DESIGN CONCEPTS AND IMPLEMENTATION OF DATABASES
Gio Wiederhold, DATABASE MODELS, April 4, 1996
Chapter 3, file D3CID.TEX

Chapter 3: Database Models;
3–1 Why models.
3–2 Entities and Relationships (was View models); from DBD 7–1.
3–3 The Structural Model
3–4 Relations, attributes and domains; from DBD 7–1, and 7–2–6
3–5 Connections
3–6 View Models
3–7 Relation Types in Database Models
Building Blocks: relations and connections; from DBD 7–3.
3–8 Summary

Chapter 3

Database Models

¡The French Connection?¿?

3-0 INTRODUCTION

When systems are complex we deal with them through simplified abstractions. The abstraction used for databases is a *database model.* Models permit formalization of selected, important aspects of databases. Focusing our attention on a few aspects helps us deal with the complexity of real-world entities and their relationships.

Knowledge about relationships helps in the creation of new information. Relationships permit entities to be combined in new and imaginative ways. A transaction program selects and combines data about entities according to relationships defined in the database model. Relationships among the data form a *structure* which establishes the meaning of the data. Relationships are independent of the processing programs which exploit them. The objective of this chapter is to develop concepts for recognizing and describing this structure, so that guidance can be provided for the design of databases.

‡‡‡‡‡‡ **Our Model** For readers with previous exposure to database models we note that the model used here, the *structural model*, is an extension of the *relational model.* The extensions permit more semantics to be expressed, while retaining the strengths of the relational model. The structural model can also be viewed as a formalization of the *entity-relationship* (ER) and *entity-category-relationship* (ECR) models.

We can use the structural model to guide relational as well as hierarchical or network implementations. Since the structural model focuses on the conceptual structure of the data, rather than on the representation within a DBMS, it is also valid when the database is stored in main memory or managed by the user on traditional files. ‡‡‡‡‡‡

Layout of this chapter We proceed from the general to the specific:

1 In Sec. 3-1 we introduce model and modeling concepts.
2 In Sec. 3-2 we describe a popular informal model, the Entity-Relationship model.
3 Section 3-3 formalizes entities using relational concepts.
4 Section 3-4 formalizes relationships using connections.
5 In Sec. 3-5 we use relations and connections to build models of database applications.
6 In Sec. 6 we introduce views to accomodate semantic differences in models.
7 In Sec. 7 we provide alternative database semantics by focusing on relation types.

These models are incomplete without facilities to manipulate them, in Chap. 4 the operations to manipulate the models are presented; these operations can be implemented to manipulate the database itself as well. The database design processes in Chap. ¡5¿ manipulate the models with the help of the operations. In Chap. ¡6¿ we discuss building an integrated database with these tools.

We carefully distinguish models and implementation. Implementations of relational models will be shown in Chap. 8?, and further implementations in Chaps. 9, 10 and 11.

3-1 MODELS

A database model is an abstraction of some aspect of a database. The abstractions are chosen so that we can focus on what is important to us and neglect, as much as practical, all else. When initially conceptualizing a database for some application we want to focus on the important entities of the enterprise being modeled. The relationships among the entities determine how the enterprise operates. For those entities and relationships we only consider the relevant attributes. A initial model should be sufficiently simple and clear that it provides a good communication medium among all users and implementors. Figure 3-1 shows some entities and relationships in our movie world, together with representative instances.

For implementation further abstractions are needed; those abstractions must permit assessment of expected database performance. After the database is implemented, yet other abstractions are needed to let the user manipulate the database without excessive concern about the actual implementation.

Entities	Objects
Movies	Casablanca; Notorious; The African Queen; The Invisible Man.
Directors	Curtiz; Hitchcock; Huston; Whale.
Actors	Ingrid Bergman; Humphrey Bogart; Cary Grant; Claude Rains; Katherine Hepburn.

Relationships	Links
Made_by	Casablanca → Curtiz;
	(Notorious, Suspicion, The Birds, The Horseplayer) → Hitchcock;
	The African Queen → Huston;
Played_in	Ingrid Bergman → (Casablanca, Notorious);
	Humphrey Bogart → (Casablanca, Sabrina, The African Queen);
	Cary Grant → (Holiday, Notorious, To Catch a Thief);
	Katherine Hepburn → (Holiday, The African Queen);
	Claude Rains → (Casablanca, Notorious, The Invisible Man);

Figure 3-1 Some entities, relationships, and some of their instances.

STRUCTURE AND SEMANTICS In the design of database we are concerned with the structure of the database. We have to select meaningful entities and their attributes. We have to identify the relationships among them and their nature. Complex entities must be decomposed, in order to permit application of relational concepts, which permit analysis of structure and synthesis of query strategies. With that in mind we develop a structural model to capture aspects of the data that must be reflected in the structure of the entities.

Entities and relationships are independent of programs that operate on the data, and their structure persists over time, even as objects which instantiate them change. The persistence leads us to be concerned with what the data in a database mean, the *semantics* of the data. Given the semantics we then determine then how the data are best structured. The structural model of a database records those semantics that can affect record and file layout for the database.

In the structural model entities are abstracted as relations and the relationships between entities are abstracted as connections. Connections also arise when complex entities are decomposed.

We will first discuss the selection of abstractions in general and in successive sections develop the particular structural model. The formality will increase as we proceed. We use the concepts of entities and relationships, developed in Sec. 3-1-2, as the informal basis for the structural model in Sec. 3-1-¡3¿

3-1-1 Introduction to Models

Entities and relationships define classes which will contain many objects and links. These concepts are powerful, but we must consider how well their definitions can cover the variety of objects in the real world. After all, each real-world object is in some way unique.

CLASSES Conceptual classes can be defined in three ways:

1 As natural classes: membership of a class is determined by what seems to make sense *for most people*
2 By extension: a set of objects is explicitly enumerated as being the members of that class
3 By definition: rules are given which decide for each object if it is a member of a particular class

Each alternative has its advantages and liabilities. In practice all three types of definitions are found. If their differences are ignored some confusion can ensue.

Natural Classes Dealing with natural classes is intuitively the most satisfactory alternative. For instance, when we see people working in a film studio we recognize *naturally* who, say, the `Actors` are, We expect a certain type of behavior from an person of that class, and can easily think of the attributes needed to distinguish one `actor` from another.

Natural knowledge helps us to write processing rules for the members of the class. We *know* that they get high `fees`, and that filming of a movie cannot proceed without their presence. People in the industry also *know* that `actors` are engaged by the `Central_casting` department,

Enumerated Classes The most rigorous definition of class membership is achieved by listing all object instances for the class, the *extension*, explicitely. This choice is convenient for small, and naturally closed sets, say, the set of continents on earth: `{Antartica, Asia, Australia, Europe, North-America, Oceania, South-America}`. This listing, if agreed upon, avoids arguments of definition. Virtually closed are the ways in which actors can carry out their trade: performing in `{movies, plays, TV shows, commercials}`. Some classes are defined as binary alternatives and are always closed and best enumerated: `sex {female, male}`.

Rule-based Classes We can define classes by *intension*, through rules of membership. Rules have the advantage that we can apply the intent to individuals, objects, and events not yet seen. This approach is appropriate for databases, since they must deal with a dynamic world, where new objects to be classified are created throughout the lifetime of the database.

The classification rules operate on attributes which describe the real-world objects, as shown in Fig. 3-2.

```
Actor ⊢ located_{Studio,          -- rules defining Actors
        performs_in{movies, plays, TV shows, commercials}
```

Figure 3-2 A classification rule.

Choice of Class Definitions Databases depend primarily on intensional classification rules for establishing entity membership for their objects. Enumeration is used mainly for abstract entities used to organize the database.

For many classes enumeration is not feasible. The set of `Actors` changes every day, and unless we define an academy of `Recognized_actors`, the class will defy closure. In order to manage classes which change their membership easily it is best to use rules and avoid enumeration.

Classification rules still exhibit a problem in common with enumeration, the requirement for predefinition.

‡‡‡‡‡‡ **Errors due to class mismatch** Computer-based systems can make errors when the processing rules set up by programmers are based on concepts defined naturally, but the actual class membership is determined by enumeration or definition. Figure 3-3 shows a case where the rules lead to a misclassification for the use of a natural class.

Our definition of `Actor` coversincludes `Asta`, the dog which acts in the `"Thin Man" Movies`.

A program, to validate working conditions for child actors now includes `Asta`.

It is easy for a programmer to fail to realize that non-human `Actors` are permitted by the classification rules. Processing rules set up by programmers are based on their expert natural knowledge of the application.

Figure 3-3 Problem of classification mismatch. ‡‡‡‡‡‡

We associate descriptive attributes with a class, as shown in Fig. 3-4. When the attributes for an object are inappropriate to the established classes we better establish a new class.

Class	Attributes
`Actors`	`stage_name, born, sex, agent, SSN, address, ...`
`Movies`	`title, year, director, producer, studio, ...`

Figure 3-4 Classes and attributes*.

Definitions may assign objects to multiple classes. Sometimes overlap is fine: `Asta`, will remain a `Dog` even though a our definition lets him also be one of the `Actors`. At other times disjointness is an advantage in understanding the database. Subclasses make distinctions among objects explicit.

*We capitalize names of classes and use lower case for their attributes.

Class		Subclass	Attributes
`Actors`			
	⟶∋	`Leading_actors`	`fee, percentage, type_cast;`
	⟶∋	`Character_actors`	`fee, type_cast, recognition;`
	⟶∋	`Supporting_actors`	`fee, type_cast;`
	⟶∋	`Extras`	`daily_fee, availability;`

Figure 3-5 Subclasses.

Subclasses When we organizing databases we often need subclasses to further distinguish objects which need differing attributes. Figure 3-5 provides an example and introduces the symbol (⟶∋) we use for subclasses.

We still find some overlap of attributes in the subclasses, and could further refine the classification as shown in Fig. 3-6.

Class		Subclass		Subsubclass	Attributes
`Actors`					
	⟶∋	`Regulars`		. . . .	`fee, type_cast;`
			⟶∋	`Leading_actors`	`percentage;`
			⟶∋	`Character_actors`	`recognition;`
			⟶∋	`Supporting_actors`	`;`
	⟶∋	`Extras`		. . . .	`daily_fee, availability;`

Figure 3-6 Subclasses and subsubclasses.

The extent to which one should refine the classifications is a pragmatic database design issue.

Classification supports the "divide-and-conquer" paradigm basic to science. For instance, if we have to select the cast for a new `Movie` we can use the classification to establish sub-tasks of finding the various classes of `Actors`. When all subtasks are done then the entire casting task is complete.

To assure success with the paradigm we would like to be sure that the set of subclasses is complete. Then we can be sure that any of the `Actors` belongs to one and only one sub- or subsub-class. The set of subclasses themselves are best defined using an enumeration. We may add an attribute to `Actors`, say "`type`", and let its value be one of {`regular, leadg, character, support, extra`}.

Careful management of classification rules is needed if we wish to ensure absence of overlap and completeness of coverage. Coverage can always be achieved by adding a `type` category of `other`, but that solutions is rarely satisfactory for subsequent automated processing. The `other` category may be helpful to collect temporarily objects for which new classification rules need to be written.

We use intensional definition of classes as the primary approach in database design, and deal with problems that arise as needed.

xq

3-2 ENTITIES AND RELATIONSHIPS

In order to build a model we have to define the internal structure of the entities and the relationships between the entities. The potential users, and their management, are the source of the information. They alone can tell the database designers what data describes the world of their concern. As the designers formalize those descriptions they will develop an understanding and be able to contribute to the specifications as well. To initiate communication we will start with some high-level, informal abstractions.

> Entities and their relationships define the objects of concern for the database users and designers.

To aid in this communication graphics are essential.

3-2-1 The Entity-Relationship Model

A popular tool to elicit specifications from users is the *Entity-Relationship* (ER) model. It associates graphic symbols with the entity and relationship concepts. Figure 3-6 presents some of the entities and relationships for the application area of our examples.

200pt Figure space

new: Square boxes for `Movies(M)`, `Famous_actors(A)`, `Directors(D)` `Studios(S)`. Diamond boxes and lines marked with the cardinality between A,M: `m Perform_in m` ; — M,D: `1 Made_By m`; — D,S: `m Work_for 1`; — A-S: `m Are_engaged_by m`; — S-M: `1 Markets m`.

Figure 3-7 An ER model for movie-making.

This initial sketch does not yet provide enough information to build a database. To each box we must attach the properties needed to represent the entites, for our `Actors` these were the `name`, `fee`, etc. Some relationships also have properties. Since `Actors` can work for multiple `Studios`, we better keep properties {`since`, `untill`} with the relationship `Are_engaged_by`.

If there are few properties they can be placed within the diagram, the convention is to put the names into ovals attached to the class. For a realistic database, which is likely to have hundreds of properties, it is best to create a separate listing, a *data dictionary*, which lists all the properties and indicate where they belong.

Other useful information in an ER diagram is the cardinality of the relationships. We have indicated with the letter `m` where *many* objects may be related to one (1) object on the other side. Notice that this model prohibits one `Director` from working for more than one `Studio`. The user has to validate such semantics, and inform the designer that this constraint is realistic.

An Entity-Relationship diagram, as shown in Fig. 3-6 helps the users and designers to obtain an initial general understanding of the scope of the database. The graphic symbols provide rapid insight into the relationships. The properties only convey some static details of the intended database. There is still not enough information to begin programming for the database. For instance, in the example, the meaning of the relationship `Are_engaged_by` is still quite poorly defined.

ENTITIES AND RELATIONSHIPS AS SUPER-CLASSES We now define further what we mean by entities and relationships. These definitions remain informal because the concepts, as we use them, are informal.

> Entities and relationships are two *super classes* which we use to classify all classes found in our database.

1 Entities define classes where the instances are identifiable objects
2 Relationships define how classes, and hence their instances, are linked to each other

In the application area chosen as an example, the movie industry, we find `Movies`, `Actors`, `Directors`, and `Studios`. The relationships among these are that both `Actors` and `Directors` cooperate to make `Movies`, and that `Directors` `Work_for` certain `Studios`, and that the `Studios` `Market` the `Movies`.

3-2-2 Entities

An entity is a class containing similar objects. All objects that are members of a class should satisfy the defined intension for that class.

The extension of an entity is the corresponding set of identifiable instances of objects with similar properties. We commonly associate with entities definite properties as `location`, `time` of existence, etc. If the values of a property within an entity class are identical they can be omitted: all `Directors` are `human`.

New entity objects can be created, modified, or destroyed, but they keep their identity. `Actors` in the database can change `hair_color`, but we will not change their `stage_name`, by which they are identified in our model. An `Actors` can stop being an `Available_actor`. It would depend on the application if the model should continue to represent former `Actors` as objects within the entitity class `People`.

