lational calculus. We look at these in more depth in Sections 4.3 and 4.4, respectively.

Example 4.4 looks at the relational rules that define certain properties that the database must satisfy.

Example 4.4 If we intend to keep only information on currently available DBMS packages in our database, we could specify that in our VERSION relation the release year of a version not go beyond the current year. We could also specify that the DBMS name be unique. With the unique name and tuple properties, it is apparent that the name determines the company that produces the DBMS and its data model. We may conclude that *Name* uniquely determines *Company* and *Name* uniquely determines *Data_Model*. ■

This unique identification is an integrity constraint, which ensures that each instance of an entity is distinguishable. Functional dependency is also a form of constraint, as it specifies which combination of values is legal. Certain constraints are defined in terms of functional dependencies between the attributes and form the basis of the normalization theory (see Chapters 6 and 7). The entity and referential integrity rules are two general rules that all relational databases are expected to satisfy. Both rules will be studied in Section 4.2.8. Additional rules may also be defined for the application in hand.

Relational database theory borrows heavily from set algebra; therefore a brief review of set concepts is given in the following section. Some data manipulation languages make use of first-order predicate calculus and the relevant material is briefly covered in Section 4.4. The material presented here is not exhaustive but should be sufficient to understand the relational model.

4.1.1 A Brief Review of Set Theory

A **set** is well-defined collection of objects. It is commonly represented by a list of its elements (called **members**) or by the specification of some membership condition. The **intension** of a set defines the permissible occurrences by specifying a membership condition. The **extension** of the set specifies one of numerous possible occurrences by explicitly listing the set members. These two methods of defining a set are illustrated in the following example.

Example 4.5 Intension of set G: $\{g | g$ is an odd positive integer less than $20\}$

Extension of set G: $\{1,3,5,7,9,11,13,15,17,19\}$ ■

A set is determined by its members. The number 3 is a member of the set G and this is denoted by $3 \in G$. Given an object g and the set G exactly one of the

Note that the value of 255 in Example 4.10 may appear to have been arbitrarily chosen. The range in fact neatly fits into a 8-bit byte. In practical database design, as in this example, the choices are never arbitrary but depend on the system requirements.

Example 4.11

In the development of a software package, an estimate of the number of lines of code is made and this can only be a positive integer greater than zero. We can therefore define a domain consisting of only positive integers for this application. ■

Definition: **Domain:**

We define a domain, D_i, as a set of values of the same data type.

The domain D_i, a set having "homogeneous" members, is conceptually similar to the data type concept in programming languages. A domain, like a data type, may be unstructured (atomic) or structured. Domain D_i is said to be simple if all its elements are nondecomposable (i.e., atomic). (When we use the term decomposable, we mean in terms of the DBMS.) In typical DBMSs, **atomic domains** are general sets, such as the sets of integers, real numbers, character strings, and so on. Atomic domains are sometimes referred to as **application-independent domains** because these general sets are not dependent on a particular application. We can also define **application-dependent domains** by specifying the values permitted in the particular database. **Structured** or **composite domains** can be specified as consisting of nonatomic values. The domain for the attribute *Address*, for instance, which specifies street number, street name, city, state, and zip or postal code is considered a composite domain.

It is unfortunate that many of the currently available commercial relational database systems do not support the concept of domains. Such support of both application-independent and user-defined domains specified as types in programming languages allows for the validation of the value assigned to an attribute.

Attributes are defined on some underlying domain. That is, they can assume values from the set of values in the domain. Attributes defined on the same domain are comparable, as these attributes draw their values from the same set. It is meaningless to compare attributes defined on different domains, as exemplified below.

Example 4.12

Assume that in a given city house numbers are between 0 and 255. The domain H_Number for the attribute *House_Numbers* can be defined to be the set of values from 0 to 255. The attribute *House_Numbers* is defined over the same domain as *Age* (Example 4.10) and without any additional constraints, they are comparable. Semantically, we say that the domain of *Age* represents a value that is a measure of a number of years and the domain H_Number represents a part of an address. Therefore, comparing the age in years of persons with the house number part of an address is meaningless. Consequently, we have to consider the domain of *House_Numbers* as distinct from the domain of Age, and these domains are not compatible. ■

It is possible, however, to relax the above rule for two semantically compatible domains D_i and D_j where $D_i \cap D_j \neq \phi$. Then attribute A_i defined on domain D_i and attribute A_j defined on D_j can be compared if $a_i \in D_i \cap D_j$ and $a_j \in D_i \cap D_j$. Here, a_i and a_j are the values of attributes A_i and A_j, respectively.

It has become traditional to denote attributes by uppercase letters from the beginning of the alphabet. Thus, *A*, *B*, *C*, . . . , with or without subscripts denote attributes. In applications, however, attributes are given meaningful names. Sets of attributes are denoted by uppercase letters from the end of the alphabets such as . . . , X, Y, Z.

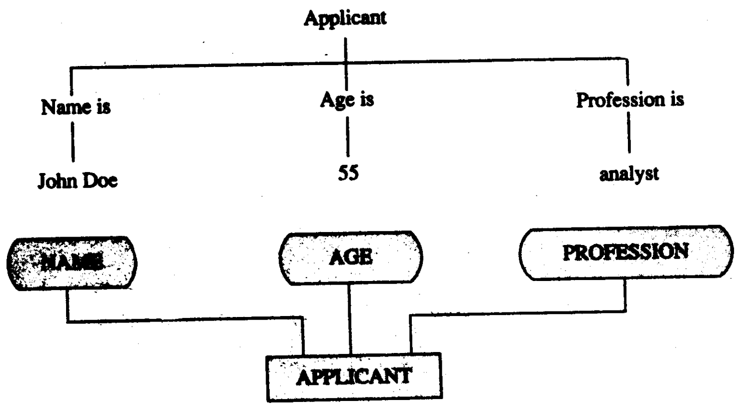
Using the concept of attributes and domains, we can now define a tuple.

4.2.2 Tuples

An entity type having n attributes can be represented by an ordered set of these attributes called an **n-tuple.** Assume that these n attributes take values from the domains $D_1, \ldots, D_n$. The representation of the entity must then be a member of the set $D_1 \times D_2 \times \ldots \times D_n$, as the resulting set of this cartesian product contains all the possible ordered n-tuples.

Example 4.13 A job applicant may be characterized for a particular application by her or his name, age, and profession. An applicant, John Doe, who is 55 years old and is an analyst, may be represented as a 3-tuple: "John Doe, 55, analyst" (Figure F). This is a possible ordered triple obtained from the cartesian product of the domain for attributes *Name*, *Age*, and *Profession*. The implication of this 3-tuple is that an instance of the entity type has the value John Doe for its attribute *Name*, the value 55 for *Age*, and the value analyst for *Profession*. ■

Figure F Representation of a association among attributes.



The value n (the number of attributes in the relation) is known as the degree or arity of the relation. A relation of degree one is called an unary relation, of degree two a binary relation, and of degree n an n-ary relation.[1] Attribute names could be considered a convenience rather than a formal requirement. However, when a number of attributes of a relation are defined on the same domain, the importance of unique attribute names becomes evident. Codd (Codd 70) originally described the relational model referring only to domains.

We formally represent a relation R as a 4-tuple:

$$R(T_R, AN_R, n, m)$$

where T_R represents the set of tuples, $m = |T_R|$ is the cardinality of the relation (i.e., the number of tuples in the relation), AN_R represents the set of attribute names, and $n = |AN_R|$ is the cardinality of the set of attribute names (the degree or arity of the relation).

In the above definition of a relation, we have specified the relation having these constituents: a set of tuples, a scheme (or set of attribute names), the degree, and the cardinality of the relation. The last two are conceptual values as they can be obtained from the set of attributes and tuples, respectively.

It is therefore more usual to represent the relation R defined on a relation scheme $\mathbf{R}$ in terms of just the scheme and set of tuples. The set of tuples of a relation, unless there is confusion, can be expressed by the name of the relation. We shall use an uppercase letter to represent both the relation name and its set of tuples and a bold uppercase letter for the relation's scheme and its set of attributes. This gives us a shorter form of the representation of a relation as simply $R(\mathbf{R})$. The degree (or arity) of the relation is given by the number of attributes in scheme $\mathbf{R}$, i.e., $|\mathbf{R}|$, while the cardinality of the relation is given by the number of tuples in R and is indicated by $|R|$. As such, $R(\mathbf{R})$ represents the relation R defined on scheme $\mathbf{R}$ having the set of tuples R.

We discuss other methods of representing a relation in the following section.

