

Database System Concepts and Architecture

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package was one tightly integrated system, to the modern DBMS packages that are modular in design, with a client/server system architecture. This evolution mirrors the trends in computing, where large centralized mainframe computers are being replaced by hundreds of distributed workstations and personal computers connected via communications networks to various types of server machines—Web servers, database servers, file servers, application servers, and so on.

In a basic client/server DBMS architecture, the system functionality is distributed between two types of modules.¹ A **client module** is typically designed so that it will run on a user workstation or personal computer. Typically, application programs and user interfaces that access the database run in the client module. Hence, the client module handles user interaction and provides the user-friendly interfaces such as forms- or menu-based GUIs (graphical user interfaces). The other kind of module, called a **server module**, typically handles data storage, access, search, and other functions. We discuss client/server architectures in more detail in Section 2.5. First, we must study more basic concepts that will give us a better understanding of modern database architectures.

In this chapter we present the terminology and basic concepts that will be used throughout the book. Section 2.1 discusses data models and defines the concepts of schemas and instances, which are fundamental to the study of database systems. Then, we discuss the three-schema DBMS architecture and data independence in Section 2.2; this provides a user's perspective on what a DBMS is supposed to do. In Section 2.3 we describe the types of interfaces and languages that are typically provided by a DBMS. Section 2.4 discusses the database system software environment.

¹As we shall see in Section 2.5, there are variations on this simple *two-tier* client/server architecture.

Section 2.5 gives an overview of various types of client/server architectures. Finally, Section 2.6 presents a classification of the types of DBMS packages. Section 2.7 summarizes the chapter.

The material in Sections 2.4 through 2.6 provides more detailed concepts that may be considered as supplementary to the basic introductory material.

2.1 Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction. **Data abstraction** generally refers to the suppression of details of data organization and storage, and the highlighting of the essential features for an improved understanding of data. One of the main characteristics of the database approach is to support data abstraction so that different users can perceive data at their preferred level of detail. A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction.² By *structure of a database* we mean the data types, relationships, and constraints that apply to the data. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.

In addition to the basic operations provided by the data model, it is becoming more common to include concepts in the data model to specify the **dynamic aspect** or **behavior** of a database application. This allows the database designer to specify a set of valid user-defined operations that are allowed on the database objects.³ An example of a user-defined operation could be `COMPUTE_GPA`, which can be applied to a `STUDENT` object. On the other hand, generic operations to insert, delete, modify, or retrieve any kind of object are often included in the *basic data model operations*. Concepts to specify behavior are fundamental to object-oriented data models (see Chapter 11) but are also being incorporated in more traditional data models. For example, object-relational models (see Chapter 11) extend the basic relational model to include such concepts, among others. In the basic relational data model, there is a provision to attach behavior to the relations in the form of persistent stored modules, popularly known as stored procedures (see Chapter 13).

2.1.1 Categories of Data Models

Many data models have been proposed, which we can categorize according to the types of concepts they use to describe the database structure. **High-level** or **conceptual data models** provide concepts that are close to the way many users perceive data, whereas **low-level** or **physical data models** provide concepts that describe the details of how data is stored on the computer storage media, typically

²Sometimes the word *model* is used to denote a specific database description, or schema—for example, *the marketing data model*. We will not use this interpretation.

³The inclusion of concepts to describe behavior reflects a trend whereby database design and software design activities are increasingly being combined into a single activity. Traditionally, specifying behavior is associated with software design.

magnetic disks. Concepts provided by low-level data models are generally meant for computer specialists, not for end users. Between these two extremes is a class of **representational** (or **implementation**) **data models**,⁴ which provide concepts that may be easily understood by end users but that are not too far removed from the way data is organized in computer storage. Representational data models hide many details of data storage on disk but can be implemented on a computer system directly.

Conceptual data models use concepts such as entities, attributes, and relationships. An **entity** represents a real-world object or concept, such as an employee or a project from the miniworld that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an association among the entities, for example, a works-on relationship between an employee and a project. Chapter 7 presents the **Entity-Relationship model**—a popular high-level conceptual data model. Chapter 8 describes additional abstractions used for advanced modeling, such as generalization, specialization, and categories (union types).

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs. These include the widely used **relational data model**, as well as the so-called legacy data models—the **network** and **hierarchical models**—that have been widely used in the past. Part 2 is devoted to the relational data model, and its constraints, operations and languages.⁵ The SQL standard for relational databases is described in Chapters 4 and 5. Representational data models represent data by using record structures and hence are sometimes called **record-based data models**.

We can regard the **object data model** as an example of a new family of higher-level implementation data models that are closer to conceptual data models. A standard for object databases called the ODMG object model has been proposed by the Object Data Management Group (ODMG). We describe the general characteristics of object databases and the object model proposed standard in Chapter 11. Object data models are also frequently utilized as high-level conceptual models, particularly in the software engineering domain.

Physical data models describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths. An **access path** is a structure that makes the search for particular database records efficient. We discuss physical storage techniques and access structures in Chapters 17 and 18. An **index** is an example of an access path that allows direct access to data using an index term or a keyword. It is similar to the index at the end of this book, except that it may be organized in a linear, hierarchical (tree-structured), or some other fashion.

⁴The term *implementation data model* is not a standard term; we have introduced it to refer to the available data models in commercial database systems.

⁵A summary of the hierarchical and network data models is included in Appendices D and E. They are accessible from the book's Web site.

2.1.2 Schemas, Instances, and Database State

In any data model, it is important to distinguish between the *description* of the database and the *database itself*. The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently.⁶ Most data models have certain conventions for displaying schemas as diagrams.⁷ A displayed schema is called a **schema diagram**. Figure 2.1 shows a schema diagram for the database shown in Figure 1.2; the diagram displays the structure of each record type but not the actual instances of records. We call each object in the schema—such as STUDENT or COURSE—a **schema construct**.

A schema diagram displays only *some aspects* of a schema, such as the names of record types and data items, and some types of constraints. Other aspects are not specified in the schema diagram; for example, Figure 2.1 shows neither the data type of each data item, nor the relationships among the various files. Many types of constraints are not represented in schema diagrams. A constraint such as *students majoring in computer science must take CS1310 before the end of their sophomore year* is quite difficult to represent diagrammatically.

The actual data in a database may change quite frequently. For example, the database shown in Figure 1.2 changes every time we add a new student or enter a new grade. The data in the database at a particular moment in time is called a **database state** or **snapshot**. It is also called the *current set* of **occurrences** or **instances** in the

Figure 2.1

Schema diagram for the database in Figure 1.2.

STUDENT

Name	Student_number	Class	Major
------	----------------	-------	-------

COURSE

Course_name	Course_number	Credit_hours	Department
-------------	---------------	--------------	------------

PREREQUISITE

Course_number	Prerequisite_number
---------------	---------------------

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
--------------------	---------------	----------	------	------------

GRADE_REPORT

Student_number	Section_identifier	Grade
----------------	--------------------	-------

⁶Schema changes are usually needed as the requirements of the database applications change. Newer database systems include operations for allowing schema changes, although the schema change process is more involved than simple database updates.

⁷It is customary in database parlance to use *schemas* as the plural for *schema*, even though *schemata* is the proper plural form. The word *scheme* is also sometimes used to refer to a schema.

database. In a given database state, each schema construct has its own *current set* of instances; for example, the STUDENT construct will contain the set of individual student entities (records) as its instances. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record or change the value of a data item in a record, we change one state of the database into another state.

The distinction between database schema and database state is very important. When we **define** a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first **populated** or **loaded** with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*.⁸ The DBMS is partly responsible for ensuring that every state of the database is a **valid state**—that is, a state that satisfies the structure and constraints specified in the schema. Hence, specifying a correct schema to the DBMS is extremely important and the schema must be designed with utmost care. The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

Although, as mentioned earlier, the schema is not supposed to change frequently, it is not uncommon that changes occasionally need to be applied to the schema as the application requirements change. For example, we may decide that another data item needs to be stored for each record in a file, such as adding the Date_of_birth to the STUDENT schema in Figure 2.1. This is known as **schema evolution**. Most modern DBMSs include some operations for schema evolution that can be applied while the database is operational.

2.2 Three-Schema Architecture and Data Independence

Three of the four important characteristics of the database approach, listed in Section 1.3, are (1) use of a catalog to store the database description (schema) so as to make it self-describing, (2) insulation of programs and data (program-data and program-operation independence), and (3) support of multiple user views. In this section we specify an architecture for database systems, called the **three-schema architecture**,⁹ that was proposed to help achieve and visualize these characteristics. Then we discuss the concept of data independence further.

⁸The current state is also called the *current snapshot* of the database. It has also been called a *database instance*, but we prefer to use the term *instance* to refer to individual records.

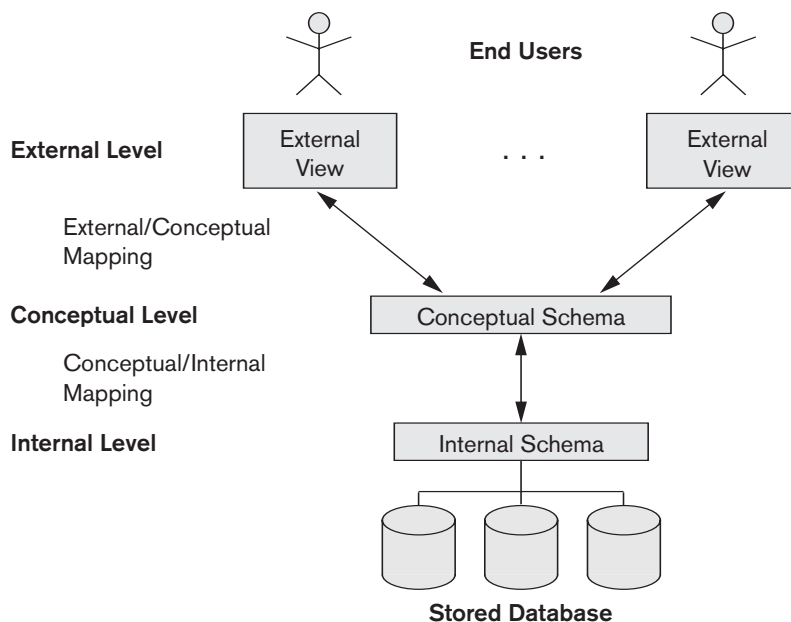
⁹This is also known as the ANSI/SPARC architecture, after the committee that proposed it (Tsichritzis and Klug 1978).

2.2.1 The Three-Schema Architecture

The goal of the three-schema architecture, illustrated in Figure 2.2, is to separate the user applications from the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented. This *implementation conceptual schema* is often based on a *conceptual schema design* in a high-level data model.
3. The **external or view level** includes a number of **external schemas** or **user views**. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. As in the previous level, each external schema is typically implemented using a representational data model, possibly based on an external schema design in a high-level data model.

Figure 2.2
The three-schema
architecture.



The three-schema architecture is a convenient tool with which the user can visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely and explicitly, but support the three-schema architecture to some extent. Some older DBMSs may include physical-level details in the conceptual schema. The three-level ANSI architecture has an important place in database technology development because it clearly separates the users' external level, the database's conceptual level, and the internal storage level for designing a database. It is very much applicable in the design of DBMSs, even today. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information (for example, a relational DBMS like Oracle uses SQL for this). Some DBMSs allow different data models to be used at the conceptual and external levels. An example is Universal Data Base (UDB), a DBMS from IBM, which uses the relational model to describe the conceptual schema, but may use an object-oriented model to describe an external schema.

Notice that the three schemas are only *descriptions* of data; the stored data that *actually* exists is at the physical level only. In a DBMS based on the three-schema architecture, each user group refers to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings**. These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

2.2.2 Data Independence

The three-schema architecture can be used to further explain the concept of **data independence**, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). In the last case, external schemas that refer only to the remaining data should not be affected. For example, the external schema of Figure 1.5(a) should not be affected by changing the `GRADE_REPORT` file (or record type) shown in Figure 1.2 into the one shown in Figure 1.6(a). Only the view definition and the mappings need to be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before.

Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. For example, providing an access path to improve retrieval speed of section records (Figure 1.2) by semester and year should not require a query such as *list all sections offered in fall 2008* to be changed, although the query would be executed more efficiently by the DBMS by utilizing the new access path.

Generally, physical data independence exists in most databases and file environments where physical details such as the exact location of data on disk, and hardware details of storage encoding, placement, compression, splitting, merging of records, and so on are hidden from the user. Applications remain unaware of these details. On the other hand, logical data independence is harder to achieve because it allows structural and constraint changes without affecting application programs—a much stricter requirement.

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog. Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.

The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

2.3 Database Languages and Interfaces

In Section 1.4 we discussed the variety of users supported by a DBMS. The DBMS must provide appropriate languages and interfaces for each category of users. In this section we discuss the types of languages and interfaces provided by a DBMS and the user categories targeted by each interface.

2.3.1 DBMS Languages

Once the design of a database is completed and a DBMS is chosen to implement the database, the first step is to specify conceptual and internal schemas for the database

and any mappings between the two. In many DBMSs where no strict separation of levels is maintained, one language, called the **data definition language (DDL)**, is used by the DBA and by database designers to define both schemas. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the **storage definition language (SDL)**, is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages. In most relational DBMSs today, there *is no specific language* that performs the role of SDL. Instead, the internal schema is specified by a combination of functions, parameters, and specifications related to storage. These permit the DBA staff to control indexing choices and mapping of data to storage. For a true three-schema architecture, we would need a third language, the **view definition language (VDL)**, to specify user views and their mappings to the conceptual schema, but in most DBMSs *the DDL is used to define both conceptual and external schemas*. In relational DBMSs, SQL is used in the role of VDL to define user or application **views** as results of predefined queries (see Chapters 4 and 5).

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a set of operations or a language called the **data manipulation language (DML)** for these purposes.

In current DBMSs, the preceding types of languages are usually *not considered distinct languages*; rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation. Storage definition is typically kept separate, since it is used for defining physical storage structures to fine-tune the performance of the database system, which is usually done by the DBA staff. A typical example of a comprehensive database language is the SQL relational database language (see Chapters 4 and 5), which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification, schema evolution, and other features. The SDL was a component in early versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

There are two main types of DMLs. A **high-level** or **nonprocedural** DML can be used on its own to specify complex database operations concisely. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so that they can be extracted by a precompiler and processed by the DBMS. A **low-level** or **procedural** DML *must* be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Therefore, it needs to use programming

language constructs, such as looping, to retrieve and process each record from a set of records. Low-level DMLs are also called **record-at-a-time** DMLs because of this property. DL/1, a DML designed for the hierarchical model, is a low-level DML that uses commands such as GET UNIQUE, GET NEXT, or GET NEXT WITHIN PARENT to navigate from record to record within a hierarchy of records in the database. High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement; therefore, they are called **set-at-a-time** or **set-oriented** DMLs. A query in a high-level DML often specifies *which* data to retrieve rather than *how* to retrieve it; therefore, such languages are also called **declarative**.

Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**.¹⁰ On the other hand, a high-level DML used in a standalone interactive manner is called a **query language**. In general, both retrieval and update commands of a high-level DML may be used interactively and are hence considered part of the query language.¹¹

Casual end users typically use a high-level query language to specify their requests, whereas programmers use the DML in its embedded form. For naive and parametric users, there usually are **user-friendly interfaces** for interacting with the database; these can also be used by casual users or others who do not want to learn the details of a high-level query language. We discuss these types of interfaces next.

2.3.2 DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following:

Menu-Based Interfaces for Web Clients or Browsing. These interfaces present the user with lists of options (called **menus**) that lead the user through the formulation of a request. Menus do away with the need to memorize the specific commands and syntax of a query language; rather, the query is composed step-by-step by picking options from a menu that is displayed by the system. Pull-down menus are a very popular technique in **Web-based user interfaces**. They are also often used in **browsing interfaces**, which allow a user to look through the contents of a database in an exploratory and unstructured manner.

Forms-Based Interfaces. A forms-based interface displays a form to each user. Users can fill out all of the **form** entries to insert new data, or they can fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have **forms specification languages**,

¹⁰In object databases, the host and data sublanguages typically form one integrated language—for example, C++ with some extensions to support database functionality. Some relational systems also provide integrated languages—for example, Oracle's PL/SQL.

¹¹According to the English meaning of the word *query*, it should really be used to describe retrievals only, not updates.

which are special languages that help programmers specify such forms. SQL*Forms is a form-based language that specifies queries using a form designed in conjunction with the relational database schema. Oracle Forms is a component of the Oracle product suite that provides an extensive set of features to design and build applications using forms. Some systems have utilities that define a form by letting the end user interactively construct a sample form on the screen.

Graphical User Interfaces. A GUI typically displays a schema to the user in diagrammatic form. The user then can specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms. Most GUIs use a **pointing device**, such as a mouse, to select certain parts of the displayed schema diagram.

Natural Language Interfaces. These interfaces accept requests written in English or some other language and attempt to *understand* them. A natural language interface usually has its own *schema*, which is similar to the database conceptual schema, as well as a dictionary of important words. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request. The capabilities of natural language interfaces have not advanced rapidly. Today, we see search engines that accept strings of natural language (like English or Spanish) words and match them with documents at specific sites (for local search engines) or Web pages on the Web at large (for engines like Google or Ask). They use predefined indexes on words and use ranking functions to retrieve and present resulting documents in a decreasing degree of match. Such “free form” textual query interfaces are not yet common in structured relational or legacy model databases, although a research area called **keyword-based querying** has emerged recently for relational databases.

Speech Input and Output. Limited use of speech as an input query and speech as an answer to a question or result of a request is becoming commonplace. Applications with limited vocabularies such as inquiries for telephone directory, flight arrival/departure, and credit card account information are allowing speech for input and output to enable customers to access this information. The speech input is detected using a library of predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech takes place.

Interfaces for Parametric Users. Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. For example, a teller is able to use single function keys to invoke routine and repetitive transactions such as account deposits or withdrawals, or balance inquiries. Systems analysts and programmers design and implement a special interface for each known class of naive users. Usually a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request. For example,

function keys in a terminal can be programmed to initiate various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

Interfaces for the DBA. Most database systems contain privileged commands that can be used only by the DBA staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

2.4 The Database System Environment

A DBMS is a complex software system. In this section we discuss the types of software components that constitute a DBMS and the types of computer system software with which the DBMS interacts.