3-2-3 Relationships

Objects define much of their meaning by participation in relationships. A person who never `performs_in` something will not be a member of the natural class of Actors. Properties we commonly associate with relationships are frequency, intensity, time of validity, etc. Relationships among objects can change: a Director may work for another Studio without the essence of either the Director or the Studio changing.

3-2-4 Properties

Entities and relationships are characterized by their attributes. The properties associated with entities and relationships are encoded as attribute-values in the records of the database files. Properties are also used to establish links, by letting a property refer to specific objects, as the

Director="Curtiz"—— works_for—— Studio="Warners".

Figure 3-8 shows some properties of our entities and relationships. Properties may be represented by numbers, character-strings, pointers, or specialized data types, as: money, time, and location. Time is a property type of special interest, and we will discuss temporal processing in Chap. ¡4-5¿. Representation choices for data are covered in Chap. ¡6-x¿.

Entities

```
Actors    {stage_name, born, sex, typecast, agent, address, ...}
Directors {ref_name, first(_)name, born, wwH, address, ...}
Movies    {title, year, director, producer, actors, studio, prc, ...}
Producers {name, born, wwH, studio, address, ...}
Studios   {name, city, country, film_production, ...}
```

Relationships

```
Cast      {Movies.actors, Actor.stage_name, role, fee}
Made_by   {Movies.director, Directors.name}
Works_for {Producers.studio, Studios.name, since, until}
Markets   {Studios.name, Movies.studio, since, until}
```

Figure 3-8 Properties for entities and relationships.

Properties of concern are, among others, the description of the Movies, with the date, the participants, the process (prc), and awards, as shown in Fig. ¡2-5¿. For the participants we need the names, the typecasting and particulars of the Actors. We may want to add the movie-making history of Famous_actors and Directors, and financial information about the Studios. The relationships will have temporal properties (since, until) if they are likely to switch entities. As we refine our applications many more properties will be specified.

We now formalize the concepts of entities and relationships into the relations and connections of the structural model.

3-3 COMPONENTS OF THE STRUCTURAL MODEL

To design a database design we must model the components which will represent the information to be stored about entities and their relationships, The model must allow the manipulation of the conceptual building blocks for the database, since designing requires synthesis and analysis of many alternatives for representation. The effects of such manipulations must be predictable, so that the model must be formal.

The abstractions must be able to represent those database semantics which are needeed by the designer to develop specifications adequate for definition of the structure of the database. That structure defines the intension, the extension is filled in after implementation. Only when the structure is well understood can a suitable file design be chosen.

The model we use for the design task is the *structural model.* It is the tool used to bridge the gap between potential users, for whom ER concpts are comfortable, and implementors, who need much more specific information.

3-3-1 The Structural Model

The structural model is based on a well-known formal model, the relational model, which defines *relations*, sets of data, and the operations needed to manipulate them. The structural model augments the relational model with a formal definition of relationships between relations, called *connections.* Connections permit specifications of structural constraints between relations, so that the integrity of the database can be maintained.

The structurally relevant semantics of the database is encoded into the connections. We distinguish four types of connections:

1 Reference, to refer to entities which further describe a property. An example is `Director ≻—— Works_for——  Studios.`

2 Ownership, to decribe properties of an entity in more detail. An example is `Famous_actors —— Got——∗ Some_awards.`

3 Subset, to link general classes to their subclasses. In our example we had `Actors —— Is——∋ Leading_actors.`

4 Identity, to define derivations of data. An example is counting stored objects, perhaps by some other property, and placing the results into a new relation: `COUNT(Movies BY director) —— Becomes——≡ Director.productivity.`

These connections will be defined in in Sec. 3-5.

We stated informally earlier that the meaning of entities is largely defined by their participation in relationships. We can now state that the semantics of relations are largely determined by their participation in connections. We can use their connections to classify relations into five types, which have distinct structural features, as summarized in Sec. 3-¡3-9¿.

The similarity of the terms *relations* and *relationships* is unfortunate, but so well established that trying to change it would create further confusion now. It is well to recall that these terms have different origins, and that they are used at different levels of abstractions, as shown in Table 3-1. More terms will be added later.

Table 3-1 Entity-relationship (ER) and structural model (SM) terminology.

Abstract layers		Implementation layers	
ER model	Structural model	Software	Hardware
Entity	Relation	File	Disk
Object	Tuple	Record	Block
Relationship	Connection	Linkage	Pointers
Property	Attribute	Data field	Word or Byte

The mappings between these layers are not one-to-one. Modeling of an entity may require multiple relations, and relations may require multiple files for their storage, or may be combined.

3-3-2 Relations as a Model for Entities

A *relation* represents a simple entity class. As shown in Fig. 3-9, it is composed of a header, the relation schema, and the relation proper, which is a corresponding table composed of tuples, one tuple for each instance of an *entity object* being represented. The relation schema describes the intent of the relation, and the table is the extension.

```
Hitch_movies RELATION:    -- US feature films by Alfred Hitchcock
 film :⟩ title,                        director, studio,      year, prc;
 ____   ______________________________ _________ ____________ ____  ___
 H33  Rebecca                          Hitchcock Selznick      1940  b-w
 H34  Foreign Correspondent            Hitchcock U.A.          1940  b-w
 H35  Mr. and Mrs. Smith               Hitchcock RKO           1941  b-w
 H36  Suspicion                        Hitchcock RKO           1941  b-w
 H37  Saboteur                         Hitchcock Universal     1942  b-w
 H38  Shadow of a Doubt                Hitchcock Universal     1943  b-w
 H39  Lifeboat                         Hitchcock Fox           1943  b-w
 H42  Spellbound                       Hitchcock Selznick      1945  b-w
 H43  Notorious                        Hitchcock Selznick      1946  b-w
 H44  The Paradine Case                Hitchcock Selznick      1947  b-w
 H45  Rope                             Hitchcock Transatlantic 1948  col
 H48  Strangers on a Train             Hitchcock Warners       1951  b-w
 H49  I Confess                        Hitchcock Warners       1952  b-w
 H50  Dial M for Murder                Hitchcock Warners       1954  col
 H51  Rear Window                      Hitchcock Paramount     1954  col
 H52  To Catch a Thief                 Hitchcock Paramount     1955  col
 H53  The Trouble with Harry           Hitchcock Paramount     1956  col
 H57  The Man Who Knew Too Much        Hitchcock Paramount     1956  col
 H61  The Wrong Man                    Hitchcock Warners       1957  b-w
 H65  Vertigo                          Hitchcock Paramount     1958  col
```

H69	North by Northwest	Hitchcock	MGM	1959	col
H73	Psycho	Hitchcock	Paramount	1960	b-w
H79	The Birds	Hitchcock	Universal	1963	col
H80	Marnie	Hitchcock	Universal	1964	col
H81	Torn Curtain	Hitchcock	Universal	1966	col
H82	Topaz	Hitchcock	Universal	1969	col
H84	Family Plot	Hitchcock	Universal	1976	col

Figure 3-9 A sample entity relation with its relation schema.

Each tuple in a relation represents some distinct object, here an instance of a `Movie` and expresses similar properties by its attributes. The distinctness of objects is assured by only permitting distinct tuples within a relation.

Object identifiers In Fig. 3-9 an object-identifier (`film`) has been added to each tuple to avoid possible confusion of `titles`. Consider the `film=H57`, and that Hitchcock directed, in 1934, a predecessor movie in England also titled "`The Man Who Knew Too Much`", now assigned the value for `film=H26`. All other fields depend on this, *ruling attribute**. The field `title` is used for queries, it contains the character-string by which we know the films. If we retrieve multiple objects another selection is needed to determine a unique object.

Attribute domains The choice of values for many of the attributes is limited to a specific *domain*. The field `studio` should only contain recognizable studio names, the `year` field for `Movies` should be greater than `1895`, and in the `prc` field we permit only two choices: {`b-w, col`}. Since the `director` of all these `Hitch_Movies` was "`Hitchcock`", this attribute could have been omitted.

In Fig. 3-10 we show another relation, `Studios`, which augments the information given in the `Movies` relation. This type of relation is referred to as a *foreign* entity relation, to distinguish it from a primary entity relation, as `Movies`. Figures 3-9 and 3-10 also show the notation we use for relations. We refer to the set of ruling attributes as the *ruling part* of the relation and the remaining attributes as the *dependent part* of the relation.

$$\text{Relation\_name RELATION: ruling\_part :}\rangle\ \text{dependent\_part;} \qquad 3\text{-}1$$

In Sec. 3-4 we define relations formally.

US_studios RELATION:

name :⟩	company,	location,	first_yr	last_yr;
U.A.	United Artists	Hollywood	1919	1983
MGM	Metro-Goldwyn-Mayer	Culver City	1920	1983
Universal	Universal Studios	North Hollywood	1921	1962
Warners	Warner Bros.	Burbank	1923	today

* An alternate term often seen in the literature is *key*. But the word key is also used for queries, as `SELECT film=H57`, and we distinguish these functions.

Columbia	Columbia Pictures	Burbank	1924	today
Paramount	Paramount Corp.	Los Angeles	1925	today
RKO	Radio Corp. of America	Hollywood	1929	1955
Fox	20th Century Fox	Century City	1935	today
Selznick	Selznick	Hollywood	1936	1944
MGM-UA	Metro-Goldwyn-Mayer	Hollywood	1983	today

Figure 3-10 A foreign entity relation.

3-3-3 Connections as a Model of Relationships

Connections formalize relationships among relations. They are more specific than relationships among entities. They always connect two relations, and the destination of a connection is always the ruling part of a relation.

For example, we can now define a connection between `Hitch_movies` and `US_studios`. Just as a relation has an extension consisting of a set of tuples, a connection will have a set of connection instances or *links*. A link exists when two tuples, each in relations for which a connection is defined, satisfy the rules for the connection. The most critical constraint is that the attribute values along the connection must match. Figure 3-11 shows the connection `Made_in` and links for selected `Hitch_Movies` made prior to 1950. We restrict here the `Movies` relation shown to the `film` code and `studio`, and the `Studios` to `name` and `location`.

$\boxed{\texttt{Pre50 movies}}$ —— `Made_in` >—— $\boxed{\texttt{US\_Studios}}$

$\underbrace{\text{tuples}}$ $\underbrace{\text{links}}$ $\underbrace{\text{tuples}}$

Pre50 RELATION: film :⟩	studio;	Made_in CONNECTION: studio >—— name;	US_studios RELATION: name :⟩	city;
H32	Selznick	Selznick—Selznick		
H33	U.A.	U.A.—U.A.	U.A.	Hollywood
H34	RKO	RKO—RKO	MGM	Culver City
H35	RKO	RKO—RKO	Universal	N. Hollywood
H36	Universal	Universal—Universal	Warners	Burbank
H37	Universal	Universal—Universal	Columbia	Burbank
H38	Fox	Fox—Fox	Paramount	Los Angeles
H41	Selznick	Selznick—Selznick	RKO	Hollywood
H42	RKO	RKO—RKO	Fox	Century City
H43	Selznick	Selznick—Selznick	Selznick	Hollywood
H45	Transatlantic	Warners—Warners		
H48	Warners			

Figure 3-11 Some links for a connection.

Not all tuples are linked. Some `studios` (`MGM, Columbia`) were not used by `Hitchcock` during that time and one `studio` was across the Atlantic Ocean.

Missing links can lead to incomplete information. If we wanted to know all cities in which `Hitch_movies` were made "`London`" would not appear. Because in most data-processing situations we must care about completeness we define constraints with our connections. Section 3-5 defines the constraints which are attached to our connection types. For the `Made_in` connection a reference connection would be appropriate. It requires existence of the referent, here a `Studios.name`. Then we must either omit "`H45 (Rope)`" or enter "`Transatlantic`" into the relation `US_studios`. The latter change modifies the semantics of the relation, so we better rename it to just `Studios`.

Figure 3-11 also shows the elements needed to specify a connection,

Connection_name CONNECTION:
Source_relation.attributes $\xrightarrow{?}\!\bullet$ Destination_relation.attributes 3-2
[Extra-constraints];

The symbol marked "?" denotes the type of the connection. We cover connections in more detail in Sec. 3-5.

3-3-4 Use of the Structural Model

When all the connections, relations, and attributes for the database have been collected, we have a structural model of the database, which should satisfy the semantics specified by the user.

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*.

3-4 RELATIONS

We now formalize the super type *entity* using the concept of *relation*. This section can be skipped by readers who only wish to understand modeling intuitively.

Relations are not restricted to the structural model. An extensive theory of relations and dependencies, which determine their configuration, has been developed. We restrict our coverage of this theory to those topics relevant to database design. Relations as used in the structural model obey all the requirements placed on them by the theory.

3-4-1 Definition

Relation tables are a set of tuples. Tuples are formed by taking one value for each attribute out of the *domain* corresponding to this attribute. By using names in the model the order of the attribute columns will be of no concern. Any column in the model is identified by giving `Relation.attribute`.

Each of the attributes A_i is bound to a domain D_i, which specifies the set of permissible data elements. In the theoretical model any such set D_j is finite. In the relation schema we find a name for each D_i. More than one attribute may use the same domain, as seen in Fig. 3-10, where `first_year` and `last_year` are both from the domain `date`. In Fig. 3-14 we have `director` and `producer`, sharing the domain `movie_people`.

A relation R with attributes A in its relation schema can then be defined as a subset of the cartesian product of the underlying domains $D(A_i)$

$$R\{A_1, A_2, \ldots, A_a\} \subset D(A_1) \times D(A_2) \times \ldots \times D(A_a) \qquad 3\text{-}3$$

and a tuple is one *row* of such a relation.

3-4-2 Dependencies

Before we can structure data into relations we must introduce *dependencies*. Relations are structured to represent functional dependencies correctly. We will encounter other dependencies later ¡in this section¿.

An attribute B is functionally dependent on an attribute A, if for each value A only one value of B can exist. In our `Movies` examples the attributes `title`, `director`, `studio`, `year`, and `prc` are functionally dependent on `film`. Our notation for this dependency is $A :\rangle\, B;$. An example was shown in the relation schema of Fig. 3-9. Often attributes depend functionally on two or more attributes $A_1, \ldots, A_r$; for instance, in `Casts` we find `actor, movie_title :⟩ typecast, fee_received;`.

A functional dependency *FD* is defined as follows:

$FD(A_{1\ldots r} \rightarrow B_i)$ if the value B_i in any tuple is always fully determined by the set of values $A_1, A_2, \ldots, A_h, \ldots A_r$, where A_h, B_i are simple attributes.

