Part

3 Database Analysis and Design Techniques

Chapter 9

Database Planning, Design, and Administration

Chapter Objectives

In this chapter you will learn:

- The main components of an information system.
- The main stages of the database system development lifecycle (DSDLC).
- The main phases of database design: conceptual, logical, and physical design.
- The benefits of Computer-Aided Software Engineering (CASE) tools.
- The types of criteria used to evaluate a DBMS.
- How to evaluate and select a DBMS.
- The distinction between data administration and database administration.
- The purpose and tasks associated with data administration and database administration.

Software has now surpassed hardware as the key to the success of many computer-based systems. Unfortunately, the track record at developing software is not particularly impressive. The last few decades have seen the proliferation of software applications ranging from small, relatively simple applications consisting of a few lines of code, to large, complex applications consisting of millions of lines of code. Many of these applications have required constant maintenance. This involved correcting faults that had been detected, implementing new user requirements, and modifying the software to run on new or upgraded platforms. The effort spent on maintenance began to absorb resources at an alarming rate. As a result, many major software projects were late, over budget, unreliable, difficult to maintain, and performed poorly. This led to what has become known as the **software crisis**. Although this term was first used in the late 1960s, more than 40 years later the crisis is still with us. As a result, some authors now refer to the software crisis as the **software depression**. As an indication of the crisis, a study carried out in the UK by OASIG, a Special Interest Group concerned with the Organizational Aspects of IT, reached the following conclusions about software projects (OASIG, 1996):

- 80–90% do not meet their performance goals;
- about 80% are delivered late and over budget;
- around 40% fail or are abandoned;
- under 40% fully address training and skills requirements;
- less than 25% properly integrate enterprise and technology objectives;
- just 10–20% meet all their success criteria.

There are several major reasons for the failure of software projects including:

- lack of a complete requirements specification;
- lack of an appropriate development methodology;
- poor decomposition of design into manageable components.

As a solution to these problems, a structured approach to the development of software was proposed called the **Information Systems Lifecycle** (ISLC) or the **Software Development Lifecycle** (SDLC). However, when the software being developed is a database system the lifecycle is more specifically referred to as the Database System Development Lifecycle (DSDLC).

Structure of this Chapter

In Section 9.1 we briefly describe the information systems lifecycle and discuss how this lifecycle relates to the database system development lifecycle. In Section 9.2 we present an overview of the stages of the database system development lifecycle. In Sections 9.3 to 9.13 we describe each stage of the lifecycle in more detail. In Section 9.14 we discuss how Computer-Aided Software Engineering (CASE) tools can provide support for the database system development lifecycle. We conclude in Section 9.15 with a discussion on the purpose and tasks associated with data administration and database administration within an organization.

9.1 The Information Systems Lifecycle

Information system	The resources that enable the collection, management, control, and dissemination of information throughout an organization.

Since the 1970s, database systems have been gradually replacing file-based systems as part of an organization's Information Systems (IS) infrastructure. At the same time there has been a growing recognition that data is an important corporate resource that should be treated with respect, like all other organizational resources. This resulted in many organizations establishing whole departments or functional areas called Data Administration (DA) and Database Administration (DBA), which are responsible for the management and control of the corporate data and the corporate database, respectively.

A computer-based information system includes a database, database software, application software, computer hardware, and personnel using and developing the system.

The database is a fundamental component of an information system, and its development and usage should be viewed from the perspective of the wider requirements of the organization. Therefore, the lifecycle of an organization's information system is inherently linked to the lifecycle of the database system that supports it. Typically, the stages in the lifecycle of an information system include: planning, requirements collection and analysis, design, prototyping, implementation, testing, conversion, and operational maintenance. In this chapter we review these stages from the perspective of developing a database system. However, it is important to note that the development of a database system should also be viewed from the broader perspective of developing a component part of the larger organization-wide information system.

Throughout this chapter we use the terms 'functional area' and 'application area' to refer to particular enterprise activities within an organization such as marketing, personnel, and stock control.

9.2 The Database System Development Lifecycle

As a database system is a fundamental component of the larger organization-wide information system, the database system development lifecycle is inherently associated with the lifecycle of the information system. The stages of the database system development lifecycle are shown in Figure 9.1. Below the name of each stage is the section in this chapter that describes that stage.

It is important to recognize that the stages of the database system development lifecycle are not strictly sequential, but involve some amount of repetition of previous stages through *feedback loops*. For example, problems encountered during database design may necessitate additional requirements collection and analysis. As there are feedback loops between most stages, we show only some of the more obvious ones in Figure 9.1. A summary of the main activities associated with each stage of the database system development lifecycle is described in Table 9.1.

For small database systems, with a small number of users, the lifecycle need not be very complex. However, when designing a medium to large database systems with tens to thousands of users, using hundreds of queries and application programs, the lifecycle can become extremely complex. Throughout this chapter we concentrate on activities associated with the development of medium to large database systems. In the following sections we describe the main activities associated with each stage of the database system development lifecycle in more detail.

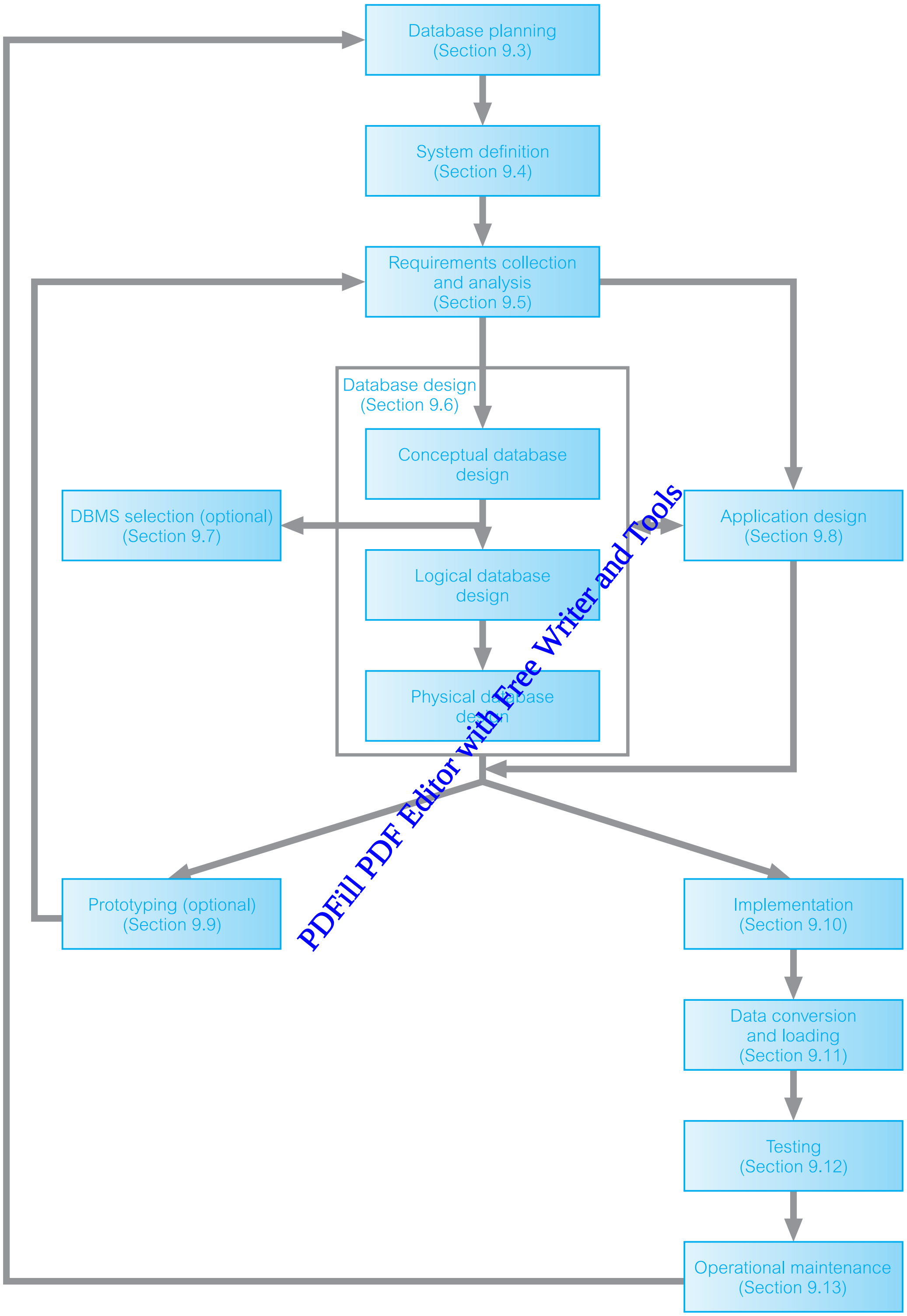

Database planning (Section 9.3)
System definition (Section 9.4)
Requirements collection and analysis (Section 9.5)
Database design (Section 9.6)
Conceptual database design
DBMS selection (optional) (Section 9.7)
Application design (Section 9.8)
Logical database design
Physical database design
Prototyping (optional) (Section 9.9)
Implementation (Section 9.10)
Data conversion and loading (Section 9.11)
Testing (Section 9.12)
Operational maintenance (Section 9.13)

Figure 9.1 The stages of the database system development lifecycle.

Table 9.1 Summary of the main activities associated with each stage of the database system development lifecycle.

Stage	Main activities
Database planning	Planning how the stages of the lifecycle can be realized most efficiently and effectively.
System definition	Specifying the scope and boundaries of the database system, including the major user views, its users, and application areas.
Requirements collection and analysis	Collection and analysis of the requirements for the new database system.
Database design	Conceptual, logical, and physical design of the database.
DBMS selection (optional)	Selecting a suitable DBMS for the database system.
Application design	Designing the user interface and the application programs that use and process the database.
Prototyping (optional)	Building a working model of the database system, which allows the designers or users to visualize and evaluate how the final system will look and function.
Implementation	Creating the physical database definitions and the application programs.
Data conversion and loading	Loading data from the old system to the new system and, where possible, converting any existing applications to run on the new database.
Testing	Database system is tested for errors and validated against the requirements specified by the users.
Operational maintenance	Database system is fully implemented. The system is continuously monitored and maintained. When necessary, new requirements are incorporated into the database system through the preceding stages of the lifecycle.

9.3 Database Planning

Database planning The management activities that allow the stages of the database system development lifecycle to be realized as efficiently and effectively as possible.

Database planning must be integrated with the overall IS strategy of the organization. There are three main issues involved in formulating an IS strategy, which are:

- identification of enterprise plans and goals with subsequent determination of information systems needs;
- evaluation of current information systems to determine existing strengths and weaknesses;
- appraisal of IT opportunities that might yield competitive advantage.

The methodologies used to resolve these issues are outside the scope of this book; however, the interested reader is referred to Robson (1997) for a fuller discussion.

An important first step in database planning is to clearly define the **mission statement** for the database system. The mission statement defines the major aims of the database system. Those driving the database project within the organization (such as the Director and/or owner) normally define the mission statement. A mission statement helps to clarify the purpose of the database system and provide a clearer path towards the efficient and effective creation of the required database system. Once the mission statement is defined, the next activity involves identifying the **mission objectives**. Each mission objective should identify a particular task that the database system must support. The assumption is that if the database system supports the mission objectives then the mission statement should be met. The mission statement and objectives may be accompanied with some additional information that specifies, in general terms, the work to be done, the resources with which to do it, and the money to pay for it all. We demonstrate the creation of a mission statement and mission objectives for the database system of *DreamHome* in Section 10.4.2.

Database planning should also include the development of standards that govern how data will be collected, how the format should be specified, what necessary documentation will be needed, and how design and implementation should proceed. Standards can be very time-consuming to develop and maintain, requiring resources to set them up initially, and to continue maintaining them. However, a well-designed set of standards provides a basis for training staff and measuring quality control, and can ensure that work conforms to a pattern, irrespective of staff skills and experience. For example, specific rules may govern how data items can be named in the data dictionary, which in turn may prevent both redundancy and inconsistency. Any legal or enterprise requirements concerning the data should be documented, such as the stipulation that some types of data must be treated confidentially.

9.4 System Definition

System definition	Describes the scope and boundaries of the database application and the major user views.

Before attempting to design a database system, it is essential that we first identify the boundaries of the system that we are investigating and how it interfaces with other parts of the organization's information system. It is important that we include within our system boundaries not only the current users and application areas, but also future users and applications. We present a diagram that represents the scope and boundaries of the *DreamHome* database system in Figure 10.10. Included within the scope and boundary of the database system are the major user views that are to be supported by the database.

User Views 9.4.1

User view Defines what is required of a database system from the perspective of a particular job role (such as Manager or Supervisor) or enterprise application area (such as marketing, personnel, or stock control).

A database system may have one or more user views. Identifying user views is an important aspect of developing a database system because it helps to ensure that no major users of the database are forgotten when developing the requirements for the new database system. User views are also particularly helpful in the development of a relatively complex database system by allowing the requirements to be broken down into manageable pieces.

A user view defines what is required of a database system in terms of the data to be held and the transactions to be performed on the data (in other words, what the users will do with the data). The requirements of a user view may be distinct to that view or overlap with other views. Figure 9.2 is a diagrammatic representation of a database system with multiple user views (denoted user view 1 to 6). Note that whereas user views (1, 2, and 3) and (5 and 6) have overlapping requirements (shown as hatched areas), user view 4 has distinct requirements.

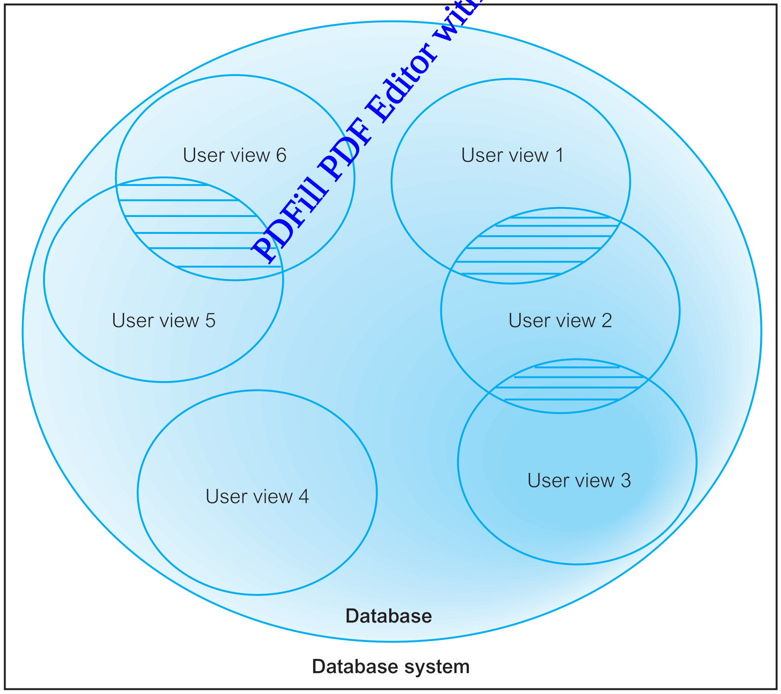


Figure 9.2 Representation of a database system with multiple user views: user views (1, 2, and 3) and (5 and 6) have overlapping requirements (shown as hatched areas), whereas user view 4 has distinct requirements.

9.5 Requirements Collection and Analysis

Requirements collection and analysis	The process of collecting and analyzing information about the part of the organization that is to be supported by the database system, and using this information to identify the requirements for the new system.

This stage involves the collection and analysis of information about the part of the enterprise to be served by the database. There are many techniques for gathering this information, called **fact-finding techniques**, which we discuss in detail in Chapter 10. Information is gathered for each major user view (that is, job role or enterprise application area), including:

- a description of the data used or generated;
- the details of how data is to be used or generated;
- any additional requirements for the new database system.

This information is then analyzed to identify the requirements (or features) to be included in the new database system. These requirements are described in documents collectively referred to as **requirements specifications** for the new database system.

Requirements collection and analysis is a preliminary stage to database design. The amount of data gathered depends on the nature of the problem and the policies of the enterprise. Too much study too soon leads to *paralysis by analysis*. Too little thought can result in an unnecessary waste of both time and money due to working on the wrong solution to the wrong problem.

The information collected at this stage may be poorly structured and include some informal requests, which must be converted into a more structured statement of requirements. This is achieved using **requirements specification techniques**, which include for example: Structured Analysis and Design (SAD) techniques, Data Flow Diagrams (DFD), and Hierarchical Input Process Output (HIPO) charts supported by documentation. As we will see shortly, Computer-Aided Software Engineering (CASE) tools may provide automated assistance to ensure that the requirements are complete and consistent. In Section 25.7 we will discuss how the Unified Modeling Language (UML) supports requirements collection and analysis.

Identifying the required functionality for a database system is a critical activity, as systems with inadequate or incomplete functionality will annoy the users, which may lead to rejection or underutilization of the system. However, excessive functionality can also be problematic as it can overcomplicate a system making it difficult to implement, maintain, use, or learn.

Another important activity associated with this stage is deciding how to deal with the situation where there is more than one user view for the database system. There are three main approaches to managing the requirements of a database system with multiple user views, namely:

- the **centralized** approach;
- the **view integration** approach;
- a combination of both approaches.

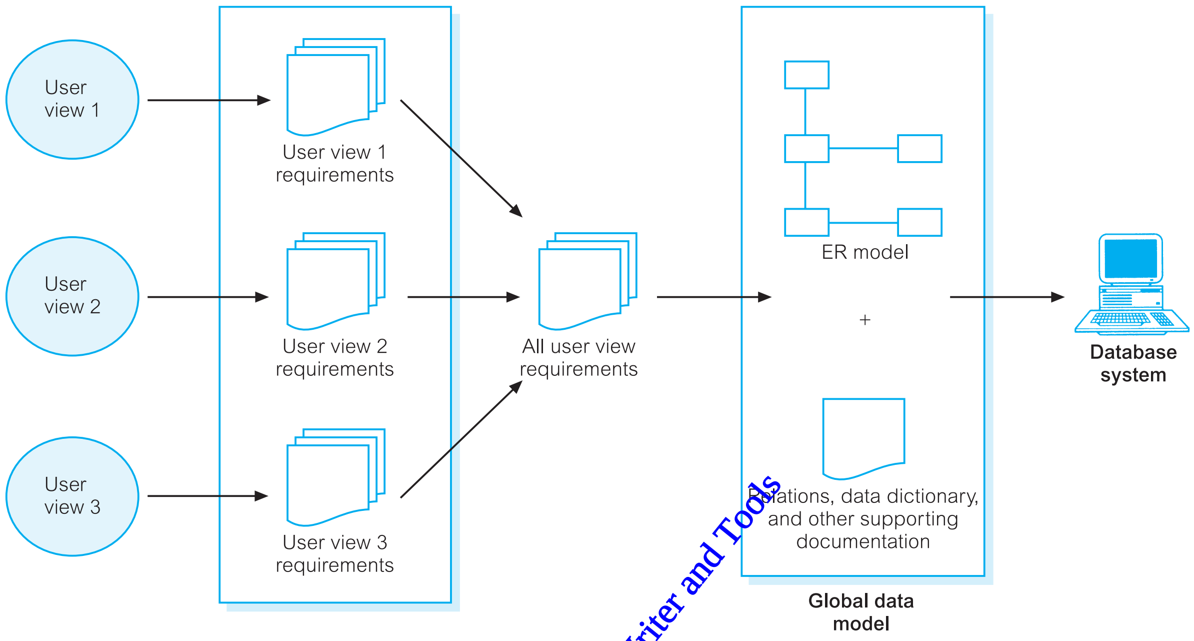


Figure 9.3 The centralized approach to managing multiple user views 1 to 3.

Centralized Approach 9.5.1

Centralized approach	Requirements for each user view are merged into a single set of requirements for the new database system. A data model representing all user views is created during the database design stage.

The centralized (or one-shot) approach involves collating the requirements for different user views into a single list of requirements. The collection of user views is given a name that provides some indication of the functional area covered by all the merged user views. In the database design stage (see Section 9.6), a global data model is created, which represents all user views. The global data model is composed of diagrams and documentation that formally describe the data requirements of the users. A diagram representing the management of user views 1 to 3 using the centralized approach is shown in Figure 9.3. Generally, this approach is preferred when there is a significant overlap in requirements for each user view and the database system is not overly complex.

View Integration Approach 9.5.2

View integration approach	Requirements for each user view remain as separate lists. Data models representing each user view are created and then merged later during the database design stage.

The view integration approach involves leaving the requirements for each user view as separate lists of requirements. In the database design stage (see Section 9.6), we first create a data model for each user view. A data model that represents a single user view (or a subset of all user views) is called a **local data model**. Each model is composed of diagrams and documentation that formally describes the requirements of one or more but not all user views of the database. The local data models are then merged at a later stage of database design to produce a **global data model**, which represents *all* user requirements for the database. A diagram representing the management of user views 1 to 3 using the view integration approach is shown in Figure 9.4. Generally, this approach is preferred

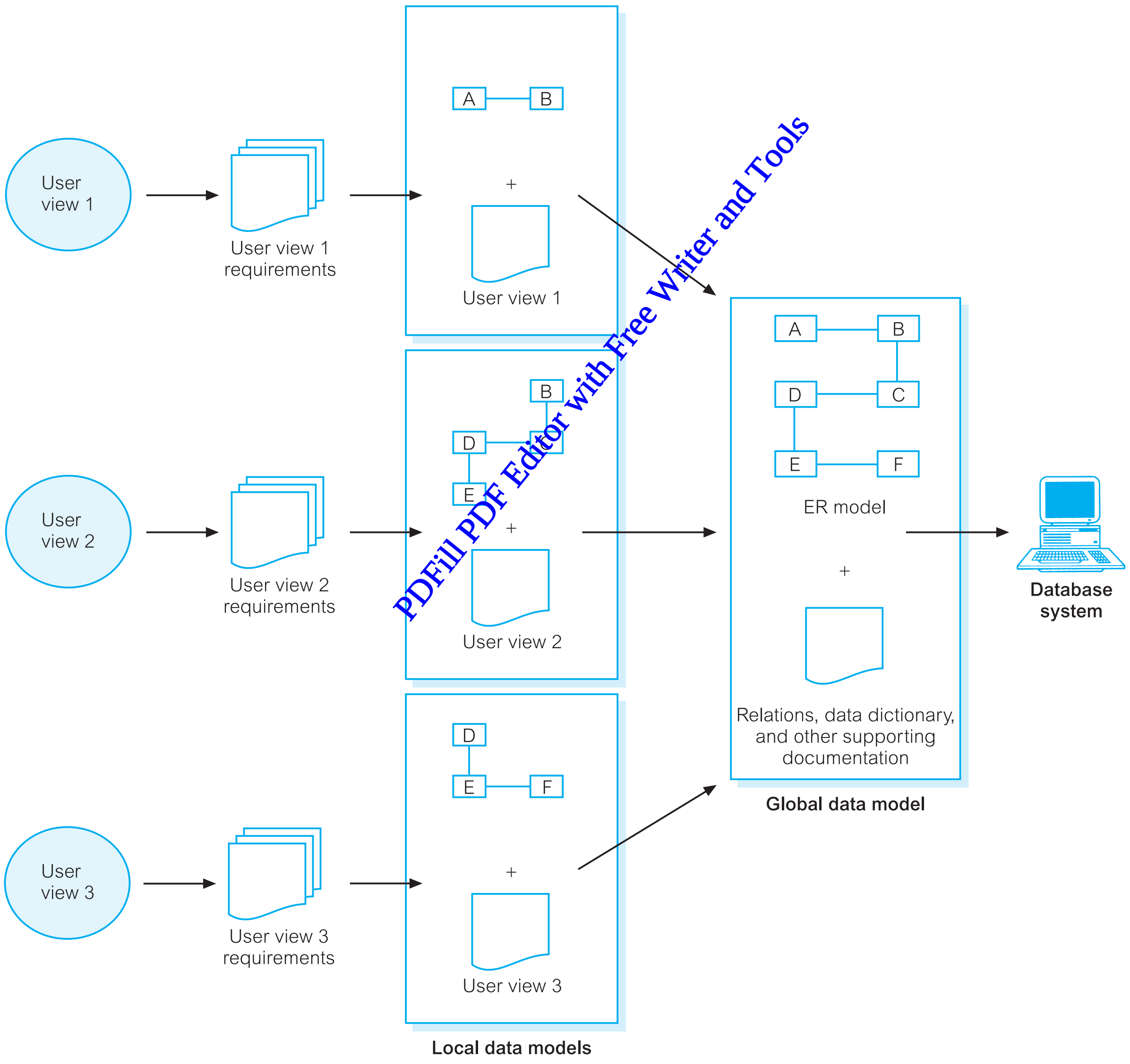


Figure 9.4 The view integration approach to managing multiple user views 1 to 3.

when there are significant differences between user views and the database system is sufficiently complex to justify dividing the work into more manageable parts. We demonstrate how to use the view integration approach in Chapter 16, Step 2.6.

For some complex database systems it may be appropriate to use a combination of both the centralized and view integration approaches to manage multiple user views. For example, the requirements for two or more user views may be first merged using the centralized approach, which is used to build a local logical data model. This model can then be merged with other local logical data models using the view integration approach to produce a global logical data model. In this case, each local logical data model represents the requirements of two or more user views and the final global logical data model represents the requirements of all user views of the database system.

We discuss how to manage multiple user views in more detail in Section 10.4.4 and using the methodology described in this book we demonstrate how to build a database for the *DreamHome* property rental case study using a combination of both the centralized and view integration approaches.

9.6 Database Design

Database design	The process of creating a design that will support the enterprise's mission statement and mission objectives for the required database system.

In this section we present an overview of the main approaches to database design. We also discuss the purpose and use of data modeling in database design. We then describe the three phases of database design, namely conceptual, logical, and physical design.

9.6.1 Approaches to Database Design

The two main approaches to the design of a database are referred to as 'bottom-up' and 'top-down'. The **bottom-up** approach begins at the fundamental level of attributes (that is, properties of entities and relationships), which through analysis of the associations between attributes, are grouped into relations that represent types of entities and relationships between entities. In Chapters 13 and 14 we discuss the process of normalization, which represents a bottom-up approach to database design. Normalization involves the identification of the required attributes and their subsequent aggregation into normalized relations based on functional dependencies between the attributes.

The bottom-up approach is appropriate for the design of simple databases with a relatively small number of attributes. However, this approach becomes difficult when applied to the design of more complex databases with a larger number of attributes, where it is difficult to establish all the functional dependencies between the attributes. As the conceptual and logical data models for complex databases may contain hundreds to thousands of attributes, it is essential to establish an approach that will simplify the design process. Also, in the initial stages of establishing the data requirements for a complex database, it may be difficult to establish all the attributes to be included in the data models.

A more appropriate strategy for the design of complex databases is to use the **top-down** approach. This approach starts with the development of data models that contain a few high-level entities and relationships and then applies successive top-down refinements to identify lower-level entities, relationships, and the associated attributes. The top-down approach is illustrated using the concepts of the Entity–Relationship (ER) model, beginning with the identification of entities and relationships between the entities, which are of interest to the organization. For example, we may begin by identifying the entities PrivateOwner and PropertyForRent, and then the relationship between these entities, PrivateOwner *Owns* PropertyForRent, and finally the associated attributes such as PrivateOwner (ownerNo, name, and address) and PropertyForRent (propertyNo and address). Building a high-level data model using the concepts of the ER model is discussed in Chapters 11 and 12.

There are other approaches to database design such as the inside-out approach and the mixed strategy approach. The **inside-out** approach is related to the bottom-up approach but differs by first identifying a set of major entities and then spreading out to consider other entities, relationships, and attributes associated with those first identified. The **mixed strategy** approach uses both the bottom-up and top-down approach for various parts of the model before finally combining all parts together.

9.6.2 Data Modeling

The two main purposes of data modeling are to assist in the understanding of the meaning (semantics) of the data and to facilitate communication about the information requirements. Building a data model requires answering questions about entities, relationships, and attributes. In doing so, the designers discover the semantics of the enterprise's data, which exist whether or not they happen to be recorded in a formal data model. Entities, relationships, and attributes are fundamental to all enterprises. However, their meaning may remain poorly understood until they have been correctly documented. A data model makes it easier to understand the meaning of the data, and thus we model data to ensure that we understand:

- each user's perspective of the data;
- the nature of the data itself, independent of its physical representations;
- the use of data across user views.

Data models can be used to convey the designer's understanding of the information requirements of the enterprise. Provided both parties are familiar with the notation used in the model, it will support communication between the users and designers. Increasingly, enterprises are standardizing the way that they model data by selecting a particular approach to data modeling and using it throughout their database development projects. The most popular high-level data model used in database design, and the one we use in this book, is based on the concepts of the Entity–Relationship (ER) model. We describe Entity–Relationship modeling in detail in Chapters 11 and 12.

Table 9.2 The criteria to produce an optimal data model.

Structural validity	Consistency with the way the enterprise defines and organizes information.
Simplicity	Ease of understanding by IS professionals and non-technical users.
Expressibility	Ability to distinguish between different data, relationships between data, and constraints.
Nonredundancy	Exclusion of extraneous information; in particular, the representation of any one piece of information exactly once.
Shareability	Not specific to any particular application or technology and thereby usable by many.
Extensibility	Ability to evolve to support new requirements with minimal effect on existing users.
Integrity	Consistency with the way the enterprise uses and manages information.
Diagrammatic representation	Ability to represent a model using an easily understood diagrammatic notation.

Criteria for data models

An *optimal* data model should satisfy the criteria listed in Table 9.2 (Fleming and Von Halle, 1989). However, sometimes these criteria are not compatible with each other and tradeoffs are sometimes necessary. For example, in attempting to achieve greater *expressibility* in a data model, we may lose *simplicity*.

9.6.3 Phases of Database Design

Database design is made up of three main phases, namely conceptual, logical, and physical design.

Conceptual database design

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

The first phase of database design is called **conceptual database design**, and involves the creation of a conceptual data model of the part of the enterprise that we are interested in modeling. The data model is built using the information documented in the users' requirements specification. Conceptual database design is entirely independent of implementation details such as the target DBMS software, application programs, programming languages, hardware platform, or any other physical considerations. In Chapter 15, we present a practical step-by-step guide on how to perform conceptual database design.

Throughout the process of developing a conceptual data model, the model is tested and validated against the users' requirements. The conceptual data model of the enterprise is a source of information for the next phase, namely logical database design.

Logical database design

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 second phase of database design is called **logical database design**, which results in the creation of a logical data model of the part of the enterprise that we interested in modeling. The conceptual data model created in the previous phase is refined and mapped on to a logical data model. The logical data model is based on the target data model for the database (for example, the relational data model).

Whereas a conceptual data model is independent of all physical considerations, a logical model is derived knowing the underlying data model of the target DBMS. In other words, we know that the DBMS is, for example, relational, network, hierarchical, or object-oriented. However, we ignore any other aspects of the chosen DBMS and, in particular, any physical details, such as storage structures or indexes.

Throughout the process of developing a logical data model, the model is tested and validated against the users' requirements. The technique of **normalization** is used to test the correctness of a logical data model. Normalization ensures that the relations derived from the data model do not display data redundancy, which can cause update anomalies when implemented. In Chapter 13 we illustrate the problems associated with data redundancy and describe the process of normalization in detail. The logical data model should also be examined to ensure that it supports the transactions specified by the users.

The logical data model is a source of information for the next phase, namely physical database design, providing the physical database designer with a vehicle for making tradeoffs that are very important to efficient database design. The logical model also serves an important role during the operational maintenance stage of the database system development lifecycle. Properly maintained and kept up to date, the data model allows future changes to application programs or data to be accurately and efficiently represented by the database.

In Chapter 16 we present a practical step-by-step guide for logical database design.

Physical database design

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.

Physical database design is the third and final phase of the database design process, during which the designer decides how the database is to be implemented. The previous phase of database design involved the development of a logical structure for the database, which describes relations and enterprise constraints. Although this structure is

DBMS-independent, it is developed in accordance with a particular data model such as the relational, network, or hierarchic. However, in developing the physical database design, we must first identify the target DBMS. Therefore, physical design is tailored to a specific DBMS system. There is feedback between physical and logical design, because decisions are taken during physical design for improving performance that may affect the structure of the logical data model.

In general, the main aim of physical database design is to describe how we intend to physically implement the logical database design. For the relational model, this involves:

- creating a set of relational tables and the constraints on these tables from the information presented in the logical data model;
- identifying the specific storage structures and access methods for the data to achieve an optimum performance for the database system;
- designing security protection for the system.

Ideally, conceptual and logical database design for larger systems should be separated from physical design for three main reasons:

- it deals with a different subject matter – the *what*, not the *how*;
- it is performed at a different time – the *what* must be understood before the *how* can be determined;
- it requires different skills, which are often found in different people.

Database design is an iterative process, which has a starting point and an almost endless procession of refinements. They should be viewed as learning processes. As the designers come to understand the workings of the enterprise and the meanings of its data, and express that understanding in the selected data models, the information gained may well necessitate changes to other parts of the design. In particular, conceptual and logical database designs are critical to the overall success of the system. If the designs are not a true representation of the enterprise, it will be difficult, if not impossible, to define all the required user views or to maintain database integrity. It may even prove difficult to define the physical implementation or to maintain acceptable system performance. On the other hand, the ability to adjust to change is one hallmark of good database design. Therefore, it is worthwhile spending the time and energy necessary to produce the best possible design.

In Chapter 2, we discussed the three-level ANSI-SPARC architecture for a database system, consisting of external, conceptual, and internal schemas. Figure 9.5 illustrates the correspondence between this architecture and conceptual, logical, and physical database design. In Chapters 17 and 18 we present a step-by-step methodology for the physical database design phase.

9.7 DBMS Selection

DBMS selection	The selection of an appropriate DBMS to support the database system.

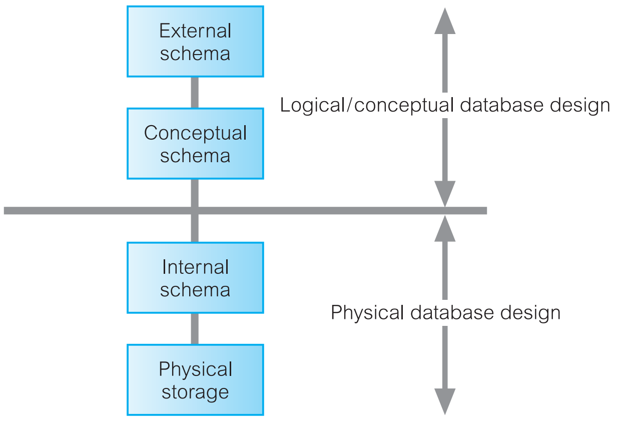


Figure 9.5 Data modeling and the ANSI-SPARC architecture.

If no DBMS exists, an appropriate part of the lifecycle in which to make a selection is between the conceptual and logical database design phases (see Figure 9.1). However, selection can be done at any time prior to logical design provided sufficient information is available regarding system requirements such as performance, ease of restructuring, security, and integrity constraints.

Although DBMS selection may be infrequent, as enterprise needs expand or existing systems are replaced, it may become necessary at times to evaluate new DBMS products. In such cases the aim is to select a system that meets the current and future requirements of the enterprise, balanced against costs that include the purchase of the DBMS product, any additional software/hardware required to support the database system, and the costs associated with changeover and staff training.

A simple approach to selection is to check off DBMS features against requirements. In selecting a new DBMS product, there is an opportunity to ensure that the selection process is well planned, and the system delivers real benefits to the enterprise. In the following section we describe a typical approach to selecting the 'best' DBMS.

9.7.1 Selecting the DBMS

The main steps to selecting a DBMS are listed in Table 9.3.

Table 9.3 Main steps to selecting a DBMS.

Define Terms of Reference of study
Shortlist two or three products
Evaluate products
Recommend selection and produce report

Define Terms of Reference of study

The Terms of Reference for the DBMS selection is established, stating the objectives and scope of the study, and the tasks that need to be undertaken. This document may also include a description of the criteria (based on the users' requirements specification) to be used to evaluate the DBMS products, a preliminary list of possible products, and all necessary constraints and timescales for the study.

Shortlist two or three products

Criteria considered to be 'critical' to a successful implementation can be used to produce a preliminary list of DBMS products for evaluation. For example, the decision to include a DBMS product may depend on the budget available, level of vendor support, compatibility with other software, and whether the product runs on particular hardware. Additional useful information on a product can be gathered by contacting existing users who may provide specific details on how good the vendor support actually is, on how the product supports particular applications, and whether or not certain hardware platforms are more problematic than others. There may also be benchmarks available that compare the performance of DBMS products. Following an initial study of the functionality and features of DBMS products, a shortlist of two or three products is identified.

The World Wide Web is an excellent source of information and can be used to identify potential candidate DBMSs. For example, the DBMS magazine's website (available at www.intelligententerprise.com) provides a comprehensive index of DBMS products. Vendors' websites can also provide valuable information on DBMS products.

Evaluate products

There are various features that can be used to evaluate a DBMS product. For the purposes of the evaluation, these features can be assessed as groups (for example, data definition) or individually (for example, data types available). Table 9.4 lists possible features for DBMS product evaluation grouped by data definition, physical definition, accessibility, transaction handling, utilities, development, and other features.

If features are checked off simply with an indication of how good or bad each is, it may be difficult to make comparisons between DBMS products. A more useful approach is to weight features and/or groups of features with respect to their importance to the organization, and to obtain an overall weighted value that can be used to compare products. Table 9.5 illustrates this type of analysis for the 'Physical definition' group for a sample DBMS product. Each selected feature is given a rating out of 10, a weighting out of 1 to indicate its importance relative to other features in the group, and a calculated score based on the rating times the weighting. For example, in Table 9.5 the feature 'Ease of reorganization' is given a rating of 4, and a weighting of 0.25, producing a score of 1.0. This feature is given the highest weighting in this table, indicating its importance in this part of the evaluation. Further, the 'Ease of reorganization' feature is weighted, for example, five times higher than the feature 'Data compression' with the lowest weighting of 0.05. Whereas, the two features 'Memory requirements' and 'Storage requirements' are given a weighting of 0.00 and are therefore not included in this evaluation.

Table 9.4 Features for DBMS evaluation.

Data definition	Physical definition
Primary key enforcement	File structures available
Foreign key specification	File structure maintenance
Data types available	Ease of reorganization
Data type extensibility	Indexing
Domain specification	Variable length fields/records
Ease of restructuring	Data compression
Integrity controls	Encryption routines
View mechanism	Memory requirements
Data dictionary	Storage requirements
Data independence	
Underlying data model	
Schema evolution	
Accessibility	**Transaction handling**
Query language: SQL2/SQL:2003/ODMG compliant	Backup and recovery routines
	Checkpointing facility
Interfacing to 3GLs	Logging facility
Multi-user	Granularity of concurrency
Security	Deadlock resolution strategy
– Access controls	Advanced transaction models
– Authorization mechanism	Parallel query processing
Utilities	**Development**
Performance measuring	4GL/5GL tools
Tuning	CASE tools
Load/unload facilities	Windows capabilities
User usage monitoring	Stored procedures, triggers, and rules
Database administration support	Web development tools
Other features	
Upgradability	Interoperability with other DBMSs and other systems
Vendor stability	Web integration
User base	Replication utilities
Training and user support	Distributed capabilities
Documentation	Portability
Operating system required	Hardware required
Cost	Network support
Online help	Object-oriented capabilities
Standards used	Architecture (2- or 3-tier client/server)
Version management	Performance
Extensibile query optimization	Transaction throughput
Scalability	Maximum number of concurrent users
Support for analytical tools	XML support

Table 9.5 Analysis of features for DBMS product evaluation.

DBMS: Sample product
Vendor: Sample vendor

Physical Definition Group

Features	Comments	Rating	Weighting	Score
File structures available	Choice of 4	8	0.15	1.2
File structure maintenance	NOT self-regulating	6	0.2	1.2
Ease of reorganization		4	0.25	1.0
Indexing		6	0.15	0.9
Variable length fields/records		6	0.15	0.9
Data compression	Specify with file structure	7	0.05	0.35
Encryption routines	Choice of 2	4	0.05	0.2
Memory requirements		0	0.00	0
Storage requirements		0	0.00	0
Totals		41	1.0	**5.75**
Physical definition group		5.75	0.25	**1.44**

We next sum together all the scores for each evaluated feature to produce a total score for the group. The score for the group is then itself subject to a weighting, to indicate its importance relative to other groups of features included in the evaluation. For example, in Table 9.5, the total score for the 'Physical definition' group is 5.75; however, this score has a weighting of 0.25.

Finally, all the weighted scores for each assessed group of features are summed to produce a single score for the DBMS product, which is compared with the scores for the other products. The product with the highest score is the 'winner'.

In addition to this type of analysis, we can also evaluate products by allowing vendors to demonstrate their product or by testing the products in-house. In-house evaluation involves creating a pilot testbed using the candidate products. Each product is tested against its ability to meet the users' requirements for the database system. Benchmarking reports published by the Transaction Processing Council can be found at www.tpc.org

Recommend selection and produce report

The final step of the DBMS selection is to document the process and to provide a statement of the findings and recommendations for a particular DBMS product.

9.8 Application Design

Application design	The design of the user interface and the application programs that use and process the database.

In Figure 9.1, observe that database and application design are parallel activities of the database system development lifecycle. In most cases, it is not possible to complete the application design until the design of the database itself has taken place. On the other hand, the database exists to support the applications, and so there must be a flow of information between application design and database design.

We must ensure that all the functionality stated in the users' requirements specification is present in the application design for the database system. This involves designing the application programs that access the database and designing the transactions, (that is, the database access methods). In addition to designing how the required functionality is to be achieved, we have to design an appropriate user interface to the database system. This interface should present the required information in a 'user-friendly' way. The importance of user interface design is sometimes ignored or left until late in the design stages. However, it should be recognized that the interface may be one of the most important components of the system. If it is easy to learn, simple to use, straightforward and forgiving, the users will be inclined to make good use of what information is presented. On the other hand, if the interface has none of these characteristics, the system will undoubtedly cause problems.

In the following sections, we briefly examine two aspects of application design, namely transaction design and user interface design.

9.8.1 Transaction Design

Before discussing transaction design we first describe what a transaction represents.

Transaction An action, or series of actions, carried out by a single user or application program, which accesses or changes the content of the database.

Transactions represent 'real world' events such as the registering of a property for rent, the addition of a new member of staff, the registration of a new client, and the renting out of a property. These transactions have to be applied to the database to ensure that data held by the database remains current with the 'real world' situation and to support the information needs of the users.

A transaction may be composed of several operations, such as the transfer of money from one account to another. However, from the user's perspective these operations still accomplish a single task. From the DBMS's perspective, a transaction transfers the database from one consistent state to another. The DBMS ensures the consistency of the database even in the presence of a failure. The DBMS also ensures that once a transaction has completed, the changes made are permanently stored in the database and cannot be lost or undone (without running another transaction to compensate for the effect of the first transaction). If the transaction cannot complete for any reason, the DBMS should ensure that the changes made by that transaction are undone. In the example of the bank transfer, if money is debited from one account and the transaction fails before crediting the other account, the DBMS should undo the debit. If we were to define the debit and credit operations as separate transactions, then once we had debited the first account and completed the transaction, we are not allowed to undo that change (without running another transaction to credit the debited account with the required amount).

The purpose of transaction design is to define and document the high-level characteristics of the transactions required on the database, including:

- data to be used by the transaction;
- functional characteristics of the transaction;
- output of the transaction;
- importance to the users;
- expected rate of usage.

This activity should be carried out early in the design process to ensure that the implemented database is capable of supporting all the required transactions. There are three main types of transactions: retrieval transactions, update transactions, and mixed transactions.

- **Retrieval transactions** are used to retrieve data for display on the screen or in the production of a report. For example, the operation to search for and display the details of a property (given the property number) is an example of a retrieval transaction.
- **Update transactions** are used to insert new records, delete old records, or modify existing records in the database. For example, the operation to insert the details of a new property into the database is an example of an update transaction.
- **Mixed transactions** involve both the retrieval and updating of data. For example, the operation to search for and display the details of a property (given the property number) and then update the value of the monthly rent is an example of a mixed transaction.

User Interface Design Guidelines 9.8.2

Before implementing a form or report, it is essential that we first design the layout. Useful guidelines to follow when designing forms or reports are listed in Table 9.6 (Shneiderman, 1992).

Meaningful title

The information conveyed by the title should clearly and unambiguously identify the purpose of the form/report.

Comprehensible instructions

Familiar terminology should be used to convey instructions to the user. The instructions should be brief, and, when more information is required, help screens should be made available. Instructions should be written in a consistent grammatical style using a standard format.

Table 9.6 Guidelines for form/report design.

Meaningful title
Comprehensible instructions
Logical grouping and sequencing of fields
Visually appealing layout of the form/report
Familiar field labels
Consistent terminology and abbreviations
Consistent use of color
Visible space and boundaries for data-entry fields
Convenient cursor movement
Error correction for individual characters and entire fields
Error messages for unacceptable values
Optional fields marked clearly
Explanatory messages for fields
Completion signal

Logical grouping and sequencing of fields

Related fields should be positioned together on the form/report. The sequencing of fields should be logical and consistent.

