SEMANTIC DATABASE DESIGN
Gio Wiederhold, CONNECTIONS February 7, 1991

Chapter 4, file S4DBD.TEX

This file is ©1977 and 1983 by McGraw-Hill and ©1987,1989 by Gio Wiederhold. All rights reserved. Do not duplicate.

Chapter 4: Connections;

4–1 Connections as a model of Relationships.
4–2 Ownership
4–3 Reference
4–4 Subset
4–5 Identity
4–6 Naming and Composite Models
4–7 Relation types
4–8 Summary

Chapter 4

'Michael, we are going to remove the entity now'.
'I want to be of assistance. There must still be a connection between me and the entity, even though I can't see it.'

Michael Crichton
Travels with my Karma, Knopf, 1988.

4-0 INTRODUCTION

In Chap. 3 we introduced models. Models use simplified abstractions to permit us to focus on the essence of our tasks. A general, high-level model presented was the Entity-Relationship model. We formalized its entities into relations. In this chapter we continue the formalization by dealing with the relationships. These will be modeled as *connections*.

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. Data semantics are independent of the processing programs which exploit them. The objective of this chapter is to develop concepts for recognizing and describing the structure, so that guidance can be provided for the design of databases.

A connection is a semantic device; connections are only *intensional*. An essential difference between relations and connections is that connections do not have

an extension. They depend for their values on the extensions of the relations they connect. Since connections specify relationships among entities in a general sense, we can use the structural model to guide relational as well as hierarchical network, or object-oriented implementations.

Layout of This Chapter This chapter focuses on connections, and on the structural model that is created when relations are linked with connections. In a closing section we show relation semantics that are imposed by their connections.

1 In Sec. 4-1 we introduce the structural model with its formal connections.
2 In Sec. 4-2 we specify ownership connections, which support nested structures.
3 Section 4-3 specifies references among relations.
4 Section 4-4 formalizes subset relationships.
5 In Sec. 4-5 we specify connections that involve computation and delay.
6 Section 4-6 discusses briefly the naming of connections and their roles.
7 Section 4-7 categorizes relations into seven semantic types, according to the connections linking them.
8 Section 4-8 summarizes the terminology that has been introduced .

The use of connections to deal with recursion is deferred to Chap. 6, where the corresponding operators are introduced as well.

4-1 THE STRUCTURAL MODEL

To design a database we model the components which will represent the information to be stored about entities and their relationships. These components are relations and connections. The model must allow them to be manipulated, since database design requires synthesis and analysis of many alternatives for representing the database. The effects of such manipulations must be predictable, so that the model must be formal.

Relations and connections are abstract concepts; they omit implementation detail. They must be able to represent those database semantics which are needed by the designer to develop specifications adequate to define the structure of the database. That structure defines our plan for the database, the *intension*. When the structure is well understood a suitable file design can be chosen. The actual extension is filled in during use, after the implementation.

The abstractions of our *structural model* focus on the structure of the database. This model bridges the gap between potential users, for whom ER concepts would be adequate, and designers and implementors, who need much more specific information.

4-1-1 Components of 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.

We stated earlier informally 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 covered in Sec. 4-6.

We introduced a number of new terms here. Table 4-1 summarizes them and places them into categories for each layer. 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.

Table 4-1 Entity-relationship (ER) and relational/structural model (SM) terms.

Abstract layers		Implementation layers	
ER model	SM model	Software	Hardware
Entity	Relation	File	Disk
Object	Tuple	Record	Block
Relationship	Connection	Pointer-field	Address-field
Link	Link	Pointer	Address
Property	Attribute	Data field	Word or Byte

The similarity of the terms *relations* and *relationships* is unfortunate, but they are so well established that trying to change them would create further confusion now. It is useful to recall that these terms have different origins, and that they are used at different levels of abstractions, as shown in Table 4-1. More terms will be added later and summarized in Table 4-6.

4-1-2 Connections as a Model of Relationships

We now formalize the second supertype from the entity-relationship model. We used relations to formalize entities; for relationships we use *connections*. We show an example of a connection in Fig. 4-2 and observe that connections imply constraints.

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

Relating Entities by Attributes Connections are instantiated via attribute values in the source and destination relations that match. One requirement is that both belong to the same domain. Domains useful for establishing connections in our model were shown in Fig. 3-20. To assure that the comparison work in practice, the domains must also be represented by the same type.

A necessary 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.

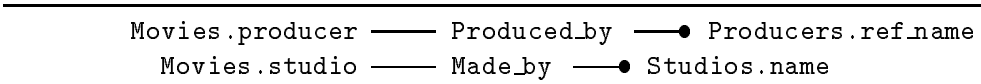


Figure 4-1 Related information.

For example, we can now define a connection between `Hitch_movies` and `US_studios`.

The concept of a connection is similar to the concept of a relation: a relation collects all attributes that are of general interest over time; a connection collects all attributes that establish generally valid linkages. A relation has an extension consisting of a set of tuples and a connection will have a set of connection instances or *links*. A link exists when two tuples, one each from the relations among which the connection is defined, satisfy the rules for the connection. For an instance of a link to exist it is critical that the attribute values along the connection match. Figure 4-2 shows the connection `Made_in` and links for selected `Hitch_movies` made prior to 1950. We restrict here the `Movies` relation to the `film` code and `studio`, and the `US_studios` to `name` and `city`.

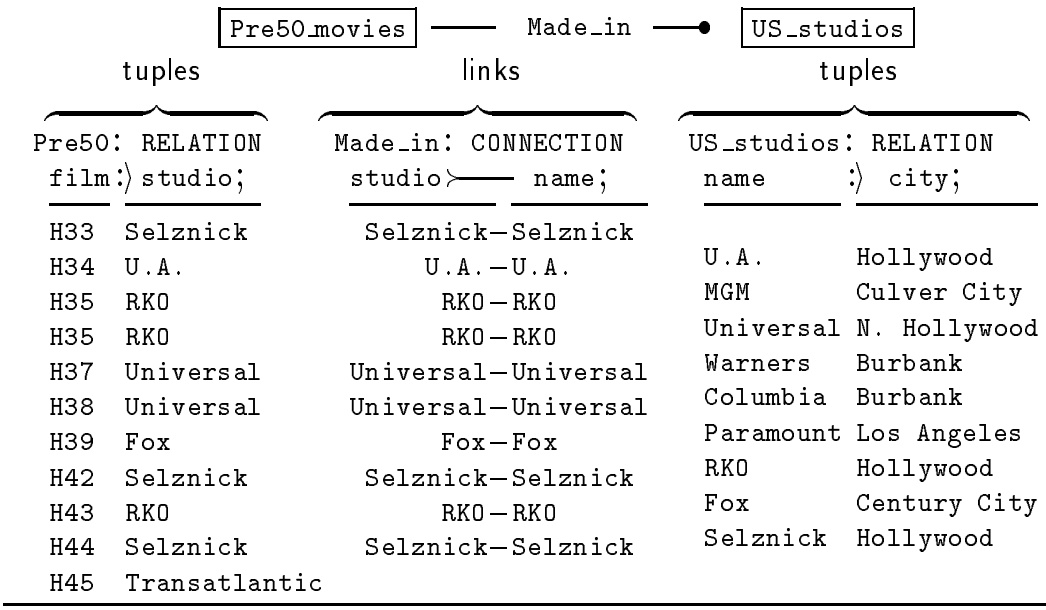


Figure 4-2 Some links for a connection.

