
Files and Databases

We have now completed Part 1 of this book, the discussion of file organizations and their design. Part 1 provides the basis for Part 2 of this book, the design of database structures. A database contains a diversity of related data so that typically several files are used to hold the data. Further files will be needed to hold descriptive information about the data and their relationships.

The analysis of files is characterized by an emphasis on performance issues. The analysis of databases concentrates on logical structures. The result of database analysis is a precise specification of the contents and the required manipulations of the database. Files and support programs that are chosen for implementation of a database have to satisfy both the database specification and provide adequate performance for the users of the database.

An integrated collection of support programs and file structures to support a database, its logical requirements, and an interface to users and user programs is called a database system. We will first deal with databases in an abstract fashion.

Chapter 7

Database Structure

An abstract term is like a valise with a false bottom, you may put in it what ideas you please, and take them out again, without being observed.

Alexis de Tocqueville
Democracy in America (1835)

We now consider the organization of a database. After some initial discussion of models we introduce some formal aspects of database semantics. In our model files are abstracted as relations and the interaction between files as connections. Section 7-3 presents six relation types and three connection types which become the building blocks for our database models. Section 7-4 defines the operations to manipulate these building blocks, and in Sec. 7-5 we discuss building an integrated database with these tools.

A *database system* is the combination of programs and files which are used together. An integrated collection of programs to support databases can form a *database management system*. A database system has to provide facilities at the level of individual data elements so that our models will consider the semantics of individual attribute types. Chapter 8 will concern itself with the description of data elements and file structures for implementation.

7-0 STRUCTURE DEFINITION

Data values derive their meaning from their relationship to other data elements. The grouping of data elements into the linear structure of a record has been the principal means to define relationships seen so far. A record typically contains data values related to one instance of some real-world entity or concept. Another linear relationship used has been the concept of seriality of records, exemplified by the *get-next* operation. More complex relationships have been encountered in the discussion of the ring-structured file, the hierarchical files, and in some of the examples of hybrid file structures. In order to be able to discuss the structure of a database consisting of many files we will need to define a number of concepts which can describe relationships among records and sets of records.

Structure in Data Processing In conventional data processing the relationship between records for a computation is implied by the processing action of programs. A program will fetch records from several files, associate the values found, and produce information for the user. The processing programs are written with knowledge of the contents of the files.

Example 7-1 Related files

For example, a vendor of goods will have

A file containing data on invoices sent for which payment is to be received

A file containing data on payments received during the day

A file giving a credit rating code to the firm's customers

The three files are structurally related. The `Payments` received are related to the `Invoices` issued through an `invoice_number` and the `Invoices` are related to the `Credit_rating` file via a `customer_number`. A computational procedure updates the credit rating based on all the invoices sent to a customer which are not marked `paid`, so that there is also a less obvious procedural relationship. Figure 7-1 illustrates the relationships recognized by an `Enter payments` application for a particular set of records.

The programmer will select records with matching values for related attributes for processing. Errors in the structure will lead to errors in the programs.

A payment without an `invoice_number` cannot be posted.

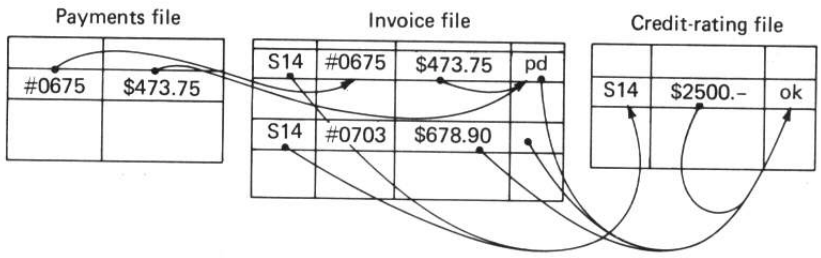


Figure 7-1 File structure to enter payments.

Introduction to Models The major method for database design is the construction of models which represent the database structure in a manner which allows the manipulation of the conceptual building blocks for the database. Only when the structure is well understood can a suitable file design be chosen. A model is hence the prime tool for the database designer. The objective of the modeling approach presented in this book is to provide a basis for the design of effective and usable databases.

The material in this chapter includes results of researchers and practitioners. While the more practical aspects of current work have been selected, many of the approaches in the literature have not been widely verified through application in the implementation of large systems. The structural model presented here has been used in a variety of situations, but any application has to be combined with *common sense* and *insight*.

This chapter will also make current research in database design more accessible. We have tried to use established terminology as much as possible. Reference to Appendix A is recommended to provide linkages to terms used in work of some authors. In the background section for this chapter a number of alternative modeling approaches are summarized in the terms developed here. Chapter 9 provides examples of the implementations of some modelling approaches.

We stated that, in conventional processing, the transaction program selects and combines records based on relationships defined by the programmer. These relationships are, however, independent of the processing programs. Relationships among the data within a file and among multiple files form a structure which establishes the meaning of the data. The objective of this chapter is to develop a means for recognizing and describing this structure in a way that guidance is provided for the design of databases and the selection of database management systems.

The tool used to describe databases is the *structural model*. It is based on a well-known formal model, the relational model, which defines single files or relations and operations applied to them. The relational model has been augmented with a formal definition of relationships between files, called *connections*. Three types of connections are used to classify relations into five types, which have distinct structural features (see Sec. 7-3-9 for a summary). When all the definitions for the database have been collected, we have a structural model of the database, or a *database model*.

The model captures some of the meaning or *semantics* of the data, so that less is left to the imagination of the programmers who write application procedures. Only the semantics that affect the structure of a database are captured in this model; finer semantic distinctions are ignored. If the information contained in a database model can be automatically interpreted by a database management program, some data-processing functions can be automated. Requests (queries and updates) to such a system can now be expressed in a higher-level language than a programming language. A language which lets us state the problem rather than the procedure to follow to get a solution, is called a *nonprocedural language*.

Our task is now to understand the basic concepts used to describe data and their relationships.

The fact that attribute values are relatable may be implied by observing that they

are selected from a common domain of values (`customer_number`, `invoice_number`, or `dollars`), although relationships (say `credit_rating_code` to `dollars`) can also exist between domains that are not explicitly the same. We will concentrate on the simpler cases where domains are identical. A function could be defined to create a derived attribute `dollar_limit` of type domain `dollars` for the `Credit_rating` in order to define procedural relationships in a structural manner.

Multiple Files and the Database In modern enterprises many functions are performed by computers. This means that many files will be in use. Many relationships will exist among these files. We defined database to be a collection of related files. In order to have a database system, the usage of these files must be integrated and coordinated. When we speak of a structure for the database, we consider all relationships among all files.

The management of available data is a major problem in many large businesses where files have been established as the need for data arose. The `Personnel` file will contain data on employee history and pay scales. When this file is processed at the end of a pay period, an output is generated which provides financial input data to an accounting system. Other financial data may come from manufacturing and sales applications. For the writing of a proposal for a new contract, data may be required from personnel and manufacturing files. For a cash-flow projection sales data will be needed together with financial data. Without a database system each time that data are jointly processed they will be selectively copied from source files and merged to construct new files appropriate for the application. The number of files in an enterprise can easily exceed several hundred, so that a major effort is required to manage the data and avoid errors.

The maintenance of a complex set of interrelated files can become a major bottleneck in a complex enterprise. When this problem is recognized, an integration of the files will be seen as desirable and a system to manage all the files as a database will be considered. Throughout the remainder of this chapter the assumption is made that the necessity for an integrated database does indeed exist.

Multiple Views If we want to consider the semantics of data in a model, we face an immediate problem. A database which is used for two different purposes may have two distinct models when viewed by the two users. While multiple views of a database may be implemented using multiple distinct file arrangements, the resulting multiplicity of files and the duplicated data in them create severe problems in updating and maintaining consistency of data.

A case of three views of the same database can occur in the `Patient-Drug` file presented in Sec. 4-5. One purpose for a drug database is to keep track of drugs prescribed for a specific problem of the patient, and the file was structured to make the drugs subordinate to the patient's problems. Certain drugs, however, will interfere with other drugs given. For a drug-interaction analysis it is preferable if all the drugs used by one patient are kept together. For a study of the effectiveness of dangerous drugs, researchers may see a hierarchy which begins with the drug type. A patient using many drugs appears in many records according to the drug researchers' view.

Dealing with several views simultaneously can make the construction of a valid database model nearly impossible. The views may conflict, although they are each correct. The more uses a database serves, the more conflicts arise between the

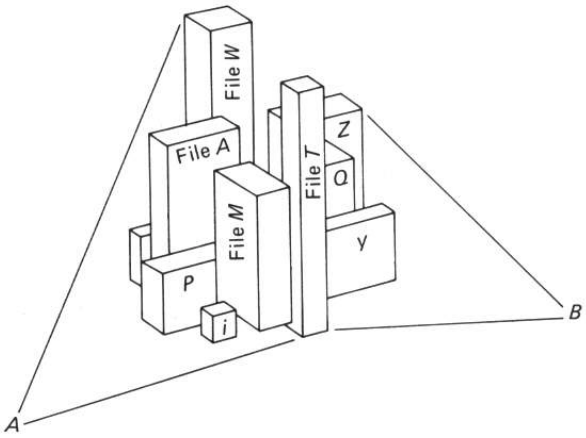


Figure 7-2 Two views of a database.

different views. In order to cope with this problem, we will define a database initially from one point of view only. Eventually the same database may be defined similarly from other points of view. The structural model provides the means to *integrate* two or more distinct views.

In the next four sections we define means to describe models for a single view. Integration of views is discussed in Sec. 7-5.

7-1 VIEW MODELS

One single view of the database can be described by one model. A view model represents a small subset of reality, appropriate to one application of the database contents. Most databases will require several view models for their specification. The parochial view-by-view approach to understand the structure of a database has the advantage that the complexities of relationships seen in real-world databases may be conquered.

Most modeling concepts apply equally well to database models as to view models. We will stress the term *view* only where the emphasis is warranted. Many models in the literature do not deal with the multiple-view problem directly, so that the term database model is often applied to what is actually a view model.

7-1-1 Relations and Their Domains

A view model is constructed from data elements and their relationships. Data elements represent values of attributes of entities. The most basic expression of a relationship is the binary relationship (Fig. 7-3*a*). The three elements of a binary relation (object, attribute, value) were recognized in the LEAP structures (Sec. 4-7).

Multiple relationships are found when data are collected and organized for processing. The steps in Fig. 7-3*a,b,c,d* provide examples of the initial building blocks of view models. First a number, say n, of simply related data elements may be placed into a list or *n-tuple*. One n-tuple represents multiple attributes of some object. This grouping simplifies the management but makes the original relationships less obvious.

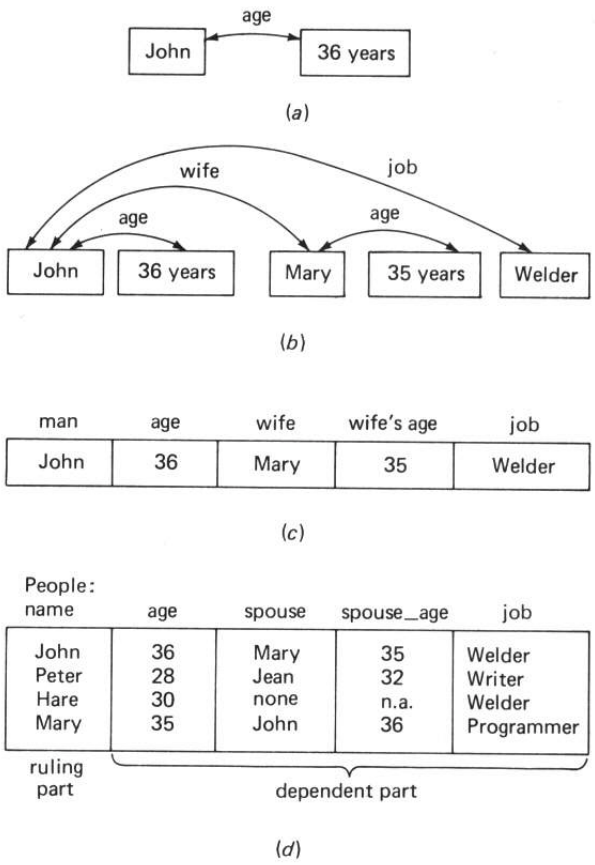


Figure 7-3 Relationships, tuples, and relations. *(a)* A binary relationship. *(b)* Multiple binary relationships. *(c)* Tuple. *(d)* Relation.

Sets of similar tuples are assembled into *relations*. Each tuple in a relation represents some distinct object and expresses similar relationships among its attributes. The choice of values for any of the attributes, the *domain*, is also specified.

Each relationship in a record is defined by the name labelling the tuple columns: the *attribute*. A column of a relation now contains similar data elements. The specific relationships among attributes is not represented in the relation but is implied by the heading line of attributes, the *relation-schema*. The ordering of attributes is immaterial as long as the tuple layout matches the relation-schema. The database structure will be defined so that the original relationships can be properly recovered.

Each tuple exists in order to describe a specific object O. We can consider one or some of the attribute columns to name or define the object. These attributes form the *ruling part*. The remaining attributes (V) are the *dependent part*. This distinction provides the essence of the original relationships among the attributes and will be vital when models are manipulated. We will make sure that the relation-schema of a view model contains only attributes which can be properly assigned to the ruling or to the dependent part.

In Fig. 7-3 the decision to include the `spouse_age` into the tuple has led to redundancy in the `People` relation. Another model for the given relationships will be less troublesome, but more than one relation will be needed.

We can summarize the important definitions given as follows:

A *tuple* is a collection of related heterogeneous atomic data elements.

A *relation* is a set of homogeneous tuples. Within one relation the data elements in a specific position of any tuple belong to the same attribute.

A *relation-schema* describes the attributes in terms of their *domain* of values and their participation in the ruling or the dependent part.

Table 7-1 compares the terminology for the four levels of abstraction which we encounter in file and database design. We now formalize these definitions.

Domains The form and content of a tuple are governed by rules derived from its membership in a relation. With each attribute a specific *domain* is defined from which values can be selected. The number of values in a domain can be extremely large or can be quite limited. We consider domain semantics formally in Sec. 7-2.

The domain of an attribute for an invoice amount can range from millions of dollars to an equally large credit. The domain D_t of an attribute `temperature` in a tuple describing people may be {`97.0, 97.1, ..., 105.0`} and the domain D_c `color`{`black, brown, white, yellow, ruddy, red, purple`}. The sizes of the two sample domains D_t, D_c are $\#D_t = 71$ and $\#D_c = 7$.

Relations Tuples are formed by taking one value for each attribute out of the domain corresponding to this attribute. Representing the domains of the data element values by sets D_j, where a D_j is known for each of the attributes A_i for each position, allows the definition of a relation R with attributes A in its relation schema to be a subset of the cartesian product of the underlying domains $D(A_i)$

$$R\{A_1, A_2, \ldots, A_a\} \subset D(A_1) \times D(A_2) \times \ldots \times D(A_a) \qquad 7\text{-}1$$

and a tuple is one *row* of such a relation. The possible range of values of an attribute A_j is the domain D_j. Attributes of relations must have distinct names; in the model the order of the columns will be of no formal concern. More than one attribute of a relation can have the same domain.

Two identical domains in one tuple could occur in a relation describing an inventory of automobiles, where each car has two attribute columns listed with the domain `color`, one for the exterior and one for the interior of the vehicle. This permits a computation to match the interior color to the color of the exterior.

These identical domains have two different *roles*. Each role of a domain in a tuple implies a relationship and a distinct column. An attribute name within a relation identifies both domain and role and is unique. We do not define a formal syntax to relate domains and attribute names In the earlier example `age` and `spouse_age` are distinct attributes which have identical domains. We will always

Table 7-1 Terminology

Hardware	Software	Model	Semantics
Disk	File	Relation	Entity set
Documentation	Schema (see Chap. 8)	Relation-schema	Description
Block	Record	Tuple	Object
Address	Key	Ruling part	Object name
Content	Goal	Dependent part	Attributes
Byte number	Attribute name	Attribute	Attribute
Data format	Data type	Domain	Value range
Byte or word	Attribute value	Data element	Value

give unique names to attributes in a relation, so that any column in the model is identified by giving `Relation.attribute`. For each attribute of any relation R_p in the model the domain D_d will be known

$$R_p \,.\, A_i \rightarrow D_d \quad \text{where } i = 1, \ldots, a_p \qquad 7\text{-}2$$

Domains of attributes have to be identical for comparison or merging of attributes. We call such attributes *compatible*.

Domains for some attributes may appear to be identical when they are actually not. An example is an employee's home and business phone numbers. Not only are the two value sets distinct, but they also obey different rules for updating. The office phone number, possibly shared with many employees, changes when the employee moves to another office. The home phone number remains valid, since it is dependent on the individual employee only. This distinction may be best expressed through the use of distinct domains, rather than through different roles on the same domain.

The simple but formal definition of a relation permits the manipulation of the model. In the construction of view models we place further restrictions on the structure of the relations. These restrictions will assure that the intent of the views is preserved. In Secs. 7-1-2 to 7-3-6 we develop these restrictions. We will require that the number of attributes is fixed, that there are no duplicated tuples, that no attributes are themselves relations, and that there is no redundancy caused by awkward assignment of attributes to relations.

7-1-2 First Normal Form

When we manipulate the relations of a the view model we treat them as mathematical sets of tuples. This requires among other things that the attributes within each tuple are ordered and complete, and that the domains permit only simple values. *Simple values* cannot be decomposed into multiple values, and cannot themselves be sets or relations.

A feature of records in programs which therefore cannot be modeled in a single relation is a nested repeating data element or a nested repeating group of elements such as a subarray for `children` in Fig. 7-4. If a nested group is of bounded size, and preferably small (here 10), such a subset can be represented by a fixed number of attributes assigned to different roles over the same domain. Since groups may be smaller than the fixed maximum number, the domain has to include the value `null` or `undefined`.

More often we remove the nested group to another relation. In Fig. 7-4 this is done with the nests of supervised employees, the `no_of_supervisees` can range from 0 for the janitor to $n-1$ for the company president. A new relation `Supervision` is used to capture all the nested `supervisees` lists for all employees. The repetition of the supervising employee's name in the `super` field provides the linkage from the tuples in the new `Employee_1` relation, so that the information about who supervises whom is not lost.

The process of removing nested groups is called *normalization*, and the resulting relations are designated to be in *first normal form*. When relationships are expressed through common domains in separate relations rather than by being in the same tuples of one relation, the structure is less tightly bound. Searching the databases for a relationship takes more effort when binding is weaker, but flexibility increases. Some binding can be provided in the relation-schema. The schema of the `Employee_1` relation indicates that there exists a connected relation `Supervision`. Relations throughout the remainder of this chapter are in first normal form. Schemas are the subject of Chap. 8.

