

It is the logical data model created in Step 2 that forms the starting point for physical database design, which is described as Steps 3 to 8 in Chapters 18 and 19. Throughout the methodology, the terms “entity” and “relationship” are used in place of “entity type” and “relationship type” where the meaning is obvious; “type” is generally only added to avoid ambiguity.

17.1 Logical Database Design Methodology for the Relational Model

This section describes the steps of the logical database design methodology for the relational model.

Step 2: Build Logical Data Model

Objective

To translate the conceptual data model into a logical data model and then to validate this model to check that it is structurally correct and able to support the required transactions.

In this step, the main objective is to translate the conceptual data model created in Step 1 into a **logical data model** of the data requirements of the enterprise. This objective is achieved by following these activities:

- Step 2.1 Derive relations for logical data model
- Step 2.2 Validate relations using normalization
- Step 2.3 Validate relations against user transactions
- Step 2.4 Check integrity constraints
- Step 2.5 Review logical data model with user
- Step 2.6 Merge logical data models into global data model (optional step)
- Step 2.7 Check for future growth

We begin by deriving a set of relations (relational schema) from the conceptual data model created in Step 1. The structure of the relational schema is validated using normalization and then checked to ensure that the relations are capable of supporting the transactions given in the users’ requirements specification. We next check whether all important integrity constraints are represented by the logical data model. At this stage, the logical data model is validated by the users to ensure that they consider the model to be a true representation of the data requirements of the enterprise.

The methodology for Step 2 is presented so that it is applicable for the design of simple to complex database systems. For example, to create a database with a single user view or with multiple user views that are managed using the centralized approach (see Section 10.5) then Step 2.6 is omitted. If, however, the database has multiple user views that are being managed using the view integration approach (see Section 10.5), then Steps 2.1 to 2.5 are repeated for the required number of data models, each of which represents different user views of the database system. In Step 2.6, these data models are merged.

Step 2 concludes with an assessment of the logical data model, which may or may not have involved Step 2.6, to ensure that the final model is able to support possible future developments. On completion of Step 2, we should have a single



Step 2.1: Derive relations for logical data model**Objective**

To create relations for the logical data model to represent the entities, relationships, and attributes that have been identified.

In this step, we derive relations for the logical data model to represent the entities, relationships, and attributes. We describe the composition of each relation using a DBDL for relational databases. Using the DBDL, we first specify the name of the relation, followed by a list of the relation's simple attributes enclosed in brackets. We then identify the primary key and any alternate and/or foreign key(s) of the relation. Following the identification of a foreign key, the relation containing the referenced primary key is given. Any derived attributes are also listed, along with how each one is calculated.

The relationship that an entity has with another entity is represented by the primary key/ foreign key mechanism. In deciding where to post (or place) the foreign key attribute(s), we must first identify the “parent” and “child” entities involved in the relationship. The parent entity refers to the entity that posts a copy of its primary key into the relation that represents the child entity, to act as the foreign key.

We describe how relations are derived for the following structures that may occur in a conceptual data model:

- (1) strong entity types;
- (2) weak entity types;
- (3) one-to-many (1:*) binary relationship types;
- (4) one-to-one (1:1) binary relationship types;
- (5) one-to-one (1:1) recursive relationship types;
- (6) superclass/subclass relationship types;
- (7) many-to-many (*:*) binary relationship types;
- (8) complex relationship types;
- (9) multi-valued attributes.



For most of the examples discussed in the following sections, we use the conceptual data model for the StaffClient user views of *DreamHome*, which is represented as an ER diagram in Figure 17.1.

(1) Strong entity types

For each strong entity in the data model, create a relation that includes all the simple attributes of that entity. For composite attributes, such as *name*, include only the constituent simple attributes: *fName* and *lName* in the relation. For example, the composition of the *Staff* relation shown in Figure 17.1 is:

Staff (staffNo, fName, lName, position, sex, DOB)

Primary Key staffNo

(2) Weak entity types

For each weak entity in the data model, create a relation that includes all the simple attributes of that entity. The primary key of a weak entity is partially or fully derived

cannot be made until after all the relationships with the owner entities have been mapped. For example, the weak entity *Preference* in Figure 17.1 is initially mapped to the following relation:

Preference (prefType, maxRent)

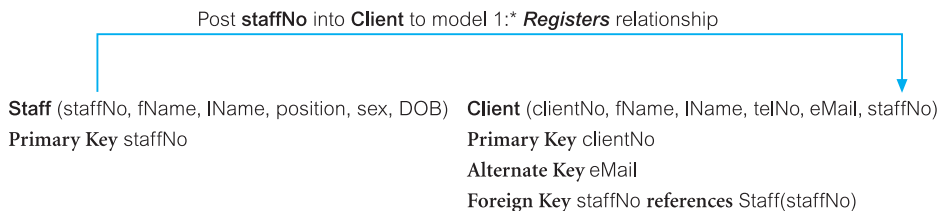
Primary Key None (at present)

In this situation, the primary key for the *Preference* relation cannot be identified until after the *States* relationship has been appropriately mapped.

(3) One-to-many (1:*) binary relationship types

For each 1:* binary relationship, the entity on the “one side” of the relationship is designated as the parent entity and the entity on the “many side” is designated as the child entity. To represent this relationship, we post a copy of the primary key attribute(s) of the parent entity into the relation representing the child entity, to act as a foreign key.

For example, the *Staff Registers Client* relationship shown in Figure 17.1 is a 1:* relationship, as a single member of staff can register many clients. In this example *Staff* is on the “one side” and represents the parent entity, and *Client* is on the “many side” and represents the child entity. The relationship between these entities is established by placing a copy of the primary key of the *Staff* (parent) entity, *staffNo*, into the *Client* (child) relation. The composition of the *Staff* and *Client* relations is:



In the case where a 1:* relationship has one or more attributes, these attributes should follow the posting of the primary key to the child relation. For example, if the *Staff Registers Client* relationship had an attribute called *dateRegister* representing when a member of staff registered the client, this attribute should also be posted to the *Client* relation, along with the copy of the primary key of the *Staff* relation, namely *staffNo*.

(4) One-to-one (1:1) binary relationship types

Creating relations to represent a 1:1 relationship is slightly more complex, as the cardinality cannot be used to help identify the parent and child entities in a relationship. Instead, the participation constraints (see Section 12.6.5) are used to help decide whether it is best to represent the relationship by combining the entities involved into one relation or by creating two relations and posting a copy of the primary key from one relation to the other. We consider how to create relations to represent the following participation constraints:

- (a) *mandatory* participation on *both* sides of 1:1 relationship;
- (b) *mandatory* participation on *one* side of 1:1 relationship;
- (c) *optional* participation on *both* sides of 1:1 relationship;
- (d) *optional* participation on *one* side of 1:1 relationship.

(a) Mandatory participation on both sides of 1:1 relationship In this case we should combine the entities involved into one relation and choose one of the primary keys of the original entities to be the primary key of the new relation, while the other (if one exists) is used as an alternate key.

The Client States Preference relationship is an example of a 1:1 relationship with mandatory participation on both sides. In this case, we choose to merge the two relations together to give the following Client relation:

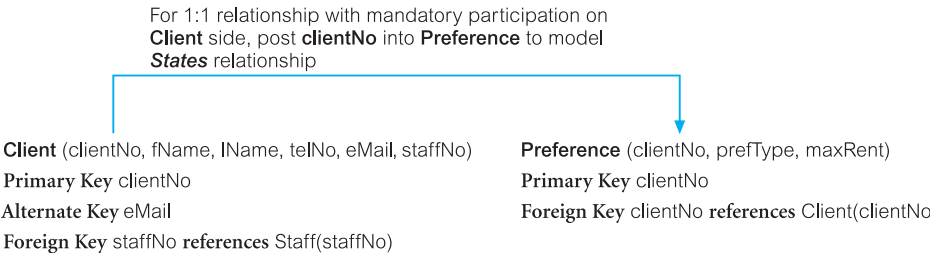
Client (clientNo, fName, lName, telNo, eMail, prefType, maxRent, staffNo)
Primary Key clientNo
Alternate Key eMail
Foreign Key staffNo **references** Staff(staffNo)

In the case where a 1:1 relationship with mandatory participation on both sides has one or more attributes, these attributes should also be included in the merged relation. For example, if the States relationship had an attribute called dateStated recording the date the preferences were stated, this attribute would also appear as an attribute in the merged Client relation.