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

Missing links can lead to incomplete information. If we wanted to know all cities in which `Pre50_Hitch_movies` were made, "`London`" would not appear. In most data-processing situations we must care about completeness, so that we will define constraints with our connections. Section 4-5 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 referenced object,

here a `Studios.name`. Then we must either omit "`H45`" from the source relation `Pre50` or enter "`Transatlantic`" into the destination relation, `US_studios`. If we choose to insert more studios, then the semantics of the relation are modified, so for clarity the destination relation should be renamed `Studios`. In a formal sense it does not matter what the name of the relation is, but a user will derive semantics from the chosen name.

Figure 4-2 also shows the elements needed to specify a connection,

$$\text{Connection\_name: CONNECTION} \qquad \text{Source.attributes} \xrightarrow{?} \bullet\ \text{Destination.attributes [Extra-constraints] ;} \tag{4-1}$$

The source and destination are relations of the model. The symbol marked "?" denotes the type of the connection. We can impose constraints beyond those implied by the connection type.

4-1-3 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 were 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 specified 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. 4-2 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. 4-1. It is composed of five elements:

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

Table 4-2 Connection types in the structural model.

Connection Type	Symbol	Direction
Ownership	——*	from single owner to multiple owned tuples
Reference	≻——	from multiple primary to single foreign tuples
Subset	——∋	from single general to single subset tuples
Identity	——≡	from single source to single derived tuples

The structurally relevant semantics of the database is encoded into these connections; we already encountered examples of the first three types.

1 References lead to entities which further describe a property. An example is `Producer` ≻—— `Works_for` —— `Studios`.

2 Ownership gives detail of multiple properties of an entity. An example is `Famous_actors` —— `Got` ——* `Some_awards`.

3 Subsets link general classes to their subclasses. In an earlier example we had `Actors` —— `Is` ——∋ `Leading_actors`.

4 Identity defines derivations of data. An example is counting stored objects, classifying them by some property, and placing the results into a new relation: `Movies.(producer WITH COUNT(*) )` —— `Becomes` ——≡ `Producer.(name,productivity)`.
Since computation is likely to induce delay, especially when relations are physically distributed, we also use the identity connection to specify asynchrony.

We provide detailed specifications for these four connection types in Secs. 4-2 to 4-5.

4-1-4 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.

Consider the `Made_in` connection between `Movies` and `Studios`. The constraint appropriate for this relationship is given by a *reference connection*; it provides a deletion constraint to assure that all `Studio` information that is referenced by any `film` tuple in the `Movies` relation will be retained. For instance, we cannot delete the information about the `Studio` named `Fox` while we have stored records in the database for the `Movie` "`H38 :) Lifeboat`" in Fig. 3-9. This constraint enforces the integrity of the entire object identified by `H38`.

Note that constraints are described in the model with the connections, but they apply to tuples.

4-2 OWNERSHIP CONNECTIONS

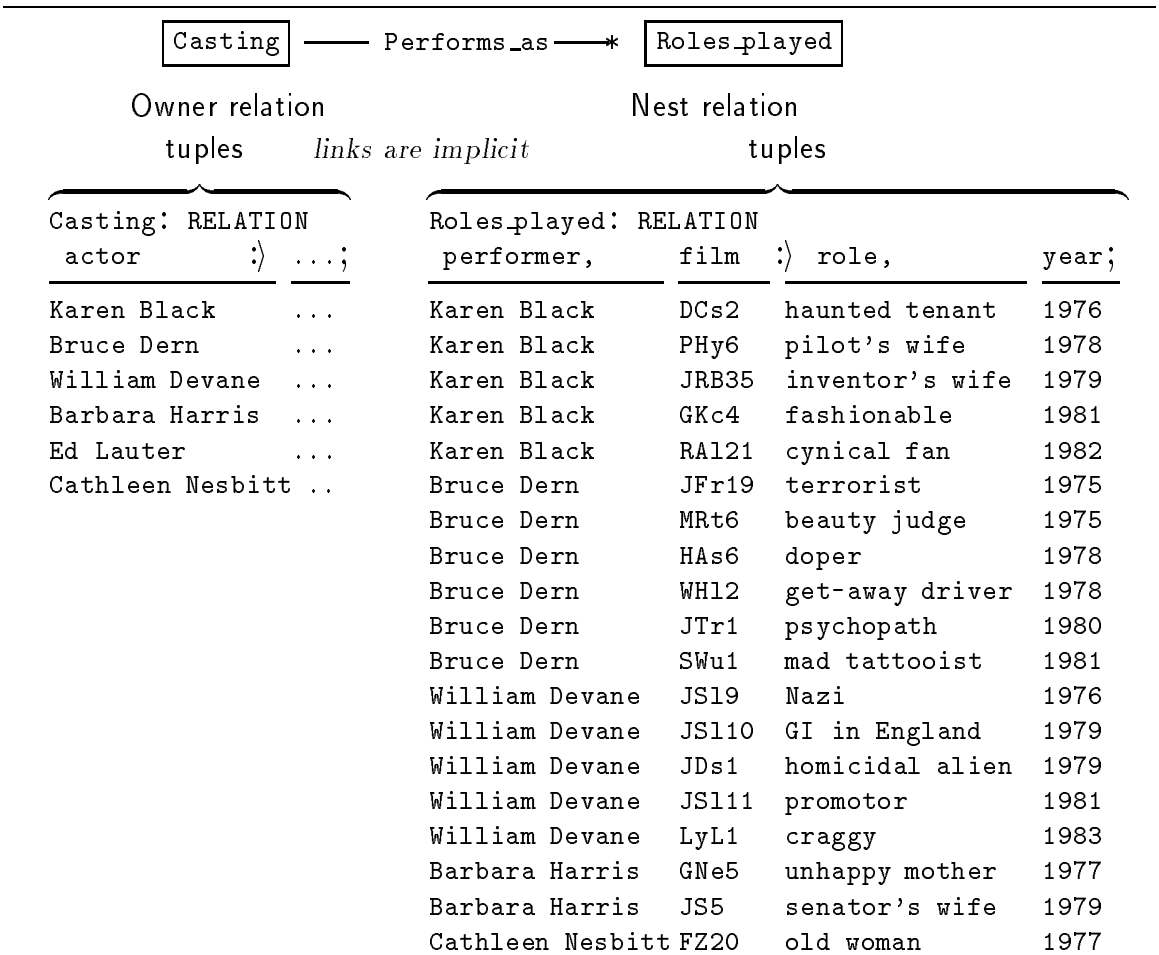
An *ownership connection* provides links for groups with multiple dependent attributes. The general form is

Name: CONNECTION Owner.ruling_part ⟶* Member.ruling_attributes ; 4-2

Figure 4-3 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.