4.2.4 Relation Representation

Conceptually, a relation can be represented as a table. Remember that the contents of a relation are positionally independent, while a table gives the impression of positional addressing. Each column of the table represents an attribute and each row represents a tuple of the relation. Figure 4.2 shows the tabular representation of the APPLICANT relation of Example 4.13.

It is a myth that a relation is just a flat file. A table is just one of the conceptual representations of a relation. It is possible to store the relations using, for instance, inverted files.

As seen in Section 4.2.2, a tuple may be represented either as a labeled n-tuple or as an ordered n-tuple. The labeled n-tuples are represented using distinct attribute names $A_1, \ldots, A_n$ and the values $a_1, \ldots, a_n$ from the corresponding domains. The labeled n-tuples consist of unordered attribute value pairs: $(A_1{:}a_1, \ldots,$

[1]A domain can be thought of as a unary relation.

Figure 4.2 Example representation of a relation as a table.

APPLICANT:

Name	*Age*	*Profession*
John Doe	55	Analyst
Mirian Taylor	31	Programmer
Abe Malcolm	28	Receptionist
Adrian Cook	33	Programmer
Liz Smith	32	Manager

$A_n : a_n$). Ordered n-tuples are represented simply as $(a_1, \ldots, a_n)$, where the values appear in the same order as their domains in the cartesian product of which the relation is a subset.

4.2.5 Keys

In the relational model, we represent the entity by a relation and use a tuple to represent an instance of the entity. Different instances of an entity type are distinguishable and this fact is established in a relation by the requirement that no two tuples of the same relation can be the same. It is possible that only a subset of the attributes of the entity, and therefore the relation, may be sufficient to distinguish between the tuples. However, for certain relations, such a subset may be the complete set of attributes. In the instance of an EMPLOYEE relation, values of an attribute such as *Emp#* may be sufficient to distinguish between employee tuples. Such a subset of attributes, let us say X of a relation $R(\mathbf{R})$, $X \subseteq \mathbf{R}$, with the following time-independent properties is called the key of the relation:

- **Unique identification:** In each tuple of R, the values of X uniquely identify that tuple. To elaborate, if s and t represent any two tuples of a relation and if the values s[X] and t[X] for the attributes in X in the tuples s and t are the same, then s and t must be the same tuple. Therefore, $s[X] = t[X] \Rightarrow s = t$. Here the symbol $\Rightarrow$ is used to indicate that the left-hand side logically implies the right-hand side.
- **Nonredundancy:** No proper subset of X has the unique identification property i.e., no attribute $K \in X$ can be discarded without violating the unique identification property.

Since duplicate tuples are not permitted in a relation, the combination of all attributes of the relation would always uniquely identify its tuples. There may be more than one key in a relation; all such keys are known as candidate keys. One of the candidate keys is chosen as the primary key; the others are known as alternate keys. An attribute that forms part of a candidate key of a relation is called a **prime attribute.**

Example 4.17 In many applications, arbitrary attributes are assigned to the objects and these attributes play the role of keys. *Emp#* is such a key (the domain for the attribute *Emp#* is application specific and unique for a given application). A Social Security number in the U.S. and a Social Insurance number in Canada also identify a person uniquely in these countries. Both numbers are of nine digits and are assigned to individuals without any coordination between these countries. It is likely that the same number may identify two different individuals. Furthermore, there are many individuals who, having lived and worked in both countries, have been assigned different values for their Social Security numbers and Social Insurance numbers. ■

4.2.6 Relationship

The key property and the fact that every tuple must have a key are used to capture relationships between entities.

Example 4.18 An employee may perform different roles in the software development teams working on different products. John Doe may be an analyst in the development team for product "Super File System" and manager of the team for product "$B^{++}1$". The different job requirements are given in the relation JOB_FUNCTION. ■

ASSIGNMENT is a relationship in Figure 4.3a between the entities Employee, Product and Job_Function. A possible representation of this relationship is by using the entities involved in the relationship:

ASSIGNMENT (Employee, Product, Job_Function)

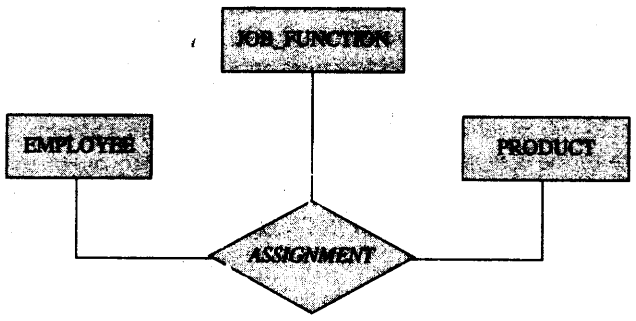
Using the unique identification properties of keys we can replace the Employee, Product, and Job_Function entities in ASSIGNMENT by their keys. The keys act as surrogates for their respective entities. We can represent, let us say, the scheduled duties of an employee by the relation scheme:

ASSIGNMENT (***Emp#***, ***Prod#***, ***Job#***)

ASSIGNMENT is a relation that establishes a relationship among three "owner" relations. Such a relation may be thought of as an **associative relation.** The key of the associative relation is always the union of the key attributes of the owner relations. Thus the key of the relation **ASSIGNMENT** is the combination of the attributes ***Emp#***, ***Prod#***, ***Job#***.

The attributes ***Emp#***, ***Prod#***, and ***Job#*** in the relation ASSIGNMENT are known as **foreign keys.** A foreign key is an attribute or set of attributes of a relation, let us say R(**R**), such that the value of each attribute in this set is that of a primary key of relation S(S) (**R** and S need not be distinct). For instance, we could not have a tuple in the ASSIGNMENT relation of Figure 3 with the value 127 for the attribute ***Emp#*** unless there were a tuple in the EMPLOYEE relation with that value for ***Emp#***. We look at rules applicable to primary and foreign keys in Section 4.2.8.

Figure 4.3 (a) E-R diagram for employee role in development teams; (b) corresponding relation schemes; and (c) sample relations.



(a)

EMPLOYEE *(Emp#, Emp_Name, Profession)*
PRODUCT *(Prod#, Prod_Name, Prod_Details)*
JOB_FUNCTION *(Job#, Title)*
ASSIGNMENT *(Emp#, Prod#, Job#)*

(b)

EMPLOYEE:

Emp#	*Name*	*Profession*
101	Jones	Analyst
103	Smith	Programmer
104	Lalonde	Receptionist
106	Byron	Receptionist
107	Evan	VP R & D
110	Drew	VP Operations
112	Smith	Manager

PRODUCT:

Prod#	*Prod_Name*	*Prod_Details*
HEAP1	HEAP_SORT	ISS module
BINS9	BINARY_SEARCH	ISS/R module
FM6	FILE_MANAGER	ISS/R-PC subsys
B++1	B++_TREE	ISS/R turbo sys
B++2	B++_TREE	ISS/R-PC turbo

JOB_FUNCTION:

Job#	*Title*
1000	CEO
900	President
800	Manager
700	Chief Programmer
600	Analyst

ASSIGNMENT:

Emp#	*Prod#*	*Job#*
107	HEAP1	800
101	HEAP1	600
110	BINS9	800
103	HEAP1	700
101	BINS9	700
110	FM6	800
107	B++1	800

(c)

4.2.7 Relational Operations

Codd (Codd 72) defined a "relationally complete" set of operations and the collection of these, which take one or more relations as their operand(s), forms the basis of relational algebra (to be discussed in Section 4.3). In the same paper Codd included the formal definition of relational calculus (now known as tuple calculus). An

Figure G Foreign keys.

Emp#	*Name*	*Manager*
101	Jones	@
103	Smith	110
104	Lalonde	107
107	Evan	110
110	Drew	112
112	Smith	112

(We can see that using a single null value for all cases can cause problems. Such problems are a topic of research and beyond the scope of this text.)

Definition: **Integrity Rule 2 (Referential Integrity):**

Given two relations R and S, suppose R refers to the relation S via a set of attributes that forms the primary key of S and this set of attributes forms a foreign key in R. Then the value of the foreign key in a tuple in R must either be equal to the primary key of a tuple of S or be entirely null.

If we have the attribute *A* of relation R(**R**) defined on domain D and the primary key of relation S(S) also defined on domain D, then the values of *A* in tuples of R(**R**) must be either null or equal to the value, let us say v, where v is the primary key value for a tuple in S(S). Note that R(R) and S(S) may be the same relation. The tuple in S(S) is called the **target** of the foreign key. The primary key of the referenced relation and the attributes in the foreign key of the referencing relation could be composite.