So that

$$FD(A_1, A_2, \ldots, A_h, \ldots, A_r) :\rangle\, B_1, B_2, \ldots, B_i, \ldots, B_d \qquad 3\text{-}4$$

Relations that obey this rule are *normalized*. Specifically, all relations that we deal with in the chapters following will be in *Boyce-Codd normal form*, as defined in Sec. 3-4-¡8¿. In Chap. 5 we describe the process of normalizing models.

Dependencies and Relation Attributes Dependencies are derived from the intensional semantics of the relation schema. This means when we now discuss attributes we only consider the model; any objects in the extension must satisfy the rules we formulate. Attributes in relations can now have only one of two functions:

1 Ruling attributes are independent, that means they are not functionally dependent on any other attributes in the relation.

2 Dependent part attributes of a relation depend on the ruling part of the relation for their definition and existence.

No other attributes occur in BNCF relations.

¡¡KEEP this here or move it Chap 5 ?? ¿¿

‡‡‡‡‡ **Proving Functional Dependencies** A dependency is derived from reasoning about the data; it cannot be deduced from any given collection of data. A dependency can be disproved by testing a database which is known to be factually correct.

If we find that for $C \leftarrow D$ the database has tuples as shown in Fig. 3-12, then some correction must be made.

```
Too_many_directors RELATION:
film :⟩ title,               director,       studio,         year, prc;

  H1 Always Tell Your Wife  Seymour Hicks  Famous Players 1922  b-w
  H1 Always Tell Your Wife  Hitchcock      Famous Players 1922  b-w
```

Figure 3-12 Violation of a given functional dependency.

In this case we will eliminate "Seymour Hicks" by defining the meaning of `director` to be the person who was director at the time the film was finished. If we wish to permit multiple directors we must reconstruct the model through normalization, as shown in Sec. ¡3-4-2¿.

If having two directors is reasonable, but there will never be more, we can assign two fields in the relation schema here: `director_1` and `director_2`. Obviously this is not a general solution. We will deal with multiple dependent values by extending the concept of dependencies and then use it to build better models. ‡‡‡‡‡

MULTIVALUED DEPENDENCIES A *multivalued dependency* occurs when an attribute value determines a set of multiple values. We consider primarily a particular type of multivalued dependency, namely an *inclusion dependency*. We encountered a potential inclusion dependency above, in Fig. 3-12, where multiple `directors` were associated with one `Movie`. If we wish to insert `Hitchcock`'s first movie into our database, the existing definition using the `Movies` relation is inadequate.

The problem with incorrectly normalized relation is the redundancy that ensues in the extension. The information that `Movie: H1` is titled `Always Tell Your Wife`, and other functionally dependent information as, `studio`, `year`, and `prc`, now appears twice.

But multiple values are common for some attributes. A `Movie` entity includes `actors`, of which there are may be many. Figure 3-13 shows the case for two films. The information set in italics is redundant.

```
Actors TABLE:
film :⟩ title, director,  studio,     {actors}*,       year, ...;
```

H29	Sabotage	Hitchcock	Lime Grove	Oscar Homolka	1936	...
H29	*Sabotage*	*Hitchcock*	*Lime Grove*	Sylvia Sidney	*1936*	...
H29	*Sabotage*	*Hitchcock*	*Lime Grove*	John Loder	*1936*	...
H29	*Sabotage*	*Hitchcock*	*Lime Grove*	Desmond Tester	*1936*	...
H29	*Sabotage*	*Hitchcock*	*Lime Grove*	William Dewhurst	*1936*	...
H29	*Sabotage*	*Hitchcock*	*Lime Grove*	Sara Allgood	*1936*	...
H37	Saboteur	Hitchcock	Universal	Robert Cummins	1942	...
H37	*Saboteur*	*Hitchcock*	*Universal*	Norman Lloyd	*1942*	...
H37	*Saboteur*	*Hitchcock*	*Universal*	Priscilla Lane	*1942*	...
H37	*Saboteur*	*Hitchcock*	*Universal*	Otto Kruger	*1942*	...
H37	*Saboteur*	*Hitchcock*	*Universal*	Alma Kruger	*1942*	...

Figure 3-13 Extension with an inclusion dependency.

Each of the actors may have attributes that are functionally dependent on the actors role in the movie. Here `Oscar Homolka` and `Norman Lloyd` play the saboteurs, and `John Loder` and `Priscilla Lane` the saviours. Representing such dependencies in such a single table is difficult.

Actually, a `Movie` entity, as shown in Fig. 3-8, includes also `producers`, of which there also may be several. All combinations represent valid information. Figure 3-14 enumerates the valid combinations for a single film. The semantics of the information shown is the group of people working together.

`Working_together ENUMERATION:`

film :⟩	title,	director,	{producers}*,	{actors}*,	year,	...;
H32	Jamaica Inn	Hitchcock	Pommer	Charles Laughton	1939	...
H32	*Jamaica Inn*	*Hitchcock*	*Pommer*	Maureen O'Hara	*1939*	...
H32	*Jamaica Inn*	*Hitchcock*	*Pommer*	Leslie Banks	*1939*	...
H32	*Jamaica Inn*	*Hitchcock*	*Pommer*	Emlyn Williams	*1939*	...
H32	*Jamaica Inn*	*Hitchcock*	*Pommer*	Robert Newton	*1939*	...
H32	*Jamaica Inn*	*Hitchcock*	Laughton	Charles Laughton	*1939*	...
H32	*Jamaica Inn*	*Hitchcock*	*Laughton*	Maureen O'Hara	*1939*	...
H32	*Jamaica Inn*	*Hitchcock*	*Laughton*	Leslie Banks	*1939*	...
H32	*Jamaica Inn*	*Hitchcock*	*Laughton*	Emlyn Williams	*1939*	...
H32	*Jamaica Inn*	*Hitchcock*	*Laughton*	Robert Newton	*1939*	...
	counts:	1	2	5	→ 10	*tuples*

Figure 3-14 Combinations due to inclusion dependencies.

The redundancy in the enumeration shown in Fig. 3-14 increases rapidly as the number of included tuples increases. For i included sets, each of size s_i we need a total of $s_1 \cdot s_2 \cdot \ldots \cdot s_i$ tuples.

More importantly, this representation is not at all clear since the dependencies leading to this representation are not explicit. If we wish to add `Hay Petrie` to the list of `actors` shown for `film=H32` we must, because of the inclusion dependency `film` → `producers`, insert two new tuples. Removing `Charles Laughton` as a coproducer would cause five tuples to be removed, but removing him as an actor would delete two tuples.

The structural model, and other models dealing with this issue, represent inclusion dependencies by an ownership connection between a main relation and nest relations for each inclusion dependency. Figure 3-15 shows the desired relations for the Movie entity thus far. We only permit one director, the actors are listed in Casts and the producers in Products.

```
Movies_info RELATION:
    film :⟩ movie_title, director, year, studio, prc;
        ——* Products RELATION:  films, producer:⟩ ;
        ——* Casts RELATION:  films, actor:⟩ role_type, fee_received;
```

Now the tuple count for the Jamaica Inn *object is 1 + 2 + 5 → 8.*

If we also wish to consider that an actor (as Peter Sellers) may play multiple character in a single movie, we can change Casts and add another nest relation Characters:

```
    Casts_M RELATION:  films, actor:⟩ fee_received;
      ——* Characters RELATION: films, actor, character:⟩ role_type;
```

Figure 3-15 Relations without internal inclusion dependencies.

An operation, the *natural join*, which combines relations, is described in Sec. ¡4-4-4¿. It permits re-creation of the enumeration when it is needed. The owned relations can be used independently as well. By listing all entries in Products for some producer, say ="Selznick" all the movies he helped produce will be shown.

The important point being made here is that

> A single relation cannot model both (1) functional dependencies and (2) inclusion dependencies.

All functional depencies can be combined into one relation, and for each inclusion dependency we need a relation.

The inclusion dependency *ID* is then formally:

$$\begin{array}{l}\text{If each set } \{B_{l,1},\ldots,B_{l,s_l}\} \text{ is fully determined by the values } A_1,\ldots,A_j \\ \text{then } ID(A_1,\ldots,A_j) = \{B_{1,1},B_{1,2},\ldots,B_{1,s_1}\},\{B_{2,1},B_{2,2},\ldots,B_{2,s_2}\},\ldots\end{array} \qquad \text{3-5}$$

A set $\{B_{1,1},B_{1,2},\ldots,B_{1,s_1}\}$, say, the producers of a Movie entity defined by A, is independent of other sets, say the actors, $\{B_{2,1},B_{2,2},\ldots,B_{2,s_2}\}$, just as the director, year, and studio of a movie are independent of each other, while each of them are dependent on the identifier (film).

Comparing Dependencies

If there is always only one entry in a set ($s_k = 1$), then $B_{k,1}$ is written B_k and functionally dependent on the $(A_1,\ldots,A_j)$. Functional dependency is hence a special case of multivalued dependency.

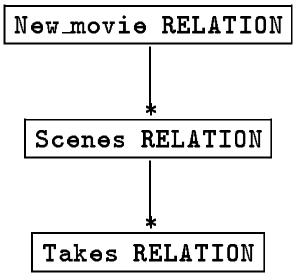
$$\text{If} \qquad FD(A) = B_K \qquad \text{then} \qquad ID(A) = B_l \qquad \text{3-6}$$

Since a *FD* is more restrictive, it is more powerful in database design than an *ID*.

Dependencies within a tuple of a fully normalized relation should be *FD*s. We are concerned only with *ID*s 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, as shown in Fig 3-16. The nest attributes form a *hierarchy*.

A Movie is traditionally composed of a number of scenes, and each scene is filmed as a number of *takes*, which are combined in the cutting room to compose the movie. Some information only pertains to the entire `New_movie`, say the `director`, some information pertains to the scene, as the `locale`, `time_of_action`, etc.
For each of the `Takes` the `reel` of film used, the `position` of the take on the reel, and other recording data must be kept. To keep track of the information we have then:



The ruling part for the `Takes` relation is composed of `film, scene_no, take_no` :⟩ .

Figure 3-16 Levels of dependencies.

Such nested hierarchies are common where large objects have to be decomposed into managable parts.

STRUCTURE VERSUS MEANING Finer distinctions among dependencies 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". Dependencies to other relations can describe the `Studios` where the `Actors` are engaged, the `Countries`, the `Agents`, etc. Our structural model only abstract the structural aspects.

Further knowledge of semantics is essential for processing. *Knowledge-based systems* attempt to broaden the scope of represented semantics to provide assistance in processing the databases beyond the task of data retrieval. ######

3-4-3 Ruling and Dependent Parts

We now have assigned attributes to relations according to the dependencies that were recognized. 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*.

Since relations are also mathematical sets, the tuples of a relation *must* be unique. Given our definitions, this means that the ruling part must be unique. In Fig. 3-17 this meant that three attributes are needed for a `Take`. At times it may mean that the ruling part includes all the attributes and the dependent part is null. This was the case in the `Products` relation of Fig. 3-15.

Uniqueness of tuples in a relation is a desirable property. It avoids complications that multiple objects respond to an operation, no matter how precise the specification is made. Having ruling parts which are *unique* imprctely makes the tuples themselves unique. Referral by the ruling part alone will be sufficient to retrieve the tuple which represents the objects of interest. Dependent attributes, say, the `income` due to a `film` are single values and easily incorporated into further computations.

Identifiers If common `names` or `titles` of objects cannot be taken to be unique then we create unique *identifiers*. In our examples, the unique identifier `film`, permits us to distinguish the two `Hitchcock` movies named "`The Man Who Knew Too Much`". The two objects are assigned the distinct identifiers `H26` and `H54`.

Having identifiers also addresses the problem associated with synonyms. For instance, film `H29` was originally titled "`Sabotage`", but appeared in the USA as "`The Woman Alone`". If access by all titles is important a new relation `Movie_titles` can be defined, which has as only dependent attribute the `film_id`. All actual dependent attributes are kept in the primary relation `Movies`, with the ruling part being the unique identifier `film`.

The user cannot be expected to know these identifiers. A search will use as key a term to match `Movie.title`. If an ambiguity arises further information, as the `year`, the `studio`, a `producer`, or the media `process` used can resolve the identity of the object.

Creating Identifiers When a new object is to be entered, a new identifier must be provided. The database system may provide a function to provide a unique identifier according to some prespecified rules, or it may simply check that the user provides an identifier which is not yet in the database. Updating to identifiers may be disallowed.

‡‡‡‡‡ **Surrogates** Modification of identifiers can have serious consequences, since cross references and uniqueness may be affected. Use of internally generated identifiers, *surrogates*, has been proposed as a solution. They are hidden from the users.

Now, instead of the system verifying the uniqueness of an object identifier the system must create a surrogate. The user is likely to still use some visible attribute as object identifier and must, as before, assure that objects can be easily found. The user is free to modify that identifier, but must realize that other information is linked according the surrogate chosen by the system, and that a modification of the identifier has no structural effect.

For instance, modifying the identifier of "`Saboteur`" to `H29` will still retrieve via surrogates as `Actors Robert Cummins` and `Priscilla Lane`, even though they continue to be shown in the `Casts` relation with identifier `H37`. An explicit operation to join the tuples on the `film` attributes will retrieve `Oscar Homolka`, `Sylvia Sidney` and other actors of the film `H29`, `Sabotage`. ¡¡refer to Fig. 3-13, or its result¿¿

The extent to which surrogates are helpful is an open question. Few DBMS's provide them, and most systems handle identifiers using visible attributes. A constraint "UNIQUE" may be provided so that the system can assure that any value appears only once in a relation. ######

Aggregate Objects For *objects which are not individually distinguishable* a single tuple and a count are an appropriate representation, as shown in Fig. 3-17. In that case deletion becomes decrementing the count and insertion becomes incrementing the count.

As part of the inventory of the supply room of the studio we have supply items that are not individually marked.

Supplies RELATION:

supply_item :⟩	count,	min_stock,	order_size,	supplier,	deliv_time;
32mm film	145	56	30	Kodak	7
64mm film	12	13	20	Kodak	30
500W HI lamps	66	75	12	GE	14
....					

Figure 3-17 Relation for aggregate objects.

Such *aggregate relations* do require special treatment in some cases. In general we focus on relations where the tuples identify single objects, events, or abstractions.

BOYCE-CODD NORMAL FORM We composed our relations to satisfy criteria of consistency and non-redundancy relative to known functional dependencies. Such relations will be in *Boyce-Codd normal form* or BNCF. The conditions that BNCF relations satisfy are

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

¡¡Said earlier in FDs? and also in Chap 5.¿¿

The definition of BNCF relations requires that each attribute is associated with a distinct set of attributes, its particular ruling part. For each ruling part we will have a relation. The order of attributes within the ruling parts does not matter. No attribute will appear in more than one dependent part, but some attributes will appear in ruling parts of other relations as well.