4-2-1 Semantics

Ownership connections define structures to serve inclusion dependencies as discussed in Chap. 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. Similarly, we can only insert owned tuples if we have an owner tuple.



`Casting: RELATION`

actor :⟩	...;
Karen Black	...
Bruce Dern	...
William Devane	...
Barbara Harris	...
Ed Lauter	...
Cathleen Nesbitt	..

`Roles_played: RELATION`

performer,	film	:⟩ role,	year;
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 fan	1982
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
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	GNe5	unhappy mother	1977
Barbara Harris	JS5	senator's wife	1979
Cathleen Nesbitt	FZ20	old woman	1977

Figure 4-3 Ownership.

The owning relation, `Casting`, has one tuple for each owner object; the owned relation has as many as needed for all the `roles` listed. A group of owned tuples, as the six `roles` shown for `Karen Black`, is called a *nest*. One nest (`Ed Lauter`'s) is empty. The owned relation (`Roles_played`) can also be referred to as the *nest relation*. Information similar to that shown in the table of Fig. 3-14 is now distributed over two BCNF relations.

Associations 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 necessary to represent relationships which have an $n : m$ cardinality. Two connections will be needed as sketched in Fig. 4-4.

Two relations connect to the `Casts` relation:

```
Movies_info  : RELATION  ——*n
                         ——*
                         Casts: RELATION film, 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.film ;
Played_in: CONNECTION Famous_actors.stage_name—* Casts.actor ;
```

Figure 4-4 Association.

In our example `Casts` is a nest 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. 4-6.

4-2-2 Structure of Ownership

Ownership links owners to detailed information, so that the j connecting attributes from the two relation schemas must match. For the owning relations these attributes are the ruling part rp. The owned relation will have an additional ruling part attribute a_{j+1} to identify individual detail tuples.

$$\begin{aligned} Owner.A_i &= Owned.A_i \text{ for } i = 1, \ldots, j \\ k &> j, \quad j = \| Owner.rp \|, \quad k = \| Owned.rp \| \end{aligned} \qquad \text{4-3}$$

The size s of these owned detail sets ranges from $0 \rightarrow \#D(A_{j+1})$ where $\#D$ is the cardinality of a domain as shown in Chap. 3-6.

We can now state the rules for ownership connections:

Ownership rules (4-4)

Semantics: Owned tuples represent sets of object attributes.

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.

Note, 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, they define the way of doing business.

4-2-3 Cardinality of Ownership

There may be no, one, or many owned detail tuples in a nest for each owning tuple. The expected cardinality of ownership is then $1 \longrightarrow\!\!* \; s$.

The number of connecting attributes increases down the ownership hierarchy. This means, in practice, that in Eq. 4-3 $j = 1$ at the top level, and 1 greater at each lower level. Very deep hierarchies are rare, because the constraints are hard to manage. Similarly, the fanout s at each level tends to be modest, because it is not desirable to have a large volume of data dependent on one owner tuple. If the fanout is large, it is often wise to check if the reference connection provides better semantics. These observations are seen in Fig. 4-3.

Cardinality constraints can be applied to s, to constrain the contents of database relations, as shown in Fig. 4-41.

For instance, an inventory application may require that an `automobile` has $2 < s \leq 6$ `wheels`; otherwise it is assigned to another vehicle type. Given

```
  Automobile_types: RELATION make, model :⟩ style, weight, ...;
  Wheels: RELATION make, model, type :⟩ diameter, rim_width, ...;
Rolls_on: CONNECTION Automobile_types.(make, model)⟶* Wheels.(make, model)
                                        CONSTRAINT(1 :  3..6) ;
```

Figure 4-41 Cardinality constraint.

Maintaining constraints requires defining the action to be taken on violations. If, in Fig. 4-3, we constrain `Performs_as` to require at least one tuple per nest, we must delete "`Ed Lauter`" or find a `role` for him. Now the ownership semantics also affect the owning relation. Such constraints complicate update transactions; insertions on both the owning and the nest relation must be executed as an *atomic transaction*.

4-3 REFERENCE CONNECTIONS

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

Name: CONNECTION Primary.attributes ≻—— Foreign.ruling_attributes ; 4-5

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 made.

4-3-1 Semantics

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 the `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 relation is not permitted as long as it is being referenced.

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 4-5 shows connections for a valid network.

Given relations as shown in Fig. 3-19 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≻—— Producers.ref_name ;
Made_by: CONNECTION Movies.director ≻—— Directors.name ;
```

Figure 4-5 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, 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 three relations from Fig. 3-20:

```
Movies_info: RELATION film :⟩ title, year, director, studio, ... ;
Producers: RELATION name :⟩ birth_name, dob, dod, origin ;
Products: RELATION films, producer :⟩ position ;
```

and the connection `Who_is` from Fig. 4-5,

```
Who_is: CONNECTION  Producer.ref_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 4-6 Association with a reference connection.

A model including these references is sketched in Fig. 4-8.

4-3-2 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.

We can now define the structural rules.

Reference Rules (4-6)

Semantics: References indicate further information pertaining to the objects.

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 4-7 shows a model with a self reference.

We wish to show costars of `film actors` in our model. The `Casts` relation from Fig. 3-19 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, costar, award ;
Starred_with : CONNECTION ( Casts.(costar,film)⟩——Casts.actors,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. 4-4.

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

Figure 4-7 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. 4-8. Primary referencing relations are shown on the left. Their ruling parts must be accessible from outside of the model.

216pt Figure space

draw to 216pt 4-31, based on info from Figs. 4-5, 4-6, 4-7. Primary relations are shown on the left.

Figure 4-8 Foreign relations and their references.

4-3-3 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 $n \succ\!\!\!-\!\!\!-\!\!\!-\, 1$, just the opposite of the ownership connection.

4-3-4 Artificial Reference Tuples

The existence of a value in the primary attribute requires the existence of a corresponding foreign tuple. If a `null` value is permitted in a primary attribute, either the semantics of the reference connection must be changed or this fact has to be noted in the foreign relation, which defines the domain. We choose the second alternative in Fig. 4-9.

Similar tuples may also be appended to referenced relations for other special values shown in Table 4-2, as Λ and n.a.

Relations containing artificial reference tuples must be carefully manipulated. For instance a `COUNT` of `Studios` should not include these tuples.