Note that it is possible to merge two entities into one relation only when there are no other direct relationships between these two entities that would prevent this, such as a 1:* relationship. If this were the case, we would need to represent the States relationship using the primary key/foreign key mechanism. We discuss how to designate the parent and child entities in this type of situation in part (c) shortly.

(b) Mandatory participation on one side of a 1:1 relationship In this case we are able to identify the parent and child entities for the 1:1 relationship using the participation constraints. The entity that has optional participation in the relationship is designated as the parent entity, and the entity that has mandatory participation in the relationship is designated as the child entity. As described previously, a copy of the primary key of the parent entity is placed in the relation representing the child entity. If the relationship has one or more attributes, these attributes should follow the posting of the primary key to the child relation.

For example, if the 1:1 Client States Preference relationship had partial participation on the Client side (in other words, not every client specifies preferences), then the Client entity would be designated as the parent entity and the Preference entity would be designated as the child entity. Therefore, a copy of the primary key of the Client (parent) entity, clientNo, would be placed in the Preference (child) relation, giving:



Note that the foreign key attribute of the *Preference* relation also forms the relation's primary key. In this situation, the primary key for the *Preference* relation could not have been identified until after the foreign key had been posted from the *Client* relation to the *Preference* relation. Therefore, at the end of this step we should identify any new primary key or candidate keys that have been formed in the process, and update the data dictionary accordingly.

(c) Optional participation on both sides of a 1:1 relationship In this case the designation of the parent and child entities is arbitrary unless we can find out more about the relationship that can help us make a decision one way or the other.

For example, consider how to represent a 1:1 *Staff Uses Car* relationship with optional participation on both sides of the relationship. (Note that the discussion that follows is also relevant for 1:1 relationships with mandatory participation for both entities where we cannot select the option to combine the entities into a single relation.) If there is no additional information to help select the parent and child entities, the choice is arbitrary. In other words, we have the choice to post a copy of the primary key of the *Staff* entity to the *Car* entity, or vice versa.

However, assume that the majority of cars, but not all, are used by staff, and that only a minority of staff use cars. The *Car* entity, although optional, is closer to being mandatory than the *Staff* entity. We therefore designate *Staff* as the parent entity and *Car* as the child entity, and post a copy of the primary key of the *Staff* entity (*staffNo*) into the *Car* relation.

(5) One-to-one (1:1) recursive relationships

For a 1:1 recursive relationship, follow the rules for participation as described previously for a 1:1 relationship. However, in this special case of a 1:1 relationship, the entity on both sides of the relationship is the same. For a 1:1 recursive relationship with mandatory participation on both sides, represent the recursive relationship as a single relation with two copies of the primary key. As before, one copy of the primary key represents a foreign key and should be renamed to indicate the relationship it represents.

For a 1:1 recursive relationship with mandatory participation on only one side, we have the option to create a single relation with two copies of the primary key as described previously, or to create a new relation to represent the relationship. The new relation would have only two attributes, both copies of the primary key. As before, the copies of the primary keys act as foreign keys and have to be renamed to indicate the purpose of each in the relation.

For a 1:1 recursive relationship with optional participation on both sides, again create a new relation as described earlier.

(6) Superclass/subclass relationship types

For each superclass/subclass relationship in the conceptual data model, we identify the superclass entity as the parent entity and the subclass entity as the child entity. There are various options on how to represent such a relationship as one or more relations. The selection of the most appropriate option is dependent on a number of factors, such as the disjointness and participation constraints on the

TABLE 17.1 Guidelines for the representation of a superclass/subclass relationship based on the participation and disjoint constraints.

PARTICIPATION CONSTRAINT	DISJOINT CONSTRAINT	RELATIONS REQUIRED
Mandatory	Nondisjoint {And}	Single relation (with one or more discriminators to distinguish the type of each tuple)
Optional	Nondisjoint {And}	Two relations: one relation for superclass and one relation for all subclasses (with one or more discriminators to distinguish the type of each tuple)
Mandatory	Disjoint {Or}	Many relations: one relation for each combined superclass/subclass
Optional	Disjoint {Or}	Many relations: one relation for superclass and one for each subclass

superclass/subclass relationship (see Section 13.1.6), whether the subclasses are involved in distinct relationships, and the number of participants in the superclass/subclass relationship. Guidelines for the representation of a superclass/subclass relationship based only on the participation and disjoint constraints are shown in Table 17.1.

For example, consider the Owner superclass/subclass relationship shown in Figure 17.1. From Table 17.1, there are various ways to represent this relationship as one or more relations, as shown in Figure 17.2. The options range from placing all the attributes into one relation with two discriminators pOwnerFlag and bOwnerFlag indicating whether a tuple belongs to a particular subclass (Option 1) to dividing the attributes into three relations (Option 4). In this case the most appropriate representation of the superclass/subclass relationship is determined by the constraints on this relationship. From Figure 17.1, the relationship that the Owner superclass has with its subclasses is *mandatory* and *disjoint*, as each member of the Owner superclass must be a member of one of the subclasses (PrivateOwner or BusinessOwner) but cannot belong to both. We therefore select Option 3 as the best representation of this relationship and create a separate relation to represent each subclass, and include a copy of the primary key attribute(s) of the superclass in each.

It must be stressed that Table 17.1 is for guidance only as there may be other factors that influence the final choice. For example, with Option 1 (mandatory, nondisjoint) we have chosen to use two discriminators to distinguish whether the tuple is a member of a particular subclass. An equally valid way to represent this would be to have one discriminator that distinguishes whether the tuple is a member of PrivateOwner, BusinessOwner, or both. Alternatively, we could dispense with discriminators all together and simply test whether one of the attributes unique to a particular subclass has a value present to determine whether the tuple is a member of that subclass. In this case, we would have to ensure that the attributes examined allowed nulls to indicate nonmembership of a particular subclass.

In Figure 17.1, there is another superclass/subclass relationship between Staff and Supervisor with optional participation. However, as the Staff superclass only has

<p><u>Option 1 – Mandatory, nondisjoint</u></p> <p>Allowner (ownerNo, address, telNo, fName, lName, bName, bType, contactName, pOwnerFlag, bOwnerFlag)</p> <p>Primary Key ownerNo</p>
<p><u>Option 2 – Optional, nondisjoint</u></p> <p>Owner (ownerNo, address, telNo)</p> <p>Primary Key ownerNo</p> <p>OwnerDetails (ownerNo, fName lName, bName, bType, contactName, pOwnerFlag, bOwnerFlag)</p> <p>Primary Key ownerNo</p> <p>Foreign Key ownerNo references Owner(ownerNo)</p>
<p><u>Option 3 – Mandatory, disjoint</u></p> <p>PrivateOwner (ownerNo, fName, lName, address, telNo)</p> <p>Primary Key ownerNo</p> <p>BusinessOwner (ownerNo, bName, bType, contactName, address, telNo)</p> <p>Primary Key ownerNo</p>
<p><u>Option 4 – Optional, disjoint</u></p> <p>Owner (ownerNo, address, telNo)</p> <p>Primary Key ownerNo</p> <p>PrivateOwner (ownerNo, fName, lName)</p> <p>Primary Key ownerNo</p> <p>Foreign Key ownerNo references Owner(ownerNo)</p> <p>BusinessOwner (ownerNo, bName, bType, contactName)</p> <p>Primary Key ownerNo</p> <p>Foreign Key ownerNo references Owner(ownerNo)</p>

Figure 17.2
Various representations of the Owner superclass/subclass relationship based on the participation and disjointness constraints shown in Table 17.1.

one subclass (Supervisor), there is no disjoint constraint. In this case, as there are many more “supervised staff” than supervisors, we choose to represent this relationship as a single relation:

Staff (staffNo, fName, lName, position, sex, DOB, supervisorStaffNo)

Primary Key staffNo

Foreign Key supervisorStaffNo references Staff(staffNo)