Referential integrity is very important. Because the foreign key is used as a surrogate for another entity, the rule enforces the existence of a tuple for the relation corresponding to the instance of the referred entity. In Example 4.19, we do not want a nonexisting employee to be manager. The integrity rule also implicitly defines the possible actions that could be taken whenever updates, insertions, and deletions are made.

If we delete a tuple that is a target of a foreign key reference, then three explicit possibilities exist to maintain database integrity:

- All tuples that contain references to the deleted tuple should also be deleted. This may cause, in turn, the deletion of other tuples. This option is referred to as a **domino** or **cascading deletion**, since one deletion leads to another.
- Only tuples that are not referenced by any other tuple can be deleted. A tuple referred by other tuples in the database cannot be deleted.
- The tuple is deleted. However, to avoid the domino effect, the pertinent foreign key attributes of all referencing tuples are set to null.

Similar actions are required when the primary key of a referenced relation is updated. An update of a primary key can be considered as a deletion followed by an insertion.

The choice of the option to use durng a tuple deletion depends on the application. For example, in most cases it would be inappropriate to delete all employees under a given manager on the manager's departure; it would be more appropriate to replace it by null. Another example is when a department is closed. If employees were assigned to departments, then the employee tuples would contain the department key too. Deletion of department tuples should be disallowed until the employees have either been reassigned or their appropriate attribute values have been set to null. The insertion of a tuple with a foreign key reference or the update of the foreign key attributes of a relation require a check that the referenced relation exists.

Although the definition of the relational model specifies the two integrity rules, it is unfortunate that these concepts are not fully implemented in all commercial relational DBMSs. The concept of referential integrity enforcement would require an explicit statement as to what should be done when the primary key of a target tuple is updated or the target tuple is deleted.

4.3 Relational Algebra

Relational algebra is a collection of operations to manipulate relations. We have informally introduced some of these operations such as join (to combine related tuples from two relations), selection (to select particular tuples of a relation) and projection (to select particular attributes of a relation). The result of each of these operations is also a relation.

Relational algebra is a procedural language. It specifies the operations to be performed on existing relations to derive result relations. Furthermore, it defines the complete scheme for each of the result relations. The relational algebraic operations can be divided into basic set-oriented operations and relational-oriented operations. The former are the traditional set operations, the latter, those for performing joins, selection, projection, and division.

4.3.1 Basic Operations

Basic operations are the traditional set operations: union, difference, intersection, and cartesian product. Three of these four basic operations—union, intersection, and difference—require that operand relations be **union compatible.**[2] Two relations are union compatible if they have the same arity and one-to-one correspondence of the attributes with the corresponding attributes defined over the same domain. The cartesian product can be defined on any two relations. Two relations P(P) and Q(Q) are

[2]We assume that in the case of the union, difference, and intersection operations, the names of the attributes of the operand relations are the same and that the result relation inherits these names. If these names are not identical, some convention, for instance, using the names from the first operand relation, must be provided to assign names to the attributes of the result relation.

The intersection operation is really unnecessary. It can be very simply expressed as:

$$P \cap Q = P - (P - Q)$$

It is, however, more convenient to write an expression with a single intersection operation than one involving a pair of difference operations.

Note that in these examples the operand and the result relation schemes, including the attribute names, are identical i.e., **P ≡ Q ≡ R.** If the attribute names of compatible relations are not identical, the naming of the attributes of the result relation will have to be resolved.

Cartesian Product (×)

The extended cartesian or simply the cartesian product of two relations is the concatenation of tuples belonging to the two relations. A new resultant relation scheme is created consisting of all possible combinations of the tuples.

$$R = P \times Q$$

where a tuple $r \in R$ is given by $\{t_1 \parallel t_2 \mid t_1 \in P \wedge t_2 \in Q\}$, i.e., the result relation is obtained by concatenating each tuple in relation P with each tuple in relation Q. Here, $\parallel$ represents the concatenation operation.

The scheme of the result relation is given by:

$$\mathbf{R} = \mathbf{P} \parallel \mathbf{Q}$$

The degree of the result relation is given by:

$$|\mathbf{R}| = |\mathbf{P}| + |\mathbf{Q}|$$

The cardinality of the result relation is given by:

$$|R| = |P| * |Q|$$

Example 4.24 The cartesian product of the PERSONNEL relation and SOFTWARE_PACKAGE relations of Figure Ji is shown in Figure Jii. Note that the relations P and Q from Figure H of Example 4.20 are a subset of the PERSONNEL relation. ■

Figure J (i) PERSONNEL(*Emp#*,*Name*) and SOFTWARE_PACKAGES(*S*) represent employees and software packages respectively; (ii) the Cartesian product of PERSONNEL and SOFTWARE_PACKAGES.

PERSONNEL:

Id	*Name*
101	Jones
103	Smith
104	Lalonde
106	Byron
107	Evan
110	Drew
112	Smith

SOFTWARE_PACKAGES:

S
J_1
J_2

(i)

P.*Id*	P.*Name*	*S*
101	Jones	J_1
101	Jones	J_2
103	Smith	J_1
103	Smith	J_2
104	Lalonde	J_1
104	Lalonde	J_2
106	Byron	J_1
106	Byron	J_2
107	Evan	J_1
107	Evan	J_2
110	Drew	J_1
110	Drew	J_2
112	Smith	J_1
112	Smith	J_2

(ii)

The union and intersection operations are associative and commutative; therefore, given relations R(**R**), S(**S**), T(**T**):

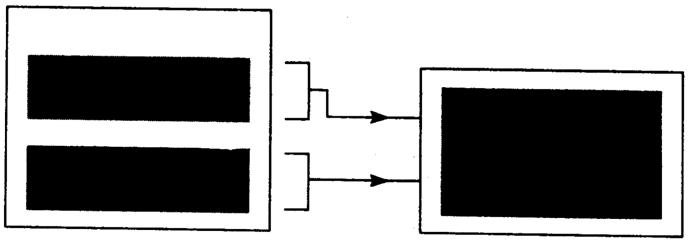
$$R \cup (S \cup T) = (R \cup S) \cup T = (S \cup R) \cup T = T \cup (S \cup R) = \ldots$$
$$R \cap (S \cap T) = (R \cap S) \cap T = \ldots$$

The difference operation, in general, is noncommutative and nonassociative.

$R - S \neq S - R$ noncommutative

$R - (S - T) \neq (R - S) - T$ nonassociative

Figure 4.6 (a) Graphical representation of selection that selects a subset of the tuples; (b) result of selection over PERSONNEL for *Id* < 105.



(a)

PERSONNEL:

Id	*Name*
101	Jones
103	Smith
104	Lalonde
106	Byron
107	Evan
110	Drew
112	Smith

Result of selection

Id	*Name*
101	Jones
103	Smith
104	Lalonde

(b)

Any finite number of predicates connected by Boolean operators may be specified in the selection operation. The predicates may define a comparison between two domain-compatible attributes or between an attribute and a constant value; if the comparison is between attribute A_i and constant c_i, then $c_i \in \mathrm{Dom}(A_i)$.

Given a relation P and a predicate expression B, the selections of those tuples of relation P that satisfy the predicate B is a relation R written as:

$$R = \sigma_B(P)$$

The above expression could be read as "select those tuples t from P in which the predicate B(t) is true." The set of tuples in relation R are in this case defined as follows:

$$R = \{t \mid t \in P \wedge B(t)\}$$

Join (⋈)

The join operator, as the name suggests, allows the combining of two relations to form a single new relation. The tuples from the operand relations that participate in the operation and contribute to the result are related. The join operation allows the processing of relationships existing between the operand relations.

In Figure D of Example 4.3 we illustrated an example of a join of the relations SOME_DBMS and VERSION. We joined those tuples of the two relations that had the same value for the common attribute *Name* defined on a common domain. In this case, this common value was used to establish a relationship between these relations. Note that referential integrity dictates that a tuple in VERSION could not exist without a tuple in SOME_DBMS with the same value for the *Name* attribute. Join is basically the cartesian product of the relations followed by a selection operation.