We wish to expand the `Movies` relation of Fig. 3-9 with the films shown in Fig. 3-24. For one `film` (`H2`) we don't know the studio. To support the

```
Shot_in: CONNECTION  Movies.studio ≻—— Studios.name ;
```

of Fig. 4-5, we add the tuple shown here to the `Studios` relation of Fig. 3-10.

```
                        Studios: RELATION
                            name:⟩ company  ...     ...    ...;
Movies: RELATION        *          n.a      n.a.    n.a    n.a.
                         ________ ↑_______ _______ ______
film:⟩ title,  ...       ________ ________ _______ ...;
H2     No. 13  1922  Hitchcock  *        silent  n.a.
_______________________________ ______________________
```

We use the ↑ here to denote one link of the connection.

Figure 4-9 Link for artificial reference tuple.

♯♯♯♯♯♯ 4-3-5 Reference versus ownership

Ownership implements a hierarchy, where all lower elements depend on the owner. Such hierarchies are common, but easily misused. The frequently used relationship of `Departments` and their `Employees` is rarely appropriate for an ownership connection. One could lose all information about the `Employees` if an administrator abolishes a `department`. In a human-oriented world model `Employees` are the primary entity, and `Departments` are merely referenced abstractions.

Use of a *reference connection* 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. 4-6. ♯♯♯♯♯♯

‡‡‡‡‡‡ 4-3-6 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 Chap. ¡7-6¿. Different views are typically maintained by separate departments in an enterprise. An example of problems with foreign relations is shown in Fig. 4-10.

We use a single `Countries` relation for the `location` of `actors` for `Casting`, the `country` that `Studios` are in, for the `background` of `Directors`, and for the `origin` of `Producers` 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 from 1946 to 1990 `Studios` were either in the
"`Federal Republic of Germany`" or in the "`German Democratic Republic`".
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 4-10 Sources of errors in references.

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

4-4 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. 3-20 we had `Famous_actors`, `Producers`, and `Directors`. These are categories in the ERC model, as presented in Chap. 3-5-3. Some `People` perform multiple tasks, so that there is overlap between the categories. If we were to keep these overlapping subsets in wholly distinct relations similar attributes of shared objects and people will be redundant. Defining a general relation `People` permits non-redundant modeling of such 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

Name: CONNECTION General.ruling_part ⟹ Subset.ruling_part
[CONSTRAINT(Membership_condition)] ;

4-4-1 Semantics of subsets

The objects described in a subset relation are not independent. They derive their existence and their basic properties from corresponding tuples in the general relation. This inheritance is related to the concept used in artificial intelligence of the "*is*-a" relationship. In relational theory subsets are seen as *unary inclusion dependencies*, and we see that a subset is operationally similar to an ownership with a maximal cardinality of one. Inclusion dependencies were discussed in Chap. 3-4-2.

Inheritance of Connections Because of the strong dependency, a subset also inherits the connection semantics of the general relation. For instance, the connection

`Directors.background` $\succ\!\!-$ `Countries,country`

is also inherited by `MGM_directors`, if we define such a subset. Another example of connection inheritance is found in Fig. 6-¡12¿.

4-4-2 Structure of subsets

The use of subset connections permits us 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 (4-8)

Semantics: Subsets identify specializations of the objects.

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 gives depth to the database and is often needed in specific views. Subsets may have further subsets. We encountered already subsets of `Actors` in Fig. 3-5 and further subsets of `Regulars` in Fig. 3-6.

We define a `Directors_H` relation as containing those directors who influenced `Hitchcock` with some specific information. Being in that relation denotes that the particular director was considered relevant to `Hitchcock`'s career. All directors are part of a general relation `All_directors`. The relation schemas are

```
ALL_directors: RELATION name :⟩ first_name, dob, dod, background;
Directors_H_i: RELATION name :⟩ first_H_i_date, H_i_location;
```

The connection between them makes sure that we have `All_directors` information for any `Directors_H_i`.

```
Infl_Hitchcock: CONNECTION All_Directors.name ⟶⊃ Directors_H_i.name;
```

Figure 4-11 Unconstrained subset connection.

In Fig. 4-12 we show the *generalization*, People, and the subset connections to previously defined relations. Getting the ruling parts to match was not simple. 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. We also show constraints, which determine if a person should be a member of a specific subset.

4-4-3 Controlling the Existence of Subset Tuples

We can have *unconstrained subset membership* or impose *membership constraints*. In the first case we can simply enter a subset tuple, as shown in Fig. 4-11, to indicate membership.

The alternative is to have an explicit membership constraint in the connection. In Fig. 4-12 we impose simple boolean membership conditions for three subsets of People and list them as a CONSTRAINT (condition). The membership condition is similar to a selection list to be presented in Chap. 5-4 in Table 5-2. A subset tuple *must* exist if the corresponding attribute in the general relation satisfies the membership condition specified in the CONSTRAINT.

```
People: RELATION     -- general relation.
      last_name, first_name :⟩ born, died, sex, 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) CONSTRAINT(is_actor);

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

Is_producer: CONNECTION    -- connection to producers subset
  People.(last_name, first_name)⟶⊃ Producer_folk.(family, given)
      CONSTRAINT(is_producer);

Producer_folk: RELATION     -- name now split into two parts
      family, given :⟩ ref_name, productivity, current_studio;

Is_director: CONNECTION    -- connection to directors subset
      People.(last_name, first_name) ⟶⊃
      Director_folk.(name, first_name) CONSTRAINT(is_director);

Director_folk: RELATION     -- name used is unique
      name :⟩ first_name, cur_fee, cur_perctg, awards ;
```

Note that the subrelations contain further data, specific to their subclass. The general relation is essential to get a complete picture.

Figure 4-12 Subset connections.

4-4-4 Disjoint Subsets

We can impose a further constraint on subsets, namely *disjointness*. This constraint affects the semantics of computations that combine results from multiple subsets. Having disjoint subsets means that a general object can only be a member of one subset. That is not true for the `People` of Fig. 4-12; actors often become directors, and directors often turn to producing. But the types of `Actors` we recognized in Fig. 3-6 were disjoint.

Disjointness can be enforced through the membership constraint of the subset connection. A straightforward way to enforce that subsets are disjoint is to have an enumerated type and simple membership constraints of the form

$$\begin{aligned} &\text{CONSTRAINT (} att = ch_i) \\ &\qquad \text{where } D(att) = ch_1 \ldots ch_i \ldots ch_k \end{aligned} \tag{4-9}$$

In the example of Fig. 3-23 the controlling attribute is `actor_status` and the choices were `(leadg,regl,charc,suppt,extra)`, and $k = 5$.