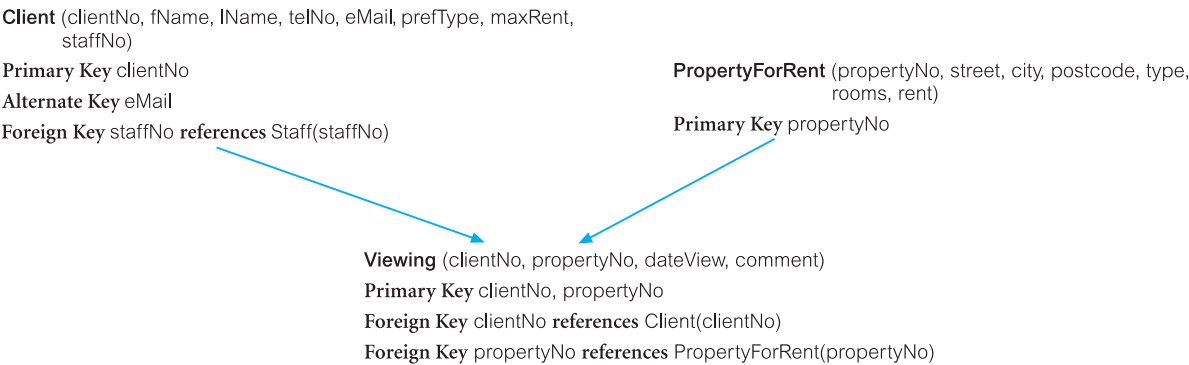
If we had left the superclass/subclass relationship as a 1:* recursive relationship as we had it originally in Figure 17.5 with optional participation on both sides this would have resulted in the same representation as previously.

(7) Many-to-many (*:*) binary relationship types

For each *:~ binary relationship, create a relation to represent the relationship and include any attributes that are part of the relationship. We post a copy of the primary key attribute(s) of the entities that participate in the relationship into the new relation, to act as foreign keys. One or both of these foreign keys will also form the primary key of the new relation, possibly in combination with one or more of the attributes of the relationship. (If one or more of the attributes that

form the relationship provide uniqueness, then an entity has been omitted from the conceptual data model, although this mapping process resolves this issue.)

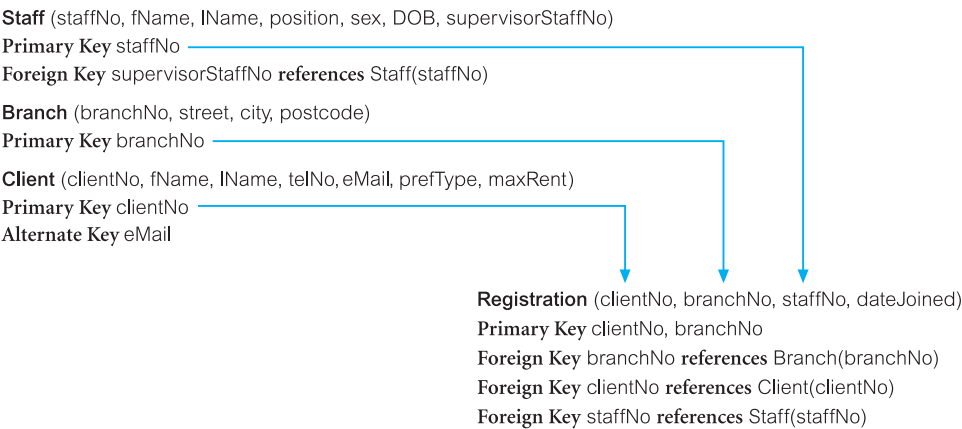
For example, consider the **Client Views PropertyForRent** relationship shown in Figure 17.1. In this example, the **Views** relationship has two attributes called **dateView** and **comments**. To represent this, we create relations for the strong entities **Client** and **PropertyForRent** and we create a relation **Viewing** to represent the relationship **Views**, to give:



(8) Complex relationship types

For each complex relationship, create a relation to represent the relationship and include any attributes that are part of the relationship. We post a copy of the primary key attribute(s) of the entities that participate in the complex relationship into the new relation, to act as foreign keys. Any foreign keys that represent a “many” relationship (for example, 1..*, 0..*) generally will also form the primary key of this new relation, possibly in combination with some of the attributes of the relationship.

For example, the ternary **Registers** relationship in the Branch user views represents the association between the member of staff who registers a new client at a branch, as shown in Figure 13.8. To represent this, we create relations for the strong entities **Branch**, **Staff**, and **Client**, and we create a relation **Registration** to represent the relationship **Registers**, to give:



Staff (staffNo, fName, lName, position, sex, DOB, supervisorStaffNo) Primary Key staffNo Foreign Key supervisorStaffNo references Staff(staffNo)	PrivateOwner (ownerNo, fName, lName, address, telNo) Primary Key ownerNo
BusinessOwner (ownerNo, bName, bType, contactName, address, telNo) Primary Key ownerNo Alternate Key bName Alternate Key telNo	Client (clientNo, fName, lName, telNo, eMail, prefType, maxRent, staffNo) Primary Key clientNo Alternate Key eMail Foreign Key staffNo references Staff(staffNo)
PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo) Primary Key propertyNo Foreign Key ownerNo references PrivateOwner(ownerNo) and BusinessOwner(ownerNo) Foreign Key staffNo references Staff(staffNo)	Viewing (clientNo, propertyNo, dateView, comment) Primary Key clientNo, propertyNo Foreign Key clientNo references Client(clientNo) Foreign Key propertyNo references PropertyForRent(propertyNo)
Lease (leaseNo, paymentMethod, depositPaid, rentStart, rentFinish, clientNo, propertyNo) Primary Key leaseNo Alternate Key propertyNo, rentStart Alternate Key clientNo, rentStart Foreign Key clientNo references Client(clientNo) Foreign Key propertyNo references PropertyForRent(propertyNo) Derived deposit (PropertyForRent.rent*2) Derived duration (rentFinish – rentStart)	

Figure 17.3 Relations for the StaffClient user views of *DreamHome*.

Note that the *Registers* relationship is shown as a binary relationship in Figure 17.1 and this is consistent with its composition in Figure 17.3. The discrepancy between how *Registers* is modeled in the StaffClient (as a binary relationship) and Branch (as a complex [ternary] relationship) user views of *DreamHome* is discussed and resolved in Step 2.6.



(9) Multi-valued attributes

For each multi-valued attribute in an entity, create a new relation to represent the multi-valued attribute, and include the primary key of the entity in the new relation to act as a foreign key. Unless the multi-valued attribute, is itself an alternate key of the entity, the primary key of the new relation is the combination of the multi-valued attribute and the primary key of the entity.

For example, in the Branch user views to represent the situation where a single branch has up to three telephone numbers, the telNo attribute of the Branch entity has been defined as being a multi-valued attribute, as shown in Figure 13.8. To represent this, we create a relation for the Branch entity and we create a new relation called Telephone to represent the multi-valued attribute telNo, to give:



TABLE 17.2 Summary of how to map entities and relationships to relations.

ENTITY/RELATIONSHIP	MAPPING
Strong entity	Create relation that includes all simple attributes.
Weak entity	Create relation that includes all simple attributes (primary key still has to be identified after the relationship with each owner entity has been mapped).
1:* binary relationship	Post primary key of entity on the “one” side to act as foreign key in relation representing entity on the “many” side. Any attributes of relationship are also posted to the “many” side.
1:1 binary relationship: (a) Mandatory participation on both sides (b) Mandatory participation on one side (c) Optional participation on both sides	Combine entities into one relation. Post primary key of entity on the “optional” side to act as foreign key in relation representing entity on the “mandatory” side. Arbitrary without further information.
Superclass/subclass relationship	See Table 17.1.
: binary relationship, complex relationship	Create a relation to represent the relationship and include any attributes of the relationship. Post a copy of the primary keys from each of the owner entities into the new relation to act as foreign keys.
Multi-valued attribute	Create a relation to represent the multi-valued attribute and post a copy of the primary key of the owner entity into the new relation to act as a foreign key.

Table 17.2 summarizes how to map entities and relationships to relations.

Document relations and foreign key attributes



At the end of Step 2.1, document the composition of the relations derived for the logical data model using the DBDL. The relations for the StaffClient user views of *DreamHome* are shown in Figure 17.3.