The ruling part of a BNCF relation 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. This means that when an identifier, as film, is in the ruling part of a relation, the object's name, as title, should not also be in the ruling part, since identifiers are assigned uniquely.

How to construct BCNF relations All relations used in the structural models must satisfy BCNF.

There are three prime ways to achieve BCNF:

1 Synthesis: One collects information about the dependencies. A table is made of all attributes to be included in the database. For all attributes the dependencies on other attributes are listed. That table is used to synthesize the relation schemas. Dependencies among relations give rise to connections.

2 Decomposition: We collect information about the entities and relationships. All attributes for the the entities and relationships selected for the database are listed. Relationships are formalized into connections, and their attributes, if any, are removed to relations. Attributes that have inclusion dependencies are seperated into distinct relations, and connections are used to note the remaining constraints. Attributes that still violate BNCF are also removed to distinct relations. If relations created in that way have matching ruling parts and object sets they may be combined. The process iterates until we have fully normalized relations and connections.

3 Constructive: One formulates immediately normalized relations. These must be verified and changed if they are not in BNCF. Then one defines connections among relations according to semantics that are collected separately.

The synthetic approach ¡is¿ reviewed in Chap. ¡4-x-y –¿, as an example of applying Armstrong's rules. The decomposition approach, normalization, is presented as part of the database design method in ¡Chap. 5-x¿.

Since we develop the semantics incrementally we follow the constructive approach in this chapter. We can now summarize the major BNCF relations we have developed for our example, and then proceed with the definition of connections.

```
Film_registry DATABASE.
Movies_info RELATION:      -- Interesting movies
    film :⟩ title, year, director, studio, prc, rating, gross_sales;
Directors RELATION:      -- Directors who made interesting movies
    ref_name :⟩ first_name, born, died, origin;
Producers RELATION:      -- Executives in the movie industry
    name :⟩ birth_name, born, died, origin;
Products RELATION:      -- Movies produced with each producer
    films, producer_name :⟩ position;
Famous_actors RELATION:      -- Famous and current actors
    stage_name :⟩ birth_name, first_name, sex, born, died,
                                                  typecast, origin;
Casting RELATION:      -- Actors available for new movies
    actor :⟩ status, height, weight, location;
Casts RELATION:      -- Movies with their actors
    film, actor :⟩ part, role, fee, award;
Studios RELATION:      -- Organizations making movies
```

```
    name :) company, city, country, year_founded, year_of_demise;
  Countries RELATION:     -- of studios, current actors and origins
    country :) language, population, film_production;
```

Figure 3-18 Relation-schemas.

We have made some *assumptions* in our model. If they are wrong, the model must be revised. In the `Film_registry` some of the debatable assumptions are:

1 We recognize only one `director` per `film`.
2 The `ref_name` chosen for the `Directors` is unique.
3 The `name` chosen for the `Producers` is unique.
4 The `stage_name` chosen for the `Famous_actors` is unique
5 Any `actor` in `Casting` is also listed in `Famous_actors`
6 The `name` chosen for the `Studios` is unique.

Stating assumptions explicitly is important. They are as much part of the model as the model itself. Omitting assumptions leads to misinterpretations.

3-4-4 Domains

In the original definition of a relation, given in Sec. 3-4-1, we introduced the concept of a *domain* as defining the set from which values for an attribute are taken. The concept of a domain is important both for theoretical and practical reasons.

If we take domains to be finite, many proof procedures are become feasible. Since domains restrict the operations which can be performed the complexity of managing a database is greatly reduced. We now expand the domain concept with some pragmatics.

The important rules are

1 Each attribute in a database is bound to some domain definition
2 The choice of values for an attribute is limited to that domain.
3 Operations which logically combine attributes require that the domains of these attributes match.
4 Connections can only be defined for attributes that match.

The relation schema now has an additional component: with each attribute we will specify a domain.

Domains and Representation There is a strong similarity between the concepts of domain and representation, which often leads to confusion.

- The domain specifies the values that an attribute may take
- The representation enables the allowable values to be stored

We can only decide on a representation once we know what the domain is. Often a representation permits the storage of values that are not within the domain. For instance, the `Casting.height` may have as domain the range of 20 .. 220 cm. The representation, perhaps a `short_integer`, may permit values from 0 .. 65 535. Another attribute, say, `film_production`, has a range of 0 .. 50 000, and uses the same representation. Although the representations match, the domains do not, and matching operations between these two attributes are questionable. ######

Naming Domains To bind the attributes to some domain we provide a domain name for every attribute. A suitable place to document domains is in the *data dictionary* which we maintain with the model. The domain definitions may use a may be kept in a seperate table or stored as entries in the same data dictionary. With each entry we will eventually show the representation type, or possibly the derivation from another domain definition. Some examples are shown in Fig. 3-19.

Domain		Type	Used by attributes
`count`		`integer`	`Countries.film_production, Rental.days`
`date`		`date_time`	`Movies.year, Directors.born, Directors.died`
`dollars`		`decimal.2`	`Movies.budget, Theatre.income`
`film_code`	`id`	`CN*`	`Movies.film, Casts.films,`
`movie_people`	`id`	`char`	`Famous_actors.stage_name, Casts.name, Movies.director, Producers.name`
`studio_name`	`id`	`char`	`Movies.studio, Studios.name`
`cinema_num`	`id`	`integer`	`Cinemas.id, Receivables.cinema`

Figure 3-19 Domains and attributes.

Domains marked "`id`" have an additional aspect; they are available as *identifiers*, as introduced in Sec. 3-4-3.

DOMAIN TYPES When we analyze the attributes for a model, we find two types of domains.

1 *Value domains*
2 *Reference domains*

Attributes in simple ruling parts typically have value domains, since no further reference is needed. In the dependent part we will find both types. Figure 3-20 shows some examples

```
Casting RELATION:
actor :>      status,        location,      sex,     height,       weight ;
(unique     ) (reference to) (reference to) (code ) (value from) (value from)
(identifier ) (Actor_states) ( Countries  ) ({M,F}) ( 90 .. 220) ( 10 .. 200)
Value dom.    Reference domains             Value domains
```

Figure 3-20 Value and reference domains.

Attributes having a value domain describe characteristic properties of an entity directly. A domain definition, kept with the relation-schema, permits some verification of the correctness values for new entries, as shown in Fig. ¡6-20 schemas¿. Not all values will exist in the database. If an `actor` leaves the profession and the tuple is deleted from the `Casting` relation, the values will also disappear. No instance of `height = 191` may exist after `Laird Cregar` died.

Other attributes describe characteristics of an entity indirectly by reference to another entity. Their domains are *reference domains*. The allowable values in a reference domain are defined by tuples in another, foreign, relation. The domain definition can be changed by an update to the foreign relation.

The `Casting` relation of Fig. 3-20 has two reference attributes.

1 <u>`status`</u> defines the type and pay method of the actor, by refering to an entry in a small relation

`Actor_states RELATION:`

`st_id :⟩`	`actor_status`	`contract`	`hourly`	`on_fee`	`percentage;`
`reglr`	`Regular`	X		X	
`leadg`	`Leading_actor`			X	X
`chrtr`	`Character_actor`	X			X
`suppt`	`Supporting_actor`	X			
`extra`	`Extra`		X		

2 <u>`location`</u> helps locate actors engaged elsewhere. This attribute refers to the `Countries` relation, defined in Fig. 3-18.

If the actor leaves the profession and the tuple is deleted from the `Casting` relation, the foreign `Studios` tuples will remain.

Figure 3-21 Reference type attributes.

Foreign tuples, which define the *domain* of some attriubute, can be manipulated without changing the model. The use of foreign relations in Fig. 3-21, permits that new `Actor_states` and additional `Studios` can be defined as needed. Tuples in those relations will continue to exist even if they are no longer used.

The primary attribute and the foreign relation can be used to establish a *reference connection* in the model. Such a connection will constrain deletion of foreign tuples.

3-4-7 Summary of Components of Relations

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.
- A *relation-schema* lists the attributes for the relation.
- With each *attribute* we identify
  - its participation in the ruling or the dependent part and
  - a *domain* as a set of permissible values
    - by *range definition*,
    - by *enumeration*, or
    - by *reference*.

Table 3-1 recaptulates the terminology we used for the four levels of abstraction which we encountered. Alternate terminology is shown in the index to this book.

3-5 CONNECTIONS

We now arrive at the second super type to be formalized: relationships. We used relations to formalize entities, for relationships we use *connections*. We showed an example of a connection in Fig. 3-11 and announced that connections imply constraints.

Motivation for Connection Constraints We expect attributes in a tuple to have valid values. That expectation includes cases where the values are instances in other, *foreign*, relations. Connection constraints define these expectations.

The constraint appropriate to the `Made_in` connection between `Movies` and `Studios` is given by a *reference connection*. The reference connection provides a constraint to assure that no `Studio` information will be deleted that is referenced by `film` tuples in the `Movies` relation. For instance, we cannot delete the information about the `Studio` named `Fox` while we have records for the `Movie` "`H38` :⟩ `Lifeboat`". This constraint enforces the integrity of the entire object identified by `H38`.

With this motivation we proceed to further detail about connections.

Relating Entities by Attributes Connections are instantiated via attribute values that match. One requirement is then that they belong to the same domain. Domains useful for establishing connections in our model were shown in Fig. 3-19. So that the comparison can work in practice the domains must also be represented by the same type.

A neccessary condition for a connection is that the domains match. But this is not a sufficient condition; not all attributes having matching domains should be connected. The essence of a connection is that there is a well specified relationship, which is expected to remain valid over time. That condition is similar to that leading to relations: we group into a relation all attributes that we expect to be dependent on the same ruling part.

```
Movies —— Producers
Movies —— Studios
```

Figure 3-22 Related information.

GENERAL SYNTAX AND SEMANTICS OF CONNECTIONS A connection formalizes a relationship that exists between two relations. The connections may derive from relationships between the entities of the original Entity-Relationship model or may have been created while entities where transformed to relations in BCNF. Whenever an entity is decomposed into multiple relations connections are created to retain the original information. It does not matter here how the connection was established. If one starts originally with a model which consists of BCNF relations the connections can be specied directly on those relations and their attributes.

A connection defines a semantic relationship among two relations. Links exist when values for the attributes specified by the connection match. In Fig. 3-11 the `Made_in` connection specified the attributes `Movie.studio` and `Studios.name`. Each match leads to a link instance.

> It is immaterial to the model if the link defined by some connection is implemented within the database. The link exists conceptually as soon as attributes for the connection match.

The connection links do not contain any new information:

> A link can be computed from the definition of a connection and the actual tuples of the connected relations.

The format of a connection was given in Eq. 3-2. It is composed of five elements:

1 The Connection_name used for identification and disambiguation when relations are connected in multiple ways
2 The type defines the logical constraints. We recognize four types, as shown in Table 3-2.
3 The source relation and attribute(s)
4 The destination relation and attribute(s) match in number and domain the source attributes. They are also always the ruling part.
5 Any additional quantitative constraints. Those will be presented when needed.

Table 3-2 Connection types in the structural model.

Connection Type	Symbol	Direction
Ownership	$\longrightarrow\!\!\ast$	from single owner to multiple owned tuples
Reference	$\succ\!\!\longrightarrow$	from multiple primary to single foreign tuples
Subset	$\longrightarrow\!\!\!\!\ni$	from single general to single subset tuples
Identity	$\longrightarrow\!\!\!=$	from single source to single derived tuples

We will now review these four connection types; we already encountered the first three types.

3-5-1 Ownership Connections

An *ownership connection* provides links for multiple included dependent attributes. The general form is

$$\text{Name CONNECTION: Owner.ruling\_part} \longrightarrow\!\!\ast \text{ Member.ruling\_attributes;} \qquad 3\text{-}7$$

Figure 3-23 shows multiple `Roles_played` by some `actors` of the `Casting` relation. Any new entry in `Roles_played` needs a corresponding `actor`. If an `actor` is removed from `Casting` all corresponding `Roles_played` information for that `actor` is no longer relevant and should be removed as well.

The ownership connections defines structures to serve inclusion dependencies as discussed in Sec. 3-4-2. The operational semantics of owned tuples are exactly the same as those of dependent attributes: they go away when the owning tuple, which after all defines the object, disappears. Similarily, we can only insert owned tuples if we have an owner tuple.

Figure 3-23 shows multiple `Roles_played` subsequently by those `actors` of the `Casting` relation who were in `Hitchcock`'s last movie, `Family Plot`. Any new entry in `Roles_played` needs a corresponding `actor`. If an `actor` is removed from `Casting` all corresponding `Roles_played` information for that `actor` is no longer relevant and should be removed as well.

$\boxed{\texttt{Casting}}$ —— `Performs_as` ——* $\boxed{\texttt{Roles\_played}}$

Owner relation Nest relation

tuples links tuples

Casting RELATION:		Roles_played RELATION:			
actor :⟩	...;	performer,	film :⟩	role,	year;
Karen Black	...	Karen Black	DCs2	haunted tenant	1976
		Karen Black	PHy6	pilot's wife	1978
		Karen Black	JRB35	inventor's wife	1979
		Karen Black	GKc4	fashionable	1981
		Karen Black	RAl21	cynical Dean fan	1982
		Karen Black	HJg5	shrewd	1983
Bruce Dern	...	Bruce Dern	JFr19	terrorist ??	1975
		Bruce Dern	MRt6	beauty judge	1975
		Bruce Dern	HAs6	doper	1978
		Bruce Dern	WHl2	get-away driver	1978
		Bruce Dern	JTr1	psychopath	1980
		Bruce Dern	SWu1	mad tattooist	1981
		Bruce Dern	JsM1	*	1982
		Bruce Dern	xxx	cult leader	1988
William Devane	...	William Devane	JSl9	Nazi?	1976
		William Devane	JSl10	GI in England	1979
		William Devane	JDs1	homicidal alien	1979
		William Devane	JSl11	promotor	1981
		William Devane	LyL1	craggy	1983
Barbara Harris	...	Barbara Harris	GNe5	unhappy mother	1977
		Barbara Harris	JS5	senator's wife	1979
Ed Lauter	...				
Cathleen Nesbitt	..	Cathleen Nesbitt	FZ20	old woman	1977

Figure 3-23 Ownership.

We refer to each group of owned tuples, as the ¡six¿ `roles` shown for `Karen Black`, as a *nest*. Some nests (`Ed Lauter`'s) are empty. The owned relation (`Roles_played`) is the *nest relation*. The owning relation has one tuple for each owner object, the owned relation as many as needed for all the `roles` listed. Information similar to that shown in the table of Fig. 3-13 is now distributed over two BCNF relations.