One of the subsets is defined as shown in Fig. 4-13

For leading actors we define

```
Lead_actors: RELATION name :⟩ fee, percentage ;
```

and graphically we have

```
Is_lead: CONNECTION Actors.stage_name ⟶⊃ Lead_Actors.name
                    CONSTRAINT ( actor_status="leadg" ) ;
```

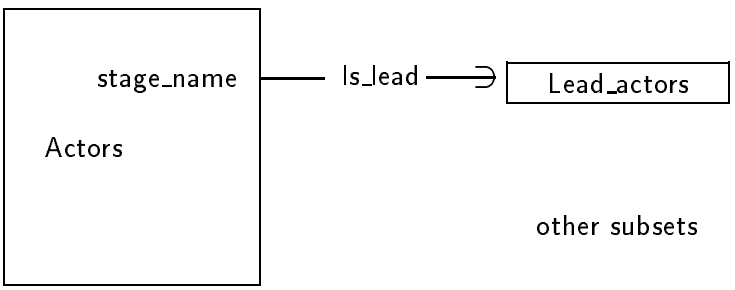


Figure 4-13 A constrained, disjoint subset

.

Furthermore, a *complete disjoint* set is assured if there is a subset connection corresponding to every one of the k choices. Having a complete set means that the general relation can either be processed subset-by-subset, for all k subsets, or directly, and that the results should be the same.

Without these membership constraints we could not be sure we would have considered all `Actors` if processing them subset by subset. Without disjointness some `Actors` might be processed twice or more often. If we charge them fees, they are likely to complain.

4-5 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)      4-10
                     | constraint_relaxation;
```

The constraint_relaxation plays an important role. 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*. In *modal logic* this operator is denoted by $\diamond$, but is not further quantified.

4-5-1 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* in the future the identity will be satisfied. At that time 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. 4-14.

In practice, a copy does not have to include all the attributes. Only relevant attributes from the source need to be copied to the destination. The `Publicity` department in Fig. 4-14 should not receive home addresses, for instance. The examples of Figs. 4-15, 4-16, and 4-17 show examples where the destination receives fewer or summarized attributes.

The publicity department of the studio uses a relation `Actors_info` to answer questions about its actors, which is a copy obtained from the personnel records relation `Actors`. The specification for its maintenance are defined as

```
Public:  CONNECTION Actors ——≡ Actors_info | Δ1d;
```

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

Figure 4-14 Use of an identity connection to link copies.

4-5-2 Delay Types

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. 4-15.

- `!` 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.

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

```
Set_hours: CONNECTION Studio_employees.(name,hours_worked)═══≡
        Payroll.(emp_name,hours_to_be_paid) | !pay_transaction ;
```

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

```
Show_schedule: CONNECTION
    Actors_schedules.(stage_name, title, character) ═══≡
        Studio_display.(emp_name,project,as) | @8:00:00;
```

Changes to the today's schedule should be displayed within five minutes:

```
Update_schedule: CONNECTION

⊆

Change_list.(stage_name, title, character) ○ (date=today) ═══≡
        Studio_display.(emp_name, project, as) | Δ5 min ;
```

Here a simple computation, selecting only changes applicable for today, is specified as well. Recomputation of the entire schedule is avoided.

Figure 4-15 Three types of delay specifications.

Fig. 4-15 also shows an example of a simple computation which dynamically selects a subset. The $\subseteq R \circ (cond)$ operator will be introduced in Chap. 5-4.

Identity rules (4-11)

Semantics: Identities define derivations of data.

Structure: Ruling parts of derived tuples match the ruling part of the source relation completely. All destination attributes of an identity relation match some source attribute or expression. Expressions giving derived values match the domain of the corresponding attribute in the destination relation. Multiple copies require multiple connection definitions.

Operation: Eventually corresponding identity tuples must be created and updated. Deletion of the source tuple forces eventual deletion of the derived tuple. The permissible delays are specified; timestamps will be needed for validation.

We are starting here a new approach to *declarative programming*.

4-5-3 Computation in Identity Connections

Computed values can be specified for dependent attributes in the source relation. The transformation computations of source values along an identity connection will apply to all valid links. The destination value will eventually contain the *derived value*, as shown in Fig. 4-16.

Say that we also have overtime hours for the employees. Overtime is to be paid at $1.5\times$ the normal rate. We can specify

```
Set_hours: CONNECTION
  studio_employees.(name, hours_worked+1.5*overtime)——≡
            Payroll.(emp_name,hours_to_be_paid) | !pay_transaction ;
```

Figure 4-16 Computed derived values on identity connection

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. 4-17. Aggregation functions as SUM are presented in Chap. 6-1.

Cinemas transmit weekly the gross receipts to the corporate office:

```
Send_money: CONNECTION
  S Cinema_receipts.(cinema_id, WITH(SUM(receipts)) o (
    day DURING[today-7,today-1)) )——≡
              Total_receipts.(theater, amount) | @Monday 8:00:00 ;
```

The temporal comparison operator DURING is defined in Table 7-¡10¿.

Figure 4-17 Delayed aggregation.

While it is possible to compute derived ruling parts as well, such computations are rare; an example is shown in Fig. ¡4-371¿. In general, only dependent part attributes participate in computed derivations.

Note, that having a specification model does not solve the implementation problem. 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. However, *without* specifications, we can never determine if the systems are behaving acceptably. Chapter 7 deals with these issues.

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 ($\equiv\equiv$). Subsets are important when views of the database, obtained from distinct groups needing the information are combined. This topic is expanded in Chap. 7-5 when discussing distributed design.

Connection name	source = destination role name → ← role name	type
Made_by	Movies.director=Directors.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.film	own
	Cast_is Performed	
Played_in	Famous_actors.stage_name=Casts.actor	own
	Played_in Has_actor	
Star_cast	Movies.film=Stars.films	own
	Star_cast_is Stars	
Starred_in	Famous_actors.stage_name=Stars.actor	own
	Starred_in Has_star	
Prod_by	Movies.film=Products.films	own
	Produced_by Film_is	
Products_of	Producer.ref_name=Products.producer	ref
	Products_of Producer_is	
Works_in	Casting.location=Countries.country	ref
	Works_in Has_available	
Is_a_director	movie_person.last_name = Directors.name	subset
	Is_director Belongs_to	

Figure 4-18 Role names.

4-6 NAMING OF CONNECTION ROLES

It is 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; those that 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 4-18 shows some examples of 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. 4-18 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.

4-7 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. For some people, or in some situations, dealing with entities and relations provides better insights than connections.

Within a database or application 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 an application. In an integrated database model the connection types are inherited from the original application 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 an application model in more detail. We also describe conditions and structural constraints. Together with normalizations these assure that all types of relations support a consistent update and deletion behavior. The relation types being introduced are summarized in Sec. 4-7-9.

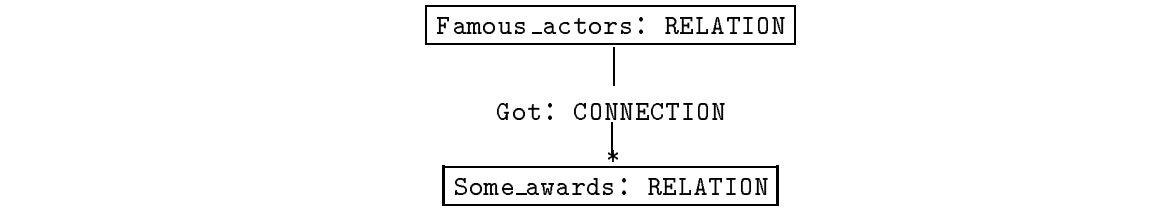
4-7-1 Primary Entity Relations

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 an application 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 an application model. The connections in an application model 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.

An application may have multiple primary entity relations. In Fig. 2-5 we showed part of a primary entity relation. The `Famous_actors` relation complements the `Movies_info` relation in the `Movie_buff` application.¿ The attributes may be basic values or references to tuples in other relations, as shown for `origin`, which refers to Fig. 4-25.