Now that each relation has its full set of attributes, we are in a position to identify any new primary and/or alternate keys. This is particularly important for weak entities that rely on the posting of the primary key from the parent entity (or entities) to form a primary key of their own. For example, the weak entity Viewing now has a composite primary key, made up of a copy of the primary key of the PropertyForRent entity (propertyNo) and a copy of the primary key of the Client entity (clientNo).

The DBDL syntax can be extended to show integrity constraints on the foreign keys (Step 2.5). The data dictionary should also be updated to reflect any new primary and alternate keys identified in this step. For example, following the posting of primary keys, the Lease relation has gained new alternate keys formed from the attributes (propertyNo, rentStart) and (clientNo, rentStart).

Step 2.2: Validate relations using normalization**Objective**

To validate the relations in the logical data model using normalization.

In the previous step we derived a set of relations to represent the conceptual data model created in Step 1. In this step we validate the groupings of attributes in each relation using the rules of normalization. The purpose of normalization is to ensure that the set of relations has a minimal yet sufficient number of attributes necessary to support the data requirements of the enterprise. Also, the relations should have minimal data redundancy to avoid the problems of update anomalies discussed in Section 14.3. However, some redundancy is essential to allow the joining of related relations.

The use of normalization requires that we first identify the functional dependencies that hold between the attributes in each relation. The characteristics of functional dependencies that are used for normalization were discussed in Section 14.4 and can be identified only if the meaning of each attribute is well understood. The functional dependencies indicate important relationships between the attributes of a relation. It is those functional dependencies and the primary key for each relation that are used in the process of normalization.

The process of normalization takes a relation through a series of steps to check whether the composition of attributes in a relation conforms or otherwise with the rules for a given normal form such as 1NF, 2NF, and 3NF. The rules for each normal form were discussed in detail in Sections 14.6 to 14.8. To avoid the problems associated with data redundancy, it is recommended that each relation be in at least 3NF.

The process of deriving relations from a conceptual data model should produce relations that are already in 3NF. If, however, we identify relations that are not in 3NF, this may indicate that part of the logical data model and/or conceptual data model is incorrect, or that we have introduced an error when deriving the relations from the conceptual data model. If necessary, we must restructure the problem relation(s) and/or data model(s) to ensure a true representation of the data requirements of the enterprise.

It is sometimes argued that a normalized database design does not provide maximum processing efficiency. However, the following points can be argued:

- A normalized design organizes the data according to its functional dependencies. Consequently, the process lies somewhere between conceptual and physical design.
- The logical design may not be the final design. It should represent the database designer's best understanding of the nature and meaning of the data required by the enterprise. If there are specific performance criteria, the physical design may be different. One possibility is that some normalized relations are denormalized, and this approach is discussed in detail in Step 7 of the physical database design methodology (see Chapter 19).
- A normalized design is robust and free of the update anomalies discussed in Section 14.3.
- Modern computers are much more powerful than those that were available a few years ago. It is sometimes reasonable to implement a design that gains ease of use at the expense of additional processing.

- To use normalization, a database designer must understand completely each attribute that is to be represented in the database. This benefit may be the most important.
- Normalization produces a flexible database design that can be extended easily.

Step 2.3: Validate relations against user transactions

Objective

To ensure that the relations in the logical data model support the required transactions.

The objective of this step is to validate the logical data model to ensure that the model supports the required transactions, as detailed in the users' requirements specification. This type of check was carried out in Step 1.8 to ensure that the conceptual data model supported the required transactions. In this step, we check whether the relations created in the previous step also support these transactions, and thereby ensure that no error has been introduced while creating relations.

Using the relations, the primary key/foreign key links shown in the relations, the ER diagram, and the data dictionary, we attempt to perform the operations manually. If we can resolve all transactions in this way, we have validated the logical data model against the transactions. However, if we are unable to perform a transaction manually, there must be a problem with the data model that must be resolved. In this case, it is likely that an error has been introduced while creating the relations and we should go back and check the areas of the data model that the transaction is accessing to identify and resolve the problem.

Step 2.4: Check integrity constraints

Objective

To check whether integrity constraints are represented in the logical data model.

Integrity constraints are the constraints that we wish to impose in order to protect the database from becoming incomplete, inaccurate, or inconsistent. Although DBMS controls for integrity constraints may or may not exist, this is not the question here. At this stage we are concerned only with high-level design, that is, specifying *what* integrity constraints are required, irrespective of *how* this might be achieved. A logical data model that includes all important integrity constraints is a “true” representation of the data requirements for the enterprise. We consider the following types of integrity constraint:

- required data;
- attribute domain constraints;
- multiplicity;
- entity integrity;
- referential integrity;
- general constraints.

Required data

Some attributes must always contain a valid value; in other words, they are not allowed to hold nulls. For example, every member of staff must have an associated

job position (such as Supervisor or Assistant). These constraints should have been identified when we documented the attributes in the data dictionary (Step 1.3).

Attribute domain constraints

Every attribute has a domain, that is, a set of values that are legal. For example, the sex of a member of staff is either “M” or “F,” so the domain of the sex attribute is a single character string consisting of “M” or “F.” These constraints should have been identified when we chose the attribute domains for the data model (Step 1.4).

Multiplicity

Multiplicity represents the constraints that are placed on relationships between data in the database. Examples of such constraints include the requirements that a branch has many staff and a member of staff works at a single branch. Ensuring that all appropriate integrity constraints are identified and represented is an important part of modeling the data requirements of an enterprise. In Step 1.2 we defined the relationships between entities, and all integrity constraints that can be represented in this way were defined and documented in this step.

Entity integrity

The primary key of an entity cannot hold nulls. For example, each tuple of the Staff relation must have a value for the primary key attribute, *staffNo*. These constraints should have been considered when we identified the primary keys for each entity type (Step 1.5).

Referential integrity

A foreign key links each tuple in the child relation to the tuple in the parent relation containing the matching candidate key value. Referential integrity means that if the foreign key contains a value, that value must refer to an existing tuple in the parent relation. For example, consider the Staff *Manages* PropertyForRent relationship. The *staffNo* attribute in the PropertyForRent relation links the property for rent to the tuple in the Staff relation containing the member of staff who manages that property. If *staffNo* is not null, it must contain a valid value that exists in the *staffNo* attribute of the Staff relation, or the property will be assigned to a nonexistent member of staff.

There are two issues regarding foreign keys that must be addressed. The first considers whether nulls are allowed for the foreign key. For example, can we store the details of a property for rent without having a member of staff specified to manage it—that is, can we specify a null *staffNo*? The issue is not whether the staff number exists, but whether a staff number must be specified. In general, if the participation of the child relation in the relationship is:

- mandatory, then nulls are not allowed;
- optional, then nulls are allowed.

The second issue we must address is how to ensure referential integrity. To do this, we specify **existence constraints** that define conditions under which a candidate key or foreign key may be inserted, updated, or deleted. For the 1:* Staff *Manages* PropertyForRent relationship, consider the following cases.

Case 1: Insert tuple into child relation (PropertyForRent) To ensure referential integrity, check that the foreign key attribute, `staffNo`, of the new `PropertyForRent` tuple is set to null or to a value of an existing `Staff` tuple.

Case 2: Delete tuple from child relation (PropertyForRent) If a tuple of a child relation is deleted referential integrity is unaffected.

Case 3: Update foreign key of child tuple (PropertyForRent) This case is similar to Case 1. To ensure referential integrity, check whether the `staffNo` of the updated `PropertyForRent` tuple is set to null or to a value of an existing `Staff` tuple.

Case 4: Insert tuple into parent relation (Staff) Inserting a tuple into the parent relation (`Staff`) does not affect referential integrity; it simply becomes a parent without any children: in other words, a member of staff without properties to manage.