Example 4.26 In Figure 4.3 we encountered the following relations:

ASSIGNMENT (***Emp#, Prod#, Job#***)
JOB_FUNCTION (***Job#***, *Title*)

Suppose we want to respond to the query "Get product number of assignments whose development teams have a chief programmer." This requires first computing the cartesian product of the ASSIGNMENT and JOB_FUNCTION relations. Let us name this product relation TEMP. This is followed by selecting those tuples of TEMP where the attribute *Title* has the value chief programmer and the value of the attribute *Job#* in ASSIGNMENT and JOB_FUNCTION are the same. The required result, shown below, is obtained by projecting these tuples on the attribute *Prod#*. The operations are specified below:

TEMP = (ASSIGNMENT × JOB_FUNCTION)

$\pi_{Prod\#}(\sigma_{Title\ =\ \text{'chief programmer'}\ \wedge\ \text{ASSIGNMENT}.Job\ =\ \text{JOB\_FUNCTION}.Job\#}$
(TEMP))

Prod#
HEAP1
BINS9

In another method of responding to this query, we can first select those tuples from the JOB_FUNCTION relation so that the value of the attribute *Title* is chief programmer. Let us call this set of tuples the relation TEMP1. We then compute the cartesian product of TEMP1 and ASSIGNMENT, calling the product TEMP2. This is followed by a projection on *Prod#* over TEMP2 to give us the required response. These operations are specified below:

TEMP1 = $(\sigma_{Title\ =\ \text{'}chief\ \text{programmer'}}$(JOB_FUNCTION))
TEMP2 = $(\sigma_{\text{ASSIGNMENT}.Job\#\ =\ \text{JOB\_FUNCTION}.Job\#}$(ASSIGNMENT ×
TEMP1))
$\pi_{Prod\#}$(TEMP2) gives the required result. ■

Notice that in the selection operation that follows the cartesian product we take only those tuples where the value of the attributes ASSIGNMENT.*Job#* and JOB_FUNCTION.*Job#* are the same. These combined operations of cartesian product followed by selection are the join operation. Note that we have qualified the identically named attributes by the name of the corresponding relation to distinguish them.

Consider the ASSIGNMENT relation of Figure 4.3c. If we want to find the coworkers in all projects (but not necessarily doing the same job) we can join ASSIGNMENT with itself on the *Prod#* attribute. However, to have unique attribute names in the result relation, we can proceed as follows. Copy ASSIGNMENT into COASSIGN(***Emp#*, *Prod#*, *Job#***) and then perform the operation given below, using qualified attribute names. The result of the operation is shown in Figure Kb. Note that a simple join of ASSIGNMENT with itself, using the definition of natural join, gives the original relation:

$$\pi_{(\text{ASSIGNMENT}.\mathit{Emp\#},\text{COASSIGN}.\mathit{Emp\#})}(\text{ASSIGNMENT} \underset{\text{ASSIGNMENT}.\mathit{Prod\#} = \text{COASSIGN}.\mathit{Prod\#}}{\bowtie} \text{COASSIGN})$$ ■

Formally, the natural join of P(**P**) and Q(**Q**) is performed on the attributes of **P** and **Q** defined on common domains, i.e., **P** ∩ **Q**. The resultant relation consists of the attributes **P** ∪ **Q**.

In the cartesian product of two relations, we take a tuple from each relation and concatenate them to obtain a tuple in the result relation. Any duplication of attributes in the tuples, as well as duplicate tuples, remains. (Note that duplicate tuples are not generated in a cartesian product of two proper relations.) In a relational join, we select the subset of the product tuples that satisfy the join predicates. In an equi-join, the predicate involves equality constraints. In a natural join, which also involves equality constraints, the common attributes are not duplicated. In the majority of cases when we speak of a join, we are actually speaking about the natural join.

If two relations that are to be joined have no domain-compatible attributes, the natural join operation is equivalent to a simple cartesian product. If they have identical relation schemes, the natural join operation is an intersection operation.

We can summarize the above discussion on the various types of join operations using the cartesian product as follows:

- The equi-join and the theta join are horizontal subsets of the cartesian product. This is equivalent to applying a selection to the resulting tuple of the cartesian product. The selection is explicitly specified in the theta join and implicitly specified in the equi-join.
- The natural join is equivalent to an equi-join with a subsequent projection to eliminate the duplicate attributes. In this sense, a natural join is both a horizontal and vertical subset of the cartesian product.

Division (÷)

Before we define the division operation, let us consider an example.

Example 4.28

Given the relations P(**P**) and Q(**Q**) as shown in Figure Li, the result of dividing P by Q is the relation R and it has two tuples. For each tuple in R, its product with the tuples of Q must be in P. In our example (a_1,b_1) and (a_1,b_2) must both be tuples in P; the same is true for (a_5,b_1) and (a_5,b_2).

Figure L Examples of the division operation. (i) R = P ÷ Q; (ii) R = P ÷ Q (P is the same as in part i); (iii) R = P ÷ Q (P is the same as in part i); (iv) R = P ÷ Q (P is the same as in part i).

P(P):

A	B
a_1	b_1
a_1	b_2
a_2	b_1
a_3	b_1
a_4	b_2
a_5	b_1
a_5	b_2

Q(Q):

B
b_1
b_2

R(R) (result):

A
a_1
a_5

(i)

Q(Q):

B
b_1

then R(R) is:

A
a_1
a_2
a_3
a_5

Q(Q):

B
b_1
b_2
b_3

then R(R) is:

A

(iii)

Q(Q):

B

then R(R) is:

A
a_1
a_2
a_3
a_4
a_5

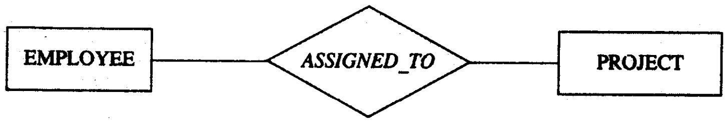
(iv)

Simply stated, the cartesian product of Q and R is a subset of P.

In Figure Lii, the result relation R has four tuples; the cartesian product of R and Q gives a resulting relation which is again a subset of P.

In Figure Liii, since there are no tuples in P with a value b_3 for the

Figure 4.7 Sample database



EMPLOYEE:

Emp#	*Name*
101	Jones
103	Smith
104	Lalonde
106	Byron
107	Evan
110	Drew
112	Smith

ASSIGNED_TO

Project#	*Emp#*
COMP453	101
COMP354	103
COMP343	104
COMP354	104
COMP231	106
COMP278	106
COMP353	106
COMP354	106
COMP453	106
COMP231	107
COMP353	107
COMP278	110
COMP353	112
COMP354	112

PROJECT:

Project#	*Project_Name*	*Chief_Architect*
COMP231	Pascal	107
COMP278	Pascal/Object	110
COMP353	Database	107
COMP354	Operating Sys	104
COMP453	Database	101

relationship *ASSIGNED_TO* between them. Some sample tuples from these relations are shown in Figure 4.7.

PROJECT ***(Project#, Project_Name, Chief_Architect)***
EMPLOYEE ***(Emp#, EmpName)***
ASSIGNED_TO ***(Project#, Emp#)***

Example 4.30 "Get *Emp#* of employees working on project COMP353." To evaluate this query, we select those tuples of relation ASSIGNED_TO such that the value of the *Project#* attribute is COMP353. We then project the result on the attribute *Emp#* to get the response relation. The query and the response relation are shown below:

$$\pi_{Emp\#}(\sigma_{Project\# = \text{'COMP353'}}(\text{ASSIGNED\_TO}))$$

Emp#
106
107
112

■

The following example entails a join of two relations.

Example 4.31

"Get details of employees (both number and name) working on project COMP353." The first part of the evaluation of this query is the same as in the query in Example 4.30. It is, however, followed by a natural join of the result with EMPLOYEE relation to gather the complete details about the employees working on project COMP353. The result and the the query are shown below:

$$\text{EMPLOYEE} \bowtie \pi_{Emp\#}(\sigma_{Project\# = \text{'COMP353'}}(\text{ASSIGNED\_TO}))$$

Emp#	*Name*
106	Byron
107	Evan
112	Smith

■

Example 4.32 requires using three relations to generate the required response.

Example 4.32

"Obtain details of employees working on the Database project." This query requires two joins. The first step is to find the number(s) of the project(s) named Database. This involves a selection of the relation PROJECT, followed by a projection on the attribute *Project#*. The result of this projection is joined with the ASSIGNED_TO relation to give tuples of the ASSIGNED _TO involving Database. This is projected on *Emp#* and subsequently joined with EMPLOYEE to get the required employee details. The query in relational algebra and the result are shown below:

$$\text{EMPLOYEE} \bowtie \pi_{Emp\#}(\text{ASSIGNED\_TO} \bowtie (\pi_{Project\#}(\sigma_{Project\_Name = \text{'Database'}}(\text{PROJECT}))))$$

Emp#	*Name*
101	Jones
106	Byron
107	Evan
112	Smith

■

$(\pi_{Emp\#}(\text{ASSIGNED\_TO} \bowtie \pi_{Project\#}(\sigma_{Emp\# = 107}(\text{ASSIGNED\_TO})))) - 107$

Emp#
106
112

■

4.4 Relational Calculus

Tuple and domain calculi are collectively referred to as relational calculus. As we have seen, queries in relational algebra are procedural. In general, a user should not have to be concerned with the details of how to obtain information. In relational calculus, a query is expressed as a formula consisting of a number of variables and an expression involving these variables. The formula describes the properties of the result relation to be obtained. There is no mechanism to specify how the formula should be evaluated. It is up to the DBMS to transform these nonprocedural queries into equivalent, efficient, procedural queries. In relational tuple calculus, the variables represent the tuples from specified relations; in relational domain calculus, the variables represent values drawn from specified domains.