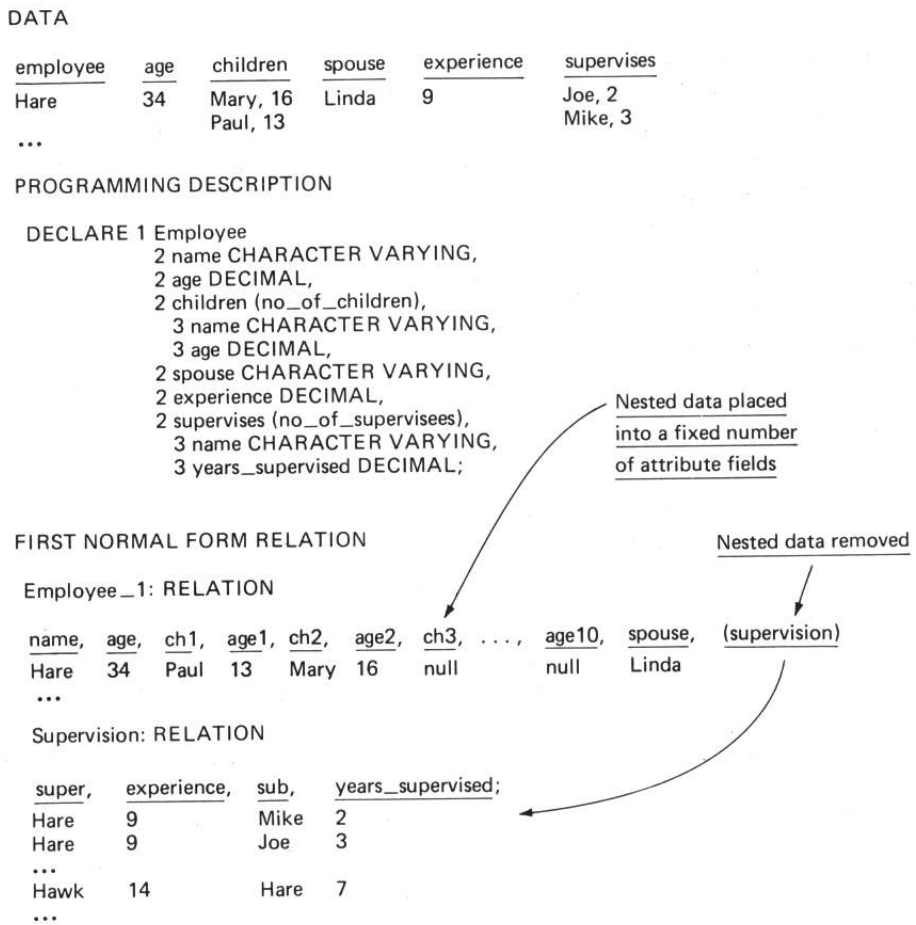


Figure 7-4 Relation in first normal form.

7-2 SEMANTICS OF RELATIONS

In order to build a model we have to define the internal structure of the relations and the relationships between the relations. In this section we present the underlying formalisms and in Sec. 7-3 we will employ them to create building blocks for database design.

7-2-1 Ruling and Dependent Parts

A tuple of a relation can be considered to be a partial description of an entity or a real-world object. Entities used in earlier examples were people, invoices, payments, and creditors.

Within one relation we distinguish the set of attributes which define the object described by the tuples - *the ruling part* - and those which provide data regarding the object - the *dependent part*.

In most our view models relations should have ruling parts which are unique, since we wish to have only one descriptive tuple per object. This immediately makes the tuples themselves unique. The ruling part should not contain redundant attributes; that is, it should not contain attributes, which when removed, would still leave a ruling part with the property of uniqueness. When an employee number is in the ruling part of a relation, the employee name is not also in the ruling part, since employee numbers are assigned uniquely. As a notational convenience in the examples that follow, we will in relation-schemas place the ruling part before the dependent part and separate them with the symbol " :⟩ ".

Example 7-2 Relation-schema for a automobile registry

```
Automobile:  RELATION
    state, license_number :⟩ make, model, year_manufactured, owner;
```

Since first-normal-form relations are also mathematical sets, the tuples of a relation *must* be unique. Given this constraint it is always possible to find a unique ruling part; in the worst case, it includes all the attributes and the dependent part is null.

Structure versus Meaning Finer distinctions among relationships can be made. A single concept, *dependent attribute*, is used to convey all the shades of meaning available through expressions such as "belongs to", "being", "married to", "possessed by", "purchased by", and "having". A model connection to another relation can describe the employees with whom a person works, the department, the position held, or the product produced. Knowledge of the semantic structure supplements in an essential manner the contents of the database itself. We treat the semantics in a simplified manner, discussing meaning only to the extent that the semantic relationships affect the structure of a view or database model.

Dependent part attributes of a relation depend on the ruling part of the relation for their definition and existence. Similarily we find that attributes in one, secondary relation may depend on attributes in another, primary relation. If the dependent attributes are in the ruling part of the secondary relation, the existence of tuples in that relation becomes dependent on attributes of the primary relation.

We will now consider some formal definitions and axioms about these dependencies. In many practical database design problems an intuitive understanding of dependencies is sufficient. This section may be skipped or skimmed lightly by readers who want to proceed to the building blocks for the structural model presented in Sec. 7-3.

7-2-2 Functional Dependencies

If the value of some attributes B is always determined by the value of other attributes A, then we say that B is functionally dependent on A. In our model the dependent part of a relation is functionally dependent on the ruling part.

A dependency is derived from reasoning about the data; it is not deducible from any given collection of data, although it can be disproved by testing a database which is factually correct.

The ruling part of example 7-2 has two attributes A_1, A_2. The concept of dependencies is the basis for establishing correctness in the models and the databases we design. A functional dependency FD is defined as follows:

Attributes B are functionally dependent on attributes A, or

$$FD(A_1, A_2, \ldots, A_h, \ldots, A_j) = B_1, B_2, \ldots, B_i, \ldots, B_k \qquad 7\text{-}3$$

if the value B_i in any tuple is always fully determined by the set of values $A_1, A_2, \ldots, A_h, \ldots, A_j$.

A_h, B_i are simple attributes as defined in Sec. 7-1-2.

To avoid redundancy in the ruling part, we require that no subset of the ruling part can be found which would also be an adequate ruling part:

$$\begin{aligned} &\text{If both} \quad FD(A_p, A_q, \ldots, A_r) = B_i \quad \text{and} \quad FD(A_1, A_2, \ldots, A_j) = B_i \\ &\qquad \text{where} \quad (A_p, A_q, \ldots, A_r) \subset (A_1, A_2, \ldots, A_j) \\ &\qquad \text{then use} \quad FD(A_p, A_q, \ldots, A_r) = B_i \end{aligned} \qquad 7\text{-}4$$

There may be more than one candidate ruling part, for an `Employee` perhaps an `employee_number` and {`Employee_name, employee_address`}. Both alternatives satisfy Eqs. 7-3 and 7-4. The choice depends on the view model.

Example 7-3 Redundancy in Dependencies

An example of a redundancy in ruling-part attributes can be found in a relation describing departments of a store. The ruling part contains both the name of the manager and the title of the department.

```
Departments:  RELATION
    manager_name, department_name :⟩ budget, no_of_personnel;
```

If we know that a department always has one manager even though one manager may direct more than one department, then FD_1(`department_name`) = `manager_name`.

The manager's name is functionally dependent on the department name, as are the other attributes (`budget, no_of_personnel`), and the `manager`'s name should be removed to the dependent part. In an `Employee` relation the name may be the ruling part. For the manager the `job = "Manager"` may appear in the dependent part.

7-2-3 Dependencies and View Models

When we construct a model, we may find functional dependencies which are redundant. A simple case of a redundant set of functional dependencies is given below:

$$FD_2(\texttt{employee}) = \texttt{spouse}$$
$$FD_3(\texttt{employee}) = \texttt{home\_address}$$
$$FD_4(\texttt{spouse}) = \texttt{home\_address}$$

This simple case can be resolved by striking out FD_3 or FD_4; the choice will be determined by the desire to have the most realistic view model for the user. A view model has to be kept small if functional dependencies are to be verified by comparision with real-world practice. The view model for the company psychiatrist may reject FD_3, but the payroll department will ignore FD_4.

We will add one more constraint to our definitions of functional dependencies in major view relations: *no reverse functional dependencies*. This rule holds only for any single view model but not for lexicons (Sec. 7-3-3) and may be violated in an integrated database model:

$$\begin{aligned}&\text{There exists no } FD, \text{ say } FD_q,\\ &\qquad \text{such that } FD_q(B_i) = \{A_p, \ldots, A_r\} \quad \text{where } i \in \{1, \ldots, k\}\end{aligned} \tag{7-5}$$

and A, B are defined as in Eqs. 7-3 and 7-4.

This means we do not recognize any dependencies from the dependent part on the ruling part within a view model. This rule also helps to avoid cases of circularity. A well-known but dull example of circularity is the set of dependencies

$$FD_5(\texttt{zip\_code}) = \texttt{city}$$
$$FD_6(\texttt{city, address}) = \texttt{zip\_code}$$

We resolve this issue in structural modeling again by having two view models, one for the mail delivery service (FD_5) and one for the senders of mail (FD_6).

The database model has to resolve differences among the view models; correct view models can always be integrated. Keeping view models simple helps attain correctness. Complex redundant functional dependencies occur when a composite attribute provides the ruling part of the relationship. Further normalization of relations, discussed below, will assist in the recognition of functional dependencies and avoidance of awkward structures.

7-2-4 Multivalued Dependencies

A *multivalued dependency* occurs when an attribute value determines a set of multiple values. Examples seen earlier were the sets of `children` and `supervisees` of an `employee` in Fig. 7-4(a). We can state for such sets of dependent attributes

$$\begin{aligned}&\text{If each set } \{B_{l,1}, \ldots, B_{l,s_l}\} \text{ is fully determined by the values } A_1, \ldots, A_j\\ &\text{then } MVD(A_1, \ldots, A_j) = \{B_{1,1}, B_{1,2}, \ldots, B_{1,s_1}\}, \{B_{2,1}, B_{2,2}, \ldots, B_{2,s_2}\}, \ldots\end{aligned} \tag{7-6}$$

The set $\{B_{1,1}, B_{1,2}, \ \ldots \ , B_{1,s_1}\}$, say, the `child`ren of the `employee` object defined by $A_1, \ldots, A_j$, is independent of the $\{B_{2,1}, B_{2,2}, \ldots, B_{2,s_2}\}$, the `supervisee`s, just as the `employee`'s `home_address` is independent of the `birthdate`.

In a model which includes any MVD, say MVD(`employee`) =`child`, all `child`ren should be represented for the `employee` whenever any `child` is, so that the data are complete. This will satisfy the MVD. If any `supervisee`s, or other $B_{l,r}$ are being represented the same rule of completeness holds.

Completeness is being further defined in dependency theory and is related to the universal instance assumption. This work is outside of the scope of this text.

If always only one entry exists in a set ($s_l = 1$), then $B_{l,1}$ or B_l is functionally dependent on the $(A_1, \ \ldots \ , A_j)$. Functional dependency is hence a special case of multivalued dependency.

$$\text{If} \qquad FD(A) = B_l \qquad \text{then} \qquad MVD(A) = B_l \qquad \text{7-7}$$

Since a FD is more restrictive, it is more powerful in database design than a MVD.

Dependencies within a tuple of a fully normalized relation should be FDs. We are concerned only with MVDs when there is no FD between the attributes. The attributes $B_{h,p}$ in multivalued dependencies may also be themselves composed of multiple values or groups. The `employee`'s `salary_history` will consist of subtuples with attributes `date_given, salary`.

Multivalued dependencies cannot be represented within a model using only normalized relations; a relation can only include any nested sets $B_{h,\ldots,s_h}$ by enumeration. Such a relation has to contain each legal combination. For i included sets this requires $s_1 \cdot s_2 \cdot \quad \ldots \quad \cdot s_i$ tuples. For the `Employee` with 3 `child`ren, 5 `supervisee`s, and 4 entries in the `salary_history` 60 tuples are implied. This representation contains obviously much redundancy and is not in first normal form. This representation is not at all clear since the dependencies leading to this representation are not explicit.

The structural model, and other models dealing with this issue, represent multivalued dependencies by an ownership connection between a main relation and nest relations for each multivalued dependency. A operation, the *natural join*, which combines the two relations, is described in Sec. 7-4-4 and permits creation of the enumerated format when it is needed.

Example 7-4 Organizing data with multivalued dependencies

As example we use the case where an employee `Joe` has multiple supervisors. Which supervisor is in charge depends on the job being worked on.Supervisor `Peter` may supervise `Joe` when doing `Maintenance` and supervisor `Paul` when `Joe` does `Development`. We have a relation-schema with the attributes {`supervisor, job, employee;`}. Neither `supervisor` nor `job` is a correct ruling part here; they are not neccessarily unique over the tuples. We want to state

`employee` :⟩ {`supervisor, job`};

Alternative single relation-schemas to store the data are

```
Supervision_job_list:  RELATION        Supervision_Employee_list:RELATION
   supervisor, job :⟩ employee;            supervisor, employee :⟩ job;
```

but these two views do not express the underlying MVD.

The set of dependent values need not be a nested group but may be any set of multiple values which are not independently functionally dependent on the ruling attribute. The problems of representation are illustrated in Example 7-4.

7-2-5 Rules for the Transformation of Relation Schemas

Given that we have been able to express all required functional and multivalued dependencies from a real world view in terms of simple and composite attributes, the relation-schemas can be manipulated. The first three transformations are known as *Armstrong's axioms*, and others are derived from these [Ullman[82]]. They represent formal rules of transformations which will, in general, be intuitively obvious during the design process.

Reflexivity Subsets of a set of attributes are functionally dependent on their full set. In terms of our definitions, there is

$$FD(A_1, A_2, \ldots, A_j) = A_k, A_m, \ldots, A_o \text{ for } k, m, \ldots, o \in \{1, 2, \ldots, j\} \qquad 7\text{-}8$$

This axiom defines the *trivial dependencies* which exist since in any relation tuples which agree in the values for $A_1, \quad \ldots \quad, A_j$ also agree in the values for any $A_k \| k \in \{1, \ldots, j\}$.

Reflexivity also applies to *MVD*s as defined in Eq. 7-6. Reflexivity of *FD*s also implies an *MVD* as demonstrated by Eq. 7-7.

Augmentation Ruling and dependent parts can be augmented with the same attributes.

$$\begin{aligned} \text{If} \quad & FD(A_1, A_2, \ldots, A_j) = B_1, B_2, \ldots, B_k \\ \text{then} \quad & FD(A_1, A_2, \ldots, A_j, C) = B_1, B_2, \ldots, B_k, C \end{aligned} \qquad 7\text{-}9$$

Augmentation also applies to *MVD*s.

Transitivity Dependencies are transitive, so that a chain of dependencies can be merged.

$$\begin{aligned} \text{If} \quad & FD(A_1, A_2, \ldots, A_j) = B_1, B_2, \ldots, B_k \\ \text{and} \quad & FD(B_1, B_2, \ldots, B_k) = C_1, C_2, \ldots, C_m \\ \text{then} \quad & FD(A_1, A_2, \ldots, A_j) = C_1, C_2, \ldots, C_m \end{aligned} \qquad 7\text{-}10$$

Transitivity as defined above applies also to *MVD*s but is actually more restrictive, because sets of attributes $B_{l,1} \ \ldots \ , B_{l,s_l}$ are not acceptable as ruling parts.

Union Dependent parts can be merged, so that we can combine relations that have the same ruling part.

$$\begin{aligned} \text{If} \quad & FD(A_1, A_2, \ldots, A_j) = B_1, B_2, \ldots, B_k \\ \text{and} \quad & FD(A_1, A_2, \ldots, A_j) = C_1, C_2, \ldots, C_m \\ \text{then} \quad & FD(A_1, A_2, \ldots, A_j) = B_1, B_2, \ldots, B_k, C_1, C_2, \ldots, C_m \end{aligned} \qquad 7\text{-}11$$

Dependent parts of *MVD*s cannot be merged, since the dependent attributes $B_{i,r}$ of an *MVD* exist independently of other *MVD*s.

We will in the view model always merge FDs having the same ruling part, so that only one relation will be needed to describe all simple facts about an entity. We applied this rule already when constructing tuples out of dependencies in Fig. 7-4.

The only time when we will have more than one relation with the same apparent ruling part $A_1, A_2, \ \ldots \ , A_j$ occurs when the domains $D(A_i)$ differ in some sense.

Example 7-5 Relations with differing ruling part domains

Consider the two relations

```
Employee:  RELATION
    employee_number :⟩ jobtitle, date_of_birth, ...;

Managers:  RELATION
    employee_number :⟩ jobtitle, date_of_birth, ..., stock_options;
```

In the `Managers` relation the domain for `employee_number` is restricted, but such restrictions are awkward to express in terms of domain rules. A better way of expressing the restrictions is to state that `jobtitle = "Manager"`.

To keep our examples clear we will always use identical attribute names A_i to imply identical domains, so that $D(A_i) = D(R_y.A_i) = D(R_z.A_i)$. The notation $D(R_x.A_i)$ to determine some D_d is based on Eq. 7-2.

Decomposition Dependent parts can be split, so that one relation can be transformed to several relations having the same ruling part.

$$\begin{array}{ll} \text{If} & FD(A_1, A_2, \ldots, A_j) = B_1, B_2, \ldots, B_k \\ \text{then} & FD(A_1, A_2, \ldots, A_j) = B_1, B_2, \ldots, B_i \\ \text{and} & FD(A_1, A_2, \ldots, A_j) = B_{i+1}, B_{i+2}, \ldots, B_k \end{array} \qquad 7\text{-}12$$

The selection of a dependent attribute B_d to become a member of the first, the second, or both relations is of course arbitrary.

View models are decomposed mainly to form building blocks for new relations. We also decompose relations to separate MVDs and FDs.

Some secondary axioms can be derived from these by straightforward processes. The union axiom (Eq. 7-11) is in fact derived using Eqs. 7-9 and 7-10 by augmenting both FDs, one with the A's and the other with the B's and applying the transitivity axiom to the results. The decomposition axiom (Eq. 7-12) is derived by selecting a subset of the B's, recognizing the FD of this subset due to reflexivity (Eq. 7-8) and applying transitivity (Eq. 7-10) to this FD and the original $FD(A) = B$.

These axioms can be used to rearrange the view models and to create the database model from the view models. They are sufficient to obtain a complete set of functional dependencies FD^+. This set is obtained by initializing FD^+ with the known set of FDs, and applying the three Armstrong axioms (Eqs. 7-8, 7-9, 7-10) in turn to FD^+ to create more entries into FD^+ for the next phase.

The process of obtaining all FD's may be costly, since the set FD^+ becomes rapidly large, but could be useful to understand a view model.

A nonredundant model in terms of functional dependencies can be obtained from a set of relation-schemas by

1 Decomposing all dependent parts to single attributes in order to create binary relation-schemas

2 Ordering the simplified relation-schemas by their ruling part

3 Removing all redundant FDs in this list by retaining only one, and only the simplest ruling part for any dependent attribute

4 Composing new relations where ruling parts match

We frequently use these axioms when trying to arrive at a semantically clear and effective model. The transformations required for this purpose are typically easy to prove correct using these axioms, and if they cannot be shown correct are likely to have an error.

Functional dependencies continue to exist among relations that have been decomposed during normalization, and may also be defined among relations that have been established separately. Multivalued dependencies will always be between distinct relations in a structural model.

