Thomas Connolly, Carolyn Begg (2015): Database Systems: A Practical Approach to Design, Implementation, and Management, Global Edition

Tittel	Database Systems: A Practical Approach to Design, Implementation, and Management, Global Edition
Forfatter	Thomas Connolly, Carolyn Begg
Utgiver	Pearson Education Limited
Årstall	2015
Utgave	6
ISBN	9781292061184
Sider	501-584

CHAPTER

16 Methodology—Conceptual Database Design

Chapter Objectives

In this chapter you will learn:

- The purpose of a design methodology.
- The three main phases of database design: conceptual, logical, and physical design.
- How to decompose the scope of the design into specific views of the enterprise.
- How to use ER modeling to build a local conceptual data model based on the information given in a view of the enterprise.
- How to validate the resultant conceptual data model to ensure that it is a true and accurate representation of a view of the enterprise.
- How to document the process of conceptual database design.
- End-users play an integral role throughout the process of conceptual database design.

In Chapter 10 we described the main stages of the database system development lifecycle, one of which is **database design**. This stage starts only after a complete analysis of the enterprise's requirements has been undertaken.

In this chapter, and Chapters 17–19, we describe a methodology for the database design stage of the database system development lifecycle for relational databases. The methodology is presented as a step-by-step guide to the three main phases of database design, namely: conceptual, logical, and physical design (see Figure 10.1). The main aim of each phase is as follows:

- **Conceptual database design**—to build the conceptual representation of the database, which includes identification of the important entities, relationships, and attributes.
- **Logical database design**—to translate the conceptual representation to the logical structure of the database, which includes designing the relations.
- **Physical database design**—to decide how the logical structure is to be physically implemented (as base relations) in the target DBMS.

Structure of this Chapter In Section 16.1 we define what a database design methodology is and review the three phases of database design. In Section 16.2 we provide an overview of the methodology and briefly describe the main activities associated with each design phase. In Section 16.3 we focus on the methodology for conceptual database design and present a detailed description of the steps required to build a conceptual data model. We use the ER modeling technique described in Chapters 12 and 13 to create the *conceptual data model*. The conceptual data model described in this chapter is the starting point for the next phase of database design described in the following chapter.

In Chapter 17 we focus on the methodology for logical database design for the relational model and present a detailed description of the steps required to convert a conceptual data model into a *logical data model*. This chapter also includes an optional step that describes how to merge two or more logical data models into a single logical data model, for those using the view integration approach (see Section 10.5) to manage the design of a database with multiple user views. The logical data model described in Chapter 17 is the starting point for the final phase of database design described in the following two chapters.

In Chapters 18 and 19 we complete the database design methodology by presenting a detailed description of the steps associated with the production of the physical database design for relational DBMSs. This part of the methodology illustrates that the development of the logical data model alone is insufficient to guarantee the optimum implementation of a database system. For example, we may have to consider modifying the logical model to achieve acceptable levels of performance.

Appendix D presents a summary of the database design methodology for those readers who are already familiar with database design and simply require an overview of the main steps. 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 added only to avoid ambiguity. In this chapter we mostly use examples from the StaffClient user views of the *DreamHome* case study documented in Section 11.4 and Appendix A.

16.1 Introduction to the Database Design Methodology

Before presenting the methodology, we discuss what a design methodology represents and describe the three phases of database design. Finally, we present guidelines for achieving success in database design.

16.1.1 What Is a Design Methodology?

Design methodology A structured approach that uses procedures, techniques, tools, and documentation aids to support and facilitate the process of design.

A design methodology consists of phases each containing a number of steps that guide the designer in the techniques appropriate at each stage of the project. A design methodology also helps the designer to plan, manage, control, and evaluate database development projects. Furthermore, it is a structured approach for analyzing and modeling a set of requirements for a database in a standardized and organized manner.

16.1.2 Conceptual, Logical, and Physical Database Design

In presenting this database design methodology, the design process is divided into three main phases: conceptual, logical, and physical database design.

Conceptual database design — The process of constructing a model of the data used in an enterprise, independent of *all* physical considerations.

The conceptual database design phase begins with the creation of a conceptual data model of the enterprise that is entirely independent of implementation details such as the target DBMS, application programs, programming languages, hardware platform, performance issues, or any other physical considerations.

Logical database design — The process of constructing a model of the data used in an enterprise based on a specific data model, but independent of a particular DBMS and other physical considerations.

The logical database design phase maps the conceptual data model on to a logical model, which is influenced by the data model for the target database (for example, the relational model). The logical data model is a source of information for the physical design phase, providing the physical database designer with a vehicle for making trade-offs that are very important to the design of an efficient database.

Physical database design — The process of producing a description of the implementation of the database on secondary storage; it describes the base relations, file organizations, and indexes used to achieve efficient access to the data, and any associated integrity constraints and security measures.

The physical database design phase allows the designer to make decisions on how the database is to be implemented. Therefore, physical design is tailored to a specific DBMS. There is feedback between physical and logical design, because decisions taken during physical design for improving performance may affect the logical data model.

16.1.3 Critical Success Factors in Database Design

The following guidelines are often critical to the success of database design:

- Work interactively with the users as much as possible.
- Follow a structured methodology throughout the data modeling process.

- Employ a data-driven approach.
- Incorporate structural and integrity considerations into the data models.
- Combine conceptualization, normalization, and transaction validation techniques into the data modeling methodology.
- Use diagrams to represent as much of the data models as possible.
- Use a Database Design Language (DBDL) to represent additional data semantics that cannot easily be represented in a diagram.
- Build a data dictionary to supplement the data model diagrams and the DBDL.
- Be willing to repeat steps.

These factors are built into the methodology we present for database design.

16.2 Overview of the Database Design Methodology

In this section, we present an overview of the database design methodology. The steps in the methodology are as follows.

Conceptual database design

Step 1 Build conceptual data model
- Step 1.1 Identify entity types
- Step 1.2 Identify relationship types
- Step 1.3 Identify and associate attributes with entity or relationship types
- Step 1.4 Determine attribute domains
- Step 1.5 lDetermine candidate, primary, and alternate key attributes
- Step 1.6 Consider use of enhanced modeling concepts (optional step)
- Step 1.7 Check model for redundancy
- Step 1.8 Validate conceptual data model against user transactions
- Step 1.9 Review conceptual data model with user

Logical database design for the relational model

Step 2 Build logical data model
- 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 model (optional step)
- Step 2.7 Check for future growth

Physical database design for relational databases

Step 3 Translate logical data model for target DBMS
- Step 3.1 Design base relations

- Step 3.2 Design representation of derived data
- Step 3.3 Design general constraints

Step 4 Design file organizations and indexes
- Step 4.1 Analyze transactions
- Step 4.2 Choose file organizations
- Step 4.3 Choose indexes
- Step 4.4 Estimate disk space requirements

Step 5 Design user views

Step 6 Design security mechanisms

Step 7 Consider the introduction of controlled redundancy

Step 8 Monitor and tune the operational system

This methodology can be used to design relatively simple to highly complex database systems. Just as the database design stage of the database systems development lifecycle (see Section 10.6) has three phases—conceptual, logical, and physical design—so too has the methodology. Step 1 creates a conceptual database design, Step 2 creates a logical database design, and Steps 3 to 8 create a physical database design. Depending on the complexity of the database system being built, some of the steps may be omitted. For example, Step 2.6 of the methodology is not required for database systems with a single user view or database systems with multiple user views being managed using the centralized approach (see Section 10.5). For this reason, we refer to the creation of a single conceptual data model only in Step 1 and single logical data model only in Step 2. However, if the database designer is using the view integration approach (see Section 10.5) to manage user views for a database system, then Steps 1 and 2 may be repeated as necessary to create the required number of models, which are then merged in Step 2.6.

In Chapter 10, we introduced the term "local conceptual data model" or "local logical data model" to refer to the modeling of one or more, but not all, user views of a database system and the term "global logical data model" to refer to the modeling of all user views of a database system. However, the methodology is presented using the more general terms "conceptual data model" and "logical data model" with the exception of the optional Step 2.6, which necessitates the use of the terms *local* logical data model and *global* logical data model, as it is this step that describes the tasks necessary to merge separate local logical data models to produce a global logical data model.

An important aspect of any design methodology is to ensure that the models produced are repeatedly validated so that they continue to be an accurate representation of the part of the enterprise being modeled. In this methodology the data models are validated in various ways such as by using normalization (Step 2.2), by ensuring the critical transactions are supported (Steps 1.8 and 2.3), and by involving the users as much as possible (Steps 1.9 and 2.5).

The logical model created at the end of Step 2 is then used as the source of information for physical database design described in Steps 3 to 8. Again, depending on the complexity of the database systems being designed and/or the functionality of the target DBMS, some of the steps of physical database design may be omitted. For example, Step 4.2 may not be applicable for certain PC-based

DBMSs. The steps of physical database design are described in detail in Chapters 18 and 19.

Database design is an iterative process that has a starting point and an almost endless procession of refinements. Although the steps of the methodology are presented here as a procedural process, it must be emphasized that this does not imply that it should be performed in this manner. It is likely that knowledge gained in one step may alter decisions made in a previous step. Similarly, it may be useful to look briefly at a later step to help with an earlier step. Therefore, the methodology should act as a framework to help guide the designer through database design effectively.

To illustrate the database design methodology we use the *DreamHome* case study. The *DreamHome* database has several user views (Director, Manager, Supervisor, Assistant, and Client) that are managed using a combination of the centralized and view integration approaches (see Section 11.4). Applying the centralized approach resulted in the identification of two collections of user views called StaffClient user views and Branch user views. The user views represented by each collection are as follows:

- **StaffClient user views**—representing Supervisor, Assistant, and Client user views;
- **Branch user views**—representing Director and Manager user views.

In this chapter, which describes Step 1 of the methodology, we use the StaffClient user views to illustrate the building of a conceptual data model, and then in the following chapter, which describes Step 2, we describe how this model is translated into a logical data model. As the StaffClient user views represent only a subset of all the user views of the *DreamHome* database, it is more correct to refer to the data models as local data models. However, as stated earlier when we described the methodology and the worked examples, for simplicity we use the terms conceptual data model and logical data model until the optional Step 2.6, which describes the integration of the local logical data models for the StaffClient user views and the Branch user views.

16.3 Conceptual Database Design Methodology

This section provides a step-by-step guide for conceptual database design.

Step 1: Build Conceptual Data Model

Objective To build a conceptual data model of the data requirements of the enterprise.

The first step in conceptual database design is to build one (or more) conceptual data models of the data requirements of the enterprise. A conceptual data model comprises:

- entity types;
- relationship types;
- attributes and attribute domains;

- primary keys and alternate keys;
- integrity constraints.

The conceptual data model is supported by documentation, including ER diagrams and a data dictionary, which is produced throughout the development of the model. We detail the types of supporting documentation that may be produced as we go through the various steps. The tasks involved in Step 1 are:

Step 1.1 Identify entity types
Step 1.2 Identify relationship types
Step 1.3 Identify and associate attributes with entity or relationship types
Step 1.4 Determine attribute domains
Step 1.5 Determine candidate, primary, and alternate key attributes
Step 1.6 Consider use of enhanced modeling concepts (optional step)
Step 1.7 Check model for redundancy
Step 1.8 Validate conceptual data model against user transactions
Step 1.9 Review conceptual data model with user

Step 1.1: Identify entity types

Objective To identify the required entity types.

The first step in building a conceptual data model is to determine and define the main objects that the users are interested in. These objects are the entity types for the model (see Section 12.1). One method of identifying entities is to examine the users' requirements specification. From this specification, we identify nouns or noun phrases that are mentioned (for example, staff number, staff name, property number, property address, rent, number of rooms). We also look for major objects, such as people, places, or concepts of interest, excluding those nouns that are merely qualities of other objects. For example, we could group staff number and staff name with an object or entity called Staff and group property number, property address, rent, and number of rooms with an entity called PropertyForRent.

An alternative way of identifying entities is to look for objects that have an existence in their own right. For example, Staff is an entity because staff exist whether or not we know their names, positions, and dates of birth. If possible, the users should assist with this activity.

It is sometimes difficult to identify entities because of the way they are presented in the users' requirements specification. Users often talk in terms of examples or analogies. Instead of talking about staff in general, users may mention people's names. In some cases, users talk in terms of job roles, particularly when people or organizations are involved. These roles may be job titles or responsibilities, such as Director, Manager, Supervisor, or Assistant.

To confuse matters further, users frequently use synonyms and homonyms. Two words are *synonyms* when they have the same meaning, for example, "branch" and "office." *Homonyms* occur when the same word can have different meanings depending on the context. For example, the word "program" has several alternative meanings such as a course of study, a series of events, a plan of work, and an item on the television.

It is not always obvious whether a particular object is an entity, a relationship, or an attribute. For example, how would we classify marriage? In fact, depending on the actual requirements, we could classify marriage as any or all of these. Design is subjective, and different designers may produce different, but equally valid, interpretations. The activity therefore relies, to a certain extent, on judgement and experience. Database designers must take a very selective view of the world and categorize the things that they observe within the context of the enterprise. Thus, there may be no unique set of entity types deducible from a given requirements specification. However, successive iterations of the design process should lead to the choice of entities that are at least adequate for the system required. For the StaffClient user views of *DreamHome*, we identify the following entities:

Staff	PropertyForRent
PrivateOwner	BusinessOwner
Client	Preference
Lease	

Document entity types

As entity types are identified, assign them names that are meaningful and obvious to the user. Record the names and descriptions of entities in a data dictionary. If possible, document the expected number of occurrences of each entity. If an entity is known by different names, the names are referred to as synonyms or *aliases*, which are also recorded in the data dictionary. Figure 16.1 shows an extract from the data dictionary that documents the entities for the StaffClient user views of *DreamHome*.

Step 1.2: Identify relationship types

Objective To identify the important relationships that exist between the entity types.

Having identified the entities, the next step is to identify all the relationships that exist between these entities (see Section 12.2). When we identify entities, one method

Entity name	*Description*	*Aliases*	*Occurrence*
Staff	General term describing all staff employed by *DreamHome*.	Employee	Each member of staff works at one particular branch.
PropertyForRent	General term describing all property for rent.	Property	Each property has a single owner and is available at one specific branch, where the property is managed by one member of staff. A property is viewed by many clients and rented by a single client, at any one time.

Figure 16.1 Extract from the data dictionary for the StaffClient user views of *DreamHome* showing a description of entities.

is to look for nouns in the users' requirements specification. Again, we can use the grammar of the requirements specification to identify relationships. Typically, relationships are indicated by verbs or verbal expressions. For example:

- Staff Manages PropertyForRent
- PrivateOwner Owns PropertyForRent
- PropertyForRent AssociatedWith Lease

The fact that the requirements specification records these relationships suggests that they are important to the enterprise and should be included in the model.

We are interested only in required relationships between entities. In the previous examples, we identified the Staff Manages PropertyForRent and the PrivateOwner Owns PropertyForRent relationships. We may also be inclined to include a relationship between Staff and PrivateOwner (for example, Staff Assists PrivateOwner). However, although this is a possible relationship, from the requirements specification, it is not a relationship that we are interested in modeling.

In most instances, the relationships are binary; in other words, the relationships exist between exactly two entity types. However, we should be careful to look out for complex relationships that may involve more than two entity types (see Section 12.2.1) and recursive relationships that involve only one entity type (see Section 12.2.2).

Great care must be taken to ensure that all the relationships that are either explicit or implicit in the users' requirements specification are detected. In principle, it should be possible to check each pair of entity types for a potential relationship between them, but this would be a daunting task for a large system comprising hundreds of entity types. On the other hand, it is unwise not to perform some such check, and the responsibility is often left to the analyst/designer. However, missing relationships should become apparent when we validate the model against the transactions that are to be supported (Step 1.8).

Use Entity–Relationship (ER) diagrams

It is often easier to visualize a complex system rather than decipher long textual descriptions of a users' requirements specification. We use ER diagrams to represent entities and how they relate to one another more easily. Throughout the database design phase, we recommend that ER diagrams be used whenever necessary to help build up a picture of the part of the enterprise that we are modeling. In this book, we use UML, but other notations perform a similar function (see Appendix C).

Determine the multiplicity constraints of relationship types

Having identified the relationships to model, we next determine the multiplicity of each relationship (see Section 12.6). If specific values for the multiplicity are known, or even upper or lower limits, document these values as well.