2.4.1 DBMS Component Modules

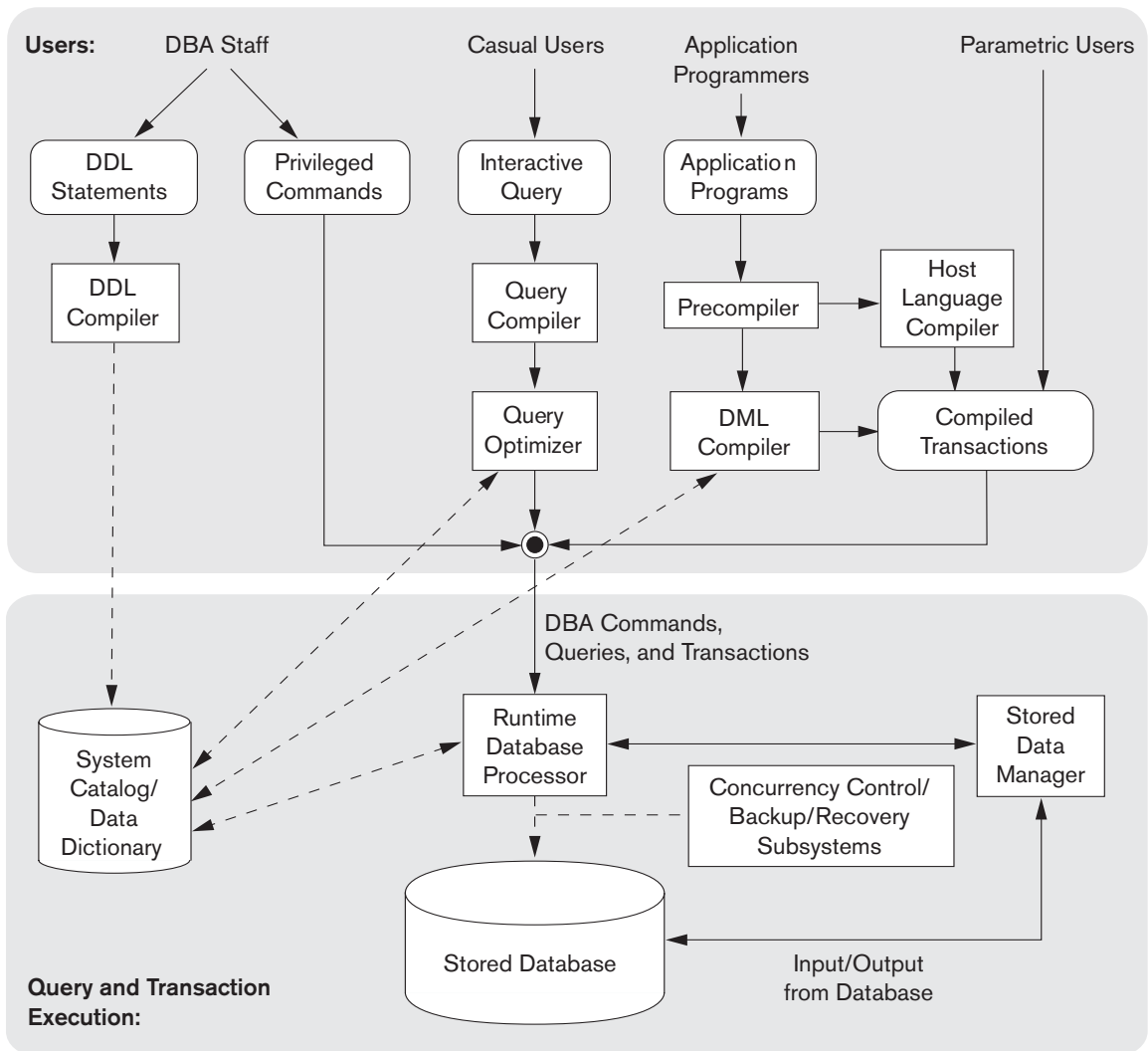
Figure 2.3 illustrates, in a simplified form, the typical DBMS components. The figure is divided into two parts. The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internals of the DBMS responsible for storage of data and processing of transactions.

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system (OS)**, which schedules disk read/write. Many DBMSs have their own **buffer management** module to schedule disk read/write, because this has a considerable effect on performance. Reducing disk read/write improves performance considerably. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.

Let us consider the top part of Figure 2.3 first. It shows interfaces for the DBA staff, casual users who work with interactive interfaces to formulate queries, application programmers who create programs using some host programming languages, and parametric users who do data entry work by supplying parameters to predefined transactions. The DBA staff works on defining the database and tuning it by making changes to its definition using the DDL and other privileged commands.

The DDL compiler processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints. In addition, the catalog stores many other types of information that are needed by the DBMS modules, which can then look up the catalog information as needed.

Casual users and persons with occasional need for information from the database interact using some form of interface, which we call the **interactive query** interface in Figure 2.3. We have not explicitly shown any menu-based or form-based interaction that may be used to generate the interactive query automatically. These queries are parsed and validated for correctness of the query syntax, the names of files and

**Figure 2.3**

Component modules of a DBMS and their interactions.

data elements, and so on by a **query compiler** that compiles them into an internal form. This internal query is subjected to query optimization (discussed in Chapters 19 and 20). Among other things, the **query optimizer** is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of correct algorithms and indexes during execution. It consults the system catalog for statistical and other physical information about the stored data and generates executable code that performs the necessary operations for the query and makes calls on the runtime processor.

Application programmers write programs in host languages such as Java, C, or C++ that are submitted to a precompiler. The **precompiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the DML compiler for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor. Canned transactions are executed repeatedly by parametric users, who simply supply the parameters to the transactions. Each execution is considered to be a separate transaction. An example is a bank withdrawal transaction where the account number and the amount may be supplied as parameters.

In the lower part of Figure 2.3, the **runtime database processor** executes (1) the privileged commands, (2) the executable query plans, and (3) the canned transactions with runtime parameters. It works with the **system catalog** and may update it with statistics. It also works with the **stored data manager**, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory. The runtime database processor handles other aspects of data transfer, such as management of buffers in the main memory. Some DBMSs have their own buffer management module while others depend on the OS for buffer management. We have shown **concurrency control** and **backup and recovery systems** separately as a module in this figure. They are integrated into the working of the runtime database processor for purposes of transaction management.

It is now common to have the **client program** that accesses the DBMS running on a separate computer from the computer on which the database resides. The former is called the **client computer** running a DBMS client software and the latter is called the **database server**. In some cases, the client accesses a middle computer, called the **application server**, which in turn accesses the database server. We elaborate on this topic in Section 2.5.

Figure 2.3 is not meant to describe a specific DBMS; rather, it illustrates typical DBMS modules. The DBMS interacts with the operating system when disk accesses—to the database or to the catalog—are needed. If the computer system is shared by many users, the OS will schedule DBMS disk access requests and DBMS processing along with other processes. On the other hand, if the computer system is mainly dedicated to running the database server, the DBMS will control main memory buffering of disk pages. The DBMS also interfaces with compilers for general-purpose host programming languages, and with application servers and client programs running on separate machines through the system network interface.

2.4.2 Database System Utilities

In addition to possessing the software modules just described, most DBMSs have **database utilities** that help the DBA manage the database system. Common utilities have the following types of functions:

- **Loading.** A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) for-

mat of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database. With the proliferation of DBMSs, transferring data from one DBMS to another is becoming common in many organizations. Some vendors are offering products that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas). Such tools are also called **conversion tools**. For the hierarchical DBMS called IMS (IBM) and for many network DBMSs including IDMS (Computer Associates), SUPRA (Cincom), and IMAGE (HP), the vendors or third-party companies are making a variety of conversion tools available (e.g., Cincom's SUPRA Server SQL) to transform data into the relational model.

- **Backup.** A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium. The backup copy can be used to restore the database in case of catastrophic disk failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex, but saves storage space.
- **Database storage reorganization.** This utility can be used to reorganize a set of database files into different file organizations, and create new access paths to improve performance.
- **Performance monitoring.** Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files or whether to add or drop indexes to improve performance.

Other utilities may be available for sorting files, handling data compression, monitoring access by users, interfacing with the network, and performing other functions.

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and the DBMS. CASE tools¹² are used in the design phase of database systems. Another tool that can be quite useful in large organizations is an expanded **data dictionary** (or **data repository**) **system**. In addition to storing catalog information about schemas and constraints, the data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an **information repository**. This information can be accessed *directly* by users or the DBA when needed. A data dictionary utility is similar to the DBMS catalog, but it includes a wider variety of information and is accessed mainly by users rather than by the DBMS software.

¹²Although CASE stands for computer-aided software engineering, many CASE tools are used primarily for database design.

Application development environments, such as PowerBuilder (Sybase) or JBuilder (Borland), have been quite popular. These systems provide an environment for developing database applications and include facilities that help in many facets of database systems, including database design, GUI development, querying and updating, and application program development.

The DBMS also needs to interface with **communications software**, whose function is to allow users at locations remote from the database system site to access the database through computer terminals, workstations, or personal computers. These are connected to the database site through data communications hardware such as Internet routers, phone lines, long-haul networks, local networks, or satellite communication devices. Many commercial database systems have communication packages that work with the DBMS. The integrated DBMS and data communications system is called a **DB/DC** system. In addition, some distributed DBMSs are physically distributed over multiple machines. In this case, communications networks are needed to connect the machines. These are often **local area networks (LANs)**, but they can also be other types of networks.

2.5 Centralized and Client/Server Architectures for DBMSs

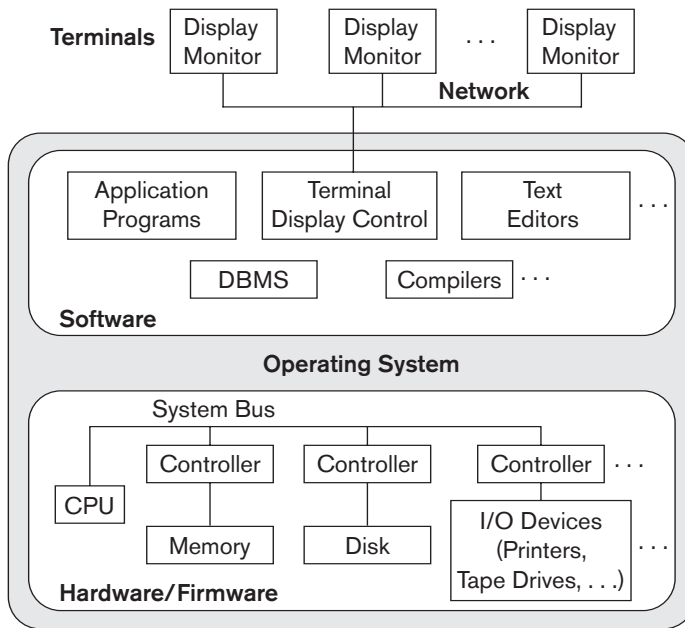
2.5.1 Centralized DBMSs Architecture

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Earlier architectures used mainframe computers to provide the main processing for all system functions, including user application programs and user interface programs, as well as all the DBMS functionality. The reason was that most users accessed such systems via computer terminals that did not have processing power and only provided display capabilities. Therefore, all processing was performed remotely on the computer system, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks.

As prices of hardware declined, most users replaced their terminals with PCs and workstations. At first, database systems used these computers similarly to how they had used display terminals, so that the DBMS itself was still a **centralized** DBMS in which all the DBMS functionality, application program execution, and user interface processing were carried out on one machine. Figure 2.4 illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

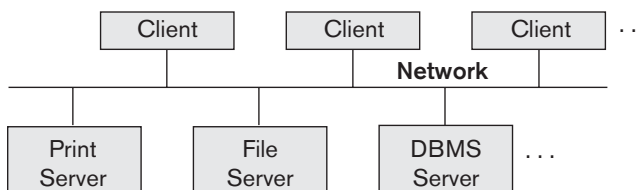
2.5.2 Basic Client/Server Architectures

First, we discuss client/server architecture in general, then we see how it is applied to DBMSs. The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, data-

**Figure 2.4**

A physical centralized architecture.

base servers, Web servers, e-mail servers, and other software and equipment are connected via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to various printers; all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to other software packages, with specialized programs—such as a CAD (computer-aided design) package—being stored on specific server machines and being made accessible to multiple clients. Figure 2.5 illustrates client/server architecture at the logical level; Figure 2.6 is a simplified diagram that shows the physical architecture. Some machines would be client sites only (for example, diskless workstations or workstations/PCs with disks that have only client software installed).

**Figure 2.5**

Logical two-tier client/server architecture.

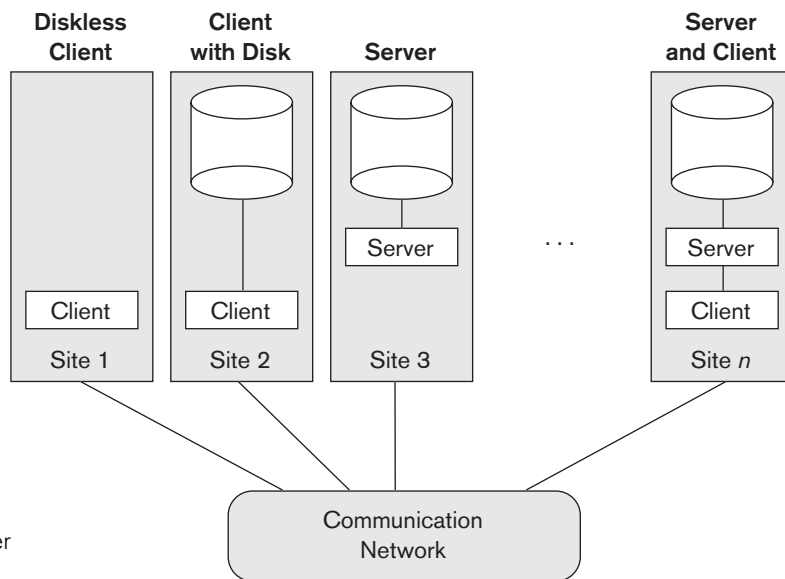


Figure 2.6
Physical two-tier client/server
architecture.

Other machines would be dedicated servers, and others would have both client and server functionality.

The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via LANs and other types of computer networks. A **client** in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality—such as database access—that does not exist at that machine, it connects to a server that provides the needed functionality. A **server** is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access. In general, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in Figure 2.6. However, it is more common that client and server software usually run on separate machines. Two main types of basic DBMS architectures were created on this underlying client/server framework: **two-tier** and **three-tier**.¹³ We discuss them next.

2.5.3 Two-Tier Client/Server Architectures for DBMSs

In relational database management systems (RDBMSs), many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL (see Chapters 4 and 5) provided a standard language for RDBMSs, this created a logical dividing point

¹³There are many other variations of client/server architectures. We discuss the two most basic ones here.

between client and server. Hence, the query and transaction functionality related to SQL processing remained on the server side. In such an architecture, the server is often called a **query server** or **transaction server** because it provides these two functionalities. In an RDBMS, the server is also often called an **SQL server**.

The user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database Connectivity (ODBC)** provides an **application programming interface (API)**, which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems. A client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites. Any query results are sent back to the client program, which can process and display the results as needed. A related standard for the Java programming language, called **JDBC**, has also been defined. This allows Java client programs to access one or more DBMSs through a standard interface.

The different approach to two-tier client/server architecture was taken by some object-oriented DBMSs, where the software modules of the DBMS were divided between client and server in a more integrated way. For example, the **server level** may include the part of the DBMS software responsible for handling data storage on disk pages, local concurrency control and recovery, buffering and caching of disk pages, and other such functions. Meanwhile, the **client level** may handle the user interface; data dictionary functions; DBMS interactions with programming language compilers; global query optimization, concurrency control, and recovery across multiple servers; structuring of complex objects from the data in the buffers; and other such functions. In this approach, the client/server interaction is more tightly coupled and is done internally by the DBMS modules—some of which reside on the client and some on the server—rather than by the users/programmers. The exact division of functionality can vary from system to system. In such a client/server architecture, the server has been called a **data server** because it provides data in disk pages to the client. This data can then be structured into objects for the client programs by the client-side DBMS software.

The architectures described here are called **two-tier architectures** because the software components are distributed over two systems: client and server. The advantages of this architecture are its simplicity and seamless compatibility with existing systems. The emergence of the Web changed the roles of clients and servers, leading to the three-tier architecture.

2.5.4 Three-Tier and n-Tier Architectures for Web Applications

Many Web applications use an architecture called the **three-tier architecture**, which adds an intermediate layer between the client and the database server, as illustrated in Figure 2.7(a).

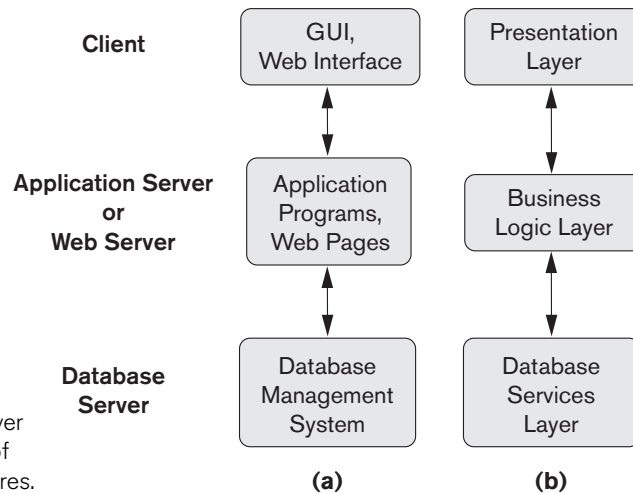


Figure 2.7
Logical three-tier client/server
architecture, with a couple of
commonly used nomenclatures.

This intermediate layer or **middle tier** is called the **application server** or the **Web server**, depending on the application. This server plays an intermediary role by running application programs and storing business rules (procedures or constraints) that are used to access data from the database server. It can also improve database security by checking a client's credentials before forwarding a request to the database server. Clients contain GUI interfaces and some additional application-specific business rules. The intermediate server accepts requests from the client, processes the request and sends database queries and commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to users in GUI format. Thus, the *user interface*, *application rules*, and *data access* act as the three tiers. Figure 2.7(b) shows another architecture used by database and other application package vendors. The presentation layer displays information to the user and allows data entry. The business logic layer handles intermediate rules and constraints before data is passed up to the user or down to the DBMS. The bottom layer includes all data management services. The middle layer can also act as a Web server, which retrieves query results from the database server and formats them into dynamic Web pages that are viewed by the Web browser at the client side.

Other architectures have also been proposed. It is possible to divide the layers between the user and the stored data further into finer components, thereby giving rise to n -tier architectures, where n may be four or five tiers. Typically, the business logic layer is divided into multiple layers. Besides distributing programming and data throughout a network, n -tier applications afford the advantage that any one tier can run on an appropriate processor or operating system platform and can be handled independently. Vendors of ERP (enterprise resource planning) and CRM (customer relationship management) packages often use a *middleware layer*, which accounts for the front-end modules (clients) communicating with a number of back-end databases (servers).

Advances in encryption and decryption technology make it safer to transfer sensitive data from server to client in encrypted form, where it will be decrypted. The latter can be done by the hardware or by advanced software. This technology gives higher levels of data security, but the network security issues remain a major concern. Various technologies for data compression also help to transfer large amounts of data from servers to clients over wired and wireless networks.

2.6 Classification of Database Management Systems

Several criteria are normally used to classify DBMSs. The first is the **data model** on which the DBMS is based. The main data model used in many current commercial DBMSs is the **relational data model**. The **object data model** has been implemented in some commercial systems but has not had widespread use. Many legacy applications still run on database systems based on the **hierarchical** and **network data models**. Examples of hierarchical DBMSs include IMS (IBM) and some other systems like System 2K (SAS Inc.) and TDMS. IMS is still used at governmental and industrial installations, including hospitals and banks, although many of its users have converted to relational systems. The network data model was used by many vendors and the resulting products like IDMS (Cullinet—now Computer Associates), DMS 1100 (Univac—now Unisys), IMAGE (Hewlett-Packard), VAX-DBMS (Digital—then Compaq and now HP), and SUPRA (Cincom) still have a following and their user groups have their own active organizations. If we add IBM's popular VSAM file system to these, we can easily say that a reasonable percentage of worldwide-computerized data is still in these so-called **legacy database systems**.

The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs called **object-relational DBMSs**. We can categorize DBMSs based on the data model: relational, object, object-relational, hierarchical, network, and other.

More recently, some experimental DBMSs are based on the XML (eXtended Markup Language) model, which is a tree-structured (hierarchical) data model. These have been called **native XML DBMSs**. Several commercial relational DBMSs have added XML interfaces and storage to their products.

The second criterion used to classify DBMSs is the **number of users** supported by the system. **Single-user systems** support only one user at a time and are mostly used with PCs. **Multiuser systems**, which include the majority of DBMSs, support concurrent multiple users.