7-2-6 Value and Reference Attributes

When we analyze the attributes for a model, we find two types of domains being used. Figure 7-5 provides examples of both types. Some attributes describe characteristic properties of an entity directly by assigning a value to the data element. A permanent domain definition, kept with the relation-schema, permits verification of values for new entries.

Examples of *value* attributes are the `height`, `weight`, and `age` of the employee. If the employee is dismissed and the employee's data tuple is deleted from the relation, the values will also disappear. No instance of `age = 34` may exist if `Hare` leaves the company.

Other attributes describe characteristics of an entity indirectly by reference to another entity. The domain used here is termed *reference domain* and the allowable values in the domain are defined by the existence of tuples in another relation. The domain definition can be changed by an update to the referenced relation. The referencing attribute and the referenced relation establish a *reference connection* in the model.

The employee tuple, for example, has an attribute which contains the name of the employee's department. The department will continue to exist when the employee leaves, and will even continue to exist, barring other action, even if all employees leave the department. We will in our models find distinct relations for the department, so that its existence is recorded in the database independently of assignments of employees.

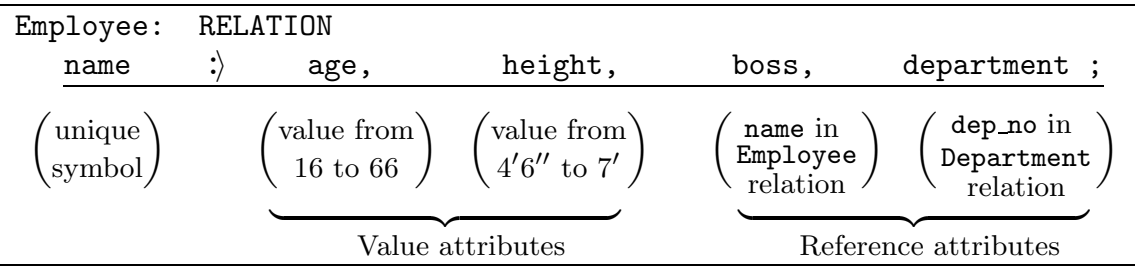


Figure 7-5 Domains for value and reference attributes.

7-3 BUILDING BLOCKS FOR MODELS

We now define relation types to be used in building view and database models. We will describe these informally, so that the function of the different types can be appreciated without reference to the formalisms presented in Sec. 7-2. We also describe conditions and normalizations which assure that all types of relations support a consistent update and deletion behavior. The relation types being introduced are summarized in Sec. 7-3-9.

7-3-1 Entity Relations

A relation which defines a set of independent objects or entities is termed an *entity relation*. The choice of entity types is a fundamental aspect of the design of a view model. Relations which define objects that are not entities (nest relations, lexicons, and associations) will be presented below and their usage will clarify the issues. Entities are typically objects that can be touched, counted, moved, purchased, or sold and that do not lose their identity during such manipulations. A change due to an update to a tuple of another relation will not require a change in an entity relation. An entity relation itself, when updated, may require further changes in subsidiary relations.

In Fig. 7-5 the referenced department was considered an entity. Even without any employees a specific department entity object will exist and be described by a tuple in the entity relation `Departments`. The tuples for the nested `children` of an `employee` will be removed when the employee quits. A new employee may require new entries to be made in the `Children` relation. We observe that from the point of view of the view model for company personnel, the children are not entities. The relation `Supervision` is not an entity relation either.

`Employee: RELATION`

name :⟩	birthdate,	height,	weight,	job,	dep_no,	health_no ;
Gerbil	1938	5'3"	173	Welder	38	854
Hare	1947	5'5"	160	Asst.welder	38	2650
Hawk	1921	5'6"	153	Manager	38	12
Hound	1956	6'2"	155	Trainee	38	1366
Havic	1938	5'4"	193	Welder	32	855

Figure 7-6 An entity relation.

The attributes may be basic values or references to tuples in other relations, as shown for the `Employee` of Fig. 7-6.

7-3-2 Nest Relations

First-normal form excludes repeating nests of attributes or of groups of attributes. During normalization we remove these nests to separate relations which we call *nest relations*. Tuples of nest relations have a ruling part which is the composite of the ruling part of the owning relation and an attribute value which is unique within the nest. This constraint permits recovery of the semantic linkage implied by the multivalued dependency from the owner tuples to the nest tuples. The structural model notes the dependency as an *ownership connection*. A nest relation derived from the example in Fig. 7-4 is shown in Fig. 7-7.

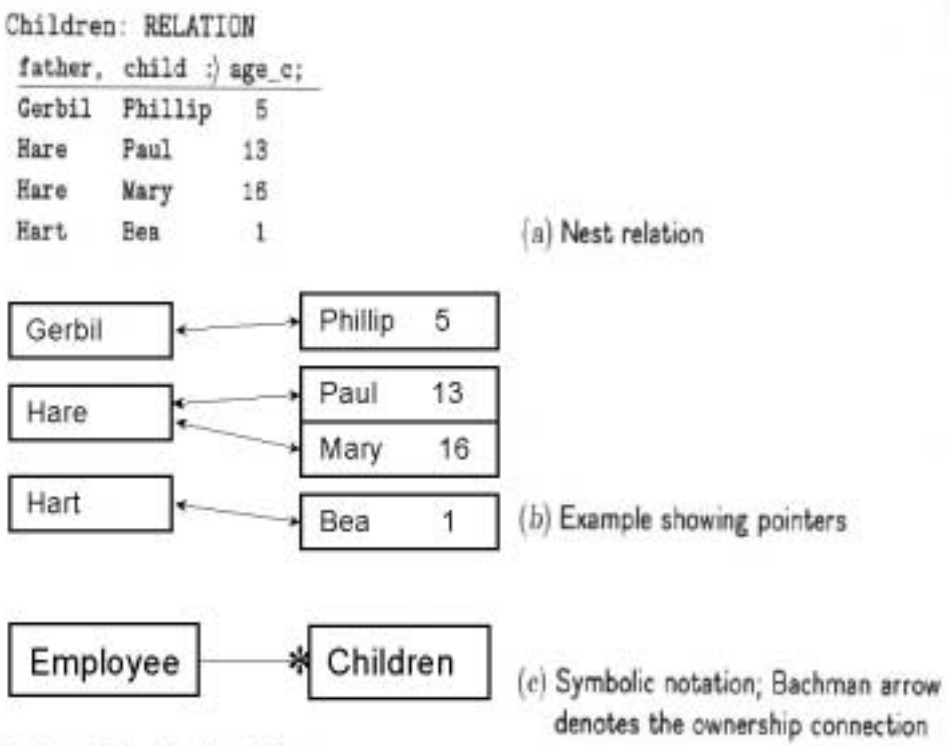


Figure 7-7 Nest relation.

The tuples of these nest relations are dependent on the owner tuples for their continued existence. This means that tuples can be inserted in a nest relation only if there is a matching owner tuple and that tuples in a nest relation have to be removed when the owner tuple is removed. This constraint will be formalized as the *ownership rules* in Sec. 7-3-6.

These rules derive from the $MVD(owner) = nest$, and assure that the actions are consistent with the actions taken if no normalization would have taken place or if the nest was expanded to a fixed set within the relation, as shown in Fig. 7-3.

The dependent part of a nest relation is ruled properly by the catenation of the parent's ruling part and the tuple-specific ruling attribute, for instance, in Fig. 7-7: `father,child`. Nest relations may, of course, be the parents of other nest relations, as shown in Fig. 7-8. This hierarchy also shows the transitivity of *MVD*s (Eq. 7-10).

Many levels of nesting can be encountered. The models may also require multiple nest relations at the same level. The `Employee` relation in Fig. 7-3 had two potential nest relations, `Children` and `Supervision`. The tuples from these two nests should not be merged (Eq. 7-11).

`Education: RELATION`

father,	child,	schooltype :⟩	school_name,	subject ;
Hare	Paul	High school	St. Mary	Science
Hare	Mary	High school	St. Mary	Social science
Hare	Mary	College	U.C. Berkeley	Political science
Hawk	Buzzy	...		
...	...			

Figure 7-8 Second-level nest relation.

7-3-3 Lexicons

In any relation the ruling and dependent parts should be clearly identified. We do often encounter relations which have more than one *candidate ruling part*, for instance

```
Department_detail:  RELATION
    dep_no ∨ dep_name :⟩ manager, location, budget, ...  ;
```

To resolve the ambiguities in the dependencies when there are multiple candidate ruling parts we remove redundant ruling parts of a relation R into a very specific type of a relation, the *lexicon* L.

After this process, for R,

$$FD(B_i) \neq (A_1, A_2, \ldots, A_j) \qquad 7\text{-}13$$

where B_i and A_i are defined as in Eq. 7-3. Now Eq. 7-5 is satisfied, except for the lexicon relation L itself.

A lexicon defines a one-to-one correspondence between two sets A and B by the binary relation $L : A \leftrightarrow B$. Here the dependent part is functionally dependent on the ruling part and the ruling part is functionally dependent on the dependent part, i.e., $FD(A) = B$ and $FD(B) = A$. In simple terms we can say that a lexicon implements equivalency.

Each tuple in a lexicon should reference a corresponding tuple in the entity relation and vice versa. We note in the view model this structural dependency using *reference connections* between the relations.

A lexicon relation, giving a listing of department names versus department numbers, is shown in Fig. 7-9. Since the ruling and the dependent part are symmetric, either attribute can be used as the ruling part in an implementation and the other part can then be uniquely determined.

The final entity relation which describes departments in detail (`Dep_detail`) has as its ruling part the `dep_name`, and the `dep_no` would be the key for the related lexicon. To get data about a department known by number, one would use the lexicon to obtain the `dep_name`, and then proceed to `Dep_detail`.

```
Departments:  RELATION
```

dep_no ⟨:⟩	dep_name ;
23	Bookkeeping
27	Auditing
31	Foundry
32	Forge
34	Stamping
38	Finishing
38	Assembly
50	Test
33	Quality control
24	Inside sales
25	Outside sales

Figure 7-9 A lexicon.

Lexicons occur frequently in operational databases, and recognition of lexicons during the view analysis simplifies the design process since ruling-part conflicts are avoided. Such conflicts also occur when distinct view models are integrated, and lexicons are often created during that process.

Another case for a lexicon appears in the `Employee` relation of Fig. 7-6. The relation has another candidate for the ruling part, namely, the `health_no`. In a view model for employee health care, this number may be the ruling part. A lexicon will connect the two view models.

Lexicons can be treated conceptually as a single attribute while designing a database model, and this approach can greatly reduce the number of possible alternatives of the model. This pseudo-attribute could be named `department_id`, deferring the decision on how the department is actually identified. Once a lexical pseudo-attribute is defined, the lexicon does not take part in the model manipulations until the file implementation is considered.

7-3-4 Second and Third Normal Form

In Fig. 7-4 we placed the years of experience of the employee in the `Supervision` relation, since this provided a meaningful separation of personal and professional data. The ruling part for this relation, using the earlier definitions, is the two attributes `super` and `sub`. The values of `years_supervised` are functionally dependent on this ruling part, but the `experience` is functionally dependent only on a subset of the ruling part, `super`. We note that the `experience` is hence redundantly recorded in every tuple beyond the first one for each supervising employee. To avoid this redundancy the attributes which depend on subsets of the ruling part are moved to a separate relation or added to another, appropriate relation.

In this case the `experience` attribute will be assigned to the `Employee` relation rather than to the `Supervision` to in order to achieve a nonredundant form as shown in Fig. 7-10. At other times a new relation may have to be defined, as shown in Example 7-5.

`Employee_2: RELATION`

name :⟩	age,	ch1,	age1,	...	spouse,	experience,	(Supervision) ;
Hare	34	Mary	16	...	Wendy	9	
...	...						

`Supervision: RELATION`

super,	sub :⟩	years_supervised,	(Employee_2.experience) ;
Hare	Mike	2	
Hare	Joe	3	
...	...	...	
Hawk	Hare	7	
...	...	...	

Figure 7-10 Second normal form relations based on Fig. 7-4.

Example 7-6 Creating a referenced entity relation

Given a relation-schema

```
Job_performance:  RELATION
    Employee_name, duty :⟩ hours_worked, bonus_pay, ...  ;
```

we may find that the `bonus_pay` is a factor which depends on the `duty` only, i.e., there is a FD(`duty`) =`bonus_pay`. A new relation will be established and the old relation will be transformed:

```
Job_performance_2:  RELATION
    employee_name, duty :⟩ hours_worked, ...  ;

                          ≻—— Duty_description:  RELATION
                                duty :⟩ bonus_pay;
```

The relation `Duty_description` collects this factor and any other information about the duties. This information is now retained even if no employee currently performs some `duty`. Without the `Duty_description` relation the `bonus_pay` must be entered with every change of `duty`.

The removal of functional dependencies on attributes which are subsets of the ruling part is called *normalization to second normal form*. It does not apply when the ruling part has only a single attribute. The dependent part of a relation in second normal form now contains only attributes that are functionally dependent on the entire ruling part. Relations in second normal form obey the rule stated as Eq. 7-4 for all subsets of the dependent part B.

New *referenced entity* relations may be established during this process; for instance, the relation `Duty_description` in Example 7-5. This type of relation also obeys all rules established for relations: since it is a set, redundant tuples will be eliminated and the number of tuples in it will be equal or less, often much less than the number of tuples in the primary or referencing relation. The referencing attribute will appear in both relations, and becomes the ruling part of the referenced relation.

Tuples which are being referenced by a tuple in a primary relation should not be deleted; otherwise processing errors may occur. The reference attribute will have a value suitable for accessing the referenced relation. Since the relation-schemas shown in our examples do not indicate the domain types, we do not know if `job` or `dep_no` reference other entity relations.

We may not be able to compute an employee's pay if the `duty` performed does not exist in the `Duty_description`.

A further transformation, to *third-normal-form*, removes any dependencies found among attributes within the dependent part. The dependent part of a relation may still contain attributes which are mutually dependent; or, formally, there may exist an $FD(B_q) = B_p$ or a $MVD(B_q) = B_p$, where B_i is defined as in Eqs. 7-3 and 7-4. New *referenced relations* are defined for such dependencies. The structural model will again note the relationship between the referencing and the referenced relation using a *reference connection*.

`Auto_section_a: RELATION`

assembly,	type :⟩	color,	colorcode,	... ;
750381	Body	Red	93471	
750381	Fender	Red	93471	
750381	Engine	Red	93471	
750381	Seatframe	White	93474	
750382	Body	White	93474	
750382	Fender	White	93474	
...	...			

(*a*) Relation with $FD(B_1) = B_2$

`Auto_section_b: RELATION`

assembly,	type :⟩	color, ... ;
750381	Body	Red
750381	Fender	Red
750381	Engine	Red
730381	Seatframe	White
730382	Body	White
730382	Fender	White
...	...	

`Colors: RELATION`

color :⟩	colorcode;
Red	93471
White	93474
Blue	93476

(*b*) Third-normal-form relations

Figure 7-11 Transformation to third-normal form.

In Fig. 7-11 the dependent part of a relation describes the colors of automobile subassemblies. Each `color` has a `colorcode` assigned. This redundancy can be removed by assigning a lexicon for `Colors`. In the final structure (*b*) only the `colors` relation has to be changed if the decision is made to change the `colorcode` of the color `red` for parts in the inventory. Such an update is correct and safe if the color code because functionally dependent on the color name.

If this functional dependency would not have been extracted and the model changed accordingly to Fig. 7-11*b*, this update would have to change many tuples in Fig. 7-11*a*. During the update transaction the file will be inconsistent. If the functional dependency were not recognized by the update transaction, some needed changes could be missed.

Redundancies in the dependent part are not restricted to one-to-one attribute relationships. An employee's job specification may contain many subsidiary attributes. Among these will be values which are functionally dependent on the job title. In order to remove this redundancy, a new referenced entity relation will be created which describes the properties of the job as shown in Fig. 7-12. A single reference attribute in the employee entity relation will then refer to the job relation by using the ruling part of the job tuples.

In the initial `People` relation (Fig. 7-3*d*), both `spouse` and `spouse_age` were included within the dependent part of the relation, but Fig. 7-3*b* indicated a relationship between these attributes: $FD(\texttt{spouse}) = \texttt{spouse\_age}$, just as $FD(\texttt{name}) = \texttt{age}$. This leads to a redundancy when a `spouse`, say `Mary`, is also entered in the `People` relation. Transformation to third normal form will eliminate this redundancy.

`Job_description: RELATION`

job :⟩	Education_required,	Experience_required;
Assistant welder	Bachelor's degree	2 years
Manager	High school diploma	12 years
Trainee	Bachelor's degree	None
Welder	Bachelor's degree	8 years
...		

Figure 7-12 An entity relation for reference.

7-3-5 Boyce-Codd-Normal Form

Normalization provided us with relations in *third normal form*. Because of the constraints we impose on our view models (Eqs. 7-4, 7-5, 7-12), we find that the view relations obtained also satisfy the conditions for *Boyce-Codd-normal form* (BCNF).

Boyce-Codd-normal form A relation is in Boyce-Codd-normal form if the only nontrivial dependencies are functional dependencies from the entire ruling part to each of the attributes in the dependent part.

The trivial dependencies to be ignored are due to reflexivity as defined in Eq. 7-8.

In the first normalization (Sec. 7-3-2) nests of attributes were removed, then lexicons (Sec. 7-3-3) were used to remove redundant candidate keys, and in the second and third normalization (Sec. 7-3-4) dependent attributes that did not depend only on all the ruling-part attributes were removed. Since Eq. 7-5 prohibits circularity in views we can now deal with BCNF relations. A sketch of dependencies within a relation reveals rapidly that a relation is in BCNF. The BCNF constraint is sufficient for all transformations we make in defining database models.

By transforming relations in a view model to relations in Boyce-Codd-normal form and establishing a number of connections between the relations we have described the structural features of a view of a database using simple building blocks. The normalized model reduces redundancy and makes explicit those functional dependencies which otherwise remain within the semantics of the tuple attribute description. The connections make the dependencies betwenn relations explicit. Both of these building blocks: BCNF relations and connections, are well defined and can be used for integration of the database and for implementing a database.

The referenced entity relations, nest relations, or lexicons created by the normalizations are constrained by their connections to the primary relation. These constraints will now be discussed.

Establishing Views The new `Employee_2` relation in Fig. 7-10, after normalization, contains data about the personal and the professional history of the employee. If it is desired to keep semantically distinct areas separate, we may employ two view models. Three relations are used for the data:

```
Personal_view:      Employee_personal: RELATION;.
Professional_view:  Employee_professional: RELATION;
Supervision: RELATION;.
```

The view models should avoid conflict and preserve meaning and understandability. Relations which contain data unrelated to each other except by common dependency on a ruling part reduce clarity; on the other hand, an excessive number of view models will make the eventual database model equally incomprehensible.

7-3-6 Connections

During the normalization process we encountered two types of connections. When entities are described independently within a view but are found to be related, further connections will be defined among the relations within a view. We will also describe now a third type of connection and review the functions of all of them.