Multiplicity constraints are used to check and maintain data quality. These constraints are assertions about entity occurrences that can be applied when the database is updated to determine whether the updates violate the stated rules of the enterprise. A model that includes multiplicity constraints more explicitly represents

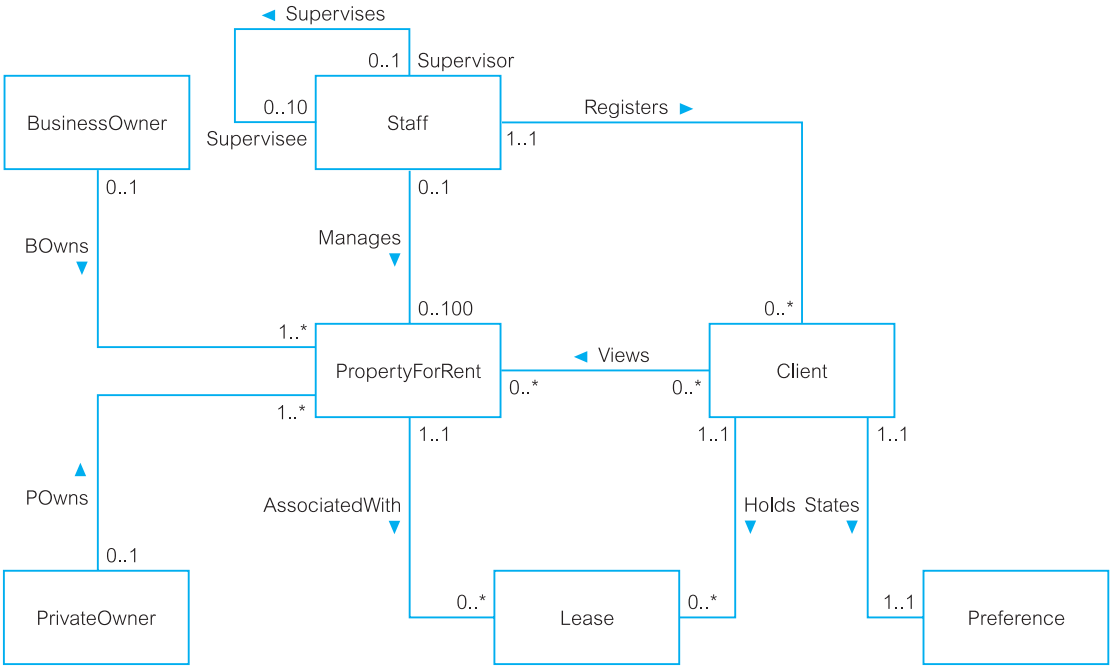


Figure 16.2 First-cut ER diagram showing entity and relationship types for the StaffClient user views of *DreamHome*.

the semantics of the relationships and results in a better representation of the data requirements of the enterprise.

Check for fan and chasm traps

Having identified the necessary relationships, check that each relationship in the ER model is a true representation of the "real world," and that fan or chasm traps have not been created inadvertently (see Section 12.7).

Figure 16.2 shows the first-cut ER diagram for the StaffClient user views of the *DreamHome* case study.

Document relationship types

As relationship types are identified, assign them names that are meaningful and obvious to the user. Also record relationship descriptions and the multiplicity constraints in the data dictionary. Figure 16.3 shows an extract from the data dictionary that documents the relationships for the StaffClient user views of *DreamHome*.

Step 1.3: Identify and associate attributes with entity or relationship types

Objective To associate attributes with appropriate entity or relationship types.

The next step in the methodology is to identify the types of facts about the entities and relationships that we have chosen to be represented in the database. In a

Entity name	Multiplicity	Relationship	Multiplicity	Entity name
Staff	0..1 0..1	*Manages* *Supervises*	0..100 0..10	PropertyForRent Staff
PropertyForRent	1..1	*AssociatedWith*	0..*	Lease

Figure 16.3
Extract from the data dictionary for the StaffClient user views of *DreamHome*, showing a description of relationships.

similar way to identifying entities, we look for nouns or noun phrases in the users' requirements specification. The attributes can be identified where the noun or noun phrase is a property, quality, identifier, or characteristic of one of these entities or relationships (see Section 12.3).

By far the easiest thing to do when we have identified an entity (x) or a relationship (y) in the requirements specification is to ask "*What information are we required to hold on x or y*?" The answer to this question should be described in the specification. However, in some cases it may be necessary to ask the users to clarify the requirements. Unfortunately, they may give answers to this question that also contain other concepts, so the users' responses must be carefully considered.

Simple/composite attributes

It is important to note whether an attribute is simple or composite (see Section 12.3.1). Composite attributes are made up of simple attributes. For example, the address attribute can be simple and hold all the details of an address as a single value, such as "115 Dumbarton Road, Glasgow, G11 6YG." However, the address attribute may also represent a composite attribute, made up of simple attributes that hold the address details as separate values in the attributes street ("115 Dumbarton Road"), city ("Glasgow"), and postcode ("G11 6YG"). The option to represent address details as a simple or composite attribute is determined by the users' requirements. If the user does not need to access the separate components of an address, we represent the address attribute as a simple attribute. On the other hand, if the user does need to access the individual components of an address, we represent the address attribute as being composite, made up of the required simple attributes.

In this step, it is important that we identify all simple attributes to be represented in the conceptual data model including those attributes that make up a composite attribute.

Single/multi-valued attributes

In addition to being simple or composite, an attribute can also be single-valued or multi-valued (see Section 12.3.2). Most attributes encountered will be single-valued, but occasionally a multi-valued attribute may be encountered; that is, an attribute that holds multiple values for a single entity occurrence. For example, we may identify the attribute telNo (the telephone number) of the Client entity as a multi-valued attribute.

On the other hand, client telephone numbers may have been identified as a separate entity from Client. This is an alternative, and equally valid, way to model this. As you will see in Step 2.1, multi-valued attributes are mapped to relations anyway, so both approaches produce the same end result.

Derived attributes

Attributes whose values are based on the values of other attributes are known as *derived attributes* (see Section 12.3.3). Examples of derived attributes include:

- the age of a member of staff;
- the number of properties that a member of staff manages;
- the rental deposit (calculated as twice the monthly rent).

Often, these attributes are not represented in the conceptual data model. However, sometimes the value of the attribute or attributes on which the derived attribute is based may be deleted or modified. In this case, the derived attribute must be shown in the data model to avoid this potential loss of information. However, if a derived attribute is shown in the model, we must indicate that it is derived. The representation of derived attributes will be considered during physical database design. Depending on how an attribute is used, new values for a derived attribute may be calculated each time it is accessed or when the value(s) it is derived from changes. However, this issue is not the concern of conceptual database design, and is discussed in more detail in Step 3.2 in Chapter 18.

Potential problems

When identifying the entities, relationships, and attributes for the user views, it is not uncommon for it to become apparent that one or more entities, relationships, or attributes have been omitted from the original selection. In this case, return to the previous steps, document the new entities, relationships, or attributes, and re-examine any associated relationships.

As there are generally many more attributes than entities and relationships, it may be useful to first produce a list of all attributes given in the users' requirements specification. As an attribute is associated with a particular entity or relationship, remove the attribute from the list. In this way, we ensure that an attribute is associated with only one entity or relationship type and, when the list is empty, that all attributes are associated with some entity or relationship type.

We must also be aware of cases where attributes appear to be associated with more than one entity or relationship type, as this can indicate the following:

(1) We have identified several entities that can be represented as a single entity. For example, we may have identified entities Assistant and Supervisor both with the attributes staffNo (the staff number), name, sex, and DOB (date of birth), which can be represented as a single entity called Staff with the attributes staffNo (the staff number), name, sex, DOB, and position (with values Assistant or Supervisor). On the other hand, it may be that these entities share many attributes but there are also attributes or relationships that are unique to each entity. In this case, we must decide whether we want to generalize the entities into a single entity such as Staff, or leave them as specialized entities representing distinct staff

roles. The consideration of whether to specialize or generalize entities was discussed in Chapter 13 and is addressed in more detail in Step 1.6.

(2) We have identified a relationship between entity types. In this case, we must associate the attribute with only *one* entity, the parent entity, and ensure that the relationship was previously identified in Step 1.2. If this is not the case, the documentation should be updated with details of the newly identified relationship. For example, we may have identified the entities Staff and PropertyForRent with the following attributes:

Staff	staffNo, name, position, sex, DOB
PropertyForRent	propertyNo, street, city, postcode, type, rooms, rent, managerName

The presence of the managerName attribute in PropertyForRent is intended to represent the relationship Staff *Manages* PropertyForRent. In this case, the managerName attribute should be omitted from PropertyForRent and the relationship *Manages* should be added to the model.

DreamHome attributes for entities

For the StaffClient user views of *DreamHome*, we identify and associate attributes with entities as follows:

Staff	staffNo, name (composite: fName, lName), position, sex, DOB
PropertyForRent	propertyNo, address (composite: street, city, postcode), type, rooms, rent
PrivateOwner	ownerNo, name (composite: fName, lName), address, telNo
BusinessOwner	ownerNo, bName, bType, address, telNo, contactName
Client	clientNo, name (composite: fName, lName), telNo, eMail
Preference	prefType, maxRent
Lease	leaseNo, paymentMethod, deposit (derived as PropertyForRent.rent*2), depositPaid, rentStart, rentFinish, duration (derived as rentFinish – rentStart)

DreamHome attributes for relationships

Some attributes should not be associated with entities, but instead should be associated with relationships. For the StaffClient user views of *DreamHome*, we identify and associate attributes with relationships, as follows:

Views	viewDate, comment

Document attributes

As attributes are identified, assign them names that are meaningful to the user. Record the following information for each attribute:

- attribute name and description;
- data type and length;
- any aliases that the attribute is known by;
- whether the attribute is composite and, if so, the simple attributes that make up the composite attribute;
- whether the attribute is multi-valued;

Entity name	*Attributes*	*Description*	*Data Type & Length*	*Nulls*	*Multi-valued*	. . .
Staff	**staffNo**	Uniquely identifies a member of staff	5 variable characters	No	No	
	name					
	fName	First name of staff	15 variable characters	No	No	
	lName	Last name of staff	15 variable characters	No	No	
	position	Job title of member of staff	10 variable characters	No	No	
	sex	Gender of member of staff	1 character (M or F)	Yes	No	
	DOB	Date of birth of member of staff	Date	Yes	No	
PropertyForRent	**propertyNo**	Uniquely identifies a property for rent	5 variable characters	No	No	

Figure 16.4 Extract from the data dictionary for the StaffClient user views of *DreamHome* showing a description of attributes.

- whether the attribute is derived and, if so, how it is to be computed;
- any default value for the attribute.

Figure 16.4 shows an extract from the data dictionary that documents the attributes for the StaffClient user views of *DreamHome*.

Step 1.4: Determine attribute domains

Objective To determine domains for the attributes in the conceptual data model.

The objective of this step is to determine domains for all the attributes in the model (see Section 12.3). A **domain** is a pool of values from which one or more attributes draw their values. For example, we may define:

- the attribute domain of valid staff numbers (staffNo) as being a five-character variable-length string, with the first two characters as letters and the next one to three characters as digits in the range 1–999;
- the possible values for the sex attribute of the Staff entity as being either "M" or "F." The domain of this attribute is a single character string consisting of the values "M" or "F."

A fully developed data model specifies the domains for each attribute and includes:

- allowable set of values for the attribute;
- sizes and formats of the attribute.

Further information can be specified for a domain, such as the allowable operations on an attribute, and which attributes can be compared with other attributes or used in combination with other attributes. However, implementing these characteristics of attribute domains in a DBMS is still the subject of research.

Document attribute domains

As attribute domains are identified, record their names and characteristics in the data dictionary. Update the data dictionary entries for attributes to record their domain in place of the data type and length information.

Step 1.5: Determine candidate, primary, and alternate key attributes

Objective To identify the candidate key(s) for each entity type and, if there is more than one candidate key, to choose one to be the primary key and the others as alternate keys.

This step is concerned with identifying the candidate key(s) for an entity and then selecting one to be the primary key (see Section 12.3.4). A **candidate key** is a minimal set of attributes of an entity that uniquely identifies each occurrence of that entity. We may identify more than one candidate key, in which case we must choose one to be the **primary key**; the remaining candidate keys are called **alternate keys**.

People's names generally do not make good candidate keys. For example, we may think that a suitable candidate key for the Staff entity would be the composite attribute name, the member of staff's name. However, it is possible for two people with the same name to join *DreamHome,* which would clearly invalidate the choice of name as a candidate key. We could make a similar argument for the names of property owners. In such cases, rather than coming up with combinations of attributes that may provide uniqueness, it may be better to use an existing attribute that would always ensure uniqueness, such as the staffNo attribute for the Staff entity and the ownerNo attribute for the PrivateOwner entity, or define a new attribute that would provide uniqueness.

When choosing a primary key from among the candidate keys, use the following guidelines to help make the selection:

- the candidate key with the minimal set of attributes;
- the candidate key that is least likely to have its values changed;
- the candidate key with fewest characters (for those with textual attribute(s));
- the candidate key with smallest maximum value (for those with numerical attribute(s));
- the candidate key that is easiest to use from the users' point of view.

In the process of identifying primary keys, note whether an entity is strong or weak. If we are able to assign a primary key to an entity, the entity is referred to as being *strong.* On the other hand, if we are unable to identify a primary key for an entity, the entity is referred to as being *weak* (see Section 12.4). The primary key of a weak entity can be identified only when we map the weak entity and its relationship with its owner entity to a relation through the placement of a foreign key in that relation. The process of mapping entities and their relationships to relations is described in Step 2.1, and therefore the identification of primary keys for weak entities cannot take place until that step.

DreamHome primary keys

The primary keys for the StaffClient user views *of DreamHome* are shown in Figure 16.5. Note that the Preference entity is a weak entity and, as identified previously, the Views relationship has two attributes, viewDate and comment.

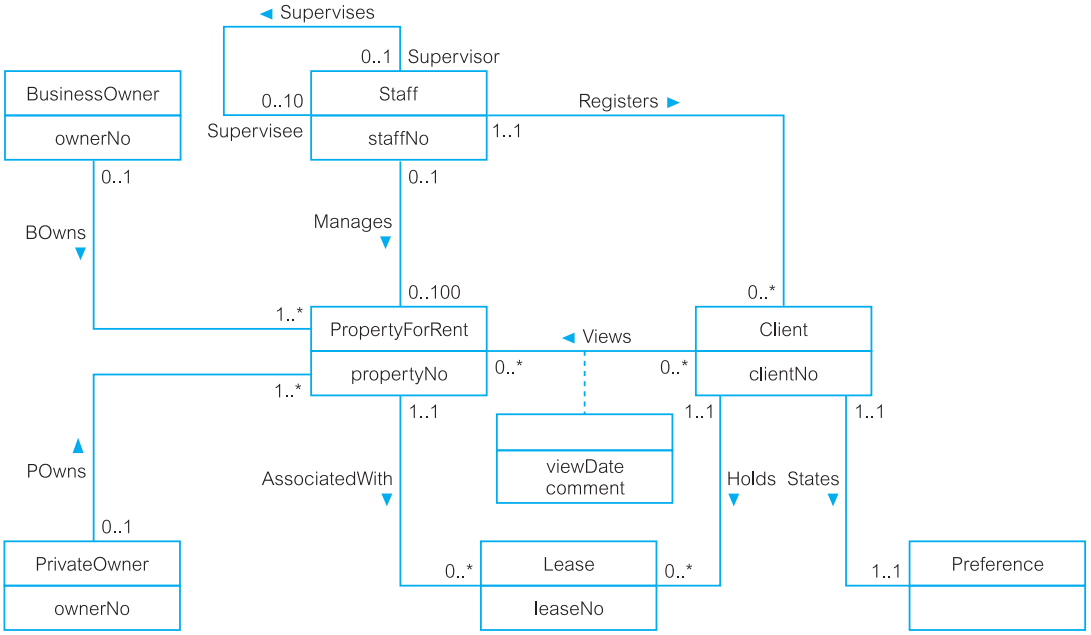


Figure 16.5 ER diagram for the StaffClient user views of *DreamHome* with primary keys added.

Document primary and alternate keys

Record the identification of primary and any alternate keys in the data dictionary.

Step 1.6: Consider use of enhanced modeling concepts (optional step)

Objective To consider the use of enhanced modeling concepts, such as specialization/generalization, aggregation, and composition.