Cardinality of Ownership There maybe no, one, or many owned tuples in a nest. The cardinality of ownership is $1 \longrightarrow\!\!* \; n$. Extra constraints can be applied to n. If `Performs_as` requires at least one tuple per nest, we must delete "`Ed Lauter`" or find a `role` for him. Such constraints complicate update transactions; insertions on both the owning and the nest relation must be executed as an *atomic transaction*.

In Fig. 3-15 we encountered ownership connections in a more complex situation. The `Movies_info` relation owns the `Casts` relation, and the tuples of the `Casts` relation are in turn connected to the `Actors` relation. Relations, as `Casts`, that represent relationships between independent entities are called *associations*. An association relation is neccessary to represent relationships which have an $n : m$ cardinality. Two connections will be needed as sketched in Fig. 3-24.

```
Two relations connect to the Casts relation:
     Movies_info  RELATION:     n
                             ————*
                         ————<Casts RELATION: films, actor :⟩ role;
Famous_actors RELATION:      m
The Casts RELATION needs two attributes in its ruling part, one for each connection:
Performers CONNECTION: Movies_info.film———* Casts.films;
Played_in CONNECTION: Famous_actors.name———< Casts.actor;
```

Figure 3-24 Association.

In our example `Casts` is a nest of of m `actors` in terms of `Famous_actors`, and of `Movies_info` in terms of n `films`. Each connection remains $1 : m$ or $1 : n$. The connections in an association need not be of *ownership* type, but could also be *reference* connections, as will be seen in Fig. 3-26.

STRUCTURE OF OWNERSHIP Ownership requires a match of the connecting attributes

$$\begin{aligned} Owner.a_i =& Owned.a_i \text{ for } i = 1, \ldots, j \\ k > j, \quad j =& \| \; Owner.rp \; \|, \quad k = \| \; Owned.rp \; \| \end{aligned} \qquad \text{3-8}$$

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. 3-4-¡6¿. We can now state the rules for ownership connections:

Ownership rules

Structure: The ruling part of the nest relation is the catenation of the ruling part of the owner relation and an attribute to distinguish individual tuples in the nests.

Operation: A new tuple can be inserted into the nest relation only if there is a matching owner tuple in the owning relation. Deletion of an owner tuple implies deletion of its nest.

We see that with the definition of an ownership connection we also prescribe constraints, and rules how the constraint is to be maintained. These rules define the semantics of the model, and once implemented define the way of doing business.

‡‡‡‡‡‡ **Danger of Using Ownership** Ownership implements a hierarchy, where all lower elements depend on on the owner. This structure is easily misused. The frequently used relationship of `Departments` and their `Employees` is rarely appropriate for an ownership connection. One could loose all information about the `Employees` if an administrator abolishes a department. In a reasonable world model `Employees` are the primary entity, and `Departments` are merely referenced abstractions.

If ownership rules are not suitable another connection type should be chosen. A *reference constraint* will only let `Departments` be deleted that have no `Employees`. In our examples we wish to retain information about the `Producers` who made the `Movies` we list in the association `Products`. We show the solution in Fig. 3-26. ‡‡‡‡‡‡

3-5-2 Reference Connections

In earlier examples we encountered relations that referenced other relations. The *reference connection* models such a relationship. The *primary relation* is the source relation of a reference connection the and *foreign relation* is the destination relation. The general form is

Name CONNECTION: Primary.attributes $\succ$—— Foreign.ruling_attributes; 3-9

The primary relation defines the object of interest, say some `Movies_info.film`, and an attribute, say `Movies.studio`. The foreign relation, `Studios`, provides additional information for the `studio` where the `film` was mode.

The foreign object is typically a higher level object or an abstraction. A single foreign tuple can hold information that otherwise would be held redundantly in many source tuples. Many `films` are made in the same `studio`, the address appears only once in `Studios` relation.

To assure that the referenced information is always available, the reference connection embodies the constraint that the foreign tuple must exist. Insertion in the primary relation is constrained, and deletion of a tuple in the foreign is not permitted while it is in use.

Multiple levels of references can be made, and multiple connections can be made to and from relations; however, circular connections will lead to update problems. Figure 3-25 shows connections for a valid network.

Given the relations as shown in Fig. 3-18 we can define

```
Shot_in CONNECTION: Movies.studio >—— Studios.name;
Located_in CONNECTION: Studios.country >—— Countries.country;
Resides_in CONNECTION: Casting.location >—— Countries.country;
Is_from CONNECTION: Producer.origin >—— Countries.country;
Who_is CONNECTION: Products.producer_name>—— Producers.name;
Made_by CONNECTION: Movies.director >—— Directors.ref_name;
```

Figure 3-25 Reference Connections.

In all these cases the reference connection will assure that destination tuples are available to further describe the source objects. The foreign relations are maintained distinctly but cooperatively with the primary relations.

The existence of a tuple in a foreign relation implies only weakly that there exists a tuple with the corresponding value in the primary relation. If there is no reference then the tuple in the foreign relation may be deleted. For instance, in the `Studios` foreign relation we have listed "`Columbia`", but this studio was not used by Hitchcock. and could be omitted from a database which focuses on Hitchcock. A foreign relation can be the source of other connections, but these will not constrain deletion.

The database model forms a network of connections of all types. As indicated earlier, *associations* can also be formed with reference connections.

Given the three relations from Fig. 3-18:

```
Movies_info RELATION: film :⟩ title, year, director, studio, ...;
  Producers RELATION: name :⟩ birth_name, born, died, origin;
  Products RELATION: films, producer_name :⟩ position;
```

and the connection `Who_is` from Fig. 3-25

```
  Who_is CONNECTION: Producer.name ——< Products.producer;
```

we can connect `Products` also to `Movies_info`, making it into an association relation by adding

```
  Produced_by CONNECTION: Movies_info.film ——* Products.film;
```

We can only keep `Products` tuples for those `films` which appear in `Movies_info`.

Figure 3-26 Association with a reference connection.

The model of references is sketched in Fig. 3-28.

STRUCTURE OF REFERENCES A tuple of the foreign entity relation provides a more detailed description for an object in the primary relation. Any attribute of a relation can reference the ruling part of a foreign relation, if its domain matches. Large, unconstrained domains do not lend themselves for referencing. For instance, *real numbers* are quite unsuitable in practice.

The destination of a reference connection is always the ruling part of the foreign relation. Most foreign relations have ruling parts composed of only one attribute; we are used to reference abstractions by simple names. We expect to find tuples in a foreign relation to match any value assumed by the primary attribute; absence of a foreign tuple constitutes an error. A foreign relation hence defines the domain for the primary attribute.

Cardinality of Reference Connections We expect to have many fewer foreign objects than source objects. One entity or abstraction will be named by many tuples. The cardinality is hence $n \succ\!\!\!\longrightarrow 1$, just the opposite of the ownership connection. That reversal made us change the direction of the reference arrow in Fig. 3-26.

We can now define the structural rules.

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

If the rules defined by a reference connection are not obeyed, vital information may not be available during processing, causing transactions to fail.

Self-reference It is possible to define a reference connection that has the same relation as source and destination. The primary attribute will differ from the foreign attribute. Figure 3-27 shows such an extension.

We wish to show costars of `film actors` in our model. The `Casts` relation from Fig. 3-18 is appropriate since it has the proper ruling parts. We add an attribute `costar` and a reference connection.

```
Casts RELATION:           -- shows actors in movies
        actor, film :⟩ part, role, award, costar;
Starred_with CONNECTION: Casts.(costar,film) >—— Casts.(actor,film);
```

Only some `actors` will have a value in `costar`. Others will have a `null` entry. We deal with the domain definition for that case by defining *artificial* entries in domains and tuples, or by defining subsets, as shown in Sec. 3-5-3.

This example also shows a case where the reference connection requires two attributes.

Figure 3-27 Self-referencing connection.

This construct introduces a cycle on the connection level, but should not cycle on the instance level. We will not permit an `actor` to use the `Starred_with` connection to refer to him- or her-self as `costar`, no matter what their personal opinions are.

We can now show the reference network we have defined in Fig. 3-28. Primary referencing relations are shown on the left. Their ruling parts must be accessible from outside of the model.

216pt Figure space

draw new based on Figs. 3-25, 3-26, 3-27. Primary relations are shown on the left.

Figure 3-28 Foreign relations and their referenes.

MAINTENANCE OF REFERENCE CONNECTIONS Errors in the maintenance of foreign relations can easily occur if they are used for multiple tasks or by multiple `views`, as defined in Sec. 3-6. Different views are typically maintained by separate departments in an enterprise. An example of problems with foreign relations is shown in Fig. 3-24.

We use a single `Countries` relation for the `location` of `actors` for `Casting`, the `country` that `Studios` are in, and the `origin` of `Producers`, `Directors`, and `Famous_actors`. Rules have to be established for maintenance of that relation. References may need to remain valid for past entries.

We have `Directors` from "`Germany`", but current `Studios` are either in the "`Federal Republic of Germany`" or in the "`Democratic Republic of Germany`". More recently "`Upper Volta`" became "`Burkina Fasso`".

The dependent part, as `population` or `film_production` becomes awkward to maintain under these modifications.

It is easiest if both old and new values may be retained. However, such duplicate entries can cause errors in summation, say, when computing the world-wide production of films.

Figure 3-24 Sources of errors in references.

Ownership connections have less of a problem, since the dependency is simpler. ######

ARTIFICIAL REFERENCE TUPLES The existence of a value in the primary attribute requires the existence of a corresponding foreign tuple. When a `null` value is permitted in the primary domain, this fact has to be noted in the foreign relation.

This is the case if we wish to expand the `Movies` relation of Fig. 3-9 with all movies for that period, as shown in Fig. 3-30. For one `film` we dont have process (`prc`) information.

```
More_movies RELATION:      -- Non-US films by Hitchcock
 film:⟩ title,             director,  studio,     year, prc, language;
 ----   ----------------   ---------  ----------  ----  ------ --------
 H16   The Manxman         Hitchcock  Elstree     1929  silent n.a.
 H40   Aventure Malgache   Hitchcock  Associated  1944  b-w    French
 H41   Bon Voyage          Hitchcock  Associated  1944  b-w    French
 H46   Under Capricorn     Hitchcock  MGM British 1949  col    Λ
 H47   Stage Fright        Hitchcock  Elstree     1950   *     Λ
 H83   Frenzy              Hitchcock  Pinewood    1972  col    Λ
```

Figure 3-30 A relation with unknown values.

To satisfy the model rules in a consistent manner, we can allow the foreign relation to have a tuple with the value *unknown* in its ruling part, and appropriate values in the primary domain. There are other conditions which may make the use of *artificial reference tuples* desirable. Some useful types of key values for artificial reference tuples are shown in Table 3-3.

Table 3-3 Values for artificial reference tuples.

Code	Value	Function	Action
$*$	`missing`	`unknown`	`"check_with_source"`
Λ	`null`	`none`	`no value needed`
`n.a.`	`none`	`not applicable here`	`ignore`
?	`any`	`placeholder`	`to be computed`

The effect of indicating a `missing` ($*$) value is to defer error messages or inconsistency reports to a later stage, so that 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.

An entry of `null` (Λ) means that no value is needed. This value will appear, for instance, in the date `died` columns for people still alive. It can also be used when default values apply, for instance here `"English"`. It can be modified when conditions change.

An entry of not applicable `n.a.` means no regular value should occur here. We might use that in the `composer` field of `silent` movies. We show another example in Fig. 3-37.

The "any" type of value (?) is useful when constructing query relations; the symbol is to be replaced with an appropriate entry by some computation.

3-5-3 Subset Connections

We often deal with entities that are quite similar, but for which different information is to be collected. In our example we have many kinds of `People` that work in the movies, but the information we should keep on them varies. In Fig. 3-18 we had `Famous_actors`, `Producers`, and `Directors`. There is also some overlap between them, so similar attributes are redundant if we keep these overlapping subsets in wholly distinct relations. Defining a general relation `People` permits non-redundant modeling of shared attributes.

A *subset connection* connects a generalization to a subset. Such relations will have matching ruling parts, but will differ in dependent attributes. The general form is

$$\text{Name CONNECTION: General.ruling\_part} \longrightarrow\!\!\!\ni \text{Subset.ruling\_part} \quad \text{[Membership-constraint] ;} \qquad \text{3-10}$$

In Fig. 3-31 we show the *generalization*, `People`, and the subset connections to previously defined relations. To model this generalization we have to make changes in the way identifying `names` are handled for the various subsets. If we had all their `Social-Security-Numbers` (`SSN`) problems associated with names could have been avoided.

```
People RELATION:      -- general relation.
        last_name, first_name :⟩ sex, born, died, origin, address,
                is_actor, is_producer, is_director;

Is_actor CONNECTION:      -- connection to actors subset
        People.(last_name, first_name) ——∋
            Famous_actors.(birth_name, first_name);

Famous_actors RELATION: -- the real name is the ruling part
        birth_name, first_name :⟩ stage_name, agent, typecast, awards;

Is_producer CONNECTION:      -- connection to producers subset

        People.(last_name, first_name) ——∋
            Producers.(last_name, first_name) Producers RELATION:     -
- name now split into two parts
        last_name, first_name :⟩ current_studio;

Is_director CONNECTION:      -- connection to directors subset
        People.(last_name, first_name) ——∋
            Director.(ref_name, first_name);

Directors RELATION:      -- reference name is unique
        ref_name :⟩ first_name, wwH, cur_fee, cur_perctg, awards;
```

Figure 3-31 Subset connections.

Subsets may have further subsets. We encountered already subsets of `Actors` in Fig. 3-5 and further subsets of `Regulars` in Fig. 3-6.

The use of subsets permits to specify a variety of dependent parts, all for the same ruling part. Tuples in a subset are incomplete without the information provided in the general tuple. We have constraints that are similar to the ownership connection, but here classes are expanded, rather than attributes.

> **Subset rules**
>
> Structure: The ruling part of a subrelation matches the ruling part of its connected general relation. A general tuple may have no or one tuple in any connected subset relation.
>
> Operation: Every subset tuple connects to one general tuple. Deletion of the general tuple forces deletion of the linked subtuple.

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. Subsets also have a role in distribution of data.

3-5-4 The Identity Connection

The *identity connection* recognizes replicated information. Replication of information occurs in databases when

1 derived results are stored in the database
2 copies of information are stored on distributed computers

The identity connection connects multiple attributes in the source relation to a tuple in the destination relation. The values should be identical. The ruling part attributes must always be included so that uniqueness of the destination tuples is assured. The general form is then

```
Name CONNECTION: Source.(ruling_part, dependent_attributes)——=
                 Destination.(ruling_part, dependent_attributes)        3 − 11
                     | constraint_relaxation;
```

The constraint_relaxation plays an important role since without it the identity constraint is quite constricting.