Visually appealing layout of the form/report

The form/report should present an attractive interface to the user. The form/report should appear balanced with fields or groups of fields evenly positioned throughout the form/report. There should not be areas of the form/report that have too few or too many fields. Fields or groups of fields should be separated by a regular amount of space. Where appropriate, fields should be vertically or horizontally aligned. In cases where a form on screen has a hardcopy equivalent, the appearance of both should be consistent.

Familiar field labels

Field labels should be familiar. For example, if Sex was replaced by Gender, it is possible that some users would be confused.

Consistent terminology and abbreviations

An agreed list of familiar terms and abbreviations should be used consistently.

Consistent use of color

Color should be used to improve the appearance of a form/report and to highlight important fields or important messages. To achieve this, color should be used in a consistent and meaningful way. For example, fields on a form with a white background may indicate data-entry fields and those with a blue background may indicate display-only fields.

Visible space and boundaries for data-entry fields

A user should be visually aware of the total amount of space available for each field. This allows a user to consider the appropriate format for the data before entering the values into a field.

Convenient cursor movement

A user should easily identify the operation required to move a cursor throughout the form/report. Simple mechanisms such as using the Tab key, arrows, or the mouse pointer should be used.

Error correction for individual characters and entire fields

A user should easily identify the operation required to make alterations to field values. Simple mechanisms should be available such as using the Backspace key or by overtyping.

Error messages for unacceptable values

If a user attempts to enter incorrect data into a field, an error message should be displayed. The message should inform the user of the error and indicate permissible values.

Optional fields marked clearly

Optional fields should be clearly identified for the user. This can be achieved using an appropriate field label or by displaying the field using a color that indicates the type of the field. Optional fields should be placed after required fields.

Explanatory messages for fields

When a user places a cursor on a field, information about the field should appear in a regular position on the screen such as a window status bar.

Completion signal

It should be clear to a user when the process of filling in fields on a form is complete. However, the option to complete the process should not be automatic as the user may wish to review the data entered.

9.9 Prototyping

At various points throughout the design process, we have the option to either fully implement the database system or build a prototype.

Prototyping Building a working model of a database system.

A prototype is a working model that does not normally have all the required features or provide all the functionality of the final system. The main purpose of developing a prototype database system is to allow users to use the prototype to identify the features of the system that work well, or are inadequate, and if possible to suggest improvements or even new features to the database system. In this way, we can greatly clarify the users' requirements for both the users and developers of the system and evaluate the feasibility of a particular system design. Prototypes should have the major advantage of being relatively inexpensive and quick to build.

There are two prototyping strategies in common use today: requirements prototyping and evolutionary prototyping. **Requirements prototyping** uses a prototype to determine the requirements of a proposed database system and once the requirements are complete the prototype is discarded. While **evolutionary prototyping** is used for the same purposes, the important difference is that the prototype is not discarded but with further development becomes the working database system.

9.10 Implementation

Implementation The physical realization of the database and application designs.

On completion of the design stages (which may or may not have involved prototyping), we are now in a position to implement the database and the application programs. The database implementation is achieved using the Data Definition Language (DDL) of the selected DBMS or a Graphical User Interface (GUI), which provides the same functionality while hiding the low-level DDL statements. The DDL statements are used to create the database structures and empty database files. Any specified user views are also implemented at this stage.

The application programs are implemented using the preferred third or fourth generation language (3GL or 4GL). Parts of these application programs are the database transactions, which are implemented using the Data Manipulation Language (DML) of the target DBMS, possibly embedded within a host programming language, such as Visual Basic (VB), VB.net, Python, Delphi, C, C++, C#, Java, COBOL, Fortran, Ada, or Pascal. We also implement the other components of the application design such as menu screens, data entry forms, and reports. Again, the target DBMS may have its own fourth generation tools that allow rapid development of applications through the provision of non-procedural query languages, reports generators, forms generators, and application generators.

Security and integrity controls for the system are also implemented. Some of these controls are implemented using the DDL, but others may need to be defined outside the DDL using, for example, the supplied DBMS utilities or operating system controls. Note that SQL (Structured Query Language) is both a DDL and a DML as described in Chapters 5 and 6.

9.11 Data Conversion and Loading

Data conversion and loading	Transferring any existing data into the new database and converting any existing applications to run on the new database.

This stage is required only when a new database system is replacing an old system. Nowadays, it is common for a DBMS to have a utility that loads existing files into the new database. The utility usually requires the specification of the source file and the target database, and then automatically converts the data to the required format of the new database files. Where applicable, it may be possible for the developer to convert and use application programs from the old system for use by the new system. Whenever conversion and loading are required, the process should be properly planned to ensure a smooth transition to full operation.

9.12 Testing

Testing	The process of running the database system with the intent of finding errors.

Before going live, the newly developed database system should be thoroughly tested. This is achieved using carefully planned test strategies and realistic data so that the entire testing process is methodically and rigorously carried out. Note that in our definition of testing we have not used the commonly held view that testing is the process of demonstrating that faults are not present. In fact, testing cannot show the absence of faults; it can show only that software faults are present. If testing is conducted successfully, it will uncover errors with the application programs and possibly the database structure. As a secondary benefit, testing demonstrates that the database and the application programs *appear* to be working according to their specification and that performance requirements appear to be satisfied. In addition, metrics collected from the testing stage provide a measure of software reliability and software quality.

As with database design, the users of the new system should be involved in the testing process. The ideal situation for system testing is to have a test database on a separate hardware system, but often this is not available. If real data is to be used, it is essential to have backups taken in case of error.

Testing should also cover usability of the database system. Ideally, an evaluation should be conducted against a usability specification. Examples of criteria that can be used to conduct the evaluation include (Sommerville, 2002):

- Learnability – How long does it take a new user to become productive with the system?
- Performance – How well does the system response match the user's work practice?
- Robustness – How tolerant is the system of user error?

- Recoverability – How good is the system at recovering from user errors?
- Adapatability – How closely is the system tied to a single model of work?

Some of these criteria may be evaluated in other stages of the lifecycle. After testing is complete, the database system is ready to be 'signed off' and handed over to the users.

9.13 Operational Maintenance

Operational maintenance The process of monitoring and maintaining the database system following installation.

In the previous stages, the database system has been fully implemented and tested. The system now moves into a maintenance stage, which involves the following activities:

- Monitoring the performance of the system. If the performance falls below an acceptable level, tuning or reorganization of the database may be required.
- Maintaining and upgrading the database system (when required). New requirements are incorporated into the database system through the preceding stages of the lifecycle.

Once the database system is fully operational, close monitoring takes place to ensure that performance remains within acceptable levels. A DBMS normally provides various utilities to aid database administration including utilities to load data into a database and to monitor the system. The utilities that allow system monitoring give information on, for example, database usage, locking efficiency (including number of deadlocks that have occurred, and so on), and query execution strategy. The Database Administrator (DBA) can use this information to tune the system to give better performance, for example, by creating additional indexes to speed up queries, by altering storage structures, or by combining or splitting tables.

The monitoring process continues throughout the life of a database system and in time may lead to reorganization of the database to satisfy the changing requirements. These changes in turn provide information on the likely evolution of the system and the future resources that may be needed. This, together with knowledge of proposed new applications, enables the DBA to engage in capacity planning and to notify or alert senior staff to adjust plans accordingly. If the DBMS lacks certain utilities, the DBA can either develop the required utilities in-house or purchase additional vendor tools, if available. We discuss database administration in more detail in Section 9.15.

When a new database application is brought online, the users should operate it in parallel with the old system for a period of time. This safeguards current operations in case of unanticipated problems with the new system. Periodic checks on data consistency between the two systems need to be made, and only when both systems appear to be producing the same results consistently, should the old system be dropped. If the changeover is too hasty, the end-result could be disastrous. Despite the foregoing assumption that the old system may be dropped, there may be situations where both systems are maintained.

CASE Tools

The first stage of the database system development lifecycle, namely database planning, may also involve the selection of suitable Computer-Aided Software Engineering (CASE) tools. In its widest sense, CASE can be applied to any tool that supports software engineering. Appropriate productivity tools are needed by data administration and database administration staff to permit the database development activities to be carried out as efficiently and effectively as possible. CASE support may include:

- a data dictionary to store information about the database system's data;
- design tools to support data analysis;
- tools to permit development of the corporate data model, and the conceptual and logical data models;
- tools to enable the prototyping of applications.

CASE tools may be divided into three categories: upper-CASE, lower-CASE, and integrated-CASE, as illustrated in Figure 9.6. **Upper-CASE** tools support the initial stages of the database system development lifecycle, from planning through to database design. **Lower-CASE** tools support the later stages of the lifecycle, from implementation through testing, to operational maintenance. **Integrated-CASE** tools support all stages of the lifecycle and thus provide the functionality of both upper- and lower-CASE in one tool.

Benefits of CASE

The use of appropriate CASE tools should improve the productivity of developing a database system. We use the term 'productivity' to relate both to the efficiency of the development process and to the effectiveness of the developed system. *Efficiency* refers to the cost, in terms of time and money, of realizing the database system. CASE tools aim to support and automate the development tasks and thus improve efficiency. *Effectiveness* refers to the extent to which the system satisfies the information needs of its users. In the pursuit of greater productivity, raising the effectiveness of the development process may be even more important than increasing its efficiency. For example, it would not be sensible to develop a database system extremely efficiently when the end-product is not what the users want. In this way, effectiveness is related to the quality of the final product. Since computers are better than humans at certain tasks, for example consistency checking, CASE tools can be used to increase the effectiveness of some tasks in the development process.

CASE tools provide the following benefits that improve productivity:

- *Standards* CASE tools help to enforce standards on a software project or across the organization. They encourage the production of standard test components that can be reused, thus simplifying maintenance and increasing productivity.
- *Integration* CASE tools store all the information generated in a repository, or data dictionary, as discussed in Section 2.7. Thus, it should be possible to store the data gathered during all stages of the database system development lifecycle. The data then can be linked together to ensure that all parts of the system are integrated. In this way,

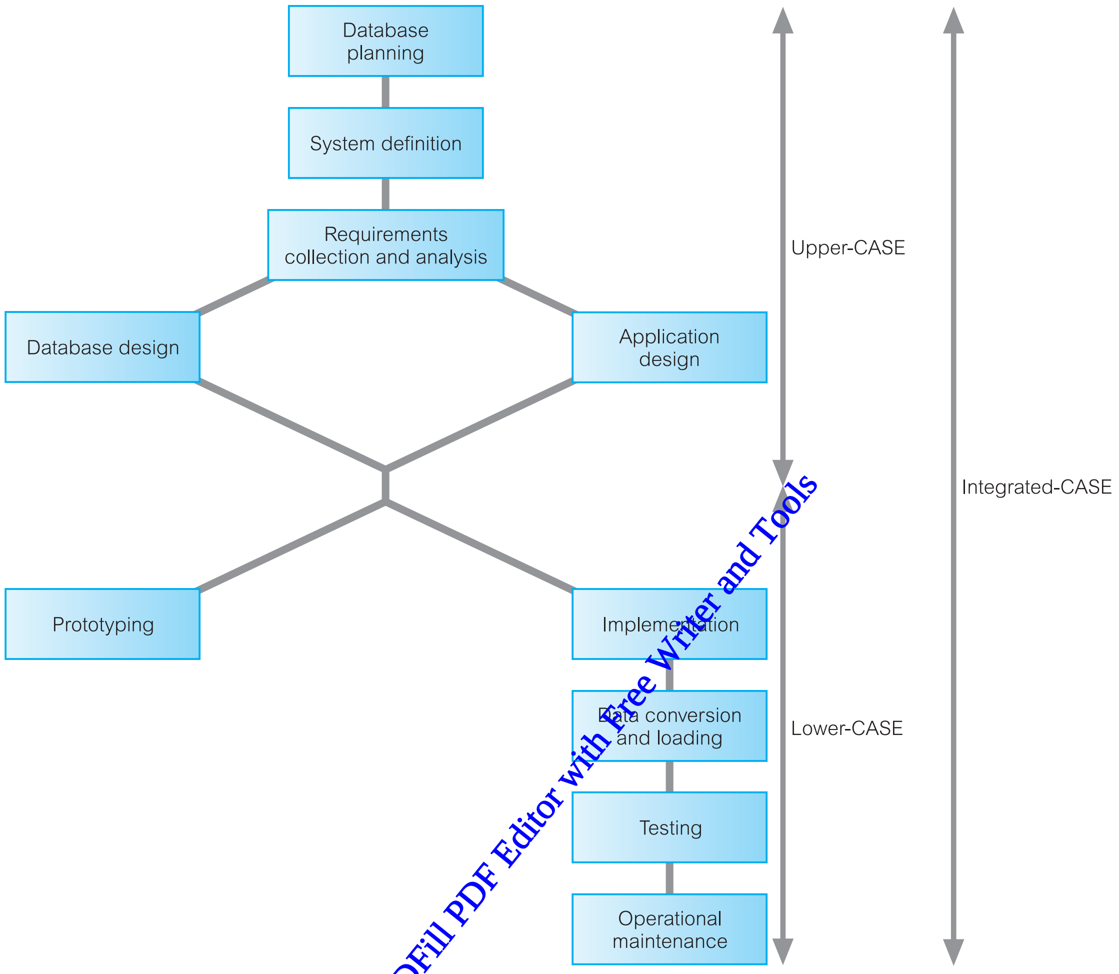


Figure 9.6 Application of CASE tools.

an organization's information system no longer has to consist of independent, unconnected components.

- *Support for standard methods* Structured techniques make significant use of diagrams, which are difficult to draw and maintain manually. CASE tools simplify this process, resulting in documentation that is correct and more current.
- *Consistency* Since all the information in the data dictionary is interrelated, CASE tools can check its consistency.
- *Automation* Some CASE tools can automatically transform parts of a design specification into executable code. This reduces the work required to produce the implemented system, and may eliminate errors that arise during the coding process.

For further information on CASE tools, the interested reader is referred to Gane (1990), Batini *et al*. (1992), and Kendall and Kendall (1995).

9.15 Data Administration and Database Administration

The Data Administrator (DA) and Database Administrator (DBA) are responsible for managing and controlling the activities associated with the corporate data and the corporate database, respectively. The DA is more concerned with the early stages of the lifecycle, from planning through to logical database design. In contrast, the DBA is more concerned with the later stages, from application/physical database design to operational maintenance. In this final section of the chapter, we discuss the purpose and tasks associated with data and database administration.

9.15.1 Data Administration

Data administration	The management of the data resource, which includes database planning, development, and maintenance of standards, policies and procedures, and conceptual and logical database design.

The Data Administrator (DA) is responsible for the corporate data resource, which includes non-computerized data, and in practice is often concerned with managing the shared data of users or application areas of an organization. The DA has the primary responsibility of consulting with and advising senior managers and ensuring that the application of database technologies continues to support corporate objectives. In some enterprises, data administration is a distinct functional area, in others it may be combined with database administration. The tasks associated with data administration are described in Table 9.7.

9.15.2 Database Administration

Database administration	The management of the physical realization of a database system, which includes physical database design and implementation, setting security and integrity controls, monitoring system performance, and reorganizing the database, as necessary.

The database administration staff are more technically oriented than the data administration staff, requiring knowledge of specific DBMSs and the operating system environment. Although the primary responsibilities are centered on developing and maintaining systems using the DBMS software to its fullest extent, DBA staff also assist DA staff in other areas, as indicated in Table 9.8. The number of staff assigned to the database administration functional area varies, and is often determined by the size of the organization. The tasks of database administration are described in Table 9.8.

Table 9.7 Data administration tasks.

Selecting appropriate productivity tools.
Assisting in the development of the corporate IT/IS and enterprise strategies.
Undertaking feasibility studies and planning for database development.
Developing a corporate data model.
Determining the organization's data requirements.
Setting data collection standards and establishing data formats.
Estimating volumes of data and likely growth.
Determining patterns and frequencies of data usage.
Determining data access requirements and safeguards for both legal and enterprise requirements.
Undertaking conceptual and logical database design.
Liaising with database administration staff and application developers to ensure applications meet all stated requirements.
Educating users on data standards and legal responsibilities.
Keeping up to date with IT/IS and enterprise developments.
Ensuring documentation is up to date and complete, including standards, policies, procedures, use of the data dictionary, and controls on end-users.
Managing the data dictionary.
Liaising with users to determine new requirements and to resolve difficulties over data access or performance.
Developing a security policy.

Table 9.8 Database administration tasks.

Evaluating and selecting DBMS products.
Undertaking physical database design.
Implementing a physical database design using a target DBMS.
Defining security and integrity constraints.
Liaising with database application developers.
Developing test strategies.
Training users.
Responsible for 'signing off' the implemented database system.
Monitoring system performance and tuning the database, as appropriate.
Performing backups routinely.
Ensuring recovery mechanisms and procedures are in place.
Ensuring documentation is complete including in-house produced material.
Keeping up to date with software and hardware developments and costs, and installing updates as necessary.

Table 9.9 Data administration and database administration – main task differences.

Data administration	Database administration
Involved in strategic IS planning	Evaluates new DBMSs
Determines long-term goals	Executes plans to achieve goals
Enforces standards, policies, and procedures	Enforces standards, policies, and procedures
Determines data requirements	Implements data requirements
Develops conceptual and logical database design	Develops logical and physical database design
Develops and maintains corporate data model	Implements physical database design
Coordinates system development	Monitors and controls database
Managerial orientation	Technical orientation
DBMS independent	DBMS dependent

Comparison of Data and Database Administration 9.15.3

The preceding sections examined the purpose and tasks associated with data administration and database administration. In this final section we briefly contrast these functional areas. Table 9.9 summarizes the *main* task differences of data administration and database administration. Perhaps the most obvious difference lies in the nature of the work carried out. Data administration staff tend to be much more managerial, whereas the database administration staff tend to be more technical.

Chapter Summary

- An **information system** is the resources that enable the collection, management, control, and dissemination of information throughout an organization.
- A computer-based information system includes the following components: database, database software, application software, computer hardware including storage media, and personnel using and developing the system.
- The database is a fundamental component of an information system, and its development and usage should be viewed from the perspective of the wider requirements of the organization. Therefore, the lifecycle of an organizational information system is inherently linked to the lifecycle of the database that supports it.
- The main stages of the **database system development lifecycle** include: database planning, system definition, requirements collection and analysis, database design, DBMS selection (optional), application design, prototyping (optional), implementation, data conversion and loading, testing, and operational maintenance.
- **Database planning** is the management activities that allow the stages of the database system development lifecycle to be realized as efficiently and effectively as possible.

- **System definition** involves identifying the scope and boundaries of the database system and user views. A **user view** defines what is required of a database system from the perspective of a particular job role (such as Manager or Supervisor) or enterprise application (such as marketing, personnel, or stock control).
- **Requirements collection and analysis** is the process of collecting and analyzing information about the part of the organization that is to be supported by the database system, and using this information to identify the requirements for the new system. There are three main approaches to managing the requirements for a database system that has multiple user views, namely the **centralized** approach, the **view integration** approach, and a combination of both approaches.
- The **centralized** approach involves merging the requirements for each user view into a single set of requirements for the new database system. A data model representing all user views is created during the database design stage. In the **view integration** approach, requirements for each user view remain as separate lists. Data models representing each user view are created then merged later during the database design stage.
- **Database design** is the process of creating a design that will support the enterprise's mission statement and mission objectives for the required database system. There are three phases of database design, namely 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.
- **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.
- **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.
- **DBMS selection** involves selecting a suitable DBMS for the database system.
- **Application design** involves user interface design and transaction design, which describes the application programs that use and process the database. A database **transaction** is an action, or series of actions, carried out by a single user or application program, which accesses or changes the content of the database.
- **Prototyping** involves building a working model of the database system, which allows the designers or users to visualize and evaluate the system.
- **Implementation** is the physical realization of the database and application designs.
- **Data conversion and loading** involves transferring any existing data into the new database and converting any existing applications to run on the new database.
- **Testing** is the process of running the database system with the intent of finding errors.
- **Operational maintenance** is the process of monitoring and maintaining the system following installation.
- **Computer-Aided Software Engineering** (CASE) applies to any tool that supports software engineering and permits the database system development activities to be carried out as efficiently and effectively as possible. CASE tools may be divided into three categories: upper-CASE, lower-CASE, and integrated-CASE.
- **Data administration** is the management of the data resource, including database planning, development and maintenance of standards, policies and procedures, and conceptual and logical database design.
- **Database administration** is the management of the physical realization of a database system, including physical database design and implementation, setting security and integrity controls, monitoring system performance, and reorganizing the database as necessary.

Review Questions

9.1 Describe the major components of an information system.

9.2 Discuss the relationship between the information systems lifecycle and the database system development lifecycle.

9.3 Describe the main purpose(s) and activities associated with each stage of the database system development lifecycle.

9.4 Discuss what a user view represents in the context of a database system.

9.5 Discuss the main approaches for managing the design of a database system that has multiple user views.

9.6 Compare and contrast the three phases of database design.

9.7 What are the main purposes of data modeling and identify the criteria for an optimal data model?

9.8 Identify the stage(s) where it is appropriate to select a DBMS and describe an approach to selecting the 'best' DBMS.

9.9 Application design involves transaction design and user interface design. Describe the purpose and main activities associated with each.

9.10 Discuss why testing cannot show the absence of faults, only that software faults are present.

9.11 Describe the main advantages of using the prototyping approach when building a database system.

9.12 Define the purpose and tasks associated with data administration and database administration.

Exercises

9.13 Assume that you are responsible for selecting a new DBMS product for a group of users in your organization. To undertake this exercise, you must first establish a set of requirements for the group and then identify a set of features that a DBMS product must provide to fulfill the requirements. Describe the process of evaluating and selecting the best DBMS product.

9.14 Describe the process of evaluating and selecting a DBMS product for each of the case studies described in Appendix B.

9.15 Investigate whether data administration and database administration exist as distinct functional areas within your organization. If identified, describe the organization, responsibilities, and tasks associated with each functional area.

Chapter 10 Fact-Finding Techniques

Chapter Objectives

In this chapter you will learn:

- When fact-finding techniques are used in the database system development lifecycle.
- The types of facts collected in each stage of the database system development lifecycle.
- The types of documentation produced in each stage of the database system development lifecycle.
- The most commonly used fact-finding techniques.
- How to use each fact-finding technique and the advantages and disadvantages of each.
- About a property rental company called *DreamHome.*
- How to apply fact-finding techniques to the early stages of the database system development lifecycle.

In Chapter 9 we introduced the stages of the database system development lifecycle. There are many occasions during these stages when it is critical that the database developer captures the necessary facts to build the required database system. The necessary facts include, for example, the terminology used within the enterprise, problems encountered using the current system, opportunities sought from the new system, necessary constraints on the data and users of the new system, and a prioritized set of requirements for the new system. These facts are captured using fact-finding techniques.

Fact-finding The formal process of using techniques such as interviews and questionnaires to collect facts about systems, requirements, and preferences.

In this chapter we discuss when a database developer might use fact-finding techniques and what types of facts should be captured. We present an overview of how these facts are used to generate the main types of documentation used throughout the database system development lifecycle. We describe the most commonly used fact-finding techniques and identify the advantages and disadvantages of each. We finally demonstrate how some of these techniques may be used during the earlier stages of the database system development lifecycle using a property management company called *DreamHome*. The *DreamHome* case study is used throughout this book.

Structure of this Chapter

In Section 10.1 we discuss when a database developer might use fact-finding techniques. (Throughout this book we use the term 'database developer' to refer to a person or group of people responsible for the analysis, design, and implementation of a database system.) In Section 10.2 we illustrate the types of facts that should be collected and the documentation that should be produced at each stage of the database system development lifecycle. In Section 10.3 we describe the five most commonly used fact-finding techniques and identify the advantages and disadvantages of each. In Section 10.4 we demonstrate how fact-finding techniques can be used to develop a database system for a case study called *DreamHome*, a property management company. We begin this section by providing an overview of the *DreamHome* case study. We then examine the first three stages of the database system development lifecycle, namely database planning, system definition, and requirements collection and analysis. For each stage we demonstrate the process of collecting data using fact-finding techniques and describe the documentation produced.

10.1 When Are Fact-Finding Techniques Used?

There are many occasions for fact-finding during the database system development lifecycle. However, fact-finding is particularly crucial to the early stages of the lifecycle including the database planning, system definition, and requirements collection and analysis stages. It is during these early stages that the database developer captures the essential facts necessary to build the required database. Fact-finding is also used during database design and the later stages of the lifecycle, but to a lesser extent. For example, during physical database design, fact-finding becomes technical as the database developer attempts to learn more about the DBMS selected for the database system. Also, during the final stage, operational maintenance, fact-finding is used to determine whether a system requires tuning to improve performance or further development to include new requirements.

Note that it is important to have a rough estimate of how much time and effort is to be spent on fact-finding for a database project. As we mentioned in Chapter 9, too much study too soon leads to *paralysis by analysis*. However, too little thought can result in an unnecessary waste of both time and money due to working on the wrong solution to the wrong problem.

10.2 What Facts Are Collected?

Throughout the database system development lifecycle, the database developer needs to capture facts about the current and/or future system. Table 10.1 provides examples of the sorts of data captured and the documentation produced for each stage of the lifecycle. As we mentioned in Chapter 9, the stages of the database system development lifecycle are

Table 10.1 Examples of the data captured and the documentation produced for each stage of the database system development lifecycle.

Stage of database system development lifecycle	Examples of data captured	Examples of documentation produced
Database planning	Aims and objectives of database project	Mission statement and objectives of database system
System definition	Description of major user views (includes job roles or business application areas)	Definition of scope and boundary of database application; definition of user views to be supported
Requirements collection and analysis	Requirements for user views; systems specifications, including performance and security requirements	Users' and system requirements specifications
Database design	Users' responses to checking the logical database design; functionality provided by target DBMS	Conceptual/logical database design (includes ER model(s), data dictionary, and relational schema); physical database design
Application design	Users' responses to checking interface design	Application design (includes description of programs and user interface)
DBMS selection	Functionality provided by target DBMS	DBMS evaluation and recommendations
Prototyping	Users' responses to prototype	Modified users' requirements and systems specifications
Implementation	Functionality provided by target DBMS	
Data conversion and loading	Format of current data; data import capabilities of target DBMS	
Testing	Test results	Testing strategies used; analysis of test results
Operational maintenance	Performance testing results; new or changing user and system requirements	User manual; analysis of performance results; modified users' requirements and systems specifications

not strictly sequential, but involve some amount of repetition of previous stages through feedback loops. This is also true for the data captured and the documentation produced at each stage. For example, problems encountered during database design may necessitate additional data capture on the requirements for the new system.

10.3 Fact-Finding Techniques

A database developer normally uses several fact-finding techniques during a single database project. There are five commonly used fact-finding techniques:

- examining documentation;
- interviewing;
- observing the enterprise in operation;
- research;
- questionnaires.

In the following sections we describe these fact-finding techniques and identify the advantages and disadvantages of each.

10.3.1 Examining Documentation

Examining documentation can be useful when we are trying to gain some insight as to how the need for a database arose. We may also find that documentation can help to provide information on the part of the enterprise associated with the problem. If the problem relates to the current system, there should be documentation associated with that system. By examining documents, forms, reports, and files associated with the current system, we can quickly gain some understanding of the system. Examples of the types of documentation that should be examined are listed in Table 10.2.

10.3.2 Interviewing

Interviewing is the most commonly used, and normally most useful, fact-finding technique. We can interview to collect information from individuals face-to-face. There can be several objectives to using interviewing, such as finding out facts, verifying facts, clarifying facts, generating enthusiasm, getting the end-user involved, identifying requirements, and gathering ideas and opinions. However, using the interviewing technique requires good communication skills for dealing effectively with people who have different values, priorities, opinions, motivations, and personalities. As with other fact-finding techniques, interviewing is not always the best method for all situations. The advantages and disadvantages of using interviewing as a fact-finding technique are listed in Table 10.3.

There are two types of interview: unstructured and structured. **Unstructured interviews** are conducted with only a general objective in mind and with few, if any, specific

Table 10.2 Examples of types of documentation that should be examined.

Purpose of documentation	Examples of useful sources
Describes problem and need for database	Internal memos, e-mails, and minutes of meetings Employee/customer complaints, and documents that describe the problem Performance reviews/reports
Describes the part of the enterprise affected by problem	Organizational chart, mission statement, and strategic plan of the enterprise Objectives for the part of the enterprise being studied Task/job descriptions Samples of completed manual forms and reports Samples of completed computerized forms and reports
Describes current system	Various types of flowcharts and diagrams Data dictionary Database system design Program documentation User/training manuals

Table 10.3 Advantages and disadvantages of using interviewing as a fact-finding technique.

Advantages	Disadvantages
Allows interviewee to respond freely and openly to questions	Very time-consuming and costly, and therefore may be impractical
Allows interviewee to feel part of project	Success is dependent on communication skills of interviewer
Allows interviewer to follow up on interesting comments made by interviewee	Success can be dependent on willingness of interviewees to participate in interviews
Allows interviewer to adapt or re-word questions during interview	
Allows interviewer to observe interviewee's body language	

questions. The interviewer counts on the interviewee to provide a framework and direction to the interview. This type of interview frequently loses focus and, for this reason, it often does not work well for database analysis and design.

In **structured interviews**, the interviewer has a specific set of questions to ask the interviewee. Depending on the interviewee's responses, the interviewer will direct additional questions to obtain clarification or expansion. **Open-ended questions** allow the interviewee to respond in any way that seems appropriate. An example of an open-ended question is: 'Why are you dissatisfied with the report on client registration?' **Closed-ended questions** restrict answers to either specific choices or short, direct responses. An example of such a question might be: 'Are you receiving the report on client registration on time?' or 'Does the report on client registration contain accurate information?' Both questions require only a 'Yes' or 'No' response.

To ensure a successful interview includes selecting appropriate individuals to interview, preparing extensively for the interview, and conducting the interview in an efficient and effective manner.

Observing the Enterprise in Operation 10.3.3

Observation is one of the most effective fact-finding techniques for understanding a system. With this technique, it is possible to either participate in, or watch, a person perform activities to learn about the system. This technique is particularly useful when the validity of data collected through other methods is in question or when the complexity of certain aspects of the system prevents a clear explanation by the end-users.

As with the other fact-finding techniques, successful observation requires preparation. To ensure that the observation is successful, it is important to know as much about the individuals and the activity to be observed as possible. For example, 'When are the low, normal, and peak periods for the activity being observed?' and 'Will the individuals be upset by having someone watch and record their actions?' The advantages and disadvantages of using observation as a fact-finding technique are listed in Table 10.4.

Research 10.3.4

A useful fact-finding technique is to research the application and problem. Computer trade journals, reference books, and the Internet (including user groups and bulletin boards) are good sources of information. They can provide information on how others have solved similar problems, plus whether or not software packages exist to solve or even partially solve the problem. The advantages and disadvantages of using research as a fact-finding technique are listed in Table 10.5.

Table 10.4 Advantages and disadvantages of using observation as a fact-finding technique.

Advantages	Disadvantages
Allows the validity of facts and data to be checked	People may knowingly or unknowingly perform differently when being observed
Observer can see exactly what is being done	May miss observing tasks involving different levels of difficulty or volume normally experienced during that time period
Observer can also obtain data describing the physical environment of the task	Some tasks may not always be performed in the manner in which they are observed
Relatively inexpensive	May be impractical
Observer can do work measurements	

Table 10.5 Advantages and disadvantages of using research as a fact-finding technique.

Advantages	Disadvantages
Can save time if solution already exists	Requires access to appropriate sources of information
Researcher can see how others have solved similar problems or met similar requirements	May ultimately not help in solving problem because problem is not documented elsewhere
Keeps researcher up to date with current developments	

10.3.5 Questionnaires

Another fact-finding technique is to conduct surveys through questionnaires. Questionnaires are special-purpose documents that allow facts to be gathered from a large number of people while maintaining some control over their responses. When dealing with a large audience, no other fact-finding technique can tabulate the same facts as efficiently. The advantages and disadvantages of using questionnaires as a fact-finding technique are listed in Table 10.6.

There are two types of questions that can be asked in a questionnaire, namely free-format and fixed-format. **Free-format questions** offer the respondent greater freedom in providing answers. A question is asked and the respondent records the answer in the space provided after the question. Examples of free-format questions are: 'What reports do you currently receive and how are they used?' and 'Are there any problems with these reports? If so, please explain.' The problems with free-format questions are that the respondent's answers may prove difficult to tabulate and, in some cases, may not match the questions asked.

Fixed-format questions require specific responses from individuals. Given any question, the respondent must choose from the available answers. This makes the results much

Table 10.6 Advantages and disadvantages of using questionnaires as a fact-finding technique.

Advantages	Disadvantages
People can complete and return questionnaires at their convenience	Number of respondents can be low, possibly only 5% to 10%
Relatively inexpensive way to gather data from a large number of people	Questionnaires may be returned incomplete
People more likely to provide the real facts as responses can be kept confidential	May not provide an opportunity to adapt or re-word questions that have been misinterpreted
Responses can be tabulated and analyzed quickly	Cannot observe and analyze the respondent's body language

easier to tabulate. On the other hand, the respondent cannot provide additional information that might prove valuable. An example of a fixed-format question is: ‘The current format of the report on property rentals is ideal and should not be changed.’ The respondent may be given the option to answer ‘Yes’ or ‘No’ to this question, or be given the option to answer from a range of responses including ‘Strongly agree’, ‘Agree’, ‘No opinion’, ‘Disagree’, and ‘Strongly disagree’.

10.4 Using Fact-Finding Techniques – A Worked Example

In this section we first present an overview of the *DreamHome* case study and then use this case study to illustrate how to establish a database project. In particular, we illustrate how fact-finding techniques can be used and the documentation produced in the early stages of the database system development lifecycle namely the database planning, system definition, and requirements collection and analysis stages.

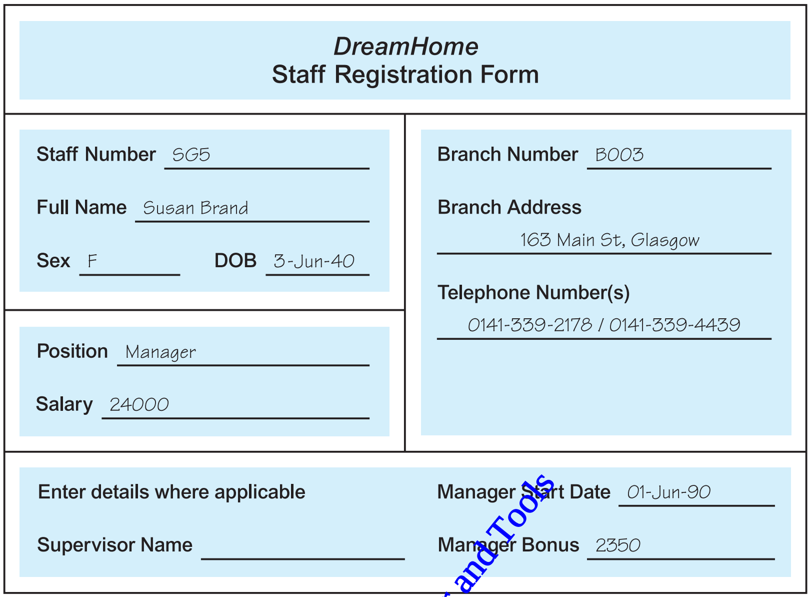
10.4.1 The *DreamHome* Case Study – An Overview

The first branch office of *DreamHome* was opened in 1992 in Glasgow in the UK. Since then, the Company has grown steadily and now has several offices in most of the main cities of the UK. However, the Company is now so large that more and more administrative staff are being employed to cope with the ever-increasing amount of paperwork. Furthermore, the communication and sharing of information between offices, even in the same city, is poor. The Director of the Company, Sally Mellweadows feels that too many mistakes are being made and that the success of the Company will be short-lived if she does not do something to remedy the situation. She knows that a database could help in part to solve the problem and requests that a database system be developed to support the running of *DreamHome*. The Director has provided the following brief description of how *DreamHome* currently operates.

DreamHome specializes in property management, by taking an intermediate role between owners who wish to rent out their furnished property and clients of *DreamHome* who require to rent furnished property for a fixed period. *DreamHome* currently has about 2000 staff working in 100 branches. When a member of staff joins the Company, the *DreamHome* staff registration form is used. The staff registration form for Susan Brand is shown in Figure 10.1.

Each branch has an appropriate number and type of staff including a Manager, Supervisors, and Assistants. The Manager is responsible for the day-to-day running of a branch and each Supervisor is responsible for supervising a group of staff called Assistants. An example of the first page of a report listing the details of staff working at a branch office in Glasgow is shown in Figure 10.2.

Each branch office offers a range of properties for rent. To offer property through *DreamHome*, a property owner normally contacts the *DreamHome* branch office nearest to the property for rent. The owner provides the details of the property and agrees an



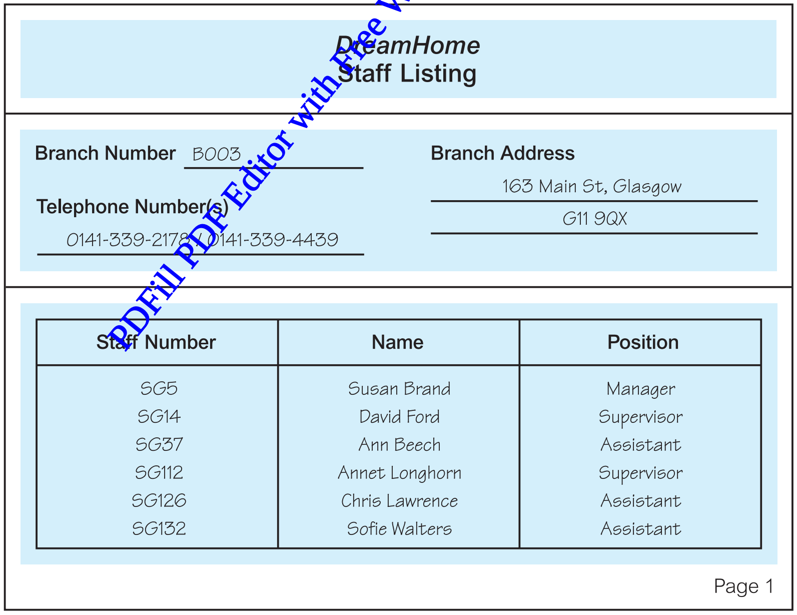
DreamHome
Staff Registration Form

Staff Number	SG5
Full Name	Susan Brand
Sex	F
DOB	3-Jun-40
Position	Manager
Salary	24000
Branch Number	B003
Branch Address	163 Main St, Glasgow
Telephone Number(s)	0141-339-2178 / 0141-339-4439

Enter details where applicable

Supervisor Name	
Manager Start Date	01-Jun-90
Manager Bonus	2350

Figure 10.1
The *DreamHome* staff registration form for Susan Brand.



DreamHome
Staff Listing

Branch Number	B003
Telephone Number(s)	0141-339-2178 / 0141-339-4439
Branch Address	163 Main St, Glasgow G11 9QX

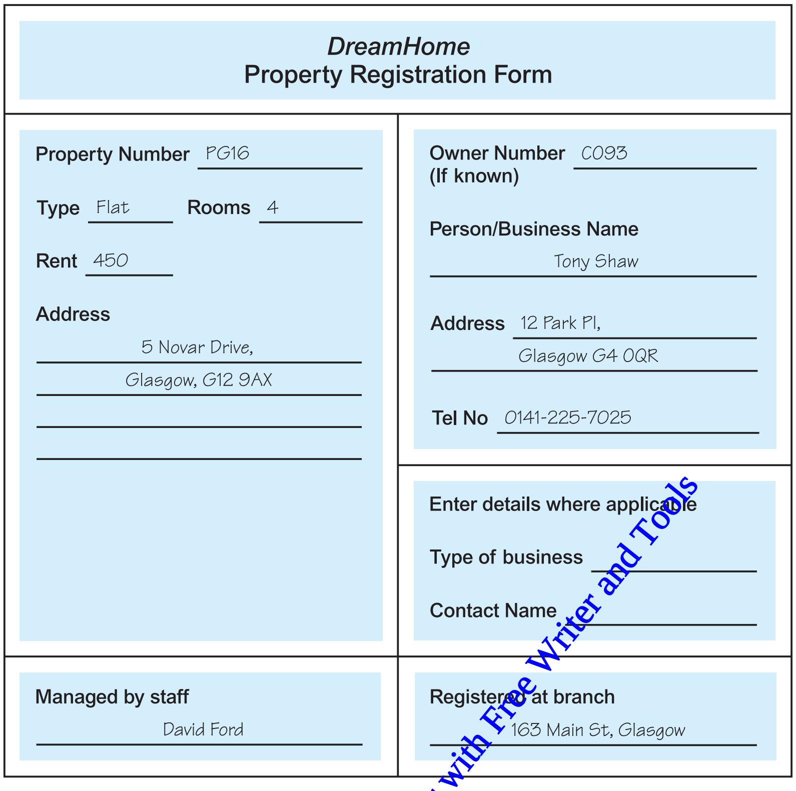
Staff Number	Name	Position
SG5	Susan Brand	Manager
SG14	David Ford	Supervisor
SG37	Ann Beech	Assistant
SG112	Annet Longhorn	Supervisor
SG126	Chris Lawrence	Assistant
SG132	Sofie Walters	Assistant

Page 1

Figure 10.2
Example of the first page of a report listing the details of staff working at a *DreamHome* branch office in Glasgow.

appropriate rent for the property with the branch Manager. The registration form for a property in Glasgow is shown in Figure 10.3.

Once a property is registered, *DreamHome* provides services to ensure that the property is rented out for maximum return for both the property owner and, of course, *DreamHome*.



DreamHome
Property Registration Form

Property Number PG16
Type Flat Rooms 4
Rent 450
Address
5 Novar Drive,
Glasgow, G12 9AX

Owner Number C093
(If known)
Person/Business Name
Tony Shaw
Address 12 Park Pl,
Glasgow G4 0QR
Tel No 0141-225-7025

Enter details where applicable
Type of business
Contact Name

Managed by staff
David Ford

Registered at branch
163 Main St, Glasgow

Figure 10.3 The *DreamHome* property registration form for a property in Glasgow.

These services include interviewing prospective renters (called clients), organizing viewings of the property by clients, advertising the property in local or national newspapers (when necessary), and negotiating the lease. Once rented, *DreamHome* assumes responsibility for the property including the collection of rent.

Members of the public interested in renting out property must first contact their nearest *DreamHome* branch office to register as clients of *DreamHome*. However, before registration is accepted, a prospective client is normally interviewed to record personal details and preferences of the client in terms of property requirements. The registration form for a client called Mike Ritchie is shown in Figure 10.4.

Once registration is complete, clients are provided with weekly reports that list properties currently available for rent. An example of the first page of a report listing the properties available for rent at a branch office in Glasgow is shown in Figure 10.5.

Clients may request to view one or more properties from the list and after viewing will normally provide a comment on the suitability of the property. The first page of a report describing the comments made by clients on a property in Glasgow is shown in Figure 10.6. Properties that prove difficult to rent out are normally advertised in local and national newspapers.

Once a client has identified a suitable property, a member of staff draws up a lease. The lease between a client called Mike Ritchie and a property in Glasgow is shown in Figure 10.7.

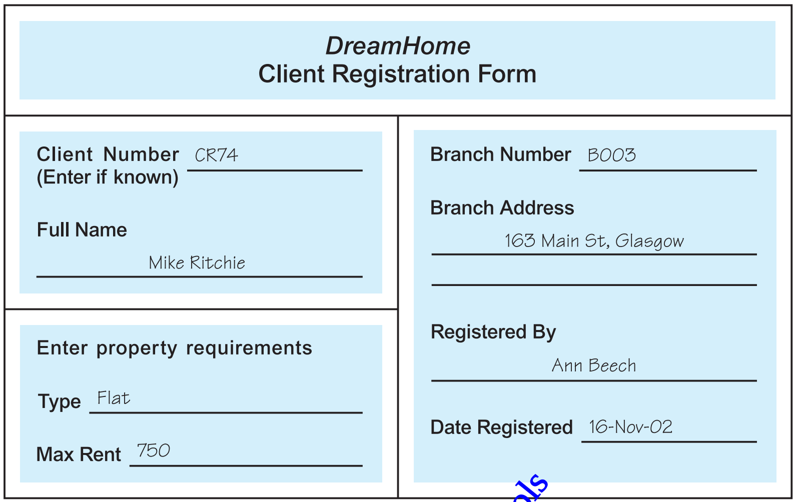


Figure 10.4
The *DreamHome* client registration form for Mike Ritchie.