In this step, we have the option to continue the development of the ER model using the enhanced modeling concepts discussed in Chapter 13, namely specialization/generalization, aggregation, and composition. If we select the specialization approach, we attempt to highlight differences between entities by defining one or more **subclasses** of a **superclass** entity. If we select the generalization approach, we attempt to identify common features between entities to define a generalizing superclass entity. We may use aggregation to represent a 'has-a' or 'is-part-of' relationship between entity types, where one represents the 'whole' and the other the 'part'. We may use composition (a special type of aggregation) to represent an association between entity types where there is a strong ownership and coincidental lifetime between the 'whole' and the 'part'.

For the StaffClient user views of *DreamHome,* we choose to generalize the two entities PrivateOwner and BusinessOwner to create a superclass Owner that contains the common attributes ownerNo, address, and telNo. The relationship that the Owner superclass has with its subclasses is *mandatory* and *disjoint,* denoted as {Mandatory, Or}: each member of the Owner superclass must be a member of one of the subclasses, but cannot belong to both.

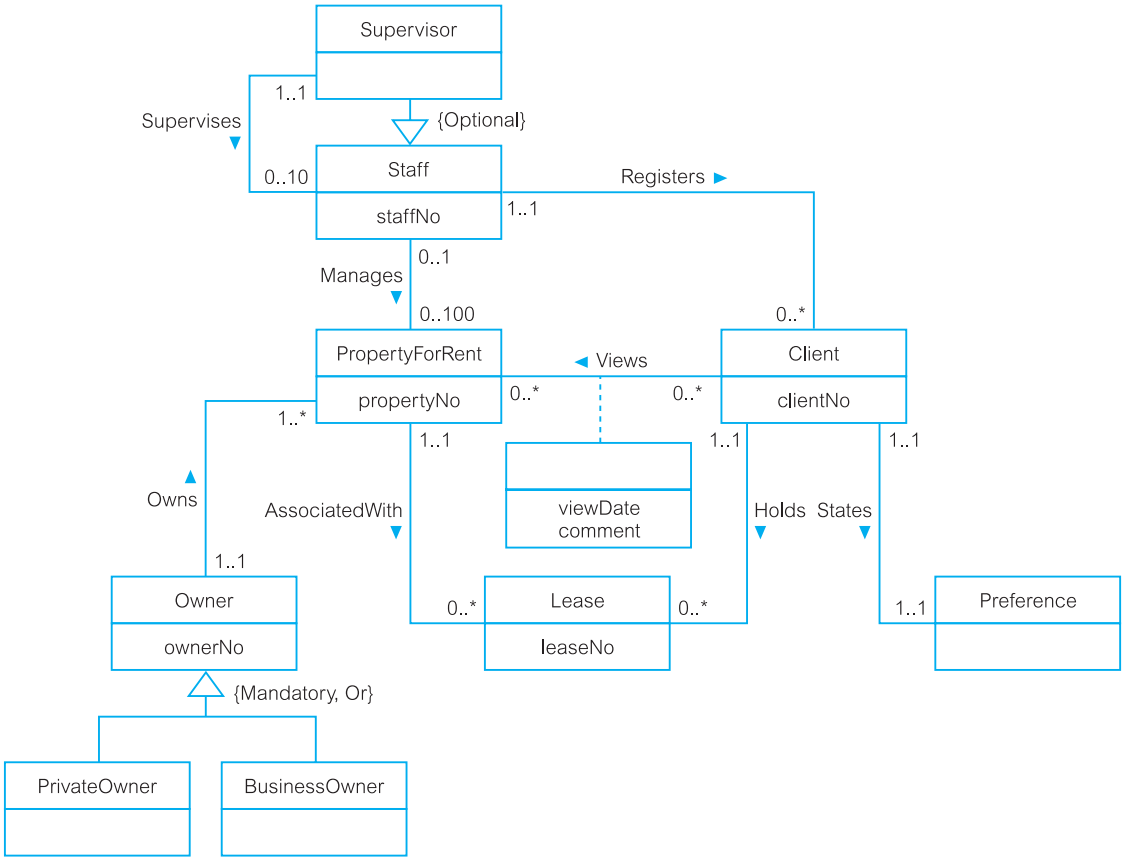


Figure 16.6 Revised ER diagram for the StaffClient user views of *DreamHome* with specialization/generalization added.

In addition, we identify one specialization subclass of Staff, namely Supervisor, specifically to model the *Supervises* relationship. The relationship that the Staff superclass has with the Supervisor subclass is *optional*: a member of the Staff superclass does not necessarily have to be a member of the Supervisor subclass. To keep the design simple, we decide not to use aggregation or composition. The revised ER diagram for the StaffClient user views *of DreamHome* is shown in Figure 16.6.

There are no strict guidelines on when to develop the ER model using enhanced modeling concepts, as the choice is often subjective and dependent on the particular characteristics of the situation that is being modeled. As a useful "rule of thumb" when considering the use of these concepts, always attempt to represent the important entities and their relationships as clearly as possible in the ER diagram. Therefore, the use of enhanced modeling concepts should be guided by the readability of the ER diagram and the clarity by which it models the important entities and relationships.

These concepts are associated with enhanced ER modeling. However, as this step is optional, we simply use the term "ER diagram" when referring to the diagrammatic representation of data models throughout the methodology.

Step 1.7: Check model for redundancy

Objective To check for the presence of any redundancy in the model.

In this step, we examine the conceptual data model with the specific objective of identifying whether there is any redundancy present and removing any that does exist. The three activities in this step are:

(1) re-examine one-to-one (1:1) relationships;
(2) remove redundant relationships;
(3) consider time dimension.

(1) Re-examine one-to-one (1:1) relationships

In the identification of entities, we may have identified two entities that represent the same object in the enterprise. For example, we may have identified the two entities Client and Renter that are actually the same; in other words, Client is a synonym for Renter. In this case, the two entities should be merged together. If the primary keys are different, choose one of them to be the primary key and leave the other as an alternate key.

(2) Remove redundant relationships

A relationship is redundant if the same information can be obtained via other relationships. We are trying to develop a minimal data model and, as redundant relationships are unnecessary, they should be removed. It is relatively easy to identify whether there is more than one path between two entities. However, this does not necessarily imply that one of the relationships is redundant, as they may represent different associations between the entities. For example, consider the relationships between the PropertyForRent, Lease, and Client entities shown in Figure 16.7. There are two ways to find out which clients rent which properties. There is the direct route using the *Rents* relationship between the Client and PropertyForRent entities, and there is the indirect route, using the *Holds* and *AssociatedWith* relationships via the Lease entity. Before we can assess whether both routes are required, we need

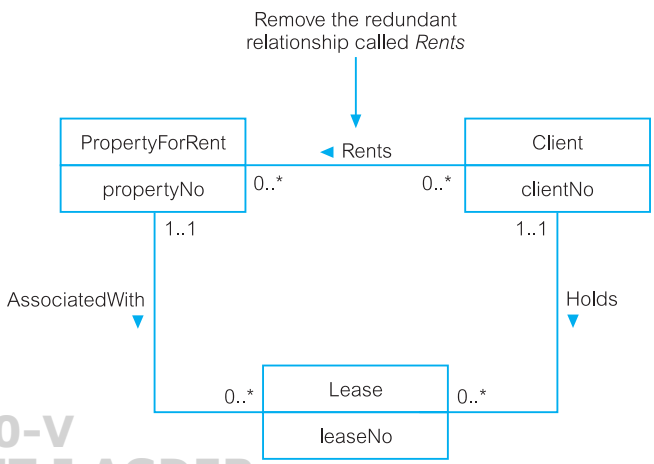


Figure 16.7 Remove the redundant relationship called *Rents*.

to establish the purpose of each relationship. The *Rents* relationship indicates which client rents which property. On the other hand, the *Holds* relationship indicates which client holds which lease, and the *AssociatedWith* relationship indicates which properties are associated with which leases. Although it is true that there is a relationship between clients and the properties they rent, this is not a direct relationship and the association is more accurately represented through a lease. The *Rents* relationship is therefore redundant and does not convey any additional information about the relationship between PropertyForRent and Client that cannot more correctly be found through the Lease entity. To ensure that we create a minimal model, the redundant *Rents* relationship must be removed.

(3) Consider time dimension

The time dimension of relationships is important when assessing redundancy. For example, consider the situation in which we wish to model the relationships between the entities Man, Woman, and Child, as illustrated in Figure 16.8. Clearly, there are two paths between Man and Child: one via the direct relationship *FatherOf* and the other via the relationships *MarriedTo* and *MotherOf*. Consequently, we may think that the relationship *FatherOf* is unnecessary. However, this would be incorrect for two reasons:

(1) The father may have children from a previous marriage, and we are modeling only the father's current marriage through a 1:1 relationship.
(2) The father and mother may not be married, or the father may be married to someone other than the mother (or the mother may be married to someone who is not the father).

In either case, the required relationship could not be modeled without the *FatherOf* relationship. The message is that it is important to examine the meaning of each relationship between entities when assessing redundancy. At the end of this step, we have simplified the local conceptual data model by removing any inherent redundancy.

Step 1.8: Validate conceptual data model against user transactions

Objective To ensure that the conceptual data model supports the required transactions.

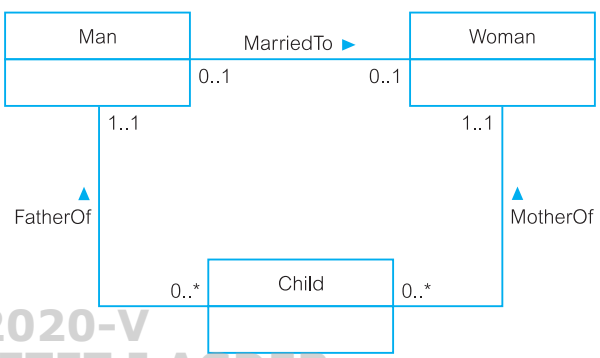


Figure 16.8 Example of a nonredundant relationship *FatherOf*.

We now have a conceptual data model that represents the data requirements of the enterprise. The objective of this step is to check the model to ensure that the model supports the required transactions. Using the model, we attempt to perform the operations manually. If we can resolve all transactions in this way, we have checked that the conceptual data model supports the required transactions. However, if we are unable to perform a transaction manually, there must be a problem with the data model, which must be resolved. In this case, it is likely that we have omitted an entity, a relationship, or an attribute from the data model.

We examine two possible approaches to ensuring that the conceptual data model supports the required transactions:

(1) describing the transactions;

(2) using transaction pathways.

Describing the transaction

Using the first approach, we check that all the information (entities, relationships, and their attributes) required by each transaction is provided by the model, by documenting a description of each transaction's requirements. We illustrate this approach for an example *DreamHome* transaction listed in Appendix A from the StaffClient user views:

> **Transaction (d): List the details of properties managed by a named member of staff at the branch**
>
> The details of properties are held in the PropertyForRent entity and the details of staff who manage properties are held in the Staff entity. In this case, we can use the Staff *Manages* PropertyForRent relationship to produce the required list.

Using transaction pathways

The second approach to validating the data model against the required transactions involves diagrammatically representing the pathway taken by each transaction directly on the ER diagram. An example of this approach for the query transactions for the StaffClient user views listed in Appendix A is shown in Figure 16.9. Clearly, the more transactions that exist, the more complex this diagram would become, so for readability we may need several such diagrams to cover all the transactions.

This approach allows the designer to visualize areas of the model that are not required by transactions and those areas that are critical to transactions. We are therefore in a position to directly review the support provided by the data model for the transactions required. If there are areas of the model that do not appear to be used by any transactions, we may question the purpose of representing this information in the data model. On the other hand, if there are areas of the model that are inadequate in providing the correct pathway for a transaction, we may need to investigate the possibility that critical entities, relationships, or attributes have been missed.

It may look like a lot of hard work to check every transaction that the model has to support in this way, and it certainly can be. As a result, it may be tempting to omit this step. However, it is very important that these checks are performed now

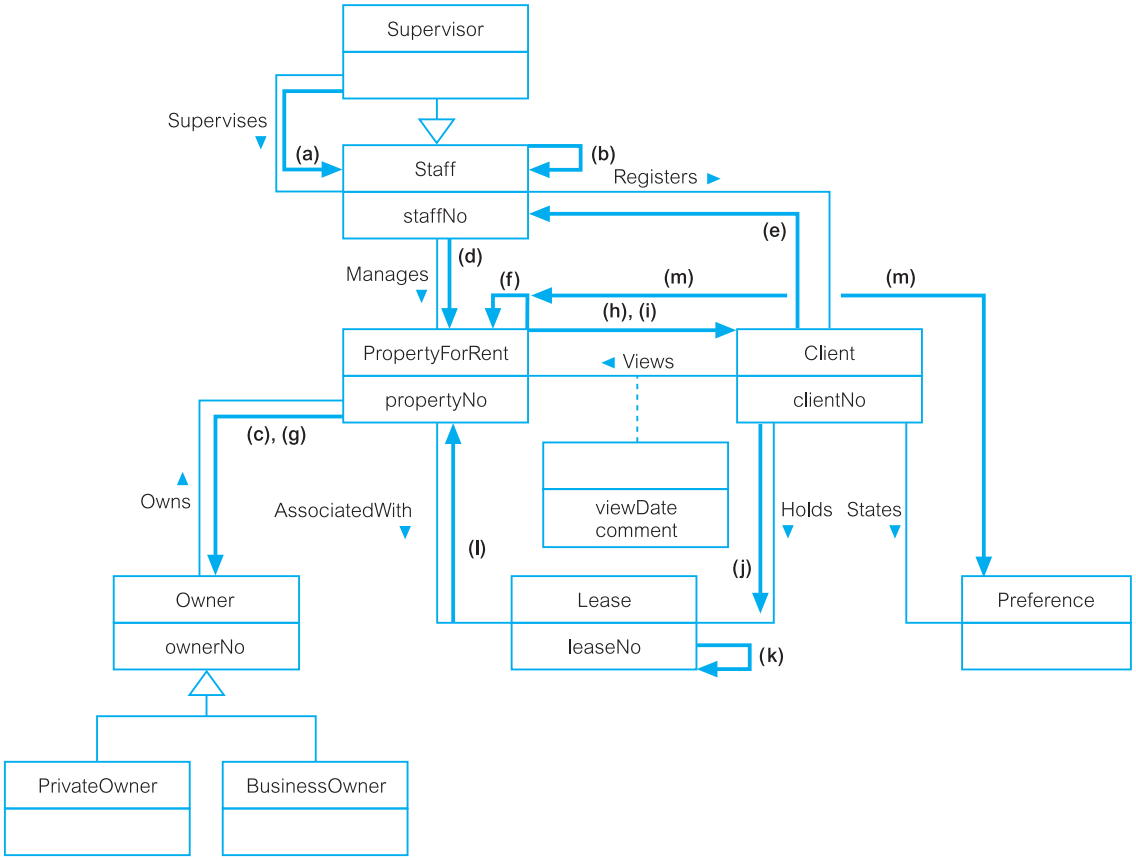


Figure 16.9 Using pathways to check whether the conceptual data model supports the user transactions.

rather than later, when it is much more difficult and expensive to resolve any errors in the data model.

Step 1.9: Review conceptual data model with user

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

Before completing Step 1, we review the conceptual data model with the user. The conceptual data model includes the ER diagram and the supporting documentation that describes the data model. If any anomalies are present in the data model, we must make the appropriate changes, which may require repeating the previous step(s). We repeat this process until the user is prepared to "sign off" the model as being a "true" representation of the part of the enterprise that we are modeling.

The steps in this methodology are summarized in Appendix D The next chapter describes the steps of the logical database design methodology.