Case 5: Delete tuple from parent relation (Staff) If a tuple of a parent relation is deleted, referential integrity is lost if there exists a child tuple referencing the deleted parent tuple; in other words, if the deleted member of staff currently manages one or more properties. There are several strategies we can consider:

- **NO ACTION**—Prevent a deletion from the parent relation if there are any referenced child tuples. In our example, “You cannot delete a member of staff if he or she currently manages any properties.”
- **CASCADE**—When the parent tuple is deleted, automatically delete any referenced child tuples. If any deleted child tuple acts as the parent in another relationship, then the delete operation should be applied to the tuples in this child relation, and so on in a cascading manner. In other words, deletions from the parent relation cascade to the child relation. In our example, “Deleting a member of staff automatically deletes all properties he or she manages.” Clearly, in this situation, this strategy would not be wise. If we have used the enhanced modeling technique of *composition* to relate the parent and child entities, **CASCADE** should be specified (see Section 13.3).
- **SET NULL**—When a parent tuple is deleted, the foreign key values in all corresponding child tuples are automatically set to null. In our example, “If a member of staff is deleted, indicate that the current assignment of those properties previously managed by that employee is unknown.” We can consider this strategy only if the attributes constituting the foreign key accept nulls.
- **SET DEFAULT**—When a parent tuple is deleted, the foreign key values in all corresponding child tuples should automatically be set to their default values. In our example, “If a member of staff is deleted, indicate that the current assignment of some properties is being handled by another (default) member of staff such as the Manager.” We can consider this strategy only if the attributes constituting the foreign key have default values defined.
- **NO CHECK**—When a parent tuple is deleted, do nothing to ensure that referential integrity is maintained.

Case 6: Update primary key of parent tuple (Staff) If the primary key value of a parent relation tuple is updated, referential integrity is lost if there exists a child tuple referencing the old primary key value; that is, if the updated member of staff

```

Staff (staffNo, fName, lName, position, sex, DOB, supervisorStaffNo)
Primary Key staffNo
Foreign Key supervisorStaffNo references Staff(staffNo) ON UPDATE CASCADE ON DELETE SET NULL

Client (clientNo, fName, lName, telNo, eMail, prefType, maxRent, staffNo)
Primary Key clientNo
Alternate Key eMail
Foreign Key staffNo references Staff(staffNo) ON UPDATE CASCADE ON DELETE NO ACTION

PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo)
Primary Key propertyNo
Foreign Key ownerNo references PrivateOwner(ownerNo) and BusinessOwner(ownerNo)
ON UPDATE CASCADE ON DELETE NO ACTION

Foreign Key staffNo references Staff(staffNo) ON UPDATE CASCADE ON DELETE SET NULL

Viewing (clientNo, propertyNo, dateView, comment)
Primary Key clientNo, propertyNo
Foreign Key clientNo references Client(clientNo) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key propertyNo references PropertyForRent(propertyNo)
ON UPDATE CASCADE ON DELETE CASCADE

Lease (leaseNo, paymentMethod, depositPaid, rentStart, rentFinish, clientNo, propertyNo)
Primary Key leaseNo
Alternate Key propertyNo, rentStart
Alternate Key clientNo, rentStart
Foreign Key clientNo references Client(clientNo) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key propertyNo references PropertyForRent(propertyNo)
ON UPDATE CASCADE ON DELETE NO ACTION

```

Figure 17.4 Referential integrity constraints for the relations in the StaffClient user views of *DreamHome*.

currently manages one or more properties. To ensure referential integrity, the strategies described earlier can be used. In the case of CASCADE, the updates to the primary key of the parent tuple are reflected in any referencing child tuples, and if a referencing child tuple is itself a primary key of a parent tuple, this update will also cascade to its referencing child tuples, and so on in a cascading manner. It is normal for updates to be specified as CASCADE.

The referential integrity constraints for the relations that have been created for the StaffClient user views of *DreamHome* are shown in Figure 17.4.



General constraints

Finally, we consider constraints known as general constraints. Updates to entities may be controlled by constraints governing the “real-world” transactions that are represented by the updates. For example, *DreamHome* has a rule that prevents a member of staff from managing more than 100 properties at the same time.

Document all integrity constraints

Document all integrity constraints in the data dictionary for consideration during physical design.

Step 2.5: Review logical data model with user**Objective**

To review the logical data model with the users to ensure that they consider the model to be a true representation of the data requirements of the enterprise.

The logical data model should now be complete and fully documented. However, to confirm that this is the case, users are requested to review the logical data model to ensure that they consider the model to be a true representation of the data requirements of the enterprise. If the users are dissatisfied with the model, then some repetition of earlier steps in the methodology may be required.

If the users are satisfied with the model, then the next step taken depends on the number of user views associated with the database and, more importantly, how they are being managed. If the database system has a single user view or multiple user views that are being managed using the centralization approach (see Section 10.5), then we proceed directly to the final step of Step 2: Step 2.7. If the database has multiple user views that are being managed using the view integration approach (see Section 10.5), then we proceed to Step 2.6. The view integration approach results in the creation of several logical data models, each of which represents one or more, but not all, user views of a database. The purpose of Step 2.6 is to merge these data models to create a single logical data model that represents all user views of a database. However, before we consider this step, we discuss briefly the relationship between logical data models and data flow diagrams.

Relationship between logical data model and data flow diagrams

A logical data model reflects the structure of stored data for an enterprise. A Data Flow Diagram (DFD) shows data moving about the enterprise and being stored in datastores. All attributes should appear within an entity type if they are held within the enterprise, and will probably be seen flowing around the enterprise as a data flow. When these two techniques are being used to model the users' requirements specification, we can use each one to check the consistency and completeness of the other. The rules that control the relationship between the two techniques are:

- each datastore should represent a whole number of entity types;
- attributes on data flows should belong to entity types.

Step 2.6: Merge logical data models into global model (optional step)**Objective**

To merge local logical data models into a single global logical data model that represents all user views of a database.

This step is necessary only for the design of a database with multiple user views that are being managed using the view integration approach. To facilitate the description of the merging process, we use the terms “local logical data model” and “global logical data model.” A **local logical data model** represents one or more but not all user views of a database whereas **global logical data model** represents *all* user views of a database. In this step we merge two or more local logical data models into a single global logical data model.

The source of information for this step is the local data models created through Step 1 and Steps 2.1 to 2.5 of the methodology. Although each local logical data model should be correct, comprehensive, and unambiguous, each model is a representation only of one or more but not all user views of a database. In other words, each model represents only part of the complete database. This may mean that there are inconsistencies as well as overlaps when we look at the complete set of user views. Thus, when we merge the local logical data models into a single global model, we must endeavor to resolve conflicts between the user views and any overlaps that exist.

Therefore, on completion of the merging process, the resulting global logical data model is subjected to validations similar to those performed on the local data models. The validations are particularly necessary and should be focused on areas of the model that are subjected to most change during the merging process.

The activities in this step include:

Step 2.6.1 Merge local logical data models into global logical data model

Step 2.6.2 Validate global logical data model

Step 2.6.3 Review global logical data model with users

We demonstrate this step using the local logical data model developed previously for the StaffClient user views of the *DreamHome* case study and using the model developed in Chapters 12 and 13 for the Branch user views of *DreamHome*. Figure 17.5 shows the relations created from the ER model for the Branch user views given in Figure 13.8. We leave it as an exercise for the reader to show that this mapping is correct (see Exercise 17.6).



Step 2.6.1: Merge local logical data models into global logical data model

Objective

To merge local logical data models into a single global logical data model.

Up to this point, for each local logical data model we have produced an ER diagram, a relational schema, a data dictionary, and supporting documentation that describes the constraints on the data. In this step, we use these components to identify the similarities and differences between the models and thereby help merge the models together.

For a simple database system with a small number of user views, each with a small number of entity and relationship types, it is a relatively easy task to compare the local models, merge them together, and resolve any differences that exist. However, in a large system, a more systematic approach must be taken. We present one approach that may be used to merge the local models together and resolve any inconsistencies found. For a discussion on other approaches, the interested reader is referred to the papers by Batini and Lanzerini (1986), Biskup and Convent (1986), Spaccapietra *et al.* (1992) and Bouguettaya *et al.* (1998).

Some typical tasks in this approach are:

- (1) Review the names and contents of entities/relations and their candidate keys.
- (2) Review the names and contents of relationships/foreign keys.
- (3) Merge entities/relations from the local data models.

