Chapter 3

The relational model and relational DBMSs

Essential reading

"An Introduction to Database Systems", sixth edition, by C. J. Date, published by Addison-Wesley, 1995, [ISBN 0-201-82458-2], *Chapter3, Chapter 4, Chapter 5 and Chapter 6* (pp. 52-178).

Alternatively

"Database Systems: A Practical Approach to Design, Implementation and Management", second edition, by T. Connolly and C. E. Begg, 1999, [ISBN 0-201-34287-1], *Chapter 3* (pp. 72 – 109).

Further reading

"Database Systems Concepts", second edition, by H. F. Korth and A. Silbershatz, published by McGraw-Hill, 1991, [ISBN 0-07-100804-7], *Chapter 3* (pp.53-95) — for a more concise and basic reading

"An Introduction to Database Systems", sixth edition, by C. J. Date, published by Addison-Wesley, 1995, [ISBN 0-201-82458-2], *Chapter 19 (Domains, relations and data types)* — for some more specialised issues on relational data objects — and *Chapter 18 (Optimisation)* — for some more specialised issues on optimisation.

The relational model - generalities

The relational model is a theory in which all data is modelled as *relations*; there are no pointers, no records, no other data structures, but only relations. This concept (relation) is formally introduced in a following section of the present chapter. Until then, we can assume that *relation* and *table* are synonyms. This is because tables are very well suited for graphically representing relations.

Employees

E_id	E_name	Salary	Dept_id
1	A. Smiths	25000	DEV
2	B. Johnson	28000	DEV
3	M. Fox	21500	DEV
4	B. Seller	20000	MA
5	C. Reade	23000	RES

Departments

Dept_id	Dept_name	Budget
DEV	Development	500000
MA	Marketing	100000
RES	Research	80000

Figure 1: Two relations illustrated as tables

For example, the information regarding the departments and the employees of an organisation can be modelled as two relations, named Employees and Departments, illustrated as tables as in Figure 1.

Two restrictive conditions are assumed by the relational model.

- All data values are *atomic* (scalar). A data value is atomic if its internal structure cannot be further decomposed into values of some more basic types. Relation 1 (see Figure 2) is incorrect according to the relational model, because the filed Children contains non-atomic values (a *set* of values, in this case). Relation 2, conversely, represents correctly the data from Relation 1.

Relation 1

Parent	Children
F. Bright	Lillian Mark Spencer
B. Johnson	Luke Baxter

Relation 2

Parent	Child
F. Bright	Lillian
F. Bright	Mark
F. Bright	Spencer
B. Johnson	Luke
B. Johnson	Baxter

Figure 2: Every relation must have atomic values

- The entire information is represented as *explicit* data values. This means that all relations[1] are defined *extensionally*. Suppose there exists a relation called "Parent" (see Figure 3).

Parent

Parent Name	Child Name
F. Bright	A. Hunter
F. Bright	M. Bright
A. Hunter	F. Spencer
M. Bright	J. M. Bright
F. Spencer	T. Spencer

Figure 3: A "Parent" relation

In order to define a base relation "Ancestor" (within the relational model, of course), meaning grand parent, grand-grand-parent, and so on, one has to *explicitly* provide the data values for this relation; the definition by means of a rule is not possible (see Figure 4).

POSSIBLE

Ancestor

Ancestor	Person
F. Bright	F. Spencer
F. Bright	J. M. Bright
F. Bright	T. Spencer
A. Hunter	T. Spencer

IMPOSSIBLE

A is **Ancestor** of B *IF* (there is an X such that)
A is **Parent** of X AND X is **Parent** of B

A is **Ancestor** of B *IF* (there is an X such that)
A is **Parent** of X AND X is **Ancestor** of B

Figure 4: Relations must be defined extensionally

There exists a mechanism by means of which rules can be used in defining relations, i.e. the *view* mechanism. However, in order to keep this introduction simple, we shall postpone this discussion until the section dedicated to this issue.

In order to define relations, domains are needed (a relation R is defined over a set of domains, as in (formally): R : D1 x D2 x … x Dn). Since relations are the only data structures used, it follows that the only data objects needed in the relational model are *domains* and *relations*. The second section of this chapter looks at *relational data objects*.

[1] All *base* relations, to be rigorous (base relations are defined later in this chapter).

Data objects need to be operated upon; the relational model provides a set of *operators* for data manipulation. Two main approaches exist with respect to relational operators:

- declarative, represented by *relational calculus*;
- procedural, represented by *relational algebra.*

In this chapter, we look only at relational algebra operators. The reason for this choice stems from the fact that the most used database language, SQL, implements relational algebra operators.

Relational algebra operators are global (more formally, *set at a time*). This means that operators are applied to relations and result in relations.

RESTRICT **Employees**
SUCH THAT Salary > 22000

E_id	**E_name**	**Salary**	**Dept_id**
1	A. Smiths	25000	DEV
2	B. Johnson	28000	DEV
5	C. Reade	23000	RES

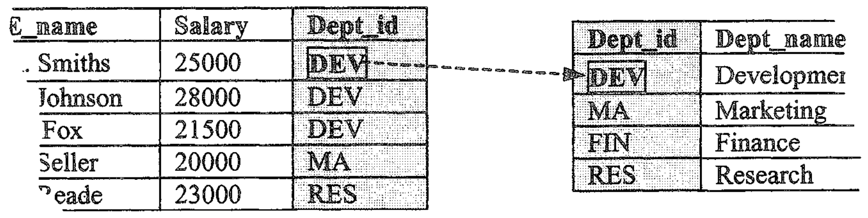
Figure 5: The result of the application of a relational operator (RESTRICT) to the relation in Figure 1

For instance, if we need to select all employees (refer to Employees in Figure 1) who earn more than 22000 then a *restrict*[1] operator can be used in the following manner[2]:

```
RESTRICT Employees SUCH THAT Salary > 22000 ;
```

The result of this expression is the relation depicted in Figure 5.

The information relevant to a real life system, is modelled, within the relational model, by means of explicitly defined relations. Since there are no pointers to connect one relation to another, how are "related" relations linked to each other? The answer is: by means of *corresponding fields*, which, on their turn, are implemented via *keys*. The diagram in Figure 6 illustrates this concept (the linked relations were only partially shown since we are only interested in the corresponding fields). Note that the links are implemented purely in terms of data values.



E_name	**Salary**	**Dept_id**
. Smiths	25000	DEV
Johnson	28000	DEV
Fox	21500	DEV
Seller	20000	MA
eade	23000	RES

Dept_id	**Dept_name**
DEV	Developmer
MA	Marketing
FIN	Finance
RES	Research

Figure 6: Corresponding fields (keys) linking tables

The data for a certain real life application, usually, is constrained to a set of *valid* values. Or, in other words, there are certain limitations that data must comply with in order to be *correct* (with respect to its meaning). For instance, in the "Employee, Department" example (Figure 1), a negative salary is not a valid data, nor is "12$00%&xx" a valid name.

Therefore, when devising a relational model, it is not sufficient to define only the structure of data. Certain limitations, that constrain data to correct values, must also be

[1] This operator is described in the section "Relational operators "of this chapter.

[2] This is an ad-hoc notation, but similar to relational algebra. It is used only for illustration purposes. It is very close to natural language so you should have no problems in understanding it. Its syntax is not relevant. You should use it merely as an intermediary tool, until relational algebra is introduced.

defined. These are called *integrity constraints*. We shall look at two generic (applicable to any relational model) integrity constraints, viz., the entity and the referential integrity constraint.

In conclusion, the relational model can be defined as a way of looking at data, as a prescription for:

- *data representation* (relational data objects);
- *data manipulation* (relational operators);
- *integrity constraints representation* (relational data integrity).

The rest of this chapter covers all these issues in detail.

You might ask the question: "Why do we need to study this theory?". The answer is simple: "One cannot be a good practitioner without having a thorough understanding of the theory". Moreover, a good understanding of the theoretical model ensures a quick understanding of any of its implementations.

Relational DBMSs

The relational model is "merely" a theory. Yet, every relational database management system (DBMS) implements this theory and, at the moment, the huge majority of all extant DBMSs are relational.

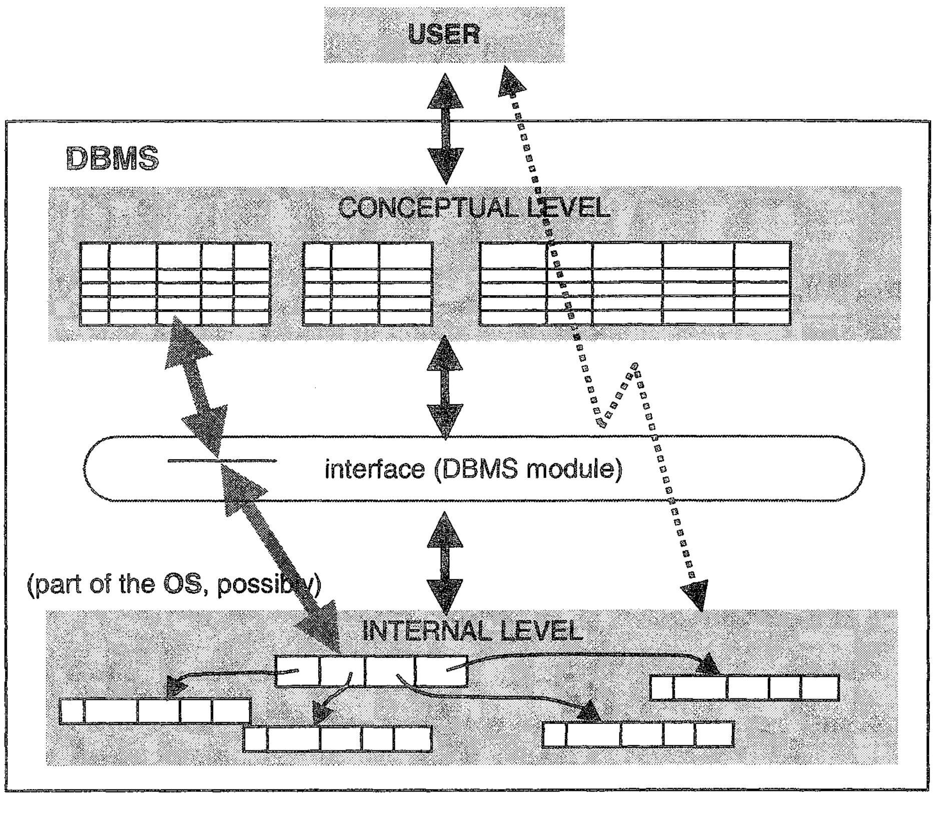


Figure 7: A relational DBMS

A *relational* DBMS is a database management system system that implements the relational model. It provides support for defining data (its structure and integrity constraints upon it) and for manipulating it according to the relational model. Therefore, a relational DMBS shields the users from the physical details of data implementation, allowing them to operate with data as exclusively relational data objects (i.e. entirely at the logical level).

However, most of the extant relational DBMSs do not force (they only allow) the user to perceive data strictly at the logical level. For of efficiency reasons, they also allow access to physical details of data implementation (see Figure 7). Another common characteristic of all relational DBMSs is that they do not support all aspects of the relational model, they implement only parts of the theory.