Chapter Summary

- A **design methodology** is a structured approach that uses procedures, techniques, tools, and documentation aids to support and facilitate the process of design.
- Database design includes three main phases: **conceptual**, **logical**, and **physical** database design.
- **Conceptual database design** is the process of constructing a model of the data used in an enterprise, independent of *all* physical considerations.
- Conceptual database design begins with the creation of a **conceptual data model** of the enterprise, which is entirely independent of implementation details such as the target DBMS, application programs, programming languages, hardware platform, performance issues, or any other physical considerations.
- **Logical database design** is the process of constructing a model of the data used in an enterprise based on a specific data model (such as the relational model), but independent of a particular DBMS and other physical considerations. Logical database design translates the conceptual data model into a **logical data model** of the enterprise.
- **Physical database design** is the process of producing a description of the implementation of the database on secondary storage; it describes the base relations, file organizations, and indexes used to achieve efficient access to the data, and any associated integrity constraints and security measures.
- The physical database design phase allows the designer to make decisions on how the database is to be implemented. Therefore, **physical design** is tailored to a specific DBMS. There is feedback between physical and conceptual/logical design, because decisions taken during physical design to improve performance may affect the structure of the conceptual/logical data model.
- There are several critical factors for the success of the database design stage, including, for example, working interactively with users and being willing to repeat steps.
- The main objective of Step 1 of the methodology is to build a conceptual data model of the data requirements of the enterprise. A conceptual data model comprises: entity types, relationship types, attributes, attribute domains, primary keys, and alternate keys.
- A conceptual data model is supported by documentation, such as ER diagrams and a data dictionary, which is produced throughout the development of the model.
- The conceptual data model is validated to ensure it supports the required transactions. Two possible approaches to ensure that the conceptual data model supports the required transactions are: (1) checking that all the information (entities, relationships, and their attributes) required by each transaction is provided by the model by documenting a description of each transaction's requirements; (2) diagrammatically representing the pathway taken by each transaction directly on the ER diagram.

Review Questions

16.1 Describe the purpose of a design methodology.

16.2 Describe the main phases involved in database design.

16.3 Identify important factors in the success of database design.

16.4 Discuss the important role played by users in the process of database design.

16.5 Describe the main objective of conceptual database design.

16.6 What is the purpose of conceptual database design? How is it utilized while designing a database?

16.7 How would you identify entity and relationship types from a user's requirements specification?

16.8 How would you identify attributes from a user's requirements specification and then associate the attributes with entity or relationship types?

16.9 Using enhanced modeling concepts in the conceptual design is optional. Describe the possible EER concepts that can be presented in a conceptual model.

16.10 How would you check a data model for redundancy? Give an example to illustrate your answer.

16.11 Discuss why you would want to validate a conceptual data model and describe two approaches to validating a conceptual data model.

16.12 Identify and describe the purpose of the documentation generated during conceptual database design.

Exercises

The *DreamHome* case study

16.13 Create a conceptual data model for the Branch user views of *DreamHome* documented in Appendix A. Compare your ER diagram with Figure 13.8 and justify any differences found.

16.14 Analyze the *DreamHome* case study and examine if there are situations that call for enhanced modeling. Present the enhanced data model of the case.

The *University Accommodation Office* case study

16.15 Describe how you will approach the task of creating the conceptual model for university accommodation documented in Appendix B.1.

16.16 Create a conceptual data model for the case study. State any assumptions necessary to support your design. Check that the conceptual data model supports the required transactions.

The *EasyDrive School of Motoring* case study

16.17 Analyze the *EasyDrive School of Motoring* case study documented in Appendix B.2 and prepare a data dictionary detailing all the key conceptual issues.

16.18 Create a conceptual data model for the case study. State any assumptions necessary to support your design. Check that the conceptual data model supports the required transactions.

The *Wellmeadows Hospital* case study

16.19 Identify user views for the Medical Director and Charge Nurse in the *Wellmeadows Hospital* case study described in Appendix B.3.

16.20 Provide a users' requirements specification for each of these user views.

16.21 Create conceptual data models for each of the user views. State any assumptions necessary to support your design.

CHAPTER

17 Methodology—Logical Database Design for the Relational Model

Chapter Objectives

In this chapter you will learn:

- How to derive a set of relations from a conceptual data model.
- How to validate these relations using the technique of normalization.
- How to validate a logical data model to ensure it supports the required transactions.
- How to merge local logical data models based on one or more user views into a global logical data model that represents all user views.
- How to ensure that the final logical data model is a true and accurate representation of the data requirements of the enterprise.

In Chapter 10, we described the main stages of the database system development lifecycle, one of which is database design. This stage is made up of three phases: conceptual, logical, and physical database design. In the previous chapter we introduced a methodology that describes the steps that make up the three phases of database design and then presented Step 1 of this methodology for conceptual database design.

In this chapter we describe Step 2 of the methodology, which translates the conceptual model produced in Step 1 into a logical data model.

The methodology for logical database design described in this book also includes an optional Step 2.6, which is required when the database has multiple user views that are managed using the view integration approach (see Section 10.5). In this case, we repeat Steps 1 to 2.5 as necessary to create the required number of **local logical data models**, which are then finally merged in Step 2.6 to form a **global logical data model**. A local logical data model represents the data requirements of one or more but not all user views of a database and a global logical data model represents the data requirements for all user views (see Section 10.5). However, after concluding Step 2.6 we cease to use the term "global logical data model" and simply refer to the final model as being a "logical data model." The final step of the logical database design phase is to consider how well the model is able to support possible future developments for the database system.

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

logical data model that is a correct, comprehensive, and unambiguous representation of the data requirements of the enterprise.

We demonstrate Step 2 using the conceptual data model created in the previous chapter for the StaffClient user views of the *DreamHome* case study and represented in Figure 17.1 as an ER diagram. We also use the Branch user views of *DreamHome*, which is represented in Figure 13.8 as an ER diagram to illustrate some concepts that are not present in the StaffClient user views and to demonstrate the merging of data models in Step 2.6.

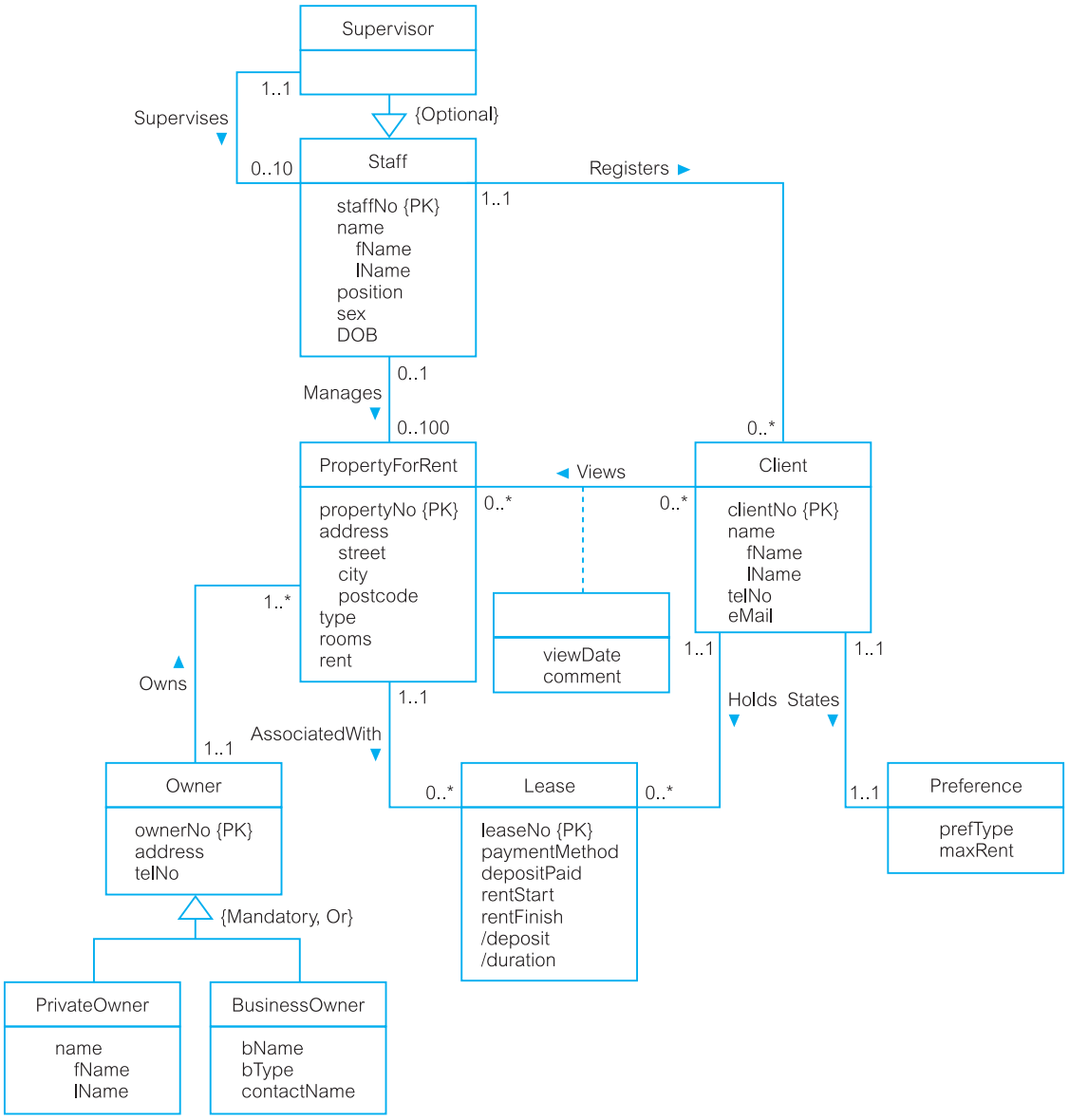


Figure 17.1 Conceptual data model for the StaffClient user views showing all attributes.

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 from each owner entity and so the identification of the primary key of a weak entity

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:

Post **staffNo** into **Client** to model 1:* ***Registers*** relationship

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

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

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.

(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:

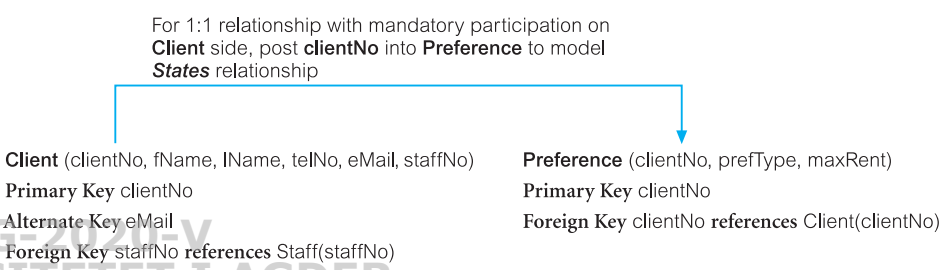
Client (clientNo, fName, lName, telNo, eMail, prefType, maxRent, staffNo)
Primary Key clientNo
Alternate Key eMail
Foreign Key staffNo **references** Staff(staffNo)

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

Option 1 – Mandatory, nondisjoint

AllOwner (ownerNo, address, telNo, fName, lName, bName, bType, contactName, pOwnerFlag, bOwnerFlag)
Primary Key ownerNo

Option 2 – Optional, nondisjoint

Owner (ownerNo, address, telNo)
Primary Key ownerNo

OwnerDetails (ownerNo, fName lName, bName, bType, contactName, pOwnerFlag, bOwnerFlag)
Primary Key ownerNo
Foreign Key ownerNo **references** Owner(ownerNo)

Option 3 – Mandatory, disjoint

PrivateOwner (ownerNo, fName, lName, address, telNo)
Primary Key ownerNo

BusinessOwner (ownerNo, bName, bType, contactName, address, telNo)
Primary Key ownerNo

Option 4 – Optional, disjoint

Owner (ownerNo, address, telNo)
Primary Key ownerNo

PrivateOwner (ownerNo, fName, lName)
Primary Key ownerNo
Foreign Key ownerNo **references** Owner(ownerNo)

BusinessOwner (ownerNo, bName, bType, contactName)
Primary Key ownerNo
Foreign Key ownerNo **references** Owner(ownerNo)

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)

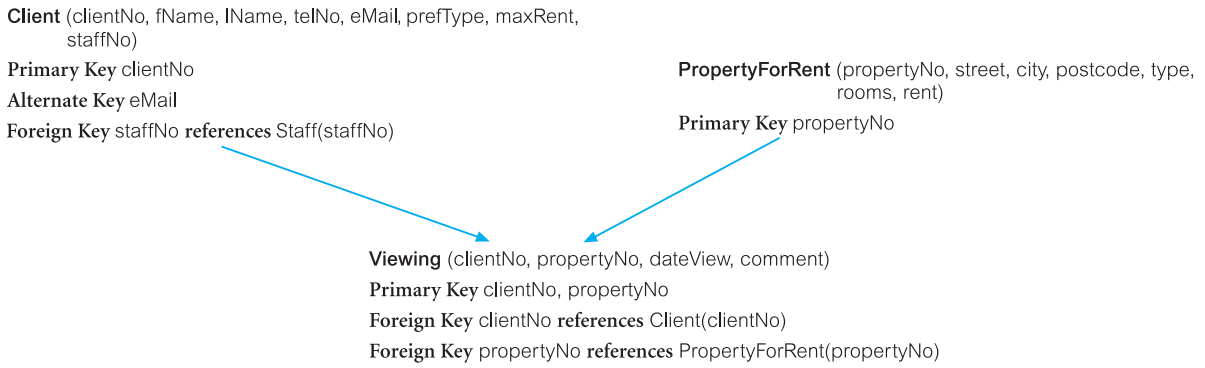
If we had left the superclass/subclass relationship as a 1:* recursive relationship as we had it originally in Figure 17.5 with optional participation on both sides this would have resulted in the same representation as previously.

(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.)

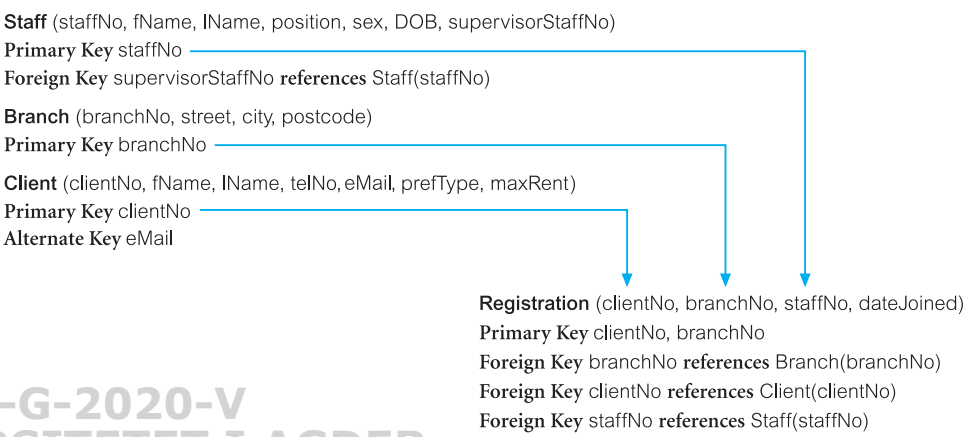
For example, consider the *:* relationship Client *Views* PropertyForRent shown in Figure 17.1. In this example, the Views relationship has two attributes called dateView and comments. To represent this, we create relations for the strong entities Client and PropertyForRent and we create a relation Viewing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

Post **branchNo** into **Telephone**

Branch (branchNo, street, city, postcode)
Primary Key branchNo

Telephone (telNo, branchNo)
Primary Key telNo
Foreign Key branchNo **references** Branch(branchNo)

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	newpaperName 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)			

[a]The Telephone relation is created from the multi-valued attribute telNo.

[b]The Registration relation is created from the ternary relationship *Registers*.

[c]The Advert relation is created from the many-to-many (*,*) relationship *Advertises*.

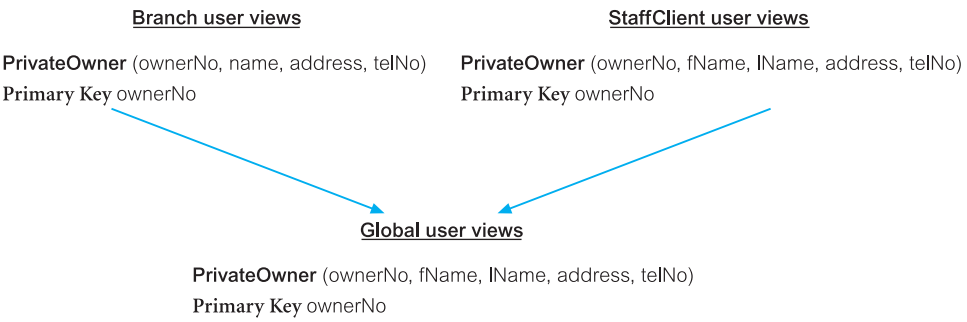


Figure 17.6 Merging the PrivateOwner relations from the Branch and StaffClient user views.

