that T_1 contains n_1 tuples, T_2 contains n_2 tuples and k tuples of T_1 are joinable with the tuples of T_2. We need to compare two scenarios:

1) Ship table T_1 to site S_2. The traffic cost is proportional to the size n_1 of T_1.

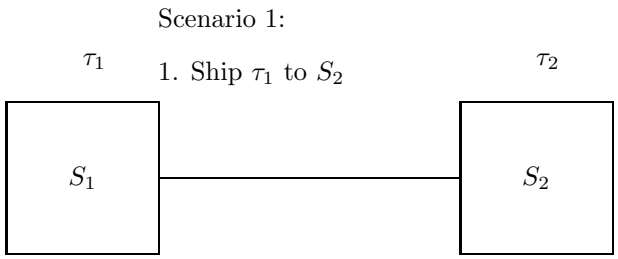

Figure 4.2: Computing a join in a two-site network

2) Ship T_2 to site S_1, compute the semijoin $T_1 \ltimes T_2$ at site S_1, ship the semijoin to site S_2 and compute the join $T_1 \bowtie T_2$ using the semijoin. The cost of the traffic is n_2+k. If this number is much smaller that n_1 the second method could be preferable.

Note that if ρ_1, ρ_2 are two relations, then $\rho_1 - (\rho_1 \ltimes \rho_2)$ is that part of ρ_1 that consists of tuples of ρ_1 that are not joinable with any tuples of ρ_2. This observation is usful in defining the third operation that we introduce in this section.

The tuples of a table T that is involved in a join with another table T‘ and are not joinable with any tuple of T' leave no trace in the join $T \bowtie T'$. By contrast, in the operation we are about to define, all tuples, joinable or not, participate in the final result.

Definition 4.3.5 Let T_1, T_2 be two tables having th headings H_1, H_2 and the contents ρ_1, ρ_2, respectively.

The *left outer join* of τ_1 and T_2 is the table named $T_1 \bowtie_l T_2$ having th heading $H_1 \cup H_2$ and the content $\rho_1 \bowtie_\ell \rho_2$, where:

$$\begin{aligned} \rho_1 \bowtie_\ell \rho_2 \quad = \quad & (\rho_1 \bowtie \rho_2) \cup \{(a_1, \ldots, a_n, \mathbf{null}, \ldots, \mathbf{null}) \mid \\ & (a_1, \ldots, a_n) \in \rho_1 - (\rho_1 \ltimes \rho_2)\}. \end{aligned}$$

The *right outer join* of T_1 and T_2 is the table named $T_1 \bowtie_r T_2$ whose heading is $H_1 \cup H_2$, having the content $\rho_1 \bowtie_r \rho_2$, where

$$\begin{aligned} \rho_1 \bowtie_r \rho_2 \quad = \quad & (\rho_1 \bowtie \rho_2) \cup \{(\mathbf{null}, \ldots, \mathbf{null}, b_1, \ldots, b_p) \mid \\ & (b_1, \ldots, b_p) \in \rho_2 - (\rho_2 \ltimes \rho_1)\} \end{aligned}$$

The *outer join of the tables* T_1 *and* T_2 is the table named $T_1 \bowtie_o T_2$, whose

heading is $H_1 \cup H_2$. The content of this table, $\rho_1 \bowtie_o \rho_2$ is $\rho_1 \bowtie_o \rho_2 = (\rho_1 \bowtie_\ell \rho_2) \cup (\rho_1 \bowtie_r \rho_2)$. □

Example 4.3.6 Let (T_1, ABD, ρ_1) and (T_2, BCD, ρ_2) be the tables considered in Example 4.3.4. The left outer join $T_1 \bowtie_\ell T_2$ is the table

$(T_1 \bowtie_\ell T_2)$

A	B	D	C
a_2	b_1	d_1	c_1
a_1	b_2	d_4	c_2
a_3	b_1	d_1	c_1
a_3	b_1	d_2	**null**
a_1	b_3	d_3	**null**

The right outer join $T_1 \bowtie_r T_2$ is:

$(T_1 \bowtie_r T_2)$

A	B	D	C
a_2	b_1	d_1	c_1
a_1	b_2	d_4	c_2
a_3	b_1	d_1	c_1
null	b_3	d_1	c_2
null	b_2	d_2	c_1

The outer join of these tables is:

$(T_1 \bowtie_\ell T_2)$

A	B	D	C
a_2	b_1	d_1	c_1
a_1	b_2	d_4	c_2
a_3	b_1	d_1	c_1
a_3	b_1	d_2	**null**
a_1	b_3	d_3	**null**
null	b_3	d_1	c_2
null	b_2	d_2	c_1

□

4.4 Exercises

1. Consider a database that consists of one table T:

   T

A	B
a	b

   Prove that there is no computation that uses renaming, union, product, difference, and selection that can compute the projection $T[A]$. Conclude that projection is an essential operation.

