Chapter 2 Database Environment

Chapter Objectives

In this chapter you will learn:

- The purpose and origin of the three-level database architecture.
- The contents of the external, conceptual, and internal levels.
- The purpose of the external/conceptual and the conceptual/internal mappings.
- The meaning of logical and physical data independence.
- The distinction between a Data Definition Language (DDL) and a Data Manipulation Language (DML).
- A classification of data models.
- The purpose and importance of conceptual modeling.
- The typical functions and services a DBMS should provide.
- The function and importance of the system catalog.
- The software components of a DBMS.
- The meaning of the client–server architecture and the advantages of this type of architecture for a DBMS.
- The function and uses of Transaction Processing (TP) Monitors.

A major aim of a database system is to provide users with an abstract view of data, hiding certain details of how data is stored and manipulated. Therefore, the starting point for the design of a database must be an abstract and general description of the information requirements of the organization that is to be represented in the database. In this chapter, and throughout this book, we use the term 'organization' loosely, to mean the whole organization or part of the organization. For example, in the *DreamHome* case study we may be interested in modeling:

- the 'real world' **entities** Staff, PropertyforRent, PrivateOwner, and Client;
- **attributes** describing properties or qualities of each entity (for example, Staff have a name, position, and salary);
- **relationships** between these entities (for example, Staff *Manages* PropertyForRent).

Furthermore, since a database is a shared resource, each user may require a different view of the data held in the database. To satisfy these needs, the architecture of most commercial DBMSs available today is based to some extent on the so-called ANSI-SPARC architecture. In this chapter, we discuss various architectural and functional characteristics of DBMSs.

Structure of this Chapter

In Section 2.1 we examine the three-level ANSI-SPARC architecture and its associated benefits. In Section 2.2 we consider the types of language that are used by DBMSs, and in Section 2.3 we introduce the concepts of data models and conceptual modeling, which we expand on in later parts of the book. In Section 2.4 we discuss the functions that we would expect a DBMS to provide, and in Sections 2.5 and 2.6 we examine the internal architecture of a typical DBMS. The examples in this chapter are drawn from the *DreamHome* case study, which we discuss more fully in Section 10.4 and Appendix A.

Much of the material in this chapter provides important background information on DBMSs. However, the reader who is new to the area of database systems may find some of the material difficult to appreciate on first reading. Do not be too concerned about this, but be prepared to revisit parts of this chapter at a later date when you have read subsequent chapters of the book.

2.1 The Three-Level ANSI-SPARC Architecture

An early proposal for a standard terminology and general architecture for database systems was produced in 1971 by the DBTG (Data Base Task Group) appointed by the Conference on Data Systems and Languages (CODASYL, 1971). The DBTG recognized the need for a two-level approach with a system view called the **schema** and user views called **subschemas**. The American National Standards Institute (ANSI) Standards Planning and Requirements Committee (SPARC), ANSI/X3/SPARC, produced a similar terminology and architecture in 1975 (ANSI, 1975). ANSI-SPARC recognized the need for a three-level approach with a system catalog. These proposals reflected those published by the IBM user organizations Guide and Share some years previously, and concentrated on the need for an implementation-independent layer to isolate programs from underlying representational issues (Guide/Share, 1970). Although the ANSI-SPARC model did not become a standard, it still provides a basis for understanding some of the functionality of a DBMS.

For our purposes, the fundamental point of these and later reports is the identification of three levels of abstraction, that is, three distinct levels at which data items can be described. The levels form a **three-level architecture** comprising an **external**, a **conceptual**, and an **internal** level, as depicted in Figure 2.1. The way users perceive the data is called the **external level**. The way the DBMS and the operating system perceive the data is the **internal level**, where the data is actually stored using the data structures and file

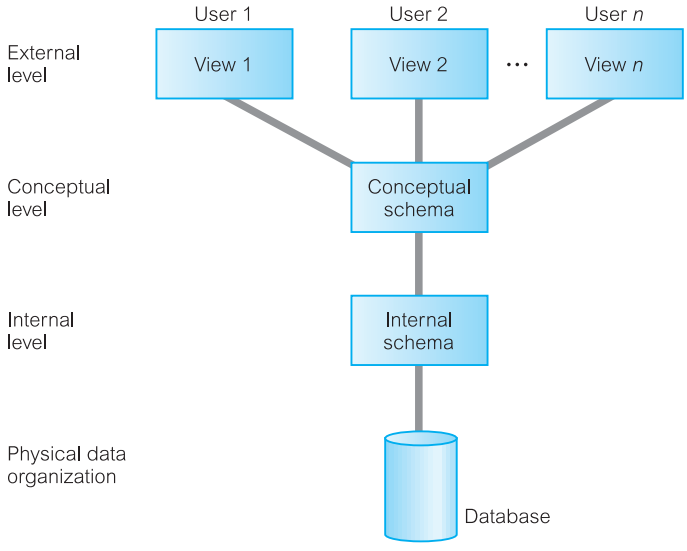

Figure 2.1
The ANSI-SPARC three-level architecture.

organizations described in Appendix C. The **conceptual level** provides both the **mapping** and the desired **independence** between the external and internal levels.

The objective of the three-level architecture is to separate each user's view of the database from the way the database is physically represented. There are several reasons why this separation is desirable:

- Each user should be able to access the same data, but have a different customized view of the data. Each user should be able to change the way he or she views the data, and this change should not affect other users.
- Users should not have to deal directly with physical database storage details, such as indexing or hashing (see Appendix C). In other words, a user's interaction with the database should be independent of storage considerations.
- The Database Administrator (DBA) should be able to change the database storage structures without affecting the users' views.
- The internal structure of the database should be unaffected by changes to the physical aspects of storage, such as the changeover to a new storage device.
- The DBA should be able to change the conceptual structure of the database without affecting all users.

External Level 2.1.1

External level	The users' view of the database. This level describes that part of the database that is relevant to each user.

The external level consists of a number of different external views of the database. Each user has a view of the 'real world' represented in a form that is familiar for that user. The external view includes only those entities, attributes, and relationships in the 'real world' that the user is interested in. Other entities, attributes, or relationships that are not of interest may be represented in the database, but the user will be unaware of them.

In addition, different views may have different representations of the same data. For example, one user may view dates in the form (day, month, year), while another may view dates as (year, month, day). Some views might include derived or calculated data: data not actually stored in the database as such, but created when needed. For example, in the *DreamHome* case study, we may wish to view the age of a member of staff. However, it is unlikely that ages would be stored, as this data would have to be updated daily. Instead, the member of staff's date of birth would be stored and age would be calculated by the DBMS when it is referenced. Views may even include data combined or derived from several entities. We discuss views in more detail in Sections 3.4 and 6.4.

2.1.2 Conceptual Level

Conceptual level The community view of the database. This level describes *what* data is stored in the database and the relationships among the data.

The middle level in the three-level architecture is the conceptual level. This level contains the logical structure of the entire database as seen by the DBA. It is a complete view of the data requirements of the organization that is independent of any storage considerations. The conceptual level represents:

- all entities, their attributes, and their relationships;
- the constraints on the data;
- semantic information about the data;
- security and integrity information.

The conceptual level supports each external view, in that any data available to a user must be contained in, or derivable from, the conceptual level. However, this level must not contain any storage-dependent details. For instance, the description of an entity should contain only data types of attributes (for example, integer, real, character) and their length (such as the maximum number of digits or characters), but not any storage considerations, such as the number of bytes occupied.

2.1.3 Internal Level

Internal level The physical representation of the database on the computer. This level describes *how* the data is stored in the database.

The internal level covers the physical implementation of the database to achieve optimal runtime performance and storage space utilization. It covers the data structures and file organizations used to store data on storage devices. It interfaces with the operating system access methods (file management techniques for storing and retrieving data records) to place the data on the storage devices, build the indexes, retrieve the data, and so on. The internal level is concerned with such things as:

- storage space allocation for data and indexes;
- record descriptions for storage (with stored sizes for data items);
- record placement;
- data compression and data encryption techniques.

Below the internal level there is a **physical level** that may be managed by the operating system under the direction of the DBMS. However, the functions of the DBMS and the operating system at the physical level are not clear-cut and vary from system to system. Some DBMSs take advantage of many of the operating system access methods, while others use only the most basic ones and create their own file organizations. The physical level below the DBMS consists of items only the operating system knows, such as exactly how the sequencing is implemented and whether the fields of internal records are stored as contiguous bytes on the disk.

Schemas, Mappings, and Instances 2.1.4

The overall description of the database is called the **database schema**. There are three different types of schema in the database and these are defined according to the levels of abstraction of the three-level architecture illustrated in Figure 2.1. At the highest level, we have multiple **external schemas** (also called **subschemas**) that correspond to different views of the data. At the conceptual level, we have the **conceptual schema**, which describes all the entities, attributes, and relationships together with integrity constraints. At the lowest level of abstraction we have the **internal schema**, which is a complete description of the internal model, containing the definitions of stored records, the methods of representation, the data fields, and the indexes and storage structures used. There is only one conceptual schema and one internal schema per database.