```
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 4-19 Nest relation.

4-7-2 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. 4-19 a nest relation for the tuples of Fig. 2-5. The tuples of these nest relations are dependent on the owner tuples for their continued existence. In Sec. 4-2 this constraint was formalized as the *ownership rules*.

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. 4-19: `stage_name, award :⟩` . This model only permits us to show one of each type of award, and hence only the `last` award of some type received is kept.

We can nest relations to multiple levels if we need to include more specific information, as sketched in Fig. 4-20. 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
   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
   Henry Fonda      AA   1981   acting     On Golden Pond
   Henry Fonda      AA   1982   honorary     n.a.
   Cary Grant       AA   1969   special      n.a.
   Gregory Peck     AA   1963   acting     To Kill a Mockingbird
```

Figure 4-20 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. 4-20 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. 3-5 has two potential nest relations, `Roles_Played` and `Schedule`. Tuples from multiple nest relations should not be stored together (Eq. 3-5), as was demonstrated when `Movies` with multiple `Directors` and `Actors` were enumerated in Fig. 3-14.

4-7-3 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. Another foreign entity relation in our example is the `Studio` relation iteself; it is referenced by `Movies`.

A foreign entity relation also obeys all rules established for relations. Since no redundant tuples are permitted, the cardinality of the foreign relation will be equal or less, often much less, than the cardinality of 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 entity relation do not depend on the tuples of the primary entity. In Fig. 3-23 `Countries` is a foreign entity relation. When no actors reference a specific `Country` entity object, say `"Austria"`, the idle information remains valid and should be kept. Such foreign information persists independently. Little delay will then occur if a new `Casting` entry requires the idle entry.

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.

4-7-4 Associative Relations

Relationships can contain data which depend on the interaction or association of multiple 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 3-27 presented that association.

Attributes values of the connections relate each tuple in the associative relation to tuples in both entity relations. It is also possible to associate three or more entity relations, although it is rare not to need intermediate pairing relations. These intermediate relations will take on the role of entities for further associations.

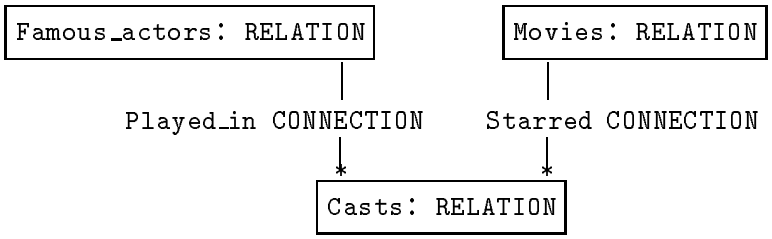
A simple example is shown in Fig. 4-21; there will not be tuples for all possible or even existing intersections in the association `Street_corner`. The semantics of having an entry in `Street_corner` is that there is a `traffic_signal`.

90pt Figure space

was Fig.7-15, fits in 90pt

Figure 4-21 Associative relation.

The connections from the entity relations to the association are ownership or reference connections, and all the connection rules apply. Ownership connections are common, and we will focus on these. An associative relation, as `Casts`, has a ruling part composed of the ruling parts of both entity relations.



Casts_in_postwar_bw_Hitchcock_films: RELATION

actors	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 4-22 Associative relation.

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

$$A_{association} = \bigcup_{owners} A_{owners} \qquad 4\text{-}12$$

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

There may be $0 \rightarrow s$ tuples corresponding to one tuple in any of the owner relations. Associations rapidly become large, since they are a subset of the crossproduct of their entities.

Associative relations often do not have a dependent part. In our model this would be appropriate for a `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. 4-23.

`Post1940_Hitchcock_Products: RELATION`

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

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

Figure 4-23 Association without dependent attributes

We can have more than one association between the same entities. The dependent parts in that case are likely to differ. We show examples, derived from distinct applications, in Fig. 4-24. Relations listing potential associations rarely have dependent attributes.

The semantic difference of multiple associations among entities is not obvious from the graphical representation of the connection structure and has to be described separately. Careful naming of the associative relation and the connection names will help.

An association can also link tuples within the same relation. Cycles in a database model create potential complexities. For instance, `Casts` can be used to determine which `Actors` worked together on a `Movie`, but a cycle via `Casts` can be used to find also second-level realtionships, namely `Actors` who worked with `Actors` who worked together. In Chap. 6 we will present operations to deal with such cycles, as well as operations to deal with the generalization, namely recursion to locate all levels of relationships.

For Movie_buffs, movies that are released:

Actors ——* Casts *—— Movies

For investors, potential, new movies:

Actors ——* Casting *—— Movies

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

If we were to associate multiple Directors with their Movies, as well as Casts and Producers, we would have four associations between Movie_people and Movies.

From another domain we can have:

Physician ——≺ Visits *—— Patients
Physician ——≺ Assigned *—— Patients

Figure 4-24 Multiple associations among entities.

4-7-5 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 example is shown in Fig. 4-25.

We referred to Countries in Studios by name, say England, but in Famous_actors by adjective, as British. We really meant the same object and intend to refer to the same 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 4-25 Multiple candidate ruling parts.

To resolve the ambiguities in the dependencies when there are multiple candidate ruling parts we move redundant ruling parts of a relation $MR\colon\ A_{L_0}, A_{L_1} :\rangle\ B;$ into a very specific type of relation, the *Lexicon* $L\colon\ A_{L_0} \langle:\rangle\ A_{L_1};$. The ambiguous relation MR becomes $R\colon\ A_{L_i} :\rangle\ B;$. After this process

$$\begin{aligned} L &: FD(A_{L_0} :\rangle\ A_{L_1});\ FD(A_{L_1} :\rangle\ A_{L_0}) \\ R &: FD(A_{L_i} :\rangle\ B); A_{L_{1-i}} \quad / \quad B \end{aligned} \tag{4-13}$$

where A and B represent attributes as in Eq. 3-4. Now Eq. 3-4 is satisfied for S, but the lexicon relation L is further constrained. The resulting lexicon for Fig. 4-25 is shown in Fig. 4-26.

Of_origin: LEXICON

country_name ⟨:⟩	origin_adjective ;
Austria	Austrian
Canada	Canadian
Great Britain	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 4-26 A Lexicon.

A lexicon implements equivalency. Tuples in a lexicon are typically referenced to find further references to matching tuples in other relations. Since the ruling and the dependent part are symmetric, either attribute can be used as the referenced attribute.

They are as constrained as any foreign relation. In models this structural dependency is represented by having *reference connections* between the 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 4-27 Use of a lexicon.

4-7-6 Subrelations

Defining relations to cover general classes is an important abstraction, Using broad concepts permits the application of analyses to large sets of objects. However, the objects often differ in detail. To capture systematic differences we maintain subrelations. Subrelations are connected to the general relation by subset connections, as introduced in Sec. 4-4. We will only summarize the effect of the alternatives presented there.

Subset relations exist in several types. We can distinguish them by the formality of membership. If the semantics of membership are expressed by rules operating on data in the general relations we have formal constraints. If we dont have rules, or the rules are not part of the formal database, then membership is determined by the programs or users manipulating the database. We consider here 2×2 types of subrelations:

Table 4-3 Types of subset relations.

		Sub-object existence: disjoint	overlapping
Sub-object selection	by attribute value	essential (one)	preferred (many)
	by insertion/deletion	unsafe	feasible

If membership in a group of subrelations is disjoint and also complete, then we have a subrelation tuple for every general tuple. Completeness can be enforced if `CONSTRAINTS`, as shown in Fig. 4-13, are defined for every possible value of an attribute. This is most likely for enumerated values. A complete set is shown in Fig. 4-28.

For `actor_status : LISTleadg,regl,charc,suppt,extra`, we can define five subset connections with `CONSTRAINTS` to maintain