Relational calculus is a query system wherein queries are expressed as variables and formulas on these variables. Such formulas describe the properties of the required result relation without specifying the method of evaluating it.

Relation calculus, which in effect means calculating with relations, is based on **predicate calculus,** which is calculating with predicates. The latter is a formal language used to symbolize logical arguments in mathematics. In the following paragraphs we briefly introduce predicate calculus; additional details are given in Chapter 16.

In formal logic the main subject matter is propositions. If, for instance, p and q are propositions, we can build other propositions "not p," "p or q," "p and q," and so on. In predicate calculus, propositions may be built not only out of other propositions but also out of elements that are not themselves propositions. In this manner we can build a proposition that specifies a certain property or characteristic of an object.

Propositions specifying a property consist of an expression that names an individual object (it may also be used to designate an object), and another expression, called the **predicate,** that stands for the property that the individual object possesses.

Example 4.38 Consider these statements:

BCD is a company
WXY is a company
Jill is an analyst
John is an analyst
Canada is a country
U.S.A. is a country

Each of these is a statement about an object having a certain feature or property. In these examples, the parts "is a company," "is an analyst," "is a country" are instances of predicates. Each describes some property or characteristic of an object. ■

A convenient method of writing the statements of Example 4.38 is to place the predicate first and follow it with the object enclosed in parentheses. Therefore, the statement "BCD is a company" can be written as "is a company(BCD)." Now we drop the "is a" part and write the first statement as "company(BCD)." Finally, if we use symbols for both the predicate and the object, we can rewrite the statements of Example 4.38 as P(x). The lowercase letters from the end of the alphabet (. . . x, y, z) denote variables, the beginning letters (a, b, c, . . .) denote constants, and uppercase letters denote predicates. P(x), where x is the argument, is a **one-place** or **monadic predicate.** DBMS(x) and COMPANY(y) are examples of monadic predicates. The variables x and y are replaceable by constants (or names of individual objects) such as DBMS(ISS).

The use of constants and variables is similar to that in some high-level languages. A constant specifies a particular value or object; a variable is used as a placeholder for the values in an expression or procedure.

Example 4.39 Consider these statements:

Jill is taller than John
WXY is bigger than BCD
Canada is north of the U.S.A.

In these statements, the predicates "is taller than," "is bigger than," "is north of" require two objects and are called **two-place predicates.** ■

In general, we have predicates of degree n, where the predicate takes n arguments. In the case of bigger_than(WXY, BCD), the predicate BIGGER_THAN specifies the relation between WXY and BCD.

Example 4.40 Let DBMS_TYPE(x,y) specify the relation between DBMSs and their data model. The predicate DBMS_TYPE takes two arguments. ■

A predicate followed by its arguments is called an **atomic formula.** Examples of these are DBMS(x), COMPANY(y), and DBMS_TYPE(x,y).

We stated earlier that predicate calculus is a formal language. A language consists of symbols. We have already seen some of the primitive symbols, i.e., variables, constants, and predicates. We can also specify logical connectors such as "not" or negation, denoted by $\neg$, "or" ($\vee$), "and" ($\wedge$), and "implication" ($\rightarrow$).

Atomic formulas may be combined using the logical connectors to generate formulas such as $P(x) \wedge Q(y)$, $P(x) \vee Q(y)$, and so on. DBMS(ISS) $\wedge$ COMPANY(BCD), for instance, can represent "ISS is a DBMS and BCD is a company."

In this formulation, we specify the set of tuples t(*Emp#*) such that the predicate is true for each element of that set. The predicate specifies that there exists some tuple, u, in the relation ASSIGNED_TO such that it has the value COMP353 for the *Project#* attribute. Also, the value for the *Emp#* attributes of the result tuple t is the same as that for the tuple u.

Free variables appear to the left of the | (bar) symbol. The variable t is a free tuple variable in the above formula and assumes whatever attributes and corresponding values, assigned to it by the formula. The formula restricts t to the relation scheme (*Emp#*).

Example 4.43 Consider this query: "Obtain a list of employees (both numbers and names) working on the project COMP353," which can be rephrased as: "Obtain employee details for those employees assigned to the project COMP353."

To verify whether or not an employee is working on COMP353, we can compare the employee's *Emp#* with *Emp#* values of tuples in the relation ASSIGNED_TO. What we are really specifying is that "for the employee whose details we want, there exists a tuple in the relation ASSIGNED _TO for that employee with the value of the attribute *Project#* in that tuple being COMP353." This is a calculuslike formulation for our query. In the database we use surrogates to represent entities. For example, *Emp#* is used to represent an employee in the ASSIGNED_TO relation (*Project#* is used to represent a project). To check if an employee is working on some project, we would need to compare the employee's surrogate, *Emp#*, from EMPLOYEE, with the tuples of the ASSIGNED_TO relation containing the project's surrogate, *Project#*. Thus, the query can be reformulated as: "Get those tuples in employee relation such that there exists an ASSIGNED_TO tuple with ASSIGNED_TO.*Emp#* = EMPLOYEE.*Emp#* and ASSIGNED_TO.*Project#* = COMP353."

In tuple calculus this can be specified as:

$$\begin{aligned}\{t \mid \exists e(e \in \text{EMPLOYEE} \wedge e[\mathit{Emp\#}] = t[\mathit{Emp\#}]\\ \wedge\, e[\mathit{EmpName}] = t[\mathit{EmpName}]\\ \wedge\, \exists u(u \in \text{ASSIGNED\_TO} \wedge u[\mathit{Emp\#}] = e[\mathit{Emp\#}]\\ \wedge\, u[\mathit{Project\#}] = \text{'COMP353'}))\}\end{aligned}$$

The above may be simplified to the following form where the domain of the free variable t is the relation EMPLOYEE.

$$\begin{aligned}\{t \mid t \in \text{EMPLOYEE}\\ \wedge\, \exists u(u \in \text{ASSIGNED\_TO} \wedge u[\mathit{Emp\#}] = t[\mathit{Emp\#}]\\ \wedge\, u[\mathit{Project\#}] = \text{'COMP353'}))\}\end{aligned}$$ ■

In the tuple calculus query formulations given above, we have only specified the characteristics of the desired result. The system is free to decide the operations and their execution order to satisfy the request. For comparison, a relational algebra like query would have to be stated as, "Select tuples from ASSIGNED_TO such that *Project#* = 'COMP353' and perform their join with the employee relation, projecting the results of the join over *Emp#* and *EmpName*." It is obvious that a calculus query is much simpler because it is devoid of procedural details.

Tuple Calculus Formulas

At this point it is useful to see how tuple calculus formulas are derived. A variable appearing in a formula is said to be free unless it is quantified by the existential (for some) quantifier, $\exists$ or the universal (for all) quantifier, $\forall$. Variables quantified by or are said to be bound.

In tuple calculus we define a qualified variable as t[*A*], where t is a tuple variable of some relation and *A* is an attribute of that relation. Two qualified variables, s[*A*] and t[*B*], are domain compatible if attributes *A* and *B* are domain compatible.

Tuple calculus formulas are built from **atoms.** An atom is either of the forms given below:

A_1. x $\in$ R, where R is a relation and x is a tuple variable.

A_2. x θ y or x θ c, where θ is one of the comparison operators $\{=, \neq, <, \leq, >, \geq\}$, x and y are domain-compatible qualified variables, and c is a domain compatible-constant.

For example, s[*A*] = t[*B*] is an atom in tuple calculus, where s and t are tuple variables.

Formulas (wffs) are built from atoms using the following rules:

B_1. An atom is a formula.

B_2. If f and g are formulas, then $\neg$f, (f), f $\vee$ g, f $\wedge$ g, f $\rightarrow$ g are also formulas.

B_3. If f(x) is a formula where x is free, then $\exists$x(f(x)) and $\forall$x(f(x)) are also formulas; however, x is now bound.