The DBMS is responsible for mapping between these three types of schema. It must also check the schemas for consistency; in other words, the DBMS must check that each external schema is derivable from the conceptual schema, and it must use the information in the conceptual schema to map between each external schema and the internal schema. The conceptual schema is related to the internal schema through a **conceptual/internal mapping**. This enables the DBMS to find the actual record or combination of records in physical storage that constitute a **logical record** in the conceptual schema, together with any constraints to be enforced on the operations for that logical record. It also allows any differences in entity names, attribute names, attribute order, data types, and so on, to be resolved. Finally, each external schema is related to the conceptual schema by the **external/conceptual mapping**. This enables the DBMS to map names in the user's view on to the relevant part of the conceptual schema.

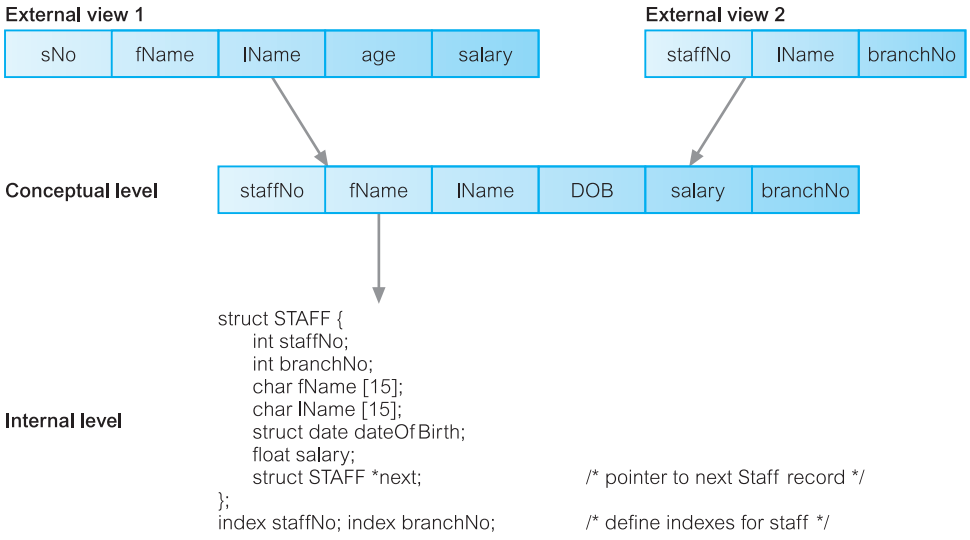

Figure 2.2
Differences between the three levels.

An example of the different levels is shown in Figure 2.2. Two different external views of staff details exist: one consisting of a staff number (sNo), first name (fName), last name (lName), age, and salary; a second consisting of a staff number (staffNo), last name (lName), and the number of the branch the member of staff works at (branchNo). These external views are merged into one conceptual view. In this merging process, the major difference is that the age field has been changed into a date of birth field, DOB. The DBMS maintains the external/conceptual mapping; for example, it maps the sNo field of the first external view to the staffNo field of the conceptual record. The conceptual level is then mapped to the internal level, which contains a physical description of the structure for the conceptual record. At this level, we see a definition of the structure in a high-level language. The structure contains a pointer, next, which allows the list of staff records to be physically linked together to form a chain. Note that the order of fields at the internal level is different from that at the conceptual level. Again, the DBMS maintains the conceptual/internal mapping.

It is important to distinguish between the description of the database and the database itself. The description of the database is the **database schema**. The schema is specified during the database design process and is not expected to change frequently. However, the actual data in the database may change frequently; for example, it changes every time we insert details of a new member of staff or a new property. The data in the database at any particular point in time is called a **database instance**. Therefore, many database instances can correspond to the same database schema. The schema is sometimes called the **intension** of the database, while an instance is called an **extension** (or **state**) of the database.

2.1.5 Data Independence

A major objective for the three-level architecture is to provide **data independence**, which means that upper levels are unaffected by changes to lower levels. There are two kinds of data independence: **logical** and **physical**.

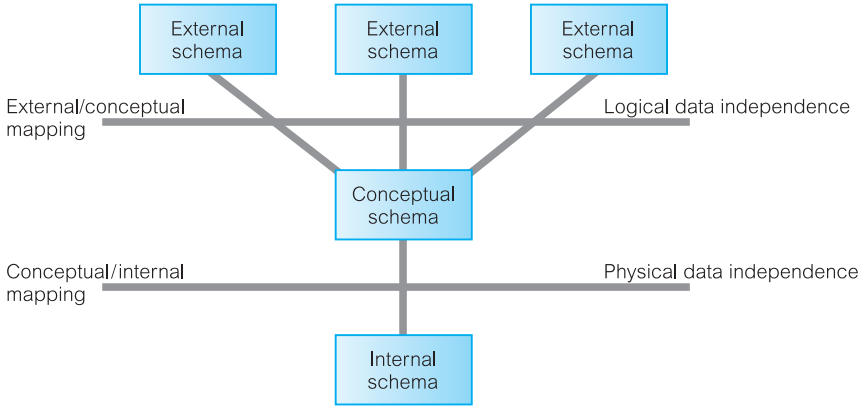

Figure 2.3
Data independence and the ANSI-SPARC three-level architecture.

Logical data independence Logical data independence refers to the immunity of the external schemas to changes in the conceptual schema.

Changes to the conceptual schema, such as the addition or removal of new entities, attributes, or relationships, should be possible without having to change existing external schemas or having to rewrite application programs. Clearly, the users for whom the changes have been made need to be aware of them, but what is important is that other users should not be.

Physical data independence Physical data independence refers to the immunity of the conceptual schema to changes in the internal schema.

Changes to the internal schema, such as using different file organizations or storage structures, using different storage devices, modifying indexes, or hashing algorithms, should be possible without having to change the conceptual or external schemas. From the users' point of view, the only effect that may be noticed is a change in performance. In fact, deterioration in performance is the most common reason for internal schema changes. Figure 2.3 illustrates where each type of data independence occurs in relation to the three-level architecture.

The two-stage mapping in the ANSI-SPARC architecture may be inefficient, but provides greater data independence. However, for more efficient mapping, the ANSI-SPARC model allows the direct mapping of external schemas on to the internal schema, thus bypassing the conceptual schema. This, of course, reduces data independence, so that every time the internal schema changes, the external schema, and any dependent application programs may also have to change.

2.2 Database Languages

A **data sublanguage** consists of two parts: a **Data Definition Language** (DDL) and a **Data Manipulation Language** (DML). The DDL is used to specify the database schema

and the DML is used to both read and update the database. These languages are called *data sublanguages* because they do not include constructs for all computing needs such as conditional or iterative statements, which are provided by the high-level programming languages. Many DBMSs have a facility for *embedding* the sublanguage in a high-level programming language such as COBOL, Fortran, Pascal, Ada, 'C', C++, Java, or Visual Basic. In this case, the high-level language is sometimes referred to as the *host language*. To compile the embedded file, the commands in the data sublanguage are first removed from the host-language program and replaced by function calls. The pre-processed file is then compiled, placed in an object module, linked with a DBMS-specific library containing the replaced functions, and executed when required. Most data sublanguages also provide *non-embedded*, or *interactive*, commands that can be input directly from a terminal.

2.2.1 The Data Definition Language (DDL)

DDL A language that allows the DBA or user to describe and name the entities, attributes, and relationships required for the application, together with any associated integrity and security constraints.

The database schema is specified by a set of definitions expressed by means of a special language called a Data Definition Language. The DDL is used to define a schema or to modify an existing one. It cannot be used to manipulate data.

The result of the compilation of the DDL statements is a set of tables stored in special files collectively called the **system catalog**. The system catalog integrates the **metadata**, that is data that describes objects in the database and makes it easier for those objects to be accessed or manipulated. The metadata contains definitions of records, data items, and other objects that are of interest to users or are required by the DBMS. The DBMS normally consults the system catalog before the actual data is accessed in the database. The terms **data dictionary** and **data directory** are also used to describe the system catalog, although the term 'data dictionary' usually refers to a more general software system than a catalog for a DBMS. We discuss the system catalog further in Section 2.4.

At a theoretical level, we could identify different DDLs for each schema in the three-level architecture, namely a DDL for the external schemas, a DDL for the conceptual schema, and a DDL for the internal schema. However, in practice, there is one comprehensive DDL that allows specification of at least the external and conceptual schemas.

2.2.2 The Data Manipulation Language (DML)

DML A language that provides a set of operations to support the basic data manipulation operations on the data held in the database.

Data manipulation operations usually include the following:

- insertion of new data into the database;
- modification of data stored in the database;
- retrieval of data contained in the database;
- deletion of data from the database.

Therefore, one of the main functions of the DBMS is to support a data manipulation language in which the user can construct statements that will cause such data manipulation to occur. Data manipulation applies to the external, conceptual, and internal levels. However, at the internal level we must define rather complex low-level procedures that allow efficient data access. In contrast, at higher levels, emphasis is placed on ease of use and effort is directed at providing efficient user interaction with the system.

The part of a DML that involves data retrieval is called a **query language**. A query language can be defined as a high-level special-purpose language used to satisfy diverse requests for the retrieval of data held in the database. The term 'query' is therefore reserved to denote a retrieval statement expressed in a query language. The terms 'query language' and 'DML' are commonly used interchangeably, although this is technically incorrect.

DMLs are distinguished by their underlying retrieval constructs. We can distinguish between two types of DML: **procedural** and **non-procedural**. The prime difference between these two data manipulation languages is that procedural languages specify *how* the output of a DML statement is to be obtained, while non-procedural DMLs describe only *what* output is to be obtained. Typically, procedural languages treat records individually, whereas non-procedural languages operate on sets of records.