Branch (branchNo, street, city, postcode, mgrStaffNo) Primary Key branchNo Alternate Key postcode Foreign Key mgrStaffNo references Manager(staffNo)	Telephone (telNo, branchNo) Primary Key telNo Foreign Key branchNo references Branch(branchNo)
Staff (staffNo, name, position, salary, supervisorStaffNo, branchNo) Primary Key staffNo Foreign Key supervisorStaffNo references Staff(staffNo) Foreign Key branchNo references Branch(branchNo)	Manager (staffNo, mgrStartDate, bonus) Primary Key staffNo Foreign Key staffNo references Staff(staffNo)
PrivateOwner (ownerNo, name, address, telNo) Primary Key ownerNo	BusinessOwner (bName, bType, contactName, address, telNo) Primary Key bName Alternate Key telNo
PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo, bName, branchNo) Primary Key propertyNo Foreign Key ownerNo references PrivateOwner(ownerNo) Foreign Key bName references BusinessOwner(bName) Foreign Key staffNo references Staff(staffNo) Foreign Key branchNo references Branch(branchNo)	Client (clientNo, name, telNo, eMail, prefType, maxRent) Primary Key clientNo Alternate Key eMail
Lease (leaseNo, paymentMethod, depositPaid, rentStart, rentFinish, clientNo, propertyNo) Primary Key leaseNo Alternate Key propertyNo, rentStart Alternate Key clientNo, rentStart Foreign Key clientNo references Client(clientNo) Foreign Key propertyNo references PropertyForRent(propertyNo) Derived deposit (PropertyForRent.rent*2) Derived duration (rentFinish – rentStart)	Registration (clientNo, branchNo, staffNo, dateJoined) Primary Key clientNo, branchNo Foreign Key clientNo references Client(clientNo) Foreign Key branchNo references Branch(branchNo) Foreign Key staffNo references Staff(staffNo)
Advert (propertyNo, newspaperName, dateAdvert, cost) Primary Key propertyNo, newspaperName, dateAdvert Foreign Key propertyNo references PropertyForRent(propertyNo) Foreign Key newspaperName references Newspaper(newspaperName)	Newspaper (newspaperName, address, telNo, contactName) Primary Key newspaperName Alternate Key telNo

Figure 17.5 Relations for the Branch user views of *DreamHome*.

- (4) Include (without merging) entities/relations unique to each local data model.
- (5) Merge relationships/foreign keys from the local data models.
- (6) Include (without merging) relationships/foreign keys unique to each local data model.
- (7) Check for missing entities/relations and relationships/foreign keys.
- (8) Check foreign keys.
- (9) Check integrity constraints.
- (10) Draw the global ER/relation diagram.
- (11) Update the documentation.

In some of these tasks, we have used the term “entities/relations” and “relationships/foreign keys.” This allows the designer to choose whether to examine the ER

models or the relations that have been derived from the ER models in conjunction with their supporting documentation, or even to use a combination of both approaches. It may be easier to base the examination on the composition of relations, as this removes many syntactic and semantic differences that may exist between different ER models possibly produced by different designers.

Perhaps the easiest way to merge several local data models together is first to merge two of the data models to produce a new model, and then successively to merge the remaining local data models until all the local models are represented in the final global data model. This may prove a simpler approach than trying to merge all the local data models at the same time.

(1) Review the names and contents of entities/relations and their candidate keys

It may be worthwhile to review the names and descriptions of entities/relations that appear in the local data models by inspecting the data dictionary. Problems can arise when two or more entities/relations:

- have the same name but are, in fact, different (homonyms);
- are the same but have different names (synonyms).

It may be necessary to compare the data content of each entity/relation to resolve these problems. In particular, use the candidate keys to help identify equivalent entities/relations that may be named differently across user views. A comparison of the relations in the Branch and StaffClient user views of *DreamHome* is shown in Table 17.3. The relations that are common to each user views are highlighted.



(2) Review the names and contents of relationships/foreign keys

This activity is the same as described for entities/relations. A comparison of the foreign keys in the Branch and StaffClient user views of *DreamHome* is shown in Table 17.4. The foreign keys that are common to each user view are highlighted. Note, in particular, that of the relations that are common to both user views, the Staff and PropertyForRent relations have an extra foreign key, branchNo.



This initial comparison of the relationship names/foreign keys in each view again gives some indication of the extent to which the user views overlap. However, it is important to recognize that we should not rely too heavily on the fact that entities or relationships with the same name play the same role in both user views. However, comparing the names of entities/relations and relationships/foreign keys is a good starting point when searching for overlap between the user views, as long as we are aware of the pitfalls.

We must be careful of entities or relationships that have the same name but in fact represent different concepts (also called homonyms). An example of this occurrence is the Staff *Manages* PropertyForRent (StaffClient user views) and Manager *Manages* Branch (Branch user views). Obviously, the *Manages* relationship in this case means something different in each user view.

We must therefore ensure that entities or relationships that have the same name represent the same concept in the “real world,” and that the names that differ in each user view represent different concepts. To achieve this, we compare the attributes (and, in particular, the keys) associated with each entity and also their associated

TABLE 17.3 A comparison of the names of entities/relations and their candidate keys in the Branch and StaffClient user views.

BRANCH USER VIEWS		STAFFCLIENT USER VIEWS	
ENTITY/RELATION	CANDIDATE KEYS	ENTITY/RELATION	CANDIDATE KEYS
Branch	branchNo postcode		
Telephone	telNo		
Staff	staffNo	Staff	staffNo
Manager	staffNo		
PrivateOwner	ownerNo	PrivateOwner	ownerNo
BusinessOwner	bName telNo	BusinessOwner	bName telNo ownerNo
Client	clientNo eMail	Client	clientNo eMail
PropertyForRent	propertyNo	PropertyForRent Viewing	propertyNo clientNo, propertyNo
Lease	leaseNo propertyNo, rentStart clientNo, rentStart	Lease	leaseNo propertyNo, rentStart clientNo, rentStart
Registration	(clientNo, branchNo)		
Newspaper	newspaperName telNo		
Advert	(propertyNo, newspaperName, dateAdvert)		

relationships with other entities. We should also be aware that entities or relationships in one user view may be represented simply as attributes in another user view. For example, consider the scenario where the Branch entity has an attribute called manager Name in one user view, which is represented as an entity called Manager in another user view.

(3) Merge entities/relations from the local data models

Examine the name and content of each entity/relation in the models to be merged to determine whether entities/relations represent the same thing and can therefore be merged. Typical activities involved in this task include:

- merging entities/relations with the same name and the same primary key;
- merging entities/relations with the same name but different primary keys;
- merging entities/relations with different names using the same or different primary keys.

TABLE 17.4 A comparison of the foreign keys in the Branch and StaffClient user views.

BRANCH USER VIEWS			STAFFCLIENT USER VIEWS		
CHILD RELATION	FOREIGN KEYS	PARENT RELATION	CHILD RELATION	FOREIGN KEYS	PARENT RELATION
Branch	mgrStaffNo →	Manager(staffNo)			
Telephone ^a	branchNo →	Branch(branchNo)			
Staff	supervisorStaffNo → branchNo →	Staff(staffNo) Branch(branchNo)	Staff	supervisorStaffNo →	Staff(staffNo)
Manager	staffNo →	Staff(staffNo)			
PrivateOwner			PrivateOwner		
Business Owner			Business Owner		
Client			Client	StaffNo →	Staff(staffNo)
PropertyForRent	ownerNo → bName → staffNo → branchNo →	PrivateOwner(ownerNo) BusinessOwner(ownerNo) Staff(staffNo) Branch(branchNo)	PropertyForRent	ownerNo → ownerNo → staffNo →	PrivateOwner(ownerNo) PrivateOwner(ownerNo) Staff(staffNo)
			Viewing	clientNo → propertyNo →	Client(clientNo) PropertyForRent(propertyNo)
Lease	clientNo → propertyNo →	Client(clientNo) PropertyForRent(propertyNo)	Lease	clientNo → propertyNo →	Client(clientNo) PropertyForRent(propertyNo)
Registration ^b	clientNo → branchNo → staffNo →	Client(clientNo) Branch(branchNo) Staff(staffNo)			
Newspaper					
Advert ^c	propertyNo → newspaperName →	PropertyForRent(propertyNo) Newspaper(newspaperName)			

^aThe Telephone relation is created from the multi-valued attribute telNo.
^bThe Registration relation is created from the ternary relationship Registers.
^cThe Advert relation is created from the many-to-many (*,*) relationship Advertises.