The third criterion is the **number of sites** over which the database is distributed. A DBMS is **centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database reside totally at a single computer site. A **distributed** DBMS (DDBMS) can have the actual database and DBMS software distributed over many sites, connected by a computer network. **Homogeneous** DDBMSs use the same DBMS software at all the sites, whereas

heterogeneous DDBMSs can use different DBMS software at each site. It is also possible to develop **middleware software** to access several autonomous preexisting databases stored under heterogeneous DBMSs. This leads to a **federated** DBMS (or **multidatabase system**), in which the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DDBMSs use client-server architecture, as we described in Section 2.5.

The fourth criterion is cost. It is difficult to propose a classification of DBMSs based on cost. Today we have open source (free) DBMS products like MySQL and PostgreSQL that are supported by third-party vendors with additional services. The main RDBMS products are available as free examination 30-day copy versions as well as personal versions, which may cost under \$100 and allow a fair amount of functionality. The giant systems are being sold in modular form with components to handle distribution, replication, parallel processing, mobile capability, and so on, and with a large number of parameters that must be defined for the configuration. Furthermore, they are sold in the form of licenses—site licenses allow unlimited use of the database system with any number of copies running at the customer site. Another type of license limits the number of concurrent users or the number of user seats at a location. Standalone single user versions of some systems like Microsoft Access are sold per copy or included in the overall configuration of a desktop or laptop. In addition, data warehousing and mining features, as well as support for additional data types, are made available at extra cost. It is possible to pay millions of dollars for the installation and maintenance of large database systems annually.

We can also classify a DBMS on the basis of the **types of access path** options for storing files. One well-known family of DBMSs is based on inverted file structures. Finally, a DBMS can be **general purpose** or **special purpose**. When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes. Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs. These fall into the category of **online transaction processing (OLTP)** systems, which must support a large number of concurrent transactions without imposing excessive delays.

Let us briefly elaborate on the main criterion for classifying DBMSs: the data model. The basic **relational data model** represents a database as a collection of tables, where each table can be stored as a separate file. The database in Figure 1.2 resembles a relational representation. Most relational databases use the high-level query language called SQL and support a limited form of user views. We discuss the relational model and its languages and operations in Chapters 3 through 6, and techniques for programming relational applications in Chapters 13 and 14.

The **object data model** defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a **class**, and classes are organized into **hierarchies** (or **acyclic graphs**). The operations of each class are specified in terms of predefined procedures called **methods**. Relational DBMSs have been extending their models to incorporate object database

concepts and other capabilities; these systems are referred to as **object-relational** or **extended relational systems**. We discuss object databases and object-relational systems in Chapter 11.

The **XML model** has emerged as a standard for exchanging data over the Web, and has been used as a basis for implementing several prototype native XML systems. XML uses hierarchical tree structures. It combines database concepts with concepts from document representation models. Data is represented as elements; with the use of tags, data can be nested to create complex hierarchical structures. This model conceptually resembles the object model but uses different terminology. XML capabilities have been added to many commercial DBMS products. We present an overview of XML in Chapter 12.

Two older, historically important data models, now known as **legacy data models**, are the network and hierarchical models. The **network model** represents data as record types and also represents a limited type of 1:N relationship, called a **set type**. A 1:N, or one-to-many, relationship relates one instance of a record to many record instances using some pointer linking mechanism in these models. Figure 2.8 shows a network schema diagram for the database of Figure 2.1, where record types are shown as rectangles and set types are shown as labeled directed arrows.

The network model, also known as the CODASYL DBTG model,¹⁴ has an associated record-at-a-time language that must be embedded in a host programming language. The network DML was proposed in the 1971 Database Task Group (DBTG) Report as an extension of the COBOL language. It provides commands for locating records directly (e.g., FIND ANY <record-type> USING <field-list>, or FIND DUPLICATE <record-type> USING <field-list>). It has commands to support traversals within set-types (e.g., GET OWNER, GET {FIRST, NEXT, LAST} MEMBER WITHIN <set-type> WHERE <condition>). It also has commands to store new data

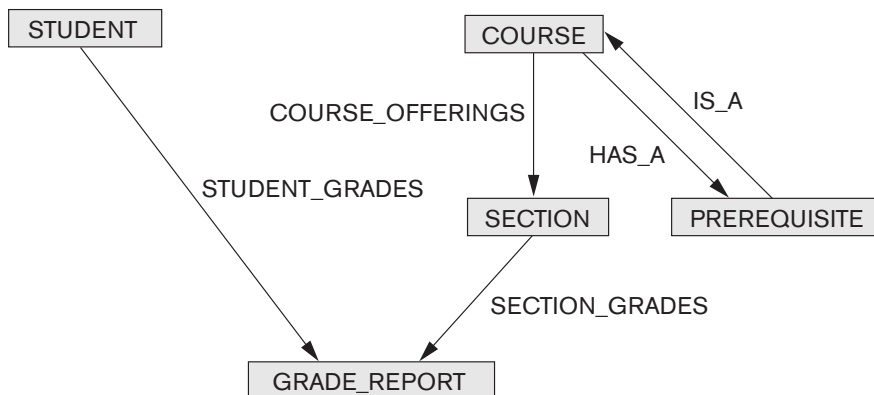


Figure 2.8

The schema of Figure 2.1 in network model notation.

¹⁴CODASYL DBTG stands for Conference on Data Systems Languages Database Task Group, which is the committee that specified the network model and its language.

(e.g., STORE <record-type>) and to make it part of a set type (e.g., CONNECT <record-type> TO <set-type>). The language also handles many additional considerations, such as the currency of record types and set types, which are defined by the current position of the navigation process within the database. It is prominently used by IDMS, IMAGE, and SUPRA DBMSs today.

The **hierarchical model** represents data as hierarchical tree structures. Each hierarchy represents a number of related records. There is no standard language for the hierarchical model. A popular hierarchical DML is DL/1 of the IMS system. It dominated the DBMS market for over 20 years between 1965 and 1985 and is still a widely used DBMS worldwide, holding a large percentage of data in governmental, health care, and banking and insurance databases. Its DML, called DL/1, was a de facto industry standard for a long time. DL/1 has commands to locate a record (e.g., GET { UNIQUE, NEXT} <record-type> WHERE <condition>). It has navigational facilities to navigate within hierarchies (e.g., GET NEXT WITHIN PARENT or GET {FIRST, NEXT} PATH <hierarchical-path-specification> WHERE <condition>). It has appropriate facilities to store and update records (e.g., INSERT <record-type>, REPLACE <record-type>). Currency issues during navigation are also handled with additional features in the language.¹⁵

2.7 Summary

In this chapter we introduced the main concepts used in database systems. We defined a data model and we distinguished three main categories:

- High-level or conceptual data models (based on entities and relationships)
- Low-level or physical data models
- Representational or implementation data models (record-based, object-oriented)

We distinguished the schema, or description of a database, from the database itself. The schema does not change very often, whereas the database state changes every time data is inserted, deleted, or modified. Then we described the three-schema DBMS architecture, which allows three schema levels:

- An internal schema describes the physical storage structure of the database.
- A conceptual schema is a high-level description of the whole database.
- External schemas describe the views of different user groups.

A DBMS that cleanly separates the three levels must have mappings between the schemas to transform requests and query results from one level to the next. Most DBMSs do not separate the three levels completely. We used the three-schema architecture to define the concepts of logical and physical data independence.

¹⁵The full chapters on the network and hierarchical models from the second edition of this book are available from this book's Companion Website at <http://www.aw.com/elmasri>.

Then we discussed the main types of languages and interfaces that DBMSs support. A data definition language (DDL) is used to define the database conceptual schema. In most DBMSs, the DDL also defines user views and, sometimes, storage structures; in other DBMSs, separate languages or functions exist for specifying storage structures. This distinction is fading away in today's relational implementations, with SQL serving as a catchall language to perform multiple roles, including view definition. The storage definition part (SDL) was included in SQL's early versions, but is now typically implemented as special commands for the DBA in relational DBMSs. The DBMS compiles all schema definitions and stores their descriptions in the DBMS catalog.

A data manipulation language (DML) is used for specifying database retrievals and updates. DMLs can be high level (set-oriented, nonprocedural) or low level (record-oriented, procedural). A high-level DML can be embedded in a host programming language, or it can be used as a standalone language; in the latter case it is often called a query language.

We discussed different types of interfaces provided by DBMSs, and the types of DBMS users with which each interface is associated. Then we discussed the database system environment, typical DBMS software modules, and DBMS utilities for helping users and the DBA staff perform their tasks. We continued with an overview of the two-tier and three-tier architectures for database applications, progressively moving toward n -tier, which are now common in many applications, particularly Web database applications.

Finally, we classified DBMSs according to several criteria: data model, number of users, number of sites, types of access paths, and cost. We discussed the availability of DBMSs and additional modules—from no cost in the form of open source software, to configurations that annually cost millions to maintain. We also pointed out the variety of licensing arrangements for DBMS and related products. The main classification of DBMSs is based on the data model. We briefly discussed the main data models used in current commercial DBMSs.

Review Questions

- 2.1. Define the following terms: *data model*, *database schema*, *database state*, *internal schema*, *conceptual schema*, *external schema*, *data independence*, *DDL*, *DML*, *SDL*, *VDL*, *query language*, *host language*, *data sublanguage*, *database utility*, *catalog*, *client/server architecture*, *three-tier architecture*, and *n-tier architecture*.
- 2.2. Discuss the main categories of data models. What are the basic differences between the relational model, the object model, and the XML model?
- 2.3. What is the difference between a database schema and a database state?
- 2.4. Describe the three-schema architecture. Why do we need mappings between schema levels? How do different schema definition languages support this architecture?

- 2.5. What is the difference between logical data independence and physical data independence? Which one is harder to achieve? Why?
- 2.6. What is the difference between procedural and nonprocedural DMLs?
- 2.7. Discuss the different types of user-friendly interfaces and the types of users who typically use each.
- 2.8. With what other computer system software does a DBMS interact?
- 2.9. What is the difference between the two-tier and three-tier client/server architectures?
- 2.10. Discuss some types of database utilities and tools and their functions.
- 2.11. What is the additional functionality incorporated in n -tier architecture ($n > 3$)?

Exercises

- 2.12. Think of different users for the database shown in Figure 1.2. What types of applications would each user need? To which user category would each belong, and what type of interface would each need?
- 2.13. Choose a database application with which you are familiar. Design a schema and show a sample database for that application, using the notation of Figures 1.2 and 2.1. What types of additional information and constraints would you like to represent in the schema? Think of several users of your database, and design a view for each.
- 2.14. If you were designing a Web-based system to make airline reservations and sell airline tickets, which DBMS architecture would you choose from Section 2.5? Why? Why would the other architectures not be a good choice?
- 2.15. Consider Figure 2.1. In addition to constraints relating the values of columns in one table to columns in another table, there are also constraints that impose restrictions on values in a column or a combination of columns within a table. One such constraint dictates that a column or a group of columns must be unique across all rows in the table. For example, in the STUDENT table, the Student_number column must be unique (to prevent two different students from having the same Student_number). Identify the column or the group of columns in the other tables that must be unique across all rows in the table.

Selected Bibliography

Many database textbooks, including Date (2004), Silberschatz et al. (2006), Ramakrishnan and Gehrke (2003), Garcia-Molina et al. (2000, 2009), and Abiteboul et al. (1995), provide a discussion of the various database concepts presented here. Tsichritzis and Lochovsky (1982) is an early textbook on data models. Tsichritzis and Klug (1978) and Jardine (1977) present the three-schema architecture, which was first suggested in the DBTG CODASYL report (1971) and later in an American National Standards Institute (ANSI) report (1975). An in-depth analysis of the relational data model and some of its possible extensions is given in Codd (1990). The proposed standard for object-oriented databases is described in Cattell et al. (2000). Many documents describing XML are available on the Web, such as XML (2005).

Examples of database utilities are the ETI Connect, Analyze and Transform tools (<http://www.eti.com>) and the database administration tool, DBArtisan, from Embarcadero Technologies (<http://www.embarcadero.com>).

This page intentionally left blank

part 2

The Relational Data Model and SQL

This page intentionally left blank

The Relational Data Model and Relational Database Constraints

This chapter opens Part 2 of the book, which covers relational databases. The relational data model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper (Codd 1970), and it attracted immediate attention due to its simplicity and mathematical foundation. The model uses the concept of a *mathematical relation*—which looks somewhat like a table of values—as its basic building block, and has its theoretical basis in set theory and first-order predicate logic. In this chapter we discuss the basic characteristics of the model and its constraints.

The first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system on the MVS operating system by IBM and the Oracle DBMS. Since then, the model has been implemented in a large number of commercial systems. Current popular relational DBMSs (RDBMSs) include DB2 and Informix Dynamic Server (from IBM), Oracle and Rdb (from Oracle), Sybase DBMS (from Sybase) and SQLServer and Access (from Microsoft). In addition, several open source systems, such as MySQL and PostgreSQL, are available.

Because of the importance of the relational model, all of Part 2 is devoted to this model and some of the languages associated with it. In Chapters 4 and 5, we describe the SQL query language, which is the *standard* for commercial relational DBMSs. Chapter 6 covers the operations of the relational algebra and introduces the relational calculus—these are two formal languages associated with the relational model. The relational calculus is considered to be the basis for the SQL language, and the relational algebra is used in the internals of many database implementations for query processing and optimization (see Part 8 of the book).

Other aspects of the relational model are presented in subsequent parts of the book. Chapter 9 relates the relational model data structures to the constructs of the ER and EER models (presented in Chapters 7 and 8), and presents algorithms for designing a relational database schema by mapping a conceptual schema in the ER or EER model into a relational representation. These mappings are incorporated into many database design and CASE¹ tools. Chapters 13 and 14 in Part 5 discuss the programming techniques used to access database systems and the notion of connecting to relational databases via ODBC and JDBC standard protocols. We also introduce the topic of Web database programming in Chapter 14. Chapters 15 and 16 in Part 6 present another aspect of the relational model, namely the formal constraints of functional and multivalued dependencies; these dependencies are used to develop a relational database design theory based on the concept known as *normalization*.

Data models that preceded the relational model include the hierarchical and network models. They were proposed in the 1960s and were implemented in early DBMSs during the late 1960s and early 1970s. Because of their historical importance and the existing user base for these DBMSs, we have included a summary of the highlights of these models in Appendices D and E, which are available on this book's Companion Website at <http://www.aw.com/elmasri>. These models and systems are now referred to as *legacy database systems*.

In this chapter, we concentrate on describing the basic principles of the relational model of data. We begin by defining the modeling concepts and notation of the relational model in Section 3.1. Section 3.2 is devoted to a discussion of relational constraints that are considered an important part of the relational model and are automatically enforced in most relational DBMSs. Section 3.3 defines the update operations of the relational model, discusses how violations of integrity constraints are handled, and introduces the concept of a transaction. Section 3.4 summarizes the chapter.

3.1 Relational Model Concepts

The relational model represents the database as a collection of *relations*. Informally, each relation resembles a table of values or, to some extent, a *flat* file of records. It is called a **flat file** because each record has a simple linear or *flat* structure. For example, the database of files that was shown in Figure 1.2 is similar to the basic relational model representation. However, there are important differences between relations and files, as we shall soon see.

When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row. For example, the first table of Figure 1.2 is called **STUDENT** because each row represents facts about a particular

¹CASE stands for computer-aided software engineering.

student entity. The column names—Name, Student_number, Class, and Major—specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation*. The data type describing the types of values that can appear in each column is represented by a *domain* of possible values. We now define these terms—*domain*, *tuple*, *attribute*, and *relation*—formally.

3.1 Domains, Attributes, Tuples, and Relations

A **domain** D is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values. Some examples of domains follow:

- **Usa_phone_numbers**. The set of ten-digit phone numbers valid in the United States.
- **Local_phone_numbers**. The set of seven-digit phone numbers valid within a particular area code in the United States. The use of local phone numbers is quickly becoming obsolete, being replaced by standard ten-digit numbers.
- **Social_security_numbers**. The set of valid nine-digit Social Security numbers. (This is a unique identifier assigned to each person in the United States for employment, tax, and benefits purposes.)
- **Names**. The set of character strings that represent names of persons.
- **Grade_point_averages**. Possible values of computed grade point averages; each must be a real (floating-point) number between 0 and 4.
- **Employee_ages**. Possible ages of employees in a company; each must be an integer value between 15 and 80.
- **Academic_department_names**. The set of academic department names in a university, such as Computer Science, Economics, and Physics.
- **Academic_department_codes**. The set of academic department codes, such as 'CS', 'ECON', and 'PHYS'.

The preceding are called *logical* definitions of domains. A **data type** or **format** is also specified for each domain. For example, the data type for the domain **Usa_phone_numbers** can be declared as a character string of the form $(ddd)ddd-dddd$, where each d is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for **Employee_ages** is an integer number between 15 and 80. For **Academic_department_names**, the data type is the set of all character strings that represent valid department names. A domain is thus given a name, data type, and format. Additional information for interpreting the values of a domain can also be given; for example, a numeric domain such as **Person_weights** should have the units of measurement, such as pounds or kilograms.

A **relation schema**² R , denoted by $R(A_1, A_2, \dots, A_n)$, is made up of a relation name R and a list of attributes, A_1, A_2, \dots, A_n . Each **attribute** A_i is the name of a role played by some domain D in the relation schema R . D is called the **domain** of A_i and is denoted by $\text{dom}(A_i)$. A relation schema is used to *describe* a relation; R is called the **name** of this relation. The **degree** (or **arity**) of a relation is the number of attributes n of its relation schema.

A relation of degree seven, which stores information about university students, would contain seven attributes describing each student. as follows:

STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)

Using the data type of each attribute, the definition is sometimes written as:

STUDENT(Name: string, Ssn: string, Home_phone: string, Address: string,
Office_phone: string, Age: integer, Gpa: real)

For this relation schema, STUDENT is the name of the relation, which has seven attributes. In the preceding definition, we showed assignment of generic types such as string or integer to the attributes. More precisely, we can specify the following previously defined domains for some of the attributes of the STUDENT relation: $\text{dom}(\text{Name}) = \text{Names}$; $\text{dom}(\text{Ssn}) = \text{Social_security_numbers}$; $\text{dom}(\text{HomePhone}) = \text{USA_phone_numbers}$ ³, $\text{dom}(\text{Office_phone}) = \text{USA_phone_numbers}$, and $\text{dom}(\text{Gpa}) = \text{Grade_point_averages}$. It is also possible to refer to attributes of a relation schema by their position within the relation; thus, the second attribute of the STUDENT relation is Ssn, whereas the fourth attribute is Address.