Procedural DMLs

Procedural DML	A language that allows the user to tell the system what data is needed and exactly *how* to retrieve the data.

With a procedural DML, the user, or more normally the programmer, specifies what data is needed and how to obtain it. This means that the user must express all the data access operations that are to be used by calling appropriate procedures to obtain the information required. Typically, such a procedural DML retrieves a record, processes it and, based on the results obtained by this processing, retrieves another record that would be processed similarly, and so on. This process of retrievals continues until the data requested from the retrieval has been gathered. Typically, procedural DMLs are embedded in a high-level programming language that contains constructs to facilitate iteration and handle navigational logic. Network and hierarchical DMLs are normally procedural (see Section 2.3).

Non-procedural DMLs

Non-procedural DML	A language that allows the user to state *what* data is needed rather than how it is to be retrieved.

Non-procedural DMLs allow the required data to be specified in a single retrieval or update statement. With non-procedural DMLs, the user specifies what data is required without specifying how it is to be obtained. The DBMS translates a DML statement into one or more procedures that manipulate the required sets of records. This frees the user from having to know how data structures are internally implemented and what algorithms are required to retrieve and possibly transform the data, thus providing users with a considerable degree of data independence. Non-procedural languages are also called *declarative languages*. Relational DBMSs usually include some form of non-procedural language for data manipulation, typically SQL (Structured Query Language) or QBE (Query-By-Example). Non-procedural DMLs are normally easier to learn and use than procedural DMLs, as less work is done by the user and more by the DBMS. We examine SQL in detail in Chapters 5, 6, and Appendix E, and QBE in Chapter 7.

2.2.3 Fourth-Generation Languages (4GLs)

There is no consensus about what constitutes a **fourth-generation language**; it is in essence a shorthand programming language. An operation that requires hundreds of lines in a third-generation language (3GL), such as COBOL, generally requires significantly fewer lines in a 4GL.

Compared with a 3GL, which is procedural, a 4GL is non-procedural: the user defines *what* is to be done, not how. A 4GL is expected to rely largely on much higher-level components known as fourth-generation tools. The user does not define the steps that a program needs to perform a task, but instead defines parameters for the tools that use them to generate an application program. It is claimed that 4GLs can improve productivity by a factor of ten, at the cost of limiting the types of problem that can be tackled. Fourth-generation languages encompass:

- presentation languages, such as query languages and report generators;
- speciality languages, such as spreadsheets and database languages;
- application generators that define, insert, update, and retrieve data from the database to build applications;
- very high-level languages that are used to generate application code.

SQL and QBE, mentioned above, are examples of 4GLs. We now briefly discuss some of the other types of 4GL.

Forms generators

A forms generator is an interactive facility for rapidly creating data input and display layouts for screen forms. The forms generator allows the user to define what the screen is to look like, what information is to be displayed, and where on the screen it is to be displayed. It may also allow the definition of colors for screen elements and other characteristics, such as bold, underline, blinking, reverse video, and so on. The better forms generators allow the creation of derived attributes, perhaps using arithmetic operators or aggregates, and the specification of validation checks for data input.

Report generators

A report generator is a facility for creating reports from data stored in the database. It is similar to a query language in that it allows the user to ask questions of the database and retrieve information from it for a report. However, in the case of a report generator, we have much greater control over what the output looks like. We can let the report generator automatically determine how the output should look or we can create our own customized output reports using special report-generator command instructions.

There are two main types of report generator: language-oriented and visually oriented. In the first case, we enter a command in a sublanguage to define what data is to be included in the report and how the report is to be laid out. In the second case, we use a facility similar to a forms generator to define the same information.

Graphics generators

A graphics generator is a facility to retrieve data from the database and display the data as a graph showing trends and relationships in the data. Typically, it allows the user to create bar charts, pie charts, line charts, scatter charts, and so on.

Application generators

An application generator is a facility for producing a program that interfaces with the database. The use of an application generator can reduce the time it takes to design an entire software application. Application generators typically consist of pre-written modules that comprise fundamental functions that most programs use. These modules, usually written in a high-level language, constitute a 'library' of functions to choose from. The user specifies *what* the program is supposed to do; the application generator determines *how* to perform the tasks.

2.3 Data Models and Conceptual Modeling

We mentioned earlier that a schema is written using a data definition language. In fact, it is written in the data definition language of a particular DBMS. Unfortunately, this type of language is too low level to describe the data requirements of an organization in a way that is readily understandable by a variety of users. What we require is a higher-level description of the schema: that is, a **data model**.

Data model An integrated collection of concepts for describing and manipulating data, relationships between data, and constraints on the data in an organization.

A model is a representation of 'real world' objects and events, and their associations. It is an abstraction that concentrates on the essential, inherent aspects of an organization and ignores the accidental properties. A data model represents the organization itself. It should provide the basic concepts and notations that will allow database designers and end-users

unambiguously and accurately to communicate their understanding of the organizational data. A data model can be thought of as comprising three components:

(1) a **structural part**, consisting of a set of rules according to which databases can be constructed;

(2) a **manipulative part**, defining the types of operation that are allowed on the data (this includes the operations that are used for updating or retrieving data from the database and for changing the structure of the database);

(3) possibly a **set of integrity constraints**, which ensures that the data is accurate.

The purpose of a data model is to represent data and to make the data understandable. If it does this, then it can be easily used to design a database. To reflect the ANSI-SPARC architecture introduced in Section 2.1, we can identify three related data models:

(1) an external data model, to represent each user's view of the organization, sometimes called the **Universe of Discourse** (UoD);

(2) a conceptual data model, to represent the logical (or community) view that is DBMS-independent;

(3) an internal data model, to represent the conceptual schema in such a way that it can be understood by the DBMS.

There have been many data models proposed in the literature. They fall into three broad categories: **object-based**, **record-based**, and **physical** data models. The first two are used to describe data at the conceptual and external levels, the latter is used to describe data at the internal level.

2.3.1 Object-Based Data Models

Object-based data models use concepts such as entities, attributes, and relationships. An **entity** is a distinct object (a person, place, thing, concept, event) in the organization that is to be represented in the database. An **attribute** is a property that describes some aspect of the object that we wish to record, and a **relationship** is an association between entities. Some of the more common types of object-based data model are:

- Entity–Relationship
- Semantic
- Functional
- Object-Oriented.

The Entity–Relationship model has emerged as one of the main techniques for database design and forms the basis for the database design methodology used in this book. The object-oriented data model extends the definition of an entity to include not only the attributes that describe the **state** of the object but also the actions that are associated with the object, that is, its **behavior**. The object is said to **encapsulate** both state and behavior. We look at the Entity–Relationship model in depth in Chapters 11 and 12 and

the object-oriented model in Chapters 25–28. We also examine the functional data model in Section 26.1.2.

Record-Based Data Models 2.3.2

In a record-based model, the database consists of a number of fixed-format records possibly of differing types. Each record type defines a fixed number of fields, each typically of a fixed length. There are three principal types of record-based logical data model: the **relational data model**, the **network data model**, and the **hierarchical data model**. The hierarchical and network data models were developed almost a decade before the relational data model, so their links to traditional file processing concepts are more evident.

Relational data model

The relational data model is based on the concept of mathematical relations. In the relational model, data and relationships are represented as tables, each of which has a number of columns with a unique name. Figure 2.4 is a sample instance of a relational schema for part of the *DreamHome* case study, showing branch and staff details. For example, it shows that employee John White is a manager with a salary of £30,000, who works at branch (branchNo) B005, which, from the first table, is at 22 Deer Rd in London. It is important to note that there is a relationship between Staff and Branch: a branch office *has* staff. However, there is no explicit link between these two tables; it is only by knowing that the attribute branchNo in the Staff relation is the same as the branchNo of the Branch relation that we can establish that a relationship exists.

Note that the relational data model requires only that the database be perceived by the user as tables. However, this perception applies only to the logical structure of the

Branch

branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Figure 2.4
A sample instance of a relational schema.

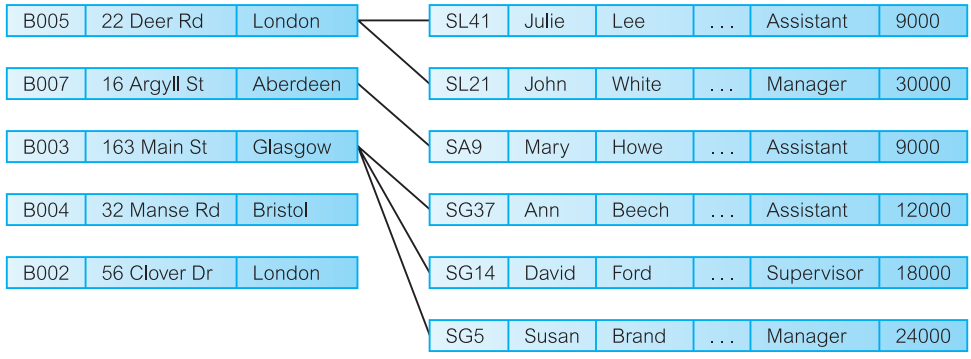

Figure 2.5
A sample instance of a network schema.

database, that is, the external and conceptual levels of the ANSI-SPARC architecture. It does not apply to the physical structure of the database, which can be implemented using a variety of storage structures. We discuss the relational data model in Chapter 3.