The logical implication expression f $\rightarrow$ g, meaning *if* f *then* g, is equivalent to $\neg$f $\vee$ g. Some well-formed formulas in tuple calculus are given below:

u $\in$ ASSIGNED_TO (declares u as a tuple variable; the domain of u is the relation ASSIGNED_TO)

u[*Project#*] = 'COMP353'

u $\in$ ASSIGNED_TO $\wedge$ u[*Project#*] = 'COMP353'

$\exists$u(u $\in$ ASSIGNED_TO $\wedge$ s $\in$ EMPLOYEE
  $\wedge$ u[*Project#*] = 'COMP353'
    $\wedge$ s[*Emp#*] = u[*Emp#*])
      (here u is a bound variable, and s is a free variable)

$\exists$u,t (u $\in$ ASSIGNED_TO $\wedge$ s $\in$ EMPLOYEE $\wedge$ t $\in$ PROJECT
    $\wedge$ t[*Project_Name*] = 'Database'
    $\wedge$ u[*Project#*] = *t*[*Project#*]
    $\wedge$ *s*[*Emp#*] = u[*Emp#*])

In the following examples we give some sample queries in tuple calculus using the relations shown in Figure 4.7.

Example 4.44

"Get complete details of employees working on a Database project," The query can be stated as given below. In this case, the tuple variable s is defined on the relation EMPLOYEE and it appears by itself to signify that we are interested in all attributes of its domain relation. We are saying that there exist tuples u and t on the domain relations ASSIGNED_TO and PROJECT, respectively, such that the conditions indicated below are true. The tuple t has for the *Project_Name* attribute a value of 'Database,' and the *Project#* in u and t are the same. The *Emp#* value of s and u are the same, as well. Note that ∃u,t(F(u,t)) is a shorthand notation for ∃u(∃t(F(u,t))).

{s | s ∈ EMPLOYEE
 ∧ ∃u, t(t ∈ PROJECT ∧ t[*Project_Name*] = 'Database'
 ∧ u ∈ ASSIGNED_TO ∧ u[*Project#*] = t[*Project#*]
 ∧ s[*Emp#*] = u[*Emp#*])}

The query "Get complete details of employees working on all Database projects" can be expressed as follows:

{s | s ∈ EMPLOYEE
 ∧ ∀t(t ∈ PROJECT ∧ t[*Project_Name*] = 'Database'
 → ∃u(u ∈ ASSIGNED_TO ∧ u[*Project#*] = t[*Project#*]
 ∧ s[*Emp#*] = u[*Emp#*]))}

An alternate method of writing this query without the logical implication is to replace f → g by its equivalent form ¬f ∨ g as follows:

{s | s ∈ EMPLOYEE
 ∧ ∀(t ∉ PROJECT ∨ t[*Project_Name*] ≠ 'Database'
 ∨ ∃u(u ∈ ASSIGNED_TO ∧ u[*Project#*] = t[*Project#*]
 ∧ s [*Emp#*] = u[*Emp#*]))} ■

Any number of tuple variables can have the same relation as their domain as illustrated in the following example.

Example 4.45

"List the complete details about employees working on both COMP353 and COMP354." In this instance, we require that there exist two tuples u_1, u_2 of the relation ASSIGNED_TO with the values COMP353 and COMP354 for the attribute *Project#*. The *Emp#* attributes of s, u_1, and u_2 are equal.

{s | s ∈ EMPLOYEE ∧ ∃u_1,u_2 (u_1 ∈ ASSIGNED_TO
 ∧ u_2 ∈ ASSIGNED_TO ∧ u_1[*Emp#*] = u_2[*Emp #*]
 ∧ s[*Emp#*] = u_1[*Emp#*] ∧ u_1[*Project#*] = 'COMP353'
 ∧ u_2[*Project#*] = 'COMP354')}

We modify the above query to read "List the complete details about employees working on either COMP353 or COMP354 or both." Here we require that there exist tuples u_1 of the relation ASSIGNED_TO with the value COMP353 or u_2 of the same relation with the value COMP354 for the attribute *Project#*. The two "there exist" clauses are connected by the ∨ operator. The *Emp#* attribute of s and either u_1 or u_2, are equal.

$\{s \mid s \in \text{EMPLOYEE} \wedge \exists u_1(u_1 \in \text{ASSIGNED\_TO}$
$\quad \wedge\ s[\textit{Emp\#}] = u_1[\textit{Emp\#}] \wedge u_1[\textit{Project\#}] = \text{'COMP353'}$
$\quad \vee\ \exists u_2(u_2 \in \text{ASSIGNED\_TO}$
$\qquad \wedge\ s[\textit{Emp\#}] = u_2[\textit{Emp\#}] \wedge u_2[\textit{Project\#}] = \text{'COMP354'})\}$

This query can be simplified to the following form:

$\{s \mid s \in \text{EMPLOYEE} \wedge \exists u_1(u_1 \in \text{ASSIGNED\_TO}$
$\quad \wedge\ s\,[\textit{Emp\#}] = u_1[\textit{Emp\#}] \wedge$
$\quad (u_1[\textit{Project\#}] = \text{'COMP353'} \vee u_1[\textit{Project\#}] = \text{'COMP354'}))\}$ ■

The following example illustrates the use of the universal quantifier.

Example 4.46 "Get the employee numbers of employees other than employee 107 who work on at least all those projects that employee 107 works on." Here a qualified variable, t[*Emp#*], is used to indicate that we are interested in finding the projection of tuple t on the attribute *Emp#*. The tuple t is from the relation ASSIGNED_TO, such that for all tuples u_1 from ASSIGNED_TO with $u_1[\textit{Emp\#}] = 107$, there exists a tuple $u_2 \in$ ASSIGNED_TO with $u_2[\textit{Emp\#}] \neq 107$. The value of the attribute *Project#* in u_2 is the same as in u_1 with identical values in the attribute *Emp#* of tuples t and u_2. The tuple expression for this query is given below:

$\{t[\textit{Emp\#}] \mid t \in \text{ASSIGNED\_TO} \wedge$
$\quad \forall u_1(u_1 \in \text{ASSIGNED\_TO} \wedge u_1[\textit{Emp\#}] = 107$
$\quad \rightarrow \exists u_2(u_2 \in \text{ASSIGNED\_TO} \wedge u_2[\textit{Emp\#}] \neq 107$
$\quad \wedge\ u_1[\textit{Project\#}] = u_2[\textit{Project\#}] \wedge t[\textit{Emp\#}] = u_2[\textit{Emp\#}]))\}$

Alternatively we can write this query without the logical implication by substituting its equivalent form $\neg f \vee g$ as follows:

$\{t[\textit{Emp\#}] \mid t \in \text{ASSIGNED\_TO} \wedge$
$\quad \forall u_1(u_1 \notin \text{ASSIGNED\_TO} \vee u_1[\textit{Emp\#}] \neq 107$
$\quad \vee\ \exists u_2(u_2 \in \text{ASSIGNED\_TO} \wedge u_2[\textit{Emp\#}] \neq 107$
$\quad \wedge\ u_1[\textit{Project\#}] = u_2[\textit{Project\#}] \wedge t[\textit{Emp\#}] = u_2[\textit{Emp\#}]))\}$

To avoid a procedural operation such as projection in a calculus query, we could define t to be on the relation scheme (*Emp#*) and rewrite this query expression as:

$\{t(\textit{Emp\#}) \mid \forall u_1(u_1 \notin \text{ASSIGNED\_TO} \vee u_1\,[\textit{Emp\#}] \neq 107$
$\quad \vee\ \exists u_2(u_2 \in \text{ASSIGNED\_TO} \wedge u_2[\textit{Emp\#}] \neq 107$
$\quad \wedge\ u_1[\textit{Project\#}] = u_2[\textit{Project\#}] \wedge t[\textit{Emp\#}] = u_2[\text{Emp\#}]))\}$ ■

Negation and its transformation is illustrated in Example 4.47.

Example 4.47 "Get employee numbers of employees who do not work on project COMP453." In this query we are interested in a qualified tuple variable.

t[*Emp#*], t ∊ ASSIGNED_TO, to satisfy the following condition: There does not exist a tuple u in the same relation such that the *Project#* attribute of u has the value COMP453 with identical values in the attribute *Emp#* of tuples t and u. The tuple calculus expression for this query is given below:

{t[[*Emp#*]| t ∊ ASSIGNED_TO ∧
 ¬∃(u ∊ ASSIGNED_TO ∧ u[[*Project#*] = 'COMP453'
 ∧ t[*Emp#*] = u[*Emp#*])}

Alternatively, we can express this query in the following equivalent form:

{t[*Emp#*]| t ∊ ASSIGNED_TO ∧
 ∀u(u ∉ ASSIGNED_TO ∨ t[*Emp#*] ≠ u[*Emp#*]
 ∨ u[*Project#*] ≠ 'COMP453')} ■

To find employees who work on all projects we use the universal quantifier and logical implication.