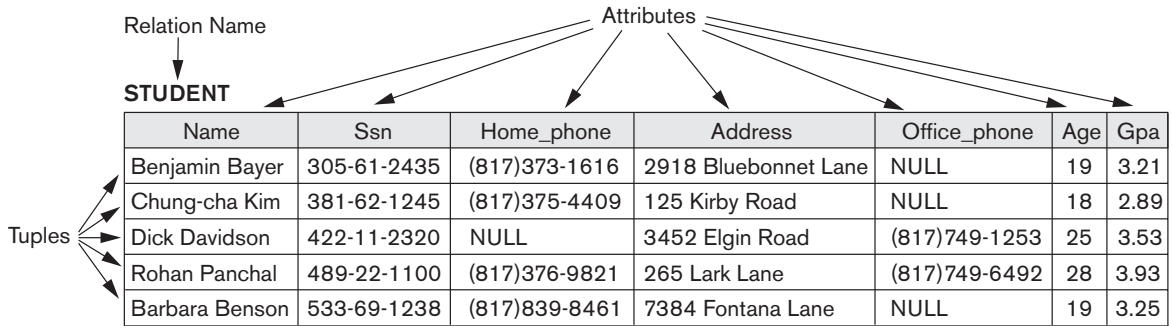
A **relation** (or **relation state**)⁴ r of the relation schema $R(A_1, A_2, \dots, A_n)$, also denoted by $r(R)$, is a set of n -tuples $r = \{t_1, t_2, \dots, t_m\}$. Each **n -tuple** t is an ordered list of n values $t = \langle v_1, v_2, \dots, v_n \rangle$, where each value v_i , $1 \leq i \leq n$, is an element of $\text{dom}(A_i)$ or is a special NULL value. (NULL values are discussed further below and in Section 3.1.2.) The i^{th} value in tuple t , which corresponds to the attribute A_i , is referred to as $t[A_i]$ or $t.A_i$ (or $t[i]$ if we use the positional notation). The terms **relation intension** for the schema R and **relation extension** for a relation state $r(R)$ are also commonly used.

Figure 3.1 shows an example of a STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity (or object). We display the relation as a table, where each tuple is shown as a *row* and each attribute corresponds to a *column header* indicating a role or interpretation of the values in that column. *NULL values* represent attributes whose values are unknown or do not exist for some individual STUDENT tuple.

²A relation schema is sometimes called a **relation scheme**.

³With the large increase in phone numbers caused by the proliferation of mobile phones, most metropolitan areas in the U.S. now have multiple area codes, so seven-digit local dialing has been discontinued in most areas. We changed this domain to *Usa_phone_numbers* instead of *Local_phone_numbers* which would be a more general choice. This illustrates how database requirements can change over time.

⁴This has also been called a **relation instance**. We will not use this term because *instance* is also used to refer to a single tuple or row.

**Figure 3.1**

The attributes and tuples of a relation STUDENT.

The earlier definition of a relation can be *restated* more formally using set theory concepts as follows. A relation (or relation state) $r(R)$ is a **mathematical relation** of degree n on the domains $\text{dom}(A_1)$, $\text{dom}(A_2)$, ..., $\text{dom}(A_n)$, which is a **subset** of the **Cartesian product** (denoted by \times) of the domains that define R :

$$r(R) \subseteq (\text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n))$$

The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the total number of values, or **cardinality**, in a domain D by $|D|$ (assuming that all domains are finite), the total number of tuples in the Cartesian product is

$$|\text{dom}(A_1)| \times |\text{dom}(A_2)| \times \dots \times |\text{dom}(A_n)|$$

This product of cardinalities of all domains represents the total number of possible instances or tuples that can ever exist in any relation state $r(R)$. Of all these possible combinations, a relation state at a given time—the **current relation state**—reflects only the valid tuples that represent a particular state of the real world. In general, as the state of the real world changes, so does the relation state, by being transformed into another relation state. However, the schema R is relatively static and changes *very* infrequently—for example, as a result of adding an attribute to represent new information that was not originally stored in the relation.

It is possible for several attributes to *have the same domain*. The attribute names indicate different **roles**, or interpretations, for the domain. For example, in the STUDENT relation, the same domain `USA_phone_numbers` plays the role of `Home_phone`, referring to the *home phone of a student*, and the role of `Office_phone`, referring to the *office phone of the student*. A third possible attribute (not shown) with the same domain could be `Mobile_phone`.

3.1.2 Characteristics of Relations

The earlier definition of relations implies certain characteristics that make a relation different from a file or a table. We now discuss some of these characteristics.

Ordering of Tuples in a Relation. A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. In other words, a relation is not sensitive to the ordering of tuples. However, in a file, records are physically stored on disk (or in memory), so there always is an order among the records. This ordering indicates first, second, *i*th, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.

Tuple ordering is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level. Many tuple orders can be specified on the same relation. For example, tuples in the STUDENT relation in Figure 3.1 could be ordered by values of Name, Ssn, Age, or some other attribute. The definition of a relation does not specify any order: There is *no preference* for one ordering over another. Hence, the relation displayed in Figure 3.2 is considered *identical* to the one shown in Figure 3.1. When a relation is implemented as a file or displayed as a table, a particular ordering may be specified on the records of the file or the rows of the table.

Ordering of Values within a Tuple and an Alternative Definition of a Relation. According to the preceding definition of a relation, an *n*-tuple is an *ordered list* of *n* values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important. However, at a more abstract level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained.

An **alternative definition** of a relation can be given, making the ordering of values in a tuple *unnecessary*. In this definition, a relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a *set* of attributes (instead of a list), and a relation state $r(R)$ is a finite set of mappings $r = \{t_1, t_2, \dots, t_m\}$, where each tuple t_i is a **mapping** from R to D , and D is the **union** (denoted by \cup) of the attribute domains; that is, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \dots \cup \text{dom}(A_n)$. In this definition, $t[A_i]$ must be in $\text{dom}(A_i)$ for $1 \leq i \leq n$ for each mapping t in r . Each mapping t_i is called a tuple.

According to this definition of tuple as a mapping, a **tuple** can be considered as a **set** of ($\langle \text{attribute} \rangle, \langle \text{value} \rangle$) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $\text{dom}(A_i)$. The ordering of attributes is *not*

Figure 3.2

The relation STUDENT from Figure 3.1 with a different order of tuples.

STUDENT

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21

important, because the *attribute name* appears with its *value*. By this definition, the two tuples shown in Figure 3.3 are identical. This makes sense at an abstract level, since there really is no reason to prefer having one attribute value appear before another in a tuple.

When a relation is implemented as a file, the attributes are physically ordered as fields within a record. We will generally use the **first definition** of relation, where the attributes and the values within tuples *are ordered*, because it simplifies much of the notation. However, the alternative definition given here is more general.⁵

Values and NULLs in the Tuples. Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes (see Chapter 7) are not allowed. This model is sometimes called the **flat relational model**. Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption.⁶ Hence, multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes in the basic relational model.⁷

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases. For example, in Figure 3.1, some STUDENT tuples have NULL for their office phones because they do not have an office (that is, office phone *does not apply* to these students). Another student has a NULL for home phone, presumably because either he does not have a home phone or he has one but we do not know it (value is *unknown*). In general, we can have several meanings for NULL values, such as *value unknown*, *value exists but is not available*, or *attribute does not apply* to this tuple (also known as *value undefined*). An example of the last type of NULL will occur if we add an attribute Visa_status to the STUDENT relation

Figure 3.3

Two identical tuples when the order of attributes and values is not part of relation definition.

$$t = \langle (\text{Name, Dick Davidson}), (\text{Ssn, 422-11-2320}), (\text{Home_phone, NULL}), (\text{Address, 3452 Elgin Road}), (\text{Office_phone, (817)749-1253}), (\text{Age, 25}), (\text{Gpa, 3.53}) \rangle$$

$$t = \langle (\text{Address, 3452 Elgin Road}), (\text{Name, Dick Davidson}), (\text{Ssn, 422-11-2320}), (\text{Age, 25}), (\text{Office_phone, (817)749-1253}), (\text{Gpa, 3.53}), (\text{Home_phone, NULL}) \rangle$$

⁵As we shall see, the alternative definition of relation is useful when we discuss query processing and optimization in Chapter 19.

⁶We discuss this assumption in more detail in Chapter 15.

⁷Extensions of the relational model remove these restrictions. For example, object-relational systems (Chapter 11) allow complex-structured attributes, as do the **non-first normal form** or **nested** relational models.

that applies only to tuples representing foreign students. It is possible to devise different codes for different meanings of NULL values. Incorporating different types of NULL values into relational model operations (see Chapter 6) has proven difficult and is outside the scope of our presentation.

The exact meaning of a NULL value governs how it fares during arithmetic aggregations or comparisons with other values. For example, a comparison of two NULL values leads to ambiguities—if both Customer A and B have NULL addresses, it *does not mean* they have the same address. During database design, it is best to avoid NULL values as much as possible. We will discuss this further in Chapters 5 and 6 in the context of operations and queries, and in Chapter 15 in the context of database design and normalization.

Interpretation (Meaning) of a Relation. The relation schema can be interpreted as a declaration or a type of **assertion**. For example, the schema of the STUDENT relation of Figure 3.1 asserts that, in general, a student entity has a Name, Ssn, Home_phone, Address, Office_phone, Age, and Gpa. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion. For example, the first tuple in Figure 3.1 asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on.

Notice that some relations may represent facts about *entities*, whereas other relations may represent facts about *relationships*. For example, a relation schema MAJORS (Student_ssn, Department_code) asserts that students major in academic disciplines. A tuple in this relation relates a student to his or her major discipline. Hence, the relational model represents facts about both entities and relationships *uniformly* as relations. This sometimes compromises understandability because one has to guess whether a relation represents an entity type or a relationship type. We introduce the Entity-Relationship (ER) model in detail in Chapter 7 where the entity and relationship concepts will be described in detail. The mapping procedures in Chapter 9 show how different constructs of the ER and EER (Enhanced ER model covered in Chapter 8) conceptual data models (see Part 3) get converted to relations.

An alternative interpretation of a relation schema is as a **predicate**; in this case, the values in each tuple are interpreted as values that *satisfy* the predicate. For example, the predicate STUDENT (Name, Ssn, ...) is true for the five tuples in relation STUDENT of Figure 3.1. These tuples represent five different propositions or facts in the real world. This interpretation is quite useful in the context of logical programming languages, such as Prolog, because it allows the relational model to be used within these languages (see Section 26.5). An assumption called **the closed world assumption** states that the only true facts in the universe are those present within the extension (state) of the relation(s). Any other combination of values makes the predicate false.

3.1.3 Relational Model Notation

We will use the following notation in our presentation:

- A relation schema R of degree n is denoted by $R(A_1, A_2, \dots, A_n)$.

- The uppercase letters Q, R, S denote relation names.
- The lowercase letters q, r, s denote relation states.
- The letters t, u, v denote tuples.
- In general, the name of a relation schema such as STUDENT also indicates the current set of tuples in that relation—the *current relation state*—whereas STUDENT(Name, Ssn, ...) refers *only* to the relation schema.
- An attribute A can be qualified with the relation name R to which it belongs by using the dot notation $R.A$ —for example, STUDENT.Name or STUDENT.Age. This is because the same name may be used for two attributes in different relations. However, all attribute names *in a particular relation* must be distinct.
- An n -tuple t in a relation $r(R)$ is denoted by $t = \langle v_1, v_2, \dots, v_n \rangle$, where v_i is the value corresponding to attribute A_i . The following notation refers to **component values** of tuples:
- Both $t[A_i]$ and $t.A_i$ (and sometimes $t[i]$) refer to the value v_i in t for attribute A_i .
- Both $t[A_u, A_w, \dots, A_z]$ and $t.(A_u, A_w, \dots, A_z)$, where A_u, A_w, \dots, A_z is a list of attributes from R , refer to the subtuple of values $\langle v_u, v_w, \dots, v_z \rangle$ from t corresponding to the attributes specified in the list.

As an example, consider the tuple $t = \langle \text{'Barbara Benson'}, \text{'533-69-1238'}, \text{'(817)839-8461'}, \text{'7384 Fontana Lane'}, \text{NULL}, 19, 3.25 \rangle$ from the STUDENT relation in Figure 3.1; we have $t[\text{Name}] = \langle \text{'Barbara Benson'} \rangle$, and $t[\text{Ssn}, \text{Gpa}, \text{Age}] = \langle \text{'533-69-1238'}, 3.25, 19 \rangle$.

3.2 Relational Model Constraints and Relational Database Schemas

So far, we have discussed the characteristics of single relations. In a relational database, there will typically be many relations, and the tuples in those relations are usually related in various ways. The state of the whole database will correspond to the states of all its relations at a particular point in time. There are generally many restrictions or **constraints** on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents, as we discussed in Section 1.6.8.

In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints. Constraints on databases can generally be divided into three main categories:

1. Constraints that are inherent in the data model. We call these **inherent model-based constraints** or **implicit constraints**.
2. Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the DDL (data definition language, see Section 2.3.1). We call these **schema-based constraints** or **explicit constraints**.

3. Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs. We call these **application-based** or **semantic constraints** or **business rules**.

The characteristics of relations that we discussed in Section 3.1.2 are the inherent constraints of the relational model and belong to the first category. For example, the constraint that a relation cannot have duplicate tuples is an inherent constraint. The constraints we discuss in this section are of the second category, namely, constraints that can be expressed in the schema of the relational model via the DDL. Constraints in the third category are more general, relate to the meaning as well as behavior of attributes, and are difficult to express and enforce within the data model, so they are usually checked within the application programs that perform database updates.

Another important category of constraints is *data dependencies*, which include *functional dependencies* and *multivalued dependencies*. They are used mainly for testing the “goodness” of the design of a relational database and are utilized in a process called *normalization*, which is discussed in Chapters 15 and 16.

The schema-based constraints include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

3.2.1 Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain $\text{dom}(A)$. We have already discussed the ways in which domains can be specified in Section 3.1.1. The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double-precision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money, or other special data types. Other possible domains may be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed. Rather than describe these in detail here, we discuss the data types offered by the SQL relational standard in Section 4.1.

3.2.2 Key Constraints and Constraints on NULL Values

In the formal relational model, a *relation* is defined as a *set of tuples*. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK ; then for any two *distinct* tuples t_1 and t_2 in a relation state r of R , we have the constraint that:

$$t_1[SK] \neq t_2[SK]$$

Any such set of attributes SK is called a **superkey** of the relation schema R . A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state r of R can have the same value for SK. Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key*, which has no redundancy. A **key** K of a relation schema R is a superkey of R with the additional property that removing any attribute A from K leaves a set of attributes K' that is not a superkey of R any more. Hence, a key satisfies two properties:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This first property also applies to a superkey.
2. It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold. This property is not required by a superkey.

Whereas the first property applies to both keys and superkeys, the second property is required only for keys. Hence, a key is also a superkey but not vice versa. Consider the STUDENT relation of Figure 3.1. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn.⁸ Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey. However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey. In general, any superkey formed from a single attribute is also a key. A key with multiple attributes must require *all* its attributes together to have the uniqueness property.

The value of a key attribute can be used to identify uniquely each tuple in the relation. For example, the Ssn value 305-61-2435 identifies uniquely the tuple corresponding to Benjamin Bayer in the STUDENT relation. Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on *every* valid relation state of the schema. A key is determined from the meaning of the attributes, and the property is *time-invariant*: It must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the STUDENT relation in Figure 3.1 as a key because it is possible that two students with identical names will exist at some point in a valid state.⁹

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key**. For example, the CAR relation in Figure 3.4 has two candidate keys: License_number and Engine_serial_number. It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined, as shown in Figure 3.4. Notice that when a relation schema has several candidate keys,

⁸Note that Ssn is also a superkey.

⁹Names are sometimes used as keys, but then some artifact—such as appending an ordinal number—must be used to distinguish between identical names.

CAR

<u>License_number</u>	<u>Engine_serial_number</u>	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 3.4

The CAR relation, with two candidate keys: License_number and Engine_serial_number.

the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. The other candidate keys are designated as **unique keys**, and are not underlined.

Another constraint on attributes specifies whether NULL values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

3.2.3 Relational Databases and Relational Database Schemas

The definitions and constraints we have discussed so far apply to single relations and their attributes. A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section we define a relational database and a relational database schema.

A **relational database schema** S is a set of relation schemas $S = \{R_1, R_2, \dots, R_m\}$ and a set of **integrity constraints** IC. A **relational database state**¹⁰ DB of S is a set of relation states $DB = \{r_1, r_2, \dots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC. Figure 3.5 shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. The underlined attributes represent primary keys. Figure 3.6 shows a relational database state corresponding to the COMPANY schema. We will use this schema and database state in this chapter and in Chapters 4 through 6 for developing sample queries in different relational languages. (The data shown here is expanded and available for loading as a populated database from the Companion Website for the book, and can be used for the hands-on project exercises at the end of the chapters.)

When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is

¹⁰A relational database *state* is sometimes called a relational database *instance*. However, as we mentioned earlier, we will not use the term *instance* since it also applies to single tuples.

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

Figure 3.5

Schema diagram for the COMPANY relational database schema.

called an **invalid state**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state**.

In Figure 3.5, the Dnumber attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called Dno in EMPLOYEE and Dnum in PROJECT. Attributes that represent the same real-world concept may or may not have identical names in different relations. Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name Name for both Pname of PROJECT and Dname of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different real-world concepts—project names and department names.

In some early versions of the relational model, an assumption was made that the same real-world concept, when represented by an attribute, would have *identical* attribute names in all relations. This creates problems when the same real-world concept is used in different roles (meanings) in the same relation. For example, the concept of Social Security number appears twice in the EMPLOYEE relation of Figure 3.5: once in the role of the employee's SSN, and once in the role of the supervisor's SSN. We are required to give them distinct attribute names—Ssn and Super_ssn, respectively—because they appear in the same relation and in order to distinguish their meaning.

Each relational DBMS must have a data definition language (DDL) for defining a relational database schema. Current relational DBMSs are mostly using SQL for this purpose. We present the SQL DDL in Sections 4.1 and 4.2.

Figure 3.6

One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Integrity constraints are specified on a database schema and are expected to hold on every valid database state of that schema. In addition to domain, key, and NOT NULL constraints, two other types of constraints are considered part of the relational model: entity integrity and referential integrity.

3.2.4 Integrity, Referential Integrity, and Foreign Keys

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation. For example, in Figure 3.6, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, first we define the concept of a *foreign key*. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2 . A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that **references** relation R_2 if it satisfies the following rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R_2 ; the attributes FK are said to **reference** or **refer to** the relation R_2 .
2. A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ or is NULL. In the former case, we have $t_1[\text{FK}] = t_2[\text{PK}]$, and we say that the tuple t_1 **references** or **refers to** the tuple t_2 .

In this definition, R_1 is called the **referencing relation** and R_2 is the **referenced relation**. If these two conditions hold, a **referential integrity constraint** from R_1 to R_2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.

To specify these constraints, first we must have a clear understanding of the meaning or role that each attribute or set of attributes plays in the various relation schemas of the database. Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas. For example, consider the database shown in Figure 3.6. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation. This means that a value of Dno in any tuple t_1 of the EMPLOYEE relation must match a value of

the primary key of DEPARTMENT—the Dnumber attribute—in some tuple t_2 of the DEPARTMENT relation, or the value of Dno *can be NULL* if the employee does not belong to a department or will be assigned to a department later. For example, in Figure 3.6 the tuple for employee ‘John Smith’ references the tuple for the ‘Research’ department, indicating that ‘John Smith’ works for this department.

Notice that a foreign key can *refer to its own relation*. For example, the attribute Super_ssn in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, Super_ssn is a foreign key that references the EMPLOYEE relation itself. In Figure 3.6 the tuple for employee ‘John Smith’ references the tuple for employee ‘Franklin Wong,’ indicating that ‘Franklin Wong’ is the supervisor of ‘John Smith.’

We can *diagrammatically display referential integrity constraints* by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. Figure 3.7 shows the schema in Figure 3.5 with the referential integrity constraints displayed in this manner.

All integrity constraints should be specified on the relational database schema (i.e., defined as part of its definition) if we want to enforce these constraints on the database states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. Most relational DBMSs support key, entity integrity, and referential integrity constraints. These constraints are specified as a part of data definition in the DDL.