The principle of relaxing the constraint is simply that *eventually* all destination tuples exist for all source tuples. The term "eventually" permits the constraint to be satisfied with *some delay*.

DELAYS IN ESTABLISHING IDENTITY The computation of derived data and the distribution of results require time; hence maintenance of an identity connection requires time. In order to build workable database systems, we must be able to specify what kind of time-delay is permissible, and for each kind specify the amount of delay acceptable. At the same time we must be able to guarantee that *sometime* the identity will be satisfied, so that then computations using the destination values will be predictable. An example of an identity connection, where a delay of a day is tolerable, is given in Fig. 3-32.

We recognize three types of temporal constraints and designate each by a symbol in the `relaxation` field of the identity connection. Examples for the three types of constraints are shown in Fig. 3-33.

- `!` Prior to execution of a transaction named `!Trans_use` the destination tuples in the identity connection have to be set. If no transaction name is specified, then `!ANY` is implied, so that the identity connection must be set prior to *any* use of the destination tuples.
- `@` At a specific time, `@hh:mm:ss`, the identity connection must have been set. Any transaction which starts subsequently will obtain up-to-date destination tuples.
- Δ After a specific interval, Δn `unit`, the identity connection must have been set. The `unit` may range from `days` to `seconds`. If the value of the interval $n = 0$, then the effect is identical to `!ANY`.

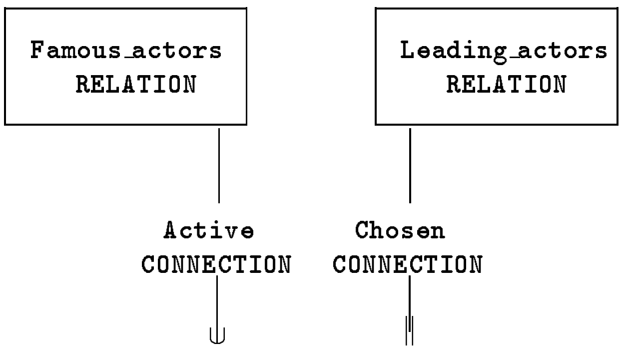
If no relaxation is specified, then the interpretation of the identity constraint is equivalent to `!ANY` as well.

Specifications do not automatically solve the implementation problems. If, during execution, the identity connection is *not* set within the time delay specified, then all subsequent transactions must be held until the setting of the identity connection is completed.

If further changes in the source relation occur before earlier ones are completed, then setting of the earlier ones must be completed before the later ones are set. How these conditions are to be achieved is not of concern in the model. Since most of the implementation problems arise in distributed systems, we present those issues in Chap.¡x-y¿. We also present there the case of symmetric identity (==).

In the `Movie_buff VIEW` we have a
subrelation of `Famous_actors` —— `active`——⊃ `Active_actors`.

When, for a new movie, in the `Movie_making VIEW`, `Leading_actors` are chosen, those actor should also be placed into the relation `Active_actors` because we can expect that soon inquiries will be made about them.
But some delay may be needed to acquire all the data needed for the corresponding `Famous_actors` general relation. The model is graphically:



```
Active_actors
  RELATION
```

We allow two days for the update and specify:

```
Leading_actors.(name, born, role) —— Active——≡
            Active_actors.(stage_name, born, typelast)|Δ2d;
```

Especially when identity connections are among distinct views the time constraints tend to be relaxed.

Figure 3-32 Use of an identity connection to link views.

Many financial transactions require that input data are up-to-date:

```
Studio.employee.(name,hours_worked)——≡
        Payroll.(name,hours_to_be_payed) | !pay_transaction ;
```

Before schedules can be displayed in the morning all committments must be accounted for:

```
Actors_schedules.(stage_name, title, character) ——≡
        Studio_display.(name,project,as) | @8:00:00;
```

Changes to the schedule for today should be rapidly displayed, five minutes is the limit:

```
Change_list.(stage_name, title, character) | date = today ——≡
        Studio_display.(name,project,as) | Δ5m;
```

Here a simple computation, selecting only changes applicable for today, is specified as well.

Figure 3-33 Three types of delay specifications.

Fig. 3-33 also shows an example of a simple computation. In general, the replicated information at the destination need not be a literal copy of the data as the source. Especially if the destination is a central node in a network only selected and aggregated information should be transmitted from the sources. The source nodes can keep much more detailed information themselves, as shown in Fig. 3-34.

Cinemas transmit weekly the gross receipts to the corporate office:

```
Cinema_receipts.(id, SUM(receipts | day = today-7 ..  today-1))——≡
                Total_receipts.(theater, amount) | @Monday 8:00:00;
```

Figure 3-34 Delayed aggregation.

Replacing Programmming with Declarations The identity connections we show here specify simple computations. Networks of such connections can describe complex computations, which are now scheduled and managed wholly by the system, rather than by explicit transaction programs. We are starting here a new approach to *declarative programming*.

3-5-5 Naming of Connections

It can be useful to attach names to connections which identify their *role*. There are typically two candidate role names for a connection, since we use distinct terms for each direction that can be traversed. Since connections have a defined semantic direction we choose in the model the name appropriate for that direction. For instance `Made_in` connects `Movies.studio`≻—— `Studios.name`.

The simpler rolename `Made` goes from `Studios` to `Movies` and may be used when querying the database.

Role names are not limited to connections among relations, but may also be used for simple attributes - which have a 1:1 dependency on their ruling parts.

Role names are useful in enhancing the semantics of the database model, and are easily related to the queries that the database will respond to.

Figure 3-35 shows some examples of role names.

Connection name	source = destination role name →	← role name	type
Made_by	Movies.director=Directors.ref_name		ref
	Made_by	Made	
Shot_in	Movies.studio=Studios.name		ref
	Shot_in	Shot	
Located_in	Studio.country=Countries.country		ref
	Located_in	Has	
Cast	Movies.film=Casts.films		own
	Cast_is	Performed	
Played_in	Famous_actors.name=Casts.actor		own
	Played_in	Has_actor	
Star_cast	Movies.film=Stars.films		own
	Star_cast_is	Stars	
Starred_in	Famous_actors.name=Stars.actor		own
	Starred_in	Has_star	
Prod_By	Movies.film=Products.films		own
	Produced_By	Produced	
Works_in	Casting.location=Countries.country		ref
	Works_in	Has_available	

Figure 3-35 Role names.

The final entry shows that there is a close relationship between an attribute name within a relation and the role name we assign to a connection.

We notice that more than one relationship can exist between entities. In our Fig. 3-35 we distinguish the `Cast` and the `Stars` of `Famous_actors`—— `Movies`.

To support that structure we need distinct association relations and corresponding connections to identify the members of the two distinct relationships.

3-6 DATABASE VIEW MODELS

Often there is more than one way to model a database. Valid differences arise if groups of potential users bring their own point-of-view to the design process. Such distinctions were already encountered above. We need to accomodate such differences.

Do deal with individual views of the database we introduce one new concept now: *View Models*. This concept becomes quite important later when presenting database design methods in Chap. 5. A view model is a submodel of a complex database; a model which is limited in context so that it can be conveniently managed by individuals or a small group working within one situation.

VIEW MODELS In this chapter we have up to now collected knowledge about `Movies` without being concerned where the knowledge came from, and in which context it would be used. When we review the structure built so far we find that there is information from two, quite distinct, points-of-view:

V1 The `Movie_buff` view provides a model of data about old films

V2 The `Central_casting` view provides access to data for making new films

The two views share some information, for instance, some `actors` who played in old movies are still available as performers today.

Different views will have differing objectives, and hence collect, even when there is overlap, different attributes on differing subsets of the same entities. In the `Movies` case we may even question if it makes sense to share the database. Since it provides a nice example we will continue to do so.

Having distinct views on one database is common even when views are less radically different, Fig. 3-36 depicts an example. Among the `Employees` in Fig. 3-36 is so much overlap that serious inconsistencies could occur if the actual database would not be shared. For instance, when an `Employee` moves, the new `address` should only appear once. Still, the `Payroll_department` must control its own database, since it is resposible that everyone who works for the company gets paid. The `Personnel_department` carries out hiring and promotion actions independently.

In company `Fox` there are two views on `Employees`,
one by the `Payroll_department` and one by the `Personell_department`.

Figure 3-36 Views.

When view models are combined we speak of an *integrated database*. In the process of integration we will identify conflicts due to differing points-of-view. These have to be resolved, as shown in Chap. 5, either by technical means or by compromise.

Individual view models should avoid compromise and focus on understandability and completeness. For instance, creating 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.

> A single view model presents a consistent, conflict free, model of the database.

3-7 RELATION TYPES IN DATABASE MODELS

Our semantics have focused on connections. We can, however, also look at the semantics that relations derive from their membership in structural connections. We will do that now, because for some people, or some situations dealing with entities and relations provides better insights.

Within a database or view model we will find seven types of relations:

1 Primary entity relations
2 Foreign entity relations
3 Nest relations
4 Associations
5 Lexicons
6 Subrelations
7 Derived relations

Some relations can simultaneously satisfy the rules for more than one type in a view. In an integrated database model the connection types are inherited from the original view models. Many relations will have to satisfy multiple connection constraints so that their type structure becomes more complex.

We now define these relation types for a view in more detail. We also describe conditions and structural constraints normalizations which assure that all types of relations support a consistent update and deletion behavior. The relation types being introduced are summarized in Sec. 3-7-¡9¿.

3-7-1 Primary Entity Relations

A relation which defines a set of independent objects or entities is an *entity relation*. The choice of primary entities is a fundamental aspect of the design of a view model. Entities typically define objects that can be touched, counted, moved, purchased, or sold and that do not lose their identity during such manipulations. Primary entities are those which are the focal concern within a view model. The connections in the view should not constrain their manipulation. Comparison of primary with other types of relations will clarify the distinction.

Operationally, an update to a tuple of another relation will not require modifying an entity relation. An update to an entity relation may require modifications in subsidiary relations.

In `Casting` the `actors` comprise the primary entity. The tuples for the nested `Roles` of an `actor` will be removed when the `actor` quits performing and is removed from the `Casting` relation. A new `actor` may require new entries to be made in the `Roles` relation. We observe that from the point of view of the `Central_casting` model the `Roles` are not entities. The relation `Countries` is not a primary entity relation either.

A view may have multiple primary entity relations. In ¡Fig. 2-6¿ we showed part of a primary entity relation. The `Famous_actors` relation complements the `Movies_info` relation in the `Movie_buff` view.¿ ¡The attributes may be basic values or references to tuples in other relations, as shown for ¡`origin`¿ of Fig. ¡2-6¿.¿

3-7-2 Nest Relations

The definition of a relation excludes complex or repeating nests of attributes. When attributes may repeat they are placed into a separate *nest relation*, linked to the primary owning relation with an ownership connection. Tuples of nest relations have a ruling part which is the composition of the ruling part of the owning relation and an attribute value which is unique within the nest. This structure assures recovery of the semantic linkage implied by the multivalued dependency from the owner tuples to the nest tuples.

We show in Fig. 3-37 a nest relation for the tuples of Fig. ¡3 or 2 -6?¿.

```
                 +--------------------------+
                 | Famous_actors RELATION   |
                 +--------------------------+
                              |
                      Got CONNECTION
                              |
                              *
                 +------------------------+
                 | Some_awards RELATION   |
                 +------------------------+

Last_awards RELATION:       -- Last award received after actors were
                            -- in b-w postwar Hitchcock feature movies
 stage_name,     award :⟩ last,  category,   for ;
 --------------  -----   ----   ---------  ---------------------------
 Martin Balsam   AA      1965   acting     A Thousand Clowns
 Ingrid Bergman  AA      1974   acting     Murder on the Orient Express
 Ingrid Bergman  BFA     1974   acting     Murder on the Orient Express
 Henry Fonda     AA      1982   honorary   n.a.
 Cary Grant      AA      1969   special    n.a.
 Gregory Peck    AA      1963   acting     To Kill a Mockingbird
```

Figure 3-37 Nest relation.

The tuples of these nest relations are dependent on the owner tuples for their continued existence. This constraint was formalized as the *ownership rules* in Sec. ¡3-3-7¿.

The dependent part of a nest relation is ruled properly by the catenation of the owners's ruling part and the tuple-specific ruling attributes, for instance, in Fig. 3-49: `actor, award :⟩`. This model only permits us to show one of each type of award, and hence only the `last` one gotten is kept.

We can nest relations to multiple levels if we need to include more specific information, as shown in Fig. 3-38. The ruling part will increase by at least one attribute.

`All_awards RELATION:      -- All awards received by Hitch_movies`

stage_name,	award :⟩	last,	category,	for ;
Martin Balsam	AA	1965	acting	A Thousand Clowns
Ethel Barrymore	AA	1944	acting	None But the Lonely Heart
Anne Baxter	AA	1946	acting	The Razor's Edge
Ingrid Bergman	AA	1944	acting	Gaslight
Ingrid Bergman	AA	1956	acting	Anastasia
Ingrid Bergman	AA	1974	acting	Murder on the Orient Express
Ingrid Bergman	BFA	1974	acting	Murder on the Orient Express
Charles Coburn	AA	1943	acting	The More the Merrier
Henry Fonda	AA	1981	acting	On Golden Pond
Henry Fonda	AA	1982	honorary	n.a.
Cary Grant	AA	1969	special	n.a.
Charles Laughton	AA	1933	acting	The Private Life of Henry VIII
Gregory Peck	AA	1963	acting	To Kill a Mockingbird
Jane Wyman	AA	1948	acting	Johnny Belinda

Figure 3-38 Second-level nest relation.

We now have a multi-level hierarchy, also seen in Fig. 3-16. Sometimes many more levels of nesting can be encountered.

‡‡‡‡‡‡ **Skipping Nests** In the case shown in Fig 3-38 the higher-level relation `Last_Awards RELATION` has become redundant All the dependent attributes in that relation appear in the `All_awards RELATION` as well. The intermediate nest relation can be omitted here; the `last` award remains computable. This case is actually rare in practical databases. ‡‡‡‡‡‡

It is common that models require multiple nest relations at the same level. The `Leading_actor` relation in Fig. 3-¡3¿ has two potential nest relations, `Roles_Played` and `Schedule`. Tuples from multiple nest relations should not be stored together (Eq. 3-7), as was shown when `Movies` with multiple `Directors` and `Actors` were enumerated in Fig. 3-172.

3-7-3 Foreign Entity Relations

A *foreign entity relation* describes an entity which is not a primary entity in a model, but which serves to expand an attribute of another relation. For example we list for each `Studio` and `Academy` the `Countries`. With this information we associate secondary dependent attributes, for instance, its `population`, the `language` spoken, etc. Such information is shown in a relation connected via a reference connection to the primary relation.