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

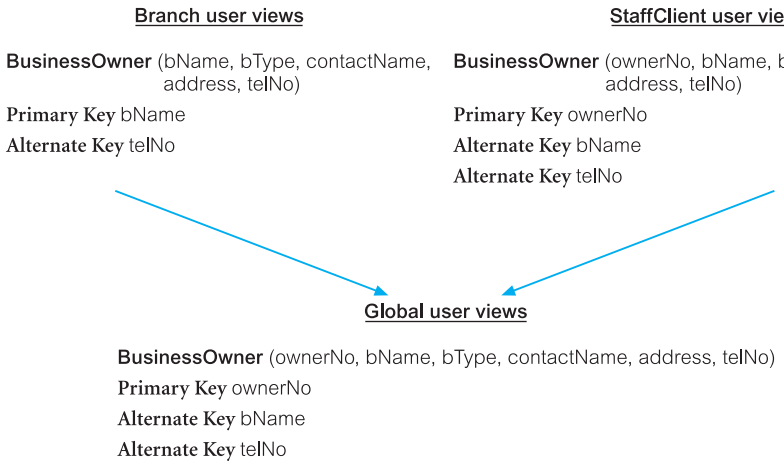


Figure 17.7 Merging the BusinessOwner relations with different primary keys.

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.

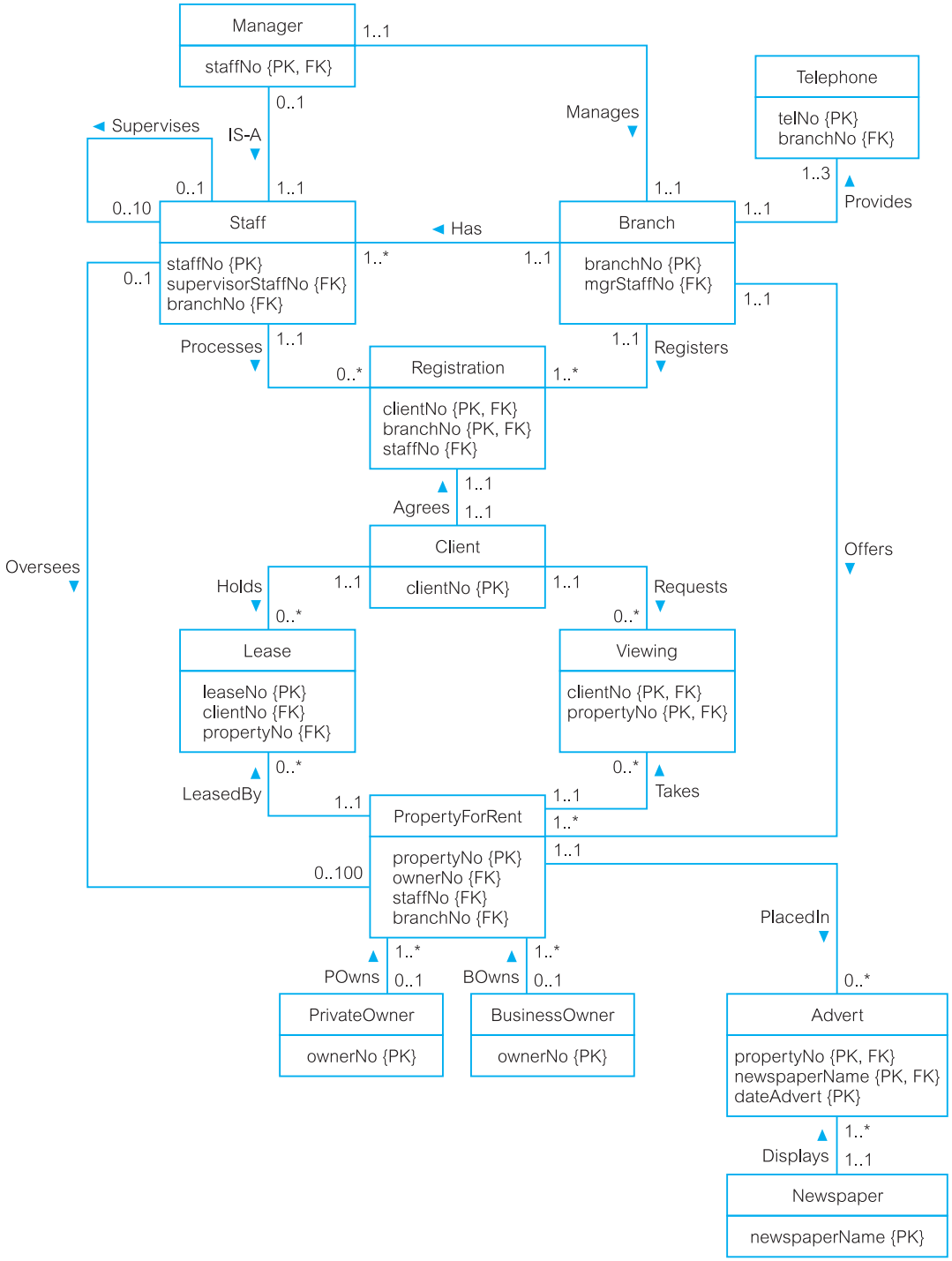


Figure 17.9 Global relation diagram for *DreamHome*.

(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

support new requirements with minimal effect on existing users. Of course, this may be very difficult to achieve, as the enterprise may not know what it wants to do in the future. Even if it does, it may be prohibitively expensive both in time and money to accommodate possible future enhancements now. Therefore, it may be necessary to be selective in what is accommodated. Consequently, it is worth examining the model to check its ability to be extended with minimal impact. However, it is not necessary to incorporate any changes into the data model unless requested by the user.

At the end of Step 2 the logical data model is used as the source of information for physical database design, which is described in the following two chapters as Steps 3 to 8 of the methodology.

For readers familiar with database design, a summary of the steps of the methodology is presented in Appendix D.

Chapter Summary

- The database design methodology includes three main phases: conceptual, logical, and physical database design.
- **Logical database design** is the process of constructing a model of the data used in an enterprise based on a specific data model but independent of a particular DBMS and other physical considerations.
- A **logical data model** includes ER diagram(s), relational schema, and supporting documentation such as the data dictionary, which is produced throughout the development of the model.
- The purpose of Step 2.1 of the methodology for logical database design is to derive a **relational schema** from the conceptual data model created in Step 1.
- In Step 2.2 the relational schema is validated using the rules of normalization to ensure that each relation is structurally correct. **Normalization** is used to improve the model so that it satisfies various constraints that avoids unnecessary duplication of data. In Step 2.3 the relational schema is also validated to ensure that it supports the transactions given in the users' requirements specification.
- In Step 2.4 the integrity constraints of the logical data model are checked. **Integrity constraints** are the constraints that are to be imposed on the database to protect the database from becoming incomplete, inaccurate, or inconsistent. The main types of integrity constraints include: required data, attribute domain constraints, multiplicity, entity integrity, referential integrity, and general constraints.
- In Step 2.5 the logical data model is validated by the users.
- Step 2.6 of logical database design is an optional step and is required only if the database has multiple user views that are being managed using the view integration approach (see Section 10.5), which results in the creation of two or more local logical data models. A **local logical data model** represents the data requirements of one or more, but not all, user views of a database. In Step 2.6 these data models are merged into a **global logical data model**, which represents the requirements of all user views. This logical data model is again validated using normalization, against the required transaction, and by users.
- Logical database design concludes with Step 2.7, which includes consideration of whether the model is capable of being extended to support possible future developments. At the end of Step 2, the logical data model, which may or may not have been developed using Step 2.6, is the source of information for physical database design described as Steps 3 to 8 in Chapters 18 and 19.

Review Questions

17.1 Describe the steps used to build a logical data model.

17.2 Describe the rules for deriving relations that represent:
- (a) strong entity types;
- (b) weak entity types;
- (c) one-to-many (1:*) binary relationship types;
- (d) one-to-one (1:1) binary relationship types;
- (e) one-to-one (1:1) recursive relationship types;
- (f) superclass/subclass relationship types;
- (g) many-to-many (*:*) binary relationship types;
- (h) complex relationship types;
- (i) multi-valued attributes.

Give examples to illustrate your answers.

17.3 Discuss how the technique of normalization can be used to validate the relations derived from the conceptual data model.

17.4 Database design is quite complex and important. Discuss the role played by users during the design process.

17.5 Describe database design language (DBDL). Discuss how it is used to derive relations during the logical database design phase.

17.6 Describe the purpose of merging data models. Discuss the difference between local logical data model and global logical data model.

17.7 Why is it important to check the logical data model for future growth?

Exercises

17.8 Derive relations from the following conceptual data model shown in Figure 17.10.

The *DreamHome* case study

17.9 Create a relational schema for the Branch user view of *DreamHome* based on the conceptual data model produced in Exercise 16.13 and compare your schema with the relations listed in Figure 17.5. Justify any differences found.

The *University Accommodation Office* case study

17.10 Create and validate a logical data model from the conceptual data model for the *University Accommodation Office* case study created in Exercise 16.16.

The *EasyDrive School of Motoring* case study

17.11 Create and validate a logical data model from the conceptual data model for the *EasyDrive School of Motoring* case study created in Exercise 16.18.

The *Wellmeadows Hospital* case study

17.12 Create and validate the local logical data models for each of the local conceptual data models of the *Wellmeadows Hospital* case study identified in Exercise 16.21.

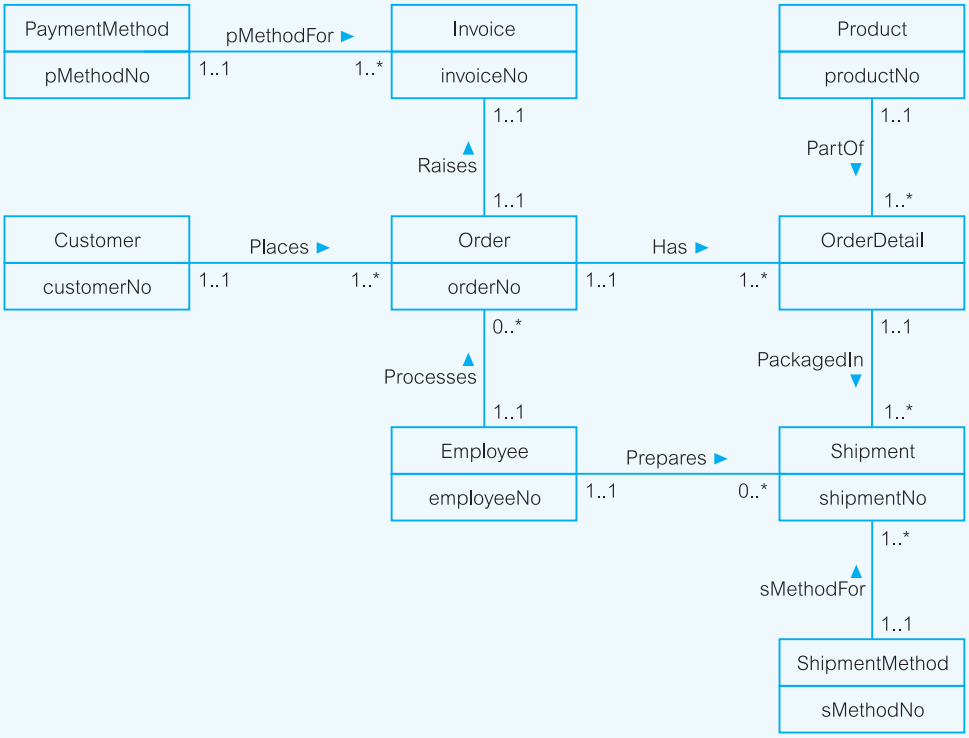


Figure 17.10 An example conceptual data model.

17.13 Merge the local data models to create a global logical data model of the *Wellmeadows Hospital* case study. State any assumptions necessary to support your design.

The *Parking Lot* case study

17.14 Present the relational schema mapped from the Parking Lot EER model shown in Figure 17.11 and described in Exercises 12.13 and 13.11.

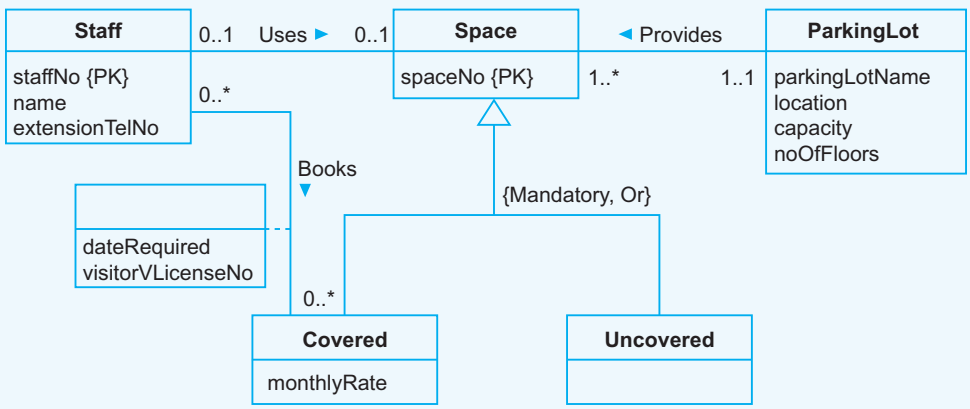


Figure 17.11 An EER model of the *Parking Lot* case study.

The *Library* case study

17.15 Describe the relational schema mapped from the Library EER model shown in Figure 17.12 and described in Exercises 12.14 and 13.12.

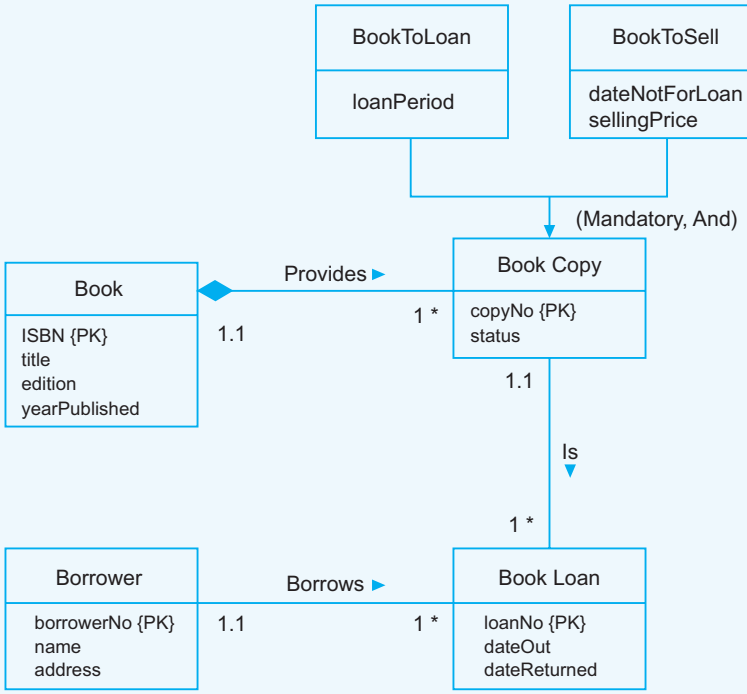


Figure 17.12 An EER model of the *Library* case study.

17.16 The ER diagram in Figure 17.13 shows only entities and primary key attributes. The absence of recognizable named entities or relationships is to emphasize the rule-based nature of the mapping rules described previously in Step 2.1 of logical database design.

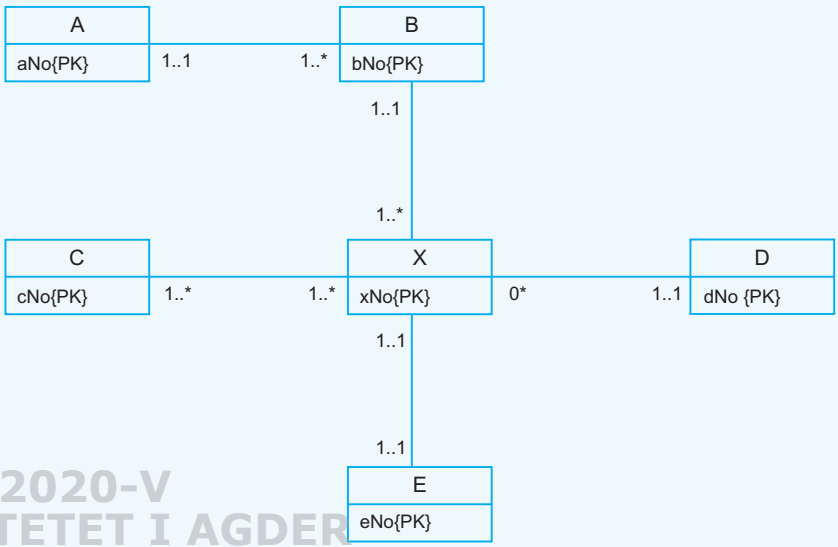


Figure 17.13 An example ER model.