Network data model

In the network model, data is represented as collections of **records**, and relationships are represented by **sets**. Compared with the relational model, relationships are explicitly modeled by the sets, which become pointers in the implementation. The records are organized as generalized graph structures with records appearing as **nodes** (also called **segments**) and sets as **edges** in the graph. Figure 2.5 illustrates an instance of a network schema for the same data set presented in Figure 2.4. The most popular network DBMS is Computer Associates' IDMS/R. We discuss the network data model in more detail on the Web site for this book (see Preface for the URL).

Hierarchical data model

The hierarchical model is a restricted type of network model. Again, data is represented as collections of **records** and relationships are represented by **sets**. However, the hierarchical model allows a node to have only one parent. A hierarchical model can be represented as a tree graph, with records appearing as **nodes** (also called **segments**) and sets as **edges**. Figure 2.6 illustrates an instance of a hierarchical schema for the same data set presented in Figure 2.4. The main hierarchical DBMS is IBM's IMS, although IMS also provides non-hierarchical features. We discuss the hierarchical data model in more detail on the Web site for this book (see Preface for the URL).

Record-based (logical) data models are used to specify the overall structure of the database and a higher-level description of the implementation. Their main drawback lies in the fact that they do not provide adequate facilities for explicitly specifying constraints on the data, whereas the object-based data models lack the means of logical structure specification but provide more semantic substance by allowing the user to specify constraints on the data.

The majority of modern commercial systems are based on the relational paradigm, whereas the early database systems were based on either the network or hierarchical data

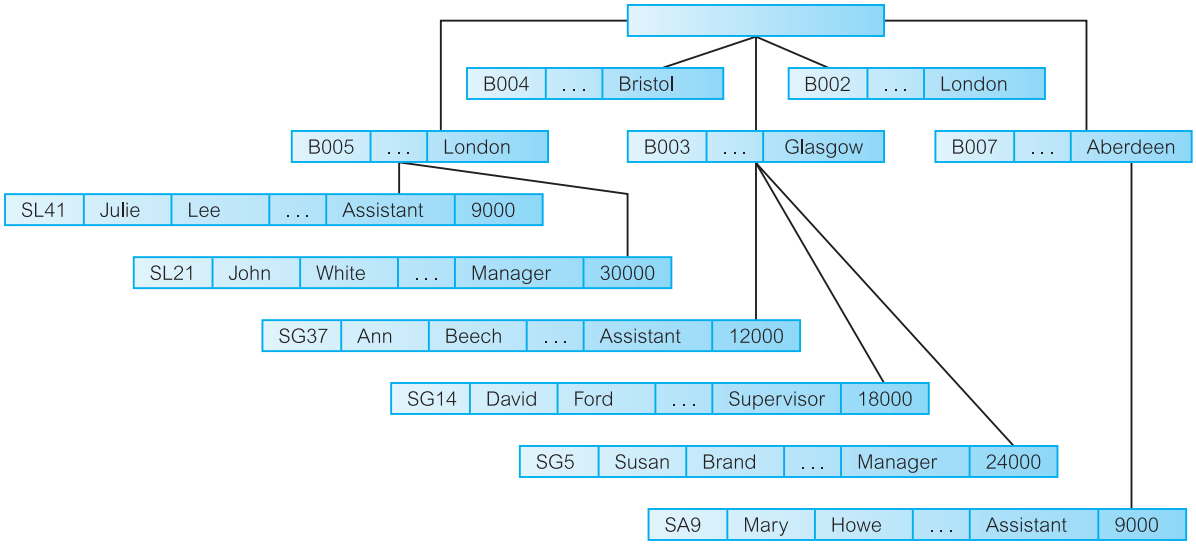

Figure 2.6 A sample instance of a hierarchical schema.

models. The latter two models require the user to have knowledge of the physical database being accessed, whereas the former provides a substantial amount of data independence. Hence, while relational systems adopt a **declarative** approach to database processing (that is, they specify *what* data is to be retrieved), network and hierarchical systems adopt a **navigational** approach (that is, they specify *how* the data is to be retrieved).

Physical Data Models 2.3.3

Physical data models describe how data is stored in the computer, representing information such as record structures, record orderings, and access paths. There are not as many physical data models as logical data models, the most common ones being the *unifying model* and the *frame memory*.

Conceptual Modeling 2.3.4

From an examination of the three-level architecture, we see that the conceptual schema is the 'heart' of the database. It supports all the external views and is, in turn, supported by the internal schema. However, the internal schema is merely the physical implementation of the conceptual schema. The conceptual schema should be a complete and accurate representation of the data requirements of the enterprise.[†] If this is not the case, some information about the enterprise will be missing or incorrectly represented and we will have difficulty fully implementing one or more of the external views.

[†] When we are discussing the organization in the context of database design we normally refer to the business or organization as the *enterprise*.

Conceptual modeling, or **conceptual database design**, is the process of constructing a model of the information use in an enterprise that is independent of implementation details, such as the target DBMS, application programs, programming languages, or any other physical considerations. This model is called a **conceptual data model**. Conceptual models are also referred to as logical models in the literature. However, in this book we make a distinction between conceptual and logical data models. The conceptual model is independent of all implementation details, whereas the logical model assumes knowledge of the underlying data model of the target DBMS. In Chapters 15 and 16 we present a methodology for database design that begins by producing a conceptual data model, which is then refined into a logical model based on the relational data model. We discuss database design in more detail in Section 9.6.

2.4 Functions of a DBMS

In this section we look at the types of function and service we would expect a DBMS to provide. Codd (1982) lists eight services that should be provided by any full-scale DBMS, and we have added two more that might reasonably be expected to be available.

(1) Data storage, retrieval, and update

A DBMS must furnish users with the ability to store, retrieve, and update data in the database.

This is the fundamental function of a DBMS. From the discussion in Section 2.1, clearly in providing this functionality the DBMS should hide the internal physical implementation details (such as file organization and storage structures) from the user.

(2) A user-accessible catalog

A DBMS must furnish a catalog in which descriptions of data items are stored and which is accessible to users.

A key feature of the ANSI-SPARC architecture is the recognition of an integrated **system catalog** to hold data about the schemas, users, applications, and so on. The catalog is expected to be accessible to users as well as to the DBMS. A system catalog, or data dictionary, is a repository of information describing the data in the database: it is, the 'data about the data' or **metadata**. The amount of information and the way the information is used vary with the DBMS. Typically, the system catalog stores:

- names, types, and sizes of data items;
- names of relationships;
- integrity constraints on the data;
- names of authorized users who have access to the data;

- the data items that each user can access and the types of access allowed; for example, insert, update, delete, or read access;
- external, conceptual, and internal schemas and the mappings between the schemas, as described in Section 2.1.4;
- usage statistics, such as the frequencies of transactions and counts on the number of accesses made to objects in the database.

The DBMS system catalog is one of the fundamental components of the system. Many of the software components that we describe in the next section rely on the system catalog for information. Some benefits of a system catalog are:

- Information about data can be collected and stored centrally. This helps to maintain control over the data as a resource.
- The meaning of data can be defined, which will help other users understand the purpose of the data.
- Communication is simplified, since exact meanings are stored. The system catalog may also identify the user or users who own or access the data.
- Redundancy and inconsistencies can be identified more easily since the data is centralized.
- Changes to the database can be recorded.
- The impact of a change can be determined before it is implemented, since the system catalog records each data item, all its relationships, and all its users.
- Security can be enforced.
- Integrity can be ensured.
- Audit information can be provided.

Some authors make a distinction between system catalog and data directory, where a data directory holds information relating to where data is stored and how it is stored. The International Organization for Standardization (ISO) has adopted a standard for data dictionaries called Information Resource Dictionary System (IRDS) (ISO, 1990, 1993). IRDS is a software tool that can be used to control and document an organization's information sources. It provides a definition for the tables that comprise the data dictionary and the operations that can be used to access these tables. We use the term 'system catalog' in this book to refer to all repository information. We discuss other types of statistical information stored in the system catalog to assist with query optimization in Section 21.4.1.

(3) Transaction support

A DBMS must furnish a mechanism which will ensure either that all the updates corresponding to a given transaction are made or that none of them is made.

A transaction is a series of actions, carried out by a single user or application program, which accesses or changes the contents of the database. For example, some simple transactions for the *DreamHome* case study might be to add a new member of staff to the database, to update the salary of a member of staff, or to delete a property from the register.

Figure 2.7
The lost update problem.

Time	T_1	T_2	bal_x
t_1		read(bal_x)	100
t_2	read(bal_x)	$bal_x = bal_x + 100$	100
t_3	$bal_x = bal_x - 10$	write(bal_x)	200
t_4	write(bal_x)		90
t_5			90

A more complicated example might be to delete a member of staff from the database *and* to reassign the properties that he or she managed to another member of staff. In this case, there is more than one change to be made to the database. If the transaction fails during execution, perhaps because of a computer crash, the database will be in an **inconsistent** state: some changes will have been made and others not. Consequently, the changes that have been made will have to be undone to return the database to a consistent state again. We discuss transaction support in Section 20.1.

(4) Concurrency control services

A DBMS must furnish a mechanism to ensure that the database is updated correctly when multiple users are updating the database concurrently.