DreamHome
Client Registration Form

Client Number (Enter if known)	CR74
Full Name	Mike Ritchie
Branch Number	B003
Branch Address	163 Main St, Glasgow
Registered By	Ann Beech
Date Registered	16-Nov-02

Enter property requirements

Type	Flat
Max Rent	750

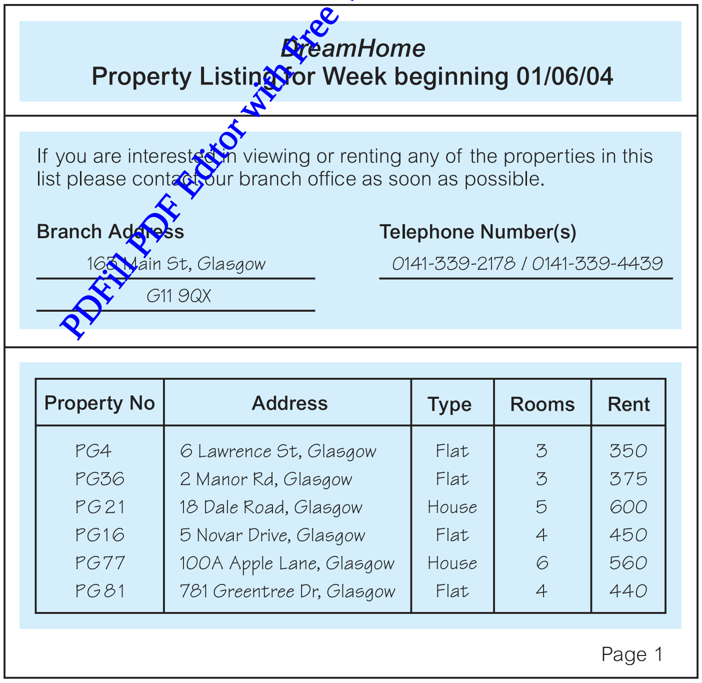


Figure 10.5
The first page of the *DreamHome* property for rent report listing property available at a branch in Glasgow.

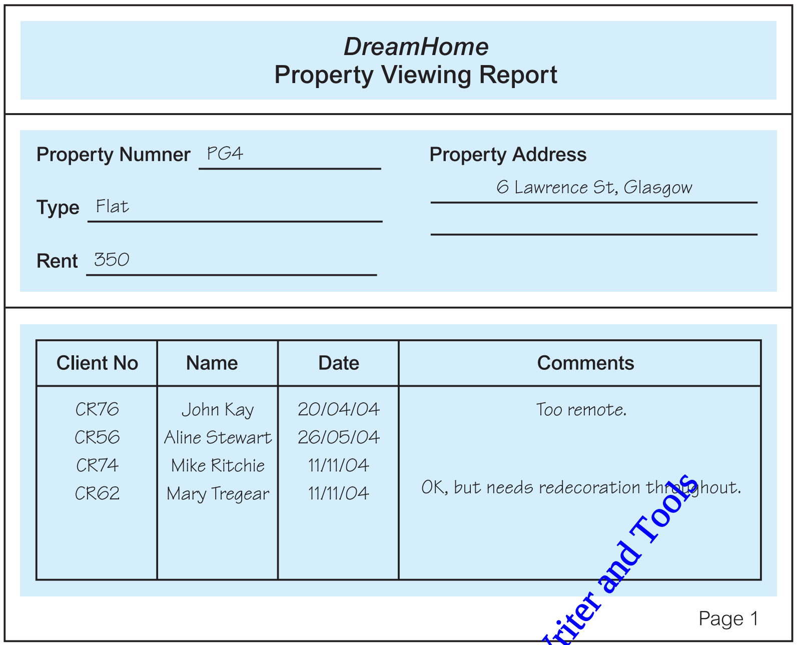
DreamHome
Property Listing for Week beginning 01/06/04

If you are interested in viewing or renting any of the properties in this list please contact our branch office as soon as possible.

Branch Address	163 Main St, Glasgow G11 9QX
Telephone Number(s)	0141-339-2178 / 0141-339-4439

Property No	Address	Type	Rooms	Rent
PG4	6 Lawrence St, Glasgow	Flat	3	350
PG36	2 Manor Rd, Glasgow	Flat	3	375
PG21	18 Dale Road, Glasgow	House	5	600
PG16	5 Novar Drive, Glasgow	Flat	4	450
PG77	100A Apple Lane, Glasgow	House	6	560
PG81	781 Greentree Dr, Glasgow	Flat	4	440

Page 1



DreamHome
Property Viewing Report

Property Numner PG4

Type Flat

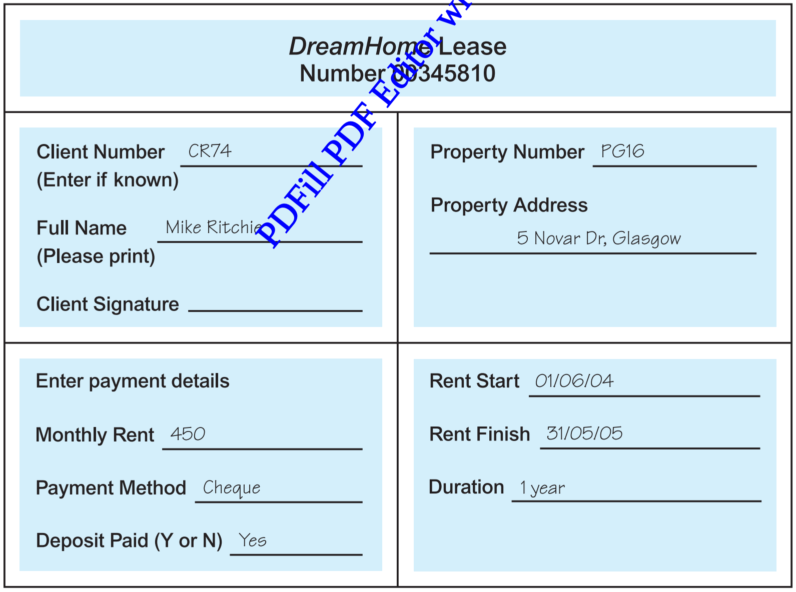
Rent 350

Property Address
6 Lawrence St, Glasgow

Client No	Name	Date	Comments
CR76	John Kay	20/04/04	Too remote.
CR56	Aline Stewart	26/05/04	
CR74	Mike Ritchie	11/11/04	
CR62	Mary Tregear	11/11/04	OK, but needs redecoration throughout.

Page 1

Figure 10.6 The first page of the *DreamHome* property viewing report for a property in Glasgow.



DreamHome Lease
Number 00345810

Client Number (Enter if known) CR74

Full Name (Please print) Mike Ritchie

Client Signature

Property Number PG16

Property Address
5 Novar Dr, Glasgow

Enter payment details

Monthly Rent 450

Payment Method Cheque

Deposit Paid (Y or N) Yes

Rent Start 01/06/04

Rent Finish 31/05/05

Duration 1 year

Figure 10.7 The *DreamHome* lease form for a client called Mike Ritchie renting a property in Glasgow.

At the end of a rental period a client may request that the rental be continued; however, this requires that a new lease be drawn up. Alternatively, a client may request to view alternative properties for the purposes of renting.

10.4.2 The *DreamHome* Case Study – Database Planning

The first step in developing a database system is to clearly define the **mission statement** for the database project, which defines the major aims of the database system. Once the mission statement is defined, the next activity involves identifying the **mission objectives**, which should identify the particular tasks that the database must support (see Section 9.3).

Creating the mission statement for the DreamHome database system

We begin the process of creating a mission statement for the *DreamHome* database system by conducting interviews with the Director and any other appropriate staff, as indicated by the Director. Open-ended questions are normally the most useful at this stage of the process. Examples of typical questions we might ask include:

'What is the purpose of your company?'

'Why do you feel that you need a database?'

'How do you know that a database will solve your problem?'

For example, the database developer may start the interview by asking the Director of *DreamHome* the following questions:

Database Developer	*What is the purpose of your company?*
Director	We offer a wide range of high quality properties for rent to clients registered at our branches throughout the UK. Our ability to offer quality properties, of course, depends upon the services we provide to property owners. We provide a highly professional service to property owners to ensure that properties are rented out for maximum return.
Database Developer	*Why do you feel that you need a database?*
Director	To be honest we can't cope with our own success. Over the past few years we've opened several branches in most of the main cities of the UK, and at each branch we now offer a larger selection of properties to a growing number of clients. However, this success has been accompanied with increasing data management problems, which means that the level of service we provide is falling. Also, there's a lack of co-operation and sharing of information between branches, which is a very worrying development.

'The purpose of the *DreamHome* database system is to maintain the data that is used and generated to support the property rentals business for our clients and property owners and to facilitate the cooperation and sharing of information between branches.'

Figure 10.8 Mission statement for the *DreamHome* database system.

Database Developer	*How do you know that a database will solve your problem?*
Director	All I know is that we are drowning in paperwork. We need something that will speed up the way we work by automating a lot of the day-to-day tasks that seem to take for ever these days. Also, I want the branches to start working together. Databases will help to achieve this, won't they?

Responses to these types of questions should help to formulate the mission statement. An example mission statement for the *DreamHome* database system is shown in Figure 10.8. When we have a clear and unambiguous mission statement that the staff of *DreamHome* agree with, we move on to define the mission objectives.

Creating the mission objectives for the *DreamHome* database system

The process of creating mission objectives involves conducting interviews with appropriate members of staff. Again, open-ended questions are normally the most useful at this stage of the process. To obtain the complete range of mission objectives, we interview various members of staff with different roles in *DreamHome*. Examples of typical questions we might ask include:

'What is your job description?'

'What kinds of tasks do you perform in a typical day?'

'What kinds of data do you work with?'

'What types of reports do you use?'

'What types of things do you need to keep track of?'

'What service does your company provide to your customers?'

These questions (or similar) are put to the Director of *DreamHome* and members of staff in the role of Manager, Supervisor, and Assistant. It may be necessary to adapt the questions as required depending on whom is being interviewed.

Director

Database Developer	*What role do you play for the company?*
Director	I oversee the running of the company to ensure that we continue to provide the best possible property rental service to our clients and property owners.

Database Developer	*What kinds of tasks do you perform in a typical day?*
Director	I monitor the running of each branch by our Managers. I try to ensure that the branches work well together and share important information about properties and clients. I normally try to keep a high profile with my branch Managers by calling into each branch at least once or twice a month.
Database Developer	*What kinds of data do you work with?*
Director	I need to see everything, well at least a summary of the data used or generated by *DreamHome*. That includes data about staff at all branches, all properties and their owners, all clients, and all leases. I also like to keep an eye on the extent to which branches advertise properties in newspapers.
Database Developer	*What types of reports do you use?*
Director	I need to know what's going on at all the branches and there's lots of them. I spend a lot of my working day going over long reports on all aspects of *DreamHome*. I need reports that are easy to access and that let me get a good overview of what's happening at a given branch and across all branches.
Database Developer	*What types of things do you need to keep track of?*
Director	As I said before, I need to have an overview of everything, I need to see the whole picture.
Database Developer	*What service does your company provide to your customers?*
Director	We try to provide the best property rental service in the UK.

Manager

Database Developer	*What is your job description?*
Manager	My job title is Manager. I oversee the day-to-day running of my branch to provide the best property rental service to our clients and property owners.
Database Developer	*What kinds of tasks do you perform in a typical day?*
Manager	I ensure that the branch has the appropriate number and type of staff on duty at all times. I monitor the registering of new properties and new clients, and the renting activity of our currently active clients. It's my responsibility to ensure that we have the right number and type of properties available to offer our clients. I sometimes get involved in negotiating leases for our top-of-the-range properties, although due to my workload I often have to delegate this task to Supervisors.
Database Developer	*What kinds of data do you work with?*
Manager	I mostly work with data on the properties offered at my branch and the owners, clients, and leases. I also need to know when properties are proving difficult to rent out so that I can arrange for them to be advertised in newspapers. I need to keep an eye on this aspect of the business because advertising can get costly. I also need access to data about staff working at my

	branch and staff at other local branches. This is because I sometimes need to contact other branches to arrange management meetings or to borrow staff from other branches on a temporary basis to cover staff shortages due to sickness or during holiday periods. This borrowing of staff between local branches is informal and thankfully doesn't happen very often. Besides data on staff, it would be helpful to see other types of data at the other branches such as data on property, property owners, clients, and leases, you know, to compare notes. Actually, I think the Director hopes that this database project is going to help promote cooperation and sharing of information between branches. However, some of the Managers I know are not going to be too keen on this because they think we're in competition with each other. Part of the problem is that a percentage of a Manager's salary is made up of a bonus, which is related to the number of properties we rent out.
Database Developer	*What types of reports do you use?*
Manager	I need various reports on staff, property, owners, clients, and leases. I need to know at a glance which properties we need to lease out and what clients are looking for.
Database Developer	*What types of things do you need to keep track of?*
Manager	I need to keep track of staff salaries. I need to know how well the properties on our books are being rented out and when leases are coming up for renewal. I also need to keep eye on our expenditure on advertising in newspapers.
Database Developer	*What service does your company provide to your customers?*
Manager	Remember that we have two types of customers, that is clients wanting to rent property and property owners. We need to make sure that our clients find the property they're looking for quickly without too much legwork and at a reasonable rent and, of course, that our property owners see good returns from renting out their properties with minimal hassle.

Supervisor

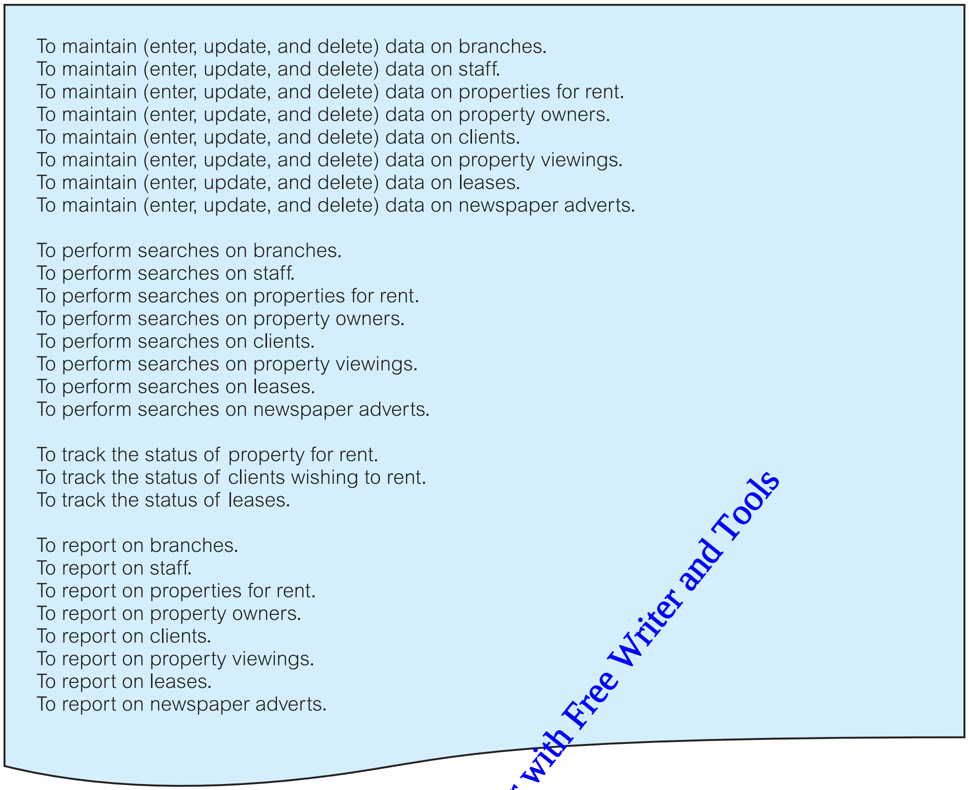
Database Developer	*What is your job description?*
Supervisor	My job title is Supervisor. I spend most of my time in the office dealing directly with our customers, that is clients wanting to rent property and property owners. I'm also responsible for a small group of staff called Assistants and making sure that they are kept busy, but that's not a problem as there's always plenty to do, it's never ending actually.
Database Developer	*What kinds of tasks do you perform in a typical day?*
Supervisor	I normally start the day by allocating staff to particular duties, such as dealing with clients or property owners, organizing for clients to view properties, and the filing of paperwork. When

a client finds a suitable property, I process the drawing up of a lease, although the Manager must see the documentation before any signatures are requested. I keep client details up to date and register new clients when they want to join the Company. When a new property is registered, the Manager allocates responsibility for managing that property to me or one of the other Supervisors or Assistants.

Database Developer	*What kinds of data do you work with?*
Supervisor	I work with data about staff at my branch, property, property owners, clients, property viewings, and leases.
Database Developer	*What types of reports do you use?*
Supervisor	Reports on staff and properties for rent.
Database Developer	*What types of things do you need to keep track of?*
Supervisor	I need to know what properties are available for rent and when currently active leases are due to expire. I also need to know what clients are looking for. I need to keep our Manager up to date with any properties that are proving difficult to rent out.

Assistant

Database Developer	*What is your job description?*
Assistant	My job title is Assistant. I deal directly with our clients.
Database Developer	*What kinds of tasks do you perform in a typical day?*
Assistant	I answer general queries from clients about properties for rent. You know what I mean: 'Do you have such and such type of property in a particular area of Glasgow?' I also register new clients and arrange for clients to view properties. When we're not too busy I file paperwork but I hate this part of the job, it's so boring.
Database Developer	*What kinds of data do you work with?*
Assistant	I work with data on property and property viewings by clients and sometimes leases.
Database Developer	*What types of reports do you use?*
Assistant	Lists of properties available for rent. These lists are updated every week.
Database Developer	*What types of things do you need to keep track of?*
Assistant	Whether certain properties are available for renting out and which clients are still actively looking for property.
Database Developer	*What service does your company provide to your customers?*
Assistant	We try to answer questions about properties available for rent such as: 'Do you have a 2-bedroom flat in Hyndland, Glasgow?' and 'What should I expect to pay for a 1-bedroom flat in the city center?'



To maintain (enter, update, and delete) data on branches.
To maintain (enter, update, and delete) data on staff.
To maintain (enter, update, and delete) data on properties for rent.
To maintain (enter, update, and delete) data on property owners.
To maintain (enter, update, and delete) data on clients.
To maintain (enter, update, and delete) data on property viewings.
To maintain (enter, update, and delete) data on leases.
To maintain (enter, update, and delete) data on newspaper adverts.

To perform searches on branches.
To perform searches on staff.
To perform searches on properties for rent.
To perform searches on property owners.
To perform searches on clients.
To perform searches on property viewings.
To perform searches on leases.
To perform searches on newspaper adverts.

To track the status of property for rent.
To track the status of clients wishing to rent.
To track the status of leases.

To report on branches.
To report on staff.
To report on properties for rent.
To report on property owners.
To report on clients.
To report on property viewings.
To report on leases.
To report on newspaper adverts.

Figure 10.9 Mission objectives for the *DreamHome* database system.

Responses to these types of questions should help to formulate the mission objectives. An example of the mission objectives for the *DreamHome* database system is shown in Figure 10.9.

10.4.3 The *DreamHome* Case Study – System Definition

The purpose of the system definition stage is to define the scope and boundary of the database system and its major user views. In Section 9.4.1 we described how a user view represents the requirements that should be supported by a database system as defined by a particular job role (such as Director or Supervisor) or business application area (such as property rentals or property sales).

Defining the systems boundary for the *DreamHome* database system

During this stage of the database system development lifecycle, further interviews with users can be used to clarify or expand on data captured in the previous stage. However, additional fact-finding techniques can also be used including examining the sample

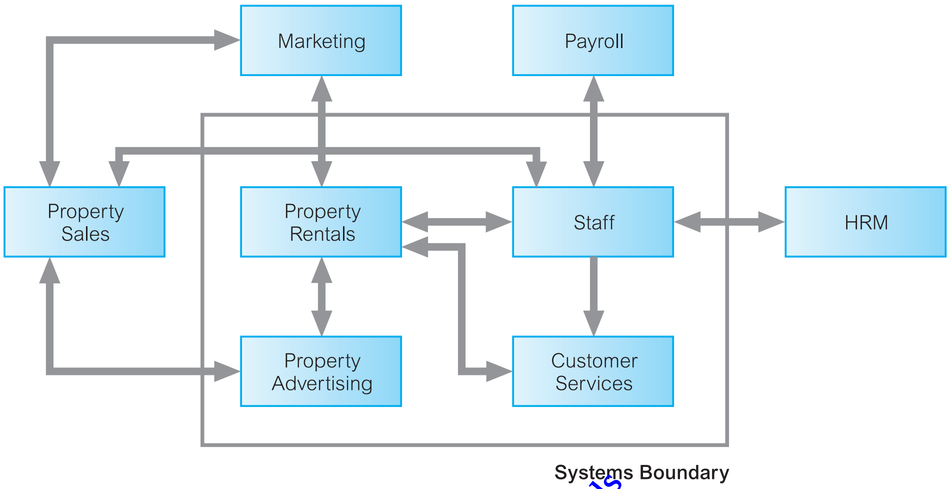


Figure 10.10 Systems boundary for the *DreamHome* database system.

documentation shown in Section 10.4.1. The data collected so far is analyzed to define the boundary of the database system. The systems boundary for the *DreamHome* database system is shown in Figure 10.10.

Identifying the major user views for the *DreamHome* database system

We now analyze the data collected so far to define the main user views of the database system. The majority of data about the user views was collected during interviews with the Director and members of staff in the role of Manager, Supervisor, and Assistant. The main user views for the *DreamHome* database system are shown in Figure 10.11.

10.4.4 The *DreamHome* Case Study – Requirements Collection and Analysis

During this stage, we continue to gather more details on the user views identified in the previous stage, to create a **users' requirements specification** that describes in detail the data to be held in the database and how the data is to be used. While gathering more information on the user views, we also collect any general requirements for the system. The purpose of gathering this information is to create a **systems specification**, which describes any features to be included in the new database system such as networking and shared access requirements, performance requirements, and the levels of security required.

As we collect and analyze the requirements for the new system we also learn about the most useful and most troublesome features of the current system. When building a new database system it is sensible to try to retain the good things about the old system while introducing the benefits that will be part of using the new system.

An important activity associated with this stage is deciding how to deal with the situation where there is more than one user view. As we discussed in Section 9.6, there are three

Data	Access Type	Director	Manager	Supervisor	Assistant
All Branches	Maintain				
	Query	X	X		
	Report	X	X		
Single Branch	Maintain		X		
	Query		X		
	Report		X		
All Staff	Maintain				
	Query	X	X		
	Report	X	X		
Branch Staff	Maintain		X		
	Query		X	X	
	Report		X	X	
All Property	Maintain				
	Query	X			
	Report	X	X		
Branch Property	Maintain		X	X	
	Query		X	X	X
	Report		X	X	X
All Owners	Maintain				
	Query	X			
	Report	X	X		
Branch Owners	Maintain		X	X	
	Query		X	X	X
	Report		X		
All Clients	Maintain				
	Query	X			
	Report	X	X		
Branch Clients	Maintain		X	X	
	Query		X	X	X
	Report		X		
All Viewings	Maintain				
	Query				
	Report				
Branch Viewings	Maintain			X	X
	Query			X	X
	Report			X	X
All Leases	Maintain				
	Query	X			
	Report	X	X		
Branch Leases	Maintain		X	X	
	Query		X	X	X
	Report		X	X	
All Newspapers	Maintain				
	Query	X			
	Report	X	X		
Branch Newspapers	Maintain		X		
	Query		X		
	Report		X		

Figure 10.11 Major user views for the *DreamHome* database system.

major approaches to dealing with multiple user views, namely the **centralized** approach, the **view integration** approach, and a combination of both approaches. We discuss how these approaches can be used shortly.

Gathering more information on the user views of the *DreamHome* database system

To find out more about the requirements for each user view, we may again use a selection of fact-finding techniques including interviews and observing the business in operation. Examples of the types of questions that we may ask about the data (represented as X) required by a user view include:

'What type of data do you need to hold on X?'

'What sorts of things do you do with the data on X?'

For example, we may ask a Manager the following questions:

Database Developer	*What type of data do you need to hold on staff?*
Manager	The types of data held on a member of staff is his or her full name, position, sex, date of birth, and salary.
Database Developer	*What sorts of things do you do with the data on staff?*
Manager	I need to be able to enter the details of new members of staff and delete their details when they leave. I need to keep the details of staff up to date and print reports that list the full name, position, and salary of each member of staff at my branch. I need to be able to allocate staff to Supervisors. Sometimes when I need to communicate with other branches, I need to find out the names and telephone numbers of Managers at other branches.

We need to ask similar questions about all the important data to be stored in the database. Responses to these questions will help identify the necessary details for the users' requirements specification.

Gathering information on the system requirements of the *DreamHome* database system

While conducting interviews about user views, we should also collect more general information on the system requirements. Examples of the types of questions that we may ask about the system include:

'What transactions run frequently on the database?'

'What transactions are critical to the operation of the organization?'

'When do the critical transactions run?'

'When are the low, normal, and high workload periods for the critical transactions?'

'What type of security do you want for the database system?'

'Is there any highly sensitive data that should be accessed only by certain members of staff?'

'What historical data do you want to hold?'

'What are the networking and shared access requirements for the database system?'

'What type of protection from failures or data loss do you want for the database system?'

For example, we may ask a Manager the following questions:

Database Developer	*What transactions run frequently on the database?*
Manager	We frequently get requests either by phone or by clients who call into our branch to search for a particular type of property in a particular area of the city and for a rent no higher than a particular amount. We also need up-to-date information on properties and clients so that reports can be run off that show properties currently available for rent and clients currently seeking property.
Database Developer	*What transactions are critical to the operation of the business?*
Manager	Again, critical transactions include being able to search for particular properties and to print out reports with up-to-date lists of properties available for rent. Our clients would go elsewhere if we couldn't provide this basic service.
Database Developer	*When do the critical transactions run?*
Manager	Every day.
Database Developer	*When are the low, normal, and high workload periods for the critical transactions?*
Manager	We're open six days a week. In general, we tend to be quiet in the mornings and get busier as the day progresses. However, the busiest time-slots each day for dealing with customers are between 12 and 2pm and 5 and 7pm.

We may ask the Director the following questions:

Database Developer	*What type of security do you want for the database system?*
Director	I don't suppose a database holding information for a property rental company holds very sensitive data, but I wouldn't want any of our competitors to see the data on properties, owners, clients, and leases. Staff should only see the data necessary to do their job in a form that suits what they're doing. For example, although it's necessary for Supervisors and Assistants to see client details, client records should only be displayed one at a time and not as a report.
Database Developer	*Is there any highly sensitive data that should be accessed only by certain members of staff?*
Director	As I said before, staff should only see the data necessary to do their jobs. For example, although Supervisors need to see data on staff, salary details should not be included.

Database Developer	*What historical data do you want to hold?*
Director	I want to keep the details of clients and owners for a couple of years after their last dealings with us, so that we can mailshot them to tell them about our latest offers, and generally try to attract them back. I also want to be able to keep lease information for a couple of years so that we can analyze it to find out which types of properties and areas of each city are the most popular for the property rental market, and so on.
Database Developer	*What are the networking and shared access requirements for the database system?*
Director	I want all the branches networked to our main branch office, here in Glasgow, so that staff can access the system from wherever and whenever they need to. At most branches, I would expect about two or three staff to be accessing the system at any one time, but remember we have about 100 branches. Most of the time the staff should be just accessing local branch data. However, I don't really want there to be any restrictions about how often or when the system can be accessed, unless it's got real financial implications.
Database Developer	*What type of protection from failures or data loss do you want for the database system?*
Director	The best of course. All our business is going to be conducted using the database, so if it goes down, we're sunk. To be serious for a minute, I think we probably have to back up our data every evening when the branch closes. What do you think?

We need to ask similar questions about all the important aspects of the system. Responses to these questions should help identify the necessary details for the system requirements specification.

Managing the user views of the *DreamHome* database system

How do we decide whether to use the centralized or view integration approach, or a combination of both to manage multiple user views? One way to help make a decision is to examine the overlap in the data used between the user views identified during the system definition stage. Table 10.7 cross-references the Director, Manager, Supervisor, and Assistant user views with the main types of data used by each user view.

We see from Table 10.7 that there is overlap in the data used by all user views. However, the Director and Manager user views and the Supervisor and Assistant user views show more similarities in terms of data requirements. For example, only the Director and Manager user views require data on branches and newspapers whereas only the Supervisor and Assistant user views require data on property viewings. Based on this analysis, we use the *centralized* approach to first merge the requirements for the Director and Manager user views (given the collective name of **Branch** user views) and the requirements for the Supervisor and Assistant user views (given the collective name of **Staff** user views). We

Table 10.7 Cross-reference of user views with the main types of data used by each.

	Director	Manager	Supervisor	Assistant
branch	X	X		
staff	X	X	X	
property for rent	X	X	X	X
owner	X	X	X	X
client	X	X	X	X
property viewing			X	X
lease	X	X	X	X
newspaper	X	X		

then develop data models representing the Branch and Staff user views and then use the *view integration* approach to merge the two data models.

Of course, for a simple case study like *DreamHome*, we could easily use the centralized approach for all user views but we will stay with our decision to create two collective user views so that we can describe and demonstrate how the view integration approach works in practice in Chapter 16.

It is difficult to give precise rules as to when it is appropriate to use the centralized or view integration approaches. The decision should be based on an assessment of the complexity of the database system and the degree of overlap between the various user views. However, whether we use the centralized or view integration approach or a mixture of both to build the underlying database, ultimately we need to re-establish the original user views (namely Director, Manager, Supervisor, and Assistant) for the working database system. We describe and demonstrate the establishment of the user views for the database system in Chapter 17.

All of the information gathered so far on each user view of the database system is described in a document called a **users' requirements specification**. The users' requirements specification describes the data requirements for each user view and examples of how the data is used by the user view. For ease of reference the users' requirements specifications for the Branch and Staff user views of the *DreamHome* database system are given in Appendix A. In the remainder of this chapter, we present the general systems requirements for the *DreamHome* database system.

The systems specification for the *DreamHome* database system

The systems specification should list all the important features for the *DreamHome* database system. The types of features that should be described in the systems specification include:

- initial database size;
- database rate of growth;
- the types and average number of record searches;

- networking and shared access requirements;
- performance;
- security;
- backup and recovery;
- legal issues.

Systems Requirements for *DreamHome* Database System

Initial database size

(1) There are approximately 2000 members of staff working at over 100 branches. There is an average of 20 and a maximum of 40 members of staff at each branch.

(2) There are approximately 100,000 properties available at all branches. There is an average of 1000 and a maximum of 3000 properties at each branch.

(3) There are approximately 60,000 property owners. There is an average of 600 and a maximum of 1000 property owners at each branch.

(4) There are approximately 100,000 clients registered across all branches. There is an average of 1000 and a maximum of 1500 clients registered at each branch.

(5) There are approximately 4,000,000 viewings across all branches. There is an average of 40,000 and a maximum of 100,000 viewings at each branch.

(6) There are approximately 400,000 leases across all branches. There are an average of 4000 and a maximum of 10,000 leases at each branch.

(7) There are approximately 50,000 newspaper adverts in 100 newspapers across all branches.

Database rate of growth

(1) Approximately 500 new properties and 200 new property owners are added to the database each month.

(2) Once a property is no longer available for renting out, the corresponding record is deleted from the database. Approximately 100 records of properties are deleted each month.

(3) If a property owner does not provide properties for rent at any time within a period of two years, his or her record is deleted. Approximately 100 property owner records are deleted each month.

(4) Approximately 20 members of staff join and leave the company each month. The records of staff who have left the company are deleted after one year. Approximately 20 staff records are deleted each month.

(5) Approximately 1000 new clients register at branches each month. If a client does not view or rent out a property at any time within a period of two years, his or her record is deleted. Approximately 100 client records are deleted each month.

(6) Approximately 5000 new viewings are recorded across all branches each day. The details of property viewings are deleted one year after the creation of the record.

(7) Approximately 1000 new leases are recorded across all branches each month. The details of property leases are deleted two years after the creation of the record.

(8) Approximately 1000 newspaper adverts are placed each week. The details of newspaper adverts are deleted one year after the creation of the record.

The types and average number of record searches

(1) Searching for the details of a branch – approximately 10 per day.

(2) Searching for the details of a member of staff at a branch – approximately 20 per day.

(3) Searching for the details of a given property – approximately 5000 per day (Monday to Thursday), approximately 10,000 per day (Friday and Saturday). Peak workloads are 12.00–14.00 and 17.00–19.00 daily.

(4) Searching for the details of a property owner – approximately 100 per day.

(5) Searching for the details of a client – approximately 1000 per day (Monday to Thursday), approximately 2000 per day (Friday and Saturday). Peak workloads are 12.00–14.00 and 17.00–19.00 daily.

(6) Searching for the details of a property viewing – approximately 2000 per day (Monday to Thursday), approximately 5000 per day (Friday and Saturday). Peak workloads are 12.00–14.00 and 17.00–19.00 daily.

(7) Searching for the details of a lease – approximately 1000 per day (Monday to Thursday), approximately 2000 per day (Friday and Saturday). Peak workloads are 12.00–14.00 and 17.00–19.00 daily.

Networking and shared access requirements

All branches should be securely networked to a centralized database located at *DreamHome*'s main office in Glasgow. The system should allow for at least two to three people concurrently accessing the system from each branch. Consideration needs to be given to the licensing requirements for this number of concurrent accesses.

Performance

(1) During opening hours but not during peak periods expect less than 1 second response for all single record searches. During peak periods expect less than 5 second response for each search.

(2) During opening hours but not during peak periods expect less than 5 second response for each multiple record search. During peak periods expect less than 10 second response for each multiple record search.

(3) During opening hours but not during peak periods expect less than 1 second response for each update/save. During peak periods expect less than 5 second response for each update/save.

Security

(1) The database should be password-protected.

(2) Each member of staff should be assigned database access privileges appropriate to a particular user view, namely Director, Manager, Supervisor, or Assistant.

(3) A member of staff should only see the data necessary to do his or her job in a form that suits what he or she is doing.

Backup and Recovery

The database should be backed up daily at 12 midnight.

Legal Issues

Each country has laws that govern the way that the computerized storage of personal data is handled. As the *DreamHome* database holds data on staff, clients, and property owners any legal issues that must be complied with should be investigated and implemented.

10.4.5 The *DreamHome* Case Study – Database Design

In this chapter we demonstrated the creation of the users' requirements specification for the Branch and Staff user views and the systems specification for the *DreamHome* database system. These documents are the sources of information for the next stage of the lifecycle called **database design**. In Chapters 15 to 18 we provide a step-by-step methodology for database design and use the *DreamHome* case study and the documents created for the *DreamHome* database system in this chapter to demonstrate the methodology in practice.

Chapter Summary

- **Fact-finding** is the formal process of using techniques such as interviews and questionnaires to collect facts about systems, requirements, and preferences.
- Fact-finding is particularly crucial to the early stages of the database system development lifecycle including the database planning, system definition, and requirements collection and analysis stages.
- The five most common fact-finding techniques are examining documentation, interviewing, observing the enterprise in operation, conducting research, and using questionnaires.
- There are two main documents created during the requirements collection and analysis stage, namely the **users' requirements specification** and the **systems specification**.
- The **users' requirements specification** describes in detail the data to be held in the database and how the data is to be used.
- The **systems specification** describes any features to be included in the database system such as the performance and security requirements.

Review Questions

10.1 Briefly describe what the process of fact-finding attempts to achieve for a database developer.

10.2 Describe how fact-finding is used throughout the stages of the database system development lifecycle.

10.3 For each stage of the database system development lifecycle identify examples of the facts captured and the documentation produced.

10.4 A database developer normally uses several fact-finding techniques during a single database project. The five most commonly used techniques are examining documentation, interviewing, observing the business in operation, conducting research, and using questionnaires. Describe each fact-finding technique and identify the advantages and disadvantages of each.

10.5 Describe the purpose of defining a mission statement and mission objectives for a database system.

10.6 What is the purpose of identifying the systems boundary for a database system?

10.7 How do the contents of a users' requirements specification differ from a systems specification?

10.8 Describe one method of deciding whether to use either the centralized or view integration approach, or a combination of both when developing a database system with multiple user views.

Exercises

10.9 Assume that you are developing a database system for your enterprise, whether it is a university (or college) or business (or department). Consider what fact-finding techniques you would use to identify the important facts needed to develop a database system. Identify the techniques that you would use for each stage of the database system development lifecycle.

10.10 Assume that you are developing a database system for the case studies described in Appendix B. Consider what fact-finding techniques you would use to identify the important facts needed to develop a database system.

10.11 Produce mission statements and mission objectives for the database systems described in the case studies given in Appendix B.

10.12 Produce a diagram to represent the scope and boundaries for the database systems described in the case studies given in Appendix B.

10.13 Identify the major user views for the database systems described in the case studies given in Appendix B.

Chapter 11

Entity–Relationship Modeling

Chapter Objectives

In this chapter you will learn:

- How to use Entity–Relationship (ER) modeling in database design.
- The basic concepts associated with the Entity–Relationship (ER) model, namely entities, relationships, and attributes.
- A diagrammatic technique for displaying an ER model using the Unified Modeling Language (UML).
- How to identify and resolve problems with ER models called connection traps.

In Chapter 10 we described the main techniques for gathering and capturing information about what the users require of a database system. Once the requirements collection and analysis stage of the database system development lifecycle is complete and we have documented the requirements for the database system, we are ready to begin the database design stage.

One of the most difficult aspects of database design is the fact that designers, programmers, and end-users tend to view data and its use in different ways. Unfortunately, unless we gain a common understanding that reflects how the enterprise operates, the design we produce will fail to meet the users' requirements. To ensure that we get a precise understanding of the nature of the data and how it is used by the enterprise, we need to have a model for communication that is non-technical and free of ambiguities. The Entity–Relationship (ER) model is one such example. ER modeling is a top-down approach to database design that begins by identifying the important data called *entities* and *relationships* between the data that must be represented in the model. We then add more details such as the information we want to hold about the entities and relationships called *attributes* and any *constraints* on the entities, relationships, and attributes. ER modeling is an important technique for any database designer to master and forms the basis of the methodology presented in this book.

In this chapter we introduce the basic concepts of the ER model. Although there is general agreement about what each concept means, there are a number of different notations that can be used to represent each concept diagrammatically. We have chosen a diagrammatic notation that uses an increasingly popular object-oriented modeling language called the **Unified Modeling Language** (UML) (Booch *et al.*, 1999). UML is the successor to a number of object-oriented analysis and design methods introduced in the 1980s and 1990s. The Object Management Group (OMG) is currently looking at the standardization of UML and it is anticipated that UML will be the *de facto* standard modeling language in the near future. Although we use the UML notation for drawing ER models, we continue to describe the concepts of ER models using traditional database terminology. In Section 25.7 we will provide a fuller discussion on UML. We also include a summary of two alternative diagrammatic notations for ER models in Appendix F.

In the next chapter we discuss the inherent problems associated with representing complex database applications using the basic concepts of the ER model. To overcome these problems, additional 'semantic' concepts were added to the original ER model resulting in the development of the Enhanced Entity–Relationship (EER) model. In Chapter 12 we describe the main concepts associated with the EER model called specialization/generalization, aggregation, and composition. We also demonstrate how to convert the ER model shown in Figure 11.1 into the EER model shown in Figure 12.8.

Structure of this Chapter

In Sections 11.1, 11.2, and 11.3 we introduce the basic concepts of the Entity–Relationship model, namely entities, relationships, and attributes. In each section we illustrate how the basic ER concepts are represented pictorially in an ER diagram using UML. In Section 11.4 we differentiate between weak and strong entities and in Section 11.5 we discuss how attributes normally associated with entities can be assigned to relationships. In Section 11.6 we describe the structural constraints associated with relationships. Finally, in Section 11.7 we identify potential problems associated with the design of an ER model called connection traps and demonstrate how these problems can be resolved.

The ER diagram shown in Figure 11.1 is an example of one of the possible end-products of ER modeling. This model represents the relationships between data described in the requirements specification for the Branch view of the *DreamHome* case study given in Appendix A. This figure is presented at the start of this chapter to show the reader an example of the type of model that we can build using ER modeling. At this stage, the reader should not be concerned about fully understanding this diagram, as the concepts and notation used in this figure are discussed in detail throughout this chapter.

11.1 Entity Types

Entity type	A group of objects with the same properties, which are identified by the enterprise as having an independent existence.

The basic concept of the ER model is the **entity type**, which represents a group of 'objects' in the 'real world' with the same properties. An entity type has an independent existence

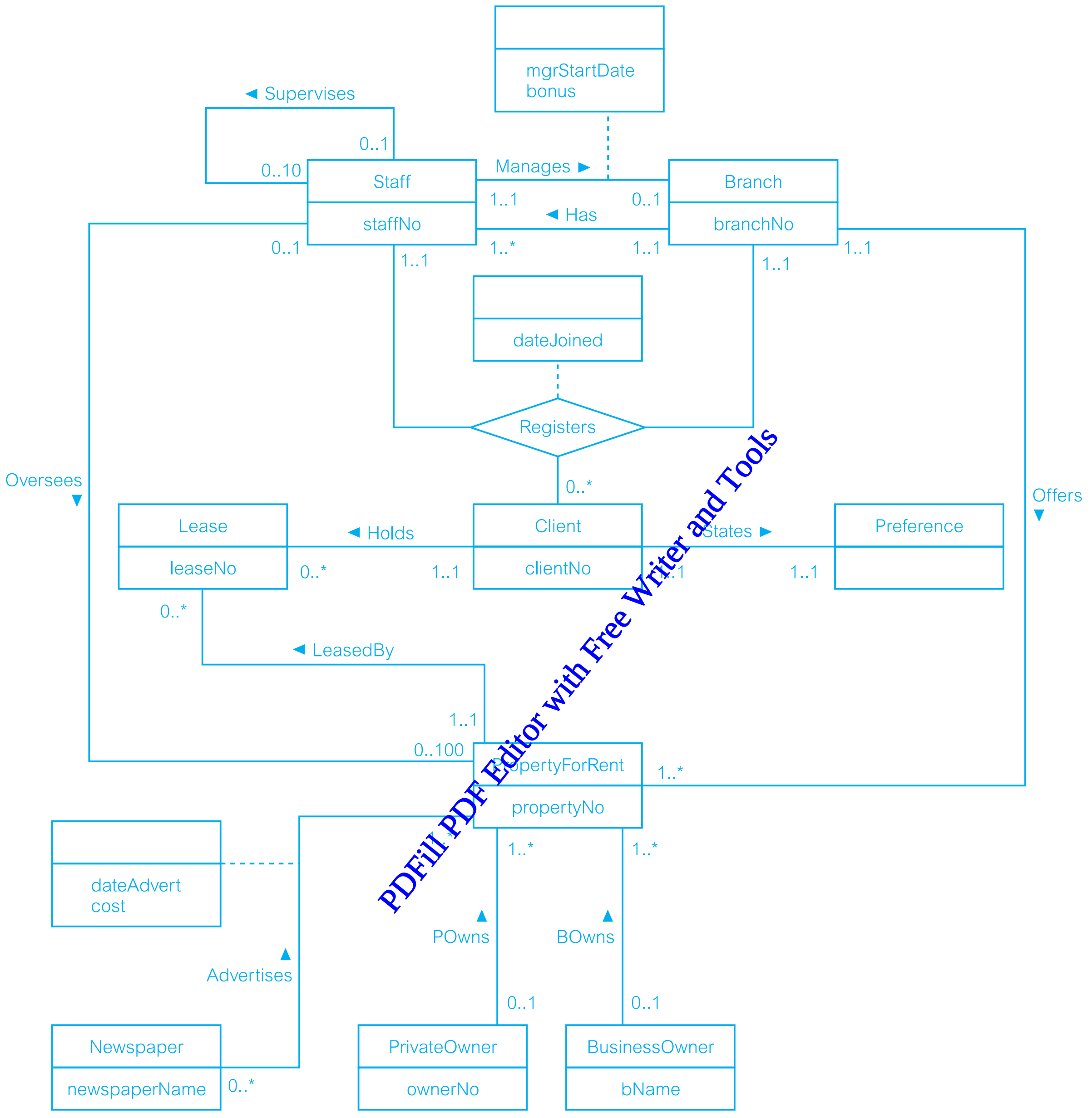


Figure 11.1 An Entity–Relationship (ER) diagram of the Branch view of *DreamHome*.

Physical existence	
Staff	Part
Property	Supplier
Customer	Product
Conceptual existence	
Viewing	Sale
Inspection	Work experience

Figure 11.2 Example of entities with a physical or conceptual existence.

and can be objects with a physical (or 'real') existence or objects with a conceptual (or 'abstract') existence, as listed in Figure 11.2. Note that we are only able to give a working definition of an entity type as no strict formal definition exists. This means that different designers may identify different entities.

Entity occurrence A uniquely identifiable object of an entity type.

Each uniquely identifiable object of an entity type is referred to simply as an **entity occurrence**. Throughout this book, we use the terms 'entity type' or 'entity occurrence'; however, we use the more general term 'entity' where the meaning is obvious.

We identify each entity type by a name and a list of properties. A database normally contains many different entity types. Examples of entity types shown in Figure 11.1 include: Staff, Branch, PropertyForRent, and PrivateOwner.

Diagrammatic representation of entity types

Each entity type is shown as a rectangle labeled with the name of the entity, which is normally a singular noun. In UML, the first letter of each word in the entity name is upper case (for example, Staff and PropertyForRent). Figure 11.3 illustrates the diagrammatic representation of the Staff and Branch entity types.