Answer the following questions with reference to how the ER model in Figure 17.13 maps to relational tables.

(a) How many relations will represent the ER model?

(b) How many foreign keys are mapped to the relation representing X?

(c) Which relation(s) will have no foreign key?

(d) Using only the letter identifier for each entity, provide appropriate names for the relations mapped from the ER model.

(e) If the cardinality for each relationship is changed to *one-to-one* with total participation for all entities, how many relations would be derived from this version of the ER model?

CHAPTER

18 Methodology—Physical Database Design for Relational Databases

Chapter Objectives

In this chapter you will learn:

- The purpose of physical database design.
- How to map the logical database design to a physical database design.
- How to design base relations for the target DBMS.
- How to design general constraints for the target DBMS.
- How to select appropriate file organizations based on analysis of transactions.
- When to use secondary indexes to improve performance.
- How to estimate the size of the database.
- How to design user views.
- How to design security mechanisms to satisfy user requirements.

In this chapter and the next we describe and illustrate by example a physical database design methodology for relational databases.

The starting point for this chapter is the logical data model and the documentation that describes the model created in the conceptual/logical database design methodology described in Chapters 16 and 17. The methodology started by producing a conceptual data model in Step 1 and then derived a set of relations to produce a logical data model in Step 2. The derived relations were validated to ensure they were correctly structured using the technique of normalization described in Chapters 14 and 15, and to ensure that they supported the transactions the users require.

In the third and final phase of the database design methodology, the designer must decide how to translate the logical database design (that is, the entities, attributes, relationships, and constraints) into a physical database design that can be implemented using the target DBMS. As many parts of physical database design are highly dependent on the target DBMS, there may be more than one way of implementing any given part of the database. Consequently, to do this work properly, the designer must be fully aware of the functionality of the target DBMS, and must understand the advantages and disadvantages of each alternative approach for a particular implementation. For some systems the designer

may also need to select a suitable storage strategy that takes account of intended database usage.

Structure of this Chapter In Section 18.1 we provide a comparison of logical and physical database design. In Section 18.2 we provide an overview of the physical database design methodology and briefly describe the main activities associated with each design phase. In Section 18.3 we focus on the methodology for physical database design and present a detailed description of the first four steps required to build a physical data model. In these steps, we show how to convert the relations derived for the logical data model into a specific database implementation. We provide guidelines for choosing storage structures for the base relations and deciding when to create indexes. In places, we show physical implementation details to clarify the discussion.

In Chapter 19 we complete our presentation of the physical database design methodology and discuss how to monitor and tune the operational system. In particular, we consider when it is appropriate to denormalize the logical data model and introduce redundancy. Appendix D presents a summary of the database design methodology for those readers who are already familiar with database design and require merely an overview of the main steps.

18.1 Comparison of Logical and Physical Database Design

In presenting a database design methodology we divide the design process into three main phases: conceptual, logical, and physical database design. The phase prior to physical design—logical database design—is largely independent of implementation details, such as the specific functionality of the target DBMS and application programs, but is dependent on the target data model. The output of this process is a logical data model consisting of an ER/relation diagram, relational schema, and supporting documentation that describes this model, such as a data dictionary. Together, these represent the sources of information for the physical design process and provide the physical database designer with a vehicle for making tradeoffs that are so important to an efficient database design.

Whereas logical database design is concerned with the *what*, physical database design is concerned with the *how*. It requires different skills that are often found in different people. In particular, the physical database designer must know how the computer system hosting the DBMS operates and must be fully aware of the functionality of the target DBMS. As the functionality provided by current systems varies widely, physical design must be tailored to a specific DBMS. However, physical database design is not an isolated activity—there is often feedback between physical, logical, and application design. For example, decisions taken during physical design for improving performance, such as merging relations together, might affect the structure of the logical data model, which will have an associated effect on the application design.

18.2 Overview of the Physical Database Design Methodology

Physical database design

The process of producing a description of the implementation of the database on secondary storage; it describes the base relations, file organizations, and indexes used to achieve efficient access to the data, and any associated integrity constraints and security measures.

The steps of the physical database design methodology are as follows:

Step 3 Translate logical data model for target DBMS
- Step 3.1 Design base relations
- Step 3.2 Design representation of derived data
- Step 3.3 Design general constraints

Step 4 Design file organizations and indexes
- Step 4.1 Analyze transactions
- Step 4.2 Choose file organizations
- Step 4.3 Choose indexes
- Step 4.4 Estimate disk space requirements

Step 5 Design user views

Step 6 Design security mechanisms

Step 7 Consider the introduction of controlled redundancy

Step 8 Monitor and tune the operational system

The physical database design methodology presented in this book is divided into six main steps, numbered consecutively from 3 to follow the three steps of the conceptual and logical database design methodology. Step 3 of physical database design involves the design of the base relations and general constraints using the available functionality of the target DBMS. This step also considers how we should represent any derived data present in the data model.

Step 4 involves choosing the file organizations and indexes for the base relations. Typically, PC DBMSs have a fixed storage structure, but other DBMSs tend to provide a number of alternative file organizations for data. From the user's viewpoint, the internal storage representation for relations should be transparent—the user should be able to access relations and tuples without having to specify where or how the tuples are stored. This requires that the DBMS provides *physical data independence*, so that users are unaffected by changes to the physical structure of the database, as discussed in Section 2.1.5. The mapping between the logical data model and physical data model is defined in the internal schema, as shown in Figure 2.1. The designer may have to provide the physical design details to both the DBMS and the operating system. For the DBMS, the designer may have to specify the file organizations that are to be used to represent each relation; for the operating system, the designer must specify details such as the location and protection for each file. We recommend that the reader reviews Appendix F on file organization and storage structures before reading Step 4 of the methodology.

Step 5 involves deciding how each user view should be implemented. Step 6 involves designing the security measures necessary to protect the data from

unauthorized access, including the access controls that are required on the base relations.

Step 7 (described in Chapter 19) considers relaxing the normalization constraints imposed on the logical data model to improve the overall performance of the system. This step should be undertaken only if necessary, because of the inherent problems involved in introducing redundancy while still maintaining consistency. Step 8 (Chapter 19) is an ongoing process of monitoring the operational system to identify and resolve any performance problems resulting from the design and to implement new or changing requirements.

Appendix D presents a summary of the methodology for those readers who are already familiar with database design and require merely an overview of the main steps.

18.3 The Physical Database Design Methodology for Relational Databases

This section provides a step-by-step guide to the first four steps of the physical database design methodology for relational databases. In places, we demonstrate the close association between physical database design and implementation by describing how alternative designs can be implemented using various target DBMSs. The remaining two steps are covered in the next chapter.

Step 3: Translate Logical Data Model for Target DBMS

Objective To produce a relational database schema from the logical data model that can be implemented in the target DBMS.

The first activity of physical database design involves the translation of the relations in the logical data model into a form that can be implemented in the target relational DBMS. The first part of this process entails collating the information gathered during logical database design and documented in the data dictionary, along with the information gathered during the requirements collection and analysis stage and documented in the systems specification. The second part of the process uses this information to produce the design of the base relations. This process requires intimate knowledge of the functionality offered by the target DBMS. For example, the designer will need to know:

- how to create base relations;
- whether the system supports the definition of primary keys, foreign keys, and alternate keys;
- whether the system supports the definition of required data (that is, whether the system allows attributes to be defined as NOT NULL);
- whether the system supports the definition of domains;
- whether the system supports relational integrity constraints;
- whether the system supports the definition of general constraints.

The three activities of Step 3 are:

Step 3.1 Design base relations
Step 3.2 Design representation of derived data
Step 3.3 Design general constraints

Step 3.1: Design base relations

Objective To decide how to represent the base relations identified in the logical data model in the target DBMS.

To start the physical design process, we first collate and assimilate the information about the relations produced during logical database design. The necessary information can be obtained from the data dictionary and the definition of the relations described using the DBDL. For each relation identified in the logical data model, we have a definition consisting of:

- the name of the relation;
- a list of simple attributes in brackets;
- the primary key and, where appropriate, alternate keys (AK) and foreign keys (FK);
- referential integrity constraints for any foreign keys identified.

From the data dictionary, we also have for each attribute:

- its domain, consisting of a data type, length, and any constraints on the domain;
- an optional default value for the attribute;
- whether the attribute can hold nulls;
- whether the attribute is derived and, if so, how it should be computed.

To represent the design of the base relations, we use an extended form of the DBDL to define domains, default values, and null indicators. For example, for the PropertyForRent relation of the *DreamHome* case study, we may produce the design shown in Figure 18.1.

Implementing base relations

The next step is to decide how to implement the base relations. This decision is dependent on the target DBMS; some systems provide more facilities than others for defining base relations. We have previously demonstrated how to implement base relations using the ISO SQL standard (Section 7.1). We also show how to implement base relations using Microsoft Office Access (Appendix H.1.3), and Oracle (Appendix H.2.3).

Document design of base relations

The design of the base relations should be fully documented, along with the reasons for selecting the proposed design. In particular, document the reasons for selecting one approach when many alternatives exist.

```
    Domain PropertyNumber:          variable length character string, length 5
    Domain Street:                  variable length character string, length 25
    Domain City:                    variable length character string, length 15
    Domain Postcode:                variable length character string, length 8
    Domain PropertyType:            single character, must be one of 'B', 'C', 'D', 'E', 'F', 'H', 'M', 'S'
    Domain PropertyRooms:           integer, in the range 1–15
    Domain PropertyRent:            monetary value, in the range 0.00–9999.99
    Domain OwnerNumber:             variable length character string, length 5
    Domain StaffNumber:             variable length character string, length 5
    Domain BranchNumber:            fixed length character string, length 4
PropertyForRent(
    propertyNo      PropertyNumber      NOT NULL,
    street          Street              NOT NULL,
    city            City                NOT NULL,
    postcode        Postcode,
    type            PropertyType        NOT NULL DEFAULT 'F',
    rooms           PropertyRooms       NOT NULL DEFAULT 4,
    rent            PropertyRent        NOT NULL DEFAULT 600,
    ownerNo         OwnerNumber         NOT NULL,
    staffNo         StaffNumber,
    branchNo        BranchNumber        NOT NULL,
    PRIMARY KEY (propertyNo),
    FOREIGN KEY (staffNo) REFERENCES Staff(staffNo) ON UPDATE CASCADE ON DELETE SET NULL,
    FOREIGN KEY (ownerNo) REFERENCES PrivateOwner(ownerNo) and BusinessOwner(ownerNo)
                                    ON UPDATE CASCADE ON DELETE NO ACTION,
    FOREIGN KEY (branchNo) REFERENCES Branch(branchNo)
                                    ON UPDATE CASCADE ON DELETE NO ACTION);
```

Figure 18.1 DBDL for the PropertyForRent relation.

Step 3.2: Design representation of derived data

Objective To decide how to represent any derived data present in the logical data model in the target DBMS.

Attributes whose value can be found by examining the values of other attributes are known as **derived** or **calculated attributes**. For example, the following are all derived attributes:

- the number of staff who work in a particular branch;
- the total monthly salaries of all staff;
- the number of properties that a member of staff handles.

Often, derived attributes do not appear in the logical data model but are documented in the data dictionary. If a derived attribute is displayed in the model, a "/" is used to indicate that it is derived (see Section 12.1.2). The first step is to examine the logical data model and the data dictionary, and produce a list of all derived attributes. From a physical database design perspective, whether a derived

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

Staff

staffNo	fName	lName	branchNo	noOfProperties
SL21	John	White	B005	0
SG37	Ann	Beech	B003	2
SG14	David	Ford	B003	1
SA9	Mary	Howe	B007	1
SG5	Susan	Brand	B003	0
SL41	Julie	Lee	B005	1

Figure 18.2 The PropertyForRent relation and a simplified Staff relation with the derived attribute noOfProperties.

attribute is stored in the database or calculated every time it is needed is a tradeoff. The designer should calculate:

- the additional cost to store the derived data and keep it consistent with operational data from which it is derived;
- the cost to calculate it each time it is required.

The less expensive option is chosen subject to performance constraints. For the previous example, we could store an additional attribute in the Staff relation representing the number of properties that each member of staff currently manages. A simplified Staff relation based on the sample instance of the *DreamHome* database shown in Figure 4.3 with the new derived attribute noOfProperties is shown in Figure 18.2.

The additional storage overhead for this new derived attribute would not be particularly significant. The attribute would need to be updated every time a member of staff were assigned to or deassigned from managing a property or the property was removed from the list of available properties. In each case, the noOfProperties attribute for the appropriate member of staff would be incremented or decremented by 1. It would be necessary to ensure that this change is made consistently to maintain the correct count and thereby ensure the integrity of the database. When a query accesses this attribute, the value would be immediately available and would not have to be calculated. On the other hand, if the attribute is not stored directly in the Staff relation, it must be calculated each time it is required. This involves a join of the Staff and PropertyForRent relations. Thus, if this type of query is frequent or is considered to be critical for performance purposes, it may be more appropriate to store the derived attribute rather than calculate it each time.

It may also be more appropriate to store derived attributes whenever the DBMS's query language cannot easily cope with the algorithm to calculate the

derived attribute. For example, SQL has a limited set of aggregate functions, as we discussed in Chapter 6.

Document design of derived data

The design of derived data should be fully documented, along with the reasons for selecting the proposed design. In particular, document the reasons for selecting one approach where many alternatives exist.

Step 3.3: Design general constraints

Objective To design the general constraints for the target DBMS.

Updates to relations may be constrained by integrity constraints governing the "real-world" transactions that are represented by the updates. In Step 3.1 we designed a number of integrity constraints: required data, domain constraints, and entity and referential integrity. In this step we have to consider the remaining *general constraints*. The design of such constraints is again dependent on the choice of DBMS; some systems provide more facilities than others for defining general constraints. As in the previous step, if the system is compliant with the SQL standard, some constraints may be easy to implement. For example, *DreamHome* has a rule that prevents a member of staff from managing more than 100 properties at the same time. We could design this constraint into the SQL CREATE TABLE statement for PropertyForRent using the following clause:

```
CONSTRAINT StaffNotHandlingTooMuch
    CHECK (NOT EXISTS (SELECT staffNo
                       FROM PropertyForRent
                       GROUP BY staffNo
                       HAVING COUNT(*) > 100))
```

Alternatively, a *trigger* could be used to enforce some constraints as we illustrated in Section 8.3. In some systems there will be no support for some or all of the general constraints and it will be necessary to design the constraints into the application. For example, there are very few relational DBMSs (if any) that would be able to handle a time constraint such as "at 17.30 on the last working day of each year, archive the records for all properties sold that year and delete the associated records."

Document design of general constraints

The design of general constraints should be fully documented. In particular, document the reasons for selecting one approach where many alternatives exist.

Step 3.4: Design File Organizations and Indexes

Objective To determine the optimal file organizations to store the base relations and the indexes that are required to achieve acceptable performance, that is, the way in which relations and tuples will be held on secondary storage.

One of the main objectives of physical database design is to store and access data in an efficient way (see Appendix F). Although some storage structures are efficient for bulk loading data into the database, they may be inefficient after that. Thus, we may have to choose to use an efficient storage structure to set up the database and then choose another for operational use.

Again, the types of file organization available are dependent on the target DBMS; some systems provide more choice of storage structures than others. It is extremely important that the physical database designer fully understands the storage structures that are available, and how the target system uses these structures. This may require the designer to know how the system's query optimizer functions. For example, there may be circumstances under which the query optimizer would not use a secondary index, even if one were available. Thus, adding a secondary index would not improve the performance of the query, and the resultant overhead would be unjustified. We discuss query processing and optimization in Chapter 23.

As with logical database design, physical database design must be guided by the nature of the data and its intended use. In particular, the database designer must understand the typical *workload* that the database must support. During the requirements collection and analysis stage, there may have been requirements specified about how fast certain transactions must run or how many transactions must be processed per second. This information forms the basis for a number of decisions that will be made during this step.