Table 7-2 Connection Types in the Structural Model

Ownership connection	$\longrightarrow\!\!*$	from single to multiple owned tuples
Reference connection	$\succ\!\!-$	from multiple to single referenced tuples
Subset connection	$\longrightarrow\!\!\supset$	from single general to single subset tuples

Ownership connections We created ownership connections in Sec. 7-3-2, when describing the dependency of nest relations on the owner relations, and will encounter them again when describing associations between relations in Sec. 7-3-7. They are used to define a specific type of *MVD* (Eq. 7-6), namely, the case where the dependent attributes of each tuple themselves form similar sets or relations, rather than arbitrary sets. We refer to these sets as the owned set; a specific form was called a nest. The owned relation will have one owned set for each tuple of the owning relation. Ownership requires a match

$$OWNER.A_i = OWNED.A_i \text{ for } i = 1, \ldots, j \qquad 7\text{-}14$$

The owned relation will have one or more additional ruling part attributes $A_{j+1}, \ldots$ to identify individual members $r \in 0, \ \ldots \ , s$ of the owned sets. The size s of these owned sets ranges from $0 \rightarrow \#D(A_{j+1}) \cdot \ \ldots \ ; \ \#D$ is the cardinality of a domain as shown in Sec. 7-1-1. We can now state the rules for ownership connections:

Ownership rules The ruling part of the owned relation is the catenation of the ruling part of the owner relation and an attribute to distinguish individuals in the owned sets. A new tuple can be inserted into the owned relation only if there is a matching owner tuple in the owning relation. Deletion of an owner tuple implies deletion of its owned set.

Reference connections In the models we identified primary entity relations and referenced entity relations. Other relation types may also include references; we will use the term *primary relations* for all referencing relations. Primary relations contain attributes whose values reference tuples of the referenced relation. A tuple of the referenced entity relation provides a more detailed description for the entity or abstraction named by the referencing attribute of the primary relation. We expect to find tuples in a referenced relation to match any value assumed by the referencing attribute; absence of a referenced tuple constitutes an error. A referenced relation hence defines the domain for the referencing attribute. Referencing alternatives are sketched in Fig. 7-13.

Reference Rules: The ruling part of a referenced relation matches the referencing attribute of the primary or referencing relation. Tuples in referenced relations may not be removed while any reference exists. The removal of referencing tuples from the primary relation does not imply removal of the corresponding referenced tuple.

If these rules are not obeyed, vital information may not be available during processing, causing transactions to fail.

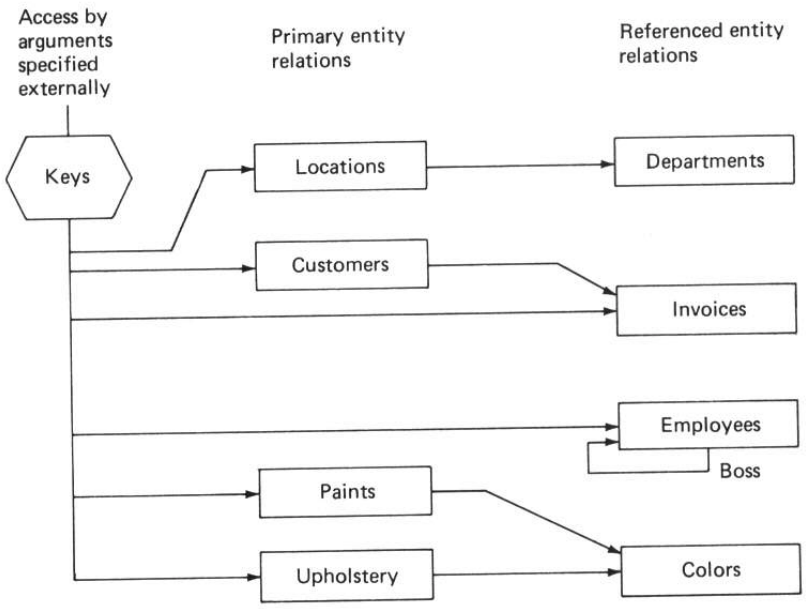


Figure 7-13 Referenced relations.

Referenced relations have to be maintained distinctly but cooperatively with the primary relations. A reference connection is based on an attribute of the primary relation, Data values being entered for that attribute must an be restricted by the domain defined by the tuples of the referenced relation. In Fig. 7-11*b* no `Mauve` fender is permitted in `Auto_section_b`.

Errors in the maintenance of referenced relations can also occur when files belonging to one database are due to different views and maintained by separate departments in an enterprise. An example where referenced relations are augmented without full consideration of the meaning of the relationship is shown in Fig. 7-14.

To avoid redundancy of `color` specifications in Fig. 7-11 a relation was created for the three colors in use at the factory. When it became desirable to specify also the paint type used, this relation was extended, but now an inappropriate type of paint is specified for the engine.

`Auto_section_c: RELATION`

assembly,	type	:⟩	color;
750381	Body		Red
750381	Fender		Red
750381	Engine		Red
750381	Seatframe		White

`Color_b: RELATION`

color	:⟩	colorcode,	painttype;
Red		93471	Acrylic
White		93474	Lacquer
Blue		93476	Acrylic

Figure 7-14 Improper extension of a referenced relation.

A lexicon is a special type of referenced relation. Because of the restriction of a one-to-one correspondence between ruling parts and dependent parts, either part of a lexicon can fulfill the ruling part function required in referenced relations.

Artificial Reference Tuples The existence of a value in the referencing attribute requires the existence of a corresponding referenced tuple. When a `null` value is permitted in the referencing domain, this fact has to be noted in the referenced relation.

This can happen if the color for a new automobile part is yet unknown or if a part, say a drive shaft, does not have a color.

To satisfy the model rules in a consistent manner, we allow the referenced relation to have a tuple with the value `null` in its ruling part, and appropriate values in the dependent domains. There are other conditions which may make the use of *artificial reference tuples* desirable. Some frequent types of key values for artificial reference tuples are shown below.

Table 7-3 Artificial Reference Tuples

Key	Condition	Function	Action
$*$	`null`	`unknown`	`"check_with_supervisor"`
Λ	`none`	`not applicable`	`ignore`
`?`	`any`	`placeholder`	`to be computed`

The last type of tuple (`any`) is useful when constructing query relations; the symbol "`?`" is to be replaced with appropriate matching entry for "`any`" color.

The effect of the use of the artificial reference tuple `null` is to defer error messages or inconsistency reports to a later stage, so that major database updates are not made impossible or delayed because of minor omissions in the input data. The extent to which this deferred binding of reference values is desirable depends on the operational and management philosophy of the database user.

Subset connections A subset connection is needed when we find relations with formally identical ruling parts, but differing attributes or domains.

Example 7-7 Decomposition to subset relations

In Fig. 7-3 we had a first-normal form relation

```
People:  RELATION
    name :⟩ age, spouse, spouse_age, job;
```

and had to remove the `spouse_age` to obtain a relation in Boyce-Codd-normal form. Those `spouse`s who are not employees are entered into a general relation, and attributes specific to employees form a subrelation. The subrelation is constructed by decomposition and the general relation by decomposition and a union where $D(\texttt{name}) = D(\texttt{spouse})$ and $D(\texttt{age}) = D(\texttt{spouse\_age})$ so that

```
All_people:RELATION          ⟶⊇          Employees:RELATION
   name :⟩ age;                              name :⟩ spouse, job;
```

A subset connection identifies the `Employee` tuples to belong to tuples of `All_people`. A change of `spouse` now no longer requires an update of `spouse_age`.

Subset relations occur frequently. We may want to collect attributes for managers or sales staff that are not needed for other employees, as was shown in Example 7-5. We may collect data on classes of vehicles such as cars, buses, and trucks. which have many commonalities and some differences.

Subset rules The ruling part of a subrelation matches the ruling part of its connected general relation. Every subset tuple depends on one general tuple. A general tuple may have no or one tuple in any connected subset relation.

The *generalization* of subclasses to larger classes is essential to data-processing; the recognition of individual subclasses and the collection of detailed data is important to give depth to the database and is often needed in specific views. The integrated database model has typically a much higher fraction of subrelations than the individual view models. During integration in Sec. 7-5 we will see how such subrelations are created.

7-3-7 Associative Relations

Finally we will consider relationships which contain data which depend on the interaction or association of entity relations. These relations will be termed *associative relations*. An associative relation has a ruling part composed of two or more attributes. The attributes relate each tuple in the associative relation to tuples in two or more owner relations. Figure 7-15 presents an associative relation. The connections from the owner relations to the association are ownership connections, and all the stated rules apply.

The attribute or attributes A_{j+1}, ... which distinguish individuals r in the owned subsets (see Eq. 7-14) are now taken from the other owner relations. Any other ruling part attributes will be redundant and removed during normalization. The number of owners is arbitrary, although during normalization we often find dependent attributes on associations owned by two relations. Associations of have as ruling part the union of the ruling part of their owners.

In general for an association

$$A_{association} = \bigcup_{owners} A_{owners} \qquad 7\text{-}15$$

where the sets A are the ruling part attributes as originally defined in Eq. 7-4 and used throughout.

There may again be $0 \rightarrow s$ tuples corresponding to one tuple in any of the owner relations. A classical example is the relationship between parts used by a manufacturing company and the suppliers of these parts, as shown in Fig. 7-16.

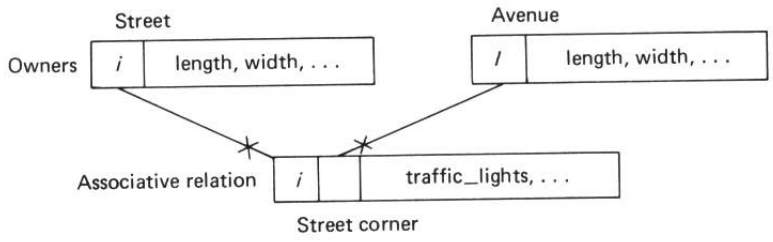


Figure 7-15 Associative relation.

Suppliers: RELATION

s_id :⟩	name ,	etc. ;
S1	Acme Screw Co.	...
S2	Bay Bolt Co.	
S3	Consolidated Nuts	
S4	Dzus Fasteners	
S5	Ilium Works	

Parts: RELATION

p_id :⟩	name,	size,	weight,	etc. ;
P2	Machine bolt	11x4	0.31	...
P3	Nut	4	0.07	
P4	Lock washer	4	0.04	
P5	Washer	4	0.02	

Supply: RELATION

s_id,	p_id :⟩	quantity ;
S1	P1	160
S1	P3	60
S2	P2	140
S2	P3	50
S2	P4	90
S4	P4	100

Figure 7-16 Associative relation with its two owners.

It is possible for associative relations to have no dependent part. This would be appropriate for a relation which describes the capability of a specific `Supplier` to deliver certain `Parts`. This relationship contains information, and must be explicitly modeled by a relation and two connections. Such a relation which associates suppliers which may be called upon to supply specific parts with the `Parts` relation is shown in Fig. 7-17.

The semantic difference between these two relations is not obvious from the connection structure and has to be described by attaching a definition to the relation. An association can also associate tuples within the same relation.

Possible_supplier: RELATION

s_id,	p_id :⟩ ;
S1	P1
S1	P2
S1	P3
S2	P2
S2	P3
S2	P4
S3	P3
S4	P3
S4	P4
S5	P3

Figure 7-17 Associative relation without a dependent part.

A simple case of an association within a relation is created if we were to keep both `streets` and `avenues` of Fig. 7-15 in one relation `Roads`. Another case occurs when an employee can have multiple supervisors. An association `Supervision_matrix` between `Employee` and supervisors in the `Employee` relation will permit description of this many-to-many relationship. We see that the ownership rules apply here too. An entry in the `Supervision_matrix` requires existence of the employee and the supervisor.

7-3-8 The View Model

We have considered up to now the relations and their connections one by one. The view model will of course have many relations and connections; values on the order of a dozen are seen in practice.

Any relation can participate in multiple connections, and hence have multiple constraining rules imposed on it. For instance, referenced relations are often shared by multiple primary relations, and the primary relations may in turn be entitity relations, nests, lexicons, associations, or other referenced entity relations.

A single-view model should, however, be clear to the participants in the design process. If it is not, it is unlikely that semantic errors will be recognized. Use of a graphic representation, with the connection symbols given in Sec. 7-3-6, reveals the structure of the model.

If a view model avoids redundancy, the only attributes that appear in more than one relation are those that define connections. The owner, the reference, and the general attribute(s) are repeated, respectively, in the owned, the referenced, and the subset relation. The number of attributes $\#a_H$ found in all $nrel$ relations is the number of all distinct attributes in all relations plus the number of attributes in connections.

$$\#a_H = \sum_{h=1}^{nrel} \#R_h.C = \# \left(\bigcup_{h=1}^{nrel} R_h.C \right) + \#a(\text{connections}) \qquad 7\text{-}16$$

where the set of attributes $C = A\&B$ is particular to each relation R_h.

We note also that all replicated attributes are ruling parts on one of the sides of the connections.

The definition of functional dependencies is useful to verify view models which have been intuitively constructed. The functional dependencies may in turn be verified by using available data. Although it is not possible to use data to *prove* the correctness of a functional dependency or of the view model, it is possible to find errors from existing data. Any inconsistencies that are found will be due either to errors in the data, say, different `age`s for the same employee; or to an incorrect perception of the real world: there is indeed a department with two managers, although we stated $FD($`department`$) =$ `manager`. In the latter case the decision has to be made whether the real world should be adjusted (the old department can be split into two departments, each with one manager) or the view model be changed (a new associative relation can be created with a ruling part composed of manager and department jointly).

It is tempting to normalize relations based on an analysis of the values of their attributes. It should be realized, however, that values in the database at a given moment in time may show relationships which are accidental and not based on true functional dependencies.

An observation made from Fig. 7-6, "All employees born in `1938` are `"Welders"`", should not be taken to be a functional dependency.

On the other hand, the main reason for the existence of databases is to support the search for new dependencies. If we find that all or many welders develop a certain type of disease, and we can eventually prevent this dependency, then the effort to develop and collect data for this database will have been very worthwhile. Such an analysis is the concern of information-retrieval processes and not part of the view model.

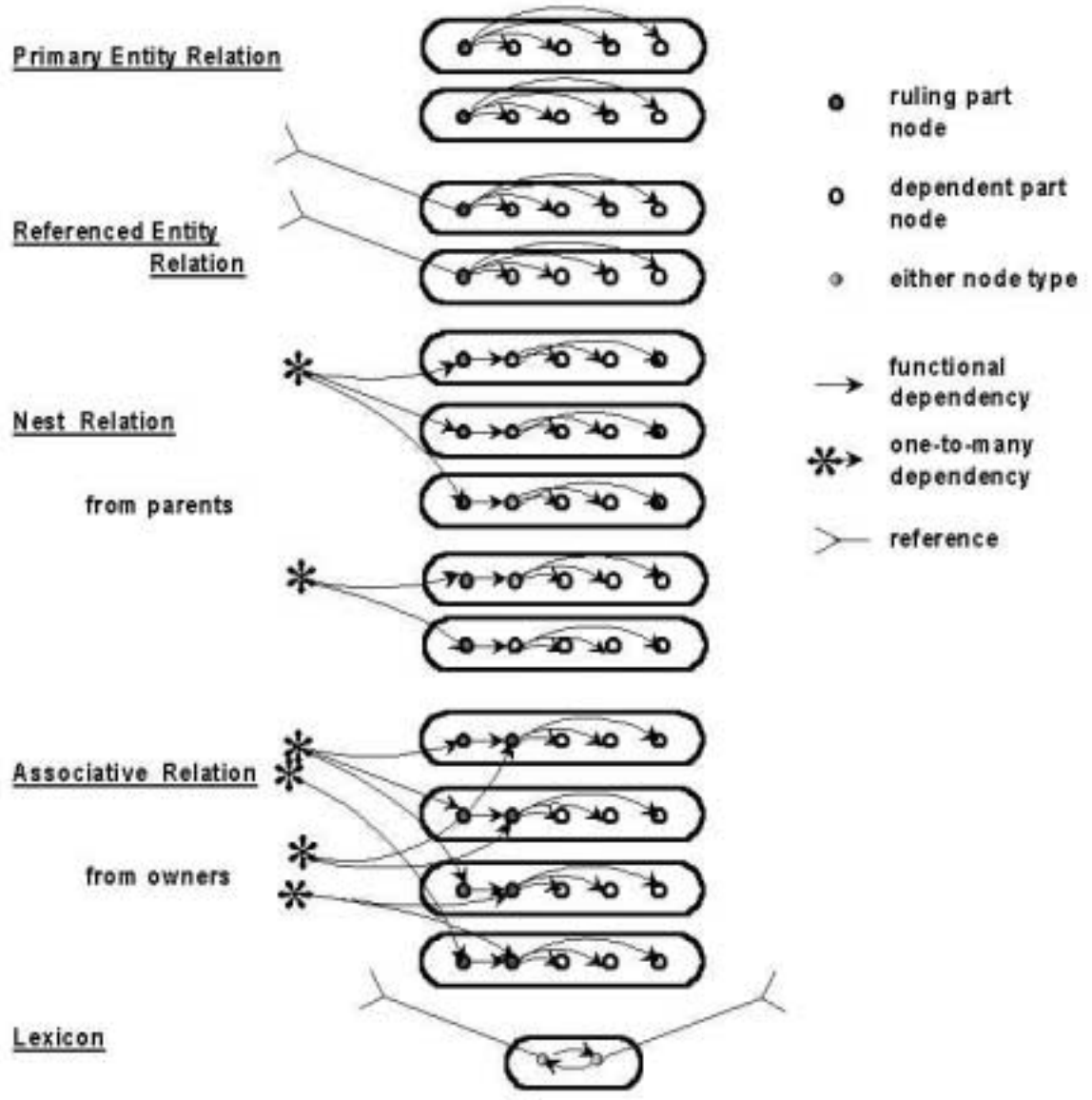


Figure 7-18 An erector set of relation types.

7-3-9 Summary

We have in this section discussed how semantic concepts regarding one particular view model or eventual database model can be described in terms of five relatively simple relation types and their subrelations:

Entity relations
Referenced entity relations
Nest relations
Associative relations
Lexicons

and three connection types

Reference connections
Ownership connections
Subset connections

The two concepts are duals of each other; a relation type defines the connections it participates in and a connection type defines the relation types it connects. Both concepts are useful, however, in dealing with models and their implementation, and both are found instantiated in database systems. Constraints between relations are more easily described using connections. Data elements and their semantics are better described using relations. A data-processing transaction is defined on both.

Figure 7-18 provides a graphical image of these types using the Bachman arrows for ownership connections, arrows with tails for reference connections, and plain arrows for functional dependencies within the tuples themselves. Use of these graphic symbols is extremely useful in presenting view models for review to the participants in a database design process. No tables or lists will illustrate the dependencies and hence the semantics of the view model as well.

When the view is approved, automatic processing of the information in the view models to arrive at a database model becomes feasible.

We will review the sematics of the relation types below, using the rules defined with the connections.

A *A primary entity relation:*
- **1** Not referenced within the view model.
- **2** The ruling part defines the entity.
- **3** The existence of tuples is determined externally, i.e., by the user.