Relational data objects - domains and relations

Terminology

- Until its formal definition, a *relation* is going to be understood as a table.
- Each *attribute* of a relation is represented as a column in its corresponding table. *Filed* is another term used for attribute.
- Each *tuple* of the relation is represented by a row in its corresponding table. *Record* is another term used for tuple.
- The number of attributes represent the *degree* of the relation. The number of tuples represent the *cardinality* of the relation.
- The formal terms, therefore recommended to be used by you, are: *relation, attribute, tuple, degree and cardinality.*

Now, we can embark on the study of the relational model in detail.

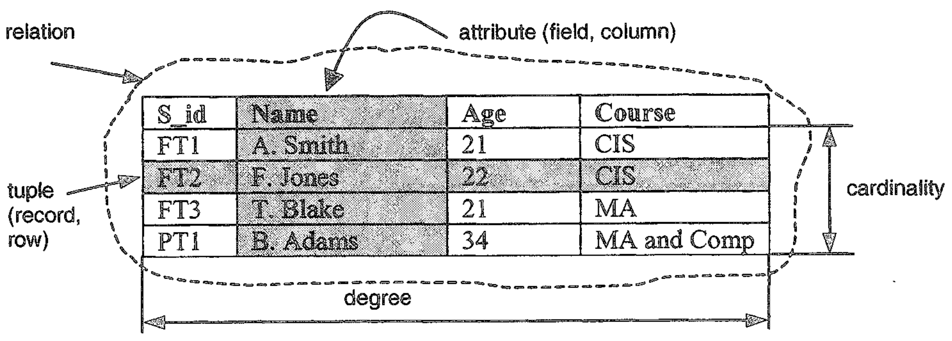

relation
attribute (field, column)
tuple (record, row)
cardinality
degree
S_id | Name | Age | Course
FT1 | A. Smith | 21 | CIS
FT2 | F. Jones | 22 | CIS
FT3 | T. Blake | 21 | MA
PT1 | B. Adams | 34 | MA and Comp

Figure 8: Terminology

Domains

Informally, a domain is a "pool of values"(Date 1995, p. 79) from which a certain attribute of a specific relation can draw its actual values; every attribute of a specific relation must be defined on (can have values from) exactly one domain. More precisely, a domain is *a set of values of the same type.*

For instance, the diagram in Figure 9 depicts possible domains corresponding to the Students relations (Figure 8).

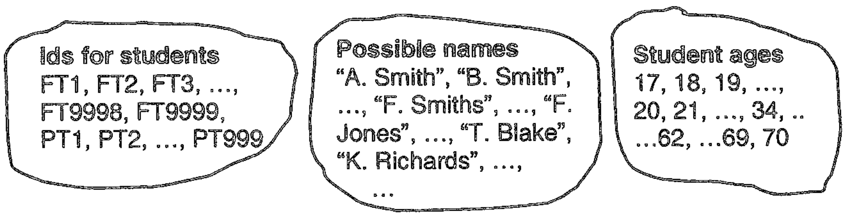

Ids for students
FT1, FT2, FT3, ..., FT9998, FT9999, PT1, PT2, ..., PT999
Possible names
"A. Smith", "B. Smith", ..., "F. Smiths", ..., "F. Jones", ..., "T. Blake", "K. Richards", ..., ...
Student ages
17, 18, 19, ..., 20, 21, ..., 34, ...62, ...69, 70

Figure 9: Domains

The domain "Ids for students", corresponding to the attribute S_id, is represented by the set of all possible identifiers for students, i.e. 9999 values for full time (FT) and 999 values for part time (PT). The whole set of values of this domain is therefore {FT1, FT2, ..., FT9999} UNION {PT1, PT2, ..., PT999}. Each tuple in Students can only have one of these values for the attribute S_id. The domain "Possible Names", associated with the attribute "Names", is represented by the set of the names of all possible students. The domain "Student ages" represents the set of integers from 17 to 70, etc.

One big advantage offered by domains is that they prevent certain meaningless operations from being performed. Suppose, for instance, a relation that comprises information about some products, as in Figure 10.

ID	**Name**	**Weight**	**Length**	**Width**	...	...

Figure 10: A relation describing products

It is perfectly reasonable to think of queries like "which are the products heavier than the average weight", "which are the products with a square shape" or "which are the products with an area less than 1m^2"", that involve comparisons between the weight of each product and an average weight, the length and the width of each product or a multiplication between the length and the width of each product, respectively.

However, it will make no sense to attempt a comparison between the weight and the length of a product or a multiplication between weight and length. Domains represent a means to avoid such situations. This achieved, within the relational model, by restricting the applicability of both scalar and relational operators. We say that *domains constrain operations* (therefore we can refer to them as *domain-constrained operations*). A scalar operator is only allowed to be applied to values of the domain it is defined on. Loosely speaking, a relational operator is allowed to be applied to two relations only if the *attribute operations* it subsumes are not applied to attributes defined on different domains (the attribute operations involve scalar operators). This is illustrated by the following example.

A *JOIN* operator represents a means of combining two relations (it is described in the "Relational operators" section) based on a common attribute. The scalar operator associated with *JOIN* is *equality*. Suppose two relations R1 and R2. If R1 has an attribute A1 with the same name and defined on the same domain as an attribute of R2, then the two relations can be joined. The attribute denoted by A1 is called a common attribute. If R1 and R2 have no common attributes then they cannot be joined. I.e., domains constrain the applicability of the *JOIN* operator.

If the DBMS supports domains, then the system will prevent the user from making mistakes such as meaningless comparisons. For the above example, consider the existence of two domains, "Dimensions" and "Weight". By defining the attributes "Length" and "Weight" on "Dimensions" and the attribute "Weight" on the domain "Weight", we *inform* the system about the *meaning* of these attributes, thus the system would be able to enforce domain-constrained operations.

Before providing a more formal definition for domains, the concept of *atomic* or *scalar* value must be introduced.

Definition: *An* atomic *or* scalar *data item is the smallest semantic unit of data, i.e. a data item that has no internal structure as far as the model or the DBMS is concerned.*

As of above, FT1, FT3, PT12, "T. Blake", 21 are all examples of atomic data values. Note that "no internal structure with respect to the model or the DMBS" does not mean that the respective data value has no internal structure at all. For instance, a name, say "T. Bake", can be viewed as consisting of an initial ("T", in our example) and a surname ("Blake"); furthermore, the surname can be viewed as a string of characters ('B', 'l', 'a', 'k', 'e'). All it means is that as we *understand* or *view* it, atomic data is not decomposable in smaller units.

As far as the *theoretical model is concerned*, data of any complexity, if viewed with no internal structure, can be considered as atomic. It is our understanding, view or assumption, for a particular application, according to which atomic data is defined. If a data type T was considered to be scalar, then there is no way of accessing (no operator that provides access to) the constituents of data values of type T. For instance, if a name of a person is considered scalar, "Martin T. Blake", then there no way of accessing its constituents (e.g. surname, "Blake" and first names, "Martin T.").

Definition: A domain is a named set of scalar values of the same type.

For a better understanding, you can view domains as being equivalent to data types in a programming language (Date 1995). As far as a DBMS is concerned, domains are not explicitly stored within the system; they are specified as part of the database definition (in the system catalogue).

Unfortunately, almost all of today's relational DBMSs provide very limited support for domains; all they supply is a set of basic data types (integer, real, string, ...), referred to as *built in* or *system defined*. What is missing is support for *user defined* domains.

On the one hand, the set of basic data types provided by relational DBMSs is limited. On the other hand, users cannot, define new data types, apart from those provided by the system. As a result, relational DBMSs are found unsuitable for many application areas (such as Geographical Information Systems and Multimedia Databases).

A domain is not just a named set of values, but it also comprises a set of *operators* that are applicable to these values. For instance, integer values can be added, subtracted, compared, etc. If a certain domain is missing from a DBMS then data values of that kind cannot be manipulated by the system. For instance, current relational DBMSs do not provide support for graphical representations (needed in Geographical Information Systems, for instance). That is, the system cannot deal with queries of the kind "identify all the buildings within a certain area", because a "within" operator does not exist, nor can it be defined. So, another big advantage offered by domains is an increased representational power. It is common-sense that the modelling capabilities increase by having a richer set of scalar values.

Domains are a powerful mechanism provided by the relational model. Date (Date 1995) claims that some drawbacks are associated with the relational model when, in fact, they are caused by DBMSs, due to *incomplete implementations* of the relational model. Therefore, the solution to these drawbacks is to build proper support for domains in relational DBMSs.

To conclude, the main two advantages that we identified, related to domains, were:

- domain-constrained operations;
- increased representational power (which somehow subsumes the above advantage).

There are further aspects related to domains, but, for the time being, this presentation suffices. It is important that you understand why domains are needed and the similarity between domains and data types.

Relations

The *relation* is the only data structure used for modelling data in the relational model. Thus far, we viewed a relation as being equivalent to a table. It is possible now to properly (formally) define the *relation* concept.

Definition: A relation, *R, on a set of domains D1, ..., Dn, consists of two parts, a* head *and a* body.

- *the* head *consists of a fixed set of attributes, described by <attribute-name : domain-name> pairs, {<A1 : D1>, <A2 : D2>, ..., <An : Dn>}; each attribute Ai corresponds to exactly one of the underlying domains Di; the attribute names are all distinct.*
- *the* body *consists of a set of tuples; each tuple is a set of <attribute-name : attribute-value> pairs; in each tuple i, there is exactly one <Aj : vij> pair for each attribute Aj in the heading; e.g. the i-th tuple of a relation whose attributes are A1, ..., An, can be represented as {<A1 : vi1>, <A2 : vi2>, ..., <An : vin>}; for any given <Aj : vij> pair, vij is a unique value from the domain Dj that is associated with the attribute Aj*

The domains D1, ..., Dn need not be all distinct.

The definition is taken from (Date 1995, p.86-87).

This seems a rather intricate definition, but, in fact, it is quite simple. Let's consider two examples to illustrate it. Firstly, a correct example of a relation. Consider the table "Students" in Figure 11.