3.2.5 Other Types of Constraints

The preceding integrity constraints are included in the data definition language because they occur in most database applications. However, they do not include a large class of general constraints, sometimes called *semantic integrity constraints*, which may have to be specified and enforced on a relational database. Examples of such constraints are *the salary of an employee should not exceed the salary of the employee’s supervisor* and *the maximum number of hours an employee can work on all projects per week is 56*. Such constraints can be specified and enforced within the application programs that update the database, or by using a general-purpose **constraint specification language**. Mechanisms called **triggers** and **assertions** can be used. In SQL, CREATE ASSERTION and CREATE TRIGGER statements can be used for this purpose (see Chapter 5). It is more common to check for these types of constraints within the application programs than to use constraint specification languages because the latter are sometimes difficult and complex to use, as we discuss in Section 26.1.

Another type of constraint is the *functional dependency* constraint, which establishes a functional relationship among two sets of attributes X and Y . This constraint specifies that the value of X determines a unique value of Y in all states of a relation; it is denoted as a functional dependency $X \rightarrow Y$. We use functional dependencies and other types of dependencies in Chapters 15 and 16 as tools to analyze the quality of relational designs and to “normalize” relations to improve their quality.

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

Figure 3.7

Referential integrity constraints displayed on the COMPANY relational database schema.

The types of constraints we discussed so far may be called **state constraints** because they define the constraints that a *valid state* of the database must satisfy. Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database.¹¹ An example of a transition constraint is: “the salary of an employee can only increase.” Such constraints are typically enforced by the application programs or specified using active rules and triggers, as we discuss in Section 26.1.

3.3 Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates*. The relational algebra operations, which can be used to specify **retrievals**, are discussed in detail in Chapter 6. A relational algebra expression forms a new relation after applying a number of algebraic operators to an existing set of relations; its main use is for querying a database to retrieve information. The user formulates a query that specifies the data of interest, and a new relation is formed by applying relational operators to retrieve this data. That **result relation** becomes the

¹¹State constraints are sometimes called *static constraints*, and transition constraints are sometimes called *dynamic constraints*.

answer to (or result of) the user's query. Chapter 6 also introduces the language called relational calculus, which is used to define the new relation declaratively without giving a specific order of operations.

In this section, we concentrate on the database **modification** or **update** operations. There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records. **Insert** is used to insert one or more new tuples in a relation, **Delete** is used to delete tuples, and **Update** (or **Modify**) is used to change the values of some attributes in existing tuples. Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this section we discuss the types of constraints that may be violated by each of these operations and the types of actions that may be taken if an operation causes a violation. We use the database shown in Figure 3.6 for examples and discuss only key constraints, entity integrity constraints, and the referential integrity constraints shown in Figure 3.7. For each type of operation, we give some examples and discuss any constraints that each operation may violate.

3.3.1 The Insert Operation

The **Insert** operation provides a list of attribute values for a new tuple t that is to be inserted into a relation R . Insert can violate any of the four types of constraints discussed in the previous section. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type. Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation $r(R)$. Entity integrity can be violated if any part of the primary key of the new tuple t is NULL. Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation. Here are some examples to illustrate this discussion.

- *Operation:*
Insert <'Cecilia', 'F', 'Kolonsky', NULL, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.
Result: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.
- *Operation:*
Insert <'Alicia', 'J', 'Zelaya', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4> into EMPLOYEE.
Result: This insertion violates the key constraint because another tuple with the same Ssn value already exists in the EMPLOYEE relation, and so it is rejected.
- *Operation:*
Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windswept, Katy, TX', F, 28000, '987654321', 7> into EMPLOYEE.
Result: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.

- *Operation:*

Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.

Result: This insertion satisfies all constraints, so it is acceptable.

If an insertion violates one or more constraints, the default option is to *reject the insertion*. In this case, it would be useful if the DBMS could provide a reason to the user as to why the insertion was rejected. Another option is to attempt to *correct the reason for rejecting the insertion*, but this is typically not used for violations caused by Insert; rather, it is used more often in correcting violations for Delete and Update. In the first operation, the DBMS could ask the user to provide a value for Ssn, and could then accept the insertion if a valid Ssn value is provided. In operation 3, the DBMS could either ask the user to change the value of Dno to some valid value (or set it to NULL), or it could ask the user to insert a DEPARTMENT tuple with Dnumber = 7 and could accept the original insertion only after such an operation was accepted. Notice that in the latter case the insertion violation can **cascade** back to the EMPLOYEE relation if the user attempts to insert a tuple for department 7 with a value for Mgr_ssn that does not exist in the EMPLOYEE relation.

3.3.2 The Delete Operation

The **Delete** operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.

- *Operation:*

Delete the WORKS_ON tuple with Essn = '999887777' and Pno = 10.

Result: This deletion is acceptable and deletes exactly one tuple.

- *Operation:*

Delete the EMPLOYEE tuple with Ssn = '999887777'.

Result: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.

- *Operation:*

Delete the EMPLOYEE tuple with Ssn = '333445555'.

Result: This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT, WORKS_ON, and DEPENDENT relations.

Several options are available if a deletion operation causes a violation. The first option, called **restrict**, is to *reject the deletion*. The second option, called **cascade**, is to *attempt to cascade (or propagate) the deletion* by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with Essn = '999887777'. A third option, called **set null** or **set default**, is to *modify the referencing attribute values* that cause the violation; each such value is either set to NULL or changed to reference

another default valid tuple. Notice that if a referencing attribute that causes a violation is *part of the primary key*, it *cannot* be set to NULL; otherwise, it would violate entity integrity.

Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, the DBMS may automatically delete all tuples from WORKS_ON and DEPENDENT with Essn = '333445555'. Tuples in EMPLOYEE with Super_ssn = '333445555' and the tuple in DEPARTMENT with Mgr_ssn = '333445555' can have their Super_ssn and Mgr_ssn values changed to other valid values or to NULL. Although it may make sense to delete automatically the WORKS_ON and DEPENDENT tuples that refer to an EMPLOYEE tuple, it may not make sense to delete other EMPLOYEE tuples or a DEPARTMENT tuple.

In general, when a referential integrity constraint is specified in the DDL, the DBMS will allow the database designer to *specify which of the options* applies in case of a violation of the constraint. We discuss how to specify these options in the SQL DDL in Chapter 4.

3.3.3 The Update Operation

The **Update** (or **Modify**) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation *R*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Here are some examples.

- *Operation:*
Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.
Result: Acceptable.
- *Operation:*
Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1.
Result: Acceptable.
- *Operation:*
Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7.
Result: Unacceptable, because it violates referential integrity.
- *Operation:*
Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654321'.
Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn.

Updating an attribute that is *neither part of a primary key nor of a foreign key* usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Modifying a primary key value is similar to deleting one tuple and inserting another in its place because we use the primary key to identify tuples. Hence, the issues discussed earlier in both Sections 3.3.1 (Insert) and 3.3.2 (Delete) come into play. If a foreign key attribute is modified, the DBMS must

make sure that the new value refers to an existing tuple in the referenced relation (or is set to NULL). Similar options exist to deal with referential integrity violations caused by Update as those options discussed for the Delete operation. In fact, when a referential integrity constraint is specified in the DDL, the DBMS will allow the user to choose separate options to deal with a violation caused by Delete and a violation caused by Update (see Section 4.2).

3.3.4 The Transaction Concept

A database application program running against a relational database typically executes one or more *transactions*. A **transaction** is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database. At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema. A single transaction may involve any number of retrieval operations (to be discussed as part of relational algebra and calculus in Chapter 6, and as a part of the language SQL in Chapters 4 and 5), and any number of update operations. These retrievals and updates will together form an atomic unit of work against the database. For example, a transaction to apply a bank withdrawal will typically read the user account record, check if there is a sufficient balance, and then update the record by the withdrawal amount.

A large number of commercial applications running against relational databases in **online transaction processing (OLTP)** systems are executing transactions at rates that reach several hundred per second. Transaction processing concepts, concurrent execution of transactions, and recovery from failures will be discussed in Chapters 21 to 23.

3.4 Summary

In this chapter we presented the modeling concepts, data structures, and constraints provided by the relational model of data. We started by introducing the concepts of domains, attributes, and tuples. Then, we defined a relation schema as a list of attributes that describe the structure of a relation. A relation, or relation state, is a set of tuples that conforms to the schema.

Several characteristics differentiate relations from ordinary tables or files. The first is that a relation is not sensitive to the ordering of tuples. The second involves the ordering of attributes in a relation schema and the corresponding ordering of values within a tuple. We gave an alternative definition of relation that does not require these two orderings, but we continued to use the first definition, which requires attributes and tuple values to be ordered, for convenience. Then, we discussed values in tuples and introduced NULL values to represent missing or unknown information. We emphasized that NULL values should be avoided as much as possible.

We classified database constraints into inherent model-based constraints, explicit schema-based constraints, and application-based constraints, otherwise known as semantic constraints or business rules. Then, we discussed the schema constraints

pertaining to the relational model, starting with domain constraints, then key constraints, including the concepts of superkey, candidate key, and primary key, and the NOT NULL constraint on attributes. We defined relational databases and relational database schemas. Additional relational constraints include the entity integrity constraint, which prohibits primary key attributes from being NULL. We described the interrelation referential integrity constraint, which is used to maintain consistency of references among tuples from different relations.

The modification operations on the relational model are Insert, Delete, and Update. Each operation may violate certain types of constraints (refer to Section 3.3). Whenever an operation is applied, the database state after the operation is executed must be checked to ensure that no constraints have been violated. Finally, we introduced the concept of a transaction, which is important in relational DBMSs because it allows the grouping of several database operations into a single atomic action on the database.

Review Questions

- 3.1. Define the following terms as they apply to the relational model of data: *domain*, *attribute*, *n-tuple*, *relation schema*, *relation state*, *degree of a relation*, *relational database schema*, and *relational database state*.
- 3.2. Why are tuples in a relation not ordered?
- 3.3. Why are duplicate tuples not allowed in a relation?
- 3.4. What is the difference between a key and a superkey?
- 3.5. Why do we designate one of the candidate keys of a relation to be the primary key?
- 3.6. Discuss the characteristics of relations that make them different from ordinary tables and files.
- 3.7. Discuss the various reasons that lead to the occurrence of NULL values in relations.
- 3.8. Discuss the entity integrity and referential integrity constraints. Why is each considered important?
- 3.9. Define *foreign key*. What is this concept used for?
- 3.10. What is a transaction? How does it differ from an Update operation?

Exercises

- 3.11. Suppose that each of the following Update operations is applied directly to the database state shown in Figure 3.6. Discuss *all* integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

- a. Insert <'Robert', 'F', 'Scott', '943775543', '1972-06-21', '2365 Newcastle Rd, Bellaire, TX', M, 58000, '888665555', 1> into EMPLOYEE.
 - b. Insert <'ProductA', 4, 'Bellaire', 2> into PROJECT.
 - c. Insert <'Production', 4, '943775543', '2007-10-01'> into DEPARTMENT.
 - d. Insert <'677678989', NULL, '40.0'> into WORKS_ON.
 - e. Insert <'453453453', 'John', 'M', '1990-12-12', 'spouse'> into DEPENDENT.
 - f. Delete the WORKS_ON tuples with Essn = '333445555'.
 - g. Delete the EMPLOYEE tuple with Ssn = '987654321'.
 - h. Delete the PROJECT tuple with Pname = 'ProductX'.
 - i. Modify the Mgr_ssn and Mgr_start_date of the DEPARTMENT tuple with Dnumber = 5 to '123456789' and '2007-10-01', respectively.
 - j. Modify the Super_ssn attribute of the EMPLOYEE tuple with Ssn = '999887777' to '943775543'.
 - k. Modify the Hours attribute of the WORKS_ON tuple with Essn = '999887777' and Pno = 10 to '5.0'.
- 3.12.** Consider the AIRLINE relational database schema shown in Figure 3.8, which describes a database for airline flight information. Each FLIGHT is identified by a Flight_number, and consists of one or more FLIGHT_LEGs with Leg_numbers 1, 2, 3, and so on. Each FLIGHT_LEG has scheduled arrival and departure times, airports, and one or more LEG_INSTANCES—one for each Date on which the flight travels. FAREs are kept for each FLIGHT. For each FLIGHT_LEG instance, SEAT_RESERVATIONS are kept, as are the AIRPLANE used on the leg and the actual arrival and departure times and airports. An AIRPLANE is identified by an Airplane_id and is of a particular AIRPLANE_TYPE. CAN_LAND relates AIRPLANE_TYPES to the AIRPORTs at which they can land. An AIRPORT is identified by an Airport_code. Consider an update for the AIRLINE database to enter a reservation on a particular flight or flight leg on a given date.
- a. Give the operations for this update.
 - b. What types of constraints would you expect to check?
 - c. Which of these constraints are key, entity integrity, and referential integrity constraints, and which are not?
 - d. Specify all the referential integrity constraints that hold on the schema shown in Figure 3.8.
- 3.13.** Consider the relation CLASS(Course#, Univ_Section#, Instructor_name, Semester, Building_code, Room#, Time_period, Weekdays, Credit_hours). This represents classes taught in a university, with unique Univ_section#s. Identify what you think should be various candidate keys, and write in your own words the conditions or assumptions under which each candidate key would be valid.

AIRPORT

<u>Airport_code</u>	Name	City	State
---------------------	------	------	-------

FLIGHT

<u>Flight_number</u>	Airline	Weekdays
----------------------	---------	----------

FLIGHT_LEG

<u>Flight_number</u>	<u>Leg_number</u>	Departure_airport_code	Scheduled_departure_time
		Arrival_airport_code	Scheduled_arrival_time

LEG_INSTANCE

<u>Flight_number</u>	<u>Leg_number</u>	<u>Date</u>	Number_of_available_seats	Airplane_id
		Departure_airport_code	Departure_time	Arrival_airport_code
				Arrival_time

FARE

<u>Flight_number</u>	<u>Fare_code</u>	Amount	Restrictions
----------------------	------------------	--------	--------------

AIRPLANE_TYPE

<u>Airplane_type_name</u>	Max_seats	Company
---------------------------	-----------	---------

CAN_LAND

<u>Airplane_type_name</u>	<u>Airport_code</u>
---------------------------	---------------------

AIRPLANE

<u>Airplane_id</u>	Total_number_of_seats	Airplane_type
--------------------	-----------------------	---------------

SEAT_RESERVATION

<u>Flight_number</u>	<u>Leg_number</u>	<u>Date</u>	<u>Seat_number</u>	Customer_name	Customer_phone
----------------------	-------------------	-------------	--------------------	---------------	----------------

Figure 3.8

The AIRLINE relational database schema.

3.14. Consider the following six relations for an order-processing database application in a company:

CUSTOMER(Cust#, Cname, City)
 ORDER(Order#, Odate, Cust#, Ord_amt)
 ORDER_ITEM(Order#, Item#, Qty)

ITEM(Item#, Unit_price)
SHIPMENT(Order#, Warehouse#, Ship_date)
WAREHOUSE(Warehouse#, City)

Here, Ord_amt refers to total dollar amount of an order; Odate is the date the order was placed; and Ship_date is the date an order (or part of an order) is shipped from the warehouse. Assume that an order can be shipped from several warehouses. Specify the foreign keys for this schema, stating any assumptions you make. What other constraints can you think of for this database?

- 3.15.** Consider the following relations for a database that keeps track of business trips of salespersons in a sales office:

SALESPERSON(Ssn, Name, Start_year, Dept_no)
TRIP(Ssn, From_city, To_city, Departure_date, Return_date, Trip_id)
EXPENSE(Trip_id, Account#, Amount)

A trip can be charged to one or more accounts. Specify the foreign keys for this schema, stating any assumptions you make.

- 3.16.** Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:

STUDENT(Ssn, Name, Major, Bdate)
COURSE(Course#, Cname, Dept)
ENROLL(Ssn, Course#, Quarter, Grade)
BOOK_ADOPTION(Course#, Quarter, Book_isbn)
TEXT(Book_isbn, Book_title, Publisher, Author)

Specify the foreign keys for this schema, stating any assumptions you make.

- 3.17.** Consider the following relations for a database that keeps track of automobile sales in a car dealership (OPTION refers to some optional equipment installed on an automobile):

CAR(Serial_no, Model, Manufacturer, Price)
OPTION(Serial_no, Option_name, Price)
SALE(Salesperson_id, Serial_no, Date, Sale_price)
SALESPERSON(Salesperson_id, Name, Phone)

First, specify the foreign keys for this schema, stating any assumptions you make. Next, populate the relations with a few sample tuples, and then give an example of an insertion in the SALE and SALESPERSON relations that *violates* the referential integrity constraints and of another insertion that does not.

- 3.18.** Database design often involves decisions about the storage of attributes. For example, a Social Security number can be stored as one attribute or split into three attributes (one for each of the three hyphen-delineated groups of numbers in a Social Security number—XXX-XX-XXXX). However, Social Security numbers are usually represented as just one attribute. The decision

is based on how the database will be used. This exercise asks you to think about specific situations where dividing the SSN is useful.

- 3.19.** Consider a **STUDENT** relation in a **UNIVERSITY** database with the following attributes (Name, Ssn, Local_phone, Address, Cell_phone, Age, Gpa). Note that the cell phone may be from a different city and state (or province) from the local phone. A possible tuple of the relation is shown below:

Name	Ssn	Local_phone	Address	Cell_phone	Age	Gpa
George Shaw	123-45-6789	555-1234	123 Main St.,	555-4321	19	3.75
William Edwards			Anytown, CA 94539			

- Identify the critical missing information from the Local_phone and Cell_phone attributes. (*Hint:* How do you call someone who lives in a different state or province?)
 - Would you store this additional information in the Local_phone and Cell_phone attributes or add new attributes to the schema for **STUDENT**?
 - Consider the Name attribute. What are the advantages and disadvantages of splitting this field from one attribute into three attributes (first name, middle name, and last name)?
 - What general guideline would you recommend for deciding when to store information in a single attribute and when to split the information?
 - Suppose the student can have between 0 and 5 phones. Suggest two different designs that allow this type of information.
- 3.20.** Recent changes in privacy laws have disallowed organizations from using Social Security numbers to identify individuals unless certain restrictions are satisfied. As a result, most U.S. universities cannot use SSNs as primary keys (except for financial data). In practice, Student_id, a unique identifier assigned to every student, is likely to be used as the primary key rather than SSN since Student_id can be used throughout the system.
- Some database designers are reluctant to use generated keys (also known as *surrogate keys*) for primary keys (such as Student_id) because they are artificial. Can you propose any natural choices of keys that can be used to identify the student record in a **UNIVERSITY** database?
 - Suppose that you are able to guarantee uniqueness of a natural key that includes last name. Are you guaranteed that the last name will not change during the lifetime of the database? If last name can change, what solutions can you propose for creating a primary key that still includes last name but remains unique?
 - What are the advantages and disadvantages of using generated (surrogate) keys?

Selected Bibliography

The relational model was introduced by Codd (1970) in a classic paper. Codd also introduced relational algebra and laid the theoretical foundations for the relational model in a series of papers (Codd 1971, 1972, 1972a, 1974); he was later given the Turing Award, the highest honor of the ACM (Association for Computing Machinery) for his work on the relational model. In a later paper, Codd (1979) discussed extending the relational model to incorporate more meta-data and semantics about the relations; he also proposed a three-valued logic to deal with uncertainty in relations and incorporating NULLs in the relational algebra. The resulting model is known as RM/T. Childs (1968) had earlier used set theory to model databases. Later, Codd (1990) published a book examining over 300 features of the relational data model and database systems. Date (2001) provides a retrospective review and analysis of the relational data model.