B *A referenced entity relation:*
- **1** Referenced from within the view model.
- **2** The ruling part defines the entity and establishes domains for referencing attributes. A ruling part typically has a single attribute.
- **3** Existence of tuples is determined externally, but deletion is constrained by existing references.

C *A nest relation:*
- **1** Each tuple must have an owner tuple within the view model.
- **2** The ruling part defines one specific owner and the tuple within the owned set.
- **3** An owned nest can have zero or more tuples.

D *An associative relation of order n:*
- **1** Each tuple has n owners within the view model.
- **2** The ruling part defines each of n specific owners only.
- **3** One combination of owners can have 0 or 1 tuple in the association.

E *A lexicon:*
- **1** Referenced within the view model.
- **2** Either part can be the ruling part.
- **3** The existence of tuples is determined externally but deletion is constrained by existing references.
- **4** Its existence is transparent to the model.

F *Subrelations:*
- **1** Referenced from any general relation.
- **2** The ruling part matches the ruling part of the general relation.
- **3** The dependent part contains attributes which do not apply to non–matching tuples within the general relation.
- **4** Insertion requires existence of a matching tuple in the general relation.

The construction of each of these relation types obeys certain rules so that we can also describe these types using a Backus-Naur notation. Since such a description is essentially syntactic, some semantic constraints are added to the rules in quotes.

Table 7-4 Backus-Naur Description of Relation Types

DEFINITIONS:		
<value attribute>	::=	"a collection of data elements of one domain"
<reference attribute>	::=	"a collection of references to one relation"
<value attributes>	::=	<value attribute>
		| <value attributes>,<value attribute>
<attributes>	::=	<value attributes>— <reference attribute>
<value key>	::=	<value attributes> "unique instances"
<reference key>	::=	<reference attribute "unique instances"
<primary entity key>	::=	<value key> "not referenced"
<referenced entity key>	::=	<reference attribute> "referenced"
<nest key>	::=	<attributes> "unique within each owned set"
<dependent part>	::=	null — <attributes>
RULING PARTS:		
<entity ruling part>	::=	<primary entity key>
<referenced ruling part>	::=	<referenced entity key>
<qualification>	::=	<general ruling part> "in owner relation"
<nest ruling part>	::=	<qualification>,<nest key>
<assoc. ruling part>	::=	<entity ruling part>,<entity ruling part>
		| <assoc. ruling part>,<entity ruling part>
<general ruling part>	::=	<entity ruling part>— <referenced ruling part>
		| <nest ruling part>— <assoc. ruling part>
RELATIONS:		
<entity>	::=	<entity ruling part> :⟩ <dependent part>
<referenced entity>	::=	<referenced ruling part> :⟩ <dependent part>
<nest>	::=	<nest ruling part> :⟩ <dependent part>
<association>	::=	<assoc. ruling part> :⟩ <dependent part>
<lexicon>	::=	<key>⟨: <key>
<subrelation>	::=	<general ruling part> :⟩ <dependent part>

It is possible to create from these basic semantic types other forms of relations which satisfy special conditions. A synthesis of published material indicates that these five types, plus the concepts of subrelations, cover the semantic possibilities controlling the structure of databases in an economic and conceptually convenient manner. Combinations of ownership and reference connections, for instance, lead to four types of $n \cdot m$ relationships among two relations. Each of the four cases has distinctive semantics.

If we have constructed and normalized a view model in an economic fashion, that is, have kept no redundant attributes, then only attributes which implement reference, ownership, or subset linkages will be duplicated. Further minimization of redundancy can be achieved by minimizing the number of relations. The extent to which this is desirable depends on the further use of the view model. If the view model is to become a basis for extended applications using the database it may by transformed to a *database submodel*.

7-4 OPERATIONS ON RELATIONS

Procedures which derive new relations by selecting from or by merging other relations can be applied to the model to formulate alternate view models. Multiple view models can be combined into database models using such procedures. Derived relations introduce redundancies when added to the view model. When a view model has been manipulated, the result has to be reevaluated and renormalized to achieve the desired degree of redundancy.

Once a database model has been defined, database submodels can be extracted which define the portion of the database which can be accessed by some group of users. This might be a user group which has provided a view but now wishes to use elements not in its original view, or it can be a user group which is new to the database. A database submodel may include *derived relations*, which are defined by transformations of the database relations and its attributes.

Derived relations are also created when a database is manipulated to provide answers to queries. Examples are found in Sec. 9-2. Of the four conventional operations between sets:

Union	$\bigcup$	set of elements of 2 or more (s) sets
Intersection	$\bigcap$	set of matching elements from s sets
Set difference	$-$	remove elements that match another set
Cartesian product	$\times$	catenate all elements from s sets

the first three ($\bigcup$, $\bigcap$, $-$) are easily applied when the relations involved have identical scope, that is have compatible relation-schemas as defined in Eq. 7-17. A fourth operation ($\times$) creates a set of all combinations of arbitrary relations. These operations are defined in Sec. 7-4-1.

Operations which are specifically defined to deal with relations are:

Projection	$\prod$	reduce the number of attributes
Selection	$\subseteq$	select specific tuples
Join	$\bowtie$	combine attributes from two relations

These operations are defined in Secs. 7-4-2 to 7-4-5.

Transformations using these operations on relations are applicable in two situations: during the design phase we apply these operations to the relation-schema in order to transform the models of the database; after implementation we apply them the relations representing the database itself in order to compute results. When relation-schemas are transformed, we wish to make sure that any defined FD and MVDs are maintained by considering Eqs. 7-4 to 7-12.

We define the operations on relations $R(S,T)$, where S specifies the relation schema and hence the attributes and their domains, and T specifies the collection of tuples in the instantiation of the relation. We let n be the number of tuples or the *cardinality* of a relation and a be the number of attribute columns, so that we have n tuples of degree a in $R(S,T)$. The range of the number of tuples n for the results is indicated for each of the operations. For clarity we define the operations in the context of two relations; their generalization to an arbitrary number of relations is obvious where applicable.

7-4-1 Union, Intersection, Difference, and Cartesian Product

Basic are the three operations which combine relations with compatible relation-schemas, as illustrated in Fig. 7-19. Relation schemas of two relations R_1, R_2, both having a attributes, are compatible when their domains are identical

$$D(R_1.A_i) = D(R_2.A_i) \quad \text{for } i = 1, \ldots, a \tag{7-17}$$

Domains were defined in Eq. 7-2. Figure 7-20 depicts the union, intersection, and difference operations.

Taking the union of two relations which have compatible relation-schemas combines the two relations. This will increase the number of tuples, unless the two relations were identical.

$$\begin{aligned} R_3(S,T_3) &= R_1(S,T_1) \bigcup R_2(S,T_2) \\ \text{so that each tuple} \quad t \in T_3 &= \text{ some } \ t \in T_1 \ \text{ or } \ t \in T_2 \\ \text{Then} \qquad & \max(n_1, n_2) \le n_3 \le n_1 + n_2 \end{aligned} \tag{7-18}$$

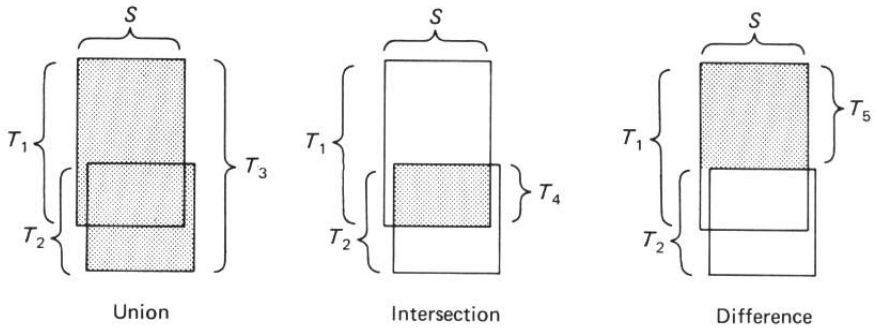


Figure 7-19 Set operations on tuples.

`Colors: RELATION`

color,	colorcode;
Red	93471
White	93474
Blue	93476

`Fashion: RELATION`

color,	colorcode;
Red	93471
Magenta	93479
Blue	93477

`Colors` $\bigcup$ `Fashion`

color,	colorcode;
Red	93471
White	93474
Blue	93476
Magenta	93479
Blue	93477

`Colors` $\bigcap$ `Fashion`

color,	colorcode;
Red	93471

`Colors` $-$ `Fashion`

color	colorcode;
White	93474
Blue	93476

Figure 7-20 Relational set operations.

The intersection of two relations will select all the identical tuples appearing in both relations

$$\begin{gathered} R_4(S,T_4) = R_1(S,T_1) \bigcap R_2(S,T_2) \\ \text{so that each tuple} \quad t \in T_4 = (\text{ some } t \in T_1 = \text{ some } t \in T_2) \\ \text{Then} \qquad 0 \le n_4 \le \min(n_1, n_2) \\ \text{also} \quad n_4 = n_1 + n_2 - n_3 \end{gathered} \tag{7-19}$$

The difference of two relations removes from the first relation all those tuples which exist also in the second relation.

$$\begin{gathered} R_5(S,T_5) = R_1(S,T_1) - R_2(S,T_2) \\ \text{so that each tuple} \quad t \in T_5 = \text{ some } t \in T_1 \text{ but } t \notin T_2 \\ \text{Then} \qquad 0 \le n_5 \le n_1 \\ \text{also} \quad n_5 = n_3 - n_2 = n_1 - n_4 \end{gathered} \tag{7-20}$$

Another definition of the difference operation (used in PRTV) does not require both relation-schemas to be the same. The tuples which have equivalent entries on the corresponding attributes are removed from relation R_1.

The cartesian product of two relations creates a set of all combinations of the tuples of the relations. Here a new relation-schema is created during the process.

$$\begin{gathered} R_6(S_6,T_6) = R_1(S_1,T_1) \times R_2(S_2,T_2) \\ \text{giving} \qquad S_6 = S_1 \ \& \ S_2 \\ \text{so that each tuple} \quad t \in T_6 = t_1 \& t_2 \quad \text{for all } t_1 \in T_1 \text{ and } t_2 \in T_2 \\ \text{Then} \qquad n_6 = n_1 n_2 \\ \text{and} \quad a_6 = a_1 + a_2 \end{gathered} \tag{7-21}$$

It is obvious that the product will be very large for all but the smallest R_1, R_2. We use this operation mainly as a basis for some further definitions.

A cartesian product of 15 `Streets` and 10 `Avenues` will provide the list of the 150 possible `Road_intersections` that could have traffic lights.

7-4-2 Projection

Projection reduces a relation $R(T,S)$ by limiting the attributes. The *projection* operation $\prod$ requires a list K containing the names $k_1, \ldots, k_l$ of the m attribute columns to be extracted. We recall that the attribute names uniquely identify the columns of the relations. We will first define an extract operation π which obtains a single value v from a single tuple t; v is the value corresponding to attribute A_p.

$$v = v_p = \pi\, t\star k \mid t = v_1\&\ldots\&v_p\&\ldots\&v_a\,,\ k = A_p \in S \qquad 7\text{-}22$$

Then, performing this extraction repeatedly, for l values on n tuples of a given relation with a attributes, we obtain the projected relation. It will have k attributes and $l \leq n$ tuples. The number of tuples is reduced if we have to strike out duplicate entries from the result relation; it will remain equal ($l = n$) if all attributes within the ruling part were included in the list K. In general,

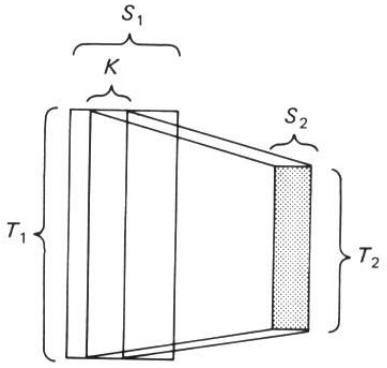
$$\begin{aligned} R_7(S_7,T_7) &= \prod R_1(S_1,T_1)\star K \mid K = \{k_1,\ldots,k_l\} \\ \text{so that the schema}\quad S_7 &= S_1 \bigcap K \\ \text{and each tuple}\quad t \in T_7 &= \pi t'\star k_1\&\ldots\&\pi t'\star k_l\,, \quad t' \in T_1 \\ \text{Then}\qquad a_7 &= l \leq a_1,\ n_7 \leq n_1 \end{aligned} \qquad 7\text{-}23$$

Projection is used during the decomposition of a relation-schema. For instance, to create referenced entity relation the source relation is projected on the reference attribute and attributes dependent on it instead of on the ruling part. The source relation is then projected to obtain a primary relation by removing these internally dependent attributes. We can now describe the transformations applied in Fig. 7-11 from (a) to (b) as $\prod$ `R`$\star$`K`, for example,

```
Colors = ∏ Auto_section_a⋆(color,colorcode)
Auto_section_b = ∏ Auto_section_a⋆(assembly,type,color)
```

The `Colors` relation in Fig. 7-11 shows how the number of tuples is reduced in order to satisfy set constraints once columns with ruling parts are excluded.

If tuples of a relation represent instances or tokens of entities, the projection by one attribute represents the categories or types for these entities. If a relation is a text, with each word being a tuple, a projection over this single attribute is the list of unique words. This concept is applied in Fig. 7-21 to the sample of the Chanson de Roland of Fig. 3-14.



$\mathcal{P}$ (k OF concordance_sample)=

soel
soels
soinst
sol
solent
solente
solenz
solor
solur
soluset
sorment
sort
sos
sous

Figure 7-21 Projection.

7-4-3 Selection of Tuples

For the set and the projection operations the actions were expressed in terms of the relation-schemas S; the same action is performed for all tuples T of the relation $R(S,T)$. The selection operation $\subseteq$ permits the creation of a relation having the same relation-schema containing a subset of the tuples based on the values of their attributes.

For selection the tuples t are selected one by one, according to an expression list L containing l selection expressions of the form $(A \ominus constant)$, where the *constant* is some value from the domain of A. Values matching A are extracted as defined in Eq. 7-22. In general a tuple will be selected if any expression in L is true, so that the expressions are combined combined using *or*. The definition of selection is now

$$\begin{aligned} R_8(S,T_8) &= \underset{L}{\subseteq} R_1(S,T_1) \mid L = \{(A_p \ominus_1 c_1) \vee \ldots \vee (A_r \ominus_l c_l)\} \\ \text{so that} \quad t \in T_8 &= \{(\pi\, t' \star A_p \ominus_1 c_1) \vee \ldots \vee (\pi\, t' \star A_r \ominus_l c_l)\}\,,\ t' \in T_1 \\ &\quad \text{with} \quad c_1 \in D(S.A_p), \ldots, c_l \in D(S.A_r) \\ \text{Then} \quad & n_8 \leq n_1 \end{aligned} \qquad \text{7-24}$$

The selection expressions in L may also be combined using other boolean functions than $\vee$; $\wedge$ and $\neg$ are common. The most common form of test in a selection expression is of course equality ($\ominus$ ="=").

In the construction of view models selection is required when the model is partioned in new ways; for instance, the tuples for `job = "Manager"` are selected to form the `Manager` subrelation.

This operation has its most important function in transactions. The retrieval of specific information from a database is best formulated in terms of projection of a list K of attributes to be retrieved and the list L of attributes and matching values which specify the conditions for selection of tuples, as shown in Table 9-1.

Selection can produce relations with none, one, or a set of tuples.

In program examples we write a request in the form

```
Selection_result = ⊆ R⋆L
```

Example 7-8 Selection of an employee category

To obtain the names and birthdates of the welders of Fig. 7-6 we create a result relation containing three tuples as follows:

```
Retrieve:  RELATION
   = ⊆ Employee⋆(job = "Welder"  ∨ job = "Asst.welder")
```

name :⟩	birthdate,	height,	weight,	job,	dep_no,	health_no ;
Gerbil	1938	5'3"	173	Welder	38	854
Hare	1947	5'5"	160	Asst.welder	38	2650
Havic	1938	5'4"	193	Welder	32	855

Selection and projection can reduce the number of tuples. If in Example 7-8 only the `birthdate` for the welders was wanted and the ruling part, here `name`, is omitted from the result, then the redundant tuple with `1938` would not show, leaving the impression that only two welders are available.

7-4-4 Join

The *join* operation $\bowtie$ composes a new relation which combines the data elements from two relations. A join can be defined as a cartesian product of two relations followed by a selection creating a subset (Eqs. 7-21 and 7-24). The subset consists of those tuples which have identical or matching values for the *join attribute*.

A join is sufficiently important and frequent that it warrants its own definition. Each relation R_1, R_2 being joined will have a join attribute J. To permit the comparison $\ominus$ the attributes $R_i.J$ have to be compatible, just as the entire relation-schemas had to be compatible for the set operations (Eq. 7-17).

$$
\begin{aligned}
R_9(S_9, T_9) = \; & R_1(S_1, T_1) \underset{J_1 \ominus J_2}{\bowtie} R_2(S_2, T_2) \\
\text{giving} \qquad & S_9 = S_1 \;\&\; S_2 \\
\text{so that each tuple} \quad & t \in T_9 = t_1 \in T_1 \& t_2 \in T_2,\; \pi\, t_{1\star} J_1 \ominus \pi\, t_{2\star} J_2 \\
\text{Then} \quad & 0 \le n_9 \le n_1 n_2 \;\text{ and }\; a_9 = a_1 + a_2
\end{aligned}
\qquad \text{7-25}
$$

The extract function π used is as defined in Eq. 7-22, but may be extended to include cases where J consists of multiple attributes. In our program examples we specify a join based on an equal comparison ($\ominus$ = `"="`) as follows:

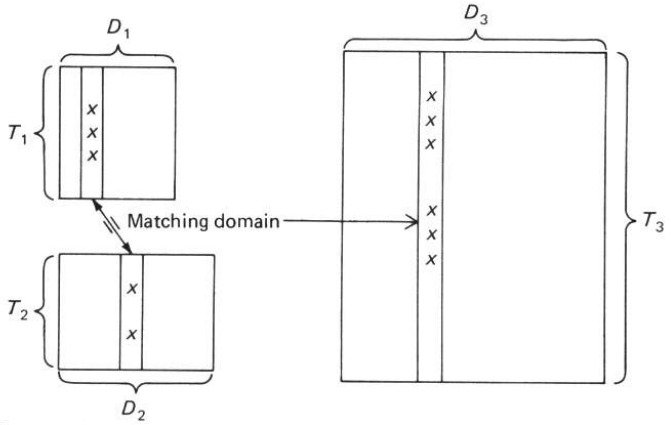
```
Join_result = R_1.J_1 ⋈ R_2.J_2
```

Pilots: RELATION

name,	lic,	duty;
Abe	727	on
Bob	737	on
Co	707	ret
Dee	727	on
Gay	767	tr.
Fil	737	on
Hap	727	off

Planes: RELATION

no,	type,	status;
101	727	ready
102	727	hold
103	737	ready
104	737	maint.
105	737	ready
106	767	order



Fly: RELATION = Pilots.lic ⋈ Planes.type

name,	type,	status,	no,	lic,	duty;
Abe	727	ready	101	727	on
Abe	727	hold	102	727	on
Bob	737	ready	103	737	on
Bob	737	maint.	104	737	on
Bob	737	ready	105	737	on
Dee	727	ready	101	727	on
Dee	727	hold	102	727	on
Fil	737	ready	103	737	on
Fil	737	maint.	104	737	on
Fil	737	ready	105	737	on
Gay	767	order	106	767	tr.
Hap	727	ready	101	727	off
Hap	727	hold	102	727	off

Figure 7-22 Join.