Id	Name	Age	Course
FT001	A. Smith	21	CIS
FT002	F. Jones	22	CIS
FT003	T. Blake	21	MAS
PT001	B. Adams	34	MAC

Figure 11: The table "Students"

The relation it illustrates can be represented as in Figure 12. The attributes of the relation are: "Id" (the Id number, unique for each student), "Name", "Age" and "Course" (having the obvious meaning). The heading of the relation is represented as a set of pairs <name : type>: "Id" is a string of length 5, "Name" a string of variable length, "Age" is an integer and "Course" is a string of length 3.

```
Students relation
Heading
   {<Id : CHAR(5)>, <Name : VAR-CHAR>, <Age : INT>, <Course : CHAR(3)>}
Body
  {
   {<Id : "FT001">, <Name : "A. Smiths">, <Age : 21>, <Course : "CIS">},
   {<Id : "FT002">, <Name : "F. Jones">, <Age : 22>, <Course : "CIS">},
   {<Id : "FT003">, <Name : "T. Blake">, <Age : 21>, <Course : "MAS">},
   {<Id : "PT001">, <Name : "B. Adams">, <Age : 34>, <Course : "MAC">}
  }
```

Figure 12: The relation "Students"

Note that the order of the pairs in each set, or the order the tuples in the body does not matter. Therefore, the representation of Students in Figure 13 is also correct.

```
Students relation
Heading
   {<Id : CHAR(5)>, <Name : VAR-CHAR>, <Course : CHAR(3)>, <Age : INT>}
Body
  {
   {<Id : "FT001">, <Name : "A. Smiths">, <Age : 21>, <Course : "CIS">},
   {<Course : "MAS">, <Id : "FT003">, <Name : "T. Blake">, <Age : 21>},
   {<Age : 34>, <Id : "PT001">, <Name : "B. Adams">, <Course : "MAC">},
   {<Name : "F. Jones">, <Id : "FT002">, <Age : 22>, <Course : "CIS">}
  }
```

Figure 13: The relation "Students" – another *correct* representation

Activity: The two representations of the Students relation are equivalent. Can you think of another equivalent representation, but different from the above two?

To make thinks even clearer we are going to look at an incorrect representation of the relation "Students" (Figure 14).

```
Students relation
Heading
   {<CHAR(5)>, <Name : VAR-CHAR>, <Course>, <Age : INT>}
Body
  {
   {<"FT001">, <Name : "A. Smiths">, <Age : 21>, <Course : "CIS">},
   {<Course : "MAS">, <Id : "FT003">, <Name : "T. Blake">},
   {<Age : 34>, <Id : "PT001">, <Name : "B. Adams">, <Course>},
   {<Name : "F. Jones">, <Id : "FT002">, <Age : 22>, <Course : "CIS">}
  }
```

Figure 14: The relation "Students" – an *incorrect* representation

The mistakes are: in the heading, (1) the first pair has no attribute name and (2) the third pair has no type declaration; in the body, (3) the first pair of the first tuple has no attribute name, (2) the second tuple has an element missing, namely the one corresponding to "Age" and (3) the third tuple has no value specified for Course.

Relations have some important properties that can be derived from the definition:

A relation cannot contain duplicate tuples or, in other words, each tuple is unique within a given relation, because the body is a set of tuples.

- A corollary of this property is that each tuple, within a given relation, is *uniquely identifiable*; any set of attributes that can be used to uniquely identify each of its tuples is called a *candidate key*[1].

Tuples are unordered, therefore statements such as "the fifth tuple" make no sense.

Attributes are unordered, therefore, statements such as "the first attribute" make no sense.
It is apparent now why tables and relations are not one and the same thing:

- the rows of a table are ordered;
- the columns of a table are ordered;
- a table can contain duplicate rows;
- a table can contain duplicate columns.

All attribute values are atomic. This property is called *first normal form*. Every relation is, by definition, in the first normal form. We shall illustrate, via an example, why relations have to be in the first normal form.

Consider the two relations (represented as tables) of Figure 15 and the following two tasks:

- indicate that a new tutor, P. Rosin, was assigned a tutee, P. Black, of age 26;
- indicate that the tutor, M. Taylor, was assigned a new tutee, M. Jackson of age 23.

Unnormalised

Tutor	**Tutee**	
M. Taylor	**Name**	**Age**
	A. Braun	22
	T. Elliot	21
	Y. Dhillon	22
P. Collins	**Name**	**Age**
	F. Reed	19
	P. Fox	22
	M. Baxter	24

Normal_1

Tutor_name	**St_name**	**St_age**
M. Taylor	A. Braun	22
M. Taylor	T. Elliot	21
M. Taylor	Y. Dhillon	22
P. Collins	F. Reed	19
P. Collins	P. Fox	22
P. Collins	M. Baxter	24

Figure 15: Un-normalised and normalised (1NF) versions of the same relations

With respect to the relation "Normal_1", each task is reflected by the insertion of a new tuple. So, the two tasks are similar as far as "Normal _1" is concerned.

[1] This issue will be discussed into more depth in the "Relational data integrity" section.

However, with respect to the relation "Unnormalised", the two tasks are (qualitatively) different. The first one only involves the insertion of a new tuple. The second one involves the insertion of tuple
{<Tutor : M. Taylor>, <Tutee : {<Name : M. Jackson>, <Age : 23>}>}
that has a non-atomic value for the Tutee attribute – namely the tuple
{<Name : M. Jackson>, <Age : 23>}.
This value, in turn, requires a tuple insertion operations. The second task requires a "tuple-insert operation that performs a tuple insert operation".

Therefore, if the relation is not normalised, at least two insert operations will be needed. The same would apply to delete operations, update operations ... The conclusion is that the DBMS is much simpler if all relations are considered to be normalised (first normal form).

At this point a further clarification must be made, namely the distinction between a relation *value* and a relation *variable*. In essence, this is similar to the distinction between a value and a variable in a programming language. In the example of Figure 16, "example_var" is a *variable* of type integer, which means that it can hold any *value* of type integer.

```
/* this is a fragment of a C program illustrating values and variables */
int example_var;
example_var = 10;                       /* example_var is assigned the integer value 10 */
example_var = example_var * 2;          /* the value of example_var is doubled */
/* etc. */
```

Figure 16: Values and variables in a programming language

Recall now the definition of a relation. Strictly speaking, it defines a *relation value*, because the body is defined as being constant in time.

The definition of a *relation variable* is very similar to the definition of a relation value. The only difference is the fact that the former allows for the body of a relation variable to vary in time. The head, however, remains always the same. In other words, tuples can be inserted and / or deleted from a relation variable, but they have to have the same structure. We conclude this point by comparing relation variables[1] with domains

- relation variables are dynamic, as opposed to domains which are static; in other words, the content of a relation variable (i.e. its body) changes over time, whereas the content of a domain does not.

It might appear that the relations in a relational database are independent, in that there is no way to relate one to another (because thy only contain explicit data vales). As a matter of fact, this is not true; the relations in a database are logically connected by means of corresponding attributes, called *keys*[2]. In one of the relations they key is called *candidate key* and in the other relation (linked to the former) the key is called *foreign key*. In the example of Figure 1 the relation "Employees" is linked to the relation "Departments" via the "common" attribute Dept_id. If one wants to find out the details of the department in which a specific employee works then one will identify the Dept_id of the respective person from Employees and then look it up in the relation Departments. For instance, A. Smiths works in the department identified by "DEV" (from the Employees relation). The department identified by "DEV" represents the Development department, having a budget of £500,00 (from the Departments relation). Dept_id in Employees is a foreign key whereas in Departments is a candidate key.

The discussion, so far, was carried out at an abstract level (i.e. the theoretical model). What is the link between the theoretical model and a relational database?

[1] From this point onwards we are going to use the term "relation" meaning either "relation variable" or "relation constant". The actual meaning will result from the context.

[2] This concept is properly described in the "Relational data integrity" section.

Data definition in a relational DBMS

As originally stated by Codd (Codd, 1970) the definition of a relational database is:

***Definition:** "A relational database is a database that is perceived by the user as a collection of normalised relations of assorted degrees"*

From what you have studied so far, you can deduce that a relational DBMS must provide a DDL that supports at least the definition of domains and relations (i.e. relation variables). SQL is a relational database language that provides such a component. A very small subset of SQL's data definition language is illustrated below. The purpose is just to give you the gist of a relational database language. The next chapter is entirely dedicated to SQL.

In defining the syntax of the language, we shall follow the following notation conventions:

- everything that is not within the "<" and ">" signs (e.g. the words in upper case letters) is a *terminal* or a keyword symbol, which means that it has to be left unchanged
- phrases within the "<" and ">" signs are *non-terminals*, which means that they have to be replaced with a corresponding value (e.g. <domain name> is to be replaced with a user defined name for a domain)

The restricted syntax for SQL's data definition language is[1]:

```
CREATE DOMAIN <domain name> AS <definition> ;
CREATE TABLE <table name> (
     <attribute name>          <domain name>,
     --
     <attribute name>          <domain name>
);
```

For instance, the definition of the relation Students used above could be:

```
CREATE DOMAIN   IdDomain          AS    CHAR(6) ;
CREATE DOMAIN   NamesDomain       AS    VARCHAR ;
CREATE DOMAIN   AgeDomain         AS    INT ;
CREATE DOMAIN   CoursesDomain     AS    CHAR(3) ;
CREATE TABLE Students (
     Id              IdDomain,
     Name            NamesDomain;
     Age             AgeDomain,
     Course          CoursesDomain
);
```

A relation, once defined, would have to be "populated" with data. Therefore, support for at least two more statements have to be provided by the DBMS, this time by the DML component, namely for inserting new tuples and deleting old tuples

A restricted syntax for SQL's INSERT and DELETE statements is

```
INSERT INTO <table> VALUES <row>;
DELETE FROM <relation> WHERE <condition for identifying tuple>;
```

Examples of their use on the Students relation:

```
INSERT INTO       Students
     VALUES       ("FT0001", "A. Smiths", 21, "CIS"}
INSERT INTO       Students
     VALUES       ("FT0002", "F. Jones", 22, "CIS"}
DELETE FROM       Students
     WHERE        IdDomain = "FT0001" ;
```

[1] SQL uses tables to implement relations.

It is possible that certain relations or domains might not be needed any longer in the database, therefore there has to be a mechanism for removing them.

In SQL these statements have the following (reduced) syntax:

```
DROP DOMAIN <domain name> ;
DROP TABLE <relation name> ;
```

An important characteristic of a relational DBMS is that both the external and the conceptual level are based on the same data model, i.e. *data is perceived at both levels as relations*.

Activity: What are the implications of this characteristic? (to answer this question, recall the requirements for DDL and DML imposed by the compliance with the three level ANSI/SPARC architecture)

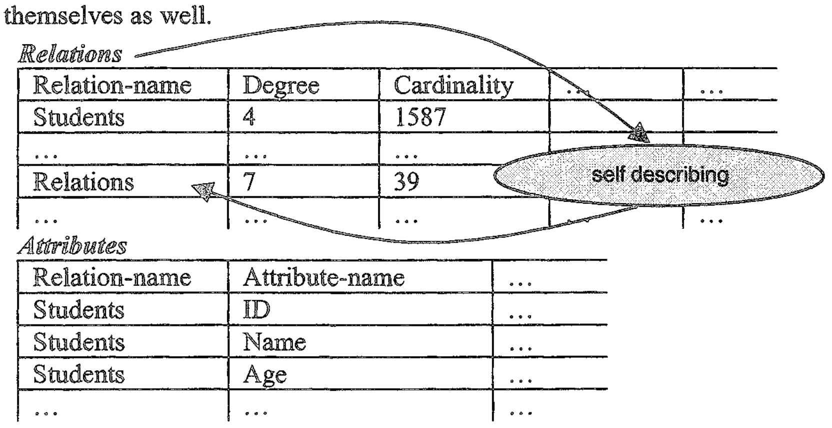
As a concluding issue we are going to have a look at a classification of relations that might exist in a relational system (you should return to this classification after you studied the next section, Relational operators, and the next chapter, SQL):

- *named* relation - a relation that has been defined for a certain database system;
- *base* relation - a relation that was defined purely in terms of its extension, i.e. for which all data values are explicitly provided; it has an independent existence;
- *derived* relation - defined in terms of other relations - either base or derived - by means of some relational expression;
- *view* - named derived relation whose extension is not stored in the system; a view can thus be considered as a virtual relation;
- *snapshot* - named derived relation but whose extension was computed and stored in the system;
- *query result* - unnamed derived relation who has no persistent existence within the system.