With these objectives in mind, we now discuss the activities in Step 4:

Step 4.1 Analyze transactions
Step 4.2 Choose file organizations
Step 4.3 Choose indexes
Step 4.4 Estimate disk space requirements

Step 4: Design File Organizations and Indexes

Step 4.1: Analyze transactions

Objective To understand the functionality of the transactions that will run on the database and to analyze the important transactions.

To carry out physical database design effectively, it is necessary to have knowledge of the transactions or queries that will run on the database. This includes both qualitative and quantitative information. In analyzing the transactions, we attempt to identify performance criteria, such as:

- the transactions that run frequently and will have a significant impact on performance;
- the transactions that are critical to the operation of the business;
- the times during the day/week when there will be a high demand made on the database (called the *peak load*).

We use this information to identify the parts of the database that may cause performance problems. At the same time, we need to identify the high-level functionality of the transactions, such as the attributes that are updated in an update transaction or the criteria used to restrict the tuples that are retrieved in a query. We use this information to select appropriate file organizations and indexes.

In many situations, it is not possible to analyze all the expected transactions, so we should at least investigate the most "important" ones. It has been suggested that the most active 20% of user queries account for 80% of the total data access (Wiederhold, 1983). This 80/20 rule may be used as a guideline in carrying out the analysis. To help identify which transactions to investigate, we can use a *transaction relation cross-reference matrix,* which shows the relations that each transaction accesses, and/or a *transaction usage map,* which diagrammatically indicates which relations are potentially heavily used. To focus on areas that may be problematic, one way to proceed is to:

(1) map all transaction paths to relations;
(2) determine which relations are most frequently accessed by transactions;
(3) analyze the data usage of selected transactions that involve these relations.

Map all transaction paths to relations

In Steps 1.8, 2.3, and 2.6.2 of the conceptual/logical database design methodology, we validated the data models to ensure they supported the transactions that the users require by mapping the transaction paths to entities/relations. If a transaction pathway diagram was used similar to the one shown in Figure 16.9, we may be able to use this diagram to determine the relations that are most frequently accessed. On the other hand, if the transactions were validated in some other way, it may be useful to create a transaction/relation cross-reference matrix. The matrix shows, in a visual way, the transactions that are required and the relations they access. For example, Table 18.1 shows a **transaction/relation cross-reference matrix** for the following selection of typical entry, update/delete, and query trans actions for *DreamHome* (see Appendix A):

(A) Enter the details for a new property and the owner (such as details of property number PG4 in Glasgow owned by Tina Murphy).
(B) Update/delete the details of a property.
(C) Identify the total number of staff in each position at branches in Glasgow.

} StaffClient view

(D) List the property number, address, type, and rent of all properties in Glasgow, ordered by rent.
(E) List the details of properties for rent managed by a named member of staff.
(F) Identify the total number of properties assigned to each member of staff at a given branch

} Branch view

The matrix indicates, for example, that transaction (A) reads the Staff table and also inserts tuples into the PropertyForRent and PrivateOwner/BusinessOwner relations. To be more useful, the matrix should indicate in each cell the number of accesses over some time interval (for example, hourly, daily, or weekly). However, to keep the matrix simple, we do not show this information. This matrix shows that both the Staff and PropertyForRent relations are accessed by five of the six transactions, and so

TABLE 18.1 Cross-referencing transactions and relations.

TRANSACTION/ RELATION	(A)				(B)				(C)				(D)				(E)				(F)			
	I	R	U	D	I	R	U	D	I	R	U	D	I	R	U	D	I	R	U	D	I	R	U	D
Branch										X				X								X		
Telephone																								
Staff		X				X				X								X				X		
Manager																								
PrivateOwner	X																							
BusinessOwner	X																							
PropertyForRent	X					X	X	X						X				X				X		
Viewing																								
Client																								
Registration																								
Lease																								
Newspaper																								
Advert																								

I = Insert; R = Read; U = Update; D = Delete

efficient access to these relations may be important to avoid performance problems. We therefore conclude that a closer inspection of these transactions and relations are necessary.

Determine frequency information

In the requirements specification for *DreamHome* given in Section 11.4.4, it was estimated that there are about 100,000 properties for rent and 2000 staff distributed over 100 branch offices, with an average of 1000 and a maximum of 3000 properties at each branch. Figure 18.3 shows the **transaction usage map** for transactions (C), (D), (E), and (F), which all access at least one of the Staff and PropertyForRent relations, with these numbers added. Due to the size of the PropertyForRent relation, it will be important that access to this relation is as efficient as possible. We may now decide that a closer analysis of transactions involving this particular relation would be useful.

In considering each transaction, it is important to know not only the average and maximum number of times that it runs per hour, but also the day and time that the transaction is run, including when the peak load is likely. For example, some transactions may run at the average rate for most of the time, but have a peak loading between 14.00 and 16.00 on a Thursday prior to a meeting on Friday morning. Other transactions may run only at specific times—for example, 17.00–19.00 on Fridays/Saturdays, which is also their peak loading.

When transactions require frequent access to particular relations, then their patern of operation is very important. If these transactions operate in a mutually

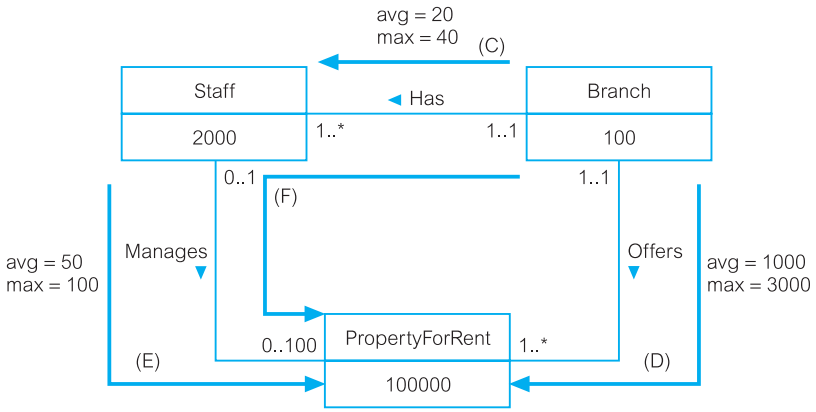


Figure 18.3
Transaction usage map for some sample transactions showing expected occurrences.

exclusive manner, the risk of likely performance problems is reduced. However, if their operating patterns conflict, potential problems may be alleviated by examining the transactions more closely to determine whether changes can be made to the structure of the relations to improve performance, as we discuss in Step 7 in the next chapter. Alternatively, it may be possible to reschedule some transactions so that their operating patterns do not conflict (for example, it may be possible to leave some summary transactions until a quieter time in the evening or overnight).

Analyze data usage

Having identified the important transactions, we now analyze each one in more detail. For each transaction, we should determine:

- The relations and attributes accessed by the transaction and the type of access; that is, whether it is an insert, update, delete, or retrieval (also known as a query) transaction.

  For an update transaction, note the attributes that are updated, as these attributes may be candidates for avoiding an access structure (such as a secondary index).

- The attributes used in any predicates (in SQL, the predicates are the conditions specified in the WHERE clause). Check to see whether the predicates involve:
  - pattern matching; for example: (name LIKE '%Smith%');
  - range searches; for example: (salary BETWEEN 10000 AND 20000);
  - exact-match key retrieval; for example: (salary = 30000).

  This applies not only to queries, but also to update and delete transactions, which can restrict the tuples to be updated/deleted in a relation.

  These attributes may be candidates for access structures.

- For a query, the attributes that are involved in the join of two or more relations.

  Again, these attributes may be candidates for access structures.

- The expected frequency at which the transaction will run; for example, the transaction will run approximately 50 times per day.

- The performance goals for the transaction; for example, the transaction must complete within 1 second.

The attributes used in any predicates for very frequent or critical transactions should have a higher priority for access structures.

Figure 18.4 shows an example of a **transaction analysis form** for transaction (D). This form shows that the average frequency of this transaction is 50 times per hour, with a peak loading of 100 times per hour daily between 17.00 and 19.00.

Transaction Analysis Form **1-Sept-2008**

Transaction (D) List the property number, address, type, and rent of all properties in Glasgow, ordered by rent

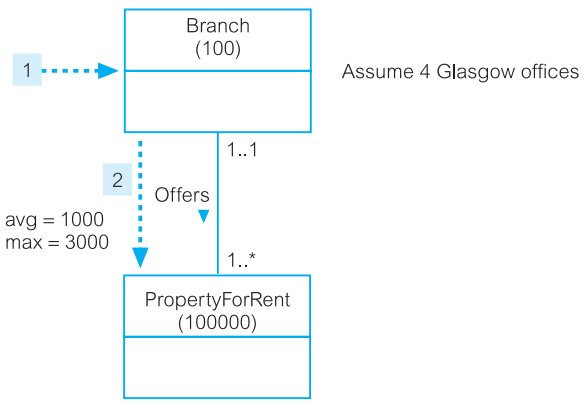
Transaction volume

Average: 50 per hour
Peak: 100 per hour (between 17.00 and 19.00 Monday–Saturday)

```
SELECT propertyNo, p.street, p.postcode, type, rent
FROM Branch b INNER JOIN PropertyForRent p ON
     b.branchNo = p.branchNo
WHERE p.city = 'Glasgow'
ORDER BY rent;
```

Predicate: p.city = 'Glasgow'
Join attributes: b.branchNo = p.branchNo
Ordering attribute: rent
Grouping attribute: none
Built-in functions: none
Attributes updated: none

Transaction usage map



Access	Entity	Type of Access	No. of References		
			Per Transaction	Avg Per Hour	Peak Per Hour
1	**Branch** (entry)	R	100	5000	10000
2	**PropertyForRent**	R	4000–12000	200000–600000	400000–1200000
Total References			**4100–12100**	**205000–605000**	**410000–1210000**

Figure 18.4 Example transaction analysis form.

In other words, typically half the branches will run this transaction per hour, and at peak time all branches will run this transaction once per hour.

The form also shows the required SQL statement and the transaction usage map. At this stage, the full SQL statement may be too detailed, but the types of details that are shown adjacent to the SQL statement should be identified, namely:

- any predicates that will be used;
- any attributes that will be required to join relations together (for a query transaction);
- attributes used to order results (for a query transaction);
- attributes used to group data together (for a query transaction);
- any built-in functions that may be used (such as AVG, SUM);
- any attributes that will be updated by the transaction.

This information will be used to determine the indexes that are required, as we discuss next. Below the transaction usage map, there is a detailed breakdown documenting:

- how each relation is accessed (reads in this case);
- how many tuples will be accessed each time the transaction is run;
- how many tuples will be accessed per hour on average and at peak loading times.

The frequency information will identify the relations that will need careful consideration to ensure that appropriate access structures are used. As mentioned previously, the search conditions used by transactions that have time constraints become higher priority for access structures.

Step 4.2: Choose file organizations

Objective To determine an efficient file organization for each base relation.

One of the main objectives of physical database design is to store and access data in an efficient way. For example, if we want to retrieve staff tuples in alphabetical order of name, sorting the file by staff name is a good file organization. However, if we want to retrieve all staff whose salary is in a certain range, searching a file ordered by staff name would not be particularly efficient. To complicate matters, some file organizations are efficient for bulk loading data into the database but inefficient after that. In other words, we may want to use an efficient storage structure to set up the database and then change it for normal operational use.

The objective of this step therefore is to choose an optimal file organization for each relation, if the target DBMS allows this. In many cases, a relational DBMS may give little or no choice for choosing file organizations, although some may be established as indexes are specified. However, as an aid to understanding file organizations and indexes more fully, we provide guidelines in Appendix F.7 for selecting a file organization based on the following types of file:

- Heap
- Hash
- Indexed Sequential Access Method (ISAM)

- B^+-tree
- Clusters

If the target DBMS does not allow the choice of file organizations, this step can be omitted.

Document choice of file organizations

The choice of file organizations should be fully documented, along with the reasons for the choice. In particular, document the reasons for selecting one approach where many alternatives exist.

Step 4.3: Choose indexes

Objective To determine whether adding indexes will improve the performance of the system.

One approach to selecting an appropriate file organization for a relation is to keep the tuples unordered and create as many **secondary indexes** as necessary. Another approach is to order the tuples in the relation by specifying a *primary* or *clustering index* (see Appendix F.5). In this case, choose the attribute for ordering or clustering the tuples as:

- the attribute that is used most often for join operations, as this makes the join operation more efficient, or
- the attribute that is used most often to access the tuples in a relation in order of that attribute.

If the ordering attribute chosen is a key of the relation, the index will be a *primary index;* if the ordering attribute is not a key, the index will be a *clustering index.* Remember that each relation can have only either a primary index or a clustering index.

Specifying indexes

We saw in Section 7.3.5 that an index can usually be created in SQL using the CREATE INDEX statement. For example, to create a primary index on the PropertyForRent relation based on the propertyNo attribute, we might use the following SQL statement:

```
CREATE UNIQUE INDEX PropertyNoInd ON PropertyForRent(propertyNo);
```

To create a clustering index on the PropertyForRent relation based on the staffNo attribute, we might use the following SQL statement:

```
CREATE INDEX StaffNoInd ON PropertyForRent(staffNo) CLUSTER;
```

As we have already mentioned, in some systems the file organization is fixed. For example, until recently Oracle has supported only B^+-trees, but has now added support for clusters. On the other hand, INGRES offers a wide set of different index structures that can be chosen using the following optional clause in the CREATE INDEX statement:

```
[STRUCTURE = BTREE | ISAM | HASH | HEAP]
```

Choosing secondary indexes

Secondary indexes provide a mechanism for specifying an additional key for a base relation that can be used to retrieve data more efficiently. For example, the PropertyForRent relation may be hashed on the property number, propertyNo, the *primary index*. However, there may be frequent access to this relation based on the rent attribute. In this case, we may decide to add rent as a *secondary index*.

There is an overhead involved in the maintenance and use of secondary indexes that has to be balanced against the performance improvement gained when retrieving data. This overhead includes:

- adding an index record to every secondary index whenever a tuple is inserted into the relation;
- updating a secondary index when the corresponding tuple in the relation is updated;
- the increase in disk space needed to store the secondary index;
- possible performance degradation during query optimization, as the query optimizer may consider all secondary indexes before selecting an optimal execution strategy.

Guidelines for choosing a "wish-list" of indexes

One approach to determining which secondary indexes are needed is to produce a *"wish-list"* of attributes that we consider to be candidates for indexing, and then to examine the impact of maintaining each of these indexes. We provide the following guidelines to help produce such a wish-list:

(1) Do not index small relations. It may be more efficient to search the relation in memory than to store an additional index structure.

(2) In general, index the primary key of a relation if it is not a key of the file organization. Although the SQL standard provides a clause for the specification of primary keys, as discussed in Section 7.2.3, it should be noted that this does not guarantee that the primary key will be indexed.

(3) Add a secondary index to a foreign key, if it is frequently accessed. For example, we may frequently join the PropertyForRent relation and the PrivateOwner/BusinessOwner relations on the attribute ownerNo, the owner number. Therefore, it may be more efficient to add a secondary index to the PropertyForRent relation based on the attribute ownerNo. Note that some DBMSs may automatically index foreign keys.

(4) Add a secondary index to any attribute that is heavily used as a secondary key (for example, add a secondary index to the PropertyForRent relation based on the attribute rent, as discussed previously).

(5) Add a secondary index on attributes that are frequently involved in:
   (a) selection or join criteria;
   (b) ORDER BY;
   (c) GROUP BY;
   (d) other operations involving sorting (such as UNION or DISTINCT).

(6) Add a secondary index on attributes involved in built-in aggregate functions, along with any attributes used for the built-in functions. For example,

to find the average staff salary at each branch, we could use the following SQL query:

```
SELECT branchNo, AVG(salary)
FROM Staff
GROUP BY branchNo;
```

From the previous guideline, we could consider adding an index to the branchNo attribute by virtue of the GROUP BY clause. However, it may be more efficient to consider an index on both the branchNo attribute and the salary attribute. This may allow the DBMS to perform the entire query from data in the index alone, without having to access the data file. This is sometimes called an **index-only plan**, as the required response can be produced using only data in the index.

(7) As a more general case of the previous guideline, add a secondary index on attributes that could result in an index-only plan.

(8) Avoid indexing an attribute or relation that is frequently updated.

(9) Avoid indexing an attribute if the query will retrieve a significant proportion (for example 25%) of the tuples in the relation. In this case, it may be more efficient to search the entire relation than to search using an index.