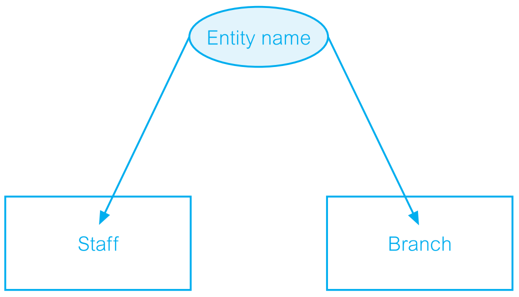


Figure 11.3 Diagrammatic representation of the Staff and Branch entity types.

11.2 Relationship Types

Relationship type A set of meaningful associations among entity types.

A **relationship type** is a set of associations between one or more participating entity types. Each relationship type is given a name that describes its function. An example of a relationship type shown in Figure 11.1 is the relationship called *POwns*, which associates the PrivateOwner and PropertyForRent entities.

As with entity types and entities, it is necessary to distinguish between the terms 'relationship type' and 'relationship occurrence'.

Relationship occurrence A uniquely identifiable association, which includes one occurrence from each participating entity type.

A **relationship occurrence** indicates the particular entity occurrences that are related. Throughout this book, we use the terms 'relationship type' or 'relationship occurrence'. However, as with the term 'entity', we use the more general term 'relationship' when the meaning is obvious.

Consider a relationship type called *Has*, which represents an association between Branch and Staff entities, that is Branch *Has* Staff. Each occurrence of the *Has* relationship associates one Branch entity occurrence with one Staff entity occurrence. We can examine examples of individual occurrences of the *Has* relationship using a *semantic net*. A semantic net is an object-level model, which uses the symbol • to represent entities and the symbol ◈ to represent relationships. The semantic net in Figure 11.4 shows three examples of the *Has* relationships (denoted r1, r2, and r3). Each relationship describes an association of a single Branch entity occurrence with a single Staff entity occurrence. Relationships are represented by lines that join each participating Branch entity with the associated Staff entity. For example, relationship r1 represents the association between Branch entity B003 and Staff entity SG37.

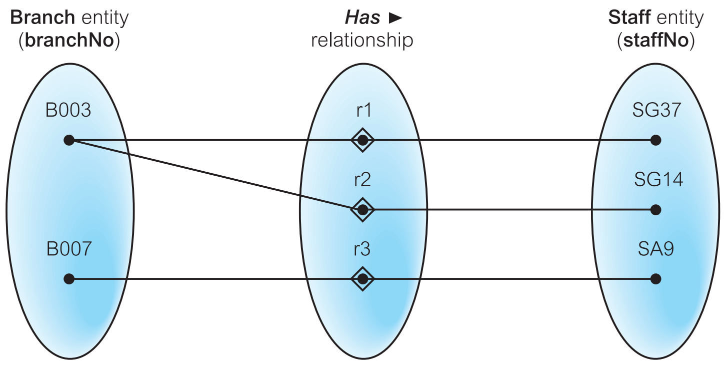

Branch entity (**branchNo**)
Has ►
relationship
Staff entity (**staffNo**)
B003
B007
r1
r2
r3
SG37
SG14
SA9

Figure 11.4 A semantic net showing individual occurrences of the *Has* relationship type.

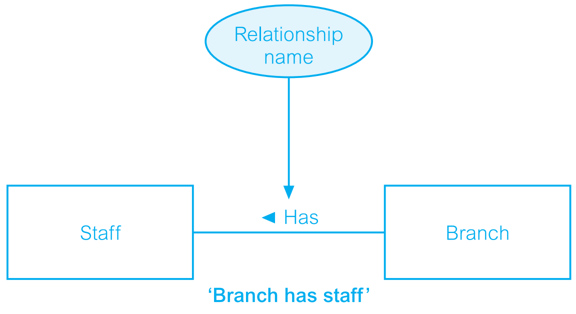


Figure 11.5 A diagrammatic representation of Branch *Has* Staff relationship type.

Note that we represent each Branch and Staff entity occurrences using values for the their primary key attributes, namely branchNo and staffNo. Primary key attributes uniquely identify each entity occurrence and are discussed in detail in the following section.

If we represented an enterprise using semantic nets, it would be difficult to understand due to the level of detail. We can more easily represent the relationships between entities in an enterprise using the concepts of the Entity–Relationship (ER) model. The ER model uses a higher level of abstraction than the semantic net by combining sets of entity occurrences into entity types and sets of relationship occurrences into relationship types.

Diagrammatic representation of relationships types

Each relationship type is shown as a line connecting the associated entity types, labeled with the name of the relationship. Normally, a relationship is named using a verb (for example, *Supervises* or *Manages*) or a short phrase including a verb (for example, *LeasedBy*). Again, the first letter of each word in the relationship name is shown in upper case. Whenever possible, a relationship name should be unique for a given ER model.

A relationship is only labeled in one direction, which normally means that the name of the relationship only makes sense in one direction (for example, Branch *Has* Staff makes more sense than Staff *Has* Branch). So once the relationship name is chosen, an arrow symbol is placed beside the name indicating the correct direction for a reader to interpret the relationship name (for example, Branch Has ► Staff) as shown in Figure 11.5.

11.2.1 Degree of Relationship Type

Degree of a relationship type	The number of participating entity types in a relationship.

The entities involved in a particular relationship type are referred to as **participants** in that relationship. The number of participants in a relationship type is called the **degree** of that

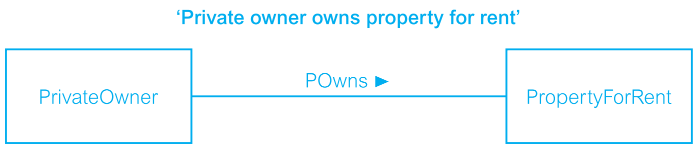


Figure 11.6 An example of a binary relationship called *POwns*.

relationship. Therefore, the degree of a relationship indicates the number of entity types involved in a relationship. A relationship of degree two is called **binary**. An example of a binary relationship is the *Has* relationship shown in Figure 11.5 with two participating entity types namely, Staff and Branch. A second example of a binary relationship is the *POwns* relationship shown in Figure 11.6 with two participating entity types, namely PrivateOwner and PropertyForRent. The *Has* and *POwns* relationships are also shown in Figure 11.1 as well as other examples of binary relationships. In fact the most common degree for a relationship is binary as demonstrated in this figure.

A relationship of degree three is called **ternary**. An example of a ternary relationship is *Registers* with three participating entity types, namely Staff, Branch, and Client. This relationship represents the registration of a client by a member of staff at a branch. The term 'complex relationship' is used to describe relationships with degrees higher than binary.

Diagrammatic representation of complex relationships

The UML notation uses a diamond to represent relationships with degrees higher than binary. The name of the relationship is displayed inside the diamond and in this case the directional arrow normally associated with the name is omitted. For example, the ternary relationship called *Registers* is shown in Figure 11.7. This relationship is also shown in Figure 11.1.

A relationship of degree four is called **quaternary**. As we do not have an example of such a relationship in Figure 11.1, we describe a quaternary relationship called *Arranges* with four participating entity types, namely Buyer, Solicitor, FinancialInstitution, and Bid in Figure 11.8. This relationship represents the situation where a buyer, advised by a solicitor and supported by a financial institution, places a bid.

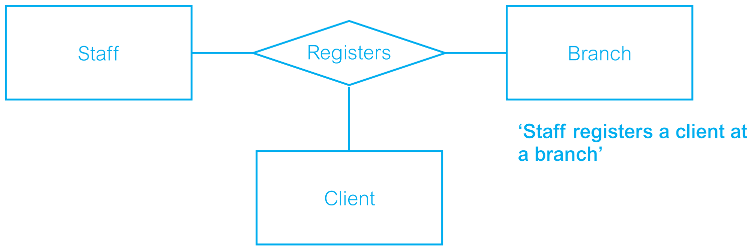


Figure 11.7 An example of a ternary relationship called *Registers*.

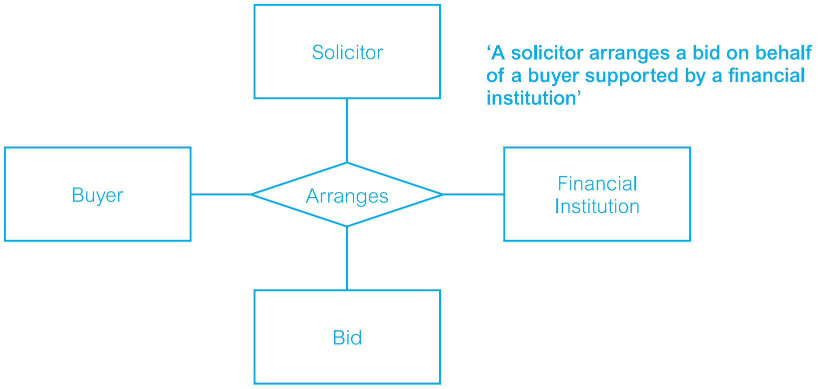

Solicitor
'A solicitor arranges a bid on behalf of a buyer supported by a financial institution'
Buyer
Arranges
Financial Institution
Bid

Figure 11.8 An example of a quaternary relationship called *Arranges*.

11.2.2 Recursive Relationship

Recursive relationship A relationship type where the *same* entity type participates more than once in *different roles*.

Consider a recursive relationship called *Supervises*, which represents an association of staff with a Supervisor where the Supervisor is also a member of staff. In other words, the Staff entity type participates twice in the *Supervises* relationship; the first participation as a Supervisor, and the second participation as a member of staff who is supervised (Supervisee). Recursive relationships are sometimes called *unary* relationships.

Relationships may be given **role names** to indicate the purpose that each participating entity type plays in a relationship. Role names can be important for recursive relationships to determine the function of each participant. The use of role names to describe the *Supervises* recursive relationship is shown in Figure 11.9. The first participation of the Staff entity type in the *Supervises* relationship is given the role name 'Supervisor' and the second participation is given the role name 'Supervisee'.

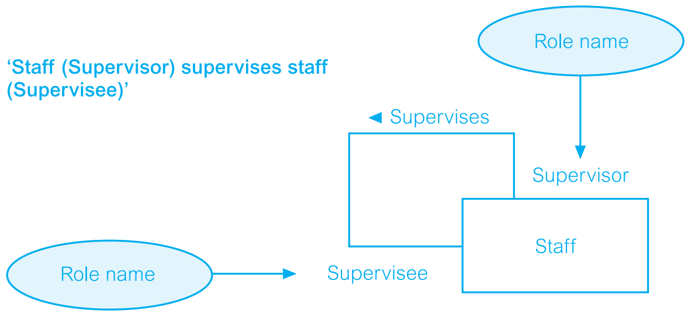


Figure 11.9 An example of a recursive relationship called *Supervises* with role names Supervisor and Supervisee.

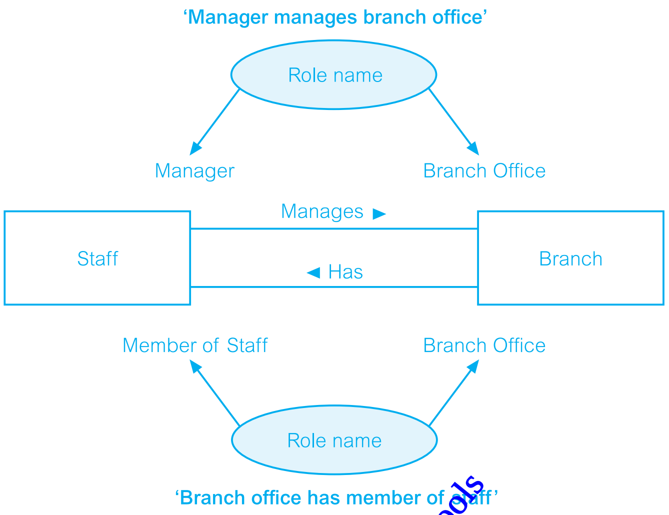


Figure 11.10 An example of entities associated through two distinct relationships called *Manages* and *Has* with role names.

Role names may also be used when two entities are associated through more than one relationship. For example, the Staff and Branch entity types are associated through two distinct relationships called *Manages* and *Has*. As shown in Figure 11.10, the use of role names clarifies the purpose of each relationship. For example, in the case of Staff *Manages* Branch, a member of staff (Staff entity) given the role name 'Manager' manages a branch (Branch entity) given the role name 'Branch Office'. Similarly, for Branch *Has* Staff, a branch, given the role name 'Branch Office' has staff given the role name 'Member of Staff'.

Role names are usually not required if the function of the participating entities in a relationship is unambiguous.

11.3 Attributes

Attribute A property of an entity or a relationship type.

The particular properties of entity types are called attributes. For example, a Staff entity type may be described by the staffNo, name, position, and salary attributes. The attributes hold values that describe each entity occurrence and represent the main part of the data stored in the database.

A relationship type that associates entities can also have attributes similar to those of an entity type but we defer discussion of this until Section 11.5. In this section, we concentrate on the general characteristics of attributes.

Attribute domain The set of allowable values for one or more attributes.

Each attribute is associated with a set of values called a domain. The domain defines the potential values that an attribute may hold and is similar to the domain concept in the relational model (see Section 3.2). For example, the number of rooms associated with a property is between 1 and 15 for each entity occurrence. We therefore define the set of values for the number of rooms (rooms) attribute of the PropertyForRent entity type as the set of integers between 1 and 15.

Attributes may share a domain. For example, the address attributes of the Branch, PrivateOwner, and BusinessOwner entity types share the same domain of all possible addresses. Domains can also be composed of domains. For example, the domain for the address attribute of the Branch entity is made up of subdomains: street, city, and postcode.

The domain of the name attribute is more difficult to define, as it consists of all possible names. It is certainly a character string, but it might consist not only of letters but also of hyphens or other special characters. A fully developed data model includes the domains of each attribute in the ER model.

As we now explain, attributes can be classified as being: *simple* or *composite*; *single-valued* or *multi-valued*; or *derived*.

Simple and Composite Attributes 11.3.1

Simple attribute An attribute composed of a single component with an independent existence.

Simple attributes cannot be further subdivided into smaller components. Examples of simple attributes include position and salary of the Staff entity. Simple attributes are sometimes called *atomic* attributes.

Composite attribute An attribute composed of multiple components, each with an independent existence.

Some attributes can be further divided to yield smaller components with an independent existence of their own. For example, the address attribute of the Branch entity with the value (163 Main St, Glasgow, G11 9QX) can be subdivided into street (163 Main St), city (Glasgow), and postcode (G11 9QX) attributes.

The decision to model the address attribute as a simple attribute or to subdivide the attribute into street, city, and postcode is dependent on whether the user view of the data refers to the address attribute as a single unit or as individual components.

Single-Valued and Multi-Valued Attributes 11.3.2

Single-valued attribute An attribute that holds a single value for each occurrence of an entity type.

The majority of attributes are single-valued. For example, each occurrence of the Branch entity type has a single value for the branch number (branchNo) attribute (for example B003), and therefore the branchNo attribute is referred to as being single-valued.

Multi-valued attribute	An attribute that holds multiple values for each occurrence of an entity type.

Some attributes have multiple values for each entity occurrence. For example, each occurrence of the Branch entity type can have multiple values for the telNo attribute (for example, branch number B003 has telephone numbers 0141-339-2178 and 0141-339-4439) and therefore the telNo attribute in this case is multi-valued. A multi-valued attribute may have a set of numbers with upper and lower limits. For example, the telNo attribute of the Branch entity type has between one and three values. In other words, a branch may have a minimum of a single telephone number to a maximum of three telephone numbers.

11.3.3 Derived Attributes

Derived attribute	An attribute that represents a value that is derivable from the value of a related attribute or set of attributes, not necessarily in the same entity type.

The values held by some attributes may be derived. For example, the value for the duration attribute of the Lease entity is calculated from the rentStart and rentFinish attributes also of the Lease entity type. We refer to the duration attribute as a derived attribute, the value of which is derived from the rentStart and rentFinish attributes.

In some cases, the value of an attribute is derived from the entity occurrences in the same entity type. For example, the total number of staff (totalStaff) attribute of the Staff entity type can be calculated by counting the total number of Staff entity occurrences.

Derived attributes may also involve the association of attributes of different entity types. For example, consider an attribute called deposit of the Lease entity type. The value of the deposit attribute is calculated as twice the monthly rent for a property. Therefore, the value of the deposit attribute of the Lease entity type is derived from the rent attribute of the PropertyForRent entity type.

11.3.4 Keys

Candidate key	The minimal set of attributes that uniquely identifies each occurrence of an entity type.

A candidate key is the minimal number of attributes, whose value(s) uniquely identify each entity occurrence. For example, the branch number (branchNo) attribute is the candidate key for the Branch entity type, and has a distinct value for each branch entity occurrence. The candidate key must hold values that are unique for every occurrence of an entity type. This implies that a candidate key cannot contain a null (see Section 3.2). For example, each branch has a unique branch number (for example, B003), and there will never be more than one branch with the same branch number.

Primary key The candidate key that is selected to uniquely identify each occurrence of an entity type.

An entity type may have more than one candidate key. For the purposes of discussion consider that a member of staff has a unique company-defined staff number (staffNo) and also a unique National Insurance Number (NIN) that is used by the Government. We therefore have two candidate keys for the Staff entity, one of which must be selected as the primary key.

The choice of primary key for an entity is based on considerations of attribute length, the minimal number of attributes required, and the future certainty of uniqueness. For example, the company-defined staff number contains a maximum of five characters (for example, SG14) while the NIN contains a maximum of nine characters (for example, WL220658D). Therefore, we select staffNo as the primary key of the Staff entity type and NIN is then referred to as the **alternate key**.

Composite key A candidate key that consists of two or more attributes.

In some cases, the key of an entity type is composed of several attributes, whose values together are unique for each entity occurrence but not separately. For example, consider an entity called Advert with propertyNo (property number), newspaperName, dateAdvert, and cost attributes. Many properties are advertised in many newspapers on a given date. To uniquely identify each occurrence of the Advert entity type requires values for the propertyNo, newspaperName, and dateAdvert attributes. Thus, the Advert entity type has a composite primary key made up of the propertyNo, newspaperName, and dateAdvert attributes.

Diagrammatic representation of attributes

If an entity type is to be displayed with its attributes, we divide the rectangle representing the entity in two. The upper part of the rectangle displays the name of the entity and the lower part lists the names of the attributes. For example, Figure 11.11 shows the ER diagram for the Staff and Branch entity types and their associated attributes.

The first attribute(s) to be listed is the primary key for the entity type, if known. The name(s) of the primary key attribute(s) can be labeled with the tag {PK}. In UML, the name of an attribute is displayed with the first letter in lower case and, if the name has more than one word, with the first letter of each subsequent word in upper case (for example, address and telNo). Additional tags that can be used include partial primary key {PPK} when an attribute forms part of a composite primary key, and alternate key {AK}. As

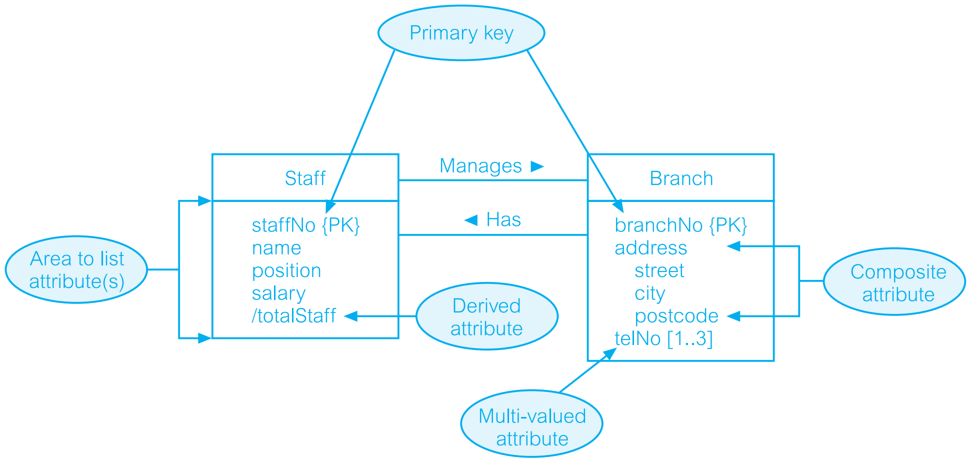


Figure 11.11 Diagrammatic representation of Staff and Branch entities and their attributes.

shown in Figure 11.11, the primary key of the Staff entity type is the staffNo attribute and the primary key of the Branch entity type is the branchNo attribute.

For some simpler database systems, it is possible to show all the attributes for each entity type in the ER diagram. However, for more complex database systems, we just display the attribute, or attributes, that form the primary key of each entity type. When only the primary key attributes are shown in the ER diagram, we can omit the {PK} tag.

For simple, single-valued attributes, there is no need to use tags and so we simply display the attribute names in a list below the entity name. For composite attributes, we list the name of the composite attribute followed below and indented to the right by the names of its simple component attributes. For example, in Figure 11.11 the composite attribute address of the Branch entity is shown, followed below by the names of its component attributes, street, city, and postcode. For multi-valued attributes, we label the attribute name with an indication of the range of values available for the attribute. For example, if we label the telNo attribute with the range [1..*], this means that the values for the telNo attribute is one or more. If we know the precise maximum number of values, we can label the attribute with an exact range. For example, if the telNo attribute holds one to a maximum of three values, we can label the attribute with [1..3].

For derived attributes, we prefix the attribute name with a '/'. For example, the derived attribute of the Staff entity type is shown in Figure 11.11 as /totalStaff.

11.4 Strong and Weak Entity Types

We can classify entity types as being strong or weak.

Strong entity type	An entity type that is *not* existence-dependent on some other entity type.

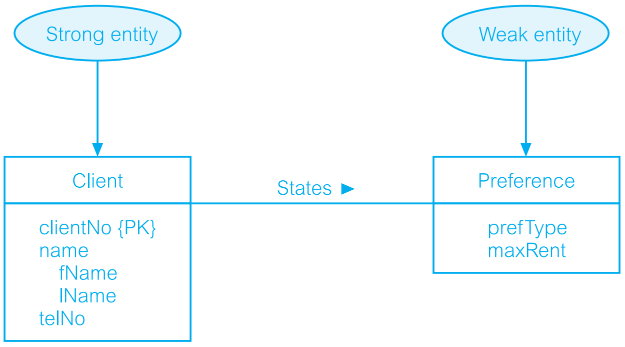


Figure 11.12 A strong entity type called Client and a weak entity type called Preference.

An entity type is referred to as being strong if its existence does not depend upon the existence of another entity type. Examples of strong entities are shown in Figure 11.1 and include the Staff, Branch, PropertyForRent, and Client entities. A characteristic of a strong entity type is that each entity occurrence is uniquely identifiable using the primary key attribute(s) of that entity type. For example, we can uniquely identify each member of staff using the staffNo attribute, which is the primary key for the Staff entity type.

Weak entity type	An entity type that is existence-dependent on some other entity type.

A weak entity type is dependent on the existence of another entity type. An example of a weak entity type called Preference is shown in Figure 11.12. A characteristic of a weak entity is that each entity occurrence cannot be uniquely identified using only the attributes associated with that entity type. For example, note that there is no primary key for the Preference entity. This means that we cannot identify each occurrence of the Preference entity type using only the attributes of this entity. We can only uniquely identify each preference through the relationship that a preference has with a client who is uniquely identifiable using the primary key for the Client entity type, namely clientNo. In this example, the Preference entity is described as having existence dependency for the Client entity, which is referred to as being the owner entity.

Weak entity types are sometimes referred to as *child*, *dependent*, or *subordinate* entities and strong entity types as *parent*, *owner*, or *dominant* entities.

11.5 Attributes on Relationships

As we mentioned in Section 11.3, attributes can also be assigned to relationships. For example, consider the relationship *Advertises*, which associates the Newspaper and PropertyForRent entity types as shown in Figure 11.1. To record the date the property was advertised and the cost, we associate this information with the *Advertises* relationship as attributes called dateAdvert and cost, rather than with the Newspaper or the PropertyForRent entities.

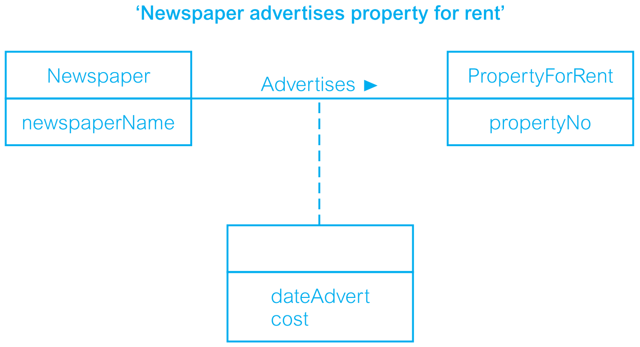


Figure 11.13 An example of a relationship called *Advertises* with attributes dateAdvert and cost.

Diagrammatic representation of attributes on relationships

We represent attributes associated with a relationship type using the same symbol as an entity type. However, to distinguish between a relationship with an attribute and an entity, the rectangle representing the attribute(s) is associated with the relationship using a dashed line. For example, Figure 11.13 shows the *Advertises* relationship with the attributes dateAdvert and cost. A second example shown in Figure 11.1 is the *Manages* relationship with the mgrStartDate and bonus attributes.

The presence of one or more attributes assigned to a relationship may indicate that the relationship conceals an unidentified entity type. For example, the presence of the dateAdvert and cost attributes on the *Advertises* relationship indicates the presence of an entity called Advert.

11.6 Structural Constraints

We now examine the constraints that may be placed on entity types that participate in a relationship. The constraints should reflect the restrictions on the relationships as perceived in the 'real world'. Examples of such constraints include the requirements that a property for rent must have an owner and each branch must have staff. The main type of constraint on relationships is called **multiplicity**.

Multiplicity The number (or range) of possible occurrences of an entity type that may relate to a single occurrence of an associated entity type through a particular relationship.

Multiplicity constrains the way that entities are related. It is a representation of the policies (or business rules) established by the user or enterprise. Ensuring that all appropriate **constraints** are identified and represented is an important part of modeling an enterprise.

As we mentioned earlier, the most common degree for relationships is binary. Binary relationships are generally referred to as being one-to-one (1:1), one-to-many (1:*), or many-to-many (*:*). We examine these three types of relationships using the following integrity constraints:

- a member of staff manages a branch (1:1);
- a member of staff oversees properties for rent (1:*);
- newspapers advertise properties for rent (*:*).

In Sections 11.6.1, 11.6.2, and 11.6.3 we demonstrate how to determine the multiplicity for each of these constraints and show how to represent each in an ER diagram. In Section 11.6.4 we examine multiplicity for relationships of degrees higher than binary.

It is important to note that not all integrity constraints can be easily represented in an ER model. For example, the requirement that a member of staff receives an additional day's holiday for every year of employment with the enterprise may be difficult to represent in an ER model.

One-to-One (1:1) Relationships 11.6.1

Consider the relationship *Manages*, which relates the Staff and Branch entity types. Figure 11.14(a) displays two occurrences of the *Manages* relationship type (denoted r1 and r2) using a semantic net. Each relationship (r*n*) represents the association between a single Staff entity occurrence and a single Branch entity occurrence. We represent each entity occurrence using the values for the primary key attributes of the Staff and Branch entities, namely staffNo and branchNo.

Determining the multiplicity

Determining the multiplicity normally requires examining the precise relationships between the data given in a enterprise constraint using sample data. The sample data may be obtained by examining filled-in forms or reports and, if possible, from discussion with users. However, it is important to stress that to reach the right conclusions about a constraint requires that the sample data examined or discussed is a true representation of all the data being modeled.

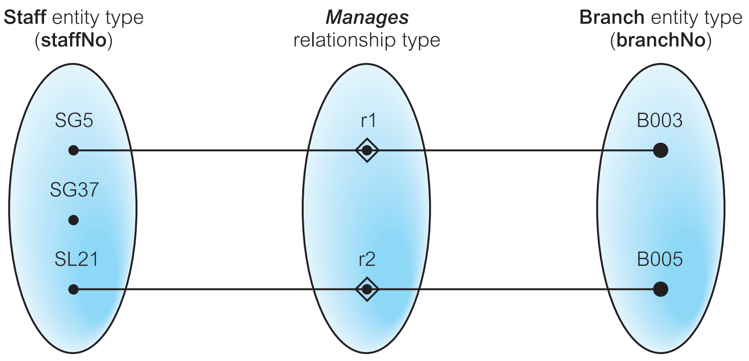
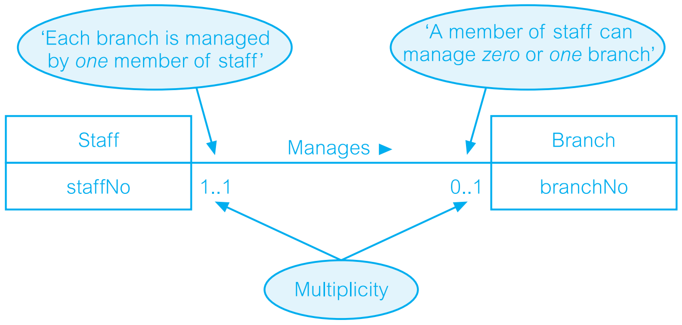

Staff entity type (**staffNo**)
Manages relationship type
Branch entity type (**branchNo**)
SG5
SG37
SL21
r1
r2
B003
B005

Figure 11.14(a) Semantic net showing two occurrences of the Staff *Manages* Branch relationship type.



'Each branch is managed by *one* member of staff'
'A member of staff can manage *zero* or *one* branch'
Staff
staffNo
Manages ▶
1..1
0..1
Branch
branchNo
Multiplicity

Figure 11.14(b) The multiplicity of the Staff *Manages* Branch one-to-one (1:1) relationship.

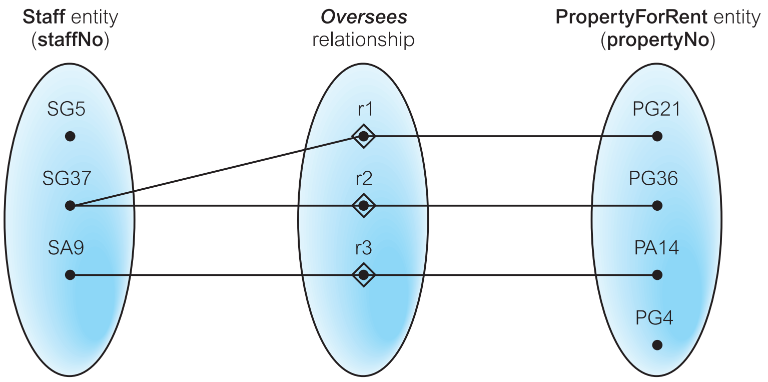
In Figure 11.14(a) we see that staffNo SG5 manages branchNo B003 and staffNo SL21 manages branchNo B005, but staffNo SG37 does not manage any branch. In other words, a member of staff can manage *zero or one* branch and each branch is managed by *one* member of staff. As there is a maximum of one branch for each member of staff involved in this relationship and a maximum of one member of staff for each branch, we refer to this type of relationship as *one-to-one*, which we usually abbreviate as (1:1).

Diagrammatic representation of 1:1 relationships

An ER diagram of the Staff *Manages* Branch relationship is shown in Figure 11.14(b). To represent that a member of staff can manage *zero or one* branch, we place a '0..1' beside the Branch entity. To represent that a branch always has *one* manager, we place a '1..1' beside the Staff entity. (Note that for a 1:1 relationship, we may choose a relationship name that makes sense in either direction.)

11.6.2 One-to-Many (1:*) Relationships

Consider the relationship *Oversees*, which relates the Staff and PropertyForRent entity types. Figure 11.15(a) displays three occurrences of the Staff *Oversees* PropertyForRent relationship



Staff entity (**staffNo**)
Oversees relationship
PropertyForRent entity (**propertyNo**)
SG5
SG37
SA9
r1
r2
r3
PG21
PG36
PA14
PG4

Figure 11.15(a) Semantic net showing three occurrences of the Staff *Oversees* PropertyForRent relationship type.

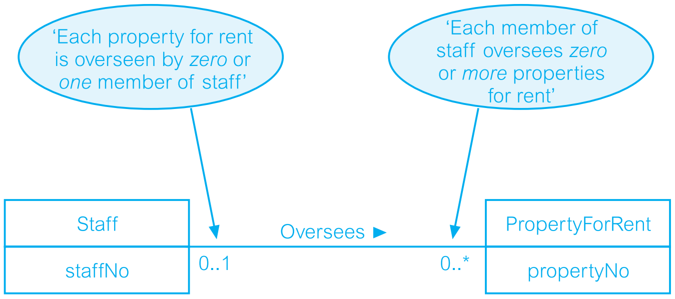

'Each property for rent is overseen by *zero* or *one* member of staff'
'Each member of staff oversees *zero* or *more* properties for rent'
Staff
staffNo
Oversees ►
0..1
0..*
PropertyForRent
propertyNo

Figure 11.15(b) The multiplicity of the Staff *Oversees* PropertyForRent one-to-many (1:*) relationship type.

type (denoted r1, r2, and r3) using a semantic net. Each relationship (r*n*) represents the association between a single Staff entity occurrence and a single PropertyForRent entity occurrence. We represent each entity occurrence using the values for the primary key attributes of the Staff and PropertyForRent entities, namely staffNo and propertyNo.

Determining the multiplicity

In Figure 11.15(a) we see that staffNo SG37 oversees propertyNos PG21 and PG36, and staffNo SA9 oversees propertyNo PA14 but staffNo SG5 does not oversee any properties for rent and propertyNo PG4 is not overseen by any member of staff. In summary, a member of staff can oversee *zero or more* properties for rent and a property for rent is overseen by *zero or one* member of staff. Therefore, for members of staff participating in this relationship there are *many* properties for rent, and for properties participating in this relationship there is a maximum of *one* member of staff. We refer to this type of relationship as *one-to-many*, which we usually abbreviate as (1:*).

Diagrammatic representation of 1:* relationships

An ER diagram of the Staff *Oversees* PropertyForRent relationship is shown in Figure 11.15(b). To represent that a member of staff can oversee *zero or more* properties for rent, we place a '0..*' beside the PropertyForRent entity. To represent that each property for rent is overseen by *zero or one* member of staff, we place a '0..1' beside the Staff entity. (Note that with 1:* relationships, we choose a relationship name that makes sense in the 1:* direction.)

If we know the actual minimum and maximum values for the multiplicity, we can display these instead. For example, if a member of staff oversees a minimum of zero and a maximum of 100 properties for rent, we can replace the '0..*' with '0..100'.

11.6.3 Many-to-Many (*:*) Relationships

Consider the relationship *Advertises*, which relates the Newspaper and PropertyForRent entity types. Figure 11.16(a) displays four occurrences of the *Advertises* relationship (denoted r1, r2, r3, and r4) using a semantic net. Each relationship (r*n*) represents the association between a single Newspaper entity occurrence and a single PropertyForRent entity occurrence.

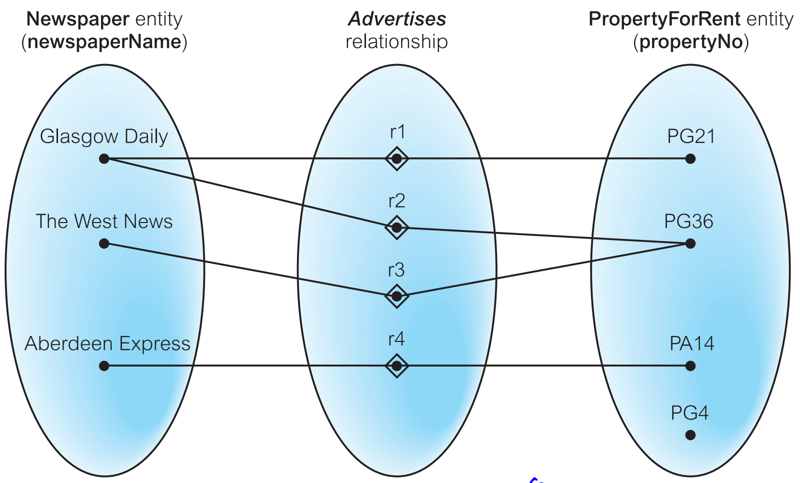


Figure 11.16(a) Semantic net showing four occurrences of the Newspaper *Advertises* PropertyForRent relationship type.

We represent each entity occurrence using the values for the primary key attributes of the Newspaper and PropertyForRent entity types, namely newspaperName and propertyNo.

Determining the multiplicity

In Figure 11.16(a) we see that the Glasgow Daily advertises propertyNos PG21 and PG36, The West News also advertises propertyNo PG36 and the Aberdeen Express advertises propertyNo PA14. However, propertyNo PG4 is not advertised in any newspaper. In other words, *one* newspaper advertises *one or more* properties for rent and *one* property for rent is advertised in *zero or more* newspapers. Therefore, for newspapers there are *many* properties for rent, and for each property for rent participating in this relationship there are *many* newspapers. We refer to this type of relationship as many-to-many, which we usually abbreviate as (*:*).

Diagrammatic representation of *:* relationships

An ER diagram of the Newspaper *Advertises* PropertyForRent relationship is shown in Figure 11.16(b). To represent that each newspaper can advertise *one or more* properties for

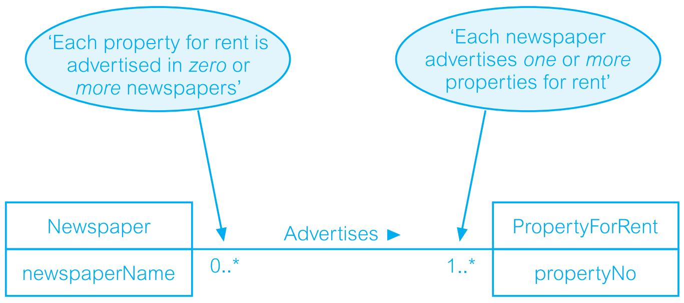

'Each property for rent is advertised in zero or more newspapers'
'Each newspaper advertises one or more properties for rent'
Newspaper
newspaperName
Advertises ►
0..*
1..*
PropertyForRent
propertyNo

Figure 11.16(b) The multiplicity of the Newspaper *Advertises* PropertyForRent many-to-many (*:*) relationship.

rent, we place a '1..*' beside the PropertyForRent entity type. To represent that each property for rent can be advertised by *zero or more* newspapers, we place a '0..*' beside the Newspaper entity. (Note that for a *:* relationship, we may choose a relationship name that makes sense in either direction.)

Multiplicity for Complex Relationships 11.6.4

Multiplicity for complex relationships, that is those higher than binary, is slightly more complex.

Multiplicity (complex relationship) The number (or range) of possible occurrences of an entity type in an *n*-ary relationship when the other (*n*–1) values are fixed.

In general, the multiplicity for *n*-ary relationships represents the potential number of entity occurrences in the relationship when (*n*−1) values are fixed for the other participating entity types. For example, the multiplicity for a ternary relationship represents the potential range of entity occurrences of a particular entity in the relationship when the other two values representing the other two entities are fixed. Consider the ternary *Registers* relationship between Staff, Branch, and Client shown in Figure 11.7. Figure 11.17(a) displays five occurrences of the *Registers* relationship (denoted r1 to r5) using a semantic net. Each relationship (r*n*) represents the association of a single Staff entity occurrence, a single Branch entity occurrence, and a single Client entity occurrence. We represent each entity occurrence using the values for the primary key attributes of the Staff, Branch, and Client entities, namely, staffNo, branchNo, and clientNo. In Figure 11.17(a) we examine the *Registers* relationship when the values for the Staff and Branch entities are fixed.

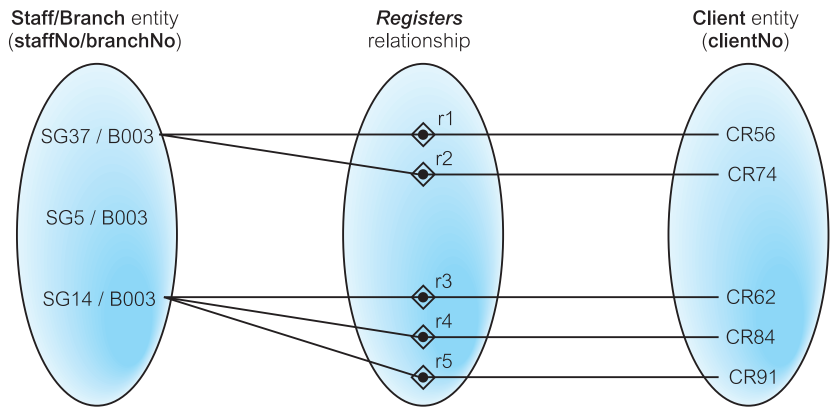


Figure 11.17(a) Semantic net showing five occurrences of the ternary *Registers* relationship with values for Staff and Branch entity types fixed.

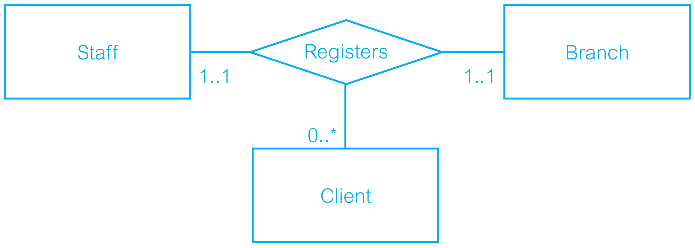


Figure 11.17(b) The multiplicity of the ternary *Registers* relationship.

Table 11.1 A summary of ways to represent multiplicity constraints.

Alternative ways to represent multiplicity constraints	Meaning
0..1	Zero or one entity occurrence
1..1 (or just 1)	Exactly one entity occurrence
0..* (or just *)	Zero or many entity occurrences
1..*	One or many entity occurrences
5..10	Minimum of 5 up to a maximum of 10 entity occurrences
0, 3, 6–8	Zero or three or six, seven, or eight entity occurrences

Determining the multiplicity

In Figure 11.17(a) with the staffNo/branchNo values fixed there are *zero or more* clientNo values. For example, staffNo SG37 at branchNo B003 registers clientNo CR56 and CR74, and staffNo SG14 at branchNo B003 registers clientNo CR62, CR84, and CR91. However, SG5 at branchNo B003 registers no clients. In other words, when the staffNo and branchNo values are fixed the corresponding clientNo values are *zero or more*. Therefore, the multiplicity of the *Registers* relationship from the perspective of the Staff and Branch entities is 0..*, which is represented in the ER diagram by placing the 0..* beside the Client entity.

If we repeat this test we find that the multiplicity when Staff/Client values are fixed is 1..1, which is placed beside the Branch entity and the Client/Branch values are fixed is 1..1, which is placed beside the Staff entity. An ER diagram of the ternary *Registers* relationship showing multiplicity is in Figure 11.17(b).

A summary of the possible ways that multiplicity constraints can be represented along with a description of the meaning is shown in Table 11.1.

11.6.5 Cardinality and Participation Constraints

Multiplicity actually consists of two separate constraints known as cardinality and participation.

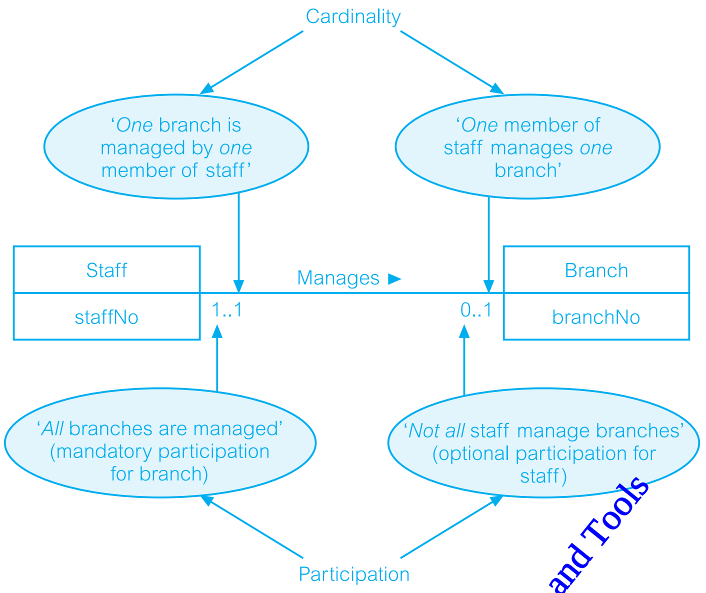

Cardinality
'One branch is managed by one member of staff'
'One member of staff manages one branch'
Staff
staffNo
Manages ►
1..1
0..1
Branch
branchNo
'All branches are managed' (mandatory participation for branch)
'Not all staff manage branches' (optional participation for staff)
Participation

Figure 11.18 Multiplicity described as cardinality and participation constraints for the Staff *Manages* Branch (1:1) relationship.

Cardinality Describes the maximum number of possible relationship occurrences for an entity participating in a given relationship type.

The cardinality of a binary relationship is what we previously referred to as a one-to-one (1:1), one-to-many (1:*), and many-to-many (*:*). The cardinality of a relationship appears as the *maximum* values for the multiplicity ranges on either side of the relationship. For example, the *Manages* relationship shown in Figure 11.18 has a one-to-one (1:1) cardinality and this is represented by multiplicity ranges with a maximum value of 1 on both sides of the relationship.

Participation Determines whether all or only some entity occurrences participate in a relationship.