The data dictionary

As described in the previous chapter, the database includes the description of its raw data in what is called the data dictionary (or catalogue; the American spelling, "catalog", is sometimes used instead). The data dictionary contains all kinds of information describing the database: schemas, mappings, integrity rules, security rules, etc. It is part of the database, therefore it is represented by means of relations (tables). They are called system relations (tables), in order to be differentiated from the user-defined relations (tables). The information stored in the data dictionary is useful to some of the modules of a relational DBMS. However, the data dictionary can be used in the same way as any other part of the database.

For example, commonly, every data dictionary (i.e. in every DBMS) contains two relations describing all the (named) relations in the database, namely *Relations (or Tables)* and *Attributes (or Columns)* (Figure 17). Note that they contain information about themselves as well.



Relations

Relation-name	Degree	Cardinality	...	...
Students	4	1587		
...	...	...		
Relations	7	39		
...	...	...		...

Attributes

Relation-name	Attribute-name	...
Students	ID	...
Students	Name	...
Students	Age	...
...	...	...

Figure 17: A part of a data dictionary

Relational operators

We have seen how the structure of data can be defined. Subsequently, this section presents *relational operators,* which allow data to be manipulated. There are two main approaches to relational operators:

- *procedural* – a set of operators by means of which we can prescribe *how* a result is going to be obtained;
- *declarative* – a set of operators by means of which we can prescribe *what* result is to be obtained.

In the first approach we have to state *which operations* are to be performed on data and *the order* in which they are to be performed. So, we have to describe *how* the result is to be computed.

In the second approach we do not make any reference to the way the result is to be computed. We only describe *what the result looks like.* We do this by stating the *conditions* that the result should satisfy.

If we refer to implementations, in the procedural approach we "tell" the system what operations to perform. In the declarative approach, we "tell" the system only what result we are interested in. It is the system's responsibility to work how to compute the result, i.e. what operations to perform.

The relational model includes:

- *relational algebra* – corresponding to the procedural approach;
- *relational calculus* – corresponding to the declarative approach.

The two approaches are, in fact, equivalent to one another, in that any expression of the relational algebra has an equivalent expression in relational calculus and the other way around. The two formalisms are different only in the style of expression.

The most popular database language, SQL, is based on relational algebra. Therefore, the rest of this section is dedicated to relational algebra[1].

Relational algebra consists of 8 basic operators[2]:

- 4 borrowed from the set theory, namely: *union, intersection, difference* and *Cartesian product*;
- 4 relational specific operators, namely: *restrict, project, join* and *divide.*

Before presenting each of these operators, two important properties of relational algebra have to be stated:

- *set at a time* operators;
- *relational closure.*

The first property refers to the fact that relational algebra operators act *globally* on relations, i.e. relations *as a whole* constitute their operands and not individual tuples.

The second property - relational closure - refers to the fact that the result of the application of any relational operator is a relation, too. This means that relational operators can be used in devising complex expressions, since the result of the application of one operator can become the input for another operator. As a parallel, think of real numbers algebra. Consider only three operators, namely +, -, and *; they are applied to real numbers and they result in real numbers. Consider that x, y, z, u, v, w are real number variables. We can construct expressions of any kind of complexity, e.g.

```
((x*x + y*z) – 2*x*z*y)
```

[1] If you are interested in finding more about relational calculus, refer to (Date 1995), Chapter 7 (pp. 185-218).

[2] As defined in (Codd, 1972).

```
(2*x + 3*y + 4*z) – (u+v)
```

Because of the closure property, these two expression can be used in other expression (i.e. they can become the input of some operators), e.g.:

```
((x*x + y*z) – 2*x*z*y) + ((2*x + 3*y + 4*z) – (u+v))*2 – y*z
```

Relational algebra expressions are constructed in a similar way. A relational algebra expression denotes a relation, therefore it can constitute, as a whole, the operand of any relational algebra operator.

These two characteristics of relational algebra, set at a time and relational closure, conduce to a very *powerful formalism.*

"Set specific" relational algebra operators

The set specific relational operators are almost identical to the set operators from Set Theory. Relations are a special kind of sets. Therefore, the set operators have to be specialised (restricted) for relations, in order to ensure relational closure.

- They assume type compatibility of relations (apart from the Cartesian product).
- They are accompanied by an attribute name inheritance mechanism.
- They are accompanied by a candidate key[1] inheritance mechanism.

Type compatibility of relations, means, intuitively, that the respective relations have to have the same shape. More formally,

Definition: *Two (or more) relations are type compatible if they have identical headings, i.e.:*
- *they have the same set of attribute names;*
- *the corresponding attributes are defined on the same domain.*

Relation1

ID	Name	Age	City
S1	A. Braun	22	London
S2	T. Elliot	21	London
S3	Y. Dhillon	22	Brighton

Relation2

ID	Name	Age	City
S1	A. Braun	22	London
S4	F. Reed	19	Leeds

Figure 18: Two type compatible relations used for illustrating the set operators

Union, intersection and difference can only be applied to type-compatible relations, and the result of their application is a relation type compatible with the operands.

Figure 21, Figure 20 and Figure 21 illustrate the application of set specific union, intersection and difference on the two relations of Figure 18.

UNION
Relation1 ∪ Relation2

ID	Name	Age	City
S1	A. Braun	22	London
S2	T. Elliot	21	London
S3	Y. Dhillon	22	Brighton
S4	F. Reed	19	Leeds

Figure 19: Set specific *union*

[1] The concept of candidate key is defined in the next section.

INTERSECTION
Relation1 ∩ Relation2

ID	Name	Age	City
S1	A. Braun	22	London

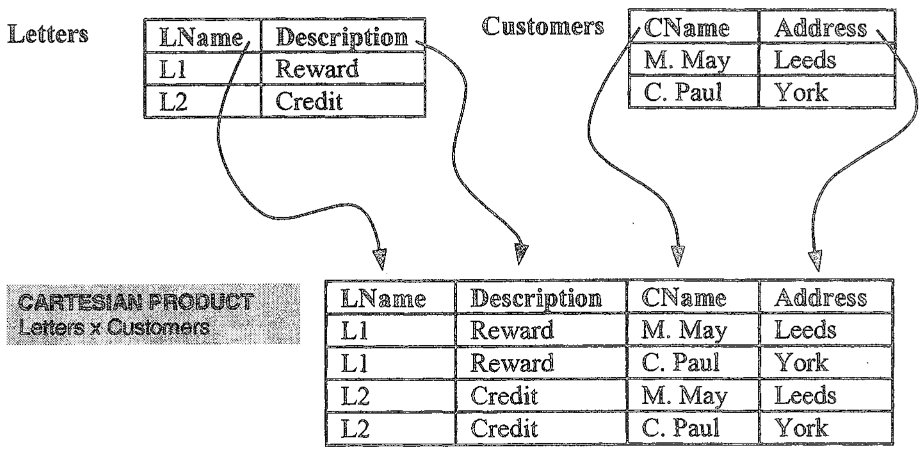
Figure 20: Set specific *intersection*

DIFFERENCE
Relation1 — Relation2

ID	Name	Age	City
S2	T. Elliot	21	London
S3	Y. Dhillon	22	Brighton

Figure 21: Set specific *difference*

For the Cartesian product, the type compatibility restriction is not necessary. The Cartesian product may be applied to any two relations. An example of the application of the Cartesian product operator is provided in Figure 22. Two "small" relations were chosen in order to keep the diagram to a reasonable size.



Letters

LName	Description
L1	Reward
L2	Credit

Customers

CName	Address
M. May	Leeds
C. Paul	York

CARTESIAN PRODUCT
Letters x Customers

LName	Description	CName	Address
L1	Reward	M. May	Leeds
L1	Reward	C. Paul	York
L2	Credit	M. May	Leeds
L2	Credit	C. Paul	York

Figure 22: Illustration of the *Cartesian product* operator

The attribute name inheritance mechanism does apply, and is well illustrated by the above example: the result relation inherits the name of its attributes from the Letters relations – LName and Description - and the Customers relation– CName and Address.

Activity: The candidate key inheritance mechanism is left to be investigated by you after you learn what a candidate key means (next section).

"Relation specific" relational algebra operators

The other 4 basic relational algebra operators are: *restriction, projection, join and division.*

Products-Details

PruductID	ProductName	Cost	InStock	Supplier
PID23	Washing machine	289	2	E-A Inc
XX24A	Dish dryer	399	0	H200
00012	Power supply extension	14.99	15	E-A Inc
MM25y	Television set	395	0	S-TV
MM45x	Television set	555	0	S-TV

Figure 23: Example relation

Restriction

We shall start with the definition.

***Definition:** A restriction operator is applied to a single relation R – i.e. is a unary operator – and produces another relation R' according to a condition C so that*

- *R' is type compatible with R;*
- *all the tuples of R' are from R;*
- *all the tuples of R' satisfy C;*
- *there are no tuples in R that satisfy C and are not in R'.*

The condition C is expressed on one of the attributes of R by means of a some scalar comparison operators. For instance, the relation Products-Details (Figure 23), describing all the products that can be supplied by a company, can be *restricted* to only those products that are available, which translates into the InStock attribute being greater than zero; the result is the Available-Prod relation (Figure 24).

Available-Prod

PruductID	ProductName	Cost	InStock	SupplyCity
PID23	Washing machine	289	2	Coventry
00012	Power supply extension	14.99	15	Coventry

Figure 24: Available products, i.e. products that are in stock

The syntax for the restriction operator is[1]:

```
<relation name> WHERE <condition>
```

where <condition> is a conditional (or truth valued) expression on the attributes of the relation <relation name>. For instance, the relation Available-Prod is defined by

```
Products-Details WHERE InStock > 0
```

<condition> can be a truth valued expression of any complexity. For instance, the restriction of Products-Details to those products whose total value in stock is over £ 500, translates into:

```
Products-Details WHERE Cost*InStock > 500
```

The conditional expression in this case involves a multiplication between a currency type value and an integer value (the result being of currency type) and then a comparison with a currency type value.

You might have realised by now that the conditional expressions you can use are determined by the domains that are given or that you have chosen, which, in turn, determine the available scalar operators. For instance, the *integer* type includes the following operators: +, -, *, div (integer division), and comparison operators such as >, <, =. The *string* type can provide operators such as: *concatenation* (of two strings), *division* (of one string into two sub-strings) and comparison operators such as *inclusion* (of a string in another) and *equality* (of two sub-strings).

It is needless to say that a scalar operator can only be applied to values corresponding to the domain it is defined on. E.g. the multiplication operator, *, can be applied to values of type integer or real, but cannot be applied, for instance, to string values.