One major objective in using a DBMS is to enable many users to access shared data concurrently. Concurrent access is relatively easy if all users are only reading data, as there is no way that they can interfere with one another. However, when two or more users are accessing the database simultaneously and at least one of them is updating data, there may be interference that can result in inconsistencies. For example, consider two transactions T_1 and T_2, which are executing concurrently as illustrated in Figure 2.7.

T_1 is withdrawing £10 from an account (with balance bal_x) and T_2 is depositing £100 into the same account. If these transactions were executed **serially**, one after the other with no interleaving of operations, the final balance would be £190 regardless of which was performed first. However, in this example transactions T_1 and T_2 start at nearly the same time and both read the balance as £100. T_2 then increases bal_x by £100 to £200 and stores the update in the database. Meanwhile, transaction T_1 decrements its copy of bal_x by £10 to £90 and stores this value in the database, overwriting the previous update and thereby 'losing' £100.

The DBMS must ensure that, when multiple users are accessing the database, interference cannot occur. We discuss this issue fully in Section 20.2.

(5) Recovery services

A DBMS must furnish a mechanism for recovering the database in the event that the database is damaged in any way.

When discussing transaction support, we mentioned that if the transaction fails then the database has to be returned to a consistent state. This may be a result of a system crash, media failure, a hardware or software error causing the DBMS to stop, or it may be the result of the user detecting an error during the transaction and aborting the transaction before it completes. In all these cases, the DBMS must provide a mechanism to recover the database to a consistent state. We discuss database recovery in Section 20.3.

(6) Authorization services

A DBMS must furnish a mechanism to ensure that only authorized users can access the database.

It is not difficult to envisage instances where we would want to prevent some of the data stored in the database from being seen by all users. For example, we may want only branch managers to see salary-related information for staff and prevent all other users from seeing this data. Additionally, we may want to protect the database from unauthorized access. The term **security** refers to the protection of the database against unauthorized access, either intentional or accidental. We expect the DBMS to provide mechanisms to ensure the data is secure. We discuss security in Chapter 19.

(7) Support for data communication

A DBMS must be capable of integrating with communication software.

Most users access the database from workstations. Sometimes these workstations are connected directly to the computer hosting the DBMS. In other cases, the workstations are at remote locations and communicate with the computer hosting the DBMS over a network. In either case, the DBMS receives requests as **communications messages** and responds in a similar way. All such transmissions are handled by a Data Communication Manager (DCM). Although the DCM is not part of the DBMS, it is necessary for the DBMS to be capable of being integrated with a variety of DCMs if the system is to be commercially viable. Even DBMSs for personal computers should be capable of being run on a local area network so that one centralized database can be established for users to share, rather than having a series of disparate databases, one for each user. This does not imply that the database has to be distributed across the network; rather that users should be able to access a centralized database from remote locations. We refer to this type of topology as *distributed processing* (see Section 22.1.1).

(8) Integrity services

A DBMS must furnish a means to ensure that both the data in the database and changes to the data follow certain rules.

Database integrity refers to the correctness and consistency of stored data: it can be considered as another type of database protection. While integrity is related to security, it has wider implications: integrity is concerned with the quality of data itself. Integrity is usually expressed in terms of *constraints*, which are consistency rules that the database is not permitted to violate. For example, we may want to specify a constraint that no member of staff can manage more than 100 properties at any one time. Here, we would want the DBMS to check when we assign a property to a member of staff that this limit would not be exceeded and to prevent the assignment from occurring if the limit has been reached.

In addition to these eight services, we could also reasonably expect the following two services to be provided by a DBMS.

(9) Services to promote data independence

A DBMS must include facilities to support the independence of programs from the actual structure of the database.

We discussed the concept of data independence in Section 2.1.5. Data independence is normally achieved through a view or subschema mechanism. Physical data independence is easier to achieve: there are usually several types of change that can be made to the physical characteristics of the database without affecting the views. However, complete logical data independence is more difficult to achieve. The addition of a new entity, attribute, or relationship can usually be accommodated, but not their removal. In some systems, any type of change to an existing component in the logical structure is prohibited.

(10) Utility services

A DBMS should provide a set of utility services.

Utility programs help the DBA to administer the database effectively. Some utilities work at the external level, and consequently can be produced by the DBA. Other utilities work at the internal level and can be provided only by the DBMS vendor. Examples of utilities of the latter kind are:

- import facilities, to load the database from flat files, and export facilities, to unload the database to flat files;
- monitoring facilities, to monitor database usage and operation;
- statistical analysis programs, to examine performance or usage statistics;
- index reorganization facilities, to reorganize indexes and their overflows;
- garbage collection and reallocation, to remove deleted records physically from the storage devices, to consolidate the space released, and to reallocate it where it is needed.

2.5 Components of a DBMS

DBMSs are highly complex and sophisticated pieces of software that aim to provide the services discussed in the previous section. It is not possible to generalize the component structure of a DBMS as it varies greatly from system to system. However, it is useful when trying to understand database systems to try to view the components and the relationships between them. In this section, we present a possible architecture for a DBMS. We examine the architecture of the Oracle DBMS in Section 8.2.2.

A DBMS is partitioned into several software components (or *modules*), each of which is assigned a specific operation. As stated previously, some of the functions of the DBMS are supported by the underlying operating system. However, the operating system provides only basic services and the DBMS must be built on top of it. Thus, the design of a DBMS must take into account the interface between the DBMS and the operating system.

The major software components in a DBMS environment are depicted in Figure 2.8. This diagram shows how the DBMS interfaces with other software components, such as user queries and access methods (file management techniques for storing and retrieving data records). We will provide an overview of file organizations and access methods in Appendix C. For a more comprehensive treatment, the interested reader is referred to Teorey and Fry (1982), Weiderhold (1983), Smith and Barnes (1987), and Ullman (1988).

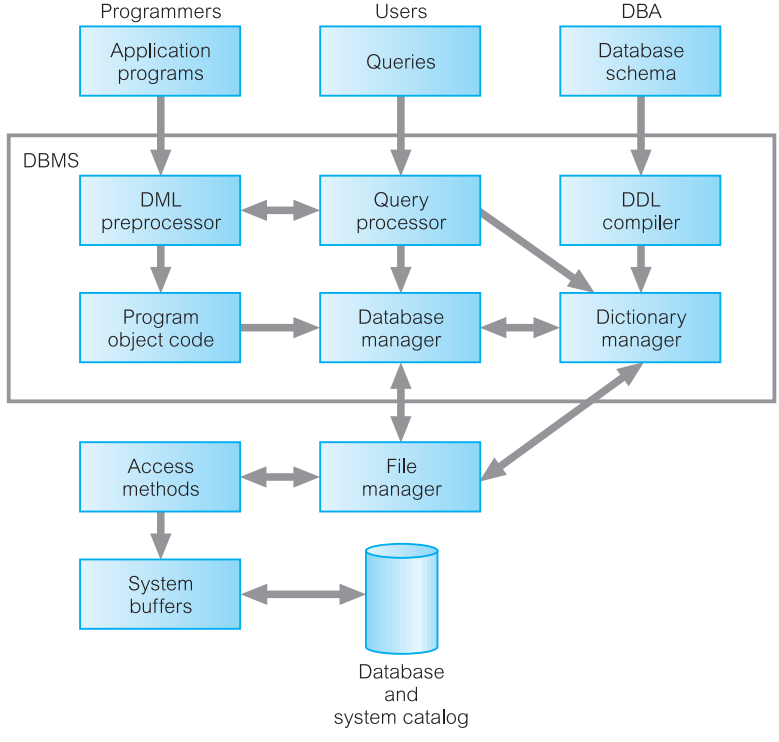

Figure 2.8
Major components of a DBMS.

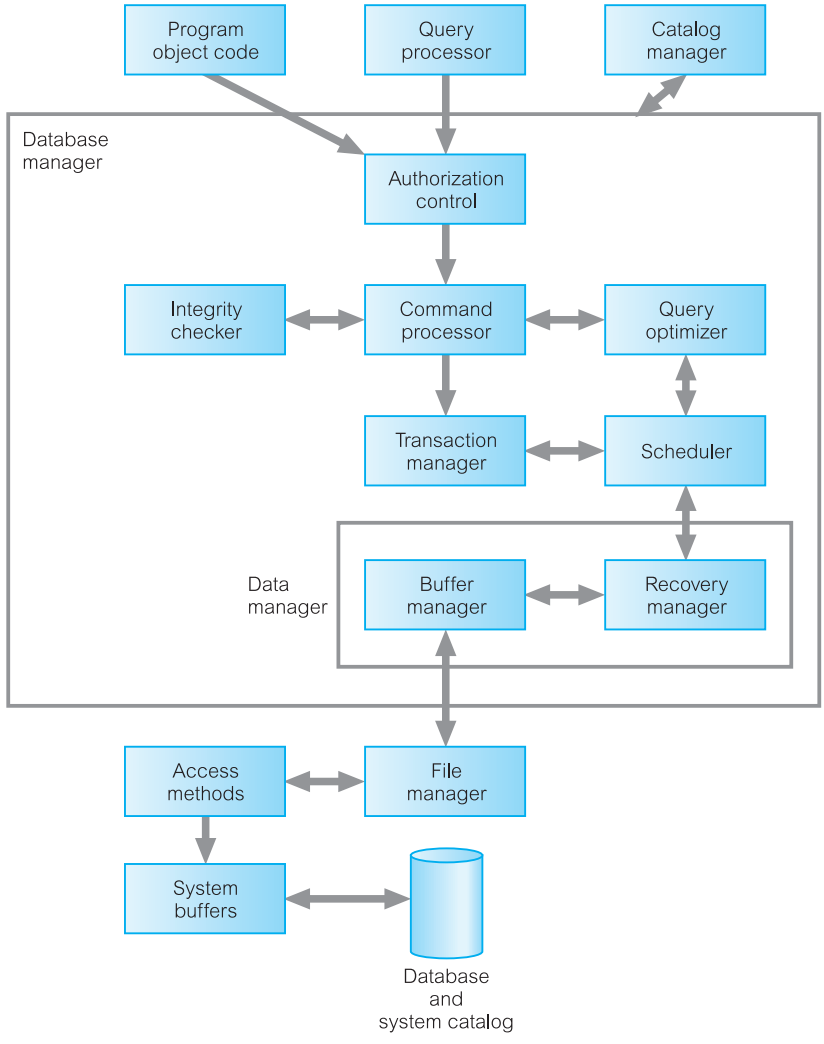

Figure 2.9
Components of a database manager.