(10) Avoid indexing attributes that consist of long character strings.

If the search criteria involve more than one predicate, and one of the terms contains an OR clause, and the term has no index/sort order, then adding indexes for the other attributes is not going to help improve the speed of the query, because a linear search of the relation will still be required. For example, assume that only the type and rent attributes of the PropertyForRent relation are indexed, and we need to use the following query:

```
SELECT *
FROM PropertyForRent
WHERE (type = 'Flat' OR rent > 500 OR rooms > 5);
```

Although the two indexes could be used to find the tuples where (type = 'Flat' or rent > 500), the fact that the rooms attribute is not indexed will mean that these indexes cannot be used for the full WHERE clause. Thus, unless there are other queries that would benefit from having the type and rent attributes indexed, there would be no benefit gained from indexing them for this query.

On the other hand, if the predicates in the WHERE clause were AND'ed together, the two indexes on the type and rent attributes could be used to optimize the query.

Removing indexes from the wish-list

Having drawn up the wish-list of potential indexes, we should now consider the impact of each of these on update transactions. If the maintenance of the index is likely to slow down important update transactions, then consider dropping the index from the list. Note, however, that a particular index may also make update operations more efficient. For example, if we want to update a member of staff's salary, given the member's staff number, staffNo, and we have an index on staffNo, then the tuple to be updated can be found more quickly.

It is a good idea to experiment when possible to determine whether an index is improving performance, providing very little improvement, or adversely impacting performance. In the last case, clearly we should remove this index from the wish-list. If there is little observed improvement with the addition of the index, further examination may be necessary to determine under what circumstances the index will be useful, and whether these circumstances are sufficiently important to warrant the implementation of the index.

Some systems allow users to inspect the optimizer's strategy for executing a particular query or update, sometimes called the **Query Execution Plan** (QEP). For example, Microsoft Office Access has a Performance Analyzer, Oracle has an EXPLAIN PLAN diagnostic utility (see Section 23.6.3), DB2 has an EXPLAIN utility, and INGRES has an online QEP-viewing facility. When a query runs slower than expected, it is worth using such a facility to determine the reason for the slowness and to find an alternative strategy that may improve the performance of the query.

If a large number of tuples are being inserted into a relation with one or more indexes, it may be more efficient to drop the indexes first, perform the inserts, and then recreate the indexes afterwards. As a rule of thumb, if the insert will increase the size of the relation by at least 10%, drop the indexes temporarily.

Updating the database statistics

The query optimizer relies on database statistics held in the system catalog to select the optimal strategy. Whenever we create an index, the DBMS automatically adds the presence of the index to the system catalog. However, we may find that the DBMS requires a utility to be run to update the statistics in the system catalog relating to the relation and the index.

Document choice of indexes

The choice of indexes should be fully documented, along with the reasons for the choice. In particular, if there are performance reasons why some attributes should not be indexed, these should also be documented.

File organizations and indexes for *DreamHome* with Microsoft Office Access

Like most, if not all, PC DBMSs, Microsoft Office Access uses a fixed file organization, so if the target DBMS is Microsoft Office Access, Step 4.2 can be omitted. Microsoft Office Access does, however, support indexes as we will now briefly discuss. In this section we use the terminology of Office Access, which refers to a relation as a *table* with *fields* and *records*.

Guidelines for indexes In Office Access, the primary key of a table is automatically indexed, but a field whose data type is Memo, Hyperlink, or OLE Object cannot be indexed. For other fields, Microsoft advises indexing a field if all the following apply:

- the field's data type is Text, Number, Currency, or Date/Time;
- the user anticipates searching for values stored in the field;

- the user anticipates sorting values in the field;
- the user anticipates storing many different values in the field. If many of the values in the field are the same, the index may not significantly speed up queries.

In addition, Microsoft advises:

- indexing fields on both sides of a join or creating a relationship between these fields, in which case Office Access will automatically create an index on the foreign key field, if one does not exist already;
- when grouping records by the values in a joined field, specifying GROUP BY for the field that is in the same table as the field the aggregate is being calculated on.

Microsoft Office Access can optimize simple and complex predicates (which are called *expressions* in Office Access). For certain types of complex expressions, Microsoft Office Access uses a data access technology called *Rushmore* to achieve a greater level of optimization. A complex expression is formed by combining two simple expressions with the AND or OR operator, such as:

branchNo = 'BOO1' **AND** rooms > 5
type = 'Flat' **OR** rent > 300

In Office Access, a complex expression is fully or partially optimizable depending on whether one or both simple expressions are optimizable, and which operator was used to combine them. A complex expression is *Rushmore-optimizable* if all three of the following conditions are true:

- the expression uses AND or OR to join two conditions;
- both conditions are made up of simple optimizable expressions;
- both expressions contain indexed fields. The fields can be indexed individually or they can be part of a multiple-field index.

Indexes for *DreamHome* Before creating the wish-list, we ignore small tables from further consideration, as small tables can usually be processed in memory without requiring additional indexes. For *DreamHome* we ignore the Branch, Telephone, Manager, and Newspaper tables from further consideration. Based on the guidelines provided earlier:

(1) Create the primary key for each table, which will cause Office Access to automatically index this field.
(2) Ensure all relationships are created in the Relationships window, which will cause Office Access to automatically index the foreign key fields.

As an illustration of which other indexes to create, we consider the query transactions listed in Appendix A for the StaffClient user views of *DreamHome*. We can produce a summary of interactions between the base tables and these transactions shown in Table 18.2. This figure shows for each table: the transaction(s) that operate on the table, the type of access (a search based on *a predicate,* a join together with *the join field,* any *ordering field,* and any *grouping field),* and the frequency with which the transaction runs.

TABLE 18.2 Interactions between base tables and query transactions for the StaffClient user views of *DreamHome*.

TABLE	TRANSACTION	FIELD	FREQUENCY PER DAY)
Staff	(a), (d)	predicate: fName, lName	20
	(a)	join: Staff on supervisorStaffNo	20
	(b)	ordering: fName, lName	20
	(b)	predicate: position	20
Client	(e)	join: Staff on staff No	1000–000
	(j)	predicate: fName, lName	1000
PropertyForRent	(c)	predicate: rentFinish	5000–10,000
	(k), (l)	predicate: rentFinish	100
	(c)	join: PrivateOwner/BusinessOwner on ownerNo	5000–10,000
	(d)	join: Staff on staff No	20
	(f)	predicate: city	50
	(f)	predicate: rent	50
	(g)	join: Client on clientNo	100
Viewing	(i)	join: Client on clientNo	100
Lease	(c)	join: PropertyForRent on propertyNo	5000–10,000
	(l)	join: PropertyForRent on propertyNo	100
	(j)	join: Client on clientNo	1000

Based on this information, we choose to create the additional indexes shown in Table 18.3. We leave it as an exercise for the reader to choose additional indexes to create in Microsoft Office Access for the transactions listed in Appendix A for the Branch view *of DreamHome* (see Exercise 18.5).

File organizations and indexes for *DreamHome* with Oracle

In this section we repeat the previous exercise of determining appropriate file organizations and indexes for the StaffClient user views of *DreamHome*. Once again,

TABLE 18.3 Additional indexes to be created in Microsoft Office Access based on the query transactions for the StaffClient user views of *DreamHome*.

TABLE	INDEX
Staff	fName, lName
	position
Client	fName, lName
PropertyForRent	rentFinish
	city
	rent

we use the terminology of the DBMS—Oracle refers to a relation as a *table* with *columns* and *rows*.

Oracle automatically adds an index for each primary key. In addition, Oracle recommends that UNIQUE indexes not be explicitly defined on tables but instead UNIQUE integrity constraints be defined on the desired columns. Oracle enforces UNIQUE integrity constraints by automatically defining a unique index on the unique key. Exceptions to this recommendation are usually performance-related. For example, using a CREATE TABLE . . . AS SELECT with a UNIQUE constraint is slower than creating the table without the constraint and then manually creating a UNIQUE index.

Assume that the tables are created with the identified primary, alternate, and foreign keys specified. We now identify whether any clusters are required and whether any additional indexes are required. To keep the design simple, we will assume that clusters are not appropriate. Again, considering just the query transactions listed in Appendix A for the StaffClient user views of *DreamHome*, there may be performance benefits in adding the indexes shown in Table 18.4. Again, we leave it as an exercise for the reader to choose additional indexes to create in Oracle for the transactions listed in Appendix A for the Branch view of *DreamHome* (see Exercise 18.6).

Step 4.4: Estimate disk space requirements

Objective To estimate the amount of disk space that will be required by the database.

It may be a requirement that the physical database implementation can be handled by the current hardware configuration. Even if this is not the case, the designer still

TABLE 18.4 Additional indexes to be created in Oracle based on the query transactions for the StaffClient user views of *DreamHome*.

TABLE	INDEX
Staff	fName, lName
	supervisorStaffNo
	position
Client	staffNo
	fName, lName
PropertyForRent	ownerNo
	staffNo
	clientNo
	rentFinish
	city
	rent
Viewing	clientNo
Lease	propertyNo
	clientNo

has to estimate the amount of disk space that is required to store the database, in case new hardware has to be procured. The objective of this step is to estimate the amount of disk space that is required to support the database implementation on secondary storage. As with the previous steps, estimating the disk usage is highly dependent on the target DBMS and the hardware used to support the database. In general, the estimate is based on the size of each tuple and the number of tuples in the relation. The latter estimate should be a maximum number, but it may also be worth considering how the relation will grow and modifying the resulting disk size by this growth factor to determine the potential size of the database in the future. In Appendix J (see companion Web site), we illustrate the process for estimating the size of relations created in Oracle.

Step 5: Design User Views

Objective To design the user views that were identified during the requirements collection and analysis stage of the database system development lifecycle.

The first phase of the database design methodology presented in Chapter 16 involved the production of a conceptual data model for either the single user view or a number of combined user views identified during the requirements collection and analysis stage. In Section 11.4.4 we identified four user views for *DreamHome* named Director, Manager, Supervisor, Assistant, and Client. Following an analysis of the data requirements for these user views, we used the centralized approach to merge the requirements for the user views as follows:

- **Branch**, consisting of the Director and Manager user views;
- **StaffClient**, consisting of the Supervisor, Assistant, and Client user views.

In Step 2 the conceptual data model was mapped to a logical data model based on the relational model. The objective of this step is to design the user views identified previously. In a standalone DBMS on a PC, user views are usually a convenience to simplify database requests. However, in a multi-user DBMS, user views play a central role in defining the structure of the database and enforcing security. In Section 7.4.7, we discussed the major advantages of views, such as data independence, reduced complexity, and customization. We previously discussed how to create views using the ISO SQL standard (Section 7.4.1), and how to create views (stored queries) in Microsoft Office Access (Appendix M; see companion Web site).

Document design of user views

The design of the individual user views should be fully documented.

Step 6: Design Security Mechanisms

Objective To design the security mechanisms for the database as specified by the users during the requirements and collection stage of the database system development lifecycle.

A database represents an essential corporate resource and so security of this resource is extremely important. During the requirements collection and analysis stage of the database system development lifecycle, specific security requirements should have been documented in the system requirements specification (see Section 11.4.4). The objective of this step is to decide how these security requirements will be realized. Some systems offer different security facilities than others. Again, the database designer must be aware of the facilities offered by the target DBMS. As we discuss in Chapter 20, relational DBMSs generally provide two types of database security:

- system security;
- data security.

System security covers access and use of the database at the system level, such as a user name and password. **Data security** covers access and use of database objects (such as relations and views) and the actions that users can have on the objects. Again, the design of access rules is dependent on the target DBMS; some systems provide more facilities than others for designing access rules. We have previously discussed how to create access rules using the discretionary GRANT and REVOKE statements of the ISO SQL standard (Section 7.6). We also show how to create access rules using Microsoft Office Access (Appendix H.1.9), and Oracle (Appendix H.2.5). We discuss security more fully in Chapter 20.

Document design of security measures

The design of the security measures should be fully documented. If the physical design affects the logical data model, this model should also be updated.

Chapter Summary

- **Physical database design** is the process of producing a description of the implementation of the database on secondary storage. It describes the base relations and the storage structures and access methods used to access the data effectively, along with any associated integrity constraints and security measures. The design of the base relations can be undertaken only once the designer is fully aware of the facilities offered by the target DBMS.
- The initial step (Step 3) of physical database design is the translation of the logical data model into a form that can be implemented in the target relational DBMS.
- The next step (Step 4) designs the file organizations and access methods that will be used to store the base relations. This involves analyzing the transactions that will run on the database, choosing suitable file organizations based on this analysis, choosing indexes and, finally, estimating the disk space that will be required by the implementation.
- **Secondary indexes** provide a mechanism for specifying an additional key for a base relation that can be used to retrieve data more efficiently. However, there is an overhead involved in the maintenance and use of secondary indexes that has to be balanced against the performance improvement gained when retrieving data.
- One approach to selecting an appropriate file organization for a relation is to keep the tuples unordered and create as many secondary indexes as necessary. Another approach is to order the tuples in the relation by specifying a primary or clustering index. One approach to determining which secondary indexes are needed is to produce a "wish-list" of attributes that we consider are candidates for indexing, and then to examine the impact of maintaining each of these indexes.

- The objective of Step 5 is to design an implementation of the user views identified during the requirements collection and analysis stage, such as using the mechanisms provided by SQL.
- A database represents an essential corporate resource, so security of this resource is extremely important. The objective of Step 6 is to design the realization of the security mechanisms identified during the requirements collection and analysis stage.

Review Questions

18.1 Physical database design depends much on the logical and conceptual design. Discuss the validity of this statement.

18.2 Discuss why it is important to analyze transactions before implementing the index.

18.3 Describe the purpose of the main steps in the physical design methodology presented in this chapter.

18.4 Describe the 80/20 rule and how it is used in the physical database model.

Exercises

The *DreamHome* case study

18.5 In Step 4.3 we chose the indexes to create in Microsoft Office Access for the query transactions listed in Appendix A for the StaffClient user views of *DreamHome*. Choose indexes to create in Microsoft Office Access for the query transactions listed in Appendix A for the Branch user views of *DreamHome*.

18.6 Repeat Exercise 18.5 using Oracle as the target DBMS.

18.7 Create a physical database design for the logical design of the *DreamHome* case study (described in Chapter 17) based on the DBMS that you have access to.

18.8 Implement this physical design for *DreamHome* created in Exercise 18.7.

The *University Accommodation Office* case study

18.9 Based on the logical data model developed in Exercise 17.10, create a physical database design for the *University Accommodation Office* case study (described in Appendix B.1) based on the DBMS that you have access to.

18.10 Implement the *University Accommodation Office* database using the physical design created in Exercise 18.9.

The *EasyDrive School of Motoring* case study

18.11 Based on the logical data model developed in Exercise 17.11, create a physical database design for the *EasyDrive School of Motoring* case study (described in Appendix B.2) based on the DBMS that you have access to.

18.12 Based on the physical design you just created for *EasyDrive School of Motoring*, describe critical performance challenges that can be foreseen.

The *Wellmeadows Hospital* case study

18.13 Based on the logical data model developed in Exercise 17.13, create a physical database design for the *Wellmeadows Hospital* case study (described in Appendix B.3) based on the DBMS that you have access to.

18.14 Implement the *Wellmeadows Hospital* database using the physical design created in Exercise 18.13.

CHAPTER

19 Methodology—Monitoring and Tuning the Operational System

Chapter Objectives

In this chapter you will learn:

- The meaning of denormalization.
- When to denormalize to improve performance.
- The importance of monitoring and tuning the operational system.
- How to measure efficiency.
- How system resources affect performance.

In the previous chapter we presented the first five steps of the physical database design methodology for relational databases. In this chapter we describe and illustrate by example the final two steps of the physical database design methodology. We provide guidelines for determining when to denormalize the logical data model and introduce redundancy, and then discuss the importance of monitoring the operational system and continuing to tune it. In places, we show physical implementation details to clarify the discussion.

19.1 Denormalizing and Introducing Controlled Redundancy

Step 7: Consider the Introduction of Controlled Redundancy

Objective To determine whether introducing redundancy in a controlled manner by relaxing the normalization rules will improve the performance of the system.

As we discussed in Chapter 14 and 15, normalization is a technique for deciding which attributes belong together in a relation. One of the basic aims of relational database design is to group attributes together in a relation because there is a functional dependency between them. The result of normalization is a logical database design that is structurally consistent and has minimal redundancy. However, it is