According to the definition, the restriction operator can only use atomic conditions. An atomic condition is expressed on one single attribute of a relation, by means of one single scalar comparison operator. Therefore, if the restriction criterion implies several atomic conditions, then the corresponding expression consists of several nested restriction-expressions. For instance, to perform a restriction based on two atomic conditions, you should write:

[1] For defining the syntax we are going to use the same convention as described on page 37.

```
(Relation WHERE Condition1) WHERE Condition2
```

However, the syntax of relational algebra allows non-atomic conditions to be used in conjunction with the restriction operator. A non-atomic condition combines truth-value expressions by means of logical operators (AND, OR, NOT). For instance, the above nested expression can be expressed as

```
Relation WHERE Condition1 AND Condition2
```

A more comprehensive example concludes some of the above points. The following expression restricts products-Details relation to those products that have the string "MM" in their ID or are supplied by "S-TV" and whose value in stock is greater than £500:

```
Products-Details  WHERE   ("MM" SUBSTRING_OF ProductID   OR
                            Supplier = "S-TV")              AND
                           Cost * InStock > 500
```

Projection

Whereas a restriction could be seen as a "*horizontal*" sub-relation, a projection can be viewed as a "*vertical*" sub-relation of a given relation. If R and R' are two relations such that R' is the projection of R, then:

- R and R' have the same cardinality;
- the order of R' is smaller than the order of R.

Its "practical" use is to leave aside some information (i.e. attributes) about an entity (i.e. relation), that are not relevant in a specific case. For instance, if you want to select only the information regarding product names, items in stock and their suppliers, you could project the Products-Details relation onto the ProductName, InStock and Supplier attributes yielding the table in Figure 25.

ProductName	InStock	Supplier
Washing machine	2	E-A Inc
Dish dryer	0	E & E Ltd
Power supply extension	15	E-A Inc
Television set	0	S-TV

Figure 25: A projection of Products-Details

The syntax for the projection operator is

```
<relation name> [ <attr name 1>, <attr name 2>, ..., <attr name n> ]
```

where <attr name 1>, ..., <attr name n> should all be attribute names of the relation denoted by <relation name>. For instance, the projection above is described by the following expression:

```
Products-Details [ ProductName, InStock, Supplier ]
```

***Definition:** Formally, given a relation R having the heading {A, B, ..., C, X, Y, ..., Z}, a projection of R on A, B, ..., C is a relation having the heading {A, B, ..., C} and the tuples {A:a, B:b, ..., C:c} derived from all the tuples of R. Of course, duplicates are eliminated.*

Suppliers

Supplier	SupplyCity	Telephone
E-A Inc	Coventry	01203 456678
E & E Ltd	London	0181 3994567
H200	London	0171 1233456
S-TV	Tokyo	0081 3 11122233

Figure 26: Information about the suppliers of the products in Products-Details

Join The join operator, as its name suggests, joins two relations together. It is similar to the Cartesian product, but the join is based on common attribute values. Informally, if two relations R1 and R2 have some attributes in common (i.e. they have the same name and are defined on the same domains), then the result of the *join* of R1 with R2 is a relation made of all 'combined' tuples t1 of R1 and t2 of R2 (i.e. t1 ∪ t2) for which t1 and t2 have the same values for the common attributes. For example, the result of the relation Products-Details (Figure 23) joined with Suppliers (Figure 26), the join being performed over the common attribute Supplier, is depicted in Figure 27.

PruductID	ProductName	Cost	InStock	Supplier	SupplyCity	Telephone
PID23	Washing machine	289	2	E-A Inc	Coventry	01203 456678
XX24A	Dish dryer	399	0	H200	London	0171 1233456
00012	Power supply extension	14.99	15	E-A Inc	Coventry	01203 456678
MM25y	Television set	395	0	S-TV	Tokyo	0081 3 11122233
MM45x	Television set	555	0	S-TV	Tokyo	0081 3 11122233

Figure 27: Joining the information about products with that about suppliers

There are two types of join operators:

- *natural join*, based on equality;
- θ-join (theta-join), based on satisfying a condition.

Formally, the natural join operation can be defined as follows.

Definition: *Consider the relations R1 and R2, having the headings {X1, X2, ..., Xm, Y1, Y2, ..., Yn} and {Y1, Y2, ..., Yn, Z1, Z2, ..., Zq} respectively; Y1, Y2, ..., Yn represent the only common attributes of R1 and R2 (i.e. they have the same name and are defined on the same domain). The* natural join *of R1 and R2, denoted*

```
R1 JOIN R2
```

is a relation having the heading {X1, X2, ..., Xm, Y1, Y2, ..., Yn, Z1, Z2, ..., Zq} and the body consisting of the set of all possible tuples {X1:x1, ..., Xm:Xm, Y1:y1, ..., Yn:yn, ..., Z1:z1, ..., Zq:zq} such that {X1:x1, ..., Xm:Xm, Y1:y1, ..., Yn:yn} apperas in R1 and {Y1:y1, ..., Yn:yn, ..., Z1:z1, ..., Zq:zq} appears in R2.

Two properties of the join operator can be derived from the definition, namely:

- R1 JOIN R2 = R2 JOIN R1 (JOIN is commutative)
- (R1 JOIN R2) JOIN R3 = R1 JOIN (R2 JOIN R3) (JOIN is associative)

The theta-join operation is defined with respect to the Cartesian product, denoted by "TIMES".

Definition: *Consider the relations R1 and R2 and suppose the attribute X belongs to R1 and Y to R2. Suppose also that θ is an operator such that x θ y is a truth valued expression for any x value of X and y value of Y. Then the θ join of R1 and R2 is defined by*

```
(R1 TIMES R2) WHERE X θ Y
```

For example, given a set of parcels and a set of cars (see Figure 28) we would like to get all the combinations pairs parcel-car such that a parcel can be transported with the pair car. This is expressed by the condition weight of parcel < capacity of car. This can be achieved by means of a theta-join, and the expression that denotes the result is (depicted in Figure 29)

```
(Parcels TIMES Cars) WHERE Weight < Capacity
```

Parcels

Parcel	Weight	Destination
P1	215	London
P2	347	London
P3	612	Glasgow

Cars

Car	Capacity	Registration
C1	500	P 123 MOB
C2	1000	R 667 LLL

Figure 28: Parcels and Cars

Parcel	Weight	Destination	Car	Capacity	Registration
P1	215	London	C1	500	P 123 MOB
P1	215	London	C2	1000	R 667 LLL
P2	347	London	C1	500	P 123 MOB
P2	347	London	C2	1000	R 667 LLL
P3	612	Glasgow	C2	1000	R 667 LLL

Figure 29: The result of a θ-join

Note that if θ is "=" then the theta-join becomes a natural join.

Division

Suppose the two relations illustrated in Figure 30, describing all the parts supplied by each supplier, Contracts, and a sets of parts, each needed for a certain purpose, Set1, Set2, Set3.

Contracts

Supplier	Part
S1	P1
S1	P2
S1	P3
S1	P4
S2	P2
S2	P3
S2	P5
S3	P3
S3	P5
S4	P3

Set1

Part
P1
P2
P3

Set2

Part
P3
P5

Set3

Part
P5

Figure 30: Suppliers and parts

Suppose we need to find out, in turn, which are all the suppliers who supply all parts for each of the three sets. The results are: (1) only S1 supplies all parts of Set1 (i.e. P1, P2 and P3), (2) only S2 and S3 supply all parts of Set2 (i.e. P3 and P4) and (3) S2, S3, and S4 supply all parts of Set3 (i.e. P5). They are illustrated in Figure 31.

Contracts DIVBY Set1

Supplier
S1

Contracts DIVBY Set2

Supplier
S2
S3

Contracts DIVBY Set3

Supplier
S2
S3
S4

Figure 31: Results of Contracts divided by Set1, Set2 and Set3 respectively

These results can be obtained by dividing Contracts to Set1, Set2 and Set3 respectively. Division is particularly useful when we need to get all groups of tuples such that a certain projection of them is identical to a given relation; more intuitively, when we need results of the kind "suppliers who supply *all* parts of a set". Formally, division is defined as:

Definition: Consider the relations R1 and R2 having the headings {X1, ..., Xn, Y1, ..., Ym} and {Y1, ..., Ym} respectively. Suppose Y1, ..., Ym are common attributes (having the same name and being defined on the same domain) of R1 and R2 (which means that R2 has no attribute that is not common with R1). The result of the division of R1 by R2, written as

```
R1 DIVBY R2
```

is a relation R3, having the heading {X1, ..., Xn} and the tuples {X1 : x1, ..., Xn : xn} such that the tuples {X1 : x1, ..., Xn : xn, Y1 : y1i, ..., Ym : ymi} (i from 1 to k, where k is the cardinality of R2) appear in R1 for all tuples {Y1 : y1i, ..., Ym : ymi} of R2 (i from 1 to k).

It is worth pointing out that the 8 operations presented do not constitute a minimal set; some of them can be expressed in terms of the others. For instance, that the theta-join operator is expressible in terms of the Cartesian product and the restriction operator (inherent in the definition); the join operator can be expressed in terms of the theta-join operator (consequently, Cartesian product and restriction). The minimal set of operations – the primitive set - is actually represented by {*restriction, projection, Cartesian product, union, difference*}. The other three can be expressed in terms of the primitive five, but since in practice there is a need for such operations (join, in particular) it is indicated that they are specifically supported by the system (language) that implements relational algebra.

The power of relational algebra consists in its ability to manipulate relations as a whole. The relational closure property allows for a wide (unlimited) set of statements to be expressed, by combining the eight basic operators. A grammar for relational algebra expressions can be found at page 144 in (Date, 1995).

In the next subsection we are going to look at some examples which illustrate the power of this formalism.

Examples

Consider the relations Students, Registrations, Tutors, Teaching and Modules as described in Figure 32. The values of these relations are not relevant and are therefore omitted.

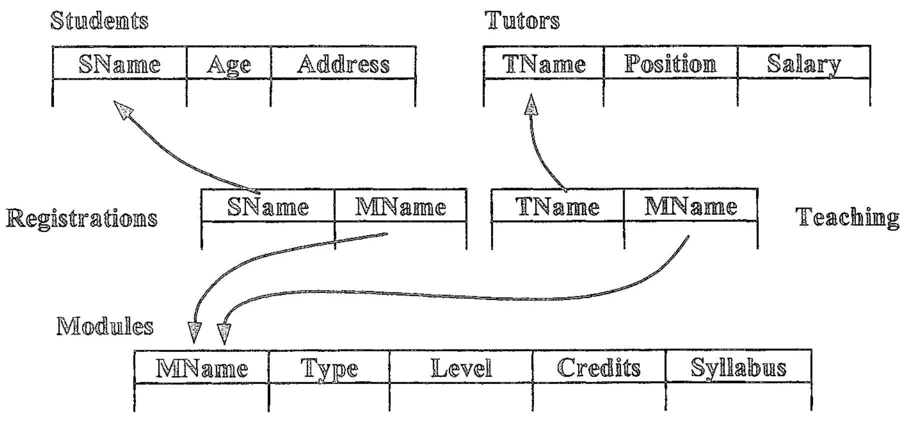


Figure 32: The relations used in exemplifying relational algebra operators

The only clarification that has to be made is about the Type of a Module; this is either compulsory or optional. All the other attributes are self explanatory. Here are some examples.