Solve the queries contained in Exercises 2–49 in relational algebra.

2. Find the names of students who live in Boston; find the names of students who live outside Boston.
3. Find all pairs of student names and course names for grades obtained during Fall of 2001.
4. Find the names of students who took some four-credit courses.
5. Find the names of students who took every four-credit course.

6. Find the names of students who took only four-credit courses.
7. Find the names of students who took no four-credit courses.
8. Find the names of students who took a course with an instructor who is also their advisor.
9. Find the names of students who took cs210 or cs310.
10. Find the names of students who took cs210 and cs310.
11. Find the names of all students who took neither cs210 nor cs310.
12. Find names of all students who took cs310 but never took cs210.
13. Find names of all students who have a cs210 grade higher than the higest grade given in cs310 and did not take any course with Prof. Evans.
14. Find the names of all students whose advisor is not a full professor.
15. Find the names of students sho took cs210 or had Prof. Smith as their advisor.
16. Find all pairs of names of students who live in the same city.
17. Find all triples of instructors' names for instructors who taught the same course.
18. Find instructors who taught students who are advised by another instructor who shares the same room.
19. Find course numbers for courses that enrol at least two students; solve the same query for courses that enrol at least three students.
20. Find course numbers for courses that enrol exactly two students;
21. Find all pairs of students' names for students who studied with the same instructor.
22. Find the names of all students for whom no other student lives in the same city.
23. Find the names of students who obtained the highest grade in cs210.
24. Find course numbers of courses taken by students who live in Boston and which are taught by an associate professor.
25. Find the names of instructors who teach courses attended by students who took a course with an instructor who is an assistant professor.
26. Find the telephone numbers of instructors who teach a course taken by any student who lives in Boston.
27. Find the lowest grade of a student who took a course during the spring of 2003.
28. Find names of students who took every course taken by Richard Pierce.
29. Find the names for students such that if prof. Evans teaches a course, then the student takes that course (although not necessarily with prof. Evans).
30. Find all pairs of names of students and instructors such that the student never took a course with the instructor.
31. Find the names of students who took only one course.
32. Find the names of students who took at least two courses.
33. Find names of courses taken by students who do not live in Massachusetts (MA).
34. Find the names of instructors who teach no course.
35. Find course numbers of courses that have never been taught.

36. Find course numbers of courses taken by students whose advisor is an instructor who taught cs110.
37. Find the highest grade of a student who never took cs110.
38. Find courses that are taught by every assistant professor.
39. Find the names of the instructors who taught only one course during the spring semester of 2001.
40. Find the names of students whose advisor did not teach them any course.
41. Find the names of students who have failed all their courses (failing is defined as a grade less than 60).
42. Find the names of students who do not have an advisor.
43. Find course names of courses taught by more than one instructor.
44. Find the names of instructors who taught every semester when a student from Rhode Island was enrolled.
45. Find course names of courses taken be every student advised by Prof. Evans.
46. Find names of students who took every course taught be an instructor who is advising at least two students.
47. Find names of instructors who teach every student they advise.
48. Find names of students who are taking every course taught by their advisor.
49. Find course numbers of courses taken by every student who lives in Rhode Island.
50. Consider a database that consists of one table T:

T

A
a_1
a_2

    Prove that there is no computation that uses renaming, union, product, difference, and projection that can compute the selection T **where** $A = a_1$. Conclude that selection is an essential operation.
51. Prove that the product of tables can be expressed using renaming and join. Conclude that there exist minimal sets of operations other than the set of basic operations.

4.5 Bibliographical Comments

The original definition of relational algebra was given by E. F. Codd in [Codd, 1972a]. Relational algebra is presented in a rigorous manner in several sources [Fejer and Simovici, 1991; Simovici and Tenney, 1995; Maier, 1983] and [Ullman, 1988a]. An excellent informal introduction can be found in [Date, 2003].