Figure 2.8 shows the following components:

- *Query processor* This is a major DBMS component that transforms queries into a series of low-level instructions directed to the database manager. We discuss query processing in Chapter 21.
- *Database manager (DM)* The DM interfaces with user-submitted application programs and queries. The DM accepts queries and examines the external and conceptual schemas to determine what conceptual records are required to satisfy the request. The DM then places a call to the file manager to perform the request. The components of the DM are shown in Figure 2.9.
- *File manager* The file manager manipulates the underlying storage files and manages the allocation of storage space on disk. It establishes and maintains the list of structures

and indexes defined in the internal schema. If hashed files are used it calls on the hashing functions to generate record addresses. However, the file manager does not directly manage the physical input and output of data. Rather it passes the requests on to the appropriate access methods, which either read data from or write data into the system buffer (or *cache*).

- *DML preprocessor* This module converts DML statements embedded in an application program into standard function calls in the host language. The DML preprocessor must interact with the query processor to generate the appropriate code.
- *DDL compiler* The DDL compiler converts DDL statements into a set of tables containing metadata. These tables are then stored in the system catalog while control information is stored in data file headers.
- *Catalog manager* The catalog manager manages access to and maintains the system catalog. The system catalog is accessed by most DBMS components.

The major software components for the *database manager* are as follows:

- *Authorization control* This module checks that the user has the necessary authorization to carry out the required operation.
- *Command processor* Once the system has checked that the user has authority to carry out the operation, control is passed to the command processor.
- *Integrity checker* For an operation that changes the database, the integrity checker checks that the requested operation satisfies all necessary integrity constraints (such as key constraints).
- *Query optimizer* This module determines an optimal strategy for the query execution. We discuss query optimization in Chapter 21.
- *Transaction manager* This module performs the required processing of operations it receives from transactions.
- *Scheduler* This module is responsible for ensuring that concurrent operations on the database proceed without conflicting with one another. It controls the relative order in which transaction operations are executed.
- *Recovery manager* This module ensures that the database remains in a consistent state in the presence of failures. It is responsible for transaction commit and abort.
- *Buffer manager* This module is responsible for the transfer of data between main memory and secondary storage, such as disk and tape. The recovery manager and the buffer manager are sometimes referred to collectively as the *data manager*. The buffer manager is sometimes known as the *cache manager*.

We discuss the last four modules in Chapter 20. In addition to the above modules, several other data structures are required as part of the physical-level implementation. These structures include data and index files, and the system catalog. An attempt has been made to standardize DBMSs, and a reference model was proposed by the Database Architecture Framework Task Group (DAFTG, 1986). The purpose of this reference model was to define a conceptual framework aiming to divide standardization attempts into manageable pieces and to show at a very broad level how these pieces could be interrelated.

2.6 Multi-User DBMS Architectures

In this section we look at the common architectures that are used to implement multi-user database management systems, namely teleprocessing, file-server, and client–server.

2.6.1 Teleprocessing

The traditional architecture for multi-user systems was teleprocessing, where there is one computer with a single central processing unit (CPU) and a number of terminals, as illustrated in Figure 2.10. All processing is performed within the boundaries of the same physical computer. User terminals are typically 'dumb' ones, incapable of functioning on their own. They are cabled to the central computer. The terminals send messages via the communications control subsystem of the operating system to the user's application program, which in turn uses the services of the DBMS. In the same way, messages are routed back to the user's terminal. Unfortunately, this architecture placed a tremendous burden on the central computer, which not only had to run the application programs and the DBMS, but also had to carry out a significant amount of work on behalf of the terminals (such as formatting data for display on the screen).

In recent years, there have been significant advances in the development of high-performance personal computers and networks. There is now an identifiable trend in industry towards **downsizing**, that is, replacing expensive mainframe computers with more cost-effective networks of personal computers that achieve the same, or even better, results. This trend has given rise to the next two architectures: file-server and client–server.

2.6.2 File-Server Architecture

In a file-server environment, the processing is distributed about the network, typically a local area network (LAN). The file-server holds the files required by the applications and the DBMS. However, the applications and the DBMS run on each workstation, requesting

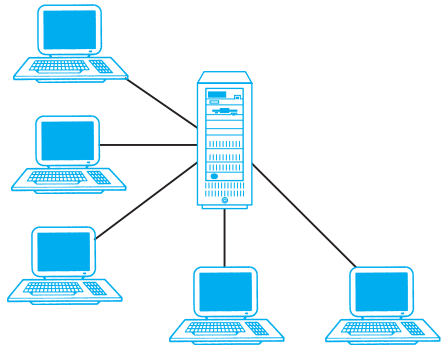

Figure 2.10
Teleprocessing topology.

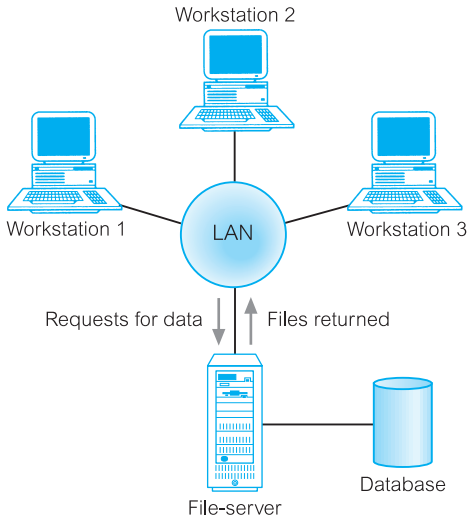

Figure 2.11 File-server architecture.

files from the file-server when necessary, as illustrated in Figure 2.11. In this way, the file-server acts simply as a shared hard disk drive. The DBMS on each workstation sends requests to the file-server for all data that the DBMS requires that is stored on disk. This approach can generate a significant amount of network traffic, which can lead to performance problems. For example, consider a user request that requires the names of staff who work in the branch at 163 Main St. We can express this request in SQL (see Chapter 5) as:

```
SELECT fName, lName
FROM Branch b, Staff s
WHERE b.branchNo = s.branchNo AND b.street = '163 Main St';
```

As the file-server has no knowledge of SQL, the DBMS has to request the files corresponding to the Branch and Staff relations from the file-server, rather than just the staff names that satisfy the query.

The file-server architecture, therefore, has three main disadvantages:

(1) There is a large amount of network traffic.

(2) A full copy of the DBMS is required on each workstation.

(3) Concurrency, recovery, and integrity control are more complex because there can be multiple DBMSs accessing the same files.

Traditional Two-Tier Client–Server Architecture 2.6.3

To overcome the disadvantages of the first two approaches and accommodate an increasingly decentralized business environment, the client–server architecture was developed. Client–server refers to the way in which software components interact to form a system.

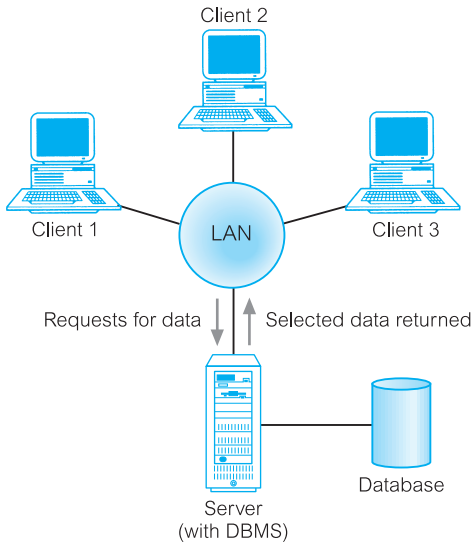

Figure 2.12
Client–server architecture.

As the name suggests, there is a **client** process, which requires some resource, and a **server**, which provides the resource. There is no requirement that the client and server must reside on the same machine. In practice, it is quite common to place a server at one site in a local area network and the clients at the other sites. Figure 2.12 illustrates the client–server architecture and Figure 2.13 shows some possible combinations of the client–server topology.

Data-intensive business applications consist of four major components: the database, the transaction logic, the business and data application logic, and the user interface. The traditional two-tier client–server architecture provides a very basic separation of these components. The client (tier 1) is primarily responsible for the *presentation* of data to the user, and the server (tier 2) is primarily responsible for supplying *data services* to the client, as illustrated in Figure 2.14. Presentation services handle user interface actions and the main business and data application logic. Data services provide limited business application logic, typically validation that the client is unable to carry out due to lack of information, and access to the requested data, independent of its location. The data can come from relational DBMSs, object-relational DBMSs, object-oriented DBMSs, legacy DBMSs, or proprietary data access systems. Typically, the client would run on end-user desktops and interact with a centralized database server over a network.