Since Codd's pioneering work, much research has been conducted on various aspects of the relational model. Todd (1976) describes an experimental DBMS called PRTV that directly implements the relational algebra operations. Schmidt and Swenson (1975) introduce additional semantics into the relational model by classifying different types of relations. Chen's (1976) Entity-Relationship model, which is discussed in Chapter 7, is a means to communicate the real-world semantics of a relational database at the conceptual level. Wiederhold and Elmasri (1979) introduce various types of connections between relations to enhance its constraints. Extensions of the relational model are discussed in Chapters 11 and 26. Additional bibliographic notes for other aspects of the relational model and its languages, systems, extensions, and theory are given in Chapters 4 to 6, 9, 11, 13, 15, 16, 24, and 25. Maier (1983) and Atzeni and De Antonellis (1993) provide an extensive theoretical treatment of the relational data model.

The Relational Algebra and Relational Calculus

In this chapter we discuss the two *formal languages* for the relational model: the relational algebra and the relational calculus. In contrast, Chapters 4 and 5 described the *practical language* for the relational model, namely the SQL standard. Historically, the relational algebra and calculus were developed before the SQL language. In fact, in some ways, SQL is based on concepts from both the algebra and the calculus, as we shall see. Because most relational DBMSs use SQL as their language, we presented the SQL language first.

Recall from Chapter 2 that a data model must include a set of operations to manipulate the database, in addition to the data model's concepts for defining the database's structure and constraints. We presented the structures and constraints of the formal relational model in Chapter 3. The basic set of operations for the relational model is the **relational algebra**. These operations enable a user to specify basic retrieval requests as *relational algebra expressions*. The result of a retrieval is a new relation, which may have been formed from one or more relations. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

The relational algebra is very important for several reasons. First, it provides a formal foundation for relational model operations. Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs), as we shall discuss in Chapter 19. Third, some of its concepts are incorporated into the SQL standard query language for RDBMSs.

Although most commercial RDBMSs in use today do not provide user interfaces for relational algebra queries, the core operations and functions in the internal modules of most relational systems are based on relational algebra operations. We will define these operations in detail in Sections 6.1 through 6.4 of this chapter.

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level *declarative* language for specifying relational queries. A relational calculus expression creates a new relation. In a relational calculus expression, there is *no order of operations* to specify how to retrieve the query result—only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus. The relational calculus is important because it has a firm basis in mathematical logic and because the standard query language (SQL) for RDBMSs has some of its foundations in a variation of relational calculus known as the tuple relational calculus.¹

The relational algebra is often considered to be an integral part of the relational data model. Its operations can be divided into two groups. One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples in the *formal* relational model (see Section 3.1). Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT (also known as CROSS PRODUCT). The other group consists of operations developed specifically for relational databases—these include SELECT, PROJECT, and JOIN, among others. First, we describe the SELECT and PROJECT operations in Section 6.1 because they are **unary operations** that operate on single relations. Then we discuss set operations in Section 6.2. In Section 6.3, we discuss JOIN and other complex **binary operations**, which operate on two tables by combining related tuples (records) based on *join conditions*. The COMPANY relational database shown in Figure 3.6 is used for our examples.

Some common database requests cannot be performed with the original relational algebra operations, so additional operations were created to express these requests. These include **aggregate functions**, which are operations that can *summarize* data from the tables, as well as additional types of JOIN and UNION operations, known as OUTER JOINS and OUTER UNIONS. These operations, which were added to the original relational algebra because of their importance to many database applications, are described in Section 6.4. We give examples of specifying queries that use relational operations in Section 6.5. Some of these same queries were used in Chapters 4 and 5. By using the same query numbers in this chapter, the reader can contrast how the same queries are written in the various query languages.

In Sections 6.6 and 6.7 we describe the other main formal language for relational databases, the **relational calculus**. There are two variations of relational calculus. The *tuple* relational calculus is described in Section 6.6 and the *domain* relational calculus is described in Section 6.7. Some of the SQL constructs discussed in Chapters 4 and 5 are based on the tuple relational calculus. The relational calculus is a formal language, based on the branch of mathematical logic called predicate cal-

¹SQL is based on tuple relational calculus, but also incorporates some of the operations from the relational algebra and its extensions, as illustrated in Chapters 4, 5, and 9.

culus.² In tuple relational calculus, variables range over *tuples*, whereas in domain relational calculus, variables range over the *domains* (values) of attributes. In Appendix C we give an overview of the Query-By-Example (QBE) language, which is a graphical user-friendly relational language based on domain relational calculus. Section 6.8 summarizes the chapter.

For the reader who is interested in a less detailed introduction to formal relational languages, Sections 6.4, 6.6, and 6.7 may be skipped.

6.1 Unary Relational Operations: SELECT and PROJECT

6.1.1 The SELECT Operation

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition**.³ One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition. Alternatively, we can consider the SELECT operation to *restrict* the tuples in a relation to only those tuples that satisfy the condition. The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are discarded. For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than \$30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

$$\sigma_{\text{Dno}=4}(\text{EMPLOYEE})$$

$$\sigma_{\text{Salary}>30000}(\text{EMPLOYEE})$$

In general, the SELECT operation is denoted by

$$\sigma_{\langle \text{selection condition} \rangle}(R)$$

where the symbol σ (sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation R . Notice that R is generally a *relational algebra expression* whose result is a relation—the simplest such expression is just the name of a database relation. The relation resulting from the SELECT operation has the *same attributes* as R .

The Boolean expression specified in $\langle \text{selection condition} \rangle$ is made up of a number of **clauses** of the form

$\langle \text{attribute name} \rangle \langle \text{comparison op} \rangle \langle \text{constant value} \rangle$

or

$\langle \text{attribute name} \rangle \langle \text{comparison op} \rangle \langle \text{attribute name} \rangle$

²In this chapter no familiarity with first-order predicate calculus—which deals with quantified variables and values—is assumed.

³The SELECT operation is different from the SELECT clause of SQL. The SELECT operation chooses tuples from a table, and is sometimes called a RESTRICT or FILTER operation.

where <attribute name> is the name of an attribute of R , <comparison op> is normally one of the operators $\{=, <, \leq, >, \geq, \neq\}$, and <constant value> is a constant value from the attribute domain. Clauses can be connected by the standard Boolean operators *and*, *or*, and *not* to form a general selection condition. For example, to select the tuples for all employees who either work in department 4 and make over \$25,000 per year, or work in department 5 and make over \$30,000, we can specify the following SELECT operation:

$$\sigma_{(Dno=4 \text{ AND } Salary>25000) \text{ OR } (Dno=5 \text{ AND } Salary>30000)}(EMPLOYEE)$$

The result is shown in Figure 6.1(a).

Notice that all the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$ can apply to attributes whose domains are *ordered values*, such as numeric or date domains. Domains of strings of characters are also considered to be ordered based on the collating sequence of the characters. If the domain of an attribute is a set of *unordered values*, then only the comparison operators in the set $\{=, \neq\}$ can be used. An example of an unordered domain is the domain Color = { 'red', 'blue', 'green', 'white', 'yellow', ... }, where no order is specified among the various colors. Some domains allow additional types of comparison operators; for example, a domain of character strings may allow the comparison operator SUBSTRING_OF.

In general, the result of a SELECT operation can be determined as follows. The <selection condition> is applied independently to each *individual tuple* t in R . This is done by substituting each occurrence of an attribute A_i in the selection condition with its value in the tuple $t[A_i]$. If the condition evaluates to TRUE, then tuple t is

Figure 6.1

Results of SELECT and PROJECT operations. (a) $\sigma_{(Dno=4 \text{ AND } Salary>25000) \text{ OR } (Dno=5 \text{ AND } Salary>30000)}(EMPLOYEE)$. (b) $\pi_{Lname, Fname, Salary}(EMPLOYEE)$. (c) $\pi_{Sex, Salary}(EMPLOYEE)$.

(a)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5

(b)

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

selected. All the selected tuples appear in the result of the SELECT operation. The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:

- (cond1 **AND** cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
- (cond1 **OR** cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
- (**NOT** cond) is TRUE if cond is FALSE; otherwise, it is FALSE.

The SELECT operator is **unary**; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually*; hence, selection conditions cannot involve more than one tuple. The **degree** of the relation resulting from a SELECT operation—its number of attributes—is the same as the degree of R . The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in R . That is, $|\sigma_C(R)| \leq |R|$ for any condition C . The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition.

Notice that the SELECT operation is **commutative**; that is,

$$\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(R)) = \sigma_{\langle \text{cond2} \rangle}(\sigma_{\langle \text{cond1} \rangle}(R))$$

Hence, a sequence of SELECTs can be applied in any order. In addition, we can always combine a **cascade** (or **sequence**) of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is,

$$\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(\dots(\sigma_{\langle \text{cond}n \rangle}(R)) \dots)) = \sigma_{\langle \text{cond1} \rangle \text{ AND } \langle \text{cond2} \rangle \text{ AND } \dots \text{ AND } \langle \text{cond}n \rangle}(R)$$

In SQL, the SELECT condition is typically specified in the WHERE clause of a query. For example, the following operation:

$$\sigma_{\text{Dno}=4 \text{ AND } \text{Salary}>25000}(\text{EMPLOYEE})$$

would correspond to the following SQL query:

```
SELECT      *
FROM        EMPLOYEE
WHERE       Dno=4 AND Salary>25000;
```

6.1.2 The PROJECT Operation

If we think of a relation as a table, the SELECT operation chooses some of the *rows* from the table while discarding other rows. The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only. Therefore, the result of the PROJECT operation can be visualized as a *vertical partition* of the relation into two relations: one has the needed columns (attributes) and contains the result of the operation, and the other contains the discarded columns. For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

$$\pi_{\text{Lname, Fname, Salary}}(\text{EMPLOYEE})$$

The resulting relation is shown in Figure 6.1(b). The general form of the PROJECT operation is

$$\pi_{\langle \text{attribute list} \rangle}(R)$$

where π (pi) is the symbol used to represent the PROJECT operation, and $\langle \text{attribute list} \rangle$ is the desired sublist of attributes from the attributes of relation R . Again, notice that R is, in general, a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a database relation. The result of the PROJECT operation has only the attributes specified in $\langle \text{attribute list} \rangle$ *in the same order as they appear in the list*. Hence, its **degree** is equal to the number of attributes in $\langle \text{attribute list} \rangle$.

If the attribute list includes only nonkey attributes of R , duplicate tuples are likely to occur. The PROJECT operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**. For example, consider the following PROJECT operation:

$$\pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$$

The result is shown in Figure 6.1(c). Notice that the tuple $\langle \text{'F', 25000} \rangle$ appears only once in Figure 6.1(c), even though this combination of values appears twice in the EMPLOYEE relation. Duplicate elimination involves sorting or some other technique to detect duplicates and thus adds more processing. If duplicates are not eliminated, the result would be a **multiset** or **bag** of tuples rather than a set. This was not permitted in the formal relational model, but is allowed in SQL (see Section 4.3).

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in R . If the projection list is a superkey of R —that is, it includes some key of R —the resulting relation has the *same number* of tuples as R . Moreover,

$$\pi_{\langle \text{list1} \rangle}(\pi_{\langle \text{list2} \rangle}(R)) = \pi_{\langle \text{list1} \rangle}(R)$$

as long as $\langle \text{list2} \rangle$ contains the attributes in $\langle \text{list1} \rangle$; otherwise, the left-hand side is an incorrect expression. It is also noteworthy that commutativity *does not* hold on PROJECT.

In SQL, the PROJECT attribute list is specified in the SELECT clause of a query. For example, the following operation:

$$\pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$$

would correspond to the following SQL query:

```
SELECT    DISTINCT Sex, Salary
FROM      EMPLOYEE
```

Notice that if we remove the keyword **DISTINCT** from this SQL query, then duplicates will not be eliminated. This option is not available in the formal relational algebra.

6.1.3 Sequences of Operations and the RENAME Operation

The relations shown in Figure 6.1 that depict operation results do not have any names. In general, for most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results. For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression, also known as an **in-line expression**, as follows:

$$\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$$

Figure 6.2(a) shows the result of this in-line relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

$$\begin{aligned} \text{DEP5_EMPS} &\leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE}) \\ \text{RESULT} &\leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5_EMPS}) \end{aligned}$$

It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression. We can also use this technique to **rename** the attributes in the intermediate and

(a)

Fname	Lname	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

(b)

TEMP

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston,TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston,TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble,TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

R

First_name	Last_name	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

Figure 6.2

Results of a sequence of operations. (a) $\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$. (b) Using intermediate relations and renaming of attributes.

result relations. This can be useful in connection with more complex operations such as UNION and JOIN, as we shall see. To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

```
TEMP ← σDno=5(EMPLOYEE)
R(First_name, Last_name, Salary) ← πFname, Lname, Salary(TEMP)
```

These two operations are illustrated in Figure 6.2(b).

If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order. For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—as a unary operator. The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

$$\rho_{S(B_1, B_2, \dots, B_n)}(R) \text{ or } \rho_S(R) \text{ or } \rho_{(B_1, B_2, \dots, B_n)}(R)$$

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new relation name, and B_1, B_2, \dots, B_n are the new attribute names. The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of R are (A_1, A_2, \dots, A_n) in that order, then each A_i is renamed as B_i .

In SQL, a single query typically represents a complex relational algebra expression. Renaming in SQL is accomplished by aliasing using **AS**, as in the following example:

```
SELECT      E.Fname AS First_name, E.Lname AS Last_name, E.Salary AS Salary
FROM        EMPLOYEE AS E
WHERE       E.Dno=5,
```

6.2 Relational Algebra Operations from Set Theory

6.2.1 The UNION, INTERSECTION, and MINUS Operations

The next group of relational algebra operations are the standard mathematical operations on sets. For example, to retrieve the Social Security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows:⁴

⁴As a single relational algebra expression, this becomes $\text{Result} \leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE})) \cup \pi_{\text{Super_ssn}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$

```

DEP5_EMPS  $\leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$ 
RESULT1  $\leftarrow \pi_{\text{Ssn}}(\text{DEP5\_EMPS})$ 
RESULT2(Ssn)  $\leftarrow \pi_{\text{Super\_ssn}}(\text{DEP5\_EMPS})$ 
RESULT  $\leftarrow \text{RESULT1} \cup \text{RESULT2}$ 

```

The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn of all employees who directly supervise an employee who works in department 5. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both (see Figure 6.3), while eliminating any duplicates. Thus, the Ssn value ‘333445555’ appears only once in the result.

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION**, **INTERSECTION**, and **SET DIFFERENCE** (also called **MINUS** or **EXCEPT**). These are **binary** operations; that is, each is applied to two sets (of tuples). When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples**; this condition has been called *union compatibility* or *type compatibility*. Two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_n)$ are said to be **union compatible** (or **type compatible**) if they have the same degree n and if $\text{dom}(A_i) = \text{dom}(B_i)$ for $1 \leq i \leq n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations R and S as follows:

- **UNION**: The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S . Duplicate tuples are eliminated.
- **INTERSECTION**: The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S .
- **SET DIFFERENCE** (or **MINUS**): The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S .

We will adopt the convention that the resulting relation has the same attribute names as the *first* relation R . It is always possible to rename the attributes in the result using the rename operator.

RESULT1	RESULT2	RESULT
Ssn	Ssn	Ssn
123456789	333445555	123456789
333445555	888665555	333445555
666884444		666884444
453453453		453453453
		888665555

Figure 6.3

Result of the UNION operation
 $\text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2}$.

Figure 6.4 illustrates the three operations. The relations STUDENT and INSTRUCTOR in Figure 6.4(a) are union compatible and their tuples represent the names of students and the names of instructors, respectively. The result of the UNION operation in Figure 6.4(b) shows the names of all students and instructors. Note that duplicate tuples appear only once in the result. The result of the INTERSECTION operation (Figure 6.4(c)) includes only those who are both students and instructors.

Notice that both UNION and INTERSECTION are *commutative operations*; that is,

$$R \cup S = S \cup R \quad \text{and} \quad R \cap S = S \cap R$$

Both UNION and INTERSECTION can be treated as *n*-ary operations applicable to any number of relations because both are also *associative operations*; that is,

$$R \cup (S \cup T) = (R \cup S) \cup T \quad \text{and} \quad (R \cap S) \cap T = R \cap (S \cap T)$$

The MINUS operation is *not commutative*; that is, in general,

$$R - S \neq S - R$$

Figure 6.4

The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT \cup INSTRUCTOR. (c) STUDENT \cap INSTRUCTOR. (d) STUDENT $-$ INSTRUCTOR. (e) INSTRUCTOR $-$ STUDENT.

(a) STUDENT

F _n	L _n
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(b)

F _n	L _n
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

F _n	L _n
Susan	Yao
Ramesh	Shah

(d)

F _n	L _n
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

Figure 6.4(d) shows the names of students who are not instructors, and Figure 6.4(e) shows the names of instructors who are not students.

Note that INTERSECTION can be expressed in terms of union and set difference as follows:

$$R \cap S = ((R \cup S) - (R - S)) - (S - R)$$

In SQL, there are three operations—UNION, INTERSECT, and EXCEPT—that correspond to the set operations described here. In addition, there are multiset operations (UNION ALL, INTERSECT ALL, and EXCEPT ALL) that do not eliminate duplicates (see Section 4.3.4).

6.2.2 The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

Next, we discuss the **CARTESIAN PRODUCT** operation—also known as **CROSS PRODUCT** or **CROSS JOIN**—which is denoted by \times . This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible. In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set). In general, the result of $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S . Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then $R \times S$ will have $n_R * n_S$ tuples.

The n -ary CARTESIAN PRODUCT operation is an extension of the above concept, which produces new tuples by concatenating all possible combinations of tuples from n underlying relations.

In general, the CARTESIAN PRODUCT operation applied by itself is generally meaningless. It is mostly useful when followed by a selection that matches values of attributes coming from the component relations. For example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

```
FEMALE_EMPS ←  $\sigma_{\text{Sex}='F'}$ (EMPLOYEE)
EMPNAMES ←  $\pi_{\text{Fname, Lname, Ssn}}$ (FEMALE_EMPS)
EMP_DEPENDENTS ← EMPNAMES  $\times$  DEPENDENT
ACTUAL_DEPENDENTS ←  $\sigma_{\text{Ssn}=\text{Essn}}$ (EMP_DEPENDENTS)
RESULT ←  $\pi_{\text{Fname, Lname, Dependent\_name}}$ (ACTUAL_DEPENDENTS)
```

The resulting relations from this sequence of operations are shown in Figure 6.5. The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMPNAMES from Figure 6.5 with DEPENDENT from Figure 3.6. In EMP_DEPENDENTS, every tuple from EMPNAMES is combined with every tuple from DEPENDENT, giving a result that is not very meaningful (every dependent is combined with *every* female employee). We want to combine a female employee tuple only with her particular dependents—namely, the DEPENDENT tuples whose

Figure 6.5

The Cartesian Product (Cross Product) operation.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

Essn value match the Ssn value of the EMPLOYEE tuple. The ACTUAL_DEPENDENTS relation accomplishes this. The EMP_DEPENDENTS relation is a good example of the case where relational algebra can be correctly applied to yield results that make no sense at all. It is the responsibility of the user to make sure to apply only meaningful operations to relations.

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can SELECT *related tuples only* from the two relations by specifying an appropriate selection condition after the Cartesian product, as we did in the preceding example. Because this sequence of CARTESIAN PRODUCT followed by SELECT is quite commonly used to combine *related tuples* from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation. We discuss the JOIN operation next.