1. Get the name of the tutors who teach at least one module.

```
Teaching [TName]
```

2. Get the name of the tutors who do not teach any module (backslash, "\", stands for set difference).

```
Tutors [TName] \ Teaching [TName]
```

3. Get the name, position and salary of the tutors who don't teach any module.

```
( Tutors [TName] \ Teaching [TName] ) JOIN Tutors
```

4. Get all the modules that are taught by Professors or are of 1cu credit.

```
( ( Tutors WHERE Position = "Professor" ) JOIN Teaching ) [MName] ∪
( Modules WHERE Credits = "1cu" ) [MName]
```

5. Get the name and the address of all students who take level 1 courses.

```
( ( Modules WHERE Level = 1) JOIN Registration JOIN Students )
[SName, Address]
```

Another expression, having the same result, could have been

```
( ( Modules WHERE Level = 1) [MName] JOIN Registration ) [SName]
JOIN
Students [SName, Address]
```

6. Get the name of all students who take all optional courses.

```
(Registration DIVBY ((Modules WHERE Type = "optional") [MName] )) [SName]
```

7. Get the names and the syllabuses for all the courses taken by the student "Michael Daniel" together with the respective tutors, name and position, who teach them.

```
( (Registrations WHERE SName = "Michael Daniel") JOIN Modules)
[ MName, Syllabus ]
JOIN
( Tutors [TName, Position] JOIN Teaching )
```

another possible expression is

```
( (Registrations WHERE SName = "Michael Daniel")
  JOIN Modules
  JOIN Teaching
  JOIN Tutors
) [ MName, Syllabus, TName, Position ]
```

Activity: In order for you to get a good grip on relational algebra and also realise its expressive power you should work through a series of examples, by formulating your own statements (queries) in natural language and then translating them in relational algebra expressions. Remember that it is possible to devise more than one relational algebra expression for one natural language statement.

Data manipulation and the optimiser

We have seen that each relational DBMS should provide a language by means of which to define the domains and relations of a relational model (DDL). In order to retrieve and update data, a DML (Data manipulation Language) should also be provided. It is very often the case that commercial (industrial) DBMSs provide a DML which implements relational algebra; i.e., the DML allows the statement of relational algebra expressions as a means to retrieving and updating data. As a matter of fact, the standard relational DB language is SQL. SQL implements a subset of relational algebra[1].

Relational algebra is considered to be a "yardstick" against which the expressive power of a relational language is measured. A (relational) database language is said to be relationally complete if it is at least as powerful as relational algebra, i.e. if any expression from relational algebra can be implemented in the language. None of the

[1] The next chapter is dedicated to the study of SQL.

extant DB languages are relationally complete (at least, the author is not aware of such an implementation).

As we pointed out before, more than one relational algebra expression can be devised for one natural language statement (query). From the point of view of relational algebra, all the respective expressions are equivalent.

However, this is not the case if we consider a database language that *implements* relational algebra (such as SQL). The data manipulation statements of such a language are based on relational algebra expressions. Consider the case of querying a database. The user issues a query – the equivalent of a relational algebra expression – and the DBMS evaluates it. The time taken to evaluate a query is dependent on the statement the DBMS is issued with; we say that some queries are faster than others. Obviously, we aim for *efficient* queries, i.e., queries that take little time for evaluation.

Relational algebra has two characteristics that are relevant in this context:

- relational algebra expressions specify the operations, but not the *order* in which they are to be performed;
- relational algebra operators are set at a time; i.e., the way they are performed on individual tuples is not specified.

The order in which the operations are performed when evaluating an expression, and the mechanisms of executing set level operators (on individual tuples) are both implementation issues. They are taken care of by a module of the DBMS called the *optimiser*. The optimiser decides upon the best evaluation strategy (mechanism) for a given expression.

A consequence of the existence of the optimiser is that the users do not have to worry about how to best state the queries as long as the expressions they devise are correct.

To make things clearer, let us consider the following example. Suppose there are two relations, Students and Registrations, describing the students of a university and the modules they take, respectively. The Students relation has 3 attributes, {Student-Name, Address, Fees} and Registrations has 2, {Student-Name, Module-Name}. Suppose also that the university has some 3000 students (i.e. 3000 tuples in the Students relation) and each students takes about 4 modules per semester (i.e. approximately 12000 tuples in the relation Registration). Suppose that the number of students that take the AI module is about 100 and that the lecturer for this module wants to send each of them a letter. The lecturer will issue the following query:
"Get the addresses of all students who take the 'AI' course"
A corresponding relational algebra statement is

```
((Students JOIN Registrations) WHERE Module-Name = "AI") [SName, Address]
```

Consider, now, two possible evaluation strategies:

- perform the join (*12000 tuples have to be joined*), then the restriction (12000 tuples have to be "searched" to identify the corresponding 100) and then the projection;
- perform the restriction first on Registration (search 12000 tuple to identify the corresponding 100), then the join (*only 100 tuples have to be joined*) and ultimately the projection.

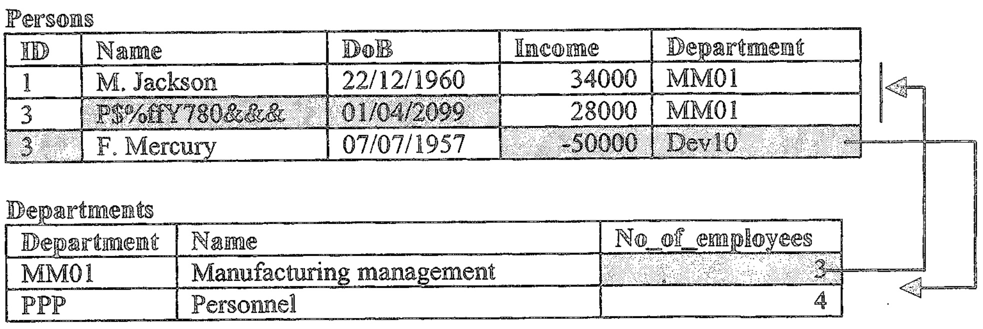
It is clear that the latter strategy is far better than the first, because it involves only 100 join operations between tuples, as opposed to 12000 for the first.

For a more detailed account of optimisation issues you can refer to Chapter 18 in (Date 1995), pp. 501-543.

Relational data integrity

Databases are systems that implement data models of real life systems. If data were to be modelled only by specifying its structure and the operations that can be performed upon it, then an important aspect of real life systems would be lost. In real life systems, all kind of *constraints* usually exist between the data values. The data incorporated within a database has to comply with these constraints in order for it to be correct, i.e. an accurate representation of reality.

For example, the data illustrated in Figure 33 (two relations, Persons and Departments) illustrate some of the possible inaccuracies in data representation. The ID attribute, of the Persons relation, was intended to be a unique identifier for a person. However, because this constraint (the uniqueness property) was not expressed in any way, it was possible to have a duplicated value in rows[1] 2 and 3. An invalid name and an invalid date of date of birth were given in row 2. Row 3 contains an invalid value for income (negative) and refers to an non-existent department (in table Departments). According to the table Persons, there are 2 persons working in the MM01 department, but according to the table Departments, there are three. All these examples illustrate the fact that certain configurations of data cannot be a valid model of reality and such situations must be somehow described and subsequently avoided.



Persons

ID	Name	DoB	Income	Department
1	M. Jackson	22/12/1960	34000	MM01
3	P$%ffY780&&&	01/04/2099	28000	MM01
3	F. Mercury	07/07/1957	-50000	Dev10

Departments

Department	Name	No_of_employees
MM01	Manufacturing management	3
PPP	Personnel	4

Figure 33: Possible inaccuracies when modelling data purely by means of relations

An informal statement (description) of some of the integrity constraints – *integrity constraints* – for the situation described above could be:

1) no two tuples in the relation Persons can have the same value for the ID attribute;
2) any date of birth (DoB) in the Persons relation has to be of the form dd/mm/yyyy and should be between 01/01/1930 and 01/01/1975;
3) the values for income should be positive (an upper limit might exist as well);
4) for each tuple in Persons, there must exist a tuple (and only one) in Departments, such that they have the same value for the attribute Department;
5) the number of persons working for a certain department, tuples in the Persons relation, must be the same with the value of the No_of_employees attribute, for the corresponding department, in the Departments relation.

Data integrity denotes the accuracy or correctness of data. In order to devise a *correct* model of a real life system, the set of constraints existing between the data values must be identified and specified.

In the context of the relational model, two types on integrity constraints can be identified, namely:

[1] We can consider the ordering of the rows (in the table), without implying that the tuples (in the relation) are ordered in any way. This is done only to make the reading easier.

- *application* or *database specific* - in that they are applicable only to the application at hand;
- *generic* or *general* – relevant to the integrity of any model.

A positive value for "Income", values of a certain format and within certain limits for "DoB", correspondence between the number of persons (tuples) belonging to a certain department in the relation Persons and the "No_of_employees" in Departments are all example of database specific integrity constraints. The relational model does not specifically cater for their expression[1], but DB languages, to a certain extent, provide support for their expression. Relational algebra expressions, in particular, can be used for their representation. This aspect will be illustrated later, in the next chapter.

The fact that the attribute chosen for tuple identification has to be unique (constraint 1 above) and the corresponding values constraint, used for linking relations (constraint 4, above) are aspects of two integrity constraints, relevant to any relational model.

The two general integrity constraints stem from the following rationales:

- an addressing mechanism must exist that provides the unique identification of each tuple within a relation;
- a tuple in a relation must not refer to another tuple (either in the same or in another relation) that does not exist.

They are modelled by means of two concepts of the relational model, namely *candidate key* and *foreign key*.

Candidate keys

Definition: Given a relation[2] R, a subset PK of the attributes of R represents a candidate key for R if, for any extension, *it has*

- *the* uniqueness *property, i.e. no distinct tuples in R have the same value for PK, and*
- *the* irreducibility *property, i.e. no proper subset of PK has the uniqueness property.*

One of the generic integrity constraints, *entity integrity*, can now be stated:

The entity integrity constraint specifies that each relation must have at least one candidate key.

This constraint is always satisfied within the relational model, i.e. every relation has at least one candidate key. Since a relation R cannot contain duplicate tuples, the entire set of attributes of R is always a *possible* candidate key. If it has the irreducibility property, then *it is* a candidate key of R (the only one). If it is reducible, then it must have a proper subset that is irreducible, hence, a candidate key.

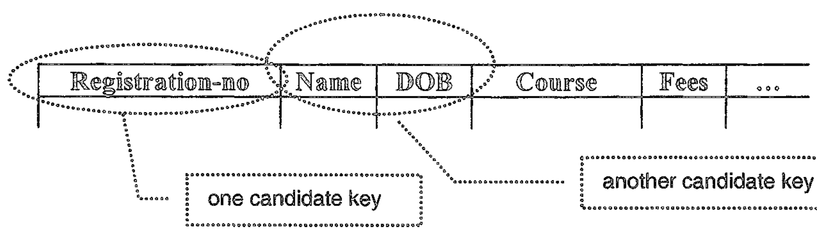


Figure 34: Candidate keys

Consider, for instance, the relation Students, in Figure 34. Each student has her/his own unique registration number. Therefore, this is a candidate key. Two different students

[1] Some of them might be expressed (enforced) by means of domains – see the end of this subsection.