A typical interaction between client and server is as follows. The client takes the user's request, checks the syntax and generates database requests in SQL or another database language appropriate to the application logic. It then transmits the message to the server, waits for a response, and formats the response for the end-user. The server accepts and processes the database requests, then transmits the results back to the client. The processing involves checking authorization, ensuring integrity, maintaining the system catalog, and performing query and update processing. In addition, it also provides concurrency and recovery control. The operations of client and server are summarized in Table 2.1.

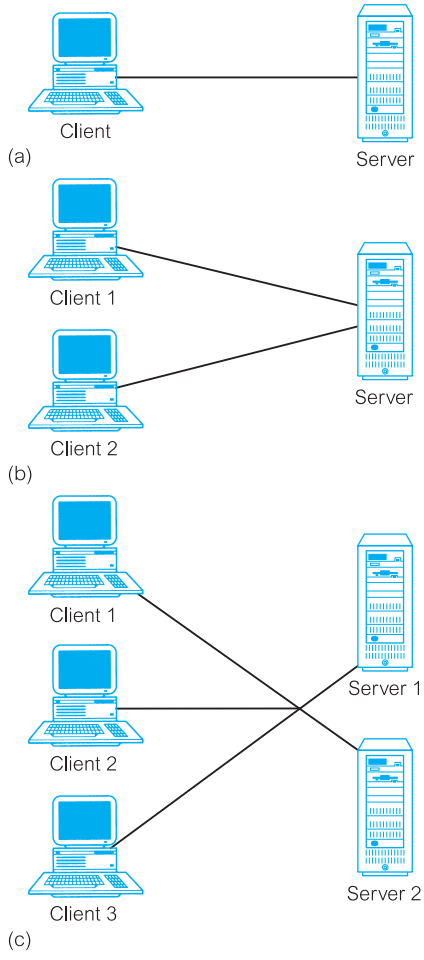

Figure 2.13
Alternative client–server topologies: (a) single client, single server; (b) multiple clients, single server; (c) multiple clients, multiple servers.

There are many advantages to this type of architecture. For example:

- It enables wider access to existing databases.
- Increased performance – if the clients and server reside on different computers then different CPUs can be processing applications in parallel. It should also be easier to tune the server machine if its only task is to perform database processing.
- Hardware costs may be reduced – it is only the server that requires storage and processing power sufficient to store and manage the database.
- Communication costs are reduced – applications carry out part of the operations on the client and send only requests for database access across the network, resulting in less data being sent across the network.

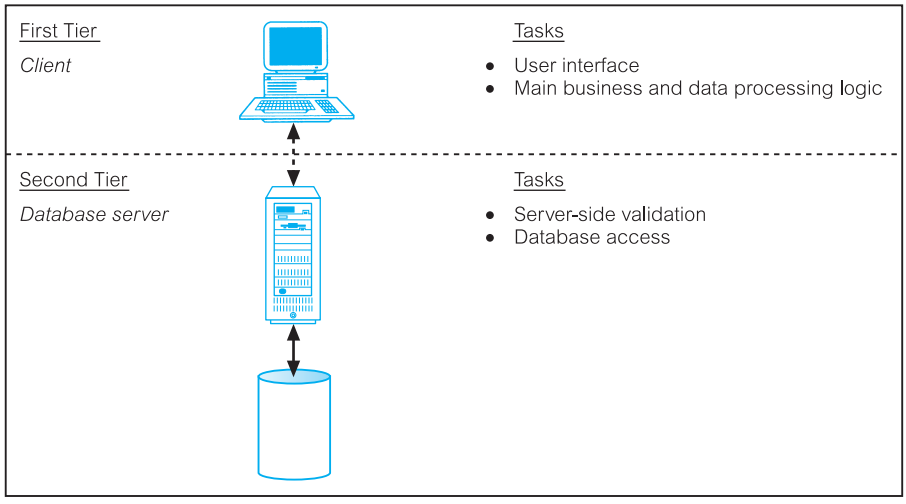

Figure 2.14 The traditional two-tier client–server architecture.

Table 2.1 Summary of client–server functions.

Client	Server
Manages the user interface	Accepts and processes database requests from clients
Accepts and checks syntax of user input	Checks authorization
Processes application logic	Ensures integrity constraints not violated
Generates database requests and transmits to server	Performs query/update processing and transmits response to client
Passes response back to user	Maintains system catalog
	Provides concurrent database access
	Provides recovery control

- Increased consistency – the server can handle integrity checks, so that constraints need be defined and validated only in the one place, rather than having each application program perform its own checking.
- It maps on to open systems architecture quite naturally.

Some database vendors have used this architecture to indicate distributed database capability, that is a collection of multiple, logically interrelated databases distributed over a computer network. However, although the client–server architecture can be used to provide distributed DBMSs, by itself it does not constitute a distributed DBMS. We discuss distributed DBMSs in Chapters 22 and 23.

2.6.4 Three-Tier Client–Server Architecture

The need for enterprise scalability challenged this traditional two-tier client–server model. In the mid-1990s, as applications became more complex and potentially could be deployed

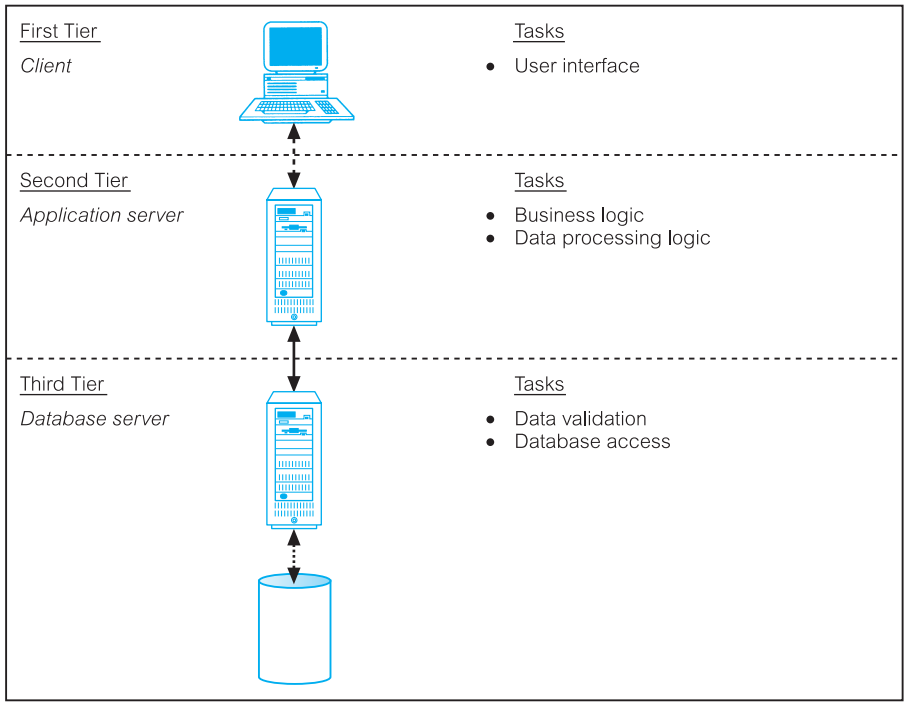

Figure 2.15
The three-tier architecture.

to hundreds or thousands of end-users, the client side presented two problems that prevented true scalability:

- A 'fat' client, requiring considerable resources on the client's computer to run effectively. This includes disk space, RAM, and CPU power.
- A significant client-side administration overhead.

By 1995, a new variation of the traditional two-tier client–server model appeared to solve the problem of enterprise scalability. This new architecture proposed three layers, each potentially running on a different platform:

(1) The user interface layer, which runs on the end-user's computer (the *client*).

(2) The business logic and data processing layer. This middle tier runs on a server and is often called the *application server*.

(3) A DBMS, which stores the data required by the middle tier. This tier may run on a separate server called the *database server*.

As illustrated in Figure 2.15 the client is now responsible only for the application's user interface and perhaps performing some simple logic processing, such as input validation, thereby providing a 'thin' client. The core business logic of the application now resides in its own layer, physically connected to the client and database server over a local area network (LAN) or wide area network (WAN). One application server is designed to serve multiple clients.

The three-tier design has many advantages over traditional two-tier or single-tier designs, which include:

- The need for less expensive hardware because the client is 'thin'.
- Application maintenance is centralized with the transfer of the business logic for many end-users into a single application server. This eliminates the concerns of software distribution that are problematic in the traditional two-tier client–server model.
- The added modularity makes it easier to modify or replace one tier without affecting the other tiers.
- Load balancing is easier with the separation of the core business logic from the database functions.

An additional advantage is that the three-tier architecture maps quite naturally to the Web environment, with a Web browser acting as the 'thin' client, and a Web server acting as the application server. The three-tier architecture can be extended to *n*-tiers, with additional tiers added to provide more flexibility and scalability. For example, the middle tier of the three-tier architecture could be split into two, with one tier for the Web server and another for the application server.

This three-tier architecture has proved more appropriate for some environments, such as the Internet and corporate intranets where a Web browser can be used as a client. It is also an important architecture for **Transaction Processing Monitors**, as we discuss next.