A foreign entity relation also obeys all rules established for relations. Specifically, since redundant tuples do not appear the number of referenced tuples in the foreign relation will be equal or less, often much less, than the number of tuples in the primary relation. The primary attribute will appear in both relations, and forms the ruling part of the foreign relation.

The tuples of a foreign relation are not dependent on the tuples of the primary entity. In Fig. 3-20 `Countries` is a foreign entity relation. Even when no actors reference a specific `country` entity object, say `"Austria"`, the information remains valid and can be kept in a tuple. Little delay will then occur if a new `Casting` entry requires this foreign entry. Other foreign entity relations referenced are an `Agent` relation for by `Actors` and a `Studio` relation for `Movies`.

The reference connection prevents tuples which are being referenced by a tuple in a primary relation from being deleted; otherwise processing errors may occur. For instance it might be difficult to include an `Actor` in a `New_Movie` if the `Agent` cannot be located.

3-7-4 Associative Relations

Relationships can contain data which depend on the interaction or association of entity relations. In the `Movie_buffs VIEW` we had `Casts`, showing which `Famous_actors` starred in which `Movies` and what `role` they played in them. Figure 3-24 presented that association. An associative relation has a ruling part composed of the ruling parts of both entity relations. The attributes relate each tuple in the associative relation to tuples both entity relations. It is also possible to associate three or more entity relations, although it is rare not need intermediate relations. These intermediate relations will take on the role of entities for further associations.

The connections from the entity relations to the association are ownership or reference connections, and all the stated rules apply. Ownership connections are common, and we will focus on these.

The attribute or attributes $A_{j+1}, \ldots$ which distinguish individuals r in the owned associating nests (see Eq. 3-8) are taken from the other owner relations. Any other ruling part attributes will be redundant in BCNF.

In general for an association

$$A_{association} = \bigcup_{owners} A_{owners} \qquad 3\text{-}12$$

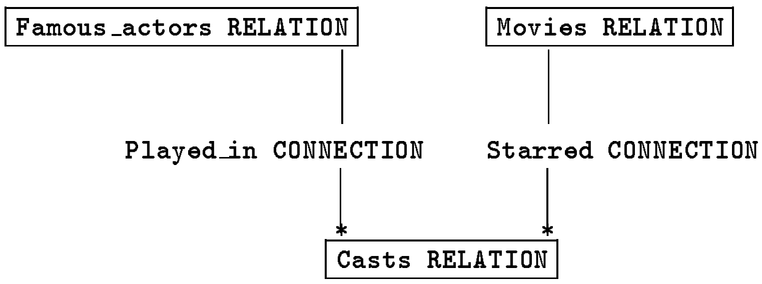
where the sets A are the ruling part attributes as originally defined in Eq. 3-4 and used throughout.

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. 3-39; the

Casts example of Fig. 3-40 is equivalent. Associations rapidly become large, since they form a crossproducts of their entites.

To be expanded to 90 pt for Figure space
was Fig.7-15

Figure 3-39 Associative relation.



Casts_in_postwar_bw_Hitchcock_films RELATION:

actor	film :⟩	role_description ;
Martin Balsam	H73	detective, victim 2
Ethel Barrymore	H44	judge's wife
Anne Baxter	H49	priest's ex girlfriend, blackmailed
Ingrid Bergman	H42	physician, saviour
Ingrid Bergman	H43	tart, saviour
Leo G. Carrol	H42	physician, murderer
Leo G. Carrol	H44	lecherous judge
Leo G. Carrol	H48	senator
Montgomery Clift	H49	priest, suspect
Michael Chekhov	H42	physician, good analyst
Charles Coburn	H44	lawyer
Marlene Dietrich	H47	suspect
Rhonda Fleming	H42	psychopath
Henry Fonda	H61	musician, suspect
John Gavin	H73	wimp, lover of victim 1
Farley Granger	H48	husband
Cary Grant	H43	US agent
Dolly Haas	H49	murderer's wife, saviour
O.E. Hasse	H49	murderer
Louis Jourdan	H44	servant, suspect, victim
Charles Laughton	H44	lecherous judge
Janet Leigh	H73	thief, victim 1
Karl Malden	H49	police inspector
Vera Miles	H61	wife, psychoneurotic
Vera Miles	H73	investigating friend, nearly victim 3
Simon Oakland	H73	dumb psychiatrist
Gregory Peck	H42	fake physician, psychopath, suspect
Gregory Peck	H44	lawyer

Anthony Perkins	H73	psychopath, suspect
Anthony Quayle	H61	young lawyer
Claude Rains	H43	Nazi, wimp
Ruth Roman	H48	victim
Ann Todd	H44	lawyer's wife, murderer
Richard Todd	H47	murderer
Alida Valli	H44	client, suspect
Robert Walker	H48	psychopath, wimp
Michael Wilding	H47	police
Jane Wyman	H47	saviour

Figure 3-40 Associative relation.

Frequently associative relations have no dependent part. In our model this would be appropriate for a the `Products` relation which associates `Producers` with the `Movies` they produced. Unless the movie-buffs are interested in financing details for the `Movies` we will have no other information to store other than the association itself.

We still need the associative relation and two connections to model the information contained in this relationship. We show part of the `Products` relation in Fig. 3-41.

`Post1940_Hitchcock_Products RELATION:`

film	producer :⟩ ;	*more:*	film	producer :⟩ ;
H33	Selznick		H42	Selznick
H34	Wanger		H43	Hitchcock
H35	Eddington		H44	Selznick
H36	Raphaelson		H45	Bernstein
H36	Hitchcock		H45	Hitchcock
H37	Lloyd		H46	Bernstein
H37	Skirball		H46	Hitchcock
H38	Skirball		H47	Hitchcock
H39	MacGowan		H47	Ahern

(All later movies directed by Hitchcock were also were produced by him.)

Figure 3-41 Association without dependent attributes

We can have more than one association between the same entities. It is for instance common to have one association for actual events, and for potential events. We show examples, derived from distinct views, in Fig. 3-42. Relations listing potential associations rarely have dependent attributes.

Actual events, movies that are released:

`Actors ——* Casts *—— Movies`

Potential events, planning new movies:

`Actors ——* Casting *—— Movies`

In our case we distinguished `Movies` and `New_movies`, so that here a further difference exists.

If we would associate multiple `Directors` with their `Movies` we would have four associations between `Movie_people` and `Movies`.

From another domain we can have:

```
Physician ——< Visits *—— Patients
Physician ——< Assigned *—— Patients
```

Figure 3-42 Multiple association among entities.

The semantic difference between such associations is not obvious from the connection structure and has to be described by careful choice of connection names as well as by attaching a definition to the relation. An association can also associate tuples within the same relation.

3-7-5 Lexicons

In any relation the ruling and dependent parts should be clearly identified. We often encounter relations which have more than one *candidate ruling part*, an instance is shown in Fig. 3-43.

We have refered to `Countries` by name, say `England`, in `Studios` and by adjective, `British`, in `Famous_actors`, but we should have only one set of data.

```
Countries RELATION:
    country-name ∨ country-adjective :⟩
        size, population, continent, language, film_production;
```

Either `country-`*name* or `country-`*adjective* could be the ruling part, but the dependent attributes would remain *FD* on both.

Figure 3-43 Multiple candidate ruling parts.

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

$$\begin{aligned} L &: FD(A) = B;\ FD(B) = A << checknotation >> \\ R &: FD(B_i) \neq (A_1, A_2, \ldots, A_j) \end{aligned} \qquad 3\text{-}13$$

where B_i and A_i are defined as in Eq. 3-4. Now Eq. 3-4 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. In the view model this structural dependency is represented by having *reference connections* between the relations.

A lexicon relation, giving a listing of `country_names` versus `origin_adjectives` is shown in Fig. 3-44. 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.

```
Of_origin LEXICON :
   country_name ⟨:⟩ origin_adjective ;
```

country_name ⟨:⟩	origin_adjective ;
Austria	Austrian
Canada	Canadian
England	British
France	French
F.R.Germany	West-German
German D.R.	East-German
Germany	German
Japan	Japanese
Holland	Dutch
Italy	Italian
Sweden	Swedish
USA	American
USSR	Russian

Figure 3-44 A Lexicon.

A lexicon is a special type of foreign 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 foreign relations.

The `People` relation uses the adjectives to denote the `origin` of `Actors` etc. To get data about a `country` from the `origin` value, the `Of_origin` lexicon is used to obtain the `country_name`, by matching `origin` with `country_adjective`, and then detail can be found in the `Countries` relation.

Figure 3-45 Use of a lexicon.

Lexicons are frequent in operational databases, and recognition of lexicons during the view model 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.

The `Of_origin` lexicon makes it possible to link `Actors.works_in` to the `Famous_actors.origin`.

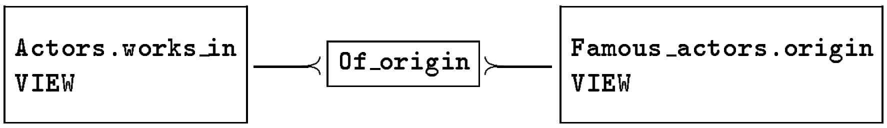


Figure 3-46 Use of a lexicon among views.

3-7-6 Subrelations

Subrelations are a result of generalizations. Defining more general classes is an important concept in abstraction, because it permits to apply analyses and results to a larger set of objects. Since the general objects will differ in some sense we often have to maintain subrelations to provide for their distinctive aspects. Subrelations are connected to the general relation using the subset connection. We will now consider how membership in subrelations is determined.

Subset relations exist in several types. The types range from providing a formal and complete classification to a type which takes care of all ad hoc subset memberships. We consider here 2×2 types of subrelations:

Table 3-4 Types of subset relations.

		Sub object existence: Disjoint	Sub object existence: Overlapping
Sub object selection	by attribute value	essential (one)	preferred (many)
	by insertion/deletion	unsafe	feasible

We will now discuss these alternatives and their interactions.

♯♯♯ **DISJOINT VERSUS OVERLAPPING SUBSETS** If we have *disjoint subsets* an object from the source relation is only represented once in the set of subsets. In our model we let an `Actors` be a member of only one of the subrelations shown in Fig. 345.

```
Leading_actor RELATION:     actor :⟩ fee, percentage, type_cast;
Character_actor RELATION:   actor :⟩ fee, type_cast, recognition;
Supporting_actor RELATION:  actor :⟩ fee, role;
Extra RELATION:             actor :⟩ daily_fee, availability;
```

Figure 3-47 Disjoint Subrelations.

We also expect each general object to appear once in a subset relation. Now the set of disjoint sub-relations matches the source relation: there will be one tuple in some subset for every tuple in the general relation.

If *overlap of subsets* is permitted then subtuples of one general tuple can be members of more than one subset. That situation is in fact appropriate for the `People` in our model and their subsets `Famous_actors`, `Producers`, and `Directors`. We have cases of overlap of all of them. ♯♯♯

♯♯♯ **MEMBERSHIP DETERMINATION** The other classification of subsets is according to how the actual membership is determined. The preferred way, *subset selection by attribute value* permits automatic decision-making, by identifying a controlling attribute in the model. The alternative is to let subset membership just be controlled by actual transactions which insert or delete subset entries.

Model Controlled Subset Membership Control is achieved by letting specified data values decide on subrelation membership. Typically one attribute in the general relation provides a value selecting the subset. There will be a subrelation for every permitted attribute value. In the `Casting` relation of Fig. 3-20 the attribute `status` selects the subrelation. The subrelations specified various pay methods, one for each status.

Using directly a single variable assures us that the subset relations will be disjoint. An `Actor` can only be in one of `status`-subrelations. In Fig. 3-48 we specify the membership constraint in the model. The domain of permissible `status` values is constrained already by the `Actor_states` relation shown in Fig. 3-21.

Given
```
    Casting RELATION: actor :⟩ status, location, sex, ...  ;
```
and
```
    Leading_actor RELATION: actor :⟩ fee, percentage, ...  ;
```
we can state
```
    Leads CONNECTION: Casting.actor ——∋ Leading_actor.actor
                                | Casting.status = "Leadg";
```
Similar connections will be given for all values of `status` for the `actors` in `Casting`.

Figure 3-48 Modeled subrelation determination.

If the membership can overlap then one boolean value is needed to determine membership for each candidate subset. Sometimes computations on more complex attributes can determine the subset membership condition, as shown later with Fig. 3-49.

Transaction Controlled Subset Membership The alternative is to let subset membership be determined by insertion and deletion operations into the subsets. That alternative is too risky for disjoint sets, but feasible for overlapping subsets. In our case we can insert, say, `Producers` at will, as long as they exist or inserted just prior into the general `People` relation.

Computing Membership The condition defining membership in a subrelation can, of course, be arbitrarily complex. To identify "Very Important Persons" one might write an expression as shown in Fig. 3-49.

```
Is_VIP CONNECTION:  Customer.name ——∋ VIP.name,
    | income > 1 000 000 OR capital > 10 000 000
     OR position > GS-14 OR Casting.status = "Leadg";
```

Figure 3-49 Complex subrelation determination. ‡‡‡‡‡‡

To be expanded to 378 pt for Figure space
was Fig.7-18

Figure 3-50 An erector set of relation types.

3-7-7 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
foreign 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 3-46 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.

Table 3-5 reviews the sematics of the relation types described, using the rules defined with the connections.

Table 3-5 Connection semantics applied to relation types.

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 foreign entity relation:*
1 Referenced from within the view model.
2 The ruling part defines the entity and establishes domains for primary 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 3-6 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 objects"*
⟨reference key⟩	::=⟨reference attribute⟩ *"unique objects"*
⟨primary entity key⟩	::=⟨value key⟩ *"not referenced"*
⟨foreign entity key⟩	::=⟨reference attribute⟩ *"foreign"*
⟨nest key⟩	::=⟨attributes⟩ *"unique within each nest"*
⟨dependent part⟩	::= null | ⟨attributes⟩
RULING PARTS:	
⟨entity ruling part⟩	::=⟨primary entity key⟩
⟨foreign ruling par⟩	::=⟨foreign 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⟩
                          |⟨foreign ruling part⟩
                          |⟨nest ruling part⟩|⟨assoc. ruling part⟩
RELATIONS:
⟨entity⟩                ::=⟨entity ruling part⟩ :⟩ ⟨dependent part⟩
⟨foreign entity⟩        ::=⟨foreign 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.*

3-8 SUMMARY

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, and its relatives and predecessors, has been used in a variety of situations, but any application has to be combined with *common sense* and *insight*.

Table 3-7 Terminology.