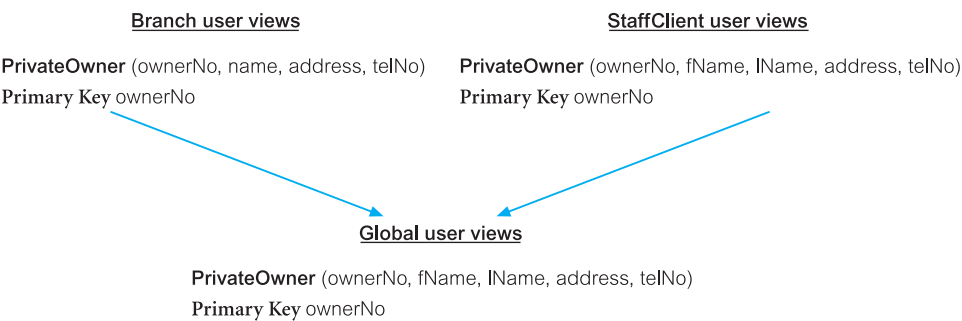


Figure 17.6 Merging the PrivateOwner relations from the Branch and StaffClient user views.

Merging entities/relations with the same name and the same primary key Generally, entities/relations with the same primary key represent the same “real-world” object and should be merged. The merged entity/relation includes the attributes from the original entities/relations with duplicates removed. For example, Figure 17.6 lists the attributes associated with the relation PrivateOwner defined in the Branch and StaffClient user views. The primary key of both relations is ownerNo. We merge these two relations together by combining their attributes, so that the merged PrivateOwner relation now has all the original attributes associated with both PrivateOwner relations. Note that there is conflict between the user views on how we should represent the name of an owner. In this situation, we should (if possible) consult the users of each user view to determine the final representation. Note that in this example, we use the decomposed version of the owner’s name, represented by the fName and lName attributes, in the merged global view.

In a similar way, from Table 17.2 the Staff, Client, PropertyForRent, and Lease relations have the same primary keys in both user views and the relations can be merged as discussed earlier.

Merging entities/relations with the same name but different primary keys In some situations, we may find two entities/relations with the same name and similar candidate keys, but with different primary keys. In this case, the entities/relations should be merged together as described previously. However, it is necessary to choose one key to be the primary key, the others becoming alternate keys. For example, Figure 17.7 lists the attributes associated with the two relations BusinessOwner defined in the two user views. The primary key of the BusinessOwner relation in the Branch user views is bName and the primary key of the BusinessOwner relation in the StaffClient user views is ownerNo. However, the alternate key for BusinessOwner in the StaffClient user views is bName. Although the primary keys are different, the primary key of BusinessOwner in the Branch user views is the alternate key of BusinessOwner in the StaffClient user views. We merge these two relations together as shown in Figure 17.7 and include bName as an alternate key.

Merging entities/relations with different names using the same or different primary keys In some cases, we may identify entities/relations that have different

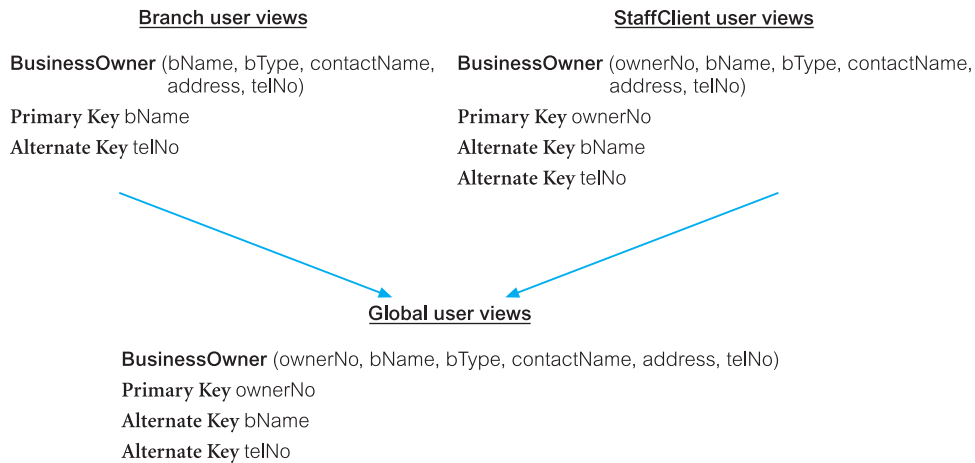


Figure 17.7 Merging the BusinessOwner relations with different primary keys.

names but appear to have the same purpose. These equivalent entities/relations may be recognized simply by:

- their name, which indicates their similar purpose;
- their content and, in particular, their primary key;
- their association with particular relationships.

An obvious example of this occurrence would be entities called *Staff* and *Employee*, which if found to be equivalent should be merged.

(4) Include (without merging) entities/relations unique to each local data model

The previous tasks should identify all entities/relations that are the same. All remaining entities/relations are included in the global model without change. From Table 17.2, the *Branch*, *Telephone*, *Manager*, *Registration*, *Newspaper*, and *Advert* relations are unique to the Branch user views, and the *Viewing* relation is unique to the StaffClient user views.

(5) Merge relationships/foreign keys from the local data models

In this step we examine the name and purpose of each relationship/foreign key in the data models. Before merging relationships/foreign keys, it is important to resolve any conflicts between the relationships, such as differences in multiplicity constraints. The activities in this step include:

- merging relationships/foreign keys with the same name and the same purpose;
- merging relationships/foreign keys with different names but the same purpose.

Using Table 17.3 and the data dictionary, we can identify foreign keys with the same name and the same purpose which can be merged into the global model.

Note that the *Registers* relationship in the two user views essentially represents the same ‘event’: in the StaffClient user views, the *Registers* relationship models a member of staff registering a client; and this is represented using *staffNo* as a foreign Key in Client: in the Branch user views, the situation is slightly more complex, due to the additional modeling of branches, and this requires a new relation called *Registration* to model a member of staff registering a client at a branch. In this case, we ignore the *Registers* relationship in the StaffClient user views and include the equivalent relationships/foreign keys from the Branch user views in the next step.

(6) Include (without merging) relationships/foreign keys unique to each local data model

Again, the previous task should identify relationships/foreign keys that are the same (by definition, they must be between the same entities/relations, which would have been merged together earlier). All remaining relationships/foreign keys are included in the global model without change.

(7) Check for missing entities/relations and relationships/foreign keys

Perhaps one of the most difficult tasks in producing the global model is identifying missing entities/relations and relationships/foreign keys between different local data models. If a corporate data model exists for the enterprise, this may reveal entities and relationships that do not appear in any local data model. Alternatively, as a preventative measure, when interviewing the users of a specific user views, ask them to pay particular attention to the entities and relationships that exist in other user views. Otherwise, examine the attributes of each entity/relation and look for references to entities/relations in other local data models. We may find that we have an attribute associated with an entity/relation in one local data model that corresponds to a primary key, alternate key, or even a non-key attribute of an entity/relation in another local data model.

(8) Check foreign keys

During this step, entities/relations and relationships/foreign keys may have been merged, primary keys changed, and new relationships identified. Confirm that the foreign keys in child relations are still correct, and make any necessary modifications. The relations that represent the global logical data model for *DreamHome* are shown in Figure 17.8.



(9) Check integrity constraints

Confirm that the integrity constraints for the global logical data model do not conflict with those originally specified for each user view. For example, if any new relationships have been identified and new foreign keys have been created, ensure that appropriate referential integrity constraints are specified. Any conflicts must be resolved in consultation with the users.