All attributes of the two relations are included in the resulting tuples. We note that if a tuple matches multiple entries in the other relation, multiple tuples are placed into the result, and if several entries in both relations match each other, their cartesian product appears in the join.

The join comes in various flavors. The most common join is the *equi-join* shown, with $\ominus$ = `"="`. Here all result tuples are formed by combining input tuples which have identical values in the attribute columns identified by J. The columns from the two relations involved in the equi-join are equal and one set can be omitted from the result relation, making this a *natural join*. If the number of attributes in J is j, the relation-schema for a natural join $R_9 = R_1 \underset{J_1=J_2}{\bowtie} R_2$ will be

$$\begin{aligned} S_9 &= S_1 \ \&S_2 - J \\ a_9 &= a_1 + a_2 - j \end{aligned} \qquad \text{7-26}$$

The equi-join is the join intended when no further specifications are given. A symbolic presentation of a join is shown in Fig. 7-22.

Joins are also implemented which use for $\ominus$ $r_1.j \neq, >, < r_2.j$. Conceptually arbitrary functions might be used for $\ominus$, so that the constraint of compatible domains for J might be relaxed. Such extensions have not been explored.

The implementation of a join rarely follows the steps implied by the definition. Generation of the cartesion product is avoided. A simple scheme is the *inner-outer-loop join*. For each tuple in R_1 all tuples of R_2 are inspected and if the condition $R_1.J \ominus R_2.J$ is true an output tuple is constructed. Other and better methods will be found in Sec. 9-3-3.

`Parts_skill_required: RELATION`

assembly,	type,	p_id :⟩	no_req,	skill_req ;
750381	Body	P1	10	Machinist
750381	Body	P2	12	Machinist
750381	Body	P4	22	Welder
750381	Fender	P1	26	Machinist
750381	Fender	P3	26	Welder

`Supply: RELATION`

s_id,	p_id :⟩	quantity ;
see Fig. 7-16		

`Parts_assembly: RELATION = Parts_skill_required.p_id` ⋈ `Supply.p_id`

assembly,	type,	p_id,	s_id :⟩	no_req,	skill_req,	#7;
750381	Body	P1	S1	10	Machinist	160
750381	Body	P2	S2	12	Machinist	140
750381	Body	P4	S2	22	Welder	90
750381	Body	P4	S4	22	Welder	100
750381	Fender	P1	S1	26	Machinist	160
750381	Fender	P3	S1	26	Welder	60
750381	Fender	P3	S2	26	Welder	50

Figure 7-23 Natural join.

A natural join is shown in Fig. 7-23. The join attribute is called `p_id` in both relations and the domain is obviously identical. While deleting the redundant join attribute we will rearrange the attributes in accordance with our ruling :⟩ dependent part conventions.

Computed Join In Eq. 7-25 we defined the join for an arbitrary computation $\ominus$. A sample join operation, based on a complex comparison, which uses the same relations is shown in Fig. 7-24. The objective is to obtain a list of production combinations for which the part `Supply.p_id` in stock is less than ten times the `assembly,type` requirement.

```
Supply:  RELATION  see Fig. 7-16
Parts_skill_required:  RELATION see Fig. 7-23
Low_stock:  RELATION = Parts_skill_required ⋈ Supply
             |  Parts_skill_required.p_id = Supply.p_id,
                Parts_skill_required.no_req * 10 > Supply.quantity;
```

assembly,	type,	p_id,	no_req,	skill_req,	s_id,	p_id',	quantity ;
750381	Body	P4	22	Welder	S2	P4	90
750381	Body	P4	22	Welder	S4	P4	100
750381	Fender	P1	26	Machinist	S1	P1	160
750381	Fender	P3	26	Welder	S1	P3	60
750381	Fender	P3	26	Welder	S2	P3	50

Figure 7-24 Computed join.

To avoid an excess of meaningless attributes in the result of a join, it is often desirable to carry out projections on the relations before joining them. It is also desirable to eliminate common attributes of the two relations which are not useful to the join. Such attributes do not exist if the relations of the database do not contain redundant attributes, as defined in Sec. 7-3-8. The joining of relations with redundant common attributes can at best reveal an inconsistency in the data (see Exercise 7-8).

7-4-5 Transforming the Database

The operations we described in Secs. 7-4-1 to 7-4-4 apply to any relation. We will now consider how these operations affect the relation types and connections of a database described by a structural model. Connections represents knowledge about dependencies among relations. If two relations are to be joined, and there is only one connection between the relations, the default condition is an equi-join along the connection. Now the required arguments for a structural equi-join operation are only the two relations.

The join of Example 7-9 is furthermore a join where no information has been lost, since all tuples from the original two relations are represented. This is true if the projections on the corresponding attributes in both relations are equal.

Example 7-9 Join along a connection

Given from Fig. 7-11 the relations and connection

```
Auto_section_b:  RELATION                    Colors:  RELATION
  assembly,  type  :⟩  color;                  color  :⟩  colorcode;

Color_specification:  CONNECTION
  Auto_section_b.color ⟩——  Colors.color;
```

the join operation on the relations

`Auto_section_a = Auto_section_b ⋈ Colors;`

will re-create the original unnormalized relation.

In Fig. 7-16 we had an association `Supply` of `Suppliers` and `Parts`. Joining

`Suppliers ⋈ Supply` or `Parts ⋈ Supply`

along the ownership connections of the association does not retain all the original information of the owners, since suppliers `S3` and `S5` and parts `P5` are not represented in the tuples of resulting relation. A crossproduct of `Suppliers` and `Parts` would keep data from all tuples; the association `Supply` contains the information about the current subset of the cartesian product represented in the `Supply` inventory.

A join does not introduce new, external information into the result. Data fields which are now replicated in multiple tuples are redundant. In Fig. 7-23 we created `Parts_assembly` by joining `Parts_skill_required` and `Supply`, on `p_id` giving information on the state of the inventory for required parts. There was no connection. Note that the result is not in second-normal form, and further, that the meaning of the data in the dependent part is not obvious. Joins along connections will in general have obvious results.

The useful information in Fig. 7-23 is the combination of `(assembly,type)`, `p_id`, and `s_id`, which informs us for each part which `Suppliers` are represented in the inventory, for each assembly and type. In bills-of-material processing a join of this type is called a *parts-explosion* because of the volume of detail it produces.

Joining Structural Relation Types We consider now how a join applies to the six relation types established in Sec. 7-3-9, since the semantic features of these relation types determine the semantics of the result relation. Since joins depend on correspondence of attributes which are defined in the view model, the results have predictable characteristics.

An *entity relation* when joined with another entity relation becomes an association, as seen with the street intersections of Fig. 7-15. The join only produces a list of all possible intersections; actual road crossings or crossings having traffic lights are a subset of the join result. If such a subset is specified, the membership in the subset association carries real information and cannot be omitted from the model. If a user's original data specification lists all actual `Road_crossings`, this set may be much smaller than the result of the join, but we can obtain a list of `Streets` and `Avenues` by projection of the attributes `street` and `avenue` in `Road_crossings`.

Lossless Joins When we manipulate a view model we may use projections to decompose relations, but do not wish to lose information. Loss occurs when subsequent joins cannot restore the original relation. The first requirement is of course that all attributes are represented in the result, i.e., the relation-schemas of the projections include all attributes from the original relation-schema. We also collect information to establish connections among the result relations. If the original relation includes attributes dependent on all of the ruling part, one of the projected relations will require this ruling part and have the same cardinality as the original relation. Then, a join along the connection will recover all the original data; such a join is defined to be a *lossless join*. Tuples subsequently added to the separate relations, say a new `color` not yet used, will not appear in the result. For a lossless join

$$\left(\prod R(S,T)\star(K\bigcup J)\right) \underset{J}{\bowtie} \left(\prod R(S,T)\star((S-K)\bigcup J)\right) = R(S,T) \qquad 7\text{-}27$$

A lossless join is guaranteed if both projections are in Boyce-Codd-normal form and J specifies the connection.

If these constraints are not followed when relations are obtained by projection and later joined again, data may be lost. When a data model is being manipulated, we are concerned that any join used to reconstruct the source specifications is loss-free; if it is, the source relations can be deleted to avoid redundancy.

When join operations are used on the database during information retrieval any compatible attributes can be used as the join attributes J. In the general case not all tuples from both relations appear in the result. In most cases this set of data presents the desired information, as shown in Example 7-10.

Example 7-10 Information loss in a join

We wish to find which `Employee` from Fig. 7-6 has the skills (`Mechanic, Welder`) to work on the assemblies given in `Parts_skill_required` of Fig. 7-23. The result is projected to retain only the attributes wanted.

```
Print_out:RELATION = ∏
     ( Parts_skill_required.skill_req ⋈ Employee.job )⋆(name, job)
```

Two employees (`Gerbil Welder` and `Havic Welder`) will appear in the result. The fact that no `Mechanic` is available is shown only implicitly.

To deal with the problem of losing information an extension of the join, the *outer-join*, has been defined [Codd[79]]. In the result will be at least one instance of each participating tuple. In the tuples that are not matched by J the missing attributes fields are filled with `null`.

Example 7-11 Outer-join

```
Print_out_all:  RELATION= ∏
     ( Parts_skill_required.skill_req ⟕ Employee.job )⋆(name, job)
```

name,	job ;
Gerbil	Welder
Havic	Welder
null	Machinist

Joins along Connections In the beginning of this section we considered joining structurally independent relations. We now review the effect of joins along connections in the structural model. While all relations in the model are in Boyce-Codd-normal form, the relations which result from joins will rarely be in Boyce-Codd-normal form.

When an entity relation is joined with its owned relation, the result will be the size of the owned relation. Tuples from the owning relation are represented only if their owned set ≥ 1.

A join with a referenced relation will increase the number of attributes, but the result has as many tuples as the entity relation. If the referenced relation is a *lexicon* the result will have one more attribute than the referenced relation. Lexicons can also be used to substitute attributes with no loss of information.

Joins between *nest relations* belonging to the same owner relation generate new nest relations which are not in second normal form, since they share at least one of the ruling-part attributes.

Joins between *associative relations* or between associative and other entity relations will generate new associative relations.

Joins between subrelations and their general relations will merely increase the number of attributes for the tuples in the subrelations. The difference of the general relation and the new relation can be used to provide an alternate view.

```
Hierarchical_view: Employees: RELATION  ⟜⟝ Managers: RELATION;
Partitioned_view:  Non_managers: RELATION ⋃  Managers: RELATION;
```

7-4-6 Sequences

In many view models the tuples in the relations are considered to be ordered. Such an order implies restrictions on the structure and operations but can also provide some benefits. We define a relation to be a *sequence* if it is ordered according to its ruling part. For numeric domains we assume ascending numerical order. Otherwise the collating sequence of the character representation provides an ordering; see, for instance, the ASCII table (14-1).

A sequential lexicon can provide sequential access to a general relation, or to a sequence which has the ruling part sequenced differently because of another attribute ordering, i.e., `child,father` :⟩ versus `father,child` :⟩ .

A number of new operations are applicable to sequences and other operations require new definitions. For instance, adding a tuple to a sequence implies insertion between specific tuples rather than the simple union which is adequate for relations.

For selection we can now specify the

`FIRST, LAST, and TUPLE(i)`

The discussions which follows are based on relations in their general form. If a sequence is required in the model, a lexicon can be used to describe the ordering requirements without invalidating the construction of the core of the model. Ordered attributes implemented through a lexicon have been termed *synthetic keys*.

7-4-7 Queries

The operations presented in Sec. 7-4 are combined to query the database content. A wide range of implementation approaches is possible for query processing. For actual data retrieval issues of ease of use and generality become important. These topics occupy most of Chap. 9.

Queries as Relations Projection and selection expressions can be combined and excecuted similar to a join when formulating a query. A query with selections and projection can be expressed in the form of a relation where the relation-schema is based on the list K and the tuples represent the selection list L. An artificial domain entry "`?`" is now used to represent the notion `any` to indicate that a data element value or set of values is to be obtained.

Example 7-12 Query in relation format

Selection is performed now using joins and projections applied to the query and data relations. A query for `name, birthdate` on `Employee` who can weld becomes

```
Age_of_welders:  QUERY  Employee
```

name,	birthdate,	job ;
?	?	Welder
?	?	Asst.welder

For general retrieval from a database selection expressions can include attributes from multiple relations.

Programmed Queries When the programming language used does not handle general sets of records, sequences become important. The information obtained from the databases has to be handled one record at a time, using the operations described in Sec. 7-4-6. It may be possible to specify that we wish to loop through all the tuples in sequence. Within the context of a current position within the sequence we can also reference the

`PRIOR, CURRENT, and NEXT tuple.`

The concept of *currency* allows the specification of tuple-by-tuple algebraic procedures which would not be applicable to sets in general.

Some algebraic operations for tuple-by-tuple functions can also be applied to unsequenced relations, but the required generalization is not always obvious. In many situations encountered in practice, the conceptual view model is based on *natural* sequences, and users tend to formulate queries oriented toward these sequential models.

An example of sequential processing is the computation of group subtotals within a relation which has as the ruling part a group attribute and a detail attribute, for instance, `s_id` and `p_id` in `Supply` of Fig. 7-16. To determine the quantity of all parts supplied by one supplier we naturally add the quantities in sequence until the value of `s_id` changes. Then a new subtotal can be computed for the next supplier group.

To obtain such subtotals in unsequenced relations is a function of the `Group by` clauses shown in Sec. 9-2. A query-processing program can use the structural or a similar model to help the user in the formulation of queries by translating from the user's conceptual view model to the implemented database model.

7-5 THE DESIGN OF A DATABASE MODEL

When a comprehensive set of view models has been established, the construction of a model for the entire database can be established. Relations from separate view models may be combined based on the attributes which they have in common. If view models have no attributes in common, there is no benefit in joining these data into a single database model. Figure 7-25 sketches the concept.

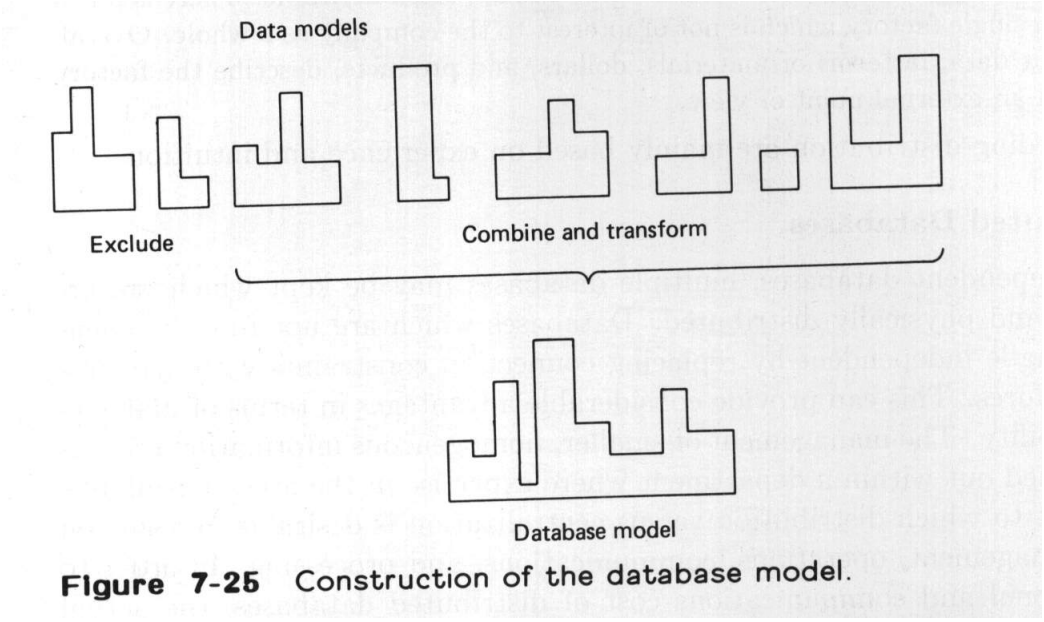


Figure 7-25 Construction of the database model.

Even when there are common attributes there may be no connections. A lack of connections indicates that the views or groups of views can be maintained independently of each other. We will call a database created from views which do not connect to other databases an *independent database*. An independent database is best maintained in a distributed manner, even if computer equipment is shared. As we discussed in Sec. 2-4-4, there are benefits to distribution, and if databases can be left smaller and managed autonomously, total costs will probably be lower.

Forcing independent databases into one integrated database is sometimes done to permit retrieval queries to access the data from multiple independent databases. Only a few database management systems today permit queries to be processed which access more than one database. The cost of combining independent databases is increased database system overhead cost in order to provide the required view model independence and protection for update transactions. Management costs, incurred to bring communality about in areas where there is little natural incentive to cooperate, may also be high.

Not even all connected view models should be integrated. The linkage between some sets of views may be relatively weak and will not warrant the integration of a view model into the database. A weak linkage may be due to a shared, but unchanging attribute. In those cases we will also design independent databases, with a procedure to keep the shared attribute synchronized.

If, for instance, employees are identified with a department, and production of goods with a department, then the list of departments may provide such a relatively constant link. Only if employees are to be related to production will there be a sufficiently strong coupling between the two areas to justify the combination of the view models.

The existence of a shared attribute which is frequently updated in two otherwise independent models provides another incentive to combine the models in order to avoid redundant update effort, even if the linkage is otherwise weak.

An example of a centralized database is found in the airlines reservations systems. The basic relations giving seat availability and flight schedules have to be shared by all users and are accessed frequently.

In manufacturing companies distributed databases are often desirable. There is much activity within a single factory which is not of interest to the company as a whole. Overall input and output data, in terms of materials, dollars, and products, describe the factory adequately from an external point of view.

Decisions regarding distribution are mainly based on experience and intuition.

7-5-1 Distributed Databases

If we have independent databases, multiple databases may be kept which are organizationally and physically distributed. Databases which are not fully independent may be made independent by replacing connection constraints with synchronization procedures. This can provide considerable advantages in terms of management and flexibility. The management of smaller, homogeneous information areas is effectively carried out within a department where expertise in the area is available.

The extent to which distribution versus centralization is desirable depends on the cost of management, operations, communications, and processing. In order to reduce operational and communications cost of distributed databases, the actual computer equipment may be shared. A distributed database does not imply physical distribution, but rather a distribution of responsibilities over multiple databases.

If the users of the separate databases are located far apart, the increased management cost for remote operation and the increased communication cost for data entry and output to the user may make separate facilities desirable. A multiple site system can be strongly integrated or distributed. The cost of the required communications for an integrated system spread over physically remote sites will make a distributed approach likely.

The constraints among connected but distributed databases implied by the connections have to be maintained. Messages will be transmitted between distributed databases to cause subsidiary transactions which maintain the databases according to the connection constraint rules described in Sec. 7-3-6.