[2] We are referring to a relation *variable*.

cannot have the same name and date of birth. Therefore, the subset {Name, DOB} represents another candidate key for Students.

If a candidate key consists of only one attribute it is said to be *simple.* If it contains more than one attribute it is *composite.*

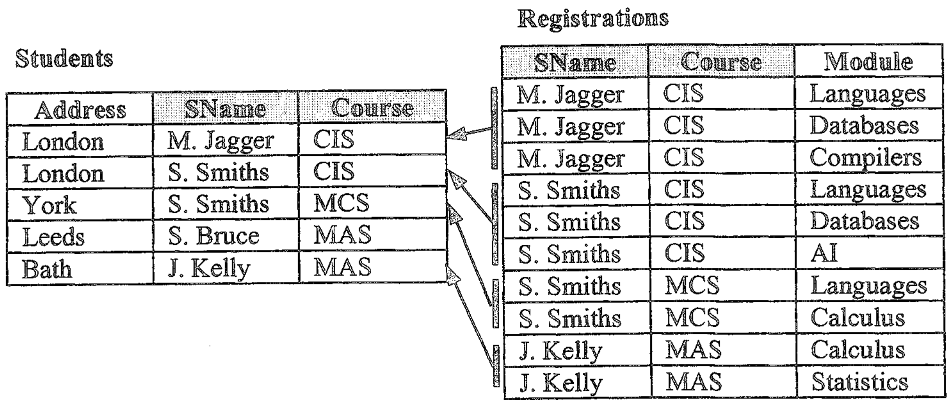
A given relation can have more than one candidate key. The key chosen for tuple addressing, from the set of candidate keys of a relation, is said to be the *primary key* of that relation. Theoretically, this choice can be performed arbitrarily, it depends entirely upon the designer's / user's choice. The rest of the candidate keys are called *alternate keys*.

For a database system, the candidate keys specification *enforces* the uniqueness property of the corresponding attributes. That is, the system "makes sure" that no update operation can result in a candidate key having duplicate values.

Foreign keys

Definition: *Suppose two relations R1 and R2. A subset of the attributes of R2, FK, is a foreign key of R2, referencing R1, if:*

- *R1 has a candidate key CK, defined on the same domains as FK, and*
- *at all times, each value of FK is equal to the value of CK in some tuple in R.*



Students

Address	SName	Course
London	M. Jagger	CIS
London	S. Smiths	CIS
York	S. Smiths	MCS
Leeds	S. Bruce	MAS
Bath	J. Kelly	MAS

Registrations

SName	Course	Module
M. Jagger	CIS	Languages
M. Jagger	CIS	Databases
M. Jagger	CIS	Compilers
S. Smiths	CIS	Languages
S. Smiths	CIS	Databases
S. Smiths	CIS	AI
S. Smiths	MCS	Languages
S. Smiths	MCS	Calculus
J. Kelly	MAS	Calculus
J. Kelly	MAS	Statistics

Figure 35: An example of a foreign key

Note that the reverse of the second requirement above is not necessary. There may exist values for CK that are not matched by any value of FK.

In the above example (Figure 35), each tuple in Registrations has a corresponding tuple (having the same value for SName) in Students. For instance:
{M. Jagger, CIS, Language} corresponds to {London, M. Jagger, CIS};
{M. Jagger, CIS, Databases} corresponds to the same {London, M. Jagger, CIS} tuple;
{S. Smiths, CIS, Languages} corresponds to {London, S. Smiths, CIS}.

However, there is not tuple in Registration that corresponds to the {Leeds, S. Bruce, MAS} tuple in Students.

A foreign key, as a candidate key, can be either simple of composite.

Let us now consider an example of a composite foreign key (Figure 35). The Students relation has {SName, Course} as primary key. {SName, Course} is a foreign key in the Modules relation, referencing Students. Students is the *target* or the *referenced relation*, whereas Modules is the *referencing relation.* {"M. Jagger", "CIS", "Databases"} is a *referencing tuple* in Modules, whereas {"M. Jagger", "CIS", "London"} is its *target* or *referenced tuple* in Students.

The fact that a relation R2 has a foreign key that references a relation R1 can be represented diagrammatically, in the form of a *referential diagram, as in* Figure 36.

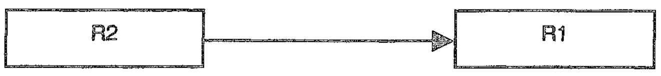


Figure 36: A simple referential diagram - R2 references R1

A referential diagram is actually constructed either for the whole database, i.e. the whole set of relations (modelling a real life system), or for a substantial part of it. For instance, for the relations in Figure 32 we have the following referential diagram.

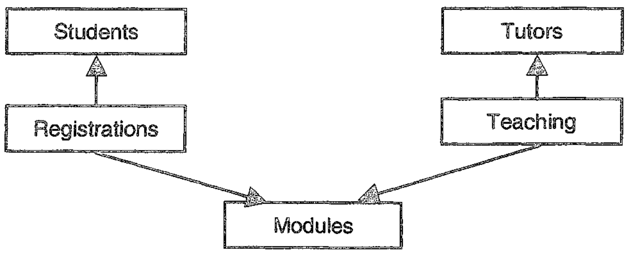


Figure 37: A more complex referential diagram

The other generic integrity constraints, *referential integrity*, can now be stated:

*The **referential integrity** constraint specifies that the database, i.e. the whole set of relations, must not contain unmatched foreign keys.*

In other words, there always must exist a target tuple for any existing referencing tuple.

For a database system, the specification of the foreign keys enforces the referential integrity constraint. That is, the system "makes sure" that the result of any update operation cannot result in unmatched foreign keys.

Nulls

When a (relational) model is built, it is usually assumed that all the analysed information is available. However, there are situations when, for a certain real life system, some information is unavailable or missing. For instance:

- the address of the customers of a certain chain of shops is confidential, therefore there may be situations when this information is not provided;
- the new students have registered with the university but they have not yet decided which modules to take; therefore this information is not yet available;
- some antique coins have been found on an architectural site, but the year is unknown.

How is the relational model / a relational system going to deal with such situations? The answer is: by means of nulls.

***Definition:** A null is a way of representing missing information.*

Note that a null is not a value; do not confuse it with zero or blank. A null is a marker and means unknown value. For clarification, consider the example in Figure 38.

Name	DOB	Sex	Course	Year1	Year2	Year3
A. Johnson	04/66	M	CIS	68	72	NULL
S. Bruce	02/72	F	NULL	NULL	NULL	NULL
P. Harris	10/73	NULL	MAS	0	48	NULL

Figure 38: A relation containing nulls

Some definitions previously given have to be revisited in the new light of the nulls.

The entity integrity constraint specifies that each relation must have at least one candidate key. *Furthermore, a candidate key is not allowed to accept null values.*

Foreign key Definition: Suppose two relations R1 and R2. A subset of the attributes of R2, FK, is a foreign key of R2, referencing R1, if:

- R1 has a candidate key CK, defined on the same domains as FK, and
- at all times, each value of FK is *either null* or is equal to the value of CK in some tuple in R.

The referential integrity constraint specifies that the database, i.e. the whole set of relations, must not contain unmatched foreign keys. *It only refers to non-null values.*

Domains and normal forms

There are two other mechanisms within the relational model that can be used for the expression of certain kind of constraints.

Firstly, there are the *domains*. Domains can be used to impose limitations on the admissible values of a certain attribute and also on the admissible operations that can be performed upon it. For instance, in order to express constraint 2 in the example above (i.e. any date of birth (DOB) in the Persons relation has to be of the form dd/mm/yyyy and should be between 01/01/1930 and 01/01/1975), we can define a domain, say DatesOfBirth, and then define the attribute DoB of this type.

As a language for illustration we shall use the "reduced" SQL language as defined at the beginning of the "Data definition in a relational DBMS" section. However, for illustration purposes we allowed for an interval of integer values and for a record to be specified by
"INTERVAL OF INTEGER [<min> .. <max>]" and
"RECORD (<type>/<type>/<type>)"
respectively, even though these data types are not supported by SQL.

```
CREATE DOMAIN Days          AS   INTEGER [1 .. 31]
CREATE DOMAIN Months        AS   INTERVAL OF INTEGER [1 .. 12]
CREATE DOMAIN Years         AS   INTERVAL OF INTEGER [1930 .. 1975]
CREATE DOMAIN DatesOfBirth AS   RECORD (Days/Months/Years)
CREATE BASE RELATION Persons (
    ID              INTEGER,
    Name            VARCHAR,
    DoB             DatesOfBirth,
    --etc.
)
```

Suppose now a slightly different definition of the base relation Persons, which allows direct access to each component of a date of birth, as illustrated below:

```
CREATE BASE RELATION Persons (
    ID                  INTEGER,
    Name                VARCHAR,
    Day_of_Birth        Days,
    Month_of_Birth      Months,
    Year_of_Birth       Years,
    --etc.
)
```

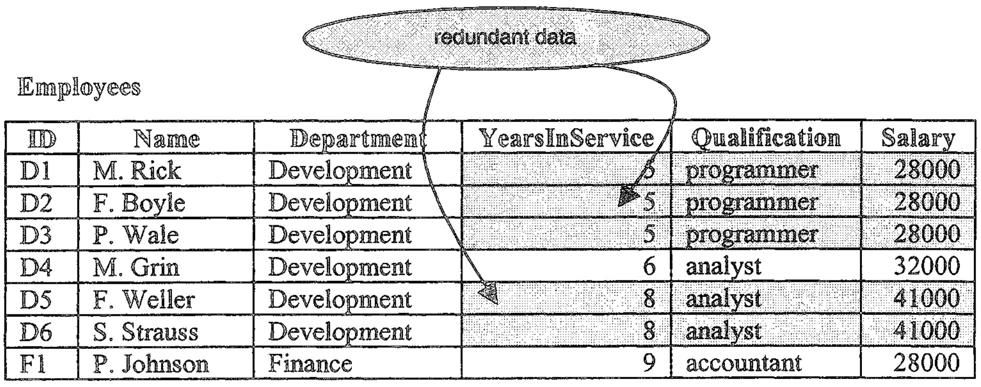
Suppose also that we consider that it makes no sense to add, multiply or divide days, months or years of birth, it only makes sense to subtract (in order to find intervals). We, therefore, have to impose these constraints on the Day_of_Birth, Month_of_Birth and Year_of_Birth attributes. We simply do not define the addition, "+", multiplication "*" and division ":" operators for the three domains, "Days", Months" and "Years".

Unfortunately, most of the existing relational DB languages, SQL in particular, do not support domains. However, you will see in a following chapter that SQL provides another mechanism to impose limitations on allowable attribute values. Moreover, by means of the same mechanism, further constraints can be imposed, such as relations between attributes.