The participation constraint represents whether all entity occurrences are involved in a particular relationship (referred to as **mandatory** participation) or only some (referred to as **optional** participation). The participation of entities in a relationship appears as the *minimum* values for the multiplicity ranges on either side of the relationship. Optional participation is represented as a minimum value of 0 while mandatory participation is shown as a minimum value of 1. It is important to note that the participation for a given entity in a relationship is represented by the minimum value on the *opposite* side of the relationship; that is the minimum value for the multiplicity beside the related entity. For example, in Figure 11.18, the optional participation for the Staff entity in the *Manages* relationship is shown as a minimum value of 0 for the multiplicity beside the Branch entity and the mandatory participation for the Branch entity in the *Manages* relationship is shown as a minimum value of 1 for the multiplicity beside the Staff entity.

A summary of the conventions introduced in this section to represent the basic concepts of the ER model is shown on the inside front cover of this book.

11.7 Problems with ER Models

In this section we examine problems that may arise when creating an ER model. These problems are referred to as **connection traps**, and normally occur due to a misinterpretation of the meaning of certain relationships (Howe, 1989). We examine two main types of connection traps, called **fan traps** and **chasm traps**, and illustrate how to identify and resolve such problems in ER models.

In general, to identify connection traps we must ensure that the meaning of a relationship is fully understood and clearly defined. If we do not understand the relationships we may create a model that is not a true representation of the 'real world'.

11.7.1 Fan Traps

Fan trap Where a model represents a relationship between entity types, but the pathway between certain entity occurrences is ambiguous.

A fan trap may exist where two or more 1:* relationships fan out from the same entity. A potential fan trap is illustrated in Figure 11.19(a), which shows two 1:* relationships (*Has* and *Operates*) emanating from the same entity called Division.

This model represents the facts that a single division operates *one or more* branches and has *one or more* staff. However, a problem arises when we want to know which members

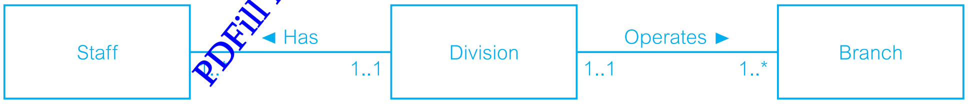


Figure 11.19(a) An example of a fan trap.

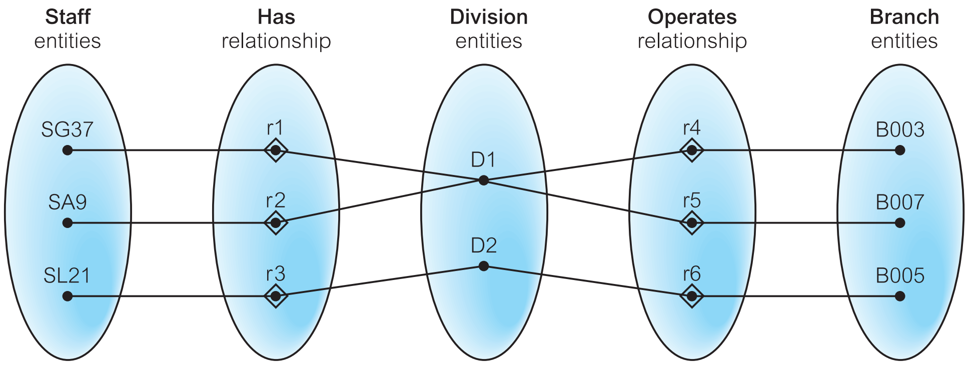


Figure 11.19(b) The semantic net of the ER model shown in Figure 11.19(a).

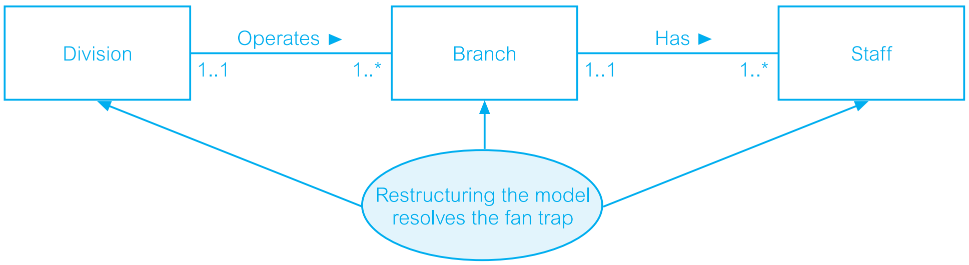


Figure 11.20(a) The ER model shown in Figure 11.19(a) restructured to remove the fan trap.

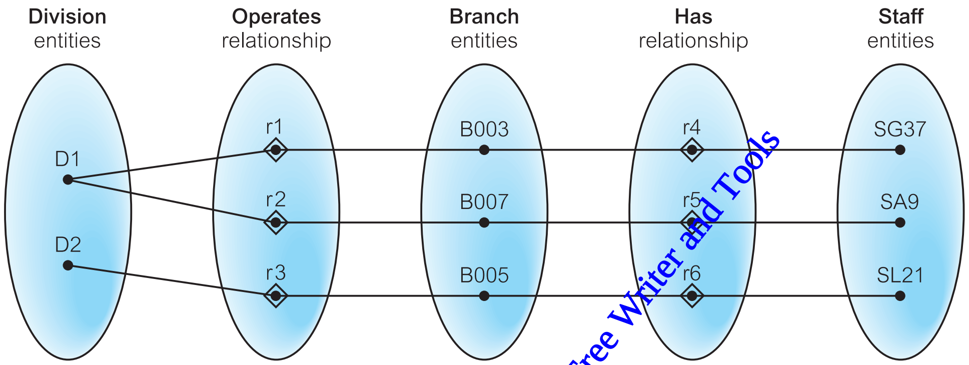


Figure 11.20(b) The semantic net of the ER model shown in Figure 11.20(a).

of staff work at a particular branch. To appreciate the problem, we examine some occurrences of the *Has* and *Operates* relationships using values for the primary key attributes of the Staff, Division, and Branch entity types as shown in Figure 11.19(b).

If we attempt to answer the question: 'At which branch does staff number SG37 work?' we are unable to give a specific answer based on the current structure. We can only determine that staff number SG37 works at Branch B003 *or* B007. The inability to answer this question specifically is the result of a fan trap associated with the misrepresentation of the correct relationships between the Staff, Division, and Branch entities. We resolve this fan trap by restructuring the original ER model to represent the correct association between these entities, as shown in Figure 11.20(a).

If we now examine occurrences of the *Operates* and *Has* relationships as shown in Figure 11.20(b), we are now in a position to answer the type of question posed earlier. From this semantic net model, we can determine that staff number SG37 works at branch number B003, which is part of division D1.

11.7.2 Chasm Traps

Chasm trap Where a model suggests the existence of a relationship between entity types, but the pathway does not exist between certain entity occurrences.

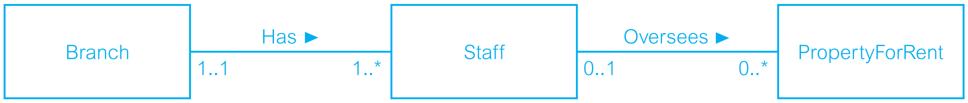


Figure 11.21(a) An example of a chasm trap.

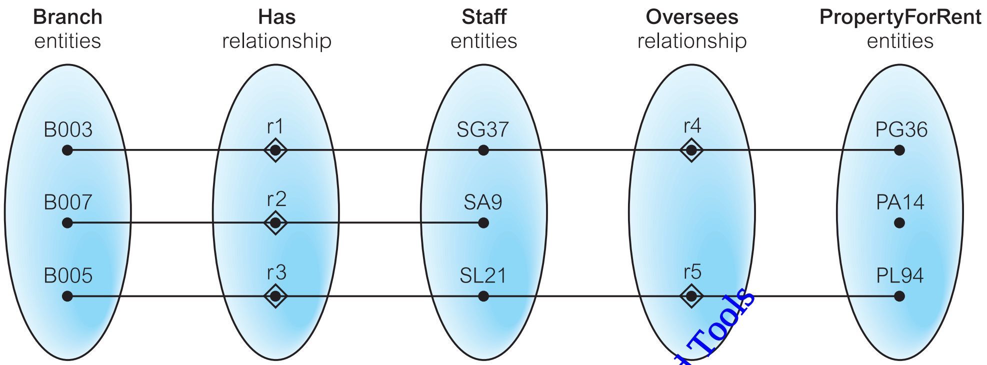


Figure 11.21(b) The semantic net of the ER model shown in Figure 11.21(a).

A chasm trap may occur where there are one or more relationships with a minimum multiplicity of zero (that is optional participation) forming part of the pathway between related entities. A potential chasm trap is illustrated in Figure 11.21(a), which shows relationships between the Branch, Staff, and PropertyForRent entities.

This model represents the facts that a single branch has *one or more* staff who oversee *zero or more* properties for rent. We also note that not all staff oversee property, and not all properties are overseen by a member of staff. A problem arises when we want to know which properties are available at each branch. To appreciate the problem, we examine some occurrences of the *Has* and *Oversees* relationships using values for the primary key attributes of the Branch, Staff, and PropertyForRent entity types as shown in Figure 11.21(b).

If we attempt to answer the question: 'At which branch is property number PA14 available?' we are unable to answer this question, as this property is not yet allocated to a member of staff working at a branch. The inability to answer this question is considered to be a loss of information (as we know a property must be available at a branch), and is the result of a chasm trap. The multiplicity of both the Staff and PropertyForRent entities in the *Oversees* relationship has a minimum value of zero, which means that some properties cannot be associated with a branch through a member of staff. Therefore to solve this problem, we need to identify the missing relationship, which in this case is the *Offers* relationship between the Branch and PropertyForRent entities. The ER model shown in Figure 11.22(a) represents the true association between these entities. This model ensures that, at all times, the properties associated with each branch are known, including properties that are not yet allocated to a member of staff.

If we now examine occurrences of the *Has*, *Oversees*, and *Offers* relationship types, as shown in Figure 11.22(b), we are now able to determine that property number PA14 is available at branch number B007.

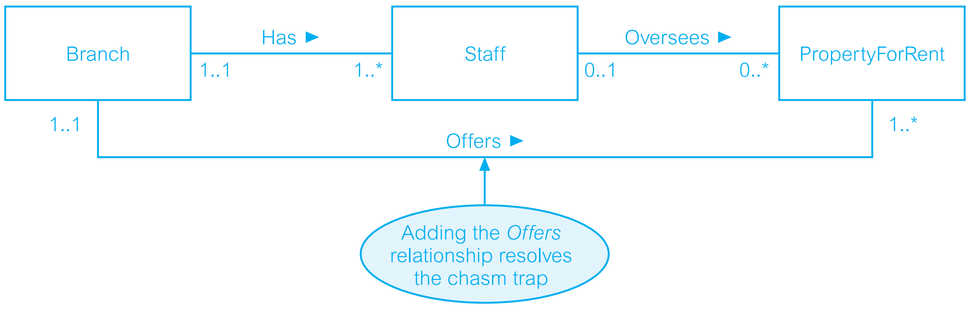


Figure 11.22(a) The ER model shown in Figure 11.21(a) restructured to remove the chasm trap.

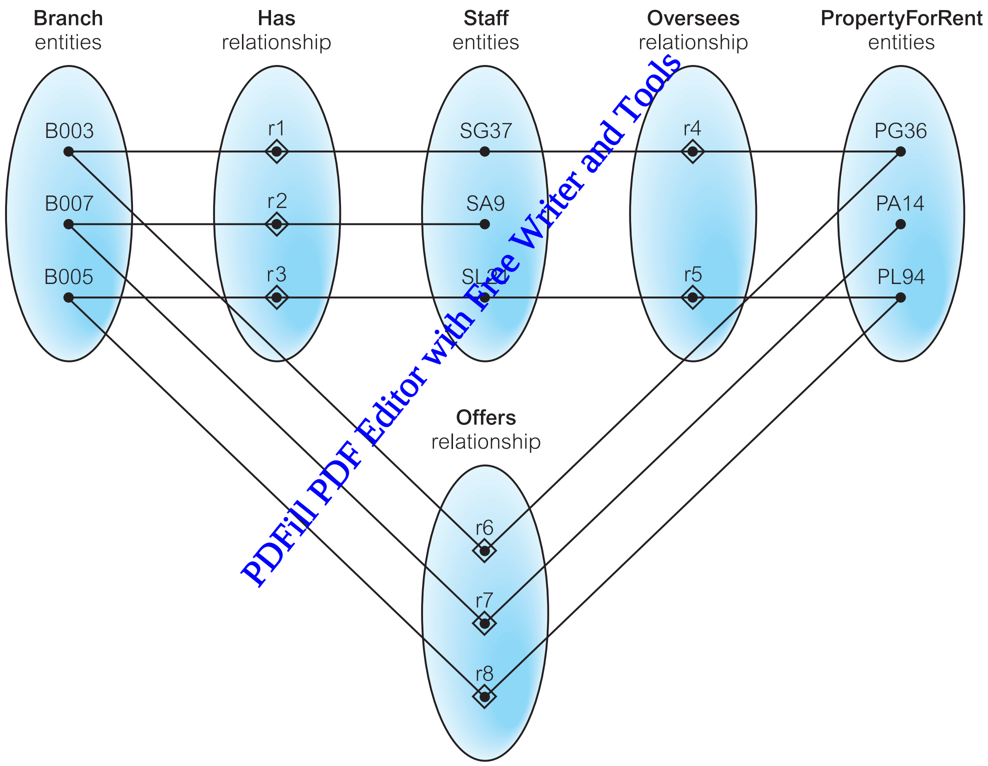


Figure 11.22(b) The semantic net of the ER model shown in Figure 11.22(a).

Chapter Summary

- An **entity type** is a group of objects with the same properties, which are identified by the enterprise as having an independent existence. An **entity occurrence** is a uniquely identifiable object of an entity type.
- A **relationship type** is a set of meaningful associations among entity types. A **relationship occurrence** is a uniquely identifiable association, which includes one occurrence from each participating entity type.
- The **degree of a relationship type** is the number of participating entity types in a relationship.
- A **recursive relationship** is a relationship type where the *same* entity type participates more than once in *different* roles.
- An **attribute** is a property of an entity or a relationship type.
- An **attribute domain** is the set of allowable values for one or more attributes.
- A **simple attribute** is composed of a single component with an independent existence.
- A **composite attribute** is composed of multiple components each with an independent existence.
- A **single-valued attribute** holds a single value for each occurrence of an entity type.
- A **multi-valued attribute** holds multiple values for each occurrence of an entity type.
- A **derived attribute** represents a value that is derivable from the value of a related attribute or set of attributes, not necessarily in the same entity.
- A **candidate key** is the minimal set of attributes that uniquely identifies each occurrence of an entity type.
- A **primary key** is the candidate key that is selected to uniquely identify each occurrence of an entity type.
- A **composite key** is a candidate key that consists of two or more attributes.
- A **strong entity type** is *not* existence-dependent on some other entity type. A **weak entity type** is existence-dependent on some other entity type.
- **Multiplicity** is the number (or range) of possible occurrences of an entity type that may relate to a single occurrence of an associated entity type through a particular relationship.
- **Multiplicity for a complex relationship** is the number (or range) of possible occurrences of an entity type in an *n*-ary relationship when the other ($n-1$) values are fixed.
- **Cardinality** describes the maximum number of possible relationship occurrences for an entity participating in a given relationship type.
- **Participation** determines whether all or only some entity occurrences participate in a given relationship.
- A **fan trap** exists where a model represents a relationship between entity types, but the pathway between certain entity occurrences is ambiguous.
- A **chasm trap** exists where a model suggests the existence of a relationship between entity types, but the pathway does not exist between certain entity occurrences.

Review Questions

11.1 Describe what entity types represent in an ER model and provide examples of entities with a physical or conceptual existence.

11.2 Describe what relationship types represent in an ER model and provide examples of unary, binary, ternary, and quaternary relationships.

11.3 Describe what attributes represent in an ER model and provide examples of simple, composite, single-valued, multi-valued, and derived attributes.

11.4 Describe what the multiplicity constraint represents for a relationship type.

11.5 What are integrity constraints and how does multiplicity model these constraints?

11.6 How does multiplicity represent both the cardinality and the participation constraints on a relationship type?

11.7 Provide an example of a relationship type with attributes.

11.8 Describe how strong and weak entity types differ and provide an example of each.

11.9 Describe how fan and chasm traps can occur in an ER model and how they can be resolved.

Exercises

11.10 Create an ER diagram for each of the following descriptions:

(a) Each company operates four departments, and each department belongs to one company.

(b) Each department in part (a) employs one or more employees, and each employee works for one department.

(c) Each of the employees in part (b) may or may not have one or more dependants, and each dependant belongs to one employee.

(d) Each employee in part (c) may or may not have an employment history.

(e) Represent all the ER diagrams described in (a), (b), (c), and (d) as a single ER diagram.

11.11 You are required to create a conceptual data model of the data requirements for a company that specializes in IT training. The company has 30 instructors and can handle up to 100 trainees per training session. The company offers five advanced technology courses, each of which is taught by a teaching team of two or more instructors. Each instructor is assigned to a maximum of two teaching teams or may be assigned to do research. Each trainee undertakes one advanced technology course per training session.

(a) Identify the main entity types for the company.

(b) Identify the main relationship types and specify the multiplicity for each relationship. State any assumptions you make about the data.

(c) Using your answers for (a) and (b), draw a single ER diagram to represent the data requirements for the company.

11.12 Read the following case study, which describes the data requirements for a video rental company. The video rental company has several branches throughout the USA. The data held on each branch is the branch address made up of street, city, state, and zip code, and the telephone number. Each branch is given a branch number, which is unique throughout the company. Each branch is allocated staff, which includes a Manager. The Manager is responsible for the day-to-day running of a given branch. The data held on a member of staff is his or her name, position, and salary. Each member of staff is given a staff number, which is unique throughout the company. Each branch has a stock of videos. The data held on a video is the catalog number, video number, title, category, daily rental, cost, status, and the names of the main actors and the director. The catalog number uniquely identifies each video. However, in most cases, there are several copies of each video at a branch, and the individual copies are identified using the video number. A video is given a category such as Action, Adult, Children, Drama, Horror, or Sci-Fi. The status indicates whether a specific copy of a video is available for rent. Before hiring a video from the company, a customer must first register as a member of a local branch. The data held on a member is the first and last name, address, and the date that the member registered at a branch. Each member is given a member number, which is unique throughout all branches of the company. Once registered, a member is free to rent videos, up to a maximum of ten at any one time. The data held on each video rented is the rental number, the full name and number of the member, the video number, title, and daily rental, and the dates the video is rented out and returned. The rental number is unique throughout the company.

(a) Identify the main entity types of the video rental company.
(b) Identify the main relationship types between the entity types described in (a) and represent each relationship as an ER diagram.
(c) Determine the multiplicity constraints for each relationships described in (b). Represent the multiplicity for each relationship in the ER diagrams created in (b).
(d) Identify attributes and associate them with entity or relationship types. Represent each attribute in the ER diagrams created in (c).
(e) Determine candidate and primary key attributes for each (strong) entity type.
(f) Using your answers (a) to (e) attempt to represent the data requirements of the video rental company as a single ER diagram. State any assumptions necessary to support your design.

Chapter 12

Enhanced Entity–Relationship Modeling

Chapter Objectives

In this chapter you will learn:

- The limitations of the basic concepts of the Entity–Relationship (ER) model and the requirements to represent more complex applications using additional data modeling concepts.
- The most useful additional data modeling concepts of the Enhanced Entity–Relationship (EER) model called specialization/generalization, aggregation, and composition.
- A diagrammatic technique for displaying specialization/generalization, aggregation, and composition in an EER diagram using the Unified Modeling Language (UML).

In Chapter 11 we discussed the basic concepts of the Entity–Relationship (ER) model. These basic concepts are normally adequate for building data models of traditional, administrative-based database systems such as stock control, product ordering, and customer invoicing. However, since the 1980s there has been a rapid increase in the development of many new database systems that have more demanding database requirements than those of the traditional applications. Examples of such database applications include Computer-Aided Design (CAD), Computer-Aided Manufacturing (CAM), Computer-Aided Software Engineering (CASE) tools, Office Information Systems (OIS) and Multimedia Systems, Digital Publishing, and Geographical Information Systems (GIS). The main features of these applications are described in Chapter 25. As the basic concepts of ER modeling are often not sufficient to represent the requirements of the newer, more complex applications, this stimulated the need to develop additional 'semantic' modeling concepts. Many different semantic data models have been proposed and some of the most important semantic concepts have been successfully incorporated into the original ER model. The ER model supported with additional semantic concepts is called the **Enhanced Entity–Relationship** (EER) model. In this chapter we describe three of the most important and useful additional concepts of the EER model, namely specialization/generalization, aggregation, and composition. We also illustrate how specialization/generalization, aggregation, and composition are represented in an EER diagram using the Unified Modeling Language (UML) (Booch *et al*., 1998). In Chapter 11 we introduced UML and demonstrated how UML could be used to diagrammatically represent the basic concepts of the ER model.

Structure of this Chapter

In Section 12.1 we discuss the main concepts associated with specialization/generalization and illustrate how these concepts are represented in an EER diagram using the Unified Modeling Language (UML). We conclude this section with a worked example that demonstrates how to introduce specialization/generalization into an ER model using UML. In Section 12.2 we describe the concept of aggregation and in Section 12.3 the related concept of composition. We provide examples of aggregation and composition and show how these concepts can be represented in an EER diagram using UML.

12.1 Specialization/Generalization

The concept of specialization/generalization is associated with special types of entities known as **superclasses** and **subclasses**, and the process of **attribute inheritance**. We begin this section by defining what superclasses and subclasses are and by examining superclass/subclass relationships. We describe the process of attribute inheritance and contrast the process of specialization with the process of generalization. We then describe the two main types of constraints on superclass/subclass relationships called participation and disjoint constraints. We show how to represent specialization/generalization in an Enhanced Entity–Relationship (EER) diagram using UML. We conclude this section with a worked example of how specialization/generalization may be introduced into the Entity–Relationship (ER) model of the Branch user views of the *DreamHome* case study described in Appendix A and shown in Figure 11.1.

12.1.1 Superclasses and Subclasses

As we discussed in Chapter 11, an entity type represents a set of entities of the same type such as Staff, Branch, and PropertyForRent. We can also form entity types into a hierarchy containing superclasses and subclasses.

Superclass	An entity type that includes one or more distinct subgroupings of its occurrences, which require to be represented in a data model.

Subclass	A distinct subgrouping of occurrences of an entity type, which require to be represented in a data model.

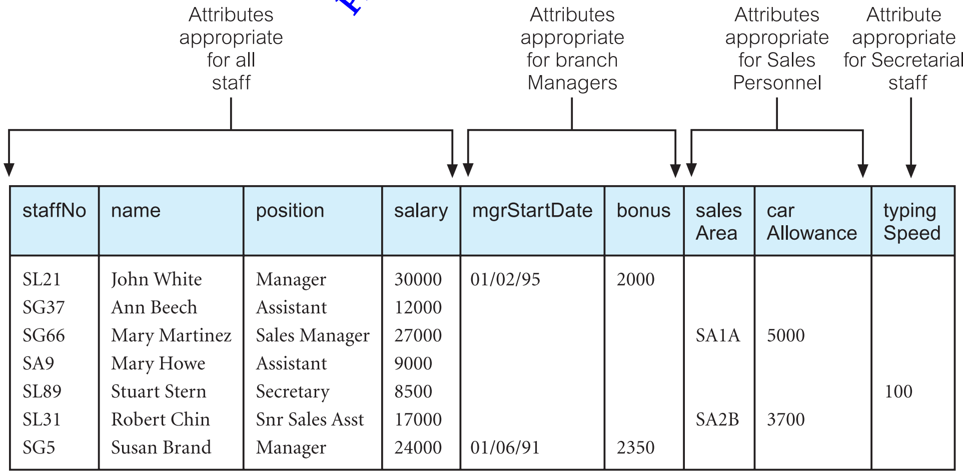
Entity types that have distinct subclasses are called superclasses. For example, the entities that are members of the Staff entity type may be classified as Manager, SalesPersonnel, and Secretary. In other words, the Staff entity is referred to as the **superclass** of the Manager, SalesPersonnel, and Secretary **subclasses**. The relationship between a superclass and any one of its subclasses is called a superclass/subclass relationship. For example, Staff/Manager has a superclass/subclass relationship.

Superclass/Subclass Relationships 12.1.2

Each member of a subclass is also a member of the superclass. In other words, the entity in the subclass is the same entity in the superclass, but has a distinct role. The relationship between a superclass and a subclass is one-to-one (1:1) and is called a superclass/subclass relationship (see Section 11.6.1). Some superclasses may contain overlapping subclasses, as illustrated by a member of staff who is both a Manager and a member of Sales Personnel. In this example, Manager and SalesPersonnel are overlapping subclasses of the Staff superclass. On the other hand, not every member of a superclass need be a member of a subclass; for example, members of staff without a distinct job role such as a Manager or a member of Sales Personnel.

We can use superclasses and subclasses to avoid describing different types of staff with possibly different attributes within a single entity. For example, Sales Personnel may have special attributes such as salesArea and carAllowance. If all staff attributes and those specific to particular jobs are described by a single Staff entity, this may result in a lot of nulls for the job-specific attributes. Clearly, Sales Personnel have common attributes with other staff, such as staffNo, name, position, and salary. However, it is the unshared attributes that cause problems when we try to represent all members of staff within a single entity. We can also show relationships that are only associated with particular types of staff (subclasses) and not with staff, in general. For example, Sales Personnel may have distinct relationships that are not appropriate for all staff, such as SalesPersonnel *Uses* Car.

To illustrate these points, consider the relation called AllStaff shown in Figure 12.1. This relation holds the details of all members of staff no matter what position they hold. A consequence of holding all staff details in one relation is that while the attributes appropriate to all staff are filled (namely, staffNo, name, position, and salary), those that are only applicable


Attributes appropriate for all staff
Attributes appropriate for branch Managers
Attributes appropriate for Sales Personnel
Attribute appropriate for Secretarial staff

staffNo	name	position	salary	mgrStartDate	bonus	sales Area	car Allowance	typing Speed
SL21	John White	Manager	30000	01/02/95	2000			
SG37	Ann Beech	Assistant	12000					
SG66	Mary Martinez	Sales Manager	27000			SA1A	5000	
SA9	Mary Howe	Assistant	9000					
SL89	Stuart Stern	Secretary	8500					100
SL31	Robert Chin	Snr Sales Asst	17000			SA2B	3700	
SG5	Susan Brand	Manager	24000	01/06/91	2350			

Figure 12.1 The AllStaff relation holding details of all staff.

to particular job roles are only partially filled. For example, the attributes associated with the Manager (mgrStartDate and bonus), SalesPersonnel (salesArea and carAllowance), and Secretary (typingSpeed) subclasses have values for those members in these subclasses. In other words, the attributes associated with the Manager, SalesPersonnel, and Secretary subclasses are empty for those members of staff not in these subclasses.

There are two important reasons for introducing the concepts of superclasses and subclasses into an ER model. Firstly, it avoids describing similar concepts more than once, thereby saving time for the designer and making the ER diagram more readable. Secondly, it adds more semantic information to the design in a form that is familiar to many people. For example, the assertions that 'Manager IS-A member of staff' and 'flat IS-A type of property', communicates significant semantic content in a concise form.

12.1.3 Attribute Inheritance

As mentioned above, an entity in a subclass represents the same 'real world' object as in the superclass, and may possess subclass-specific attributes, as well as those associated with the superclass. For example, a member of the SalesPersonnel subclass *inherits* all the attributes of the Staff superclass such as staffNo, name, position, and salary together with those specifically associated with the SalesPersonnel subclass such as salesArea and carAllowance.

A subclass is an entity in its own right and so it may also have one or more subclasses. An entity and its subclasses and their subclasses, and so on, is called a **type hierarchy**. Type hierarchies are known by a variety of names including: **specialization hierarchy** (for example, Manager is a specialization of Staff), **generalization hierarchy** (for example, Staff is a generalization of Manager), and **IS-A hierarchy** (for example, Manager IS-A (member of) Staff). We describe the process of specialization and generalization in the following sections.

A subclass with more than one superclass is called a **shared subclass**. In other words, a member of a shared subclass must be a member of the associated superclasses. As a consequence, the attributes of the superclasses are inherited by the shared subclass, which may also have its own additional attributes. This process is referred to as **multiple inheritance**.

12.1.4 Specialization Process

Specialization	The process of maximizing the differences between members of an entity by identifying their distinguishing characteristics.

Specialization is a top-down approach to defining a set of superclasses and their related subclasses. The set of subclasses is defined on the basis of some distinguishing characteristics of the entities in the superclass. When we identify a set of subclasses of an entity type, we then associate attributes specific to each subclass (where necessary), and also identify any relationships between each subclass and other entity types or subclasses (where necessary). For example, consider a model where all members of staff are represented as an entity called Staff. If we apply the process of specialization on the Staff entity, we attempt to identify differences between members of this entity such as members with distinctive attributes and/or relationships. As described earlier, staff with the job roles of Manager, Sales Personnel, and Secretary have distinctive attributes and therefore we identify Manager, SalesPersonnel, and Secretary as subclasses of a specialized Staff superclass.

Generalization Process 12.1.5

Generalization The process of minimizing the differences between entities by identifying their common characteristics.

The process of generalization is a bottom-up approach, which results in the identification of a generalized superclass from the original entity types. For example, consider a model where Manager, SalesPersonnel, and Secretary are represented as distinct entity types. If we apply the process of generalization on these entities, we attempt to identify similarities between them such as common attributes and relationships. As stated earlier, these entities share attributes common to all staff, and therefore we identify Manager, SalesPersonnel, and Secretary as subclasses of a generalized Staff superclass.

As the process of generalization can be viewed as the reverse of the specialization process, we refer to this modeling concept as 'specialization/generalization'.

Diagrammatic representation of specialization/generalization

UML has a special notation for representing specialization/generalization. For example, consider the specialization/generalization of the Staff entity into subclasses that represent job roles. The Staff superclass and the Manager, SalesPersonnel, and Secretary subclasses can be represented in an Enhanced Entity–Relationship (EER) diagram as illustrated in Figure 12.2. Note that the Staff superclass and the subclasses, being entities, are represented as rectangles. The subclasses are attached by lines to a triangle that points toward the superclass. The label below the specialization/generalization triangle, shown as {Optional, And}, describes the constraints on the relationship between the superclass and its subclasses. These constraints are discussed in more detail in Section 12.1.6.

Attributes that are specific to a given subclass are listed in the lower section of the rectangle representing that subclass. For example, salesArea and carAllowance attributes are only associated with the SalesPersonnel subclass, and are not applicable to the Manager or Secretary subclasses. Similarly, we show attributes that are specific to the Manager (mgrStartDate and bonus) and Secretary (typingSpeed) subclasses.

Attributes that are common to all subclasses are listed in the lower section of the rectangle representing the superclass. For example, staffNo, name, position, and salary attributes are common to all members of staff and are associated with the Staff superclass. Note that we can also show relationships that are only applicable to specific subclasses. For example, in Figure 12.2, the Manager subclass is related to the Branch entity through the

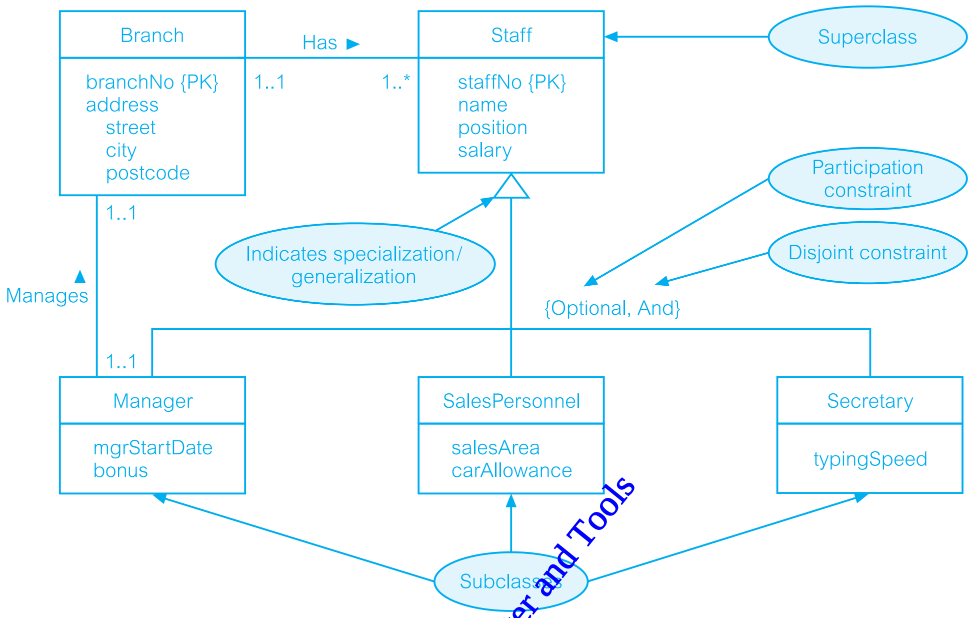


Figure 12.2 Specialization/generalization of the Staff entity into subclasses representing job roles.

Manages relationship, whereas the Staff superclass is related to the Branch entity through the *Has* relationship.

We may have several specializations of the same entity based on different distinguishing characteristics. For example, another specialization of the Staff entity may produce the subclasses FullTimePermanent and PartTimeTemporary, which distinguishes between the types of employment contract for members of staff. The specialization of the Staff entity type into job role and contract of employment subclasses is shown in Figure 12.3. In this figure, we show attributes that are specific to the FullTimePermanent (salaryScale and holidayAllowance) and PartTimeTemporary (hourlyRate) subclasses.

As described earlier, a superclass and its subclasses and their subclasses, and so on, is called a type hierarchy. An example of a type hierarchy is shown in Figure 12.4, where the job roles specialization/generalization shown in Figure 12.2 are expanded to show a shared subclass called SalesManager and the subclass called Secretary with its own subclass called AssistantSecretary. In other words, a member of the SalesManager shared subclass must be a member of the SalesPersonnel and Manager subclasses as well as the Staff superclass. As a consequence, the attributes of the Staff superclass (staffNo, name, position, and salary), and the attributes of the subclasses SalesPersonnel (salesArea and carAllowance) and Manager (mgrStartDate and bonus) are inherited by the SalesManager subclass, which also has its own additional attribute called salesTarget.

AssistantSecretary is a subclass of Secretary, which is a subclass of Staff. This means that a member of the AssistantSecretary subclass must be a member of the Secretary subclass and the Staff superclass. As a consequence, the attributes of the Staff superclass (staffNo, name, position, and salary) and the attribute of the Secretary subclass (typingSpeed) are inherited by the AssistantSecretary subclass, which also has its own additional attribute called startDate.

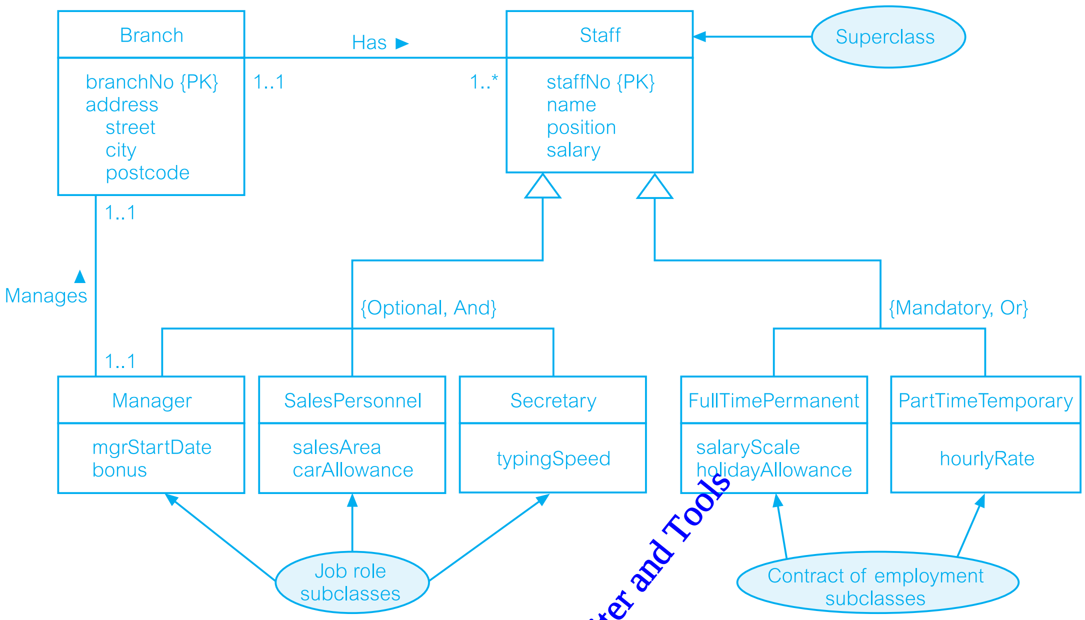


Figure 12.3 Specialization/generalization of the Staff entity into subclasses representing job roles and contracts of employment.

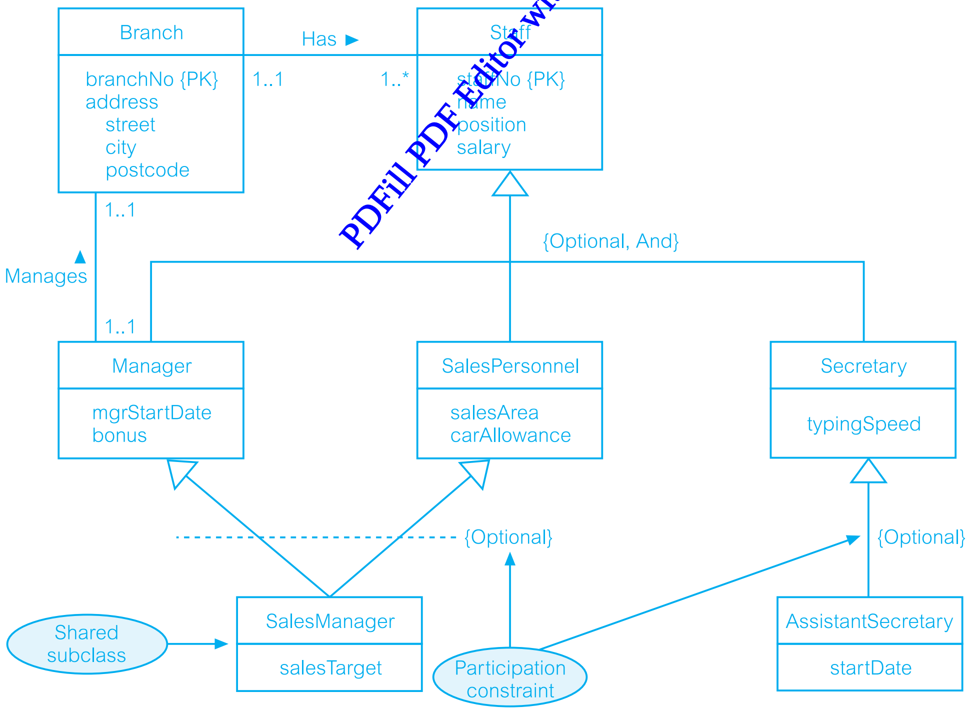


Figure 12.4 Specialization/generalization of the Staff entity into job roles including a shared subclass called SalesManager and a subclass called Secretary with its own subclass called AssistantSecretary.

12.1.6 Constraints on Specialization/Generalization

There are two constraints that may apply to a specialization/generalization called **participation constraints** and **disjoint constraints**.

Participation constraints

Participation constraint	Determines whether every member in the superclass must participate as a member of a subclass.

A participation constraint may be **mandatory** or **optional**. A superclass/subclass relationship with mandatory participation specifies that every member in the superclass must also be a member of a subclass. To represent mandatory participation, 'Mandatory' is placed in curly brackets below the triangle that points towards the superclass. For example, in Figure 12.3 the contract of employment specialization/generalization is mandatory participation, which means that every member of staff must have a contract of employment.

A superclass/subclass relationship with optional participation specifies that a member of a superclass need not belong to any of its subclasses. To represent optional participation, 'Optional' is placed in curly brackets below the triangle that points towards the superclass. For example, in Figure 12.3 the job role specialization/generalization has optional participation, which means that a member of staff need not have an additional job role such as a Manager, Sales Personnel, or Secretary.

Disjoint constraints

Disjoint constraint	Describes the relationship between members of the subclasses and indicates whether it is possible for a member of a superclass to be a member of one, or more than one, subclass.

The disjoint constraint only applies when a superclass has more than one subclass. If the subclasses are **disjoint**, then an entity occurrence can be a member of only one of the subclasses. To represent a disjoint superclass/subclass relationship, 'Or' is placed next to the participation constraint within the curly brackets. For example, in Figure 12.3 the subclasses of the contract of employment specialization/generalization is disjoint, which means that a member of staff must have a full-time permanent *or* a part-time temporary contract, but not both.

If subclasses of a specialization/generalization are not disjoint (called **nondisjoint**), then an entity occurrence may be a member of more than one subclass. To represent a nondisjoint superclass/subclass relationship, 'And' is placed next to the participation constraint within the curly brackets. For example, in Figure 12.3 the job role specialization/generalization is nondisjoint, which means that an entity occurrence can be a member of both the Manager, SalesPersonnel, and Secretary subclasses. This is confirmed by the presence of the shared subclass called SalesManager shown in Figure 12.4. Note that it is not necessary to include the disjoint constraint for hierarchies that have a single subclass at a given level and for this reason only the participation constraint is shown for the SalesManager and AssistantSecretary subclasses of Figure 12.4.

The disjoint and participation constraints of specialization and generalization are distinct, giving rise to four categories: 'mandatory and disjoint', 'optional and disjoint', 'mandatory and nondisjoint', and 'optional and nondisjoint'.

12.1.7 Worked Example of using Specialization/Generalization to Model the Branch View of *DreamHome* Case Study

The database design methodology described in this book includes the use of specialization/generalization as an optional step (Step 1.6) in building an EER model. The choice to use this step is dependent on the complexity of the enterprise (or part of the enterprise) being modeled and whether using the additional concepts of the EER model will help the process of database design.

In Chapter 11 we described the basic concepts necessary to build an ER model to represent the Branch user views of the *DreamHome* case study. This model was shown as an ER diagram in Figure 11.1. In this section, we show how specialization/generalization may be used to convert the ER model of the Branch user views into an EER model.

As a starting point, we first consider the entities shown in Figure 11.1. We examine the attributes and relationships associated with each entity to identify any similarities or differences between the entities. In the Branch user views' requirements specification there are several instances where there is the potential to use specialization/generalization as discussed below.

(a) For example, consider the Staff entity in Figure 11.1, which represents all members of staff. However, in the data requirements specification for the Branch user views of the *DreamHome* case study given in Appendix A, there are two key job roles mentioned namely Manager and Supervisor. We have three options as to how we may best model members of staff. The first option is to represent all members of staff as a generalized Staff entity (as in Figure 11.1), the second option is to create three distinct entities Staff, Manager, and Supervisor, and the third option is to represent the Manager and Supervisor entities as subclasses of a Staff superclass. The option we select is based on the commonality of attributes and relationships associated with each entity. For example, all attributes of the Staff entity are represented in the Manager and Supervisor entities, including the same primary key, namely staffNo. Furthermore, the Supervisor entity does not have any additional attributes representing this job role. On the other hand, the Manager entity has two additional attributes, namely mgrStartDate and bonus. In addition, both the Manager and Supervisor entities are associated with distinct relationships, namely Manager *Manages* Branch and Supervisor *Supervises* Staff. Based on this information, we select the third option and create Manager and Supervisor subclasses of the Staff superclass, as shown in Figure 12.5. Note that in this EER diagram, the subclasses are shown above the superclass. The relative positioning of the subclasses and superclass is not significant, however; what is important is that the specialization/generalization triangle points toward the superclass.

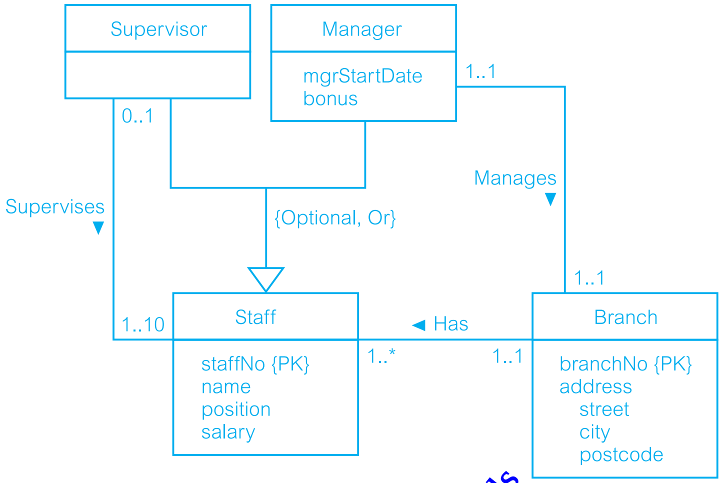


Figure 12.5 Staff superclass with Supervisor and Manager subclasses.