Example 4.48 "Compile a list of employee numbers of employees who work on all projects." The qualified tuple variable t[*Emp#*] satisfies the following predicates: For all tuples p from PROJECT, there exists a tuple u in ASSIGNED_TO such that the value of *Project#* in u and p are the same, and furthermore, the value of the qualified tuple variables t[*Emp#*] *and u*[*Emp#*] are the same.

{t[*Emp#*]| t ∊ ASSIGNED_TO ∧
 ∀p(p ∊ PROJECT → ∃u(u ∊ ASSIGNED_TO
 ∧ p[*Project#*] = u[*Project#*]
 ∧ *t*[*Emp#*] = u[*Emp#*]))}

The above can be rewritten as:

{t[*Emp#*]| t ∊ ASSIGNED_TO ∧
 ∀p(p ∉ PROJECT ∨ ∃u(u ∊ ASSIGNED_TO
 ∧ p[*Project#*] = u[*Project#*]
 ∧ t[*Emp#*] = u[*Emp#*]))} ■

The following example illustrates a method of finding employees who work at least one of a selected group of projects.

Example 4.49 "Get employee numbers of employees, not including employee 107, who work on at least one project that employee 107 works on." We are concerned here with a tuple t such that there exist tuples s and u in the relation ASSIGNED_TO, such that for the tuples s and u, the value of *Project#* is identical with the value of the attribute *Emp#;* in s, 107 and in t, not 107. The value of the attribute *Emp#* in t and u is the same. This query can be expressed in tuple calculus as follows:

$$\begin{aligned}
&\{t[\mathit{Emp\#}] \mid t \in \text{ASSIGNED\_TO} \wedge \\
&\quad \exists s, u\ (s \in \text{ASSIGNED\_TO} \wedge u \in \text{ASSIGNED\_TO} \\
&\quad \wedge s[\mathit{Project\#}] = u[\mathit{Project\#}] \\
&\quad \wedge s[\mathit{Emp\#}] = 107 \\
&\quad \wedge t[\mathit{Emp\#}] \neq 107 \\
&\quad \wedge t[\mathit{Emp\#}] = u[\mathit{Emp\#}])\} \quad \blacksquare
\end{aligned}$$

We can use tuple calculus to define the division operation on the two relations P(**P**) and Q(**Q**), where $\mathbf{Q} \subseteq \mathbf{P}$:

$$R = P \div Q$$

The tuples in R are those projections of P on the set of attributes **P − Q** such that each tuple in the relation Q, when concatenated with all the tuples in R, gives the tuples in P. We can express this conditions for tuples in R as follows:

$$R = \{t \mid t \in P[\mathbf{P-Q}] \wedge \forall s (s \in Q \wedge (t \| s \in P)\}$$

To simplify the above, we can say that the tuples in R are those projection of tuples in P such that for all tuples s in Q there is a tuple u in P, which when projected on **Q** gives s and when projected on **P − Q** gives the tuples in R. In other words, the tuples in R are elements of the projection of P, on **P − Q**, each of which when concatenated with all tuples s of Q is an element of P. We can express this modification to conditions for tuples in R as follows:

$$R = \{t \mid t \in P[\mathbf{P-Q}] \wedge \forall s (s \in Q \rightarrow \exists u (u \in P \wedge u[\mathbf{Q}] = s \wedge u[\mathbf{P-Q}] = t[\mathbf{P-Q}])))\}$$

From this second specification, we can express the division operation in terms of the other relational algebraic operations as:

$$R = P \div Q = \pi_{\mathbf{P-Q}}(P) - \pi_{\mathbf{P-Q}}((\pi_{\mathbf{P-Q}}(P) \times Q) - P)$$

We illustrate the above using the relations P(**P**) and Q(**Q**) shown in Figure Li of Example 4.28. The term $\pi_{\mathbf{P-Q}}(P)$ gives all objects in the relation P. Some of these objects do not have all the properties given in Q. The term $\pi_{\mathbf{P-Q}}(P) \times Q - P$ gives those tuples of P that will not participate in the result of the division. To find the objects that do not have all the properties in Q, we project these nonparticipating tuples on the attributes **P − Q**. The result is obtained by subtracting these nonparticipating objects from all objects. These steps are illustrated in Figure 4.8.

4.4.2 Domain Calculus

As in tuple calculus, a **domain calculus** expression is of the form

$$\{X \mid F(X)\}$$

where F is a formula on X and X represents a set of domain variables. The expression characterizes X such that F(X) is true.

For the examples in this section, we continue to use the same database that we

$<a,b> \in$ ASSIGNED_TO (declares a and b as domain variables defined on the domain of the attributes of the ASSIGNED_TO relation)
$a =$ 'COMP353'
$<a,b> \in$ ASSIGNED_TO $\wedge$ $a =$ 'COMP353'
$\exists a,b$ ($<a,b> \in$ ASSIGNED_TO $\wedge$ $<c,d> \in$ EMPLOYEE $\wedge$ $a =$ 'COMP353' $\wedge$ $b=c$)
$\exists a,b,e,f$ ($<a,b> \in$ ASSIGNED_TO $\wedge$ $<c,d> \in$ EMPLOYEE
  $\wedge$ $<e,f,g> \in$ PROJECT
  $\wedge$ $b = c \wedge a = e \wedge f =$ 'Database')

(Note that g is used as a placeholder, so that we know what domain the variable belongs to.)

Here we give some sample queries in domain calculus. We continue to use the relations given below and shown in Figure 4.7 for these queries:

PROJECT ***(Project#**, Project_Name, Chief_Architect)*
EMPLOYEE ***(Emp#**, EmpName)*
ASSIGNED_TO ***(Project#, Emp#)***

Furthermore, we use the domain variables $p_i \in$ Dom*(Project#)*, $n_i \in$ Dom*(Project_Name)*, $c_i \in$ Dom*(Chief_Architect)*, $e_i \in$ Dom*(Emp#)*, $m_i \in$ Dom*(EmpName)*, where Dom*(Project#)*, etc. are the domains of the corresponding attributes. The expression $<p_1,e_1> \in$ ASSIGNED_TO evaluates as true if and only if there exists a tuple in relation ASSIGNED_TO with the current value of the corresponding domain variables. As before we use the notation $\exists p_1,e_1(P)$ as shorthand for $\exists p_1 \exists e_1(P)$.

Example 4.52

The query "Compile the details of employees working on a Database project" can be stated as:

{e,m | $\exists p_1,e_1,p_2,n_2$ ($<p_1,e_1> \in$ ASSIGNED_TO
  $\wedge$ $<e,m> \in$ EMPLOYEE
  $\wedge$ $<p_2,n_2,c_2> \in$ PROJECT
  $\wedge$ $e_1 = e \wedge p_1 = p_2 \wedge n_2 =$ 'Database')} ■

Any number of domain variables can be defined on the domains of the attributes of a relation as illustrated below.

Example 4.53

Compile the details of employees working on both COMP353 and COMP354.

{e,m | $\exists p_1,e_1,p_2,e_2$ ($<e,m> \in$ EMPLOYEE
  $\wedge$ $<p_1,e_1> \in$ ASSIGNED_TO
  $\wedge$ $<p_2,e_2> \in$ ASSIGNED_TO
  $\wedge$ $e = e_1 \wedge e = e_2$
  $\wedge$ $p_1 =$ 'COMP353' $\wedge$ $p_2 =$ 'COMP354')} ■

The use of the universal quantifier and logical implication is demonstrated in Example 4.54.

Example 4.54 "Obtain the employee numbers of employees, other than employee 107, who work on at least all those projects that employee 107 works on."

$$\{e \mid <p,e> \in \text{ASSIGNED\_TO} \; \forall p_1,e_1 \; (<p_1,e_1> \in \text{ASSIGNED\_TO} \wedge e_1 = 107 \rightarrow (\exists p_2,e_2(<p_2,e_2> \in \text{ASSIGNED\_TO} \wedge e_2 \neq 107 \wedge p_1 = p_2 \wedge e = e_2))\}$$

An equivalent form of this query where the implication is replaced by the $\vee$ operator is given below:

$$\{e \mid <p,e> \in \text{ASSIGNED\_TO} \wedge \forall p_1,e_1(<p_1,e_1> \notin \text{ASSIGNED\_TO} \vee e_1 \neq 107 \vee (\exists p_2,e_2(<p_2,e_2> \in \text{ASSIGNED\_TO} \wedge e_2 \neq 107 \wedge p_1 = p_2 \wedge e = e_2))\} \; \blacksquare$$

Negation is illustrated in Example 4.55.