Semantics	Model	Software	Hardware
Entity set	Relation	File	Disk
Description	Relation-schema	Schema (see Chap. 8)	Documentation
Object	Tuple	Record	Block

Relationship	Connection		
Link	Link	Ref.Pointer	Pointer
Object name	Ruling part	Key	Address
Properties	Dependent part	Goal	Content
Property	Attribute	Attribute name	Byte number
Value range	Domain	Data type	Data format
Data element	Value	Attribute value	Byte or word

We now summarize all the prior examples in a model which brings together all important relations and connections we have encountered. We do not list the attributes here. The chapter on schemas (Chap. 6) will expand on that aspect of database modeling.

To be expanded to 400 pt for Figure space
new, to be drawn

Figure 3-51 A comprehensive model.

BACKGROUND AND REFERENCES

In Chap. 1 we provided some general background about the development of databases. Databases, as a scientific area, could not flourish until models were available that provided an adequate abstraction.

We can find relevant citations to modeling of knowledge in the earliest philosophical literature, for instance in Frege[1892] Writings of Gottlob Frege; Blackwell, Oxford, 1960.] Quantitative analysis came later; in 1948 Shannon[62] published a paper defining the concept of information in a formal sense.

An attempt at a formalization of the underlying concepts of files, attributes and mappings was made by a CODASYL sponsored Development Committee Bosak[62]. The complexity of the model discouraged its use.

In parallel a scientific approach to database problems began to be concerned with the concepts of data and data structure (Steel[64], Mealy[67], Bachman[72], Senko[75,77], Kent[78], Lucas[81], and Tsichritzis[82]), and this, influenced by the developing ideas of structure in programming (Dÿkstra[71], Dahl[72], Parnas[72]), has led to an approximate agreement of structural layers; a taxonomy is given in Booth[73].

Models also enable a formalization and integration of diverging user requirements (view models) and subsequently the match of requirements to DBMS capabilities. Tools associated with these models make user involvement in the design process possible. The effectiveness of a model is limited, of course, to the area where its abstractions are relevant. A major effort to develop a complete set of modeling procedures for database design has been supported by the Italian research establishment Ceri[83].

When devising a model there has to be a tradeoff between simplicity and power. The number of primitive concepts in a modeling method must be small, so that the user of the model can easily comprehend the database model. At the same time, it is necessary that a model can describe all of the essential features of the problem being addressed. We believe that the seven primitives of the structural model (relations, attributes, domains, and four types of connections) are both manageable and adequate Miller[56] for our objective. The search for a compact unifying model which can do everything for everybody is undoubtedly as elusive as the search for the ideal programming language. Large, baroque models are as unattractive as large, baroque programming languages.

We will now discuss the background of some common models.

Relational Model The milestone formalization for databases was provided by Ted Codd[70], then of IBM research. The relational model he presented and extended in papers at a 1971 Courant Computer Science Symposium on databases (Rustin[72]) enabled understanding of the problems which were hampering database systems due to their steadily increasing complexity. The model formalizes a simple tabular view of the database and a small, well-defined algebra.

Data describing entities are normalized into the tabular structure so that the operations can deal with the data as relations. Relations are an extension of the mathematical concept of sets, and using them linked database concepts to a body of existing theory, which in turn increased interest in the theory. The relational model did not attempt to include the relationships and semantics between the relations or consider the performance of the databases.

The relational model has seen the widest extent of theoretical effort. Books by Ullman[xx], Maier[xx] summarize the state of the art. Concepts as ruling parts (keys) and reference constraints (foreign keys) were included later Codd[tas], but have seen less formal exposition.

Implementation-based Models The relational model was in part motivated by the complexity seen in existing databases of the late 1960s. Existing approaches to database implementation included implementations of relationships. To clarify their approaches the developers of systems which used links among file records started explaining their work in model terms as well. Network models are now used to describe the CODASYL development and related sytems, while hierarchical models are used to describe IMS and approaches like SYSTEM-2000 Sibley[76].

The intensity of discussions among modelers increased, creating attention and confusion. Whenever some concern of a researcher or builder is not addressed by existing models new models are defined and promoted. Why can database researchers not agree on a single model to describe databases and the data contained in them? We will consider this issue.

Models based on a DBMS implementations are always suspect. For such models there is good reason to ignore valid concepts that are not available. There is also a desire in such cases to explain within one model everything which distinguishes the implementation. This includes features that should not have been there and those that are irrelevant at a chosen level of abstraction.

The family of ER models Recent models for data concentrate on representing the relationships among the conceptual entities, and on the mapping of these conceptual structures to files and linkages. Relations and connections may be used as an important formal, but intermediate concept.

- The entity-relationship model supports the specifications of user views and logical requirements Chen[76].
- In functional models the relationships are modeled by functions. This approach helps formalize the procedural constraints that are now hidden in the programs which operate on the database Buneman[80], Shipman[81].

- The structural model, we focus on here, defines relationships as formal connections between relations. This model supports transformations to permit integration of diverse user views and the specification of quantitative factors for database design decisions Wiederhold[83].
- In the universal relation model the entire database is viewed as a single relation and all relationships are described through a family of dependencies.

The Entity-Relationship Model (ER) The original ER model Chen[80] led to a series of conferences where further developments were published Chen[xx] ...... It related well to needs of the community and was wll accepted. The structural model was presented in Wiederhold[80] and used for integration of data models by ElMasri[80], who also compared its coverage to other models ElMasri[80b]. Several commercial versions of ER modeling are now being supported [Appletin[xx], ...

A number of further research results are also available. Bradley[88] analyzes problems that occur when connections are not bound to at least one ruling part.

The Functional model

Universal Relation Model In the relational model we use dependency knowlege to allocate attributes to appropriate relations. The universal model specifies the intension as the set of dependencies and the extension conceptually as a single relation. In the universal model the dependencies remain unbound and the user no longer has to visualize the assignment of attributes to relations Ullamn[xx]. The universal relation can be seen as one large matrix, with all candidate attributes across and all entities and their components represented by rows. The matrix will be sparse.

The conceptual simplicity of this model provides great power for the formal analysis of the semantics among the data and this area has become an academic subdiscipline in its own right Fagin[82] Implementation of such a model use a mapping to multiple conventional relational files.

A *universal relation scheme* lets the user see all the data as a single relation. All the component attributes will have unique names. All connections are deduced from specification of the dependencies.

The complexities of a normalized model are now hidden from the user, and the underlying database relations can be freely rearranged to support optimal processing strategies. Any processing that relates attributes by paths other than defined dependencies will still require explicit definitions, as the pure relational model does. If multiple paths between the attributes are possible, either user interaction, heuristics, or a set of predefined rules can be used to determine the best path.

Moving Among Models Transformations between all these model types are possible. Where one model does not express some concepts of another model some semantics are lost. In the converse case the converted model is incomplete. In all cases some concepts translate awkwardly, so that clarity of expression is lost for those concepts. For a casual user the choice of models is not always obvious. Model builders rarely state what their models will not represent.

The development of models permits dealing with realistic applications in an abstract fashion. The notion of modeling is also likely to become more important in other areas of computer science that deal with complex systems. Databases tasks are often complex; detailed specifications for some applications require sets of documents many inches thick. At the same time the complexity of many actual DBMS's is such that a description of their features does not provide an understanding of their capabilities.

EXERCISES

1 List some additional examples of the class defintions described in Sec. 3-1-1.

2 Sketch an Entity-Relationship Diagram for all entities that appear in the film registry of Fig. 3-18. Note that some entities can encompass more than one structural model relation.

3^p Define the Entities and Relationships which will be part of your application. Show them in graphical (ER) form. To make the project interesting, but nor excessively complex, we expect that you come up with an application with 3 to 6 entities and 6 to 10 relationships.

4^p For each of the Entities and Relationships that you defined, list the attributes with their domains. If attributes appear repeatedly just list them as "Like ...". It is wise do create the list using a text processor, so you can use the work for later exercises.

5 Show the cardinality or relative cardinality for each of the entities and relationships of the film registry of Fig. 3-18. You must be imaginative in your assumptions.

6^p List the expected cardinality of objects for each entity shown in the Entity-Relationship Diagram for your application. Estimate the expected relative cardinality for each relationship shown.

7 Show the ruling parts for Figure 3-8.

71 We wish to access the `Movies` database easily by `titles`, including synonyms. Show the relations and sketch the access procedure. State any assumptions you made.

8 ¡¡domains¿¿

9 Why is the `type_cast` attribute in the ruling part in the `Roles` relation of Fig. 3-23?

10 Why did we not define a
`Came_from CONNECTION Actor.origin` $\succ$—— `Countries.language:` ?

11^p Start a Data Dictionary by listing all the attributes for which your database is likely to contain information. For each attribute identify to which entity or relationship (as defined in Ex. 3-3) they belong.

12^p Expand the Data Dictionary created in Fig. 3-11 with user information. Use descriptive terms for the users or kinds of users you envisage. Show for each attribute the likely owner of the data, the sources for new or changed information, and the candidate users for the information.

13 Specify the subset connections appropriate for Fig. 3-6.

14 For a model consisting of `Employees`, having `Skills`, assigned to `Departments`, define the connections and assign role names to both directions of each connection.

15 State that some `Employees` are `Managers`, using a connection and assign role names to both directions.

16 Associate the entities `Pilots` and `Airplanes` for an airline having a bunch of pilots and airplanes of several times. Pilots may be qualified to fly more than one model of airplane. Define the connections and assign role names to both directions.

17[p] For each attribute in the Data Dictionary show the domains, null-constraint, and roles (key, dependency, reference). Make sure that you distinguish between domain definition and domain representation.

171 Consider the relations shown in Fig. 3-18 and determine if they are primary or referenced entity relations, nests, associations, etc., according to the categorizations of Sec. 3-7.

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

19 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.

20 We want to add more `Movies`, `Actors`, and `Awards` to our file and find for

stage_name	award	year	category	for_title
Shirley MacLaine	BFA	1959	acting	Ask Any Girl
Shirley MacLaine	BFA	1959	acting	The Apartment
Shirley MacLaine	AA	1983	acting	Terms of Endearment

Find a problem in our model with entering this data, and suggest a correction.

SOLUTIONS TO THE EXERCISES

3-1. More Classes

3-2. E-R Diagram for complete film registry

3-5. Cardinalities We have 85 Hitchcock movies (only subsets are shown here), with several hundred famous actors. About 20 studies were used, and Hitchcock worked with about 30 producers and 10 other directors. The total number of films made in the world must exceed 10 000, and the other numbers will increase a bit less than proportionally if the entire world production is taken.

The relative cardinality of the `Made_in` relationship is about 5:1. All other relationships have multiples on both sides, `Casts` is about 2:10 and `Products`, relating `Movies` to `Producer` is about 3:1.1.

3-6. Project Cardinalities Solution 3-5 provides guidance on the type of information to expect.

3-71. Ruling Parts

```
Movies    {title :⟩ year, director, producer, actors, studio, prc, ...}
Actors    {stage_name :⟩ born, sex, agent, typecast, SSN, address, ...}
Directors {ref_name :⟩ born, sex, studio, SSN, address, ...}
Studios   {name :⟩ city, country, film_production ...}
Works_for {Directors.studio, Studios.name :⟩ since, until}
Markets   {Studios.name, Movies.studio :⟩ since, until}
Cast      {Movies.actors, Actor.stage_name :⟩ role, fee}
```

3-71. Synonyms

```
Movies RELATION:
    film :⟩ movie_title, director, studio, prc;
All_titles RELATION:
    Movie_titles, year :⟩ film_id;
```

If the given `title` is not unique in `Movies_title` then ask for the `year`. Then fetch from `Movies` the `film` corresponding to the `film_id`.

Assumption: In no single `year` were two movies made with the same `title`. (Other assumptions can be made, and will lead to different ruling parts for `All_titles`. We have already shown that the assumption that the combination of `title` and `director` is not unique.)

3-8. Domains ¡¡Copy from MOVIES.TEX¿¿

3-9. Ruling Part Since it is possible to play different `roles` in the same year, the `last_performed` year, which is otherwise a reasonable choice, does not guarantee uniqueness.

3-10. Why no `Came_from Connection`? The destination of a connection must be a ruling part, so that a direct connection is not permitted. No unique linkage could be assured.

Join operations (see Chap. 5) can compute the linkages here. Note that our model assumes only one language per country, requiring that regions as "`Wales`" be made a seperate countries. A fully realistic model would require an associative relation `Spoken RELATION : language, country :⟩ ;`.

3-13. SUBSET CONNECTIONS

3-14. EMPLOYEES AND DEPARTMENTS

Connection name	source role name →	destination ← role name	type
Emp_dep	Employee works for	Department employs	reference
Emp_skills	Employee has	Skills are found in	ownership

3-15. SOME EMPLOYEES ARE MANAGERS Make sure that the Managers don't own the employees; those semantics would rarely be acceptable: if a Manager were fired the Employees would also be deleted.

Connection name	source role name →	destination ← role name	type
Emp_mgr	Employee may_be_a	Manager is_ a	subset

3-16. PILOTS AND AIRPLANES. Since the exercise describes a $m : n$ relationship we have to introduce a new relation: `Airplane_model` which associates `Pilots` and the instances of `Airplanes` and their connections. These connections are then:

Connection name	source role name →	destination ← role name	type
Flies	Pilot is_qualified_to_fly	Airplane_model can be flown by	ownership
model	Airplanes is_of_model	Airplane_model is_in_fleet	reference

3-17. RELATION TYPES. Note that Fig. 3-18 concerns a `Film_registry` database view.

```
Movies_info RELATION:     -- Primary entity
Directors RELATION:     -- Reference entity
Producers RELATION:     -- Primary entity
Products RELATION:     -- Association
Famous_actors RELATION:     -- Referenced Entity
Casting RELATION:     -- Subrelation of Famous_actors in this view
Casts RELATION:     -- Association
```

```
    Studios RELATION:     -- Referenced Entity
    Countries RELATION:     -- Referenced Entity
```

3-18. Relation Types in Literature The classification of the prior exercise can provide guidance.

3-20. Awards for Shirley MacLaine The ruling part of `Awards` only has `name, award, year`, but `Shirley MacLaine` got two different `BFA` awards in 1959. The best fix possible is (a) to replace the `year` with the `title` of the film, only then non-film awards look awkward. Another fix is (b) to add a sequence number into the ruling part, which is effective, but ugly. Adding (c) the filmcodes to the ruling part leads to redundancy, as will be shown in Chap. ¡4 with normalization¿. The formal, most complex solution is (d) to define two subrelations, one for `Film_awards` and one for `Non_film_awards`, and then to use solution (a) for the first subrelation. We still retain the assumption that there will be, from one `academy`. only one `Non_film_awards` per year, else we have to include the `category` in the ruling part of the ¡¡ ¿¿

We have here a case where there is no *best* and elegant solution. Systems which permit *nested relations* within other relations offer an escape out of this impasse, but have more complex computational rules.