In SQL, CARTESIAN PRODUCT can be realized by using the CROSS JOIN option in joined tables (see Section 5.1.6). Alternatively, if there are two tables in the WHERE clause and there is no corresponding join condition in the query, the result will also be the CARTESIAN PRODUCT of the two tables (see Q10 in Section 4.3.3).

6.3 Binary Relational Operations: JOIN and DIVISION

6.3.1 The JOIN Operation

The JOIN operation, denoted by \bowtie , is used to combine *related tuples* from two relations into single “longer” tuples. This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations. To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department. To get the manager’s name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple. We do this by using the JOIN operation and then projecting the result over the necessary attributes, as follows:

```
DEPT_MGR  $\leftarrow$  DEPARTMENT  $\bowtie_{\text{Mgr\_ssn}=\text{Ssn}}$  EMPLOYEE
RESULT  $\leftarrow \pi_{\text{Dname, Lname, Fname}}(\text{DEPT\_MGR})$ 
```

The first operation is illustrated in Figure 6.6. Note that Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation. This referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.

The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation. However, JOIN is very important because it is used very frequently when specifying database queries. Consider the earlier example illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

```
EMP_DEPENDENTS  $\leftarrow$  EMPNAMES  $\times$  DEPENDENT
ACTUAL_DEPENDENTS  $\leftarrow \sigma_{\text{Ssn}=\text{Essn}}(\text{EMP\_DEPENDENTS})$ 
```

DEPT_MGR

Dname	Dnumber	Mgr_ssn	...	Fname	Minit	Lname	Ssn	...
Research	5	333445555	...	Franklin	T	Wong	333445555	...
Administration	4	987654321	...	Jennifer	S	Wallace	987654321	...
Headquarters	1	888665555	...	James	E	Borg	888665555	...

Figure 6.6

Result of the JOIN operation $\text{DEPT_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr_ssn}=\text{Ssn}} \text{EMPLOYEE}$.

These two operations can be replaced with a single JOIN operation as follows:

$\text{ACTUAL_DEPENDENTS} \leftarrow \text{EMP_NAMES} \bowtie_{\text{Ssn}=\text{Essn}} \text{DEPENDENT}$

The general form of a JOIN operation on two relations⁵ $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_m)$ is

$R \bowtie_{\langle \text{join condition} \rangle} S$

The result of the JOIN is a relation Q with $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S —*whenever the combination satisfies the join condition*. This is the main difference between **CARTESIAN PRODUCT** and **JOIN**. In **JOIN**, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the **CARTESIAN PRODUCT** *all* combinations of tuples are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to **TRUE** is included in the resulting relation Q as a *single combined tuple*.

A general join condition is of the form

$\langle \text{condition} \rangle \text{ AND } \langle \text{condition} \rangle \text{ AND } \dots \text{ AND } \langle \text{condition} \rangle$

where each $\langle \text{condition} \rangle$ is of the form $A_i \theta B_j$, A_i is an attribute of R , B_j is an attribute of S , A_i and B_j have the same domain, and θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$. A JOIN operation with such a general join condition is called a **THETA JOIN**. Tuples whose join attributes are **NULL** or for which the join condition is **FALSE** *do not* appear in the result. In that sense, the JOIN operation does *not* necessarily preserve all of the information in the participating relations, because tuples that do not get combined with matching ones in the other relation do not appear in the result.

⁵Again, notice that R and S can be any relations that result from general *relational algebra expressions*.

6.3.2 Variations of JOIN: The EQUIJOIN and NATURAL JOIN

The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is $=$, is called an **EQUIJOIN**. Both previous examples were EQUIJOINS. Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple. For example, in Figure 6.6, the values of the attributes *Mgr_ssn* and *Ssn* are identical in every tuple of *DEPT_MGR* (the EQUIJOIN result) because the equality join condition specified on these two attributes *requires the values to be identical* in every tuple in the result. Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN**—denoted by $*$ —was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.⁶ The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first.

Suppose we want to combine each *PROJECT* tuple with the *DEPARTMENT* tuple that controls the project. In the following example, first we rename the *Dnumber* attribute of *DEPARTMENT* to *Dnum*—so that it has the same name as the *Dnum* attribute in *PROJECT*—and then we apply NATURAL JOIN:

$$\text{PROJ_DEPT} \leftarrow \text{PROJECT} * \rho_{(\text{Dname}, \text{Dnum}, \text{Mgr_ssn}, \text{Mgr_start_date})}(\text{DEPARTMENT})$$

The same query can be done in two steps by creating an intermediate table *DEPT* as follows:

$$\begin{aligned} \text{DEPT} &\leftarrow \rho_{(\text{Dname}, \text{Dnum}, \text{Mgr_ssn}, \text{Mgr_start_date})}(\text{DEPARTMENT}) \\ \text{PROJ_DEPT} &\leftarrow \text{PROJECT} * \text{DEPT} \end{aligned}$$

The attribute *Dnum* is called the **join attribute** for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations. The resulting relation is illustrated in Figure 6.7(a). In the *PROJ_DEPT* relation, each tuple combines a *PROJECT* tuple with the *DEPARTMENT* tuple for the department that controls the project, but *only one join attribute value* is kept.

If the attributes on which the natural join is specified already *have the same names in both relations*, renaming is unnecessary. For example, to apply a natural join on the *Dnumber* attributes of *DEPARTMENT* and *DEPT_LOCATIONS*, it is sufficient to write

$$\text{DEPT_LOCS} \leftarrow \text{DEPARTMENT} * \text{DEPT_LOCATIONS}$$

The resulting relation is shown in Figure 6.7(b), which combines each department with its locations and has one tuple for each location. In general, the join condition for NATURAL JOIN is constructed by equating *each pair of join attributes* that have the same name in the two relations and combining these conditions with **AND**. There can be a list of join attributes from each relation, and each corresponding pair must have the same name.

⁶NATURAL JOIN is basically an EQUIJOIN followed by the removal of the superfluous attributes.

(a)

PROJ_DEPT

Pname	<u>Pnumber</u>	Plocation	Dnum	Dname	Mgr_ssn	Mgr_start_date
ProductX	1	Bellaire	5	Research	333445555	1988-05-22
ProductY	2	Sugarland	5	Research	333445555	1988-05-22
ProductZ	3	Houston	5	Research	333445555	1988-05-22
Computerization	10	Stafford	4	Administration	987654321	1995-01-01
Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS

Dname	Dnumber	Mgr_ssn	Mgr_start_date	Location
Headquarters	1	888665555	1981-06-19	Houston
Administration	4	987654321	1995-01-01	Stafford
Research	5	333445555	1988-05-22	Bellaire
Research	5	333445555	1988-05-22	Sugarland
Research	5	333445555	1988-05-22	Houston

Figure 6.7

Results of two NATURAL JOIN operations. (a) PROJ_DEPT \leftarrow PROJECT * DEPT.
 (b) DEPT_LOCS \leftarrow DEPARTMENT * DEPT_LOCATIONS.

A more general, *but nonstandard* definition for NATURAL JOIN is

$$Q \leftarrow R \star_{\langle \text{list1} \rangle, \langle \text{list2} \rangle} S$$

In this case, $\langle \text{list1} \rangle$ specifies a list of i attributes from R , and $\langle \text{list2} \rangle$ specifies a list of i attributes from S . The lists are used to form equality comparison conditions between pairs of corresponding attributes, and the conditions are then ANDed together. Only the list corresponding to attributes of the first relation R — $\langle \text{list1} \rangle$ —is kept in the result Q .

Notice that if no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples. In general, if R has n_R tuples and S has n_S tuples, the result of a JOIN operation $R \bowtie_{\langle \text{join condition} \rangle} S$ will have between zero and $n_R * n_S$ tuples. The expected size of the join result divided by the maximum size $n_R * n_S$ leads to a ratio called **join selectivity**, which is a property of each join condition. If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

As we can see, a single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins**, to distinguish them from a different join variation called

outer joins (see Section 6.4.4). Informally, an *inner join* is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. Note that sometimes a join may be specified between a relation and itself, as we will illustrate in Section 6.4.3. The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an *n-way join*. For example, consider the following three-way join:

$$((\text{PROJECT} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT}) \bowtie_{\text{Mgr_ssn=Ssn}} \text{EMPLOYEE})$$

This combines each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager. The net result is a consolidated relation in which each tuple contains this project-department-manager combined information.

In SQL, JOIN can be realized in several different ways. The first method is to specify the <join conditions> in the WHERE clause, along with any other selection conditions. This is very common, and is illustrated by queries Q1, Q1A, Q1B, Q2, and Q8 in Sections 4.3.1 and 4.3.2, as well as by many other query examples in Chapters 4 and 5. The second way is to use a nested relation, as illustrated by queries Q4A and Q16 in Section 5.1.2. Another way is to use the concept of joined tables, as illustrated by the queries Q1A, Q1B, Q8B, and Q2A in Section 5.1.6. The construct of joined tables was added to SQL2 to allow the user to specify explicitly all the various types of joins, because the other methods were more limited. It also allows the user to clearly distinguish join conditions from the selection conditions in the WHERE clause.

6.3.3 A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations $\{\sigma, \pi, \cup, \rho, -, \times\}$ is a **complete** set; that is, any of the other original relational algebra operations can be expressed as a *sequence of operations from this set*. For example, the INTERSECTION operation can be expressed by using UNION and MINUS as follows:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

Although, strictly speaking, INTERSECTION is not required, it is inconvenient to specify this complex expression every time we wish to specify an intersection. As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation, as we discussed:

$$R \bowtie_{\langle \text{condition} \rangle} S \equiv \sigma_{\langle \text{condition} \rangle} (R \times S)$$

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations. Hence, the various JOIN operations are also *not strictly necessary* for the expressive power of the relational algebra. However, they are important to include as separate operations because they are convenient to use and are very commonly applied in database applications. Other operations have been included in the basic relational algebra for convenience rather than necessity. We discuss one of these—the DIVISION operation—in the next section.

6.3.4 The DIVISION Operation

The DIVISION operation, denoted by \div , is useful for a special kind of query that sometimes occurs in database applications. An example is *Retrieve the names of employees who work on **all** the projects that ‘John Smith’ works on.* To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that ‘John Smith’ works on in the intermediate relation SMITH_PNOS:

```

SMITH ← σFname=‘John’ AND Lname=‘Smith’(EMPLOYEE)
SMITH_PNOS ← πPno(WORKS_ON ⋈Essn=SSn SMITH)

```

Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

```

SSN_PNOS ← πEssn, Pno(WORKS_ON)

```

Finally, apply the DIVISION operation to the two relations, which gives the desired employees’ Social Security numbers:

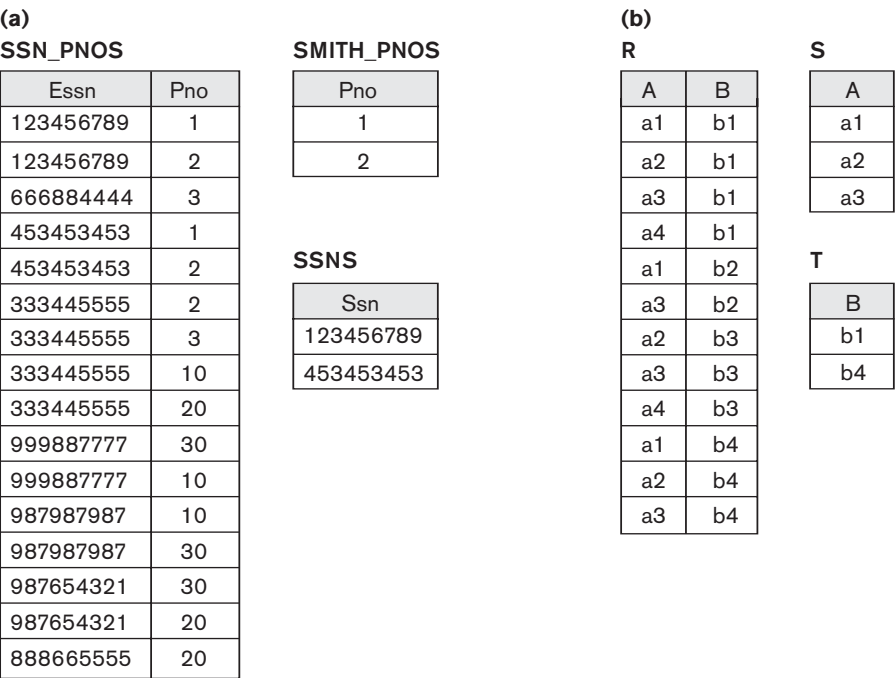
```

SSNS(Ssn) ← SSN_PNOS ÷ SMITH_PNOS
RESULT ← πFname, Lname(SSNS * EMPLOYEE)

```

The preceding operations are shown in Figure 6.8(a).

Figure 6.8
 The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.



In general, the DIVISION operation is applied to two relations $R(Z) \div S(X)$, where the attributes of R are a subset of the attributes of S ; that is, $X \subseteq Z$. Let Y be the set of attributes of R that are not attributes of S ; that is, $Y = Z - X$ (and hence $Z = X \cup Y$). The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = t_S$ for *every* tuple t_S in S . This means that, for a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with *every* tuple in S . Note that in the formulation of the DIVISION operation, the tuples in the denominator relation S restrict the numerator relation R by selecting those tuples in the result that match all values present in the denominator. It is not necessary to know what those values are as they can be computed by another operation, as illustrated in the SMITH_PNOS relation in the above example.

Figure 6.8(b) illustrates a DIVISION operation where $X = \{A\}$, $Y = \{B\}$, and $Z = \{A, B\}$. Notice that the tuples (values) b_1 and b_4 appear in R in combination with all three tuples in S ; that is why they appear in the resulting relation T . All other values of B in R do not appear with all the tuples in S and are not selected: b_2 does not appear with a_2 , and b_3 does not appear with a_1 .

The DIVISION operation can be expressed as a sequence of π , \times , and $-$ operations as follows:

$$\begin{aligned} T1 &\leftarrow \pi_Y(R) \\ T2 &\leftarrow \pi_Y((S \times T1) - R) \\ T &\leftarrow T1 - T2 \end{aligned}$$

The DIVISION operation is defined for convenience for dealing with queries that involve *universal quantification* (see Section 6.6.7) or the *all* condition. Most RDBMS implementations with SQL as the primary query language do not directly implement division. SQL has a roundabout way of dealing with the type of query illustrated above (see Section 5.1.4, queries Q3A and Q3B). Table 6.1 lists the various basic relational algebra operations we have discussed.

6.3.5 Notation for Query Trees

In this section we describe a notation typically used in relational systems to represent queries internally. The notation is called a *query tree* or sometimes it is known as a *query evaluation tree* or *query execution tree*. It includes the relational algebra operations being executed and is used as a possible data structure for the internal representation of the query in an RDBMS.

A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands (represented by its child nodes) are available, and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Table 6.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R .	$\sigma_{\langle \text{selection condition} \rangle}(R)$
PROJECT	Produces a new relation with only some of the attributes of R , and removes duplicate tuples.	$\pi_{\langle \text{attribute list} \rangle}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$, OR $R_1 \bowtie_{(\langle \text{join attributes 1} \rangle), (\langle \text{join attributes 2} \rangle)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 \star_{\langle \text{join condition} \rangle} R_2$, OR $R_1 \star_{(\langle \text{join attributes 1} \rangle), (\langle \text{join attributes 2} \rangle)} R_2$ OR $R_1 \star R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2 ; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2 .	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Figure 6.9 shows a query tree for Query 2 (see Section 4.3.1): *For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.* This query is specified on the relational schema of Figure 3.5 and corresponds to the following relational algebra expression:

$$\pi_{Pnumber, Dnum, Lname, Address, Bdate}(((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber}(DEPARTMENT)) \bowtie_{Mgr_ssn=Ssn}(EMPLOYEE))$$

In Figure 6.9, the three leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE. The relational algebra operations in the expression

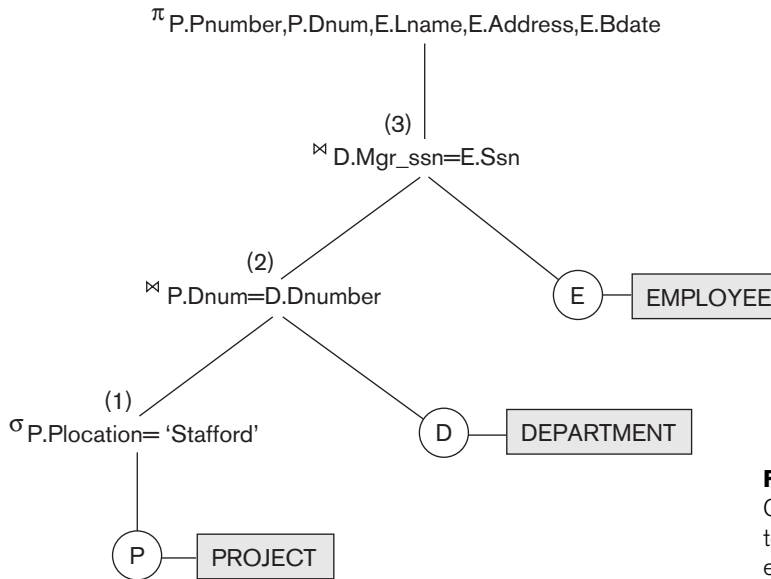


Figure 6.9
Query tree corresponding
to the relational algebra
expression for Q2.

are represented by internal tree nodes. The query tree signifies an explicit order of execution in the following sense. In order to execute Q2, the node marked (1) in Figure 6.9 must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2). Similarly, node (2) must begin to execute and produce results before node (3) can start execution, and so on. In general, a query tree gives a good visual representation and understanding of the query in terms of the relational operations it uses and is recommended as an additional means for expressing queries in relational algebra. We will revisit query trees when we discuss query processing and optimization in Chapter 19.

6.4 Additional Relational Operations

Some common database requests—which are needed in commercial applications for RDBMSs—cannot be performed with the original relational algebra operations described in Sections 6.1 through 6.3. In this section we define additional operations to express these requests. These operations enhance the expressive power of the original relational algebra.

6.4.1 Generalized Projection

The generalized projection operation extends the projection operation by allowing functions of attributes to be included in the projection list. The generalized form can be expressed as:

$$\pi_{F_1, F_2, \dots, F_n}(R)$$

where F_1, F_2, \dots, F_n are functions over the attributes in relation R and may involve arithmetic operations and constant values. This operation is helpful when developing reports where computed values have to be produced in the columns of a query result.

As an example, consider the relation

EMPLOYEE (Ssn, Salary, Deduction, Years_service)

A report may be required to show

Net Salary = Salary – Deduction,
 Bonus = 2000 * Years_service, and
 Tax = 0.25 * Salary.

Then a generalized projection combined with renaming may be used as follows:

REPORT $\leftarrow \rho_{(Ssn, Net_salary, Bonus, Tax)}(\pi_{Ssn, Salary - Deduction, 2000 * Years_service, 0.25 * Salary}(EMPLOYEE)).$

6.4.2 Aggregate Functions and Grouping

Another type of request that cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database. Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples. These functions are used in simple statistical queries that summarize information from the database tuples. Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.

Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function *independently to each group*. An example would be to group EMPLOYEE tuples by Dno, so that each group includes the tuples for employees working in the same department. We can then list each Dno value along with, say, the average salary of employees within the department, or the number of employees who work in the department.