Example 4.55 "Get employee numbers of employees who do not work on the COMP453 project."

$$\{e \mid \exists p \; (<p,e> \in \text{ASSIGNED\_TO} \wedge \forall p_1,e_1 \; (<p_1,e_1> \notin \text{ASSIGNED\_TO} \vee p_1 \neq \text{COMP453} \vee e_1 \neq e))\} \; \blacksquare$$

Another example of the use of the universal qualifier and logical implication is given below.

Example 4.56 "What are the employee numbers of employees who work on all projects?"

$$\{e \mid \exists \; p(<p,e> \in \text{ASSIGNED\_TO} \wedge \forall p_1(<p_1,n_1,c_1> \in \text{PROJECT} \rightarrow <p_1,e> \in \text{ASSIGNED\_TO}))\} \; \blacksquare$$

The domain calculus formula to find employees who are assigned to at least one of a selected group of projects is given in Example 4.57.

Example 4.57 "Get the employee numbers of employees, other than employee 107, who work on at least one project that employee 107 works on."

$$\{e \mid \exists\, p,p_1,e_1,p_2,e_2(<p,e> \in \text{ASSIGNED\_TO} \wedge <p_1,e_1> \in \text{ASSIGNED\_TO} \wedge <p_2,e_2> \in \text{ASSIGNED\_TO} \wedge e_2 \neq 107 \wedge p_1 = p_2 \wedge e_1 = 107 \wedge e = e_2)\}$$ ■

4.5 Concluding Remarks on Data Manipulation

Consider tuple calculus expression:

$$\{x \mid x \notin R\}$$

Evaluating this expression generates tuples that are not in the relation R and entails generating an infinite number of tuples. If the domain of the tuple variable x were a relation scheme X, the tuples generated would be an indeterminate number of such tuples on the relation scheme X. However, in spite of this limitation, the number of tuples generated will be immense and the majority of these tuples are not likely to be in the actual database. In a database application an additional limitation is imposed: that all evaluating is done with respect to the content of the database at the time of the evaluation of the query. This further limitation generates, for the above expression, only those tuples that are in the database and not in the relation R. However, this evaluation is also prohibitively expensive in terms of computing resources used.

For relational calculus, by definition, infinite relations might be generated. In practice, this might be limited to finite relations because of condition imposed in the formula. It is therefore clear that the tuple relation calculus formulas are not only wffs, but they do not generate infinite relations. This in turn requires that the domain of the formula be clearly defined. The domain of a formula F(X), where X is a set of tuple variables, is the set of values either appearing explicitly in the formula or being referenced in it. The values that appear explicitly are constants and the values being referenced are from the relations appearing in the formula. Each such relation is assumed to be of finite cardinality. The purpose of defining the domain of a formula is to ensure that the result relation generated by evaluating the formula is also in the domain of the formula. This ensures that the result relation is finite and only tuples from the domain of the formula have to be examined in evaluating the expression. Such a tuple relational calculus expression is said to be **safe.**

The concept of safety can be applied to domain calculus expressions by defining a domain of a domain calculus expression and by ensuring that the result relation is within this domain. If we limit the relational calculus expressions to safe expressions, then tuple calculus and domain calculus are equivalent. Furthermore, both are equivalent to relational algebra. This means that for every safe relational calculus expression there exists a relational algebraic expression and vice versa. Also, we can write an equivalent domain calculus expression for a tuple calculus expression and vice versa.

Even though the final calculus expression for a query is more compact than an algebraic expression, it does not mean that calculus is a better interface, particularly with complex queries. It is natural to break such queries down into smaller steps (as in the case of the algebraic formulation; we presented a few examples of this in Section 4.3.3) and then compose the steps into a neat calculus formula. This may be the reason behind the success of SQL as a relational query language. SQL is clearly not assertional and includes intersection, union, and difference operations.

In Sections 4.3 and 4.4 we considered the features of relational data manipulation operations using relational algebra and relational calculus, respectively. The data manipulation language for the DBMS must supplement them with additional capabilities, such as relation creation, deletion, and modifications. Facilities are also provided for the insertion, deletion, and modification of tuples. These additional operations enables users to manipulate and update the data contained in the database. In the derivation operations, the attributes of one tuple are compared with attributes of another tuple or constants. In the alteration operations, the attribute values are altered or tuples are removed or inserted. As in the case of other relational operations, compatibility is also required in derivation and alteration operations.

A number of query languages based on the concepts of these sections have been developed. Three of these query languages (SQL, QUEL, and QBE) have gained wider acceptance than the others. SQL is in widespread use and, with an ANSI standard definition, has become the de facto query language for relational database systems. This in no way detracts from the elegance of QUEL. We consider all three languages in Chapter 5.

Relational Algebra vs. Relational Calculus

The relational algebra operations described in Section 4.3 allow the manipulation of relations and provide a means of formally expressing queries. The sequence of operations necessary to answer the query is also inherent in the relational algebraic expression. In other words, relational algebra is a procedural language. In Section 4.4 we considered two nonprocedural relational calculus query systems: tuple and domain calculus. In calculus queries we specify only the information required, not how it is obtained.

It can be proved that the expressive power of relational algebra and relational calculus are equivalent (Ullm 82). This means that any query that could be expressed in relational algebra could be expressed by formulas in relational calculus. Furthermore, any safe formula of relational calculus may be translated into a relational algebraic query.

There have been a number of proposed changes and additions to both relational algebra and calculus; for instance, the need for aggregation (average, count, and other such functions) and update operations in these query systems. Many researchers recognize this as omissions from the original formulation of relational algebra and calculus.

4.6 Physical Implementation Issues

So far, we have considered the relational model and the operations defined in the model. We have refrained from mentioning any implementation issues because, to the end user, these are of little concern. The relational algebra operations in some respects define what is to be done, but even then the DBMS can optimize the actual processing of the query and perform the operations in a different order (see Chapter

10 on query processing). In relational calculi we do not even specify the operations. To the users, the DBMS is a black box that insulates them from the details of file definitions and file management software as supported by the operating system. As we mentioned in Chapter 1, one function of the DBMS is to provide physical data independence.

The DBA cannot optimize the database for all possible query formulations. Thus, for every relation the anticipated volume of different types of queries, updates, and so on is estimated to come up with an anticipated usage pattern. Based on these statistics, decisions on physical organization are made. For example, it would be inappropriate to provide an access structure (say a B^+-tree) for every attribute of every relation; these secondary access structures have storage and search overheads.

The DBMS can make use of all the features of the file management system. As most DBMSs have versions that run on different machines and under different operating system environments, the DBMS may support file systems not available under the host machine environment. Thus, every DBMS defines the file and index structures it supports. The DBA chooses the most appropriate file organization. In the event of changes to usage patterns or to expedite the processing of certain queries, a reorganization can take place.

A large number of queries requires the joining of two relations. It may be appropriate to keep the joining tuples of the two relations either as linked records or physically grouped into a single record.

We may consider a relation to be implemented in terms of a single (or multiple) file(s) and a tuple of the relation to be a record (or collection of records). For the file, we may define a storage strategy, for example, sequential, indexed, or random, and for each attribute we can define additional access structures.

The more powerful DBMSs allow a great deal of implementation detail to be defined for the relations. The more common but less powerful DBMSs (mostly on microcomputers) allow very simple definitions, for example, indexing on certain attributes (this is usually a B^+-tree index). Some systems require the index to be regenerated after any modification to the indexing attribute values. Additional commands for sorting and other such operations are also supported. The typical file organization is plain sequential. (In fact, many micro-based DBMSs confuse a relation or table with a flat sequential file.)

A single relation may be stored in more than one file, i.e., some attributes in one, the rest in others. This is known as **fragmentation.** This may be done to improve the retrieval of certain attribute values; by reducing the size of the tuple in a given file more tuples can be fetched in a single physical access. The system associates the same internally generated identifier, called the **tuple identifier,** to the different fragments of each tuple. Based on these tuple identifiers a complete tuple is easy to reconstruct.

In addition to making use of the file system,[3] the DBMS must keep track of the details of each relation and its attribute defined in the database. All such information is kept in the directory. The directory can be implemented using a number of system-defined and -maintained relations. For each relation, the system may maintain a tuple in some system relation, recording the relation name, creator, date, size, storage

[3]To achieve satisfactory performance, many DBMSs develop their own file management systems and use disk input/output routines that directly access the secondary storage devices.