Secondly, there are the *normal forms.* Three particular kind of constraints, called functional dependencies, multiple dependencies and join dependencies) are expressed by means of normal forms. They are treated in detail in Chapter 5. However, for the sake of completeness, they were also mentioned in this section.

Loosely speaking, to normalise a relation R means to decompose it into a set of relations of smaller degree, with a view to eliminating certain dependencies between its attributes. Such dependencies cause redundant data to exist in the original relation R, which, in turn, may lead to problems with respect to update operations.

For example, consider the relation Employees, as illustrated in Figure 39.


redundant data

Employees

ID	Name	Department	YearsInService	Qualification	Salary
D1	M. Rick	Development	5	programmer	28000
D2	F. Boyle	Development	5	programmer	28000
D3	P. Wale	Development	5	programmer	28000
D4	M. Grin	Development	6	analyst	32000
D5	F. Weller	Development	8	analyst	41000
D6	S. Strauss	Development	8	analyst	41000
F1	P. Johnson	Finance	9	accountant	28000

Figure 39: A relation describing the employees of a company

The attribute ID is the unique candidate key. Moreover, the salary of an employee depends entirely on the job title and the years that person has spent in service. This means that, if we know the number of years an employee was in service and his/her job, we also know the salary of the employee. For instance a programmer who has been for five years in service earns £ 28000.

The dependency (constraint) between YearsInService, Qualification and Salary determine some redundant data (see Figure 39). In order to express this dependency (constraint) and inherently to eliminate the redundant data, the relation Employees is decomposed into two *equivalent* relations, Employees' and Salaries.

Employees'

ID	Name	Department	YearsInService	Qualification
D1	M. Rick	Development	5	programmer
D2	F. Boyle	Development	5	programmer
D3	P. Wale	Development	5	programmer
D4	M. Grin	Development	6	analyst
D5	F. Weller	Development	8	analyst
D6	S. Strauss	Development	8	analyst
F1	P. Johnson	Finance	9	accountant

Salaries

YearsInService	Qualification	Salary
5	programmer	28000
6	analyst	32000
8	analyst	41000
9	accountant	28000

Figure 40: A normalisation of the Employees relation (Figure 39)

Integrity constraints definition and foreign key rules

It is not sufficient for a data definition language (DDL) to support only the definition of the structure of data. It also has to support the definition of integrity constraints on data. Since the two generic integrity constraints are represented by means of keys, the DDL has to support their definition.

SQL supports the definition of keys, so we shall continue to use it for illustrations. Three more notational conventions are needed:

- "@" in front of a construct means a list of those elements, separated by comma
- the "::=" symbol means "is by definition";
- the vertical bar symbol, "|", signifies exclusive selection (either-or).

The syntax for data definition in SQL[1], allowing the definition of keys, is:

```
CREATE TABLE <relation name> (
      @<attribute definition>,
      <primary key definition>,
      @<candidate key definition>,
      @<foreign key definition>
);
<primary key definition> ::=     PRIMARY KEY ( <set of attributes> )
<candidate key definition > ::= CANDIDATE KEY ( <set of attributes> )
<foreign key definition> ::=     FOREIGN KEY ( <set of attributes> )
                                 REFERENCES <relation name>
```

For instance, the example relations in Figure 33, Persons and Departments, can be defined as (note that a slight modification has been made to Persons, namely that no ID attribute is used, but, instead, any person is fully identifiable by their name and date of birth):

```
CREATE TABLE Persons (
      Name              VARCHAR,
      DOB               DatesOfBirth,
      Income            INTEGER,
      Department        CHAR(5),
      PRIMARY KEY       (Name, DOB),
      FOREIGN KEY       (Department)  REFERENCES Departments
);
CREATE TABLE Departments (
      Department        CHAR(5),
      Name              VARCHAR,
      No_of_employees   INTEGER,
      PRIMARY KEY       (Department)
);
```

Since relations are linked to each other via foreign keys, i.e. by means of the values of some of their attributes, a question arises naturally: what happens when one of these values is changed in one of the relations, how is the link going to be maintained? More precisely, what happens when the value of the primary key attribute of a target tuple changes? What should happen to the referencing tuples? The answer lies in the *foreign key rules*.

There are two possible relevant situations:

- the target tuple is going to be deleted;
- the primary key attribute of the target tuple is going to be modified.

[1] still simplified

If no other action is taken, then, in both cases, the referencing tuples will be left referring to a tuple that does not exist anymore. Such a situation is not acceptable. Therefore, some "extra" operations must be performed by the DBMS in order to maintain the consistency of the database.

SQL provides two types of actions that a DBMS can perform automatically in case a target tuple is modified, to *restrict* or to *cascade*. Consider the following four examples.

A database, amongst others, comprises two relations describing its employees and their children. Their definition is

```
CREATE TABLE Employees (
     ID                       INTEGER,
     Name                     VARCHAR,
     -- other attributes
     PRIMARY KEY              (ID)
);
CREATE TABLE Children (
     Child-Name               VARCHAR,
     ID                       VARCHAR,  --the parent's ID
     -- other attributes
     PRIMARY KEY              (Child-Name, ID),
     FOREIGN KEY              (ID) REFERENCES Employees
);
```

If an employee ceases to work for the company, the respective tuple from Employees has to be deleted. Suppose this person has some children, reflected by the corresponding tuples in Children. What is going to happen to these tuples? The answer is that the children have to be deleted from the database together with their parent. So, the delete operation upon the parent has to be *cascaded* upon the referencing tuples.

Consider now the case of a university database, that contains, amongst others, two relations about its students and the books they have borrowed from the library.

```
CREATETABLE Students (
     S-ID                     INTEGER,
     Name                     VARCHAR,
     -- other attributes
     PRIMARY KEY (ID)
);
CREATE TABLE Borrowings (
     Book-Title               VARCHAR,
     S-ID                     VARCHAR,
     -- other attributes
     PRIMARY KEY              (Book-Title, S-ID),
     FOREIGN KEY              (S-ID) REFERENCES Students
);
```

When a student finishes the degree (graduates), the corresponding tuple has to be deleted from Students. However, a graduate student might still have some books borrowed from the library. What should happen to the corresponding tuples in the relation Borrowings? The answer is that the deletion of a tuple in the Students relation should not be allowed until there is no tuple in the relation Borrowings that references it. In other words a students cannot be deleted from the university's database until the student has returned all his/her books to the library. It is said that the deletion of the Students tuple has to be *restricted* by the referencing tuples in Borrowings.

An updating example. Suppose that a company's database contains, amongst others, two relations, Employees and Departments. Suppose the relation Employees has an attribute Department, specifying the department's id for which the employee works. This is a foreign key attribute referencing the Departments relation (the department's id is a primary key in Departments). What happens if the id of a certain department is going to be updated (because the management decided to change it), say, from "Dev" to "Development_1 "? The answer is that the corresponding value must be updated in the Employees relation as well. For the given example, all the employees who worked for "Dev" should now work for "Development_1". It is said that the update was *cascaded.*

Finally, reconsider the database of a university, but two other relations this time, Registrations and Modules. The first relation describes which modules are taken by each student, and the latter describes each module in part:

```
CREATE TABLE Modules (
     Module-ID               CHAR(5),
     Module-Name             VARCHAR,
     -- other attributes
     PRIMARY KEY             (Module-ID)
);
CREATE TABLE Registrations (
     Student-ID              INTEGER,
     Module-ID               CHAR(5),
     --- other attributes
     PRIMARY KEY             (Student-ID, Module-ID),
     FOREIGN KEY             (Module-ID) REFERENCES Modules,
     --- others
);
```

Suppose that the Database Systems module has to change its ID from CIS205 to CIS209. Most probably that this change implies also a change in the structure of the course. Can this update be performed in the Modules relation? The answer is yes, but only if there are no students registered for CIS205; the former CIS205 cannot cease to exist in the database (by updating it to CIS209) as far as there still are students taking it. So, the update in Modules must be *restricted* if there exist any referencing tuple in Registrations.

It can be inferred from these examples that two actions are possible in case of an updating operation (deletion or modification) of a primary key of a relation that is referred by another relation:

- for the update of the target tuple to be *cascaded* (i.e. propagated) to the referencing tuples;
- for the update of the target tuple to be *restricted,* in case there exist any referencing tuples.

SQL allows the specification of foreign key rules:

```
CREATE TABLE <relation name> (
     <attributes definition>,
     <primary and candidate keys definition>,
     @<foreign key and foreign key rules definition>
);
<foreign key and foreign key rules definition> ::=
     FOREIGN KEY ( <set of attributes> ) REFERENCES <relation name>
             ON DELETE <option>
             ON UPDATE <option>
<option> ::= CASCADE I RESTRICT
```

For instance, for the first example above, the definition of the Children relation is:

```
CREATE TABLE Children (
     Child-Name              VARCHAR,
     ID                      VARCHAR,  --the parent's ID
     -- other attributes
     PRIMARY KEY             (Child-Name, ID),
     FOREIGN KEY             (ID) REFERENCES Employees
                             ON DELETE CASCADE
                             ON UPDATE CASCADE
)
```

The syntax of real implementations of DDLS is usually extended to provide support for the expression of other kind of integrity constraints. For instance, SQL supports the definition of constraints between attributes of different relations, by means of truth valued expressions. Such mechanisms are presented in Chapter 4 (describing the SQL language). They were mentioned here only for the sake of completeness.

As a final remark, *a database is considered correct if it satisfies the logical AND of all integrity rules.*

Conclusions

This chapter presented the main aspects of the relational model and how they are operationalised in a relational DBMS. The relational model is a theory by means of which the information related to a real life application can be modelled. The main advantage of the relational model consists in the synergy between its simplicity and its power of expression. Hence, the relational model was almost universally adopted as the theoretical basis for database systems; at the moment, it represents the de facto standard data model.

The next chapter presents the most popular language that implements the relational model, SQL. In the main you will learn how to create, query and maintain a database. The following chapters will then present and answer to the question: how can a *good* database be developed?

Learning outcomes

On completion of this chapter you should be able to:

- describe how a real life system can be modelled within a data model;
- describe the relational model and the way it is used in a relational DBMS;
- be familiar with the terminology of the relational model;
- describe the concept of domains;
- describe the concept of relations and discuss the properties relations;
- discuss, in general terms (i.e. based on a hypothetical language), how the relational data objects are operationalised (used in a relational DBMS);
- describe each of the operators of relational algebra;
- be able to express natural language statements, representing information to be inferred from relations, as relational algebra expressions;
- explain the way relational algebra is used in the context of DBMSs (including the optimiser);
- present different types of inconsistencies that can exist within a relational model;
- define and classify integrity constraints;
- define the concepts of candidate, primary, alternate and foreign key;
- discuss the issue of null values;
- describe how the definition of (generic) integrity constraints is operationalised (including the foreign key rules).

Bibliography

Codd, E. F., "A relational Model of Data for Large Shared Databanks." CACM 13, No. 6 (June 1970).

Codd, E. F., "Relational Completeness of Database Sublanguages", in Data Base Systems, Courant Computer Science Symposia Series 6. Englewood Cliffs, N.J.: Prentice Hall, 1972.