```
Lead_actor: RELATION    actor:⟩ fee,percentage, type_cast ;
Regular_actor: RELATION actor:⟩ contract, fee, type_cast ;
Character: RELATION     actor:⟩ contract, percentage, type_cast, recognition;
Support_actor: RELATION actor:⟩ contract, role;
Extra: RELATION         actor:⟩ hourly_rate, availability;
```

Each actor is also represented in one of thes relations.

Figure 4-28 Complete disjoint subrelations.

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. 4-29.

```
Is_VIP:  CONNECTION  Customer.name ⟶⊃ VIP.name |
    CONSTRAINT( income > 1 000 000 OR capital > 10 000 000  OR
          position > GS-14 OR Casting.status = "Leadg" ) ;
```

Figure 4-29 Complex subrelation determination. ‡‡‡‡‡‡

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 incomplete and overlapping subsets. In Fig. 4-11 we placed no constraints on the relation containing directors which influenced `Hitchcock`, except, of course, that they be in the `Directors` relation. It would be very difficult to write a formal constraint for that case.

4-7-7 Summary of Relation Types

In this section we presented how semantic concepts that connect relations in an application model can be presented in terms of eight relation types:

Entity relations
Foreign entity relations
Nest relations
Associative relations
Lexicons
Subrelations
Derived relations *See Chap. 7.2*

378pt Figure space

was Fig.7-18, 378pt, Change directions of second set of pointers for nest

---

Figure 4-30 An erector set of relation types.

Figure 4-30 provides a graphical image of the types covered this far using the graphic symbols we have used for connections and plain arrows for functional dependencies within the tuples themselves. Use of graphic symbols is extremely useful in presenting application models for review to the participants in a database design process. Tables or lists do not effectively illustrate the dependencies and hence the semantics of the application model.

The relation types are based on the four connection types defined in this chapter:

Reference connections
Ownership connections
Subset connections
Identity connections

The two concepts are duals of each other; a relation type defines the connection types 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.

Constraints between relations are more easily described using connections. Data elements and their semantics are better described using relations. Depending on the style of DBMS one concept may receive more emphasis in a database systems. A data-processing transaction acts on relations, but is constrained by connections. When the application is approved, automatic processing of the information in the application models to arrive at a database model becomes feasible. Table 4-4 reviews the semantics of the relation types described, using the rules defined with the connections.

Derived relations are *materialized* and stored relations that can be automatically recomputed from base relations in the database. Use of the identity connection can lead to derived relations, Derivations are covered in Chap. 7.

Table 4-4 Connection semantics applied to relation types.

A	*A primary entity relation:*
1	Not referenced within the application 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 application 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 application 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.

Table 4-4 continued Connection semantics.

D *An associative relation of order n:*
1 Each tuple has n owners within the application 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 application 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.

G *Derived relations:*
1 Generated from any source relation or equivalent expression.
2 The ruling part matches the ruling part of the source relation.
3 The dependent part is the same and eventually contains the same values.
4 Insertion requires existence of a matching tuple in the source 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 italics.

It is possible to create from these basic semantic relation types other forms of relations which satisfy special conditions. A synthesis of published material indicates that these seven types, and their combinations, cover the semantic possibilities controlling the structure of databases in an economically 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 the application model in an economical 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 application model. If the application model is to become a basis for extended applications using the database, it may be transformed to a *database view model*.

¡¡A classical example is the *inventory* which associates parts used by a manufacturing company and the suppliers of these parts. ¿¿

Table 4-5 Backus-Naur description of relation and connection 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⟩	
⟨source attributes⟩	::=⟨value attributes⟩|⟨expressions⟩	
⟨derived attributes⟩	::=⟨identifiers⟩	
⟨identifiers⟩	::=⟨value attributes⟩	*unique objects*
⟨reference key⟩	::=⟨reference attribute⟩	*unique objects*
⟨primary entity key⟩	::=⟨identifiers⟩	*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 part⟩	::=⟨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⟩	
⟨subset ruling part⟩	::=⟨entity ruling part⟩	
⟨ruling part⟩	::=⟨entity ruling part⟩|⟨foreign ruling part⟩|⟨nest ruling part⟩	
	|⟨assoc. ruling part⟩|⟨general ruling part⟩|⟨subset 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⟩	::=⟨ruling part⟩⟨:⟩ ⟨dependent part⟩;	
⟨subrelation⟩	::=⟨subset ruling part⟩ :⟩ ⟨dependent part⟩;	
⟨derivation⟩	::=⟨ruling part⟩ :⟩ ⟨derived attributes⟩;	
CONNECTIONS:		
⟨reference⟩	::=⟨reference attribute⟩$\succ$—— :⟩ ⟨foreign ruling part⟩	
⟨ownership⟩;	::=⟨ruling part⟩——$*$ ⟨nest ruling part⟩[⟨cardinality];	
⟨subset⟩	::=⟨general ruling part⟩——$\ni$ ⟨subset ruling part⟩[⟨membership⟩];	
⟨identity⟩	::=⟨source attributes⟩——$\equiv$ ⟨derivation⟩⟨delay⟩];	
CONSTRAINTS:		
⟨cardinality⟩	::=*minimum, maximum number of members*	
⟨membership⟩	::=CONSTRAINT(*select conditions for membership*)	
⟨delay⟩	::=!⟨procedure⟩ | @time | Δ interval	

4-8 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*.

From this chapter you should have gained insight into how significant data semantics can be defined and used to specify a correct structure of the database. Table 4-6 recapitulates the terms we have encountered. If any modeling terms are unclear, you can use the index to find where they are descibed.

Table 4-6 Terminology.

Semantics	Model	Software	Hardware
Entity set	Relation	File	Disk
Description	Relation-schema	Schema (see Chap. 8)	Documentation
Object	Tuple	Record	Block
Relationship	Connection	Pointer-type	Address
Link	Link	Ref.Pointer	Pointer
	Role	Pointer direction	
Object name	Ruling part	Key	Address-field
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

4-8-2 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. This model should satisfy the data semantics of the application 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 program of the database management system, some data-processing functions can be automated. Requests (queries and updates) to such a system can now be expressed in a language which is at a higher-level than a programming language. Such a language, which lets us state the goal to be achieved, rather than the procedure to follow to get a solution, is called a *non-procedural language*. Processing of statements given in a non-procedural language requires a program which has an inventory of solution methods, and can select the best combination to achieve the goal.

The direction towards non-procedural methods is taken a step further with the *identity connection*. We can use one connection to specify a simple computation. Networks of multiple identity connections can describe complex computations, which are now scheduled and managed wholly by the system, rather than by explicit transaction programs. By replacing the control aspects of programmming with declarations we are removing the most error-prone aspect of programming.

4-8-3 A Comprehensive Model

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.

380pt Figure space

new, to be drawn, in 380 pt, To include refs of Fig. 4-6

Figure 4-31 A comprehensive model.

BACKGROUND AND REFERENCES

This chapter specifies a major augmentation to the relation model: connections. The concept of connections is related to a wide variety of abstractions that have been use to model databases.

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 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 for our objective Miller[56]. 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 presented the relational model and some early, implementation-oriented models in Chap. 3. We now present the background of some design-oriented models.

The family of ER models User 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 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]]. A disadvantage is that functions are so powerful that the semantics are difficult to capture.
- The structural model formalizes relationships as connections between relations and adds operational semantics. This model also 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.
- Binary relational models reduce relations to have one attribute in the dependent part.

We discuss these models briefly further.

The entity-relationship model (ER) The original ER model led to a series of conferences where further developments were published [Chen[80] ¡...¿]. It related well to the needs of the community and was well accepted. The structural model was presented in Wiederhold[80] and used for integration of data models by ElMasri[80a], who also compared its coverage to other models [ElMasri[80b]]. Versions of ER modeling are now being supported commercially [Appleton[xx], ...].

The structural model The model used here was defined partially in [Wiederhold[77]] as an attempt to represent the structural features of databases and their representation models within the formal environment of the relation model. It was further formalized in Wiederhold[80] and used to define a method to integrate databases [ElMasri[79]]. In ElMasri[80] it was compared with other models, and a query language exploiting its semantics was presented in Elmasri[81]. The concept of an identity connection was defined in Wiederhold[87].

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 ¡insert Q here¿

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 [Ullman^{xx}]. The universal relation can be seen as one large matrix, with all candidate attributes labelling columns 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.

Binary relation model Models used by Abrial[¡xx¿] and Feldman[¡xx¿] reduce relationships to their essentials by only representing binary relationships. All tuples are triplets of the form `entity-id, attribute, value`. Many such relations are needed. No further schema is required now, all tuples look the same. Some of the tuples can of course express a schema, as `relation-schema-id, has-attribute, attribute`. Some triplets may be used to specify inversions of attribute direction as `value, is-used-in, entities`.

An effect of the complete decomposition is that normalization is no longer needed. The equivalent of a relation is constructed from n triplets containing `relation-name, has-member, entity-id`. Reference values implement arbitrary linkages . New tuple types can be inserted at any time.

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. The 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 We define a relation for our actors listing their particular skills as