The specialization/generalization of the Staff entity is optional and disjoint (shown as {Optional, Or}), as not all members of staff are Managers or Supervisors, and in addition a single member of staff cannot be both a Manager and a Supervisor. This representation is particularly useful for displaying the shared attributes associated with these subclasses and the Staff superclass and also the distinct relationships associated with each subclass, namely Manager *Manages* Branch and Supervisor *Supervises* Staff.

(b) Next, consider for specialization/generalization the relationship between owners of property. The data requirements specification for the Branch user views describes two types of owner, namely PrivateOwner and BusinessOwner as shown in Figure 11.1. Again, we have three options as to how we may best model owners of property. The first option is to leave PrivateOwner and BusinessOwner as two distinct entities (as shown in Figure 11.1), the second option is to represent both types of owner as a generalized Owner entity, and the third option is to represent the PrivateOwner and BusinessOwner entities as subclasses of an Owner superclass. Before we are able to reach a decision we first examine the attributes and relationships associated with these entities. PrivateOwner and BusinessOwner entities share common attributes, namely address and telNo and have a similar relationship with property for rent (namely PrivateOwner *POwns* PropertyForRent and BusinessOwner *BOwns* PropertyForRent). However, both types of owner also have different attributes; for example, PrivateOwner has distinct attributes ownerNo and name, and BusinessOwner has distinct attributes bName, bType, and contactName. In this case, we create a superclass called Owner, with PrivateOwner and BusinessOwner as subclasses as shown in Figure 12.6.

The specialization/generalization of the Owner entity is mandatory and disjoint (shown as {Mandatory, Or}), as an owner must be either a private owner *or* a business owner, but cannot be both. Note that we choose to relate the Owner superclass to the PropertyForRent entity using the relationship called Owns.

The examples of specialization/generalization described above are relatively straightforward. However, the specialization/generalization process can be taken further as illustrated in the following example.

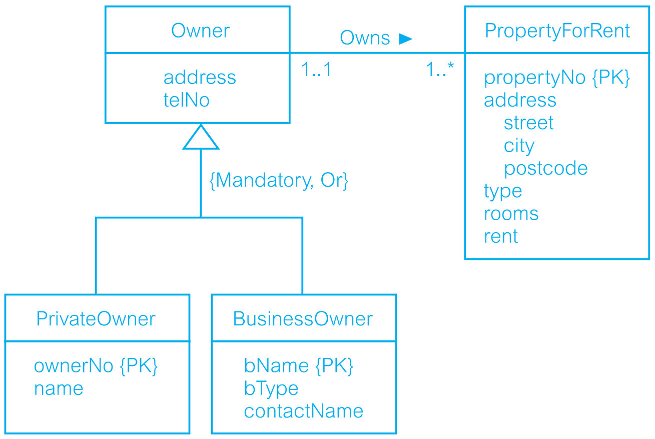


Figure 12.6 Owner superclass with PrivateOwner and BusinessOwner subclasses.

(c) There are several persons with common characteristics described in the data requirements specification for the Branch user views of the *DreamHome* case study. For example, members of staff, private property owners, and clients all have number and name attributes. We could create a Person superclass with Staff (including Manager and Supervisor subclasses), PrivateOwner, and Client as subclasses, as shown in Figure 12.7.

We now consider to what extent we wish to use specialization/generalization to represent the Branch user views of the *DreamHome* case study. We decide to use the specialization/generalization examples described in (a) and (b) above but not (c), as shown in Figure 12.8. To simplify the EER diagram only attributes associated with primary keys or

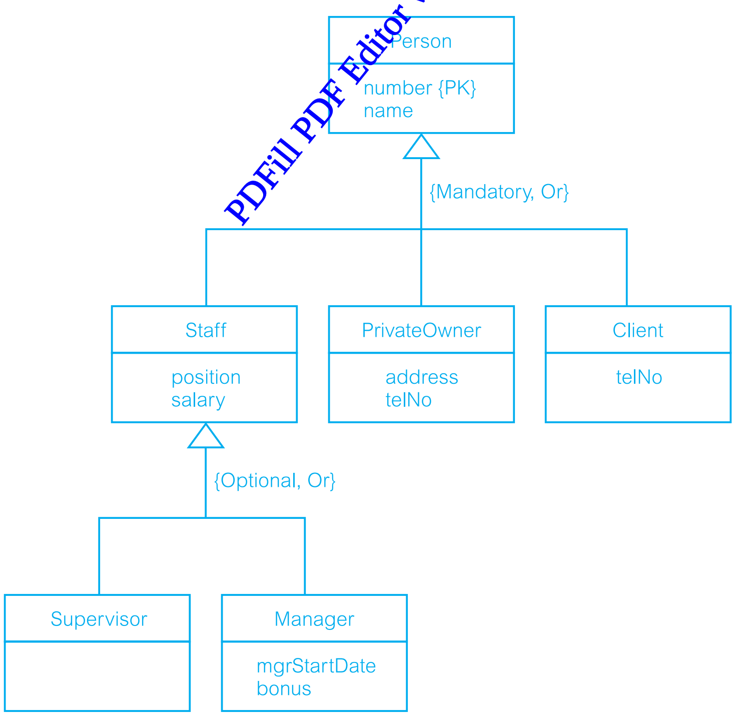


Figure 12.7 Person superclass with Staff (including Supervisor and Manager subclasses), PrivateOwner, and Client subclasses.

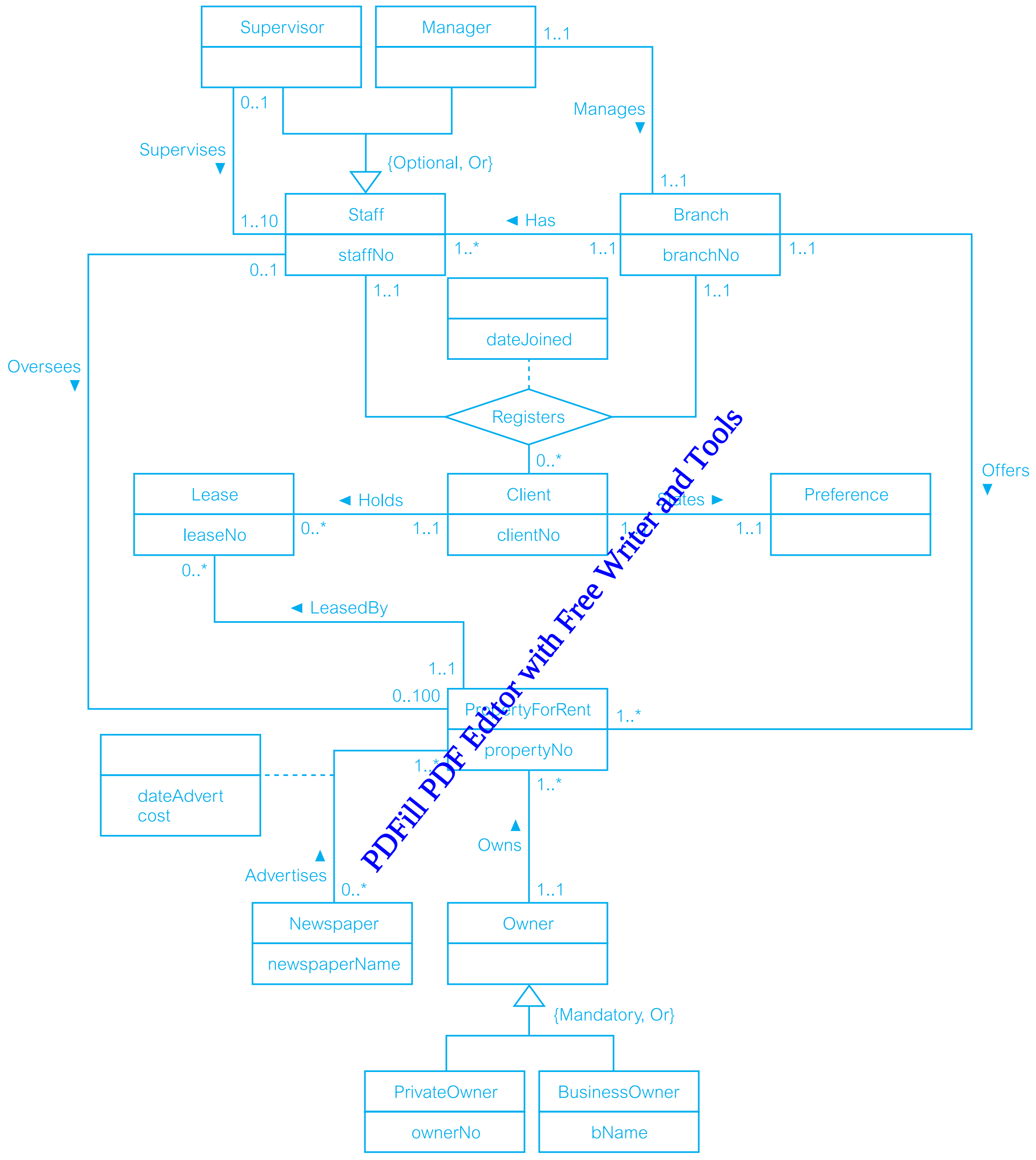


Figure 12.8 An Enhanced Entity–Relationship (EER) model of the Branch user views of *DreamHome* with specialization/generalization.

relationships are shown. We leave out the representation shown in Figure 12.7 from the final EER model because the use of specialization/generalization in this case places too much emphasis on the relationship between entities that are persons rather than emphasizing the relationship between these entities and some of the core entities such as Branch and PropertyForRent.

The option to use specialization/generalization, and to what extent, is a subjective decision. In fact, the use of specialization/generalization is presented as an optional step in our methodology for conceptual database design discussed in Chapter 15, Step 1.6.

As described in Section 2.3, the purpose of a data model is to provide the concepts and notations that allow database designers and end-users to unambiguously and accurately communicate their understanding of the enterprise data. Therefore, if we keep these goals in mind, we should only use the additional concepts of specialization/generalization when the enterprise data is too complex to easily represent using only the basic concepts of the ER model.

At this stage we may consider whether the introduction of specialization/generalization to represent the Branch user views of *DreamHome* is a good idea. In other words, is the requirement specification for the Branch user views better represented as the ER model shown in Figure 11.1 or as the EER model shown in Figure 12.8? We leave this for the reader to consider.

12.2 Aggregation

Aggregation Represents a 'has-a' or 'is-part-of' relationship between entity types, where one represents the 'whole' and the other the 'part'.

A relationship represents an association between two entity types that are conceptually at the same level. Sometimes we want to model a 'has-a' or 'is-part-of' relationship, in which one entity represents a larger entity (the 'whole'), consisting of smaller entities (the 'parts'). This special kind of relationship is called an **aggregation** (Booch *et al*., 1998). Aggregation does not change the meaning of navigation across the relationship between the whole and its parts, nor does it link the lifetimes of the whole and its parts. An example of an aggregation is the *Has* relationship, which relates the Branch entity (the 'whole') to the Staff entity (the 'part').

Diagrammatic representation of aggregation

UML represents aggregation by placing an open diamond shape at one end of the relationship line, next to the entity that represents the 'whole'. In Figure 12.9, we redraw part of the EER diagram shown in Figure 12.8 to demonstrate aggregation. This EER diagram displays two examples of aggregation, namely Branch *Has* Staff and Branch *Offers* PropertyForRent. In both relationships, the Branch entity represents the 'whole' and therefore the open diamond shape is placed beside this entity.

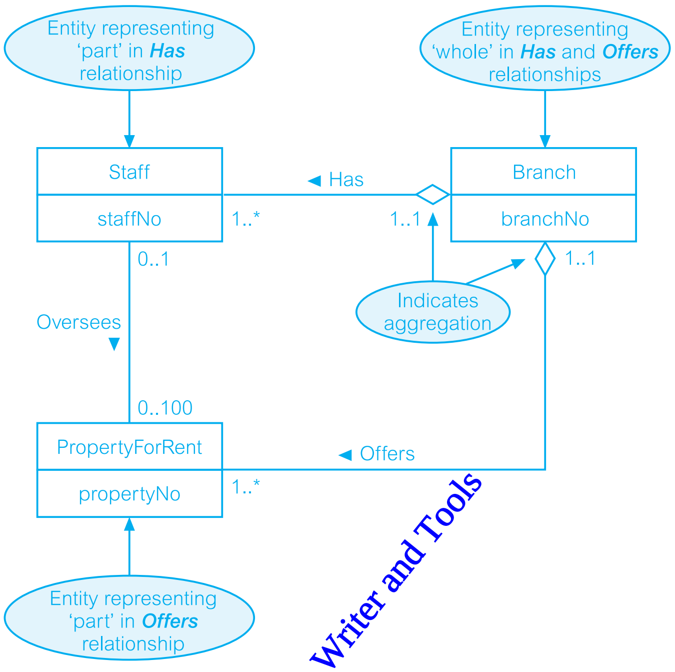

Entity representing 'part' in *Has* relationship
Entity representing 'whole' in *Has* and *Offers* relationships
Staff
staffNo
◀ Has
Branch
branchNo
1..*
1..1
0..1
1..1
Oversees
Indicates aggregation
0..100
PropertyForRent
propertyNo
◀ Offers
1..*
Entity representing 'part' in *Offers* relationship

Figure 12.9 Examples of aggregation: Branch *Has* Staff and Branch *Offers* PropertyForRent.

12.3 Composition

Composition A specific form of aggregation that represents an association between entities, where there is a strong ownership and coincidental lifetime between the 'whole' and the 'part'.

Aggregation is entirely conceptual and does nothing more than distinguish a 'whole' from a 'part'. However, there is a variation of aggregation called **composition** that represents a strong ownership and coincidental lifetime between the 'whole' and the 'part' (Booch *et al*., 1998). In a composite, the 'whole' is responsible for the disposition of the 'parts', which means that the composition must manage the creation and destruction of its 'parts'. In other words, an object may only be part of one composite at a time. There are no examples of composition in Figure 12.8. For the purposes of discussion, consider an example of a composition, namely the *Displays* relationship, which relates the Newspaper entity to the Advert entity. As a composition, this emphasizes the fact that an Advert entity (the 'part') belongs to exactly one Newspaper entity (the 'whole'). This is in contrast to aggregation, in which a part may be shared by many wholes. For example, a Staff entity may be 'a part of' one or more Branches entities.

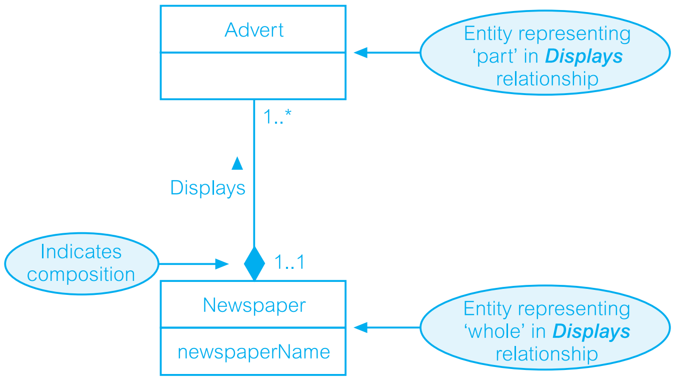

Advert
Entity representing 'part' in *Displays* relationship
1..*
Displays
Indicates composition
1..1
Newspaper
newspaperName
Entity representing 'whole' in *Displays* relationship

Figure 12.10 An example of composition: Newspaper *Displays* Advert.

Diagrammatic representation of composition

UML represents composition by placing a filled-in diamond shape at one end of the relationship line next to the entity that represents the 'whole' in the relationship. For example, to represent the Newspaper *Displays* Advert composition, the filled-in diamond shape is placed next to the Newspaper entity, which is the 'whole' in this relationship, as shown in Figure 12.10.

As discussed with specialization/generalization, the options to use aggregation and composition, and to what extent, are again subjective decisions. Aggregation and composition should only be used when there is a requirement to emphasize special relationships between entity types such as 'has-a' or 'is-part-of', which has implications on the creation, update, and deletion of these closely related entities. We discuss how to represent such constraints between entity types in our methodology for logical database design in Chapter 16, Step 2.4.

If we remember that the major aim of a data model is to unambiguously and accurately communicate an understanding of the enterprise data. We should only use the additional concepts of aggregation and composition when the enterprise data is too complex to easily represent using only the basic concepts of the ER model.

Chapter Summary

- A **superclass** is an entity type that includes one or more distinct subgroupings of its occurrences, which require to be represented in a data model. A **subclass** is a distinct subgrouping of occurrences of an entity type, which require to be represented in a data model.
- **Specialization** is the process of maximizing the differences between members of an entity by identifying their distinguishing features.
- **Generalization** is the process of minimizing the differences between entities by identifying their common features.
- There are two constraints that may apply to a specialization/generalization called **participation constraints** and **disjoint constraints**.

- A **participation constraint** determines whether every member in the superclass must participate as a member of a subclass.
- A **disjoint constraint** describes the relationship between members of the subclasses and indicates whether it is possible for a member of a superclass to be a member of one, or more than one, subclass.
- **Aggregation** represents a 'has-a' or 'is-part-of' relationship between entity types, where one represents the 'whole' and the other the 'part'.
- **Composition** is a specific form of aggregation that represents an association between entities, where there is a strong ownership and coincidental lifetime between the 'whole' and the 'part'.

Review Questions

12.1 Describe what a superclass and a subclass represent.

12.2 Describe the relationship between a superclass and its subclass.

12.3 Describe and illustrate using an example the process of attribute inheritance.

12.4 What are the main reasons for introducing the concepts of superclasses and subclasses into an ER model?

12.5 Describe what a shared subclass represents and how this concept relates to multiple inheritance.

12.6 Describe and contrast the process of specialization with the process of generalization.

12.7 Describe the two main constraints that apply to a specialization/generalization relationship.

12.8 Describe and contrast the concepts of aggregation and composition and provide an example of each.

Exercises

12.9 Consider whether it is appropriate to introduce the enhanced concepts of specialization/generalization, aggregation, and/or composition for the case studies described in Appendix B.

12.10 Consider whether it is appropriate to introduce the enhanced concepts of specialization/generalization, aggregation, and/or composition into the ER model for the case study described in Exercise 11.12. If appropriate, redraw the ER diagram as an EER diagram with the additional enhanced concepts.

Chapter 13 Normalization

Chapter Objectives

In this chapter you will learn:

- The purpose of normalization.
- How normalization can be used when designing a relational database.
- The potential problems associated with redundant data in base relations.
- The concept of functional dependency, which describes the relationship between attributes.
- The characteristics of functional dependencies used in normalization.
- How to identify functional dependencies for a given relation.
- How functional dependencies identify the primary key for a relation.
- How to undertake the process of normalization.
- How normalization uses functional dependencies to group attributes into relations that are in a known normal form.
- How to identify the most commonly used normal forms, namely First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF).
- The problems associated with relations that break the rules of 1NF, 2NF, or 3NF.
- How to represent attributes shown on a form as 3NF relations using normalization.

When we design a database for an enterprise, the main objective is to create an accurate representation of the data, relationships between the data, and constraints on the data that is pertinent to the enterprise. To help achieve this objective, we can use one or more database design techniques. In Chapters 11 and 12 we described a technique called Entity–Relationship (ER) modeling. In this chapter and the next we describe another database design technique called **normalization**.

Normalization is a database design technique, which begins by examining the relationships (called functional dependencies) between attributes. Attributes describe some property of the data or of the relationships between the data that is important to the enterprise. Normalization uses a series of tests (described as normal forms) to help identify the optimal grouping for these attributes to ultimately identify a set of suitable relations that supports the data requirements of the enterprise.

While the main purpose of this chapter is to introduce the concept of functional dependencies and describe normalization up to Third Normal Form (3NF), in Chapter 14 we take a more formal look at functional dependencies and also consider later normal forms that go beyond 3NF.

Structure of this Chapter

In Section 13.1 we describe the purpose of normalization. In Section 13.2 we discuss how normalization can be used to support relational database design. In Section 13.3 we identify and illustrate the potential problems associated with data redundancy in a base relation that is not normalized. In Section 13.4 we describe the main concept associated with normalization called functional dependency, which describes the relationship between attributes. We also describe the characteristics of the functional dependencies that are used in normalization. In Section 13.5 we present an overview of normalization and then proceed in the following sections to describe the process involving the three most commonly used normal forms, namely First Normal Form (1NF) in Section 13.6, Second Normal Form (2NF) in Section 13.7, and Third Normal Form (3NF) in Section 13.8. The 2NF and 3NF described in these sections are based on the *primary key* of a relation. In Section 13.9 we present general definitions for 2NF and 3NF based on all *candidate keys* of a relation.

Throughout this chapter we use examples taken from the *DreamHome* case study described in Section 10.4 and documented in Appendix A.

13.1 The Purpose of Normalization

Normalization A technique for producing a set of relations with desirable properties, given the data requirements of an enterprise.

The purpose of normalization is to identify a suitable set of relations that support the data requirements of an enterprise. The characteristics of a suitable set of relations include the following:

- the *minimal* number of attributes necessary to support the data requirements of the enterprise;
- attributes with a close logical relationship (described as functional dependency) are found in the same relation;
- *minimal* redundancy with each attribute represented only once with the important exception of attributes that form all or part of foreign keys (see Section 3.2.5), which are essential for the joining of related relations.

The benefits of using a database that has a suitable set of relations is that the database will be easier for the user to access and maintain the data, and take up minimal storage space on the computer. The problems associated with using a relation that is not appropriately normalized is described later in Section 13.3.

How Normalization Supports Database Design

Normalization is a formal technique that can be used at any stage of database design. However, in this section we highlight two main approaches for using normalization, as illustrated in Figure 13.1. Approach 1 shows how normalization can be used as a bottom-up standalone database design technique while Approach 2 shows how normalization can be used as a validation technique to check the structure of relations, which may have been created using a top-down approach such as ER modeling. No matter which approach is used the goal is the same that of creating a set of well-designed relations that meet the data requirements of the enterprise.

Figure 13.1 shows examples of data sources that can be used for database design. Although, the users' requirements specification (see Section 9.5) is the preferred data source, it is possible to design a database based on the information taken directly from other data sources such as forms and reports, as illustrated in this chapter and the next.

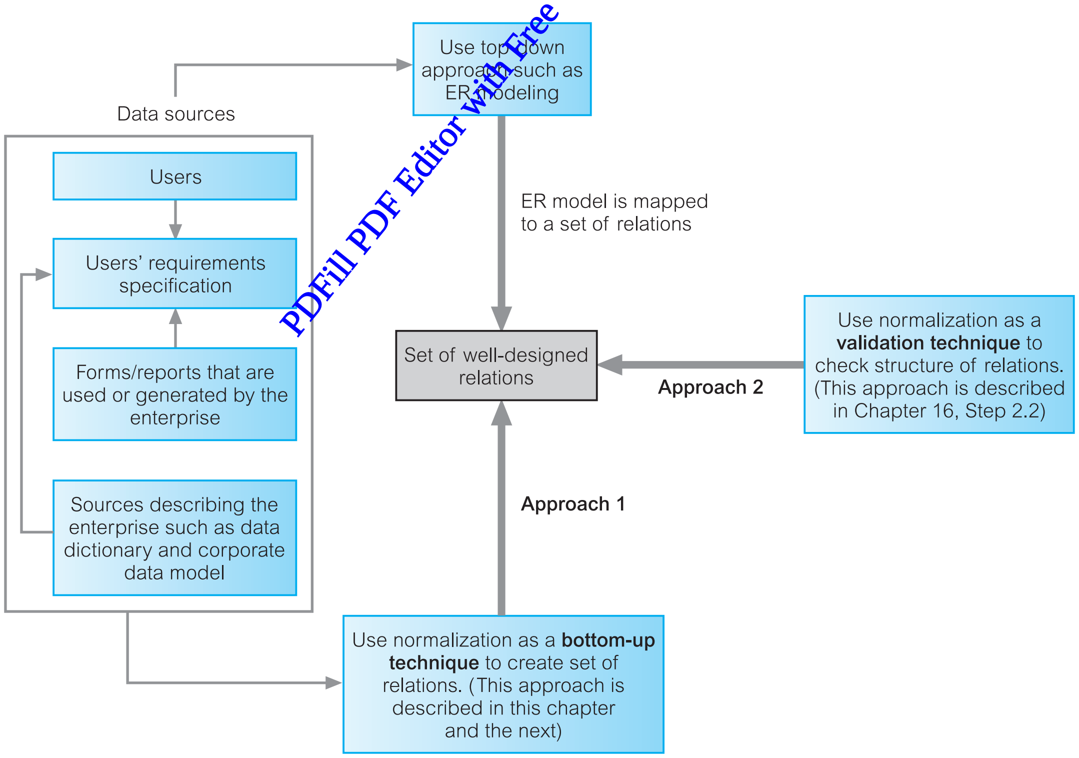

Use top-down approach such as ER modeling
Data sources
Users
Users' requirements specification
Forms/reports that are used or generated by the enterprise
Sources describing the enterprise such as data dictionary and corporate data model
ER model is mapped to a set of relations
Set of well-designed relations
Approach 2
Use normalization as a validation technique to check structure of relations. (This approach is described in Chapter 16, Step 2.2)
Approach 1
Use normalization as a bottom-up technique to create set of relations. (This approach is described in this chapter and the next)

Figure 13.1 How normalization can be used to support database design.

Figure 13.1 also shows that the same data source can be used for both approaches; however, although this is true in principle, in practice the approach taken is likely to be determined by the size, extent, and complexity of the database being described by the data sources and by the preference and expertise of the database designer. The opportunity to use normalization as a bottom-up standalone technique (Approach 1) is often limited by the level of detail that the database designer is reasonably expected to manage. However, this limitation is not applicable when normalization is used as a validation technique (Approach 2) as the database designer focuses on only part of the database, such as a single relation, at any one time. Therefore, no matter what the size or complexity of the database, normalization can be usefully applied.

13.3 Data Redundancy and Update Anomalies

As stated in Section 13.1 a major aim of relational database design is to group attributes into relations to minimize data redundancy. If this aim is achieved, the potential benefits for the implemented database include the following:

- updates to the data stored in the database are achieved with a minimal number of operations thus reducing the opportunities for data inconsistencies occurring in the database;
- reduction in the file storage space required by the base relations thus minimizing costs.

Of course, relational databases also rely on the existence of a certain amount of data redundancy. This redundancy is in the form of copies of primary keys (or candidate keys) acting as foreign keys in related relations to enable the modeling of relationships between data.

In this section we illustrate the problems associated with unwanted data redundancy by comparing the Staff and Branch relations shown in Figure 13.2 with the StaffBranch relation

Figure 13.2 Staff and Branch relations.

Staff

staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003
SA9	Mary Howe	Assistant	9000	B007
SG5	Susan Brand	Manager	24000	B003
SL41	Julie Lee	Assistant	9000	B005

Branch

branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

StaffBranch

staffNo	sName	position	salary	branchNo	bAddress
SL21	John White	Manager	30000	B005	22 Deer Rd, London
SG37	Ann Beech	Assistant	12000	B003	163 Main St, Glasgow
SG14	David Ford	Supervisor	18000	B003	163 Main St, Glasgow
SA9	Mary Howe	Assistant	9000	B007	16 Argyll St, Aberdeen
SG5	Susan Brand	Manager	24000	B003	163 Main St, Glasgow
SL41	Julie Lee	Assistant	9000	B005	22 Deer Rd, London

Figure 13.3 StaffBranch relation.

shown in Figure 13.3. The StaffBranch relation is an alternative format of the Staff and Branch relations. The relations have the form:

Staff	(<u>staffNo</u>, sName, position, salary, branchNo)
Branch	(<u>branchNo</u>, bAddress)
StaffBranch	(<u>staffNo</u>, sName, position, salary, branchNo, bAddress)

Note that the primary key for each relation is underlined.

In the StaffBranch relation there is redundant data; the details of a branch are repeated for every member of staff located at that branch. In contrast, the branch details appear only once for each branch in the Branch relation, and only the branch number (branchNo) is repeated in the Staff relation to represent where each member of staff is located. Relations that have redundant data may have problems called **update anomalies**, which are classified as insertion, deletion, or modification anomalies.

13.3.1 Insertion Anomalies

There are two main types of insertion anomaly, which we illustrate using the StaffBranch relation shown in Figure 13.3.

- To insert the details of new members of staff into the StaffBranch relation, we must include the details of the branch at which the staff are to be located. For example, to insert the details of new staff located at branch number B007, we must enter the correct details of branch number B007 so that the branch details are consistent with values for branch B007 in other tuples of the StaffBranch relation. The relations shown in Figure 13.2 do not suffer from this potential inconsistency because we enter only the appropriate branch number for each staff member in the Staff relation. Instead, the details of branch number B007 are recorded in the database as a single tuple in the Branch relation.
- To insert details of a new branch that currently has no members of staff into the StaffBranch relation, it is necessary to enter nulls into the attributes for staff, such as staffNo. However, as staffNo is the primary key for the StaffBranch relation, attempting to enter nulls for staffNo violates entity integrity (see Section 3.3), and is not allowed. We therefore cannot enter a tuple for a new branch into the StaffBranch relation with a null for the staffNo. The design of the relations shown in Figure 13.2 avoids this problem

because branch details are entered in the Branch relation separately from the staff details. The details of staff ultimately located at that branch are entered at a later date into the Staff relation.

13.3.2 Deletion Anomalies

If we delete a tuple from the StaffBranch relation that represents the last member of staff located at a branch, the details about that branch are also lost from the database. For example, if we delete the tuple for staff number SA9 (Mary Howe) from the StaffBranch relation, the details relating to branch number B007 are lost from the database. The design of the relations in Figure 13.2 avoids this problem, because branch tuples are stored separately from staff tuples and only the attribute branchNo relates the two relations. If we delete the tuple for staff number SA9 from the Staff relation, the details on branch number B007 remain unaffected in the Branch relation.

13.3.3 Modification Anomalies

If we want to change the value of one of the attributes of a particular branch in the StaffBranch relation, for example the address for branch number B003, we must update the tuples of all staff located at that branch. If this modification is not carried out on all the appropriate tuples of the StaffBranch relation, the database will become inconsistent. In this example, branch number B003 may appear to have different addresses in different staff tuples.

The above examples illustrate that the Staff and Branch relations of Figure 13.2 have more desirable properties than the StaffBranch relation of Figure 13.3. This demonstrates that while the StaffBranch relation is subject to update anomalies, we can avoid these anomalies by decomposing the original relation into the Staff and Branch relations. There are two important properties associated with decomposition of a larger relation into smaller relations:

- The **lossless-join** property ensures that any instance of the original relation can be identified from corresponding instances in the smaller relations.
- The **dependency preservation** property ensures that a constraint on the original relation can be maintained by simply enforcing some constraint on each of the smaller relations. In other words, we do not need to perform joins on the smaller relations to check whether a constraint on the original relation is violated.

Later in this chapter, we discuss how the process of normalization can be used to derive well-formed relations. However, we first introduce functional dependencies, which are fundamental to the process of normalization.

13.4 Functional Dependencies

An important concept associated with normalization is **functional dependency**, which describes the relationship between attributes (Maier, 1983). In this section we describe functional dependencies and then focus on the particular characteristics of functional dependencies that are useful for normalization. We then discuss how functional dependencies can be identified and use to identify the primary key for a relation.

Characteristics of Functional Dependencies 13.4.1

For the discussion on functional dependencies, assume that a relational schema has attributes (A, B, C, . . . , Z) and that the database is described by a single **universal relation** called R = (A, B, C, . . . , Z). This assumption means that every attribute in the database has a unique name.

Functional dependency Describes the relationship between attributes in a relation. For example, if A and B are attributes of relation R, B is functionally dependent on A (denoted A → B), if each value of A is associated with exactly one value of B. (A and B may each consist of one or more attributes.)

Functional dependency is a property of the meaning or semantics of the attributes in a relation. The semantics indicate how attributes relate to one another, and specify the functional dependencies between attributes. When a functional dependency is present, the dependency is specified as a **constraint** between the attributes.

Consider a relation with attributes A and B, where attribute B is functionally dependent on attribute A. If we know the value of A and we examine the relation that holds this dependency, we find only one value of B in all the tuples that have a given value of A, at any moment in time. Thus, when two tuples have the same value of A, they also have the same value of B. However, for a given value of B there may be several different values of A. The dependency between attributes A and B can be represented diagrammatically, as shown Figure 13.4.

An alternative way to describe the relationship between attributes A and B is to say that 'A functionally determines B'. Some readers may prefer this description, as it more naturally follows the direction of the functional dependency arrow between the attributes.

Determinant Refers to the attribute, or group of attributes, on the left-hand side of the arrow of a functional dependency.

When a functional dependency exists, the attribute or group of attributes on the left-hand side of the arrow is called the **determinant**. For example, in Figure 13.4, A is the determinant of B. We demonstrate the identification of a functional dependency in the following example.

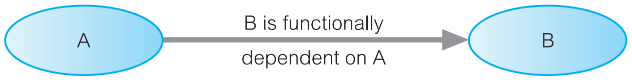


Figure 13.4 A functional dependency diagram.

Example 13.1 An example of a functional dependency

Consider the attributes staffNo and position of the Staff relation in Figure 13.2. For a specific staffNo, for example SL21, we can determine the position of that member of staff as Manager. In other words, staffNo functionally determines position, as shown in Figure 13.5(a). However, Figure 13.5(b) illustrates that the opposite is not true, as position does not functionally determine staffNo. A member of staff holds one position; however, there may be several members of staff with the same position.

The relationship between staffNo and position is one-to-one (1:1): for each staff number there is only one position. On the other hand, the relationship between position and staffNo is one-to-many (1:*): there are several staff numbers associated with a given position. In this example, staffNo is the determinant of this functional dependency. For the purposes of normalization we are interested in identifying functional dependencies between attributes of a relation that have a one-to-one relationship between the attribute(s) that makes up the determinant on the left-hand side and the attribute(s) on the right-hand side of a dependency.

When identifying functional dependencies between attributes in a relation it is important to distinguish clearly between the values held by an attribute at a given point in time and the *set of all possible values* that an attribute may hold at different times. In other words, a functional dependency is a property of a relational schema (intension) and not a property of a particular instance of the schema (extension) (see Section 3.2.1). This point is illustrated in the following example.

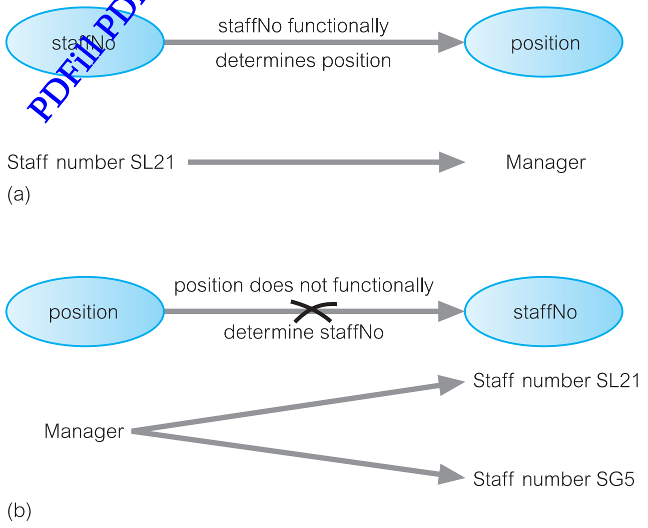


Figure 13.5 (a) staffNo functionally determines position (staffNo → position); (b) position does *not* functionally determine staffNo (position ↛ staffNo).

Example 13.2 Example of a functional dependency that holds for all time

Consider the values shown in staffNo and sName attributes of the Staff relation in Figure 13.2. We see that for a specific staffNo, for example SL21, we can determine the name of that member of staff as John White. Furthermore, it appears that for a specific sName, for example, John White, we can determine the staff number for that member of staff as SL21. Can we therefore conclude that the staffNo attribute functionally determines the sName attribute and/or that the sName attribute functionally determines the staffNo attribute? If the values shown in the Staff relation of Figure 13.2 represent the *set of all possible values* for staffNo and sName attributes then the following functional dependencies hold:

staffNo → sName
sName → staffNo

However, if the values shown in the Staff relation of Figure 13.2 simply represent a *set of values* for staffNo and sName attributes at a given moment in time, then we are not so interested in such relationships between attributes. The reason is that we want to identify functional dependencies that hold for all possible values for attributes of a relation as these represent the types of integrity constraints that we need to identify. Such constraints indicate the limitations on the values that a relation can legitimately assume.

One approach to identifying the set of all possible values for attributes in a relation is to more clearly understand the purpose of each attribute in that relation. For example, the purpose of the values held in the staffNo attribute is to uniquely identify each member of staff, whereas the purpose of the values held in the sName attribute is to hold the names of members of staff. Clearly, the statement that if we know the staff number (staffNo) of a member of staff we can determine the name of the member of staff (sName) remains true. However, as it is possible for the sName attribute to hold duplicate values for members of staff with the same name, then for some members of staff in this category we would not be able to determine their staff number (staffNo). The relationship between staffNo and sName is one-to-one (1:1): for each staff number there is only one name. On the other hand, the relationship between sName and staffNo is one-to-many (1:*): there can be several staff numbers associated with a given name. The functional dependency that remains true after consideration of all possible values for the staffNo and sName attributes of the Staff relation is:

staffNo → sName

An additional characteristic of functional dependencies that is useful for normalization is that their determinants should have the minimal number of attributes necessary to maintain the functional dependency with the attribute(s) on the right hand-side. This requirement is called **full functional dependency**.

Full functional dependency	Indicates that if A and B are attributes of a relation, B is fully functionally dependent on A if B is functionally dependent on A, but not on any proper subset of A.

A functional dependency A → B is a *full* functional dependency if removal of any attribute from A results in the dependency no longer existing. A functional dependency A → B is a **partially dependency** if there is some attribute that can be removed from A and yet the dependency still holds. An example of how a full functional dependency is derived from a partial functional dependency is presented in Example 13.3.

Example 13.3 Example of a full functional dependency

Consider the following functional dependency that exists in the Staff relation of Figure 13.2:

staffNo, sName → branchNo

It is correct to say that each value of (staffNo, sName) is associated with a single value of branchNo. However, it is not a full functional dependency because branchNo is also functionally dependent on a subset of (staffNo, sName), namely staffNo. In other words, the functional dependency shown above is an example of a partial dependency. The type of functional dependency that we are interested in identifying is a full functional dependency as shown below.

staffNo → branchNo

Additional examples of partial and full functional dependencies are discussed in Section 13.7.

In summary, the functional dependencies that we use in normalization have the following characteristics:

- There is a *one-to-one* relationship between the attribute(s) on the left-hand side (determinant) and those on the right-hand side of a functional dependency. (Note that the relationship in the opposite direction, that is from the right- to the left-hand side attributes, can be a one-to-one relationship or one-to-many relationship.)
- They hold for *all* time.
- The determinant has the *minimal* number of attributes necessary to maintain the dependency with the attribute(s) on the right-hand side. In other words, there must be a full functional dependency between the attribute(s) on the left- and right-hand sides of the dependency.

So far we have discussed functional dependencies that we are interested in for the purposes of normalization. However, there is an additional type of functional dependency called a **transitive dependency** that we need to recognize because its existence in a relation can potentially cause the types of update anomaly discussed in Section 13.3. In this section we simply describe these dependencies so that we can identify them when necessary.

Transitive dependency A condition where A, B, and C are attributes of a relation such that if A → B and B → C, then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C).

An example of a transitive dependency is provided in Example 13.4.

Example 13.4 Example of a transitive functional dependency

Consider the following functional dependencies within the StaffBranch relation shown in Figure 13.3:

staffNo → sName, position, salary, branchNo, bAddress
branchNo → bAddress

The transitive dependency branchNo → bAddress exists on staffNo via branchNo. In other words, the staffNo attribute functionally determines the bAddress via the branchNo attribute and neither branchNo nor bAddress functionally determines staffNo. An additional example of a transitive dependency is discussed in Section 13.8.

In the following sections we demonstrate approaches to identifying a set of functional dependencies and then discuss how these dependencies can be used to identify a primary key for the example relations.

13.4.2 Identifying Functional Dependencies

Identifying all functional dependencies between a set of attributes should be quite simple if the meaning of each attribute and the relationships between the attributes are well understood. This type of information may be provided by the enterprise in the form of discussions with users and/or appropriate documentation such as the users' requirements specification. However, if the users are unavailable for consultation and/or the documentation is incomplete, then, depending on the database application, it may be necessary for the database designer to use their common sense and/or experience to provide the missing information. Example 13.5 illustrates how easy it is to identify functional dependencies between attributes of a relation when the purpose of each attribute and the attributes' relationships are well understood.

Example 13.5 Identifying a set of functional dependencies for the StaffBranch relation

We begin by examining the semantics of the attributes in the StaffBranch relation shown in Figure 13.3. For the purposes of discussion we assume that the position held and the branch determine a member of staff's salary. We identify the functional dependencies based on our understanding of the attributes in the relation as:

staffNo $\longrightarrow$ sName, position, salary, branchNo, bAddress
branchNo $\longrightarrow$ bAddress
bAddress $\longrightarrow$ branchNo
branchNo, position $\longrightarrow$ salary
bAddress, position $\longrightarrow$ salary

We identify five functional dependencies in the StaffBranch relation with staffNo, branchNo, bAddress, (branchNo, position), and (bAddress, position) as determinants. For each functional dependency, we ensure that *all* the attributes on the right-hand side are functionally dependent on the determinant on the left-hand side.

As a contrast to this example we now consider the situation where functional dependencies are to be identified in the absence of appropriate information about the meaning of attributes and their relationships. In this case, it may be possible to identify functional dependencies if sample data is available that is a true representation of *all* possible data values that the database may hold. We demonstrate this approach in Example 13.6.

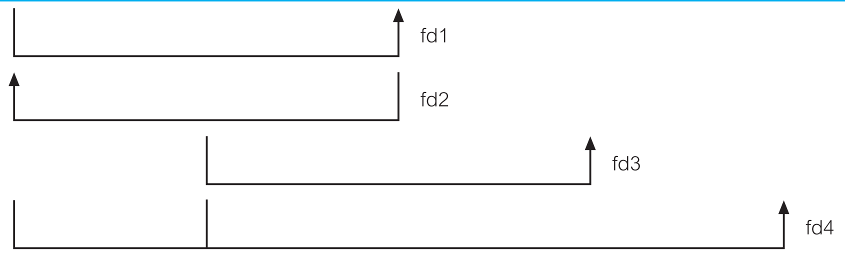
Example 13.6 Using sample data to identify functional dependencies

Consider the data for attributes denoted A, B, C, D, and E in the Sample relation of Figure 13.6. It is important first to establish that the data values shown in this relation are representative of all possible values that can be held by attributes A, B, C, D, and E. For the purposes of this example, let us assume that this is true despite the relatively small amount of data shown in this relation. The process of identifying the functional dependencies (denoted fd1 to fd4) that exist between the attributes of the Sample relation shown in Figure 13.6 is described below.

Figure 13.6 The Sample relation displaying data for attributes A, B, C, D, and E and the functional dependencies (fd1 to fd4) that exist between these attributes.

Sample Relation

A	B	C	D	E
a	b	z	w	q
e	b	r	w	p
a	d	z	w	t
e	d	r	w	q
a	f	z	s	t
e	f	r	s	t



To identify the functional dependencies that exist between attributes A, B, C, D, and E, we examine the Sample relation shown in Figure 13.6 and identify when values in one column are consistent with the presence of particular values in other columns. We begin with the first column on the left-hand side and work our way over to the right-hand side of the relation and then we look at combinations of columns, in other words where values in two or more columns are consistent with the appearance of values in other columns.

For example, when the value 'a' appears in column A the value 'z' appears in column C, and when 'e' appears in column A the value 'r' appears in column C. We can therefore conclude that there is a one-to-one (1:1) relationship between attributes A and C. In other words, attribute A functionally determines attribute C and this is shown as functional dependency 1 (fd1) in Figure 13.6. Furthermore, as the values in column C are consistent with the appearance of particular values in column A, we can also conclude that there is a (1:1) relationship between attributes C and A. In other words, C functionally determines A and this is shown as fd2 in Figure 13.6. If we now consider attribute B, we can see that when 'b' or 'd' appears in column B then 'w' appears in column D and when 'f' appears in column B then 's' appears in column D. We can therefore conclude that there is a (1:1) relationship between attributes B and D. In other words, B functionally determines D and this is shown as fd3 in Figure 13.6. However, attribute D does *not* functionally determine attribute B as a single unique value in column D such as 'w' is not associated with a single consistent value in column B. In other words, when 'w' appears in column D the values 'b' *or* 'd' appears in column B. Hence, there is a one-to-many relationship between attributes D and B. The final single attribute to consider is E and we find that the values in this column are not associated with the consistent appearance of particular values in the other columns. In other words, attribute E does not functionally determine attributes A, B, C, or D.

We now consider combinations of attributes and the appearance of consistent values in other columns. We conclude that unique combination of values in columns A and B such as (a, b) is associated with a single value in column E, which in this example is 'q'. In other words attributes (A, B) functionally determines attribute E and this is shown as fd4 in Figure 13.6. However, the reverse is not true, as we have already stated that attribute E does not functionally determine any other attribute in the relation. We complete the examination of the relation shown in Figure 13.6 by considering all the remaining combinations of columns.