2.6.5 Transaction Processing Monitors

TP Monitor A program that controls data transfer between clients and servers in order to provide a consistent environment, particularly for online transaction processing (OLTP).

Complex applications are often built on top of several **resource managers** (such as DBMSs, operating systems, user interfaces, and messaging software). A Transaction Processing Monitor, or TP Monitor, is a middleware component that provides access to the services of a number of resource managers and provides a uniform interface for programmers who are developing transactional software. A TP Monitor forms the middle tier of a three-tier architecture, as illustrated in Figure 2.16. TP Monitors provide significant advantages, including:

- *Transaction routing* The TP Monitor can increase scalability by directing transactions to specific DBMSs.
- *Managing distributed transactions* The TP Monitor can manage transactions that require access to data held in multiple, possibly heterogeneous, DBMSs. For example, a transaction may require to update data items held in an Oracle DBMS at site 1, an Informix DBMS at site 2, and an IMS DBMS as site 3. TP Monitors normally control transactions using the X/Open Distributed Transaction Processing (DTP) standard. A

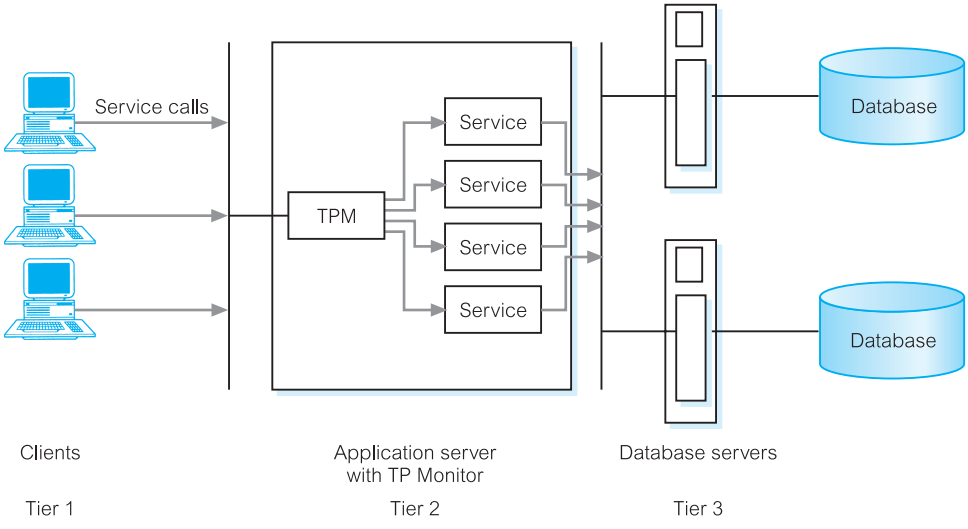

Figure 2.16
Transaction Processing Monitor as the middle tier of a three-tier client–server architecture.

DBMS that supports this standard can function as a resource manager under the control of a TP Monitor acting as a transaction manager. We discuss distributed transactions and the DTP standard in Chapters 22 and 23.

- *Load balancing* The TP Monitor can balance client requests across multiple DBMSs on one or more computers by directing client service calls to the least loaded server. In addition, it can dynamically bring in additional DBMSs as required to provide the necessary performance.
- *Funneling* In environments with a large number of users, it may sometimes be difficult for all users to be logged on simultaneously to the DBMS. In many cases, we would find that users generally do not need continuous access to the DBMS. Instead of each user connecting to the DBMS, the TP Monitor can establish connections with the DBMSs as and when required, and can funnel user requests through these connections. This allows a larger number of users to access the available DBMSs with a potentially much smaller number of connections, which in turn would mean less resource usage.
- *Increased reliability* The TP Monitor acts as a *transaction manager*, performing the necessary actions to maintain the consistency of the database, with the DBMS acting as a *resource manager*. If the DBMS fails, the TP Monitor may be able to resubmit the transaction to another DBMS or can hold the transaction until the DBMS becomes available again.

TP Monitors are typically used in environments with a very high volume of transactions, where the TP Monitor can be used to offload processes from the DBMS server. Prominent examples of TP Monitors include CICS and Encina from IBM (which are primarily used on IBM AIX or Windows NT and bundled now in the IBM TXSeries) and Tuxedo from BEA Systems.

Chapter Summary

- The ANSI-SPARC database architecture uses **three levels** of abstraction: **external**, **conceptual**, and **internal**. The **external level** consists of the users' views of the database. The **conceptual level** is the community view of the database. It specifies the information content of the entire database, independent of storage considerations. The conceptual level represents all entities, their attributes, and their relationships, as well as the constraints on the data, and security and integrity information. The **internal level** is the computer's view of the database. It specifies how data is represented, how records are sequenced, what indexes and pointers exist, and so on.
- The **external/conceptual mapping** transforms requests and results between the external and conceptual levels. The **conceptual/internal mapping** transforms requests and results between the conceptual and internal levels.
- A **database schema** is a description of the database structure. Data independence makes each level immune to changes to lower levels. **Logical data independence** refers to the immunity of the external schemas to changes in the conceptual schema. **Physical data independence** refers to the immunity of the conceptual schema to changes in the internal schema.
- A data sublanguage consists of two parts: a **Data Definition Language (DDL)** and a **Data Manipulation Language (DML)**. The DDL is used to specify the database schema and the DML is used to both read and update the database. The part of a DML that involves data retrieval is called a **query language**.
- A **data model** is a collection of concepts that can be used to describe a set of data, the operations to manipulate the data, and a set of integrity constraints for the data. They fall into three broad categories: **object-based** data models, **record-based** data models, and **physical** data models. The first two are used to describe data at the conceptual and external levels; the latter is used to describe data at the internal level.
- Object-based data models include the Entity–Relationship, semantic, functional, and object-oriented models. Record-based data models include the relational, network, and hierarchical models.
- **Conceptual modeling** is the process of constructing a detailed architecture for a database that is independent of implementation details, such as the target DBMS, application programs, programming languages, or any other physical considerations. The design of the conceptual schema is critical to the overall success of the system. It is worth spending the time and energy necessary to produce the best possible conceptual design.
- **Functions** and **services** of a multi-user DBMS include data storage, retrieval, and update; a user-accessible catalog; transaction support; concurrency control and recovery services; authorization services; support for data communication; integrity services; services to promote data independence; utility services.
- The **system catalog** is one of the fundamental components of a DBMS. It contains 'data about the data', or **metadata**. The catalog should be accessible to users. The Information Resource Dictionary System is an ISO standard that defines a set of access methods for a data dictionary. This allows dictionaries to be shared and transferred from one system to another.
- **Client–server** architecture refers to the way in which software components interact. There is a **client** process that requires some resource, and a **server** that provides the resource. In the two-tier model, the client handles the user interface and business processing logic and the server handles the database functionality. In the Web environment, the traditional two-tier model has been replaced by a three-tier model, consisting of a user interface layer (the **client**), a business logic and data processing layer (the **application server**), and a DBMS (the **database server**), distributed over different machines.
- A **Transaction Processing (TP) Monitor** is a program that controls data transfer between clients and servers in order to provide a consistent environment, particularly for online transaction processing (OLTP). The advantages include transaction routing, distributed transactions, load balancing, funneling, and increased reliability.

Review Questions

2.1 Discuss the concept of data independence and explain its importance in a database environment.

2.2 To address the issue of data independence, the ANSI-SPARC three-level architecture was proposed. Compare and contrast the three levels of this model.

2.3 What is a data model? Discuss the main types of data model.

2.4 Discuss the function and importance of conceptual modeling.

2.5 Describe the types of facility you would expect to be provided in a multi-user DBMS.

2.6 Of the facilities described in your answer to Question 2.5, which ones do you think would *not* be needed in a standalone PC DBMS? Provide justification for your answer.

2.7 Discuss the function and importance of the system catalog.

2.8 Describe the main components in a DBMS and suggest which components are responsible for each facility identified in Question 2.5.

2.9 What is meant by the term 'client–server architecture' and what are the advantages of this approach? Compare the client–server architecture with two other architectures.

2.10 Compare and contrast the two-tier client–server architecture for traditional DBMSs with the three-tier client–server architecture. Why is the latter architecture more appropriate for the Web?

2.11 What is a TP Monitor? What advantages does a TP Monitor bring to an OLTP environment?

Exercises

2.12 Analyze the DBMSs that you are currently using. Determine each system's compliance with the functions that we would expect to be provided by a DBMS. What type of language does each system provide? What type of architecture does each DBMS use? Check the accessibility and extensibility of the system catalog. Is it possible to export the system catalog to another system?

2.13 Write a program that stores names and telephone numbers in a database. Write another program that stores names and addresses in a database. Modify the programs to use external, conceptual, and internal schemas. What are the advantages and disadvantages of this modification?

2.14 Write a program that stores names and dates of birth in a database. Extend the program so that it stores the format of the data in the database: in other words, create a system catalog. Provide an interface that makes this system catalog accessible to external users.

2.15 How would you modify your program in Exercise 2.13 to conform to a client–server architecture? What would be the advantages and disadvantages of this modification?

THOMAS CONNOLLY • CAROLYN BEGG

DATABASE SYSTEMS

A Practical Approach to Design, Implementation, and Management

www.booksites.net/connbegg

ADDISON WESLEY

FOURTH EDITION