Each database in the distributed set will have its internal connections and some connections to the other sites. The relations and connections made available at one site can be described by one *database submodel*. A database submodel may represent a single view or be augmented and modified to take into account information and data from other views included in the database. A site could also have a global, integrated model of all the data in the distributed databases.

If a database which operates at one site has the right to access data from databases at other sites, it may be wise to have a copy of the *global database model* available at each site, although only data for the local database submodel will be stored on site. The ability to change even the local part of a database model locally will now be constrained, since remote models will be affected, even if their databases are not affected by the model change.

Since communication of voluminous data between distinct databases will be more difficult than within a database, we find two features related to the implementation of database models in a distributed system. We find messages being used to communicate among subsystems, and we find *replication* of relations.

When a relation is replicated in two databases, the databases are synchronized by keeping those relations identical. We can now speak of an *identity* connection between the copies of the replicated relations in the distributed databases. If the only constraints between independent databases are formulated as identity connections, the management rules for distributed databases are simplified. All other connections now are maintained within a database. Most work (see Sec. 13-2) on integrity of distributed databases is based on replication of relations. Retrieval operations on replicated data can be performed locally, permitting rapid query processing.

If at all possible one database should be designated as the primary database; all updates are performed there and subsequently update messages are sent to the databases having identity connections. If update responsibility cannot be assigned to one database, the message traffic required to maintain integrity increases greatly. In some cases periodic switching of the primary responsibility is possible.

An example occurs in a bank, where during business hours the primary databases are in the branch office. After daily closing the correspondence of the local data with the central office data is verified, and primary responsibilty is given to the central site. Overnight the central database can be rapidly updated with transactions which arrive from other banks at the central office. Update messages are communicated to the branch offices. In the morning responsibility is switched to the branches after an integrity verification.

We note that the creation of database submodels implies the existence of an integrated database model, even though the data may not be integrated. In a distributed database there may be a global schema based on the integrated database model to aid global queries. The next section will apply to distributed as well as to centralized databases.

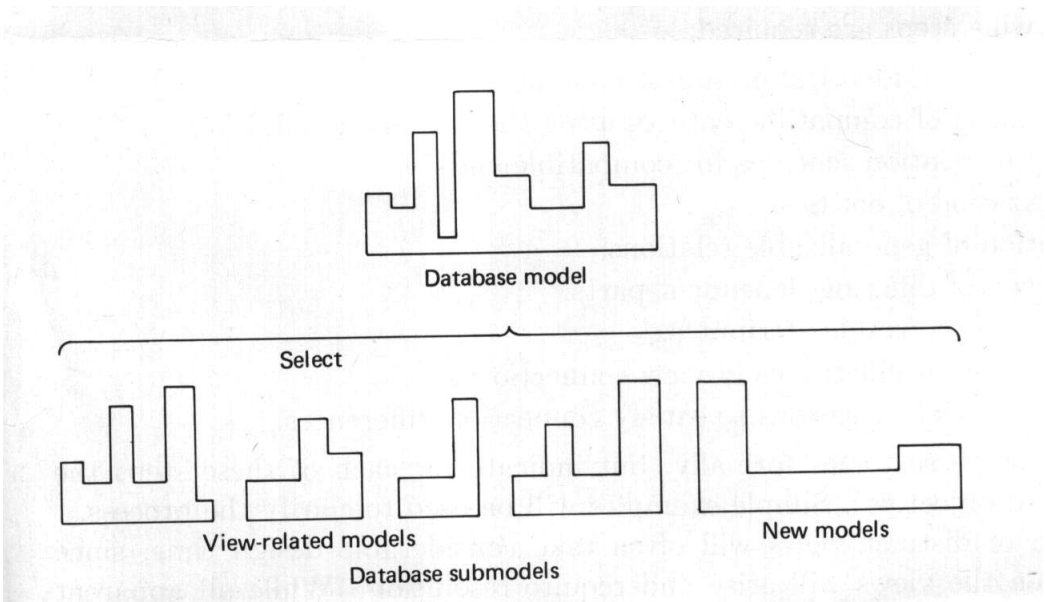


Figure 7-26 Establishing database submodels.

7-5-2 The Integrated Database

Once the decisions have been made on which view models are to be included in a single database model, the integrated model can be constructed. The database model will consist of relations of various types, and the connections between the relations. The combination may look like a tree, like a number of trees (a *forest*), or like a network when displayed.

Objectives When the integrated database is being constructed, a number of objectives can be considered.

1 Obtain relations with the greatest degree of semantic clarity.
2 Retain view independence to simplify subsequent distribution.
3 Have the smallest number of relations.
4 Have the smallest number of tuples.
5 Let the number of data elements stored will be minimal.
6 Let the number of connections between relations and shared attributes be minimal.
7 Let the total activity along all connections between relations be minimal.

Rules to establish optimality according to the last four criteria have been studied using functional dependencies between attributes as the basic elements for decision making. In many practical instances the database designs based on any of the six criteria will not differ greatly.

Semantic clarity is enhanced when strongly related attributes are grouped together, and this may be accomplished with a limited number of relations and interrelation connections. Normalization will often have increased the number of tuples and reduced the number of data elements. Integration can reduce the number of total tuples by combining them, but typically increases the total number of relations and their connections.

Transformation for Integration To integrate view models into a database model the following steps are required:

1 Identification of identical or subset domains.
2 Identification of compatible entities using these domain definitions.
3 Merging of relation schemas for compatible entities.
4 Generalization of entities.
5 Adaptation of generalizable relations.
6 Integration of differing dependent parts.
7 Definition of derivable attributes.
8 Identification of differences in view connections.
9 Database model extensions to satisfy connection differences.

We will not treat these topics formally, but indicate for each of these steps the requirements and objectives. Simple examples will be used to clarify the process.

Integration of distinct views will often take considerable design time, since conflicts between the views will arise and require resolution. While all apparent conflicts can be solved on a technical level when using the structural model, all technical transformations should include consideration of the user's intent.

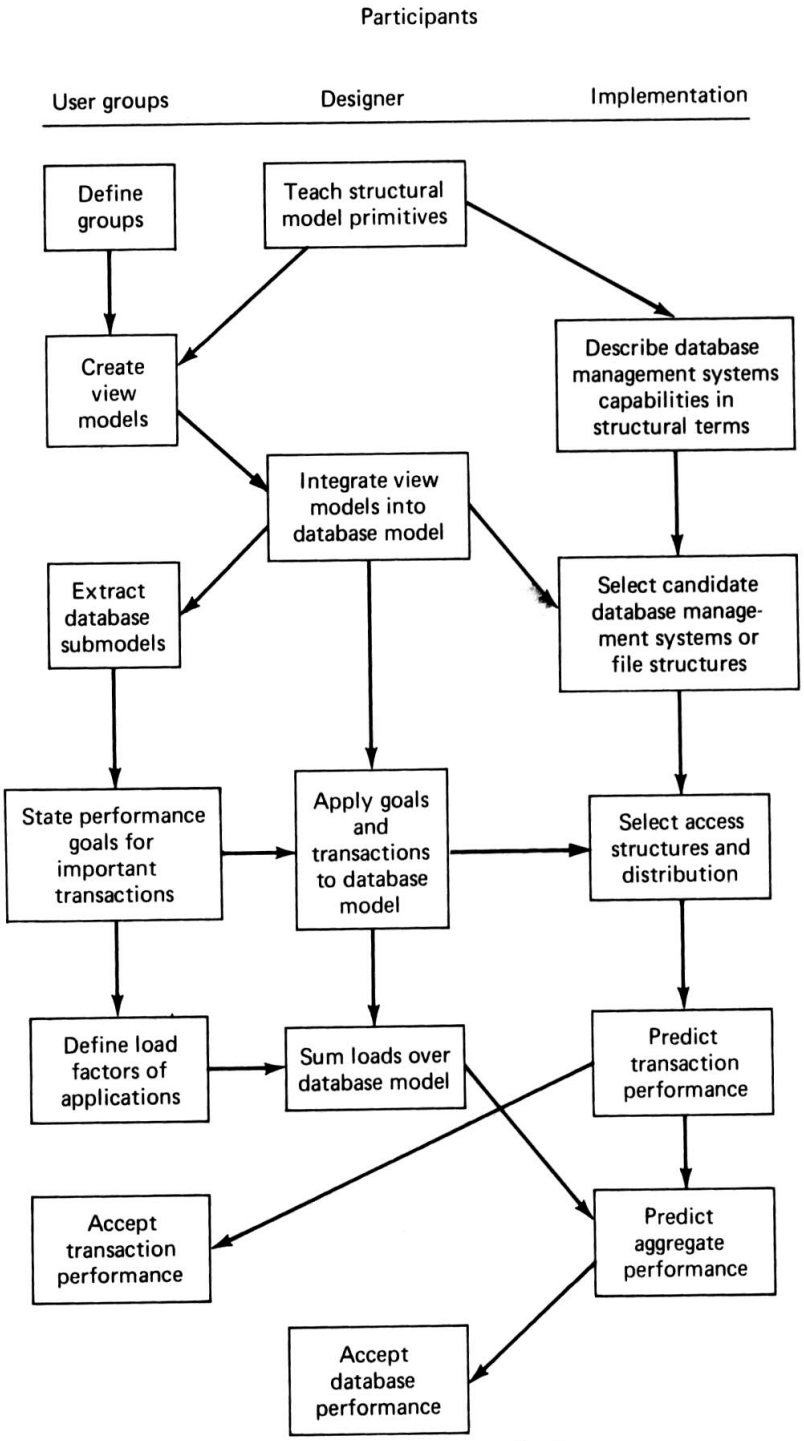


Figure 7-27 Functions during the design process.

Figure 7-27 brings together the design phases considered in this chapter and some of the issues discussed earlier. Such a diagram is of course simplified and ignores all the feedback and iterations which go on during a real design process. the time required for such a design ranges from a couple of weeks for a well-understood application which does not pose performance problems to more than a year for systems where major distinct applications are integrated and performance demands are stringent.

Identification of Domains All definitions of FDs and MVDs which lead to the entire model structure are based on the definition of attributes and their domains. In practice most domains are easy to recognize, but often difficult to define exactly. Where the domains define real entities: people, parts, planes, etc., the domains are defined by their use in the view. Where domains define abstractions: color, departments, schedules, etc., the domains may be formally defined. Domains coming from views which were obtained independently will often partially overlap.

The `colors` in cars had two roles, which might be associated with two views. The interior colors may not include metallic hues available for the exterior.

Identification of compatible entities We will first consider entities to be defined by the ruling part of view relations. Precise domain definitions applied to the ruling part attributes define the entity to be collected into the database precisely. Domain differences may reveal that different views keep data on differing subsets of an entity class.

The `Employee` relations are a prime example. If the enterprise hires consultants, the view of the personnel department may not include them, but the payroll department will, since they will have to be paid, and taxes have to be withheld on their pay and reported to the government.

Two solutions to resolve view conflicts are feasible: one is to change one or both views until they are compatible, the other is to include both views in the model. The first solution keeps database models simpler but may do violence to the user intent and make the resulting database unacceptable. Some compromises are easy; others quite impossible. The second solution is technical, and we will concentrate on that approach throughout.

Using the structural model we create subrelations and general relations in the database model to resolve conflicts. Specific cases are reviewed below.

The assignment of attributes to ruling and dependent parts was based on the concept of functional dependencies. We note that the term *always* was used in the definition of a functional dependency. A model describes a stable situation. We cannot base the design or the integration on the current contents of a database which may exhibit additional, but temporary relationships. Functional dependencies have to remain valid although data values change.

However, a functional dependency can be disproved by facts found in the database, as shown in Example 7-13.

Example 7-13 Relation-schema for a telephone book

A telephone book might be set up as

```
Telephone_book:  RELATION
    name :⟩ address, telephone_number;
```

but later shown not to represent a correct functional dependency if for one `name` combination multiple telephone numbers are found. A change is

```
Telephone_book:  RELATION
    name, address :⟩ telephone_number;
```

Merging of relations which have compatible entities Consider R_1 to come from view model *V1* and R_2 to come from *V2*. If the ruling part attributes A have completely identical domains $D(R.A)$ the relation-schemas can be merged and a single relation R_{db} will appear in the database model. The dependent part may have more attributes; we will consider that aspect further.

$$D(R_1.A) = D(R_2.A) \;\Rightarrow\; R_1(S_1, T) \bigcup R_2(S_2, T) \;\text{ so that }\; R_{db}(S_1 \bigcup S_2, T) \qquad 7\text{-}28$$

If a ruling part attribute A has $D(R_1.A)$ which is a proper subset of $D(R_2.A)$ the relation R_1 becomes a subrelation R_1' in the model, with a connection

$$D(R_1.A) \subset D(R_2.A) \;\Rightarrow\; R_2 \longrightarrow\!\!\supset R_1' \qquad 7\text{-}29$$

The dependent part of R_1' will not include attributes already available in R_2. The use of a subset can be avoided by taking the union as in Eq. 7-28 and permitting `null` values to appear in the attributes $R_2(S_2 - S_1)$ for which no tuples R_1 appear. The introduction of `null`s for the sake of reducing the number of relations in the model hides, however, some of the semantics.

Generalization of entities Often entities are not subsets of each other, but are generalizable.

The personnel department above may not consider consultants but may keep data on retired people, who are paid by an outside pension plan. Now the payroll view is no longer a proper subset of the personnel view, although their overlap is great.

If two key domains overlap, a new general relation R_g will be created

$$\begin{aligned} D(R_1.A) \bigcup D(R_2.A) \to D(R_g.A) \;\Rightarrow\; & R_g \longrightarrow\!\!\supset R_1' \\ & R_g \longrightarrow\!\!\supset R_2' \end{aligned} \qquad 7\text{-}30$$

The general relation contains all common data. Subrelations will be retained to hold instances and data particular to the subgroups. The general relation is the one to be used for general queries, where presence of special categories, here `consultants` and `retirees`, could bias the results.

Adaptation of generalizable relations Even when ruling-part domains do not overlap, the entities may be generalizable. These cases are recognized by finding relation-schemas that are similar.

Trucks, vans, and cars are generalizable to vehicles, and this generalization is useful because similar data are kept on all of them. They also share as ruling part the attribute `license_number`.

Integration of differing dependent parts We merged in the earlier steps relations based on compatibility of entities. Often this required the creation of subrelations. Now all the relations have been established. We now assign dependent attributes to the most general relation possible. If the original relations have dependent-part attributes that do not correspond to attributes within the general relation, these attributes have to be managed distinctly. The two approaches used earlier are feasible: changing the domains obtained from the views by augmenting the domains

values so that the domains can range over all attributes is the first approach; creating subrelations is the second approach.

In the first approach an adaptation is made to the domain, the relation with fewer dependent attributes is augmented, and then the integrated relation is created. augmented with the dependent attributes on that domain. A typical augmentation is performed by taking the cartesian product of the relation with the constant `none`. Equivalently a join of relation and constant with the condition $\ominus$ = `true` may be used.

If, for instance, the attribute `cargo_capacity` is missing from `Cars`, a value "`Passengers only`" is added to the domain for `cargo_capacity` and the relation-schema and the tuples for `Cars` are augmented with this attribute prior to the merger into `Vehicles`.

Sometimes both relations need to be augmented before integration, as shown in Example 7-14.

Populating the database with many data elements of little value not only wastes space but can also confuse the user. The value `null` leads to particular problems, for instance an average of an attribute column should not include `null` values in the count. Also a join on an attribute which includes `null` in its domain will return unexpected tuples in the result. Section 8-1-3 discusses this issue. The first approach, defining special values within the domain, is often easy but should be used with care.

Example 7-14 Generalizing two relation

The integrated database is to have a single `Education` relation, created from the similar `Highschool_record` and `College_record` relations. Since the attribute `minor` does not exist in the high school record, the constant attribute `null` is added to the relation. Given the relations

```
Highschool_record:  RELATION
    child, schoolname :⟩ subject;
```

and

```
College_record:  RELATION
    child, schoolname :⟩ major, minor;
```

the relation `Education` can be generated as follows

```
Education:  RELATION =
   ( Highschool_record  ×  (schooltype="high school",minor="null") )
  ⋃ ( College_record  ×  (schooltype="college") );
    child, schooltype, schoolname :⟩ major, minor;
```

The new attribute `schooltype` distinguishes the tuples obtained from the two views when needed. Since we do not know whether `schoolname` is unique in the combined domain, `schooltype` has to appear in the ruling part.

The second, technical solution is again to create subrelations. The subrelations contain all those dependent attributes which are not appropriate to the general relation.

We find subrelations of the `Employees` for `Managers` with their `stock_options`, for `Salesmen` with their `sales_quotas` and `territories`, and for `Retirees` with the attribute `last_day_worked`. We note that having subrelations in the database model does not imply having corresponding files. There may well be one `EMPLOYEE` file with a number of variant records.

Derivable Attributes We may find in some views attributes which are derivable from other attributes. These embody strong connections which are easy to miss. Only the enumeration of a more general category, *computational functional dependencies*, of the FDs treated earlier will show such redundancy. Analysis of the data does not show conflicts in a model containing derived values, since the attributes are not comparable with the source attributes. Such attributes may have existed within view models or arisen after integration.

Example 7-15 Derived attribute

Perhaps a field `price` appeared on bills, but not within the same view as the detailed invoice, which was always recomputed. After integration we note the computation explicitly, as seen in the `price` field of the `Invoice` below.

```
Invoice:  RELATION
    date, customer, item :⟩ value, number, price = number * value;
```

Derived attributes are redundant, but not always obviously so, so that inclusion of the derivation in the relation-schema can prevent later problems. Queries can now automatically incorporate the computation when a derivable attribute is requested.

Identification of differences in view connections The connections in the view models imply the constraints that update transactions must obey in order to keep the database according to the semantics specified by the user. The connections can also be exploited at retrieval time to simplify the formulation of queries (see Sec. 10-2-3). All the known connections are mapped into the integrated model and identified with their original view.

Conflicts may be found among relation pairs that now have more than one connection among them. Again two choices are available: adaptation to a single connection type and a change of one users semantics view, or the use of subrelations to represent the differing original views.

Satisfy connection differences We may find that relations to be merged have differences in connections. While there are many combinations, the essential conflict is the ownership constraint versus the reference constraint. The subset connection can be taken as a special case of an ownership. Subrelations permit again a technical solution. An illustration follows.

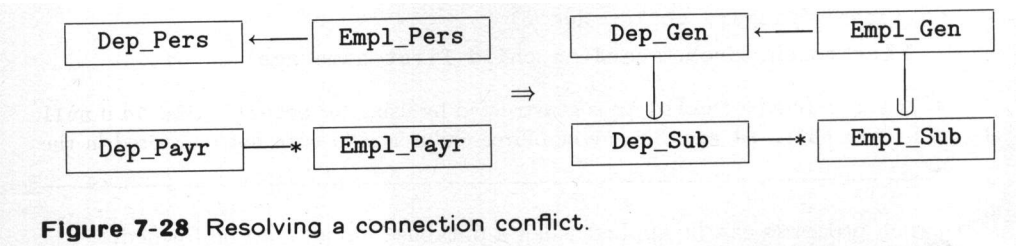


Figure 7-28 Resolving a connection conflict.