In summary, we describe the function dependencies between attributes A to E in the Sample relation shown in Figure 13.6 as follows:

$A \rightarrow C$	(fd1)
$C \rightarrow A$	(fd2)
$B \rightarrow D$	(fd3)
$A, B \rightarrow E$	(fd4)

13.4.3 Identifying the Primary Key for a Relation using Functional Dependencies

The main purpose of identifying a set of functional dependencies for a relation is to specify the set of integrity constraints that must hold on a relation. An important integrity constraint to consider first is the identification of candidate keys, one of which is selected to be the primary key for the relation. We demonstrate the identification of a primary key for a given relation in the following two examples.

Example 13.7 Identifying the primary key for the StaffBranch relation

In Example 13.5 we describe the identification of five functional dependencies for the StaffBranch relation shown in Figure 13.3. The determinants for these functional dependencies are staffNo, branchNo, bAddress, (branchNo, position), and (bAddress, position).

To identify the candidate key(s) for the StaffBranch relation, we must identify the attribute (or group of attributes) that uniquely identifies each tuple in this relation. If a relation has more than one candidate key, we identify the candidate key that is to act as the primary key for the relation (see Section 3.2.5). All attributes that are not part of the primary key (non-primary-key attributes) should be functionally dependent on the key.

The only candidate key of the StaffBranch relation, and therefore the primary key, is staffNo, as *all* other attributes of the relation are functionally dependent on staffNo. Although branchNo, bAddress, (branchNo, position), and (bAddress, position) are determinants in this relation, they are not candidate keys for the relation.

Example 13.8 Identifying the primary key for the Sample relation

In Example 13.6 we identified four functional dependencies for the Sample relation. We examine the determinant for each functional dependency to identify the candidate key(s) for the relation. A suitable determinant must functionally determine the other attributes in the relation. The determinants in the Sample relation are A, B, C, and (A, B). However, the only determinant that functionally determines all the other attributes of the relation is (A, B). In particular, A functionally determines C, B functionally determines D, and (A, B) functionally determines E. In other words, the attributes that make up the determinant (A, B) can determine all the other attributes in the relation either separately as A or B or together as (A, B). Hence, we see that an essential characteristic for a candidate key of a relation is that the attributes of a determinant either individually or working together must be able to functionally determine *all* the other attributes in the relation. This is not a characteristic of the other determinants in the Sample relation (namely A, B, or C) as in each case they can determine only one other attribute in the relation. As there are no other candidate keys for the Sample relation (A, B) is identified as the primary key for this relation.

So far in this section we have discussed the types of functional dependency that are most useful in identifying important constraints on a relation and how these dependencies can be used to identify a primary key (or candidate keys) for a given relation. The concepts of functional dependencies and keys are central to the process of normalization. We continue the discussion on functional dependencies in the next chapter for readers interested in a more formal coverage of this topic. However, in this chapter, we continue by describing the process of normalization.

13.5 The Process of Normalization

Normalization is a formal technique for analyzing relations based on their primary key (or candidate keys) and functional dependencies (Codd, 1972b). The technique involves a series of rules that can be used to test individual relations so that a database can be normalized to any degree. When a requirement is not met, the relation violating the requirement must be decomposed into relations that individually meet the requirements of normalization.

Three normal forms were initially proposed called First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF). Subsequently, R. Boyce and E.F. Codd introduced a stronger definition of third normal form called Boyce–Codd Normal Form (BCNF) (Codd, 1974). With the exception of 1NF, all these normal forms are based on functional dependencies among the attributes of a relation (Maier, 1983). Higher normal forms that go beyond BCNF were introduced later such as Fourth Normal Form (4NF) and Fifth Normal Form (5NF) (Fagin, 1977, 1979). However, these later normal forms deal with situations that are very rare. In this chapter we describe only the first three normal forms and leave discussions on BCNF, 4NF, and 5NF to the next chapter.

Normalization is often executed as a series of steps. Each step corresponds to a specific normal form that has known properties. As normalization proceeds, the relations become progressively more restricted (stronger) in format and also less vulnerable to update anomalies. For the relational data model, it is important to recognize that it is only First Normal Form (1NF) that is critical in creating relations; all subsequent normal forms are optional. However, to avoid the update anomalies discussed in Section 13.3, it is generally recommended that we proceed to at least Third Normal Form (3NF). Figure 13.7 illustrates the relationship between the various normal forms. It shows that some 1NF relations are also in 2NF and that some 2NF relations are also in 3NF, and so on.

In the following sections we describe the process of normalization in detail. Figure 13.8 provides an overview of the process and highlights the main actions taken in each step of the process. The number of the section that covers each step of the process is also shown in this figure.

In this chapter, we describe normalization as a bottom-up technique extracting information about attributes from sample forms that are first transformed into table format,

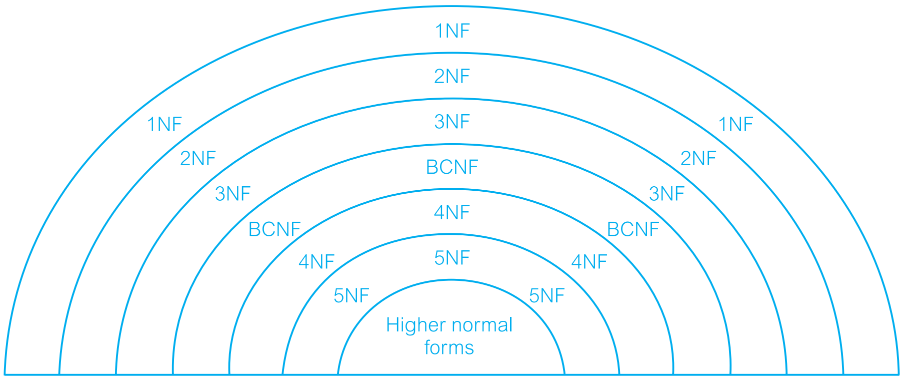
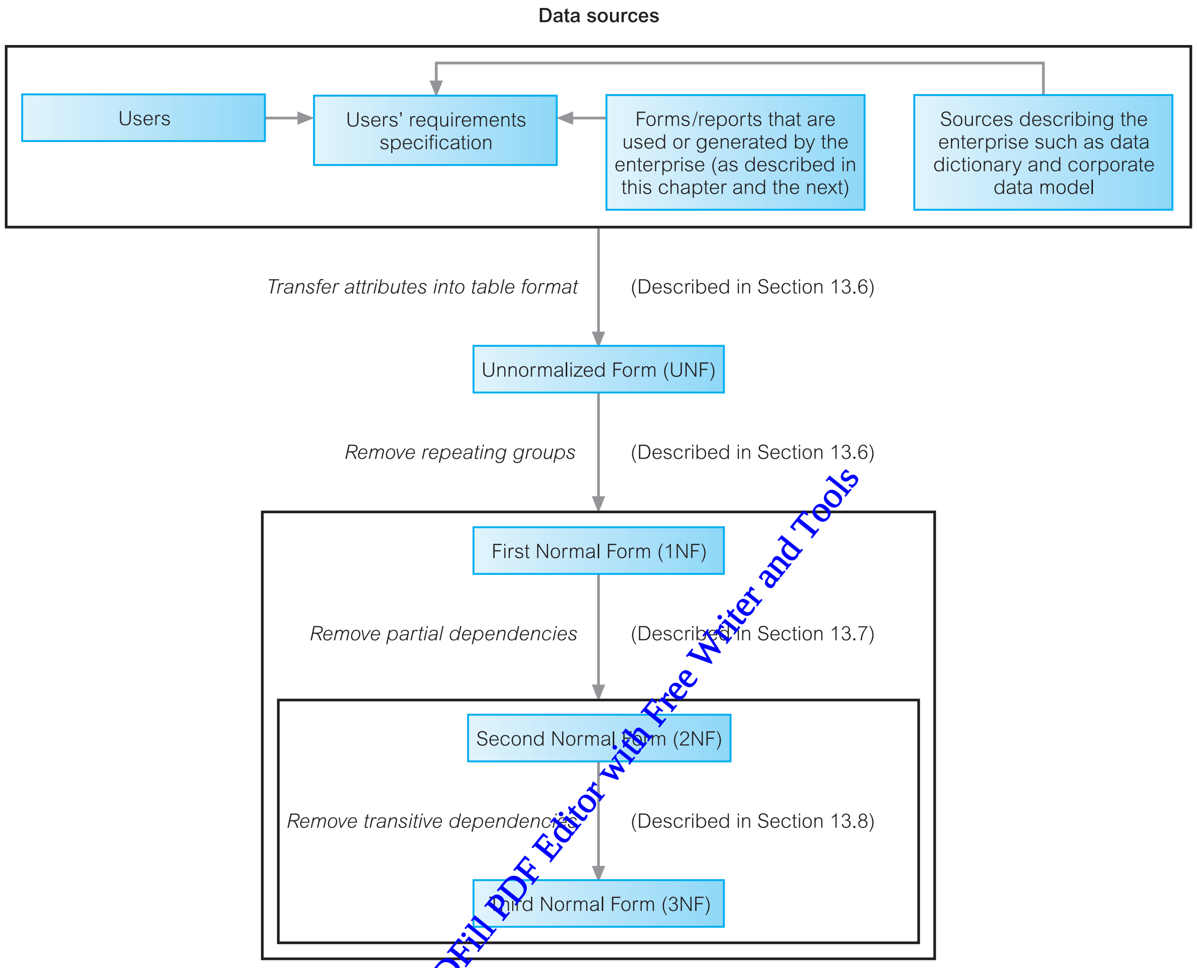


Figure 13.7 Diagrammatic illustration of the relationship between the normal forms.



Data sources
Users
Users' requirements specification
Forms/reports that are used or generated by the enterprise (as described in this chapter and the next)
Sources describing the enterprise such as data dictionary and corporate data model
Transfer attributes into table format
(Described in Section 13.6)
Unnormalized Form (UNF)
Remove repeating groups
(Described in Section 13.6)
First Normal Form (1NF)
Remove partial dependencies
(Described in Section 13.7)
Second Normal Form (2NF)
Remove transitive dependencies
(Described in Section 13.8)
Third Normal Form (3NF)

Figure 13.8 Diagrammatic illustration of the process of normalization.

which is described as being in Unnormalized Form (UNF). This table is then subjected progressively to the different requirements associated with each normal form until ultimately the attributes shown in the original sample forms are represented as a set of 3NF relations. Although the example used in this chapter proceeds from a given normal form to the one above, this is not necessarily the case with other examples. As shown in Figure 13.8, the resolution of a particular problem with, say, a 1NF relation may result in the relation being transformed to 2NF relations or in some cases directly into 3NF relations in one step.

To simplify the description of normalization we assume that a set of functional dependencies is given for each relation in the worked examples and that each relation has a designated primary key. In other words, it is essential that the meaning of the attributes and their relationships is well understood before beginning the process of normalization. This information is fundamental to normalization and is used to test whether a relation is in a particular normal form. In Section 13.6 we begin by describing First Normal Form (1NF). In Sections 13.7 and 13.8 we describe Second Normal Form (2NF) and Third Normal

Forms (3NF) based on the *primary key* of a relation and then present a more general definition of each in Section 13.9. The more general definitions of 2NF and 3NF take into account all *candidate keys* of a relation rather than just the primary key.

13.6 First Normal Form (1NF)

Before discussing First Normal Form, we provide a definition of the state prior to First Normal Form.

Unnormalized Form (UNF) A table that contains one or more repeating groups.

First Normal Form (1NF) A relation in which the intersection of each row and column contains one and only one value.

In this chapter, we begin the process of normalization by first transferring the data from the source (for example, a standard data entry form) into table format with rows and columns. In this format, the table is in Unnormalized Form and is referred to as an **unnormalized table**. To transform the unnormalized table to First Normal Form we identify and remove repeating groups within the table. A repeating group is an attribute, or group of attributes, within a table that occurs with multiple values for a single occurrence of the nominated key attribute(s) for that table. Note that in this context, the term 'key' refers to the attribute(s) that uniquely identify each row within the unnormalized table. There are two common approaches to removing repeating groups from unnormalized tables:

(1) *By entering appropriate data in the empty columns of rows containing the repeating data*. In other words, we fill in the blanks by duplicating the nonrepeating data, where required. This approach is commonly referred to as 'flattening' the table.

(2) *By placing the repeating data, along with a copy of the original key attribute(s), in a separate relation*. Sometimes the unnormalized table may contain more than one repeating group, or repeating groups within repeating groups. In such cases, this approach is applied repeatedly until no repeating groups remain. A set of relations is in 1NF if it contains no repeating groups.

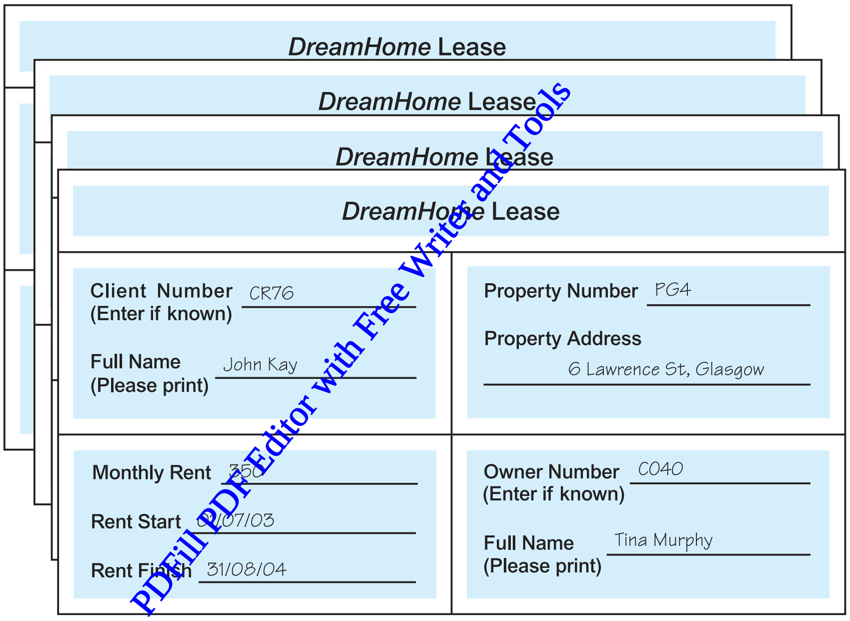
For both approaches, the resulting tables are now referred to as 1NF relations containing atomic (or single) values at the intersection of each row and column. Although both approaches are correct, approach 1 introduces more redundancy into the original UNF table as part of the 'flattening' process, whereas approach 2 creates two or more relations with less redundancy than in the original UNF table. In other words, approach 2 moves the original UNF table further along the normalization process than approach 1. However, no matter which initial approach is taken, the original UNF table will be normalized into the same set of 3NF relations.

We demonstrate both approaches in the following worked example using the *DreamHome* case study.

Example 13.9 First Normal Form (1NF)

A collection of (simplified) *DreamHome* leases is shown in Figure 13.9. The lease on top is for a client called John Kay who is leasing a property in Glasgow, which is owned by Tina Murphy. For this worked example, we assume that a client rents a given property only once and cannot rent more than one property at any one time.

Sample data is taken from two leases for two different clients called John Kay and Aline Stewart and is transformed into table format with rows and columns, as shown in Figure 13.10. This is an example of an unnormalized table.



DreamHome Lease

Client Number (Enter if known) CR76

Full Name (Please print) John Kay

Property Number PG4

Property Address 6 Lawrence St, Glasgow

Monthly Rent 350

Rent Start 01/07/03

Rent Finish 31/08/04

Owner Number (Enter if known) CO40

Full Name (Please print) Tina Murphy

Figure 13.9 Collection of (simplified) *DreamHome* leases.

Figure 13.10 ClientRental unnormalized table.

ClientRental

clientNo	cName	propertyNo	pAddress	rentStart	rentFinish	rent	ownerNo	oName
CR76	John Kay	PG4	6 Lawrence St, Glasgow	1-Jul-03	31-Aug-04	350	CO40	Tina Murphy
		PG16	5 Novar Dr, Glasgow	1-Sep-04	1-Sep-05	450	CO93	Tony Shaw
CR56	Aline Stewart	PG4	6 Lawrence St, Glasgow	1-Sep-02	10-June-03	350	CO40	Tina Murphy
		PG36	2 Manor Rd, Glasgow	10-Oct-03	1-Dec-04	375	CO93	Tony Shaw
		PG16	5 Novar Dr, Glasgow	1-Nov-05	10-Aug-06	450	CO93	Tony Shaw

We identify the key attribute for the ClientRental unnormalized table as clientNo. Next, we identify the repeating group in the unnormalized table as the property rented details, which repeats for each client. The structure of the repeating group is:

Repeating Group = (propertyNo, pAddress, rentStart, rentFinish, rent, ownerNo, oName)

As a consequence, there are multiple values at the intersection of certain rows and columns. For example, there are two values for propertyNo (PG4 and PG16) for the client named John Kay. To transform an unnormalized table into 1NF, we ensure that there is a single value at the intersection of each row and column. This is achieved by removing the repeating group.

With the first approach, we remove the repeating group (property rented details) by entering the appropriate client data into each row. The resulting first normal form ClientRental relation is shown in Figure 13.11.

In Figure 13.12, we present the functional dependencies (fd1 to fd6) for the ClientRental relation. We use the functional dependencies (as discussed in Section 13.4.3) to identify candidate keys for the ClientRental relation as being composite keys comprising (clientNo,

ClientRental

clientNo	propertyNo	cName	pAddress	rentStart	rentFinish	rent	ownerNo	oName
CR76	PG4	John Kay	6 Lawrence St, Glasgow	1-Jul-03	31-Aug-04	350	CO40	Tina Murphy
CR76	PG16	John Kay	5 Novar Dr, Glasgow	1-Sep-04	1-Sep-05	450	CO93	Tony Shaw
CR56	PG4	Aline Stewart	6 Lawrence St, Glasgow	1-Sep-02	10-Jun-03	350	CO40	Tina Murphy
CR56	PG36	Aline Stewart	2 Manor Rd, Glasgow	10-Oct-03	1-Dec-04	375	CO93	Tony Shaw
CR56	PG16	Aline Stewart	5 Novar Dr, Glasgow	1-Nov-05	10-Aug-06	450	CO93	Tony Shaw

Figure 13.11 First Normal Form ClientRental relation.

ClientRental

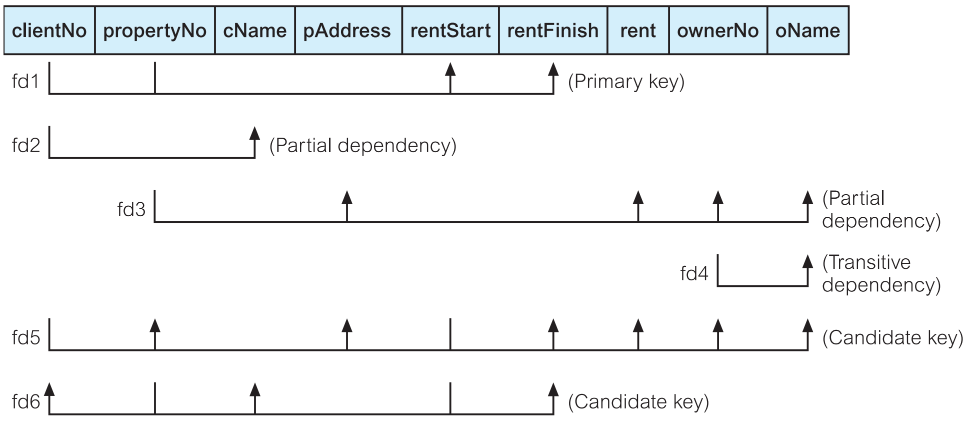


Figure 13.12 Functional dependencies of the ClientRental relation.

Figure 13.13 Alternative 1NF Client and PropertyRentalOwner relations.

Client

clientNo	cName
CR76	John Kay
CR56	Aline Stewart

PropertyRentalOwner

clientNo	propertyNo	pAddress	rentStart	rentFinish	rent	ownerNo	oName
CR76	PG4	6 Lawrence St, Glasgow	1-Jul-03	31-Aug-04	350	CO40	Tina Murphy
CR76	PG16	5 Novar Dr, Glasgow	1-Sep-04	1-Sep-05	450	CO93	Tony Shaw
CR56	PG4	6 Lawrence St, Glasgow	1-Sep-02	10-Jun-03	350	CO40	Tina Murphy
CR56	PG36	2 Manor Rd, Glasgow	10-Oct-03	1-Dec-04	375	CO93	Tony Shaw
CR56	PG16	5 Novar Dr, Glasgow	1-Nov-05	10-Aug-06	450	CO93	Tony Shaw

propertyNo), (clientNo, rentStart), and (propertyNo, rentStart). We select (clientNo, propertyNo) as the primary key for the relation, and for clarity we place the attributes that make up the primary key together at the left-hand side of the relation. In this example, we assume that the rentFinish attribute is not appropriate as a component of a candidate key as it may contain nulls (see Section 3.3.1).

The ClientRental relation is defined as follows:

ClientRental (<u>clientNo</u>, <u>propertyNo</u>, cName, pAddress, rentStart, rentFinish, rent, ownerNo, oName)

The ClientRental relation is in 1NF as there is a single value at the intersection of each row and column. The relation contains data describing clients, property rented, and property owners, which is repeated several times. As a result, the ClientRental relation contains significant data redundancy. If implemented, the 1NF relation would be subject to the update anomalies described in Section 13.3. To remove some of these, we must transform the relation into Second Normal Form, which we discuss shortly.

With the second approach, we remove the repeating group (property rented details) by placing the repeating data along with a copy of the original key attribute (clientNo) in a separate relation, as shown in Figure 13.13.

With the help of the functional dependencies identified in Figure 13.12 we identify a primary key for the relations. The format of the resulting 1NF relations are as follows:

Client	(<u>clientNo</u>, cName)
PropertyRentalOwner	(<u>clientNo</u>, <u>propertyNo</u>, pAddress, rentStart, rentFinish, rent, ownerNo, oName)

The Client and PropertyRentalOwner relations are both in 1NF as there is a single value at the intersection of each row and column. The Client relation contains data describing clients and the PropertyRentalOwner relation contains data describing property rented by clients and property owners. However, as we see from Figure 13.13, this relation also contains some redundancy and as a result may suffer from similar update anomalies to those described in Section 13.3.

To demonstrate the process of normalizing relations from 1NF to 2NF, we use only the ClientRental relation shown in Figure 13.11. However, recall that both approaches are correct, and will ultimately result in the production of the same relations as we continue the process of normalization. We leave the process of completing the normalization of the Client and PropertyRentalOwner relations as an exercise for the reader, which is given at the end of this chapter.

13.7 Second Normal Form (2NF)

Second Normal Form (2NF) is based on the concept of full functional dependency, which we described in Section 13.4. Second Normal Form applies to relations with composite keys, that is, relations with a primary key composed of two or more attributes. A relation with a single-attribute primary key is automatically in at least 2NF. A relation that is not in 2NF may suffer from the update anomalies discussed in Section 13.3. For example, suppose we wish to change the rent of property number PG4. We have to update two tuples in the ClientRental relation in Figure 13.11. If only one tuple is updated with the new rent, this results in an inconsistency in the database.

Second Normal Form (2NF) A relation that is in First Normal Form and every non-primary-key attribute is fully functionally dependent on the primary key.

The normalization of 1NF relations to 2NF involves the removal of partial dependencies. If a partial dependency exists, we remove the partially dependent attribute(s) from the relation by placing them in a new relation along with a copy of their determinant. We demonstrate the process of converting 1NF relations to 2NF relations in the following example.

Example 13.10 Second Normal Form (2NF)

As shown in Figure 13.12, the ClientRental relation has the following functional dependencies:

fd1	clientNo, propertyNo → rentStart, rentFinish	(Primary key)
fd2	clientNo → cName	(Partial dependency)
fd3	propertyNo → pAddress, rent, ownerNo, oName	(Partial dependency)
fd4	ownerNo → oName	(Transitive dependency)
fd5	clientNo, rentStart → propertyNo, pAddress, rentFinish, rent, ownerNo, oName	(Candidate key)
fd6	propertyNo, rentStart → clientNo, cName, rentFinish	(Candidate key)

Using these functional dependencies, we continue the process of normalizing the ClientRental relation. We begin by testing whether the ClientRental relation is in 2NF by identifying the presence of any partial dependencies on the primary key. We note that the

Figure 13.14
Second Normal Form relations derived from the ClientRental relation.

Client

clientNo	cName
CR76	John Kay
CR56	Aline Stewart

Rental

clientNo	propertyNo	rentStart	rentFinish
CR76	PG4	1-Jul-03	31-Aug-04
CR76	PG16	1-Sep-04	1-Sep-05
CR56	PG4	1-Sep-02	10-Jun-03
CR56	PG36	10-Oct-03	1-Dec-04
CR56	PG16	1-Nov-05	10-Aug-06

PropertyOwner

propertyNo	pAddress	rent	ownerNo	oName
PG4	6 Lawrence St, Glasgow	350	CO40	Tina Murphy
PG16	5 Novar Dr, Glasgow	450	CO93	Tony Shaw
PG36	2 Manor Rd, Glasgow	375	CO93	Tony Shaw

client attribute (cName) is partially dependent on the primary key, in other words, on only the clientNo attribute (represented as fd2). The property attributes (pAddress, rent, ownerNo, oName) are partially dependent on the primary key, that is, on only the propertyNo attribute (represented as fd3). The property rented attributes (rentStart and rentFinish) are fully dependent on the whole primary key; that is the clientNo and propertyNo attributes (represented as fd1).

The identification of partial dependencies within the ClientRental relation indicates that the relation is not in 2NF. To transform the ClientRental relation into 2NF requires the creation of new relations so that the non-primary-key attributes are removed along with a copy of the part of the primary key on which they are fully functionally dependent. This results in the creation of three new relations called Client, Rental, and PropertyOwner, as shown in Figure 13.14. These three relations are in Second Normal Form as every non-primary-key attribute is fully functionally dependent on the primary key of the relation. The relations have the following form:

Client	(<u>clientNo</u>, cName)
Rental	(<u>clientNo</u>, <u>propertyNo</u>, rentStart, rentFinish)
PropertyOwner	(<u>propertyNo</u>, pAddress, rent, ownerNo, oName)

13.8 Third Normal Form (3NF)

Although 2NF relations have less redundancy than those in 1NF, they may still suffer from update anomalies. For example, if we want to update the name of an owner, such as Tony Shaw (ownerNo CO93), we have to update two tuples in the PropertyOwner relation of Figure 13.14. If we update only one tuple and not the other, the database would be in an inconsistent state. This update anomaly is caused by a transitive dependency, which we described in Section 13.4. We need to remove such dependencies by progressing to Third Normal Form.

Third Normal Form (3NF) A relation that is in First and Second Normal Form and in which no non-primary-key attribute is transitively dependent on the primary key.

The normalization of 2NF relations to 3NF involves the removal of transitive dependencies. If a transitive dependency exists, we remove the transitively dependent attribute(s) from the relation by placing the attribute(s) in a new relation along with a copy of the determinant. We demonstrate the process of converting 2NF relations to 3NF relations in the following example.

Example 13.11 Third Normal Form (3NF)

The functional dependencies for the Client, Rental, and PropertyOwner relations, derived in Example 13.10, are as follows:

Client		
fd2	clientNo → cName	(Primary key)
Rental		
fd1	clientNo, propertyNo → rentStart, rentFinish	(Primary key)
fd5′	clientNo, rentStart → propertyNo, rentFinish	(Candidate key)
fd6′	propertyNo, rentStart → clientNo, rentFinish	(Candidate key)
PropertyOwner		
fd3	propertyNo → pAddress, rent, ownerNo, oName	(Primary key)
fd4	ownerNo → oName	(Transitive dependency)

All the non-primary-key attributes within the Client and Rental relations are functionally dependent on only their primary keys. The Client and Rental relations have no transitive dependencies and are therefore already in 3NF. Note that where a functional dependency (fd) is labeled with a prime (such as fd5′), this indicates that the dependency has altered compared with the original functional dependency shown in Figure 13.12.

All the non-primary-key attributes within the PropertyOwner relation are functionally dependent on the primary key, with the exception of oName, which is transitively dependent on ownerNo (represented as fd4). This transitive dependency was previously identified in Figure 13.12. To transform the PropertyOwner relation into 3NF we must first remove this transitive dependency by creating two new relations called PropertyForRent and Owner, as shown in Figure 13.15. The new relations have the form:

PropertyForRent	(propertyNo, pAddress, rent, ownerNo)
Owner	(ownerNo, oName)

The PropertyForRent and Owner relations are in 3NF as there are no further transitive dependencies on the primary key.

Figure 13.15 Third Normal Form relations derived from the PropertyOwner relation.

PropertyForRent

propertyNo	pAddress	rent	ownerNo
PG4	6 Lawrence St, Glasgow	350	CO40
PG16	5 Novar Dr, Glasgow	450	CO93
PG36	2 Manor Rd, Glasgow	375	CO93

Owner

ownerNo	oName
CO40	Tina Murphy
CO93	Tony Shaw

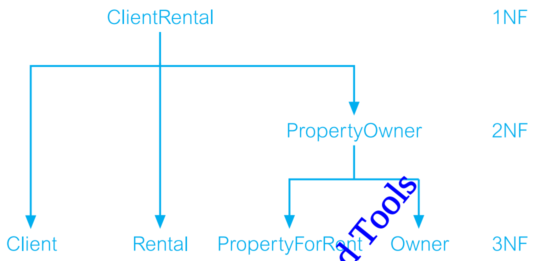


Figure 13.16 The decomposition of the ClientRental 1NF relation into 3NF relations.

The ClientRental relation shown in Figure 13.11 has been transformed by the process of normalization into four relations in 3NF. Figure 13.16 illustrates the process by which the original 1NF relation is decomposed into the 3NF relations. The resulting 3NF relations have the form:

Client	(clientNo, cName)
Rental	(clientNo, propertyNo, rentStart, rentFinish)
PropertyForRent	(propertyNo, pAddress, rent, ownerNo)
Owner	(ownerNo, oName)

The original ClientRental relation shown in Figure 13.11 can be recreated by joining the Client, Rental, PropertyForRent, and Owner relations through the primary key/foreign key mechanism. For example, the ownerNo attribute is a primary key within the Owner relation and is also present within the PropertyForRent relation as a foreign key. The ownerNo attribute acting as a primary key/foreign key allows the association of the PropertyForRent and Owner relations to identify the name of property owners.

The clientNo attribute is a primary key of the Client relation and is also present within the Rental relation as a foreign key. Note in this case that the clientNo attribute in the Rental relation acts both as a foreign key and as part of the primary key of this relation. Similarly, the propertyNo attribute is the primary key of the PropertyForRent relation and is also present within the Rental relation acting both as a foreign key and as part of the primary key for this relation.

In other words, the normalization process has decomposed the original ClientRental relation using a series of relational algebra projections (see Section 4.1). This results in a **lossless-join** (also called *nonloss*- or *nonadditive*-join) decomposition, which is reversible using the natural join operation. The Client, Rental, PropertyForRent, and Owner relations are shown in Figure 13.17.

Client

clientNo	cName
CR76	John Kay
CR56	Aline Stewart

Rental

clientNo	propertyNo	rentStart	rentFinish
CR76	PG4	1-Jul-03	31-Aug-04
CR76	PG16	1-Sep-04	1-Sep-05
CR56	PG4	1-Sep-02	10-Jun-03
CR56	PG36	10-Oct-03	1-Dec-04
CR56	PG16	1-Nov-05	10-Aug-06

PropertyForRent

propertyNo	pAddress	rent	ownerNo
PG4	6 Lawrence St, Glasgow	350	CO40
PG16	5 Novar Dr, Glasgow	450	CO93
PG36	2 Manor Rd, Glasgow	375	CO93

Owner

ownerNo	oName
CO40	Tina Murphy
CO93	Tony Shaw

Figure 13.17 A summary of the 3NF relations derived from the ClientRental relation.

13.9 General Definitions of 2NF and 3NF

The definitions for 2NF and 3NF given in Sections 13.7 and 13.8 disallow partial or transitive dependencies on the *primary key* of relations to avoid the update anomalies described in Section 13.3. However, these definitions do not take into account other candidate keys of a relation, if any exist. In this section, we present more general definitions for 2NF and 3NF that take into account candidate keys of a relation. Note that this requirement does not alter the definition for 1NF as this normal form is independent of keys and functional dependencies. For the general definitions, we define that a candidate-key attribute is part of any candidate key and that partial, full, and transitive dependencies are with respect to all candidate keys of a relation.

Second Normal Form (2NF) A relation that is in First Normal Form and every non-candidate-key attribute is fully functionally dependent on *any candidate key*.

Third Normal Form (3NF) A relation that is in First and Second Normal Form and in which no non-candidate-key attribute is transitively dependent on *any candidate key*.

When using the general definitions of 2NF and 3NF we must be aware of partial and transitive dependencies on all candidate keys and not just the primary key. This can make the process of normalization more complex; however, the general definitions place additional constraints on the relations and may identify hidden redundancy in relations that could be missed.

The tradeoff is whether it is better to keep the process of normalization simpler by examining dependencies on primary keys only, which allows the identification of the most problematic and obvious redundancy in relations, or to use the general definitions and increase the opportunity to identify missed redundancy. In fact, it is often the case that whether we use the definitions based on primary keys or the general definitions of 2NF and 3NF, the decomposition of relations is the same. For example, if we apply the general definitions of 2NF and 3NF to Examples 13.10 and 13.11 described in Sections 13.7 and 13.8, the same decomposition of the larger relations into smaller relations results. The reader may wish to verify this fact.

In the following chapter we re-examine the process of identifying functional dependencies that are useful for normalization and take the process of normalization further by discussing normal forms that go beyond 3NF such as Boyce–Codd Normal Form (BCNF). Also in this chapter we present a second worked example taken from the *DreamHome* case study that reviews the process of normalization from UNF through to BCNF.

Chapter Summary

- **Normalization** is a technique for producing a set of relations with desirable properties, given the data requirements of an enterprise. Normalization is a formal method that can be used to identify relations based on their keys and the functional dependencies among their attributes.
- Relations with data redundancy suffer from **update anomalies**, which can be classified as insertion, deletion, and modification anomalies.
- One of the main concepts associated with normalization is **functional dependency**, which describes the relationship between attributes in a relation. For example, if A and B are attributes of relation R, B is functionally dependent on A (denoted A → B), if each value of A is associated with exactly one value of B. (A and B may each consist of one or more attributes.)
- The **determinant** of a functional dependency refers to the attribute, or group of attributes, on the left-hand side of the arrow.
- The main characteristics of functional dependencies that we use for normalization have a one-to-one relationship between attribute(s) on the left- and right-hand sides of the dependency, hold for all time, and are fully functionally dependent.
- **Unnormalized Form** (**UNF**) is a table that contains one or more repeating groups.
- **First Normal Form** (**1NF**) is a relation in which the intersection of each row and column contains one and only one value.
- **Second Normal Form** (**2NF**) is a relation that is in First Normal Form and every non-primary-key attribute is fully functionally dependent on the *primary key*. **Full functional dependency** indicates that if A and B are attributes of a relation, B is fully functionally dependent on A if B is functionally dependent on A but not on any proper subset of A.
- **Third Normal Form** (**3NF**) is a relation that is in First and Second Normal Form in which no non-primary-key attribute is transitively dependent on the *primary key*. **Transitive dependency** is a condition where A, B, and C are attributes of a relation such that if A → B and B → C, then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C).
- **General definition for Second Normal Form** (**2NF**) is a relation that is in First Normal Form and every non-candidate-key attribute is fully functionally dependent on *any candidate key*. In this definition, a candidate-key attribute is part of any candidate key.
- **General definition for Third Normal Form** (**3NF**) is a relation that is in First and Second Normal Form in which no non-candidate-key attribute is transitively dependent on *any candidate key*. In this definition, a candidate-key attribute is part of any candidate key.

Review Questions

13.1 Describe the purpose of normalizing data.

13.2 Discuss the alternative ways that normalization can be used to support database design.

13.3 Describe the types of update anomaly that may occur on a relation that has redundant data.

13.4 Describe the concept of functional dependency.

13.5 What are the main characteristics of functional dependencies that are used for normalization?

13.6 Describe how a database designer typically identifies the set of functional dependencies associated with a relation.

13.7 Describe the characteristics of a table in Unnormalized Form (UNF) and describe how such a table is converted to a First Normal Form (1NF) relation.

13.8 What is the minimal normal form that a relation must satisfy? Provide a definition for this normal form.

13.9 Describe the two approaches to converting an Unnormalized Form (UNF) table to First Normal Form (1NF) relation(s).

13.10 Describe the concept of full functional dependency and describe how this concept relates to 2NF. Provide an example to illustrate your answer.

13.11 Describe the concept of transitive dependency and describe how this concept relates to 3NF. Provide an example to illustrate your answer.

13.12 Discuss how the definitions of 2NF and 3NF based on primary keys differ from the general definitions of 2NF and 3NF. Provide an example to illustrate your answer.

Exercises

13.13 Continue the process of normalizing the Client and PropertyRentalOwner 1NF relations shown in Figure 13.13 to 3NF relations. At the end of this process check that the resultant 3NF relations are the same as those produced from the alternative ClientRental 1NF relation shown in Figure 13.16.

13.14 Examine the Patient Medication Form for the *Wellmeadows Hospital* case study shown in Figure 13.18.

(a) Identify the functional dependencies represented by the attributes shown in the form in Figure 13.18. State any assumptions you make about the data and the attributes shown in this form.

(b) Describe and illustrate the process of normalizing the attributes shown in Figure 13.18 to produce a set of well-designed 3NF relations.

(c) Identify the primary, alternate, and foreign keys in your 3NF relations.

13.15 The table shown in Figure 13.19 lists sample dentist/patient appointment data. A patient is given an appointment at a specific time and date with a dentist located at a particular surgery. On each day of patient appointments, a dentist is allocated to a specific surgery for that day.

(a) The table shown in Figure 13.19 is susceptible to update anomalies. Provide examples of insertion, deletion, and update anomalies.

(b) Identify the functional dependencies represented by the attributes shown in the table of Figure 13.19. State any assumptions you make about the data and the attributes shown in this table.

(c) Describe and illustrate the process of normalizing the table shown in Figure 13.19 to 3NF relations. Identify the primary, alternate, and foreign keys in your 3NF relations.

13.16 An agency called *Instant Cover* supplies part-time/temporary staff to hotels within Scotland. The table shown in Figure 13.20 displays sample data, which lists the time spent by agency staff working at various hotels. The National Insurance Number (NIN) is unique for every member of staff.

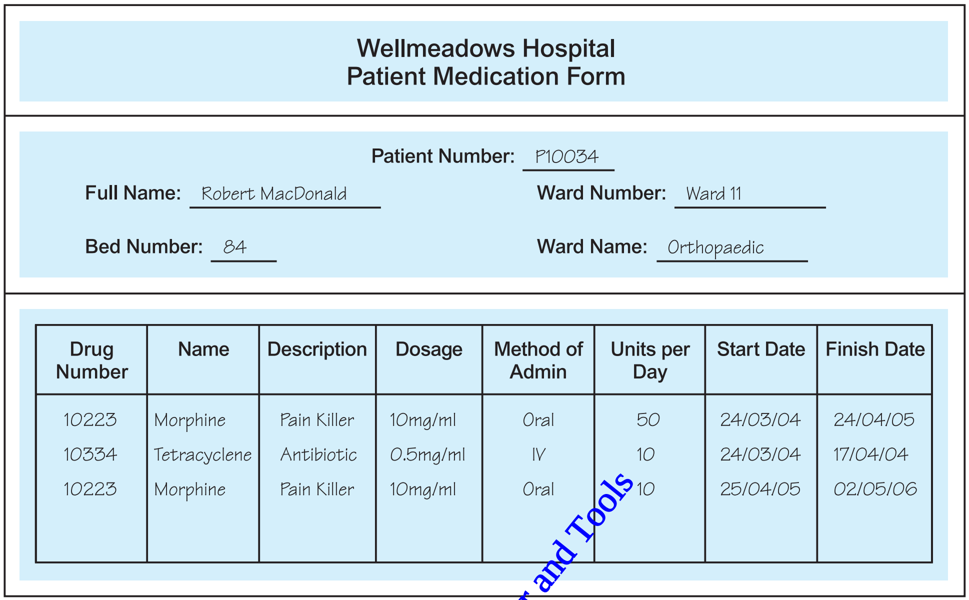


Figure 13.18 The *Wellmeadows Hospital* Patient Medication Form.

Wellmeadows Hospital
Patient Medication Form

Patient Number: P10034

Full Name: Robert MacDonald **Ward Number:** Ward 11

Bed Number: 84 **Ward Name:** Orthopaedic

Drug Number	Name	Description	Dosage	Method of Admin	Units per Day	Start Date	Finish Date
10223	Morphine	Pain Killer	10mg/ml	Oral	50	24/03/04	24/04/05
10334	Tetracyclene	Antibiotic	0.5mg/ml	IV	10	24/03/04	17/04/04
10223	Morphine	Pain Killer	10mg/ml	Oral	10	25/04/05	02/05/06

Figure 13.19 Table displaying sample dentist/patient appointment data.

staffNo	dentistName	patNo	patName	appointment		surgeryNo
				date	time	
S1011	Tony Smith	P100	Gillian White	12-Sep-04	10.00	S15
S1011	Tony Smith	P105	Jill Bell	12-Sep-04	12.00	S15
S1024	Helen Pearson	P108	Ian MacKay	12-Sep-04	10.00	S10
S1024	Helen Pearson	P108	Ian MacKay	14-Sep-04	14.00	S10
S1032	Robin Plevin	P105	Jill Bell	14-Sep-04	16.30	S15
S1032	Robin Plevin	P110	John Walker	15-Sep-04	18.00	S13

Figure 13.20 Table displaying sample data for the *Instant Cover* agency.

NIN	contractNo	hours	eName	hNo	hLoc
1135	C1024	16	Smith J	H25	East Kilbride
1057	C1024	24	Hocine D	H25	East Kilbride
1068	C1025	28	White T	H4	Glasgow
1135	C1025	15	Smith J	H4	Glasgow

(a) The table shown in Figure 13.20 is susceptible to update anomalies. Provide examples of insertion, deletion, and update anomalies.

(b) Identify the functional dependencies represented by the attributes shown in the table of Figure 13.20. State any assumptions you make about the data and the attributes shown in this table.

(c) Describe and illustrate the process of normalizing the table shown in Figure 13.20 to 3NF. Identify primary, alternate and foreign keys in your relations.

Chapter 14

Advanced Normalization

Chapter Objectives

In this chapter you will learn:

- How inference rules can identify a set of *all* functional dependencies for a relation.
- How inference rules called Armstrong's axioms can identify a *minimal* set of useful functional dependencies from the set of all functional dependencies for a relation.
- Normal forms that go beyond Third Normal Form (3NF), which includes Boyce–Codd Normal Form (BCNF), Fourth Normal Form (4NF), and Fifth Normal Form (5NF).
- How to identify Boyce–Codd Normal Form (BCNF).
- How to represent attributes shown on a report as BCNF relations using normalization.
- The concept of multi-valued dependencies and 4NF.
- The problems associated with relations that break the rules of 4NF.
- How to create 4NF relations from a relation which breaks the rules of 4NF.
- The concept of join dependency and 5NF.
- The problems associated with relations that break the rules of 5NF.
- How to create 5NF relations from a relation which breaks the rules of 5NF.

In the previous chapter we introduced the technique of normalization and the concept of functional dependencies between attributes. We described the benefits of using normalization to support database design and demonstrated how attributes shown on sample forms are transformed into First Normal Form (1NF), Second Normal Form (2NF), and then finally Third Normal Form (3NF) relations. In this chapter, we return to consider functional dependencies and describe normal forms that go beyond 3NF such as Boyce–Codd Normal Form (BCNF), Fourth Normal Form (4NF), and Fifth Normal Form (5NF). Relations in 3NF are normally sufficiently well structured to prevent the problems associated with data redundancy, which was described in Section 13.3. However, later normal forms were created to identify relatively rare problems with relations that, if not corrected, may result in undesirable data redundancy.

Structure of this Chapter

With the exception of 1NF, all normal forms discussed in the previous chapter and in this chapter are based on functional dependencies among the attributes of a relation. In Section 14.1 we continue the discussion on the concept of functional dependency which was introduced in the previous chapter. We present a more formal and theoretical aspect of functional dependencies by discussing inference rules for functional dependencies.

In the previous chapter we described the three most commonly used normal forms: 1NF, 2NF, and 3NF. However, R. Boyce and E.F. Codd identified a weakness with 3NF and introduced a stronger definition of 3NF called Boyce–Codd Normal Form (BCNF) (Codd, 1974), which we describe in Section 14.2. In Section 14.3 we present a worked example to demonstrate the process of normalizing attributes originally shown on a report into a set of BCNF relations.

Higher normal forms that go beyond BCNF were introduced later, such as Fourth (4NF) and Fifth (5NF) Normal Forms (Fagin, 1977, 1979). However, these later normal forms deal with situations that are very rare. We describe 4NF and 5NF in Sections 14.4 and 14.5.

To illustrate the process of normalization examples are drawn from the *DreamHome* case study described in Section 10.4 and documented in Appendix A.