Branch (branchNo, street, city, postcode, mgrStaffNo) Primary Key branchNo Alternate Key postcode Foreign Key mgrStaffNo references Manager(staffNo)	Telephone (telNo, branchNo) Primary Key telNo Foreign Key branchNo references Branch(branchNo)
Staff (staffNo, fName, lName, position, sex, DOB, salary, supervisorStaffNo, branchNo) Primary Key staffNo Foreign Key supervisorStaffNo references Staff(staffNo) Foreign Key branchNo references Branch(branchNo)	Manager (staffNo, mgrStartDate, bonus) Primary Key staffNo Foreign Key staffNo references Staff(staffNo)
PrivateOwner (ownerNo, fName, lName, address, telNo) Primary Key ownerNo	BusinessOwner (ownerNo, bName, bType, contactName, address, telNo) Primary Key ownerNo Alternate Key bName Alternate Key telNo
PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo) Primary Key propertyNo Foreign Key ownerNo references PrivateOwner(ownerNo) and BusinessOwner(ownerNo) Foreign Key staffNo references Staff(staffNo) Foreign Key branchNo references Branch(branchNo)	Viewing (clientNo, propertyNo, dateView, comment) Primary Key clientNo, propertyNo Foreign Key clientNo references Client(clientNo) Foreign Key propertyNo references PropertyForRent(propertyNo)
Client (clientNo, fName, lName, telNo, eMail, prefType, maxRent) Primary Key clientNo Alternate Key eMail	Registration (clientNo, branchNo, staffNo, dateJoined) Primary Key clientNo, branchNo Foreign Key clientNo references Client(clientNo) Foreign Key branchNo references Branch(branchNo) Foreign Key staffNo references Staff(staffNo)
Lease (leaseNo, paymentMethod, depositPaid, rentStart, rentFinish, clientNo, propertyNo) Primary Key leaseNo Alternate Key propertyNo, rentStart Alternate Key clientNo, rentStart Foreign Key clientNo references Client(clientNo) Foreign Key propertyNo references PropertyForRent(propertyNo) Derived deposit (PropertyForRent.rent*2) Derived duration (rentFinish – rentStart)	Newspaper (newspaperName, address, telNo, contactName) Primary Key newspaperName Alternate Key telNo
Advert (propertyNo, newspaperName, dateAdvert, cost) Primary Key propertyNo, newspaperName, dateAdvert Foreign Key propertyNo references PropertyForRent(propertyNo) Foreign Key newspaperName references Newspaper(newspaperName)	

Figure 17.8 Relations that represent the global logical data model for *DreamHome*.

(10) Draw the global ER/relation diagram

We now draw a final diagram that represents all the merged local logical data models. If relations have been used as the basis for merging, we call the resulting diagram a **global relation diagram**, which shows primary keys and foreign keys. If local ER diagrams have been used, the resulting diagram is simply a global ER diagram. The global relation diagram for *DreamHome* is shown in Figure 17.9.



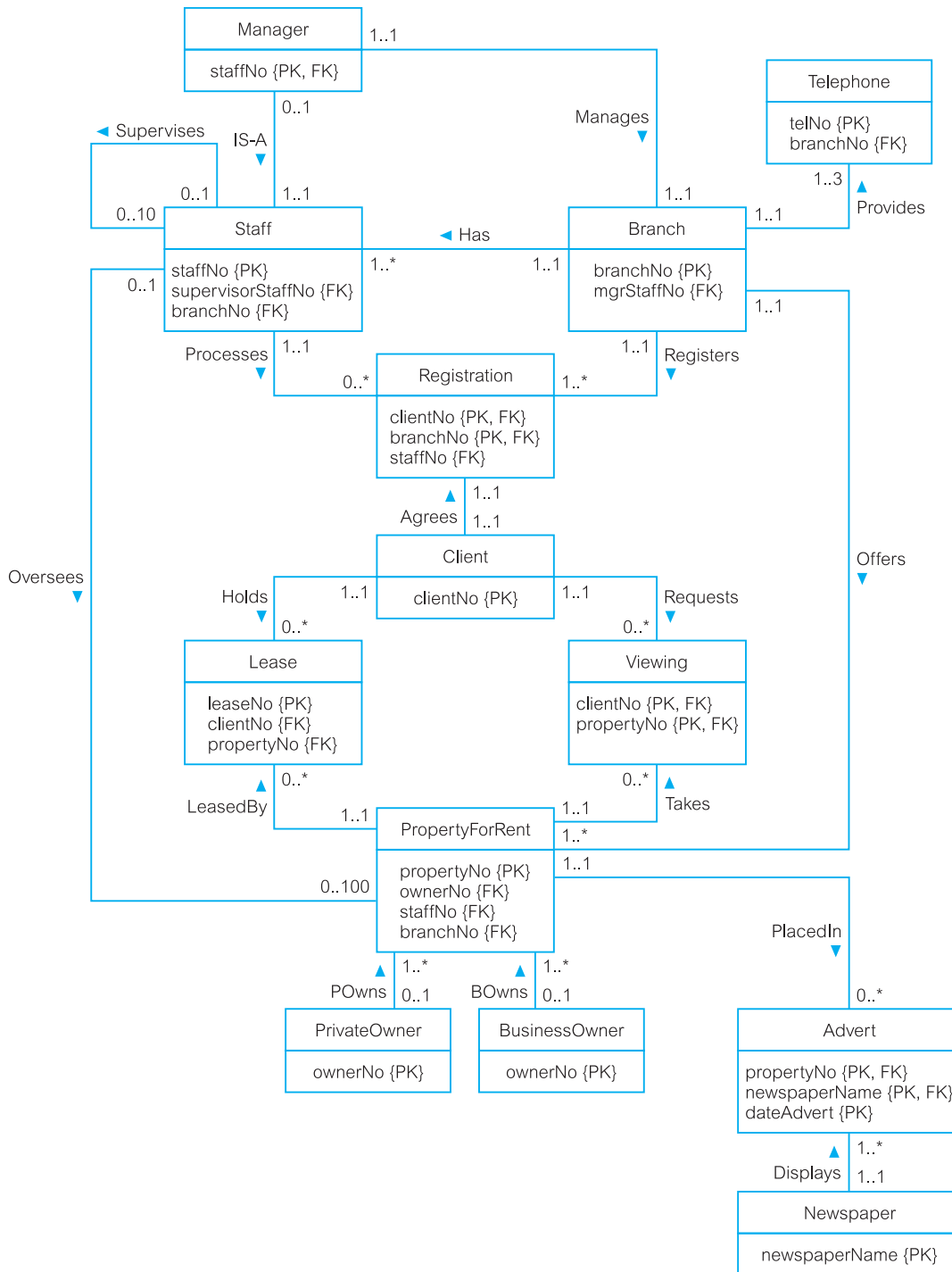


Figure 17.9 Global relation diagram for *DreamHome*.

(11) Update the documentation

Update the documentation to reflect any changes made during the development of the global data model. It is very important that the documentation is up to date and reflects the current data model. If changes are made to the model subsequently, either during database implementation or during maintenance, then the documentation should be updated at the same time. Out-of-date information will cause considerable confusion at a later time.

Step 2.6.2: Validate global logical data model

Objective

To validate the relations created from the global logical data model using the technique of normalization and to ensure that they support the required transactions, if necessary.

This step is equivalent to Steps 2.2 and 2.3, in which we validated each local logical data model. However, it is necessary to check only those areas of the model that resulted in any change during the merging process. In a large system, this will significantly reduce the amount of rechecking that needs to be performed.

Step 2.6.3: Review global logical data model with users

Objective

To review the global logical data model with the users to ensure that they consider the model to be a true representation of the data requirements of an enterprise.

The global logical data model for the enterprise should now be complete and accurate. The model and the documentation that describes the model should be reviewed with the users to ensure that it is a true representation of the enterprise.

To facilitate the description of the tasks associated with Step 2.6, it is necessary to use the terms “*local* logical data model” and “*global* logical data model.” However, at the end of this step when the local data models have been merged into a *single* global data model, the distinction between the data models that refer to some or all user views of a database is no longer necessary. Therefore, after completing this step we refer to the single global data model using the simpler term “logical data model” for the remaining steps of the methodology.

Step 2.7: Check for future growth

Objective

To determine whether there are any significant changes likely in the foreseeable future and to assess whether the logical data model can accommodate these changes.

Logical database design concludes by considering whether the logical data model (which may or may not have been developed using Step 2.6) is capable of being extended to support possible future developments. If the model can sustain current requirements only, then the life of the model may be relatively short and significant reworking may be necessary to accommodate new requirements. It is important to develop a model that is *extensible* and has the ability to evolve to