The personnel department views employees as primary entities, and includes a reference attribute to the department relation in their dependent part. The payroll department takes a narrower view; it considers only employees whose pay can be charged to a department. Here the department relation owns the employees. The transformation is shown in Fig. 7-28.

Now employees can be deleted from the payroll, but remain available to personnel. The general department relation may have a department `Limbo` as a reference for personnel not currently associated with a department.

We will illustrate some of the transformations occurring during integration. In this example ruling parts are being transformed to compatible entities. To permit a view access using the original ruling part, a lexicon is created.

Example 7-16 Integration

A paternalistic corporation provides a school for the employees' and other children in the area. For budget analysis purposes it is desirable to combine the employee view model and the school view model, so that school benefits can be credited to the employee's account.

The school view model contains the relation for the entity `schoolchildren`:

```
Schoolchildren:  RELATION
    school_child_id :⟩ guardian, child_first_name, age, ...;
```

We had earlier the nest relation `Children` in the model containing `Employees`.

```
Children:  RELATION
    father, child :⟩ age_c, ...;
```

The following steps are carried out in order to create the database model shown in Fig. 7-29.

1 The two attributes, father's name and child's name which constituted the ruling part of `Children` will be combined into one attribute, called `child_id_emp`. This attribute becomes the ruling part of a `Children'` relation.

2 An associative relation is constructed from the two attributes `father` and `child_id_emp` using the projection,

`Employee_child = ∏ Children'⋆(father,child_id_emp)`

3 The data on the children in both relations is merged by the outer-join:

`Children'⋆(father, child, age_c)`
`⟕schoolchildren⋆(guardian,child_first_name,age)`

4 A final ruling part `child_id` is constructed by using for `school_child_id` if `null` the value from `child_id_emp`, for any employee children who were not registered in the school.

Many such processes can be applied when a database model is to be formed.

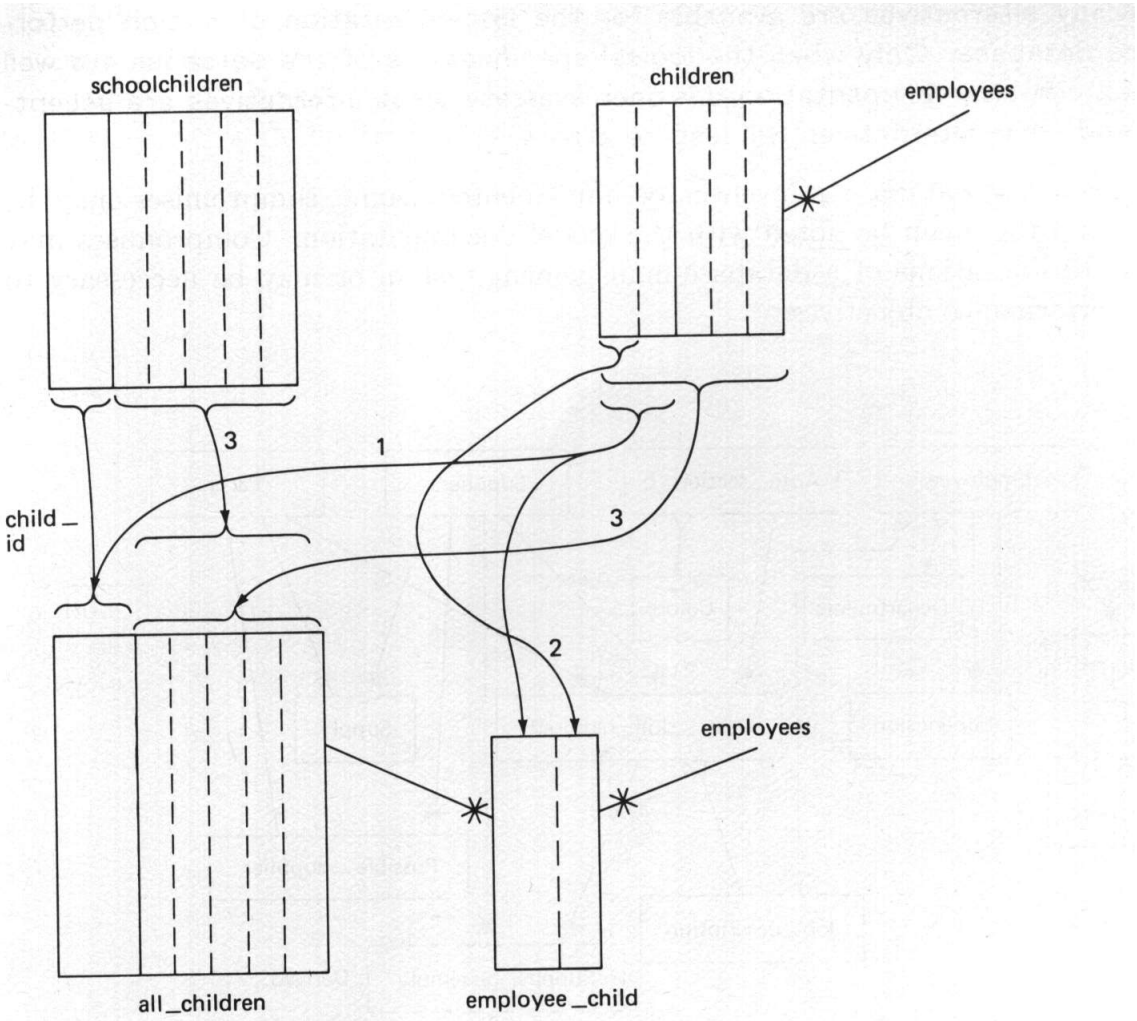


Figure 7-29 A transformation during creation of the database model.

The Datamodel after Integration The integrated database model may be complex but presents an accurate description of the user requirements. Non trivial databases in use have included from two to a few hundred relations, and from a few dozen to nearly a thousand attributes. Because of the size of integrated database models a tabular representation is preferred. Software packages have been developed to maintain much of the type of information we used in modeling, and greater automation of the databases design process is foreseen. The integrated model can be used by programmers developing a database or as the skeleton for a database to be supported by a database management system. Chapter 8 will present ways to specify the desired model and its implementation to various database-management systems.

Database designs often become unwieldly if they are forced too early into the straitjacket of a specific database-management system. The ability to design a database model without concerns for specifics of physical implementation is considered critical for the successful development of databases which are to serve multiple objectives. This does not mean that we no longer care about performance. The model provides a basis for the design of a well-performing system.

Many alternatives are available for the implementation of a high performance database. Only when the logical specifications of the database are well defined can an implementation designer evaluate what alternatives are acceptable and what alternatives will lead to errors.

When the database is eventually implemented, some compromises may be made, and these can be noted with the model documentation. Compromises may be due to limitations of a database-management system or may be neccessary to meet performance objectives.

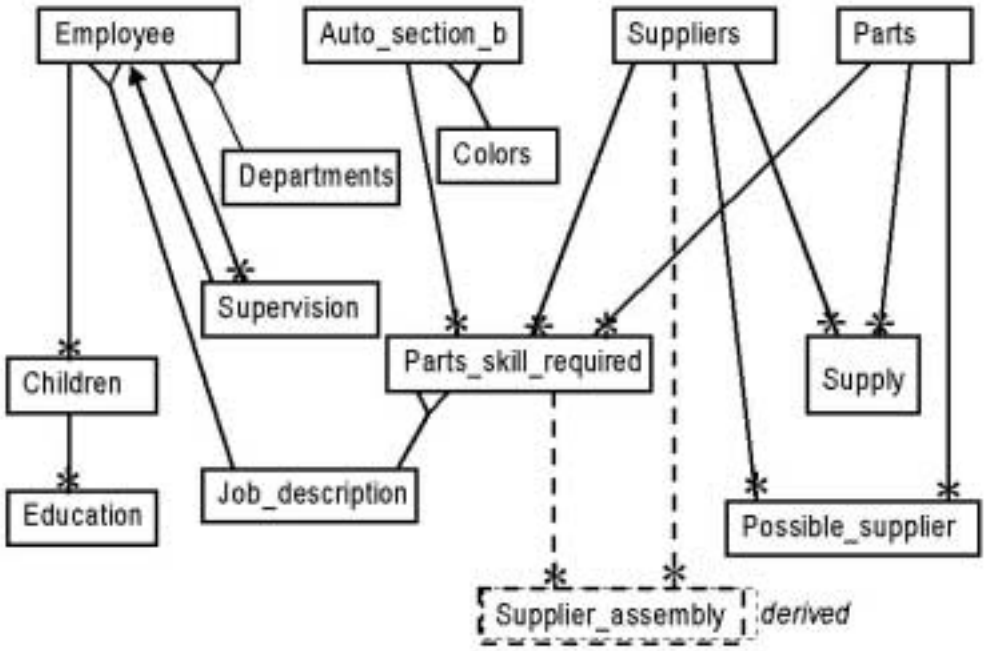


Figure 7-30 Semantic relationships in a database model.

Some observations can be made about the relation types found in integrated database models:

Former entity relations may now be referenced by relations from other views, constraining the freedom of deletion that existed previously, unless subrelations are used to resolve the view differences. Referenced entity relations which expand an attribute of an entity relation in detail, often used originally in a specialized view, are now widely available.

Two relations which share a dependent attribute domain may now both reference a new referenced entity relation to show this relation more explicitly. The use of a shared entity relation assures continued domain compatibility.

Associations appear among relations which now share owned relations. The number of pure nest relations, frequent in the hierarchical models which dominate individual views, is reduced.

We observe that integrated models contain many subrelations. Multiple subrelations and their general relation are often implemented within a single file.

Figure 7-30 depicts a connection graph based on the relations used earlier in this chapter.

Database Submodels When the view models have been transformed as indicated to accommodate the database model, the views presented by the original applications have been broadened and perhaps altered. It is desirable to maintain a description of the database in terms of the original view models, since that provides the documentation for the design.

In order to accommodate access to the database after integration, we define further database submodels. Database submodels permit an application to access the integrated database, taking advantage of new relations to which access has been made available, and at the same time presenting the structure and attributes largely in terms of its original model.

Sometimes it is better to adapt the database submodel to be a proper subset of the integrated database, since this may provide a more realistic view of the actual operation and its constraints. When database submodels are subsets in the database model, such a transformation from database model to database submodel only requires selection and can be easily achieved.

When substantial transformations have taken place, automatic transformation of queries phrased according to different submodels is difficult to achieve. We see little of that concept in practice. An approach to this problem is presented in Sec. 9-6, in an example of a database-system implementation.

Means to describe the relationships in the database are shown in the next chapter.

BACKGROUND AND REFERENCES

In order to develop the conceptual basis for database planning we have used relations as the basic units and classified five types of relations in order to describe the required choices. Many other models to aid database design have been developed and have influenced the concepts presented here.

The *relational model* which provides the basis for the conceptual view presented in this chapter allows a great deal of formal analysis. Several early papers (Langefors[63], Levien[67], and Childs[68]) suggested the use of a model based on set theory to describe file and database structures. Most of the current activity was kicked off by E.F. Codd, who, in a series of papers (Codd[70,74], and in Rustin[72)], presented the relational model in terms of its relevance to database design and implementation. In a second paper in Rustin[72], Codd describes the required operations and their power. Associated early contributions are Heath and Date in Codd[71], Delobel[73], Armstrong[74], Wang[75], and Forsyth in King[75], which provided mathematically oriented criteria for the optimization, manipulation, and demonstration of the correctness of the manipulations in relational models.

Fagin[77] defined multivalued dependencies and a fourth normal form. Up to seven normalizations have been proposed (Beeri[78]). Issues of relational decomposition are summarized by Maier[80]. The mathematical theory of dependencies in databases is now a subfield of its own, with a rapidly exploding literature. The set of axioms defining transformations is due to Armstrong[74]. A basic text is Ullman[82] and recent results may be found in journals, see for instance, Sadri[82] and in the proceedings of the ACM-PODS conferences (Aho[82]).

A tool in the developmemt of dependency theory is the *universal relation*. All data is placed into a single abstract unnormalized relation representing all entities and all their attributes. Many cells will be null. The formal knowledge to manipulate the universal relation consists of a collection of various dependencies which relate the attributes.

In our structural model we avoid many of these issues by concentrating on the simpler view models. The practical importance and sufficiency of Boyce-Codd-normal form was presented by LeDoux[82]. Date in Tou[74] expresses strongly the distinction between the database submodel and the database model. Several related papers appear in King[75].

Integration of databases is presented by ElMasri[79], Hubbard[81], and Navathe[82]. It also appears as phase in Schkolnick[78] and Lum[79]. Generalization of entities is addressed by Smith[77]. The concept of multiple linked databases is due to Litwin[81L] and tested within the POLYPHEME project.Languages to describe the semantics have been defined by Mylopoulos[80] and Hammer[81]. Work by Earley[71,73] is oriented toward language aspects of relational structures.

In order to construct relevant database models, the semantics of the interrrelational structure have played an important part in this chapter. Work by Schmid in King[75] and Manacher in Kerr[75] was especially influential. ElMasri[80] compares the how the structural concepts are modeled in alternative schemes. Questions of semantic relevance of the structure of databases were analyzed in Langefors[73], Deheneffe[74], Hainaut[74], Robinson in Douque[76], and Roussopoulos in Kerr[75]. Kent[78] reports on experience with a conceptual modeling tool.

Integration of relations can lead to `null` entries. Vassiliou[80], Goldstein[81], Lipski[79], and Zaniolo[82] consider the problem of nulls, Imielinski[81] extends the relational operations to deal with nulls.

Later work by Codd[79] and Date[82] extends the semantics of the original relational model. The new model, RM/T, includes 5 referenced relation types and 8 integrity rules, which can be mappped into the structural model. An exhaustive review is in Date[82]. Some of this work is influenced by the concepts of semantic nets; a comprehensive reference is Schank[73]. Joins are extended by LaCroix[76].

Other types of models can be easily related to the concepts presented here. Some of the models which are attracting attention are summarized here.

Entity models use the concept of the entity as their basic unit. Data structures are composed of entities defined by information requirements. Entities have properties, which were in our model decomposed into attributes, subsets, or data elements. Extending entities with the notion of relationships, as done by Chen[76], extends the descriptive capability, and this work has also spawned much interest, collections are found in Chen[80] and successor proceedings. Relationships may be simple $(1:n)$, as our ownership and reference connections, or $m:n$, as modelled by associations and other multiple connections. Such relationships are *essential*, since the information is stored within them. The guidance given by these models tends to be less formal than the rules based on relations.

Functional models formalize the relationships among entities. Shipman[81] provides a language to express them and Buneman[82] provides a query language based on the functional concept.

An *access-path model* defines the database as collections of sequential transformation sequences, required to obtain a physically coded entity in response to a logical query. A rigorous hierarchy of the transformations allows the prediction of information loss and access cost at the various levels. Entity and access-path models (DIAM) have been developed by Senko[73] and compared with alternate approaches in Bachman[75] and by Hall in Neuhold[76]. A level to model hardware functions is also available. DIAM is shown to be capable of modeling many current concepts (Senko in Rustin[72], in Benci[75], in Kerr[75],

in Douque[76], and in Neuhold[76]). The flexibility of the model may diminish its didactic power.

A *hierarchical model* uses the nest concept as its basic unit. Trees are created by nesting nests of nests, and forests are created by collecting trees. The view model and the database model are identical. This model has been used to develop access and update strategies. Hierarchical and network models have been influenced by early applied work as Hsiao[71] and Bachman in Jardine[74].

A *network model* uses the association as its basic unit. An integrated model is created by defining independent entities, their nests, and the possible or actual binding between them through the associations. In the traditional network model, abstracted from the CODASYL specification, all joins of interest were predefined so that the operations stress navigation between bound data elements. Recent developmemts are separating the logical and implementation design phases, as shown in this book.

The commonality of the various models is becoming obvious. Their strength tend to be relative.

In Rustin[74] network and relational approaches were presented. A lively discussion between camps committed to various approaches has appeared in the Proceedings of the ACM-SIGFIDET and -SIGMOD conferences. Misunderstandings developed because of the difference in origin; the relational approach was initially conceived as a modelling tool and the network approach began as a generalization of descriptions of actual databases. Sometimes the alternatives were used as strawmen to defend philosophies. A special issue of ACM COMPUTING SURVEYS (Sibley[76]) provided a forum for several approaches.

Data models which integrate these and information structure concepts are presented in Sundgren[75] and Kobayashi[75], those notions are further developed by Cook in King[75], and Chen[76]. The transformations of databases between models of the relational and network type has been defined by Adiba in Nijssen[76], De[78], Zaniolo in Bernstein[79], and Lien[82]. Lien[81] considers hierarchies in relational darabases. Date[80] presents a unified language to deal with access communality. Many issues in modeling of databases are summarized in Tsichritzis[81].

The proceedings of several conferences, two sponsored by the SHARE organization (Jardine[74,77]), one by the Institut de l'Informatique in Namur (Benci[75]), and a series organized by IFIP technical committees (Klimbie[75], Douque[76], Neuhold[76], Nijssen[77], Schneider[79], and Bracchi[79]), contain papers which discuss many aspects of database design. Abstract issues of modelling were the topic in Brodie[81]. Another source of current material on database modelling are the proceedings of the annual VLDB conferences. Selected papers appear in ACM's Transactions on Database Systems, a number of them have been cited above.

EXERCISES

1 Why is the `state` attribute in the ruling part in the `Automobile` relation of Example 7-2?

2 Sketch two view models for student grades.

3 Give the maximal size of a relation following Eq. 7-1.

4 What is the projection ∏ ((birthdate,job) OF employee), given Fig. 7-6?

5 What is the join of relations employee and job-description?

6 Show why the number of tuples in the difference of two relations is as shown in Section 7-4-1.

7 What is the expected number of tuples from a join? The limits were given in Section 7-4-3.

8 Why would one be interested in predicting the number of tuples?

9 Carry out a join of the relations Employee_2 and employee. Which attributes control the join? Describe the cause for any inconsistencies you find. How could they have been avoided?

10 Construct two relations which, when joined, reveal an inconsistency. What would you do if such a problem occurred in the execution of a program?

11 Using Figs. 7-6 and 7-7, construct a relation

```
PTA: RELATION
    father, age, child, age_c, schooltype, schoolname;
```

Define the reverse process. Show that there is no information loss when applying both transformations in sequence. Is this realistic?

What additional information is needed in reality?

12 Take a paper describing a database example and define the files, sets, or relations used in terms of entity relations, lexicons, etc.

13 Is it necessary for proper execution of a join that the relations are in any particular normal form?

14 Flowchart or write in a high-level language the elemental steps required to carry out the set difference (—) operation on two relations and two sequences.

15 Assume that the database shown in Fig. 7-29 is too large to manage and should be distributed. How could it be split in two parts, and what would be the problems in the distributed operation?

16 Read one of the papers on database models from the references given and compare the exposition with this chapter. Consider particularly which concepts described here are lacking in the paper and vice versa. Prepare a cross-reference dictionary for the defined terms.

17 If it turns out to be very difficult to integrate view models into a database model, what does that indicate?