14.1 More on Functional Dependencies

One of the main concepts associated with normalization is **functional dependency**, which describes the relationship between attributes (Maier, 1983). In the previous chapter we introduced this concept. In this section we describe this concept in a more formal and theoretical way by discussing inference rules for functional dependencies.

14.1.1 Inference Rules for Functional Dependencies

In Section 13.4 we identified the characteristics of the functional dependencies that are most useful in normalization. However, even if we restrict our attention to functional dependencies with a one-to-one (1:1) relationship between attributes on the left- and right-hand sides of the dependency that hold for all time and are fully functionally dependent, then the complete set of functional dependencies for a given relation can still be very large. It is important to find an approach that can reduce that set to a manageable size. Ideally, we want to identify a set of functional dependencies (represented as X) for a relation that is smaller than the complete set of functional dependencies (represented as Y) for that relation and has the property that every functional dependency in Y is implied by the functional dependencies in X. Hence, if we enforce the integrity constraints defined by the functional dependencies in X, we automatically enforce the integrity constraints defined in the larger set of functional dependencies in Y. This requirement suggests that there must be functional dependencies that can be inferred from other functional dependencies. For example, functional dependencies A → B and B → C in a relation implies that the functional dependency A → C also holds in that relation. A → C is an example of a **transitive** functional dependency and was discussed previously in Sections 13.4 and 13.7.

How do we begin to identify useful functional dependencies on a relation? Normally, the database designer starts by specifying functional dependencies that are semantically obvious; however, there are usually numerous other functional dependencies. In fact, the task of specifying all possible functional dependencies for 'real' database projects is more often than not, impractical. However, in this section we do consider an approach that helps identify the complete set of functional dependencies for a relation and then discuss how to achieve a minimal set of functional dependencies that can represent the complete set.

The set of all functional dependencies that are implied by a given set of functional dependencies X is called the **closure** of X, written X^+. We clearly need a set of rules to help compute X^+ from X. A set of inference rules, called **Armstrong's axioms**, specifies how new functional dependencies can be inferred from given ones (Armstrong, 1974). For our discussion, let A, B, and C be subsets of the attributes of the relation R. Armstrong's axioms are as follows:

(1) **Reflexivity:** If B is a subset of A, then A → B
(2) **Augmentation:** If A → B, then A,C → B,C
(3) **Transitivity:** If A → B and B → C, then A → C

Note that each of these three rules can be directly proved from the definition of functional dependency. The rules are **complete** in that given a set X of functional dependencies, all functional dependencies implied by X can be derived from X using these rules. The rules are also **sound** in that no additional functional dependencies can be derived that are not implied by X. In other words, the rules can be used to derive the closure of X^+.

Several further rules can be derived from the three given above that simplify the practical task of computing X^+. In the following rules, let D be another subset of the attributes of relation R, then:

(4) **Self-determination:** A → A
(5) **Decomposition:** If A → B,C, then A → B and A → C
(6) **Union:** If A → B and A → C, then A → B,C
(7) **Composition:** If A → B and C → D then A,C → B,D

Rule 1 Reflexivity and Rule 4 Self-determination state that a set of attributes always determines any of its subsets or itself. Because these rules generate functional dependencies that are always true, such dependencies are trivial and, as stated earlier, are generally not interesting or useful. Rule 2 Augmentation states that adding the same set of attributes to both the left- and right-hand sides of a dependency results in another valid dependency. Rule 3 Transitivity states that functional dependencies are transitive. Rule 5 Decomposition states that we can remove attributes from the right-hand side of a dependency. Applying this rule repeatedly, we can decompose A → B, C, D functional dependency into the set of dependencies A → B, A → C, and A → D. Rule 6 Union states that we can do the opposite: we can combine a set of dependencies A → B, A → C, and A → D into a single functional dependency A → B, C, D. Rule 7 Composition is more general than Rule 6 and states that we can combine a set of non-overlapping dependencies to form another valid dependency.

To begin to identify the set of functional dependencies F for a relation, typically we first identify the dependencies that are determined from the semantics of the attributes of the relation. Then we apply Armstrong's axioms (Rules 1 to 3) to infer additional functional dependencies that are also true for that relation. A systematic way to determine these additional functional dependencies is to first determine each set of attributes A that appears on the left-hand side of some functional dependencies and then to determine the set of *all* attributes that are dependent on A. Thus, for each set of attributes A we can determine the set A^+ of attributes that are functionally determined by A based on F; (A^+ is called the **closure of A under F**).

14.1.2 Minimal Sets of Functional Dependencies

In this section, we introduce what is referred to as **equivalence** of sets of functional dependencies. A set of functional dependencies Y is **covered by** a set of functional dependencies X, if every functional dependency in Y is also in X^+; that is, every dependency in Y can be inferred from X. A set of functional dependencies X is minimal if it satisfies the following conditions:

- Every dependency in X has a single attribute on its right-hand side.
- We cannot replace any dependency A → B in X with dependency C → B, where C is a proper subset of A, and still have a set of dependencies that is equivalent to X.
- We cannot remove any dependency from X and still have a set of dependencies that is equivalent to X.

A minimal set of dependencies should be in a standard form with no redundancies. A minimal cover of a set of functional dependencies X is a minimal set of dependencies X_{min} that is equivalent to X. Unfortunately there can be several minimal covers for a set of functional dependencies. We demonstrate the identification of the minimal cover for the StaffBranch relation in the following example.

Example 14.1 Identifying the minimal set of functional dependencies of the StaffBranch relation

We apply the three conditions described above on the set of functional dependencies for the StaffBranch relation listed in Example 13.5 to produce the following functional dependencies:

staffNo → sName
staffNo → position
staffNo → salary
staffNo → branchNo
staffNo → bAddress

branchNo → bAddress
bAddress → branchNo
branchNo, position → salary
bAddress, position → salary

These functional dependencies satisfy the three conditions for producing a minimal set of functional dependencies for the StaffBranch relation. Condition 1 ensures that every dependency is in a standard form with a single attribute on the right-hand side. Conditions 2 and 3 ensure that there are no redundancies in the dependencies either by having redundant attributes on the left-hand side of a dependency (Condition 2) or by having a dependency that can be inferred from the remaining functional dependencies in X (Condition 3).

In the following section we return to consider normalization. We begin by discussing Boyce–Codd Normal Form (BCNF), a stronger normal form than 3NF.

14.2 Boyce–Codd Normal Form (BCNF)

In the previous chapter we demonstrated how 2NF and 3NF disallow partial and transitive dependencies on the *primary key* of a relation, respectively. Relations that have these types of dependencies may suffer from the update anomalies discussed in Section 13.3. However, the definition of 2NF and 3NF discussed in Sections 13.7 and 13.8, respectively, do not consider whether such dependencies remain on other candidate keys of a relation, if any exist. In Section 13.9 we presented general definitions for 2NF and 3NF that disallow partial and transitive dependencies on any *candidate key* of a relation, respectively. Application of the general definitions of 2NF and 3NF may identify additional redundancy caused by dependencies that violate one or more candidate keys. However, despite these additional constraints, dependencies can still exist that will cause redundancy to be present in 3NF relations. This weakness in 3NF, resulted in the presentation of a stronger normal form called Boyce–Codd Normal Form (Codd, 1974).

14.2.1 Definition of Boyce–Codd Normal Form

Boyce–Codd Normal Form (BCNF) is based on functional dependencies that take into account all candidate keys in a relation; however, BCNF also has additional constraints compared with the general definition of 3NF given in Section 13.9.

Boyce–Codd Normal Form (BCNF)	A relation is in BCNF, if and only if, every determinant is a candidate key.

To test whether a relation is in BCNF, we identify all the determinants and make sure that they are candidate keys. Recall that a determinant is an attribute, or a group of attributes, on which some other attribute is fully functionally dependent.

The difference between 3NF and BCNF is that for a functional dependency A → B, 3NF allows this dependency in a relation if B is a primary-key attribute and A is not a candidate key, whereas BCNF insists that for this dependency to remain in a relation, A must be a candidate key. Therefore, Boyce–Codd Normal Form is a stronger form of 3NF, such that every relation in BCNF is also in 3NF. However, a relation in 3NF is not necessarily in BCNF.

Before considering the next example, we re-examine the Client, Rental, PropertyForRent, and Owner relations shown in Figure 13.17. The Client, PropertyForRent, and Owner relations are all in BCNF, as each relation only has a single determinant, which is the candidate key. However, recall that the Rental relation contains the three determinants (clientNo, propertyNo), (clientNo, rentStart), and (propertyNo, rentStart), originally identified in Example 13.11, as shown below:

fd1	clientNo, propertyNo → rentStart, rentFinish
fd5′	clientNo, rentStart → propertyNo, rentFinish
fd6′	propertyNo, rentStart → clientNo, rentFinish

As the three determinants of the Rental relation are also candidate keys, the Rental relation is also already in BCNF. Violation of BCNF is quite rare, since it may only happen under specific conditions. The potential to violate BCNF may occur when:

- the relation contains two (or more) composite candidate keys; or
- the candidate keys overlap, that is have at least one attribute in common.

In the following example, we present a situation where a relation violates BCNF and demonstrate the transformation of this relation to BCNF. This example demonstrates the process of converting a 1NF relation to BCNF relations.

Example 14.2 Boyce–Codd Normal Form (BCNF)

In this example, we extend the *DreamHome* case study to include a description of client interviews by members of staff. The information relating to these interviews is in the ClientInterview relation shown in Figure 14.1. The members of staff involved in interviewing clients are allocated to a specific room on the day of interview. However, a room may be allocated to several members of staff as required throughout a working day. A client is only interviewed once on a given date, but may be requested to attend further interviews at later dates.

The ClientInterview relation has three candidate keys: (clientNo, interviewDate), (staffNo, interviewDate, interviewTime), and (roomNo, interviewDate, interviewTime). Therefore the ClientInterview relation has three composite candidate keys, which overlap by sharing the

Figure 14.1 ClientInterview relation.

ClientInterview

clientNo	interviewDate	interviewTime	staffNo	roomNo
CR76	13-May-05	10.30	SG5	G101
CR56	13-May-05	12.00	SG5	G101
CR74	13-May-05	12.00	SG37	G102
CR56	1-Jul-05	10.30	SG5	G102

common attribute interviewDate. We select (clientNo, interviewDate) to act as the primary key for this relation. The ClientInterview relation has the following form:

ClientInterview (clientNo, interviewDate, interviewTime, staffNo, roomNo)

The ClientInterview relation has the following functional dependencies:

fd1	clientNo, interviewDate → interviewTime, staffNo, roomNo	(Primary key)
fd2	staffNo, interviewDate, interviewTime → clientNo	(Candidate key)
fd3	roomNo, interviewDate, interviewTime → staffNo, clientNo	(Candidate key)
fd4	staffNo, interviewDate → roomNo	

We examine the functional dependencies to determine the normal form of the ClientInterview relation. As functional dependencies fd1, fd2, and fd3 are all candidate keys for this relation, none of these dependencies will cause problems for the relation. The only functional dependency that requires discussion is (staffNo, interviewDate) → roomNo (represented as fd4). Even though (staffNo, interviewDate) is not a candidate key for the ClientInterview relation this functional dependency is allowed in 3NF because roomNo is a primary-key attribute being part of the candidate key (roomNo, interviewDate, interviewTime). As there are no partial or transitive dependencies on the primary key (clientNo, interviewDate), and functional dependency fd4 is allowed, the ClientInterview relation is in 3NF.

However, this relation is not in BCNF (a stronger normal form of 3NF) due to the presence of the (staffNo, interviewDate) determinant, which is not a candidate key for the relation. BCNF requires that all determinants in a relation must be a candidate key for the relation. As a consequence the ClientInterview relation may suffer from update anomalies. For example, to change the room number for staff number SG5 on the 13-May-05 we must update two tuples. If only one tuple is updated with the new room number, this results in an inconsistent state for the database.

To transform the ClientInterview relation to BCNF, we must remove the violating functional dependency by creating two new relations called Interview and StaffRoom, as shown in Figure 14.2. The Interview and StaffRoom relations have the following form:

Interview (clientNo, interviewDate, interviewTime, staffNo)
StaffRoom (staffNo, interviewDate, roomNo)

Interview

clientNo	interviewDate	interviewTime	staffNo
CR76	13-May-05	10.30	SG5
CR56	13-May-05	12.00	SG5
CR74	13-May-05	12.00	SG37
CR56	1-Jul-05	10.30	SG5

StaffRoom

staffNo	interviewDate	roomNo
SG5	13-May-05	G101
SG37	13-May-05	G102
SG5	1-Jul-05	G102

Figure 14.2 The Interview and StaffRoom BCNF relations.

We can decompose any relation that is not in BCNF into BCNF as illustrated. However, it may not always be desirable to transform a relation into BCNF; for example, if there is a functional dependency that is not preserved when we perform the decomposition (that is, the determinant and the attributes it determines are placed in different relations). In this situation, it is difficult to enforce the functional dependency in the relation, and an important constraint is lost. When this occurs, it may be better to stop at 3NF, which always preserves dependencies. Note in Example 14.2, in creating the two BCNF relations from the original ClientInterview relation, we have 'lost' the functional dependency, roomNo, interviewDate, interviewTime → staffNo, clientNo (represented as fd3), as the determinant for this dependency is no longer in the same relation. However, we must recognize that if the functional dependency, staffNo, interviewDate → roomNo (represented as fd4) is not removed, the ClientInterview relation will have data redundancy.

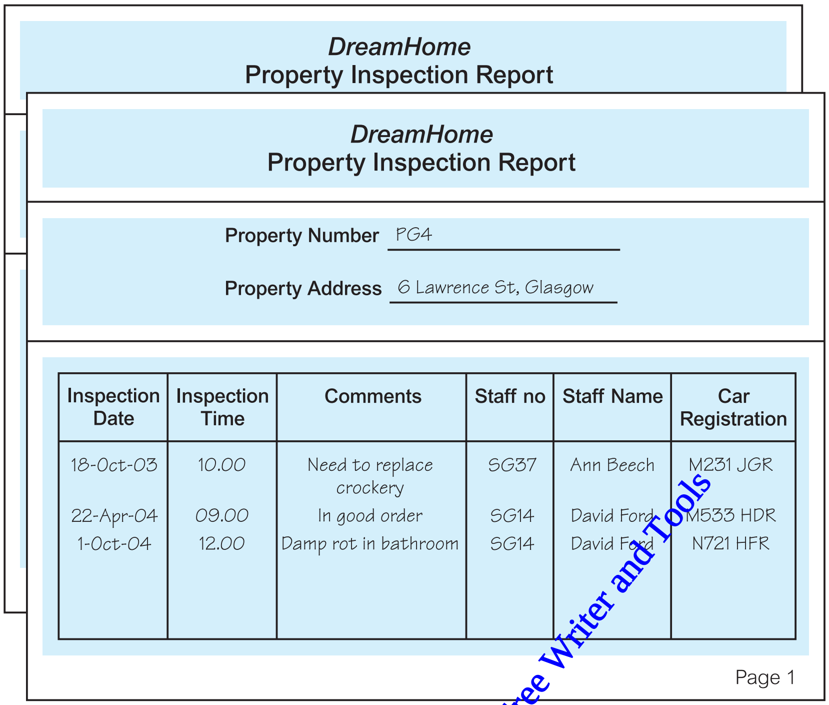
The decision as to whether it is better to stop the normalization at 3NF or progress to BCNF is dependent on the amount of redundancy resulting from the presence of fd4 and the significance of the 'loss' of fd3. For example, if it is the case that members of staff conduct only one interview per day, then the presence of fd4 in the ClientInterview relation will not cause redundancy and therefore the decomposition of this relation into two BCNF relations is not helpful or necessary. On the other hand, if members of staff conduct numerous interviews per day, then the presence of fd4 in the ClientInterview relation will cause redundancy and normalization of this relation to BCNF is recommended. However, we should also consider the significance of losing fd3; in other words, does fd3 convey important information about client interviews that must be represented in one of the resulting relations? The answer to this question will help to determine whether it is better to retain all functional dependencies or remove data redundancy.

14.3 Review of Normalization up to BCNF

The purpose of this section is to review the process of normalization described in the previous chapter and in Section 14.2. We demonstrate the process of transforming attributes displayed on a sample report from the *DreamHome* case study into a set of Boyce–Codd Normal Form relations. In this worked example we use the definitions of 2NF and 3NF that are based on the primary key of a relation. We leave the normalization of this worked example using the general definitions of 2NF and 3NF as an exercise for the reader.

Example 14.3 First normal form (1NF) to Boyce–Codd Normal Form (BCNF)

In this example we extend the *DreamHome* case study to include property inspection by members of staff. When staff are required to undertake these inspections, they are allocated a company car for use on the day of the inspections. However, a car may be allocated to several members of staff as required throughout the working day. A member of staff may inspect several properties on a given date, but a property is only inspected once on a given date. Examples of the *DreamHome* Property Inspection Report are



DreamHome
Property Inspection Report

DreamHome
Property Inspection Report

Property Number: PG4

Property Address: 6 Lawrence St, Glasgow

Inspection Date	Inspection Time	Comments	Staff no	Staff Name	Car Registration
18-Oct-03	10.00	Need to replace crockery	SG37	Ann Beech	M231 JGR
22-Apr-04	09.00	In good order	SG14	David Ford	M533 HDR
1-Oct-04	12.00	Damp rot in bathroom	SG14	David Ford	N721 HFR

Page 1

Figure 14.3 *DreamHome* Property Inspection reports.

StaffPropertyInspection

propertyNo	pAddress	iDate	iTime	comments	staffNo	sName	carReg
PG4	6 Lawrence St, Glasgow	18-Oct-03 22-Apr-04 1-Oct-04	10.00 09.00 12.00	Need to replace crockery In good order Damp rot in bathroom	SG37 SG14 SG14	Ann Beech David Ford David Ford	M231 JGR M533 HDR N721 HFR
PG16	5 Novar Dr, Glasgow	22-Apr-04 24-Oct-04	13.00 14.00	Replace living room carpet Good condition	SG14 SG37	David Ford Ann Beech	M533 HDR N721 HFR

Figure 14.4 StaffPropertyInspection unnormalized table.

presented in Figure 14.3. The report on top describes staff inspections of property PG4 in Glasgow.

First Normal Form (1NF)

We first transfer sample data held on two property inspection reports into table format with rows and columns. This is referred to as the StaffPropertyInspection unnormalized table and is shown in Figure 14.4. We identify the key attribute for this unnormalized table as propertyNo.

We identify the repeating group in the unnormalized table as the property inspection and staff details, which repeats for each property. The structure of the repeating group is:

Repeating Group = (iDate, iTime, comments, staffNo, sName, carReg)

Figure 14.5 The First Normal Form (1NF) StaffPropertyInspection relation.

StaffPropertyInspection

propertyNo	iDate	iTime	pAddress	comments	staffNo	sName	carReg
PG4	18-Oct-03	10.00	6 Lawrence St, Glasgow	Need to replace crockery	SG37	Ann Beech	M231 JGR
PG4	22-Apr-04	09.00	6 Lawrence St, Glasgow	In good order	SG14	David Ford	M533 HDR
PG4	1-Oct-04	12.00	6 Lawrence St, Glasgow	Damp rot in bathroom	SG14	David Ford	N721 HFR
PG16	22-Apr-04	13.00	5 Novar Dr, Glasgow	Replace living room carpet	SG14	David Ford	M533 HDR
PG16	24-Oct-04	14.00	5 Novar Dr, Glasgow	Good condition	SG37	Ann Beech	N721 HFR

As a consequence, there are multiple values at the intersection of certain rows and columns. For example, for propertyNo PG4 there are three values for iDate (18-Oct-03, 22-Apr-04, 1-Oct-04). We transform the unnormalized form to first normal form using the first approach described in Section 13.6. With this approach, we remove the repeating group (property inspection and staff details) by entering the appropriate property details (nonrepeating data) into each row. The resulting first normal form StaffPropertyInspection relation is shown in Figure 14.5.

In Figure 14.6, we present the functional dependencies (fd1 to fd6) for the StaffPropertyInspection relation. We use the functional dependencies (as discussed in Section 13.4.3) to identify candidate keys for the StaffPropertyInspection relation as being composite keys comprising (propertyNo, iDate), (staffNo, iDate, iTime), and (carReg, iDate, iTime). We select (propertyNo, iDate) as the primary key for this relation. For clarity, we place the attributes that make up the primary key together, at the left-hand side of the relation. The StaffPropertyInspection relation is defined as follows:

StaffPropertyInspection (propertyNo, iDate, iTime, pAddress, comments, staffNo, sName, carReg)

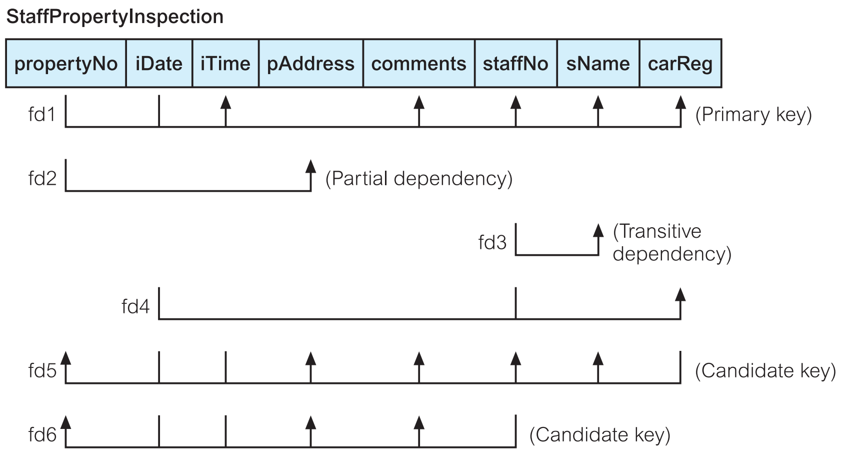


Figure 14.6 Functional dependencies of the StaffPropertyInspection relation.

The StaffPropertyInspection relation is in first normal form (1NF) as there is a single value at the intersection of each row and column. The relation contains data describing the inspection of property by members of staff, with the property and staff details repeated several times. As a result, the StaffPropertyInspection relation contains significant redundancy. If implemented, this 1NF relation would be subject to update anomalies. To remove some of these, we must transform the relation into second normal form.

Second Normal Form (2NF)

The normalization of 1NF relations to 2NF involves the removal of partial dependencies on the primary key. If a partial dependency exists, we remove the functionally dependent attributes from the relation by placing them in a new relation with a copy of their determinant.

As shown in Figure 14.6, the functional dependencies (fd1 to fd6) of the StaffPropertyInspection relation are as follows:

fd1	propertyNo, iDate → iTime, comments, staffNo, sName, carReg	(Primary key)
fd2	propertyNo → pAddress	(Partial dependency)
fd3	staffNo → sName	(Transitive dependency)
fd4	staffNo, iDate → carReg	
fd5	carReg, iDate, iTime → propertyNo, pAddress, comments, staffNo, sName	(Candidate key)
fd6	staffNo, iDate, iTime → propertyNo, pAddress, comments	(Candidate key)

Using the functional dependencies, we continue the process of normalizing the StaffPropertyInspection relation. We begin by testing whether the relation is in 2NF by identifying the presence of any partial dependencies on the primary key. We note that the property attribute (pAddress) is partially dependent on part of the primary key, namely the propertyNo (represented as fd2), whereas the remaining attributes (iTime, comments, staffNo, sName, and carReg) are fully dependent on the whole primary key (propertyNo and iDate), (represented as fd1). Note that although the determinant of the functional dependency staffNo, iDate → carReg (represented as fd4) only requires the iDate attribute of the primary key, we do not remove this dependency at this stage as the determinant also includes another non-primary-key attribute, namely staffNo. In other words, this dependency is *not* wholly dependent on part of the primary key and therefore does not violate 2NF.

The identification of the partial dependency (propertyNo → pAddress) indicates that the StaffPropertyInspection relation is not in 2NF. To transform the relation into 2NF requires the creation of new relations so that the attributes that are not fully dependent on the primary key are associated with only the appropriate part of the key.

The StaffPropertyInspection relation is transformed into second normal form by removing the partial dependency from the relation and creating two new relations called Property and PropertyInspection with the following form:

Property	(propertyNo, pAddress)
PropertyInspection	(propertyNo, iDate, iTime, comments, staffNo, sName, carReg)

These relations are in 2NF, as every non-primary-key attribute is functionally dependent on the primary key of the relation.

Third Normal Form (3NF)

The normalization of 2NF relations to 3NF involves the removal of transitive dependencies. If a transitive dependency exists, we remove the transitively dependent attributes from the relation by placing them in a new relation along with a copy of their determinant. The functional dependencies within the Property and PropertyInspection relations are as follows:

Property Relation

fd2	propertyNo → pAddress

PropertyInspection Relation

fd1	propertyNo, iDate → iTime, comments, staffNo, sName, carReg
fd3	staffNo → sName
fd4	staffNo, iDate → carReg
fd5′	carReg, iDate, iTime → propertyNo, comments, staffNo, sName
fd6′	staffNo, iDate, iTime → propertyNo, comments

As the Property relation does not have transitive dependencies on the primary key, it is therefore already in 3NF. However, although all the non-primary-key attributes within the PropertyInspection relation are functionally dependent on the primary key, sName is also transitively dependent on staffNo (represented as fd3). We also note the functional dependency staffNo, iDate → carReg (represented as fd4) has a non-primary-key attribute carReg partially dependent on a non-primary-key attribute, staffNo. We do not remove this dependency at this stage as part of the determinant for this dependency includes a primary-key attribute, namely iDate. In other words, this dependency is *not* wholly transitively dependent on non-primary-key attributes and therefore does not violate 3NF. (In other words, as described in Section 13.9, when considering all candidate keys of a relation, the staffNo, iDate → carReg dependency is allowed in 3NF because carReg is a primary-key attribute as it is part of the candidate key (carReg, iDate, iTime) of the original PropertyInspection relation.)

To transform the PropertyInspection relation into 3NF, we remove the transitive dependency (staffNo → sName) by creating two new relations called Staff and PropertyInspect with the form:

Staff	(<u>staffNo</u>, sName)
PropertyInspect	(<u>propertyNo</u>, <u>iDate</u>, iTime, comments, staffNo, carReg)

The Staff and PropertyInspect relations are in 3NF as no non-primary-key attribute is wholly functionally dependent on another non-primary-key attribute. Thus, the StaffPropertyInspection relation shown in Figure 14.5 has been transformed by the process of normalization into three relations in 3NF with the following form:

Property	(<u>propertyNo</u>, pAddress)
Staff	(<u>staffNo</u>, sName)
PropertyInspect	(<u>propertyNo</u>, <u>iDate</u>, iTime, comments, staffNo, carReg)

Boyce–Codd Normal Form (BCNF)

We now examine the Property, Staff, and PropertyInspect relations to determine whether they are in BCNF. Recall that a relation is in BCNF if every determinant of a relation is a candidate key. Therefore, to test for BCNF, we simply identify all the determinants and make sure they are candidate keys.

The functional dependencies for the Property, Staff, and PropertyInspect relations are as follows:

<u>Property Relation</u>

fd2	propertyNo → pAddress

<u>Staff Relation</u>

fd3	staffNo → sName

<u>PropertyInspect Relation</u>

fd1′	propertyNo, iDate → iTime, comments, staffNo, carReg
fd4	staffNo, iDate → carReg
fd5′	carReg, iDate, iTime → propertyNo, comments, staffNo
fd6′	staffNo, iDate, iTime → propertyNo, comments

We can see that the Property and Staff relations are already in BCNF as the determinant in each of these relations is also the candidate key. The only 3NF relation that is not in BCNF is PropertyInspect because of the presence of the determinant (staffNo, iDate), which is not a candidate key (represented as fd4). As a consequence the PropertyInspect relation may suffer from update anomalies. For example, to change the car allocated to staff number SG14 on the 22-Apr-03, we must update two tuples. If only one tuple is updated with the new car registration number, this results in an inconsistent state for the database.

To transform the PropertyInspect relation into BCNF, we must remove the dependency that violates BCNF by creating two new relations called StaffCar and Inspection with the form:

StaffCar	(<u>staffNo</u>, <u>iDate</u>, carReg)
Inspection	(<u>propertyNo</u>, <u>iDate</u>, iTime, comments, staffNo)

The StaffCar and Inspection relations are in BCNF as the determinant in each of these relations is also a candidate key.

In summary, the decomposition of the StaffPropertyInspection relation shown in Figure 14.5 into BCNF relations is shown in Figure 14.7. In this example, the decomposition of the

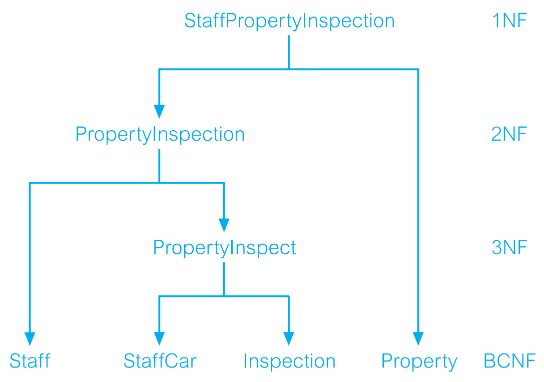


Figure 14.7 Decomposition of the StaffPropertyInspection relation into BCNF relations.

original StaffPropertyInspection relation to BCNF relations has resulted in the 'loss' of the functional dependency: carReg, iDate, iTime → propertyNo, pAddress, comments, staffNo, sName, as parts of the determinant are in different relations (represented as fd5). However, we recognize that if the functional dependency, staffNo, iDate → carReg (represented as fd4) is not removed, the PropertyInspect relation will have data redundancy.

The resulting BCNF relations have the following form:

Property (propertyNo, pAddress)
Staff (staffNo, sName)
Inspection (propertyNo, iDate, iTime, comments, staffNo)
StaffCar (staffNo, iDate, carReg)

The original StaffPropertyInspection relation shown in Figure 14.5 can be recreated from the Property, Staff, Inspection, and StaffCar relations using the primary key/foreign key mechanism. For example, the attribute staffNo is a primary key within the Staff relation and is also present within the Inspection relation as a foreign key. The foreign key allows the association of the Staff and Inspection relations to identify the name of the member of staff undertaking the property inspection.

14.4 Fourth Normal Form (4NF)

Although BCNF removes any anomalies due to functional dependencies, further research led to the identification of another type of dependency called a **Multi-Valued Dependency** (MVD), which can also cause data redundancy (Fagin, 1977). In this section, we briefly describe a multi-valued dependency and the association of this type of dependency with Fourth Normal Form (4NF).

14.4.1 Multi-Valued Dependency

The possible existence of multi-valued dependencies in a relation is due to First Normal Form, which disallows an attribute in a tuple from having a set of values. For example, if we have two multi-valued attributes in a relation, we have to repeat each value of one of the attributes with every value of the other attribute, to ensure that tuples of the relation are consistent. This type of constraint is referred to as a multi-valued dependency and results in data redundancy. Consider the BranchStaffOwner relation shown in Figure 14.8(a), which

Figure 14.8(a) The BranchStaffOwner relation.

BranchStaffOwner

branchNo	sName	oName
B003	Ann Beech	Carol Farrel
B003	David Ford	Carol Farrel
B003	Ann Beech	Tina Murphy
B003	David Ford	Tina Murphy

displays the names of members of staff (sName) and property owners (oName) at each branch office (branchNo). In this example, assume that staff name (sName) uniquely identifies each member of staff and that the owner name (oName) uniquely identifies each owner.

In this example, members of staff called Ann Beech and David Ford work at branch B003, and property owners called Carol Farrel and Tina Murphy are registered at branch B003. However, as there is no direct relationship between members of staff and property owners at a given branch office, we must create a tuple for every combination of member of staff and owner to ensure that the relation is consistent. This constraint represents a multi-valued dependency in the BranchStaffOwner relation. In other words, a MVD exists because two independent 1:* relationships are represented in the BranchStaffOwner relation.

Multi-Valued Dependency (MVD) Represents a dependency between attributes (for example, A, B, and C) in a relation, such that for each value of A there is a set of values for B and a set of values for C. However, the set of values for B and C are independent of each other.

We represent a MVD between attributes A, B, and C in a relation using the following notation:

$$A \twoheadrightarrow B$$
$$A \twoheadrightarrow C$$

For example, we specify the MVD in the BranchStaffOwner relation shown in Figure 14.8(a) as follows:

$$\text{branchNo} \twoheadrightarrow \text{sName}$$
$$\text{branchNo} \twoheadrightarrow \text{oName}$$

A multi-valued dependency can be further defined as being **trivial** or **nontrivial**. A MVD $A \twoheadrightarrow B$ in relation R is defined as being trivial if (a) B is a subset of A *or* (b) $A \cup B = R$. A MVD is defined as being nontrivial if neither (a) nor (b) is satisfied. A trivial MVD does not specify a constraint on a relation, while a nontrivial MVD does specify a constraint.

The MVD in the BranchStaffOwner relation shown in Figure 14.8(a) is nontrivial as neither condition (a) nor (b) is true for this relation. The BranchStaffOwner relation is therefore constrained by the nontrivial MVD to repeat tuples to ensure the relation remains consistent in terms of the relationship between the sName and oName attributes. For example, if we wanted to add a new property owner for branch B003 we would have to create two new tuples, one for each member of staff, to ensure that the relation remains consistent. This is an example of an update anomaly caused by the presence of the nontrivial MVD.

Even though the BranchStaffOwner relation is in BCNF, the relation remains poorly structured, due to the data redundancy caused by the presence of the nontrivial MVD. We clearly require a stronger form of BCNF that prevents relational structures such as the BranchStaffOwner relation.

Figure 14.8(b) The BranchStaff and BranchOwner 4NF relations.

BranchStaff

branchNo	sName
B003	Ann Beech
B003	David Ford

BranchOwner

branchNo	oName
B003	Carol Farrel
B003	Tina Murphy

14.4.2 Definition of Fourth Normal Form

Fourth Normal Form (4NF) A relation that is in Boyce–Codd normal form and does not contain nontrivial multi-valued dependencies.

Fourth Normal Form (4NF) is a stronger normal form than BCNF as it prevents relations from containing nontrivial MVDs, and hence data redundancy (Fagin, 1977). The normalization of BCNF relations to 4NF involves the removal of the MVD from the relation by placing the attribute(s) in a new relation along with a copy of the determinant(s).

For example, the BranchStaffOwner relation in Figure 14.8(a) is not in 4NF because of the presence of the nontrivial MVD. We decompose the BranchStaffOwner relation into the BranchStaff and BranchOwner relations, as shown in Figure 14.8(b). Both new relations are in 4NF because the BranchStaff relation contains the trivial MVD branchNo $\twoheadrightarrow$ sName, and the BranchOwner relation contains the trivial MVD branchNo $\twoheadrightarrow$ oName. Note that the 4NF relations do not display data redundancy and the potential for update anomalies is removed. For example, to add a new property owner for branch B003, we simply create a single tuple in the BranchOwner relation.

For a detailed discussion on 4NF the interested reader is referred to Date (2003), Elmasri and Navathe (2003), and Hawryszkiewycz (1994).

14.5 Fifth Normal Form (5NF)

Whenever we decompose a relation into two relations the resulting relations have the lossless-join property. This property refers to the fact that we can rejoin the resulting relations to produce the original relation. However, there are cases were there is the requirement to decompose a relation into more than two relations. Although rare, these cases are managed by join dependency and Fifth Normal Form (5NF). In this section we briefly describe the lossless-join dependency and the association with 5NF.

14.5.1 Lossless-Join Dependency

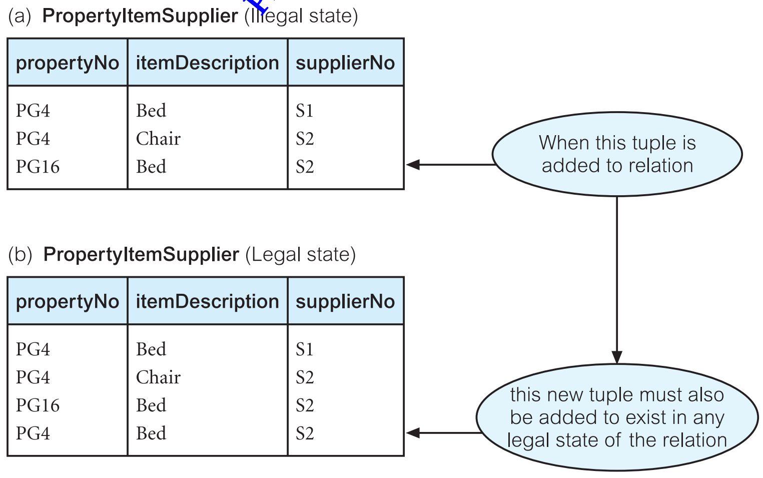
Lossless-join dependency A property of decomposition, which ensures that no spurious tuples are generated when relations are reunited through a natural join operation.

In splitting relations by projection, we are very explicit about the method of decomposition. In particular, we are careful to use projections that can be reversed by joining the resulting relations, so that the original relation is reconstructed. Such a decomposition is called a **lossless-join** (also called a *nonloss-* or *nonadditive*-join) decomposition, because it preserves all the data in the original relation and does not result in the creation of additional spurious tuples. For example, Figures 14.8(a) and (b) show that the decomposition of the BranchStaffOwner relation into the BranchStaff and BranchOwner relations has the lossless-join property. In other words, the original BranchStaffOwner relation can be reconstructed by performing a natural join operation on the BranchStaff and BranchOwner relations. In this example, the original relation is decomposed into two relations. However, there are cases were we require to perform a lossless-join decompose of a relation into more than two relations (Aho *et al.*, 1979). These cases are the focus of the lossless-join dependency and Fifth Normal Form (5NF).

14.5.2 Definition of Fifth Normal Form

Fifth Normal Form (5NF) A relation that has no join dependency.

Fifth Normal Form (5NF) (also called *Project-Join Normal Form* (PJNF)) specifies that a 5NF relation has no join dependency (Fagin, 1979). To examine what a join dependency means, consider as an example the PropertyItemSupplier relation shown in Figure 14.9(a). This relation describes properties (propertyNo) that require certain items (itemDescription), which are supplied by suppliers (supplierNo) to the properties (propertyNo). Furthermore, whenever a property (p) requires a certain item (i) and a supplier (s) supplies that item (i) and the supplier (s) already supplies *at least one* item to that property (p), then the supplier (s) will also supply the required item (i) to property (p). In this example, assume that a description of an item (itemDescription) uniquely identifies each type of item.


(a) **PropertyItemSupplier** (Illegal state)

propertyNo	itemDescription	supplierNo
PG4	Bed	S1
PG4	Chair	S2
PG16	Bed	S2

(b) **PropertyItemSupplier** (Legal state)

propertyNo	itemDescription	supplierNo
PG4	Bed	S1
PG4	Chair	S2
PG16	Bed	S2
PG4	Bed	S2

When this tuple is added to relation
this new tuple must also be added to exist in any legal state of the relation

Figure 14.9
(a) Illegal state for PropertyItemSupplier relation and
(b) legal state for PropertyItemSupplier relation.

To identify the type of constraint on the PropertyItemSupplier relation in Figure 14.9(a), consider the following statement:

If	Property **PG4** requires **Bed**	(from data in tuple 1)
	Supplier **S2** supplies property **PG4**	(from data in tuple 2)
	Supplier **S2** provides **Bed**	(from data in tuple 3)
Then	Supplier **S2** provides **Bed** for property **PG4**	

This example illustrates the cyclical nature of the constraint on the PropertyItemSupplier relation. If this constraint holds then the tuple (PG4, Bed, S2) must exist in any legal state of the PropertyItemSupplier relation as shown in Figure 14.9(b). This is an example of a type of update anomaly and we say that this relation contains a join dependency (JD).

Join dependency Describes a type of dependency. For example, for a relation R with subsets of the attributes of R denoted as A, B, . . . , Z, a relation R satisfies a join dependency if and only if every legal value of R is equal to the join of its projections on A, B, . . . , Z.

As the PropertyItemSupplier relation contains a join dependency, it is therefore not in 5NF. To remove the join dependency, we decompose the PropertyItemSupplier relation into three 5NF relations, namely PropertyItem (R1), ItemSupplier (R2), and PropertySupplier (R3) relations, as shown in Figure 14.10. We say that the PropertyItemSupplier relation with the form (A, B, C) satisfies the join dependency JD (R1(A, B), R2(B, C), R3(A, C)).

It is important to note that performing a natural join on any two relations will produce spurious tuples; however, performing the join on all three will recreate the original PropertyItemSupplier relation.

For a detailed discussion on 5NF the interested reader is referred to Date (2003), Elmasri and Navathe (2003), and Hawryszkiewycz (1994).

Figure 14.10 PropertyItem, ItemSupplier, and PropertySupplier 5NF relations.

PropertyItem

propertyNo	itemDescription
PG4	Bed
PG4	Chair
PG16	Bed

ItemSupplier

itemDescription	supplierNo
Bed	S1
Chair	S2
Bed	S2

PropertySupplier

propertyNo	supplierNo
PG4	S1
PG4	S2
PG16	S2

Chapter Summary

- **Inference rules** can be used to identify the set of *all* functional dependencies associated with a relation. This set of dependencies can be very large for a given relation.
- Inference rules called **Armstrong's axioms** can be used to identify a *minimal* set of functional dependencies from the set of all functional dependencies for a relation.
- **Boyce–Codd Normal Form** (**BCNF**) is a relation in which every determinant is a candidate key.
- **Fourth Normal Form** (**4NF**) is a relation that is in BCNF and does not contain nontrivial multi-valued dependencies. A **multi-valued dependency** (**MVD**) represents a dependency between attributes (A, B, and C) in a relation, such that for each value of A there is a set of values of B and a set of values for C. However, the set of values for B and C are independent of each other.
- A **lossless-join dependency** is a property of decomposition, which means that no spurious tuples are generated when relations are combined through a natural join operation.
- **Fifth Normal Form** (**5NF**) is a relation that contains no join dependency. For a relation R with subsets of attributes of R denoted as A, B, . . . , Z, a relation R satisfies a **join dependency** if and only if every legal value of R is equal to the join of its projections on A, B, . . . , Z.

Review Questions

14.1 Describe the purpose of using inference rules to identify functional dependencies for a given relation.

14.2 Discuss the purpose of Armstrong's axioms.

14.3 Discuss the purpose of Boyce–Codd Normal Form (BCNF) and discuss how BCNF differs from 3NF. Provide an example to illustrate your answer.

14.4 Describe the concept of multi-valued dependency and discuss how this concept relates to 4NF. Provide an example to illustrate your answer.

14.5 Describe the concept of join dependency and discuss how this concept relates to 5NF. Provide an example to illustrate your answer.

Exercises

14.6 On completion of Exercise 13.14 examine the 3NF relations created to represent the attributes shown in the *Wellmeadows Hospital* form shown in Figure 13.18. Determine whether these relations are also in BCNF. If not, transform the relations that do not conform into BCNF.

14.7 On completion of Exercise 13.15 examine the 3NF relations created to represent the attributes shown in the relation that displays dentist/patient appointment data in Figure 13.19. Determine whether these relations are also in BCNF. If not, transform the relations that do not conform into BCNF.

14.8 On completion of Exercise 13.16 examine the 3NF relations created to represent the attributes shown in the relation displaying employee contract data for an agency called *Instant Cover* in Figure 13.20. Determine whether these relations are also in BCNF. If not, transform the relations that do not conform into BCNF.

14.9 The relation shown in Figure 14.11 lists members of staff (staffName) working in a given ward (wardName) and patients (patientName) allocated to a given ward. There is no relationship between members of staff and

Figure 14.11 The WardStaffPatient relation.

wardName	staffName	patientName
Pediatrics	Kim Jones	Claire Johnson
Pediatrics	Kim Jones	Brian White
Pediatrics	Stephen Ball	Claire Johnson
Pediatrics	Stephen Ball	Brian White

patients in each ward. In this example assume that staff name (staffName) uniquely identifies each member of staff and that the patient name (patientName) uniquely identifies each patient.

(a) Describe why the relation shown in Figure 14.11 is not in 4NF.

(b) The relation shown in Figure 14.11 is susceptible to update anomalies. Provide examples of insertion, deletion, and update anomalies.

(c) Describe and illustrate the process of normalizing the relation shown in Figure 14.11 to 4NF.

14.10 The relation shown in Figure 14.12 describes hospitals (hospitalName) that require certain items (itemDescription), which are supplied by suppliers (supplierNo) to the hospitals (hospitalName). Furthermore, whenever a hospital (h) requires a certain item (i) and a supplier (s) supplies that item (i) and the supplier (s) already supplies *at least one* item to that hospital (h), then the supplier (s) will also supply the required item (i) to the hospital (h). In this example, assume that a description of an item (itemDescription) uniquely identifies each type of item.

(a) Describe why the relation shown in Figure 14.12 is not in 5NF.

(b) Describe and illustrate the process of normalizing the relation shown in Figure 14.12 to 5NF.

Figure 14.12 The HospitalItemSupplier relation.

hospitalName	itemDescription	supplierNo
Western General	Antiseptic Wipes	S1
Western General	Paper Towels	S2
Yorkhill	Antiseptic Wipes	S2
Western General	Antiseptic Wipes	S2