```
Skills: RELATION
   actor,       skill,         degree,     since,  example;
   ------------ -------------- ----------- ------- --------
   John Wayne   rides horses   well        1937    XYZ13
   John Wayne   shoots         excellent   1943    XXX14
   ...          ...            ...         ...
   ------------------------------------------------------
```

What are appropriate connection types for `Actors` ——• `Skills` and `Skills` ——• `Movies`? Specify the full connection.

2 Assume that the ownership connection `Performs_as` (Fig. 4-3) has not been maintained and `Casting.actor = "Barbara Harris"` has been deleted. What will happen if we look for the ages of `Actors` in `Movies` while playing the role of a wife, as recognized by a string search for `.. wife ..`?

7 What are the semantics of the `Casts` example shown in Fig. 4-22? What cardinality do you expect for `Casts`. Make reasonable assumptions about the cardinality of the entities.

9 Discuss why the `role` attribute might be better placed in the ruling part in the `Roles_played` relation of Fig. 4-3? Hint: look at the structure of `Casts_M` in Fig. 3-15.

10 Why did we not define a

`Came_from: CONNECTION Actor.origin` ≻—— `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. 4-3) they belong.

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

14 For a model consisting of `Employees`, each having `Skills` and assigned to multiple `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 types. Pilots may be qualified to fly more than one airplane type. Define the connections and assign role names to both directions.

SOLUTIONS TO THE EXERCISES

4-1. Connections The Skills depend on the Actors for their existence:

```
Has_skills: CONNECTION Actors.stagename ⟶* Skills.actor ;.
```

The movie citation example should always reference existing Movies.

```
Where_is : CONNECTION Actors.example ≻—— Movies.film;.
```

4-2. Missing link We get an unexpected failure when a link is missing. To compute the age we need both the dob of the Actors and the year of the Movies. If the programs don't fail, we will still get more films than actors.

4-7. Association semantics Having a tuple in Casts means that the actor performed in that film. Even without dependent attributes there is an information value to finding a tuple in Casts. The association is quite sparse, for about 5000 films and 3000 actors we had about $40\,000$ of the possible $15\,000\,000$ entries. Any assumption within a factor of 10 is fine, but the products should be proportional. The number of listed actors for a film varies from 0 to 30. Animated films have no actors, but we also have films listed where the actors are not known.

4-9. Ruling Part Since it is possible to play different roles in one movie, as commonly done by Peter Sellers, the current ruling part is not necessarily unique.

The other choice to make, the last_performed year, which is otherwise a reasonable choice, does not guarantee uniqueness.

4-10. 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. 5-5) can compute the linkages here. Note that our model assumes only one language per country, requiring that regions as "Wales" be made a separate countries. A more realistic model would require an associative relation Spoken: RELATION language, country :⟩;.

4-13. Subset Connections

4-14. 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

4-15. 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 managed Employees would also be deleted.

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

4-16. Pilots and Airplanes. Since the exercise describes an $m:n$ relationship, we have to introduce a new relation: Airplane_types 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_types can be flown by	ownership
Is_type	Airplanes is_of_type	Airplane_types is_in_fleet	reference