We can define an AGGREGATE FUNCTION operation, using the symbol \mathfrak{S} (pronounced *script F*)⁷, to specify these types of requests as follows:

$\langle \text{grouping attributes} \rangle \mathfrak{S} \langle \text{function list} \rangle (R)$

where $\langle \text{grouping attributes} \rangle$ is a list of attributes of the relation specified in R , and $\langle \text{function list} \rangle$ is a list of ($\langle \text{function} \rangle \langle \text{attribute} \rangle$) pairs. In each such pair, $\langle \text{function} \rangle$ is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT—and $\langle \text{attribute} \rangle$ is an attribute of the relation specified by R . The

⁷There is no single agreed-upon notation for specifying aggregate functions. In some cases a “script A” is used.

resulting relation has the grouping attributes plus one attribute for each element in the function list. For example, to retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes as indicated below, we write:

$$\rho_{R(Dno, No_of_employees, Average_sal)}(Dno \bowtie COUNT Ssn, AVERAGE Salary (EMPLOYEE))$$

The result of this operation on the EMPLOYEE relation of Figure 3.6 is shown in Figure 6.10(a).

In the above example, we specified a list of attribute names—between parentheses in the RENAME operation—for the resulting relation R . If no renaming is applied, then the attributes of the resulting relation that correspond to the function list will each be the concatenation of the function name with the attribute name in the form $\langle \text{function} \rangle_ \langle \text{attribute} \rangle$.⁸ For example, Figure 6.10(b) shows the result of the following operation:

$$Dno \bowtie COUNT Ssn, AVERAGE Salary (EMPLOYEE)$$

If no grouping attributes are specified, the functions are applied to *all the tuples* in the relation, so the resulting relation has a *single tuple only*. For example, Figure 6.10(c) shows the result of the following operation:

$$\bowtie COUNT Ssn, AVERAGE Salary (EMPLOYEE)$$

It is important to note that, in general, duplicates are *not eliminated* when an aggregate function is applied; this way, the normal interpretation of functions such as

Figure 6.10

The aggregate function operation.

- $\rho_{R(Dno, No_of_employees, Average_sal)}(Dno \bowtie COUNT Ssn, AVERAGE Salary (EMPLOYEE)).$
- $Dno \bowtie COUNT Ssn, AVERAGE Salary (EMPLOYEE).$
- $\bowtie COUNT Ssn, AVERAGE Salary (EMPLOYEE).$

(a)

Dno	No_of_employees	Average_sal
5	4	33250
4	3	31000
1	1	55000

(b)

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

(c)

Count_ssn	Average_salary
8	35125

⁸Note that this is an arbitrary notation we are suggesting. There is no standard notation.

SUM and AVERAGE is computed.⁹ It is worth emphasizing that the result of applying an aggregate function is a relation, not a scalar number—even if it has a single value. This makes the relational algebra a closed mathematical system.

6.4.3 Recursive Closure Operations

Another type of operation that, in general, cannot be specified in the basic original relational algebra is **recursive closure**. This operation is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor. This relationship is described by the foreign key `Super_ssn` of the `EMPLOYEE` relation in Figures 3.5 and 3.6, and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor). An example of a recursive operation is to retrieve all supervisees of an employee e at all levels—that is, all employees e' directly supervised by e , all employees e'' directly supervised by each employee e' , all employees e''' directly supervised by each employee e'' , and so on.

It is relatively straightforward in the relational algebra to specify all employees supervised by e at a *specific level* by joining the table with itself one or more times. However, it is difficult to specify all supervisees at *all* levels. For example, to specify the `Ssns` of all employees e' directly supervised—at *level one*—by the employee e whose name is ‘James Borg’ (see Figure 3.6), we can apply the following operation:

$$\begin{aligned} \text{BORG_SSN} &\leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Fname}=\text{'James'} \text{ AND } \text{Lname}=\text{'Borg'}}(\text{EMPLOYEE})) \\ \text{SUPERVISION}(\text{Ssn1}, \text{Ssn2}) &\leftarrow \pi_{\text{Ssn}, \text{Super_ssn}}(\text{EMPLOYEE}) \\ \text{RESULT1}(\text{Ssn}) &\leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2}=\text{Ssn}} \text{BORG_SSN}) \end{aligned}$$

To retrieve all employees supervised by Borg at level 2—that is, all employees e'' supervised by some employee e' who is directly supervised by Borg—we can apply another **JOIN** to the result of the first query, as follows:

$$\text{RESULT2}(\text{Ssn}) \leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2}=\text{Ssn}} \text{RESULT1})$$

To get both sets of employees supervised at levels 1 and 2 by ‘James Borg’, we can apply the **UNION** operation to the two results, as follows:

$$\text{RESULT} \leftarrow \text{RESULT2} \cup \text{RESULT1}$$

The results of these queries are illustrated in Figure 6.11. Although it is possible to retrieve employees at each level and then take their **UNION**, we cannot, in general, specify a query such as “retrieve the supervisees of ‘James Borg’ at all levels” without utilizing a looping mechanism unless we know the maximum number of levels.¹⁰ An operation called the *transitive closure* of relations has been proposed to compute the recursive relationship as far as the recursion proceeds.

⁹In SQL, the option of eliminating duplicates before applying the aggregate function is available by including the keyword `DISTINCT` (see Section 4.4.4).

¹⁰The SQL3 standard includes syntax for recursive closure.

SUPERVISION

(Borg's Ssn is 888665555)

(Ssn) (Super_ssn)

Ssn1	Ssn2
123456789	333445555
333445555	888665555
999887777	987654321
987654321	888665555
666884444	333445555
453453453	333445555
987987987	987654321
888665555	null

RESULT1

Ssn
333445555
987654321

(Supervised by Borg)

RESULT2

Ssn
123456789
999887777
666884444
453453453
987987987

(Supervised by
Borg's subordinates)**RESULT**

Ssn
123456789
999887777
666884444
453453453
987987987
333445555
987654321

(RESULT1 \cup RESULT2)**Figure 6.11**

A two-level recursive query.

6.4.4 OUTER JOIN Operations

Next, we discuss some additional extensions to the JOIN operation that are necessary to specify certain types of queries. The JOIN operations described earlier match tuples that satisfy the join condition. For example, for a NATURAL JOIN operation $R * S$, only tuples from R that have matching tuples in S —and vice versa—appear in the result. Hence, tuples without a *matching* (or *related*) tuple are eliminated from the JOIN result. Tuples with NULL values in the join attributes are also eliminated. This type of join, where tuples with no match are eliminated, is known as an **inner join**. The join operations we described earlier in Section 6.3 are all inner joins. This amounts to the loss of information if the user wants the result of the JOIN to include all the tuples in one or more of the component relations.

A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in R , or all those in S , or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation. This satisfies the need of queries in which tuples from two tables are

to be combined by matching corresponding rows, but without losing any tuples for lack of matching values. For example, suppose that we want a list of all employee names as well as the name of the departments they manage *if they happen to manage a department*; if they do not manage one, we can indicate it with a NULL value. We can apply an operation **LEFT OUTER JOIN**, denoted by \bowtie , to retrieve the result as follows:

$$\text{TEMP} \leftarrow (\text{EMPLOYEE} \bowtie_{\text{Ssn}=\text{Mgr_ssn}} \text{DEPARTMENT})$$

$$\text{RESULT} \leftarrow \pi_{\text{Fname, Minit, Lname, Dname}}(\text{TEMP})$$

The **LEFT OUTER JOIN** operation keeps every tuple in the *first*, or *left*, relation R in $R \bowtie S$; if no matching tuple is found in S , then the attributes of S in the join result are filled or *padded* with NULL values. The result of these operations is shown in Figure 6.12.

A similar operation, **RIGHT OUTER JOIN**, denoted by \bowtie , keeps every tuple in the *second*, or *right*, relation S in the result of $R \bowtie S$. A third operation, **FULL OUTER JOIN**, denoted by \bowtie , keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with NULL values as needed. The three outer join operations are part of the SQL2 standard (see Section 5.1.6). These operations were provided later as an extension of relational algebra in response to the typical need in business applications to show related information from multiple tables exhaustively. Sometimes a complete reporting of data from multiple tables is required whether or not there are matching values.

6.4.5 The OUTER UNION Operation

The **OUTER UNION** operation was developed to take the union of tuples from two relations that have some common attributes, but are *not union (type) compatible*. This operation will take the UNION of tuples in two relations $R(X, Y)$ and $S(X, Z)$ that are **partially compatible**, meaning that only some of their attributes, say X , are union compatible. The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either

Figure 6.12

The result of a LEFT OUTER JOIN operation.

RESULT

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

relation are also kept in the result relation $T(X, Y, Z)$. It is therefore the same as a FULL OUTER JOIN on the common attributes.

Two tuples t_1 in R and t_2 in S are said to **match** if $t_1[X] = t_2[X]$. These will be combined (unioned) into a single tuple in t . Tuples in either relation that have no matching tuple in the other relation are padded with NULL values. For example, an OUTER UNION can be applied to two relations whose schemas are STUDENT(Name, Ssn, Department, Advisor) and INSTRUCTOR(Name, Ssn, Department, Rank). Tuples from the two relations are matched based on having the same combination of values of the shared attributes—Name, Ssn, Department. The resulting relation, STUDENT_OR_INSTRUCTOR, will have the following attributes:

STUDENT_OR_INSTRUCTOR(Name, Ssn, Department, Advisor, Rank)

All the tuples from both relations are included in the result, but tuples with the same (Name, Ssn, Department) combination will appear only once in the result. Tuples appearing only in STUDENT will have a NULL for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a NULL for the Advisor attribute. A tuple that exists in both relations, which represent a student who is also an instructor, will have values for all its attributes.¹¹

Notice that the same person may still appear twice in the result. For example, we could have a graduate student in the Mathematics department who is an instructor in the Computer Science department. Although the two tuples representing that person in STUDENT and INSTRUCTOR will have the same (Name, Ssn) values, they will not agree on the Department value, and so will not be matched. This is because Department has two different meanings in STUDENT (the department where the person studies) and INSTRUCTOR (the department where the person is employed as an instructor). If we wanted to apply the OUTER UNION based on the same (Name, Ssn) combination only, we should rename the Department attribute in each table to reflect that they have different meanings and designate them as not being part of the union-compatible attributes. For example, we could rename the attributes as MajorDept in STUDENT and WorkDept in INSTRUCTOR.

6.5 Examples of Queries in Relational Algebra

The following are additional examples to illustrate the use of the relational algebra operations. All examples refer to the database in Figure 3.6. In general, the same query can be stated in numerous ways using the various operations. We will state each query in one way and leave it to the reader to come up with equivalent formulations.

Query 1. Retrieve the name and address of all employees who work for the ‘Research’ department.

¹¹Note that OUTER UNION is equivalent to a FULL OUTER JOIN if the join attributes are *all* the common attributes of the two relations.

```

RESEARCH_DEPT  $\leftarrow \sigma_{\text{Dname}='Research'}(\text{DEPARTMENT})$ 
RESEARCH_EMPS  $\leftarrow (\text{RESEARCH_DEPT} \bowtie_{\text{Dnumber}=\text{Dno}} \text{EMPLOYEE})$ 
RESULT  $\leftarrow \pi_{\text{Fname, Lname, Address}}(\text{RESEARCH_EMPS})$ 

```

As a single in-line expression, this query becomes:

```

 $\pi_{\text{Fname, Lname, Address}}(\sigma_{\text{Dname}='Research'}(\text{DEPARTMENT} \bowtie_{\text{Dnumber}=\text{Dno}} (\text{EMPLOYEE})))$ 

```

This query could be specified in other ways; for example, the order of the JOIN and SELECT operations could be reversed, or the JOIN could be replaced by a NATURAL JOIN after renaming one of the join attributes to match the other join attribute name.

Query 2. For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.

```

STAFFORD_PROJS  $\leftarrow \sigma_{\text{Plocation}='Stafford'}(\text{PROJECT})$ 
CONTR_DEPTS  $\leftarrow (\text{STAFFORD_PROJS} \bowtie_{\text{Dnum}=\text{Dnumber}} \text{DEPARTMENT})$ 
PROJ_DEPT_MGRS  $\leftarrow (\text{CONTR_DEPTS} \bowtie_{\text{Mgr\_ssn}=\text{Ssn}} \text{EMPLOYEE})$ 
RESULT  $\leftarrow \pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(\text{PROJ_DEPT_MGRS})$ 

```

In this example, we first select the projects located in Stafford, then join them with their controlling departments, and then join the result with the department managers. Finally, we apply a project operation on the desired attributes.

Query 3. Find the names of employees who work on *all* the projects controlled by department number 5.

```

DEPT5_PROJS  $\leftarrow \rho_{(\text{Pno})}(\pi_{\text{Pnumber}}(\sigma_{\text{Dnum}=5}(\text{PROJECT})))$ 
EMP_PROJ  $\leftarrow \rho_{(\text{Ssn, Pno})}(\pi_{\text{Essn, Pno}}(\text{WORKS\_ON}))$ 
RESULT_EMP_SSNS  $\leftarrow \text{EMP\_PROJ} \div \text{DEPT5\_PROJS}$ 
RESULT  $\leftarrow \pi_{\text{Lname, Fname}}(\text{RESULT\_EMP\_SSNS} * \text{EMPLOYEE})$ 

```

In this query, we first create a table DEPT5_PROJS that contains the project numbers of all projects controlled by department 5. Then we create a table EMP_PROJ that holds (Ssn, Pno) tuples, and apply the division operation. Notice that we renamed the attributes so that they will be correctly used in the division operation. Finally, we join the result of the division, which holds only Ssn values, with the EMPLOYEE table to retrieve the desired attributes from EMPLOYEE.

Query 4. Make a list of project numbers for projects that involve an employee whose last name is ‘Smith’, either as a worker or as a manager of the department that controls the project.

```

SMITHS(Essn)  $\leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Lname}='Smith'}(\text{EMPLOYEE}))$ 
SMITH_WORKER_PROJS  $\leftarrow \pi_{\text{Pno}}(\text{WORKS\_ON} * \text{SMITHS})$ 
MGRS  $\leftarrow \pi_{\text{Lname, Dnumber}}(\text{EMPLOYEE} \bowtie_{\text{Ssn}=\text{Mgr\_ssn}} \text{DEPARTMENT})$ 
SMITH_MANAGED_DEPTS(Dnum)  $\leftarrow \pi_{\text{Dnumber}}(\sigma_{\text{Lname}='Smith'}(\text{MGRS}))$ 
SMITH_MGR_PROJS(Pno)  $\leftarrow \pi_{\text{Pnumber}}(\text{SMITH\_MANAGED\_DEPTS} * \text{PROJECT})$ 
RESULT  $\leftarrow (\text{SMITH\_WORKER\_PROJS} \cup \text{SMITH\_MGR\_PROJS})$ 

```

In this query, we retrieved the project numbers for projects that involve an employee named Smith as a worker in SMITH_WORKER_PROJS. Then we retrieved the project numbers for projects that involve an employee named Smith as manager of the department that controls the project in SMITH_MGR_PROJS. Finally, we applied the **UNION** operation on SMITH_WORKER_PROJS and SMITH_MGR_PROJS. As a single in-line expression, this query becomes:

$$\pi_{Pno} (WORKS_ON \bowtie_{Essn=Ssn} (\pi_{Ssn} (\sigma_{Lname='Smith'}(EMPLOYEE)))) \cup \pi_{Pno} ((\pi_{Dnumber} (\sigma_{Lname='Smith'}(\pi_{Lname, Dnumber}(EMPLOYEE)))) \bowtie_{Ssn=Mgr_ssn} DEPARTMENT)) \bowtie_{Dnumber=Dnum} PROJECT)$$

Query 5. List the names of all employees with two or more dependents.

Strictly speaking, this query cannot be done in the *basic (original) relational algebra*. We have to use the AGGREGATE FUNCTION operation with the COUNT aggregate function. We assume that dependents of the *same* employee have *distinct* Dependent_name values.

$$T1(Ssn, No_of_dependents) \leftarrow_{Essn} \mathfrak{S} \text{ COUNT Dependent_name } (DEPENDENT) \\ T2 \leftarrow \sigma_{No_of_dependents > 2}(T1) \\ RESULT \leftarrow \pi_{Lname, Fname}(T2 * EMPLOYEE)$$

Query 6. Retrieve the names of employees who have no dependents.

This is an example of the type of query that uses the MINUS (SET DIFFERENCE) operation.

$$ALL_EMPS \leftarrow \pi_{Ssn}(EMPLOYEE) \\ EMPS_WITH_DEPS(Ssn) \leftarrow \pi_{Essn}(DEPENDENT) \\ EMPS_WITHOUT_DEPS \leftarrow (ALL_EMPS - EMPS_WITH_DEPS) \\ RESULT \leftarrow \pi_{Lname, Fname}(EMPS_WITHOUT_DEPS * EMPLOYEE)$$

We first retrieve a relation with all employee Ssns in ALL_EMPS. Then we create a table with the Ssns of employees who have at least one dependent in EMPS_WITH_DEPS. Then we apply the SET DIFFERENCE operation to retrieve employees Ssns with no dependents in EMPS_WITHOUT_DEPS, and finally join this with EMPLOYEE to retrieve the desired attributes. As a single in-line expression, this query becomes:

$$\pi_{Lname, Fname}((\pi_{Ssn}(EMPLOYEE) - \rho_{Ssn}(\pi_{Essn}(DEPENDENT)))) * EMPLOYEE)$$

Query 7. List the names of managers who have at least one dependent.

$$MGRS(Ssn) \leftarrow \pi_{Mgr_ssn}(DEPARTMENT) \\ EMPS_WITH_DEPS(Ssn) \leftarrow \pi_{Essn}(DEPENDENT) \\ MGRS_WITH_DEPS \leftarrow (MGRS \cap EMPS_WITH_DEPS) \\ RESULT \leftarrow \pi_{Lname, Fname}(MGRS_WITH_DEPS * EMPLOYEE)$$

In this query, we retrieve the Ssns of managers in MGRS, and the Ssns of employees with at least one dependent in EMPS_WITH_DEPS, then we apply the SET INTERSECTION operation to get the Ssns of managers who have at least one dependent.

As we mentioned earlier, the same query can be specified in many different ways in relational algebra. In particular, the operations can often be applied in various orders. In addition, some operations can be used to replace others; for example, the **INTERSECTION** operation in Q7 can be replaced by a **NATURAL JOIN**. As an exercise, try to do each of these sample queries using different operations.¹² We showed how to write queries as single relational algebra expressions for queries Q1, Q4, and Q6. Try to write the remaining queries as single expressions. In Chapters 4 and 5 and in Sections 6.6 and 6.7, we show how these queries are written in other relational languages.

6.6 The Tuple Relational Calculus

In this and the next section, we introduce another formal query language for the relational model called **relational calculus**. This section introduces the language known as **tuple relational calculus**, and Section 6.7 introduces a variation called **domain relational calculus**. In both variations of relational calculus, we write one **declarative** expression to specify a retrieval request; hence, there is no description of how, or *in what order*, to evaluate a query. A calculus expression specifies *what* is to be retrieved rather than *how* to retrieve it. Therefore, the relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request *in a particular order* of applying the operations; thus, it can be considered as a **procedural** way of stating a query. It is possible to nest algebra operations to form a single expression; however, a certain order among the operations is always explicitly specified in a relational algebra expression. This order also influences the strategy for evaluating the query. A calculus expression may be written in different ways, but the way it is written has no bearing on how a query should be evaluated.

It has been shown that any retrieval that can be specified in the basic relational algebra can also be specified in relational calculus, and vice versa; in other words, the **expressive power** of the languages is *identical*. This led to the definition of the concept of a *relationally complete* language. A relational query language *L* is considered **relationally complete** if we can express in *L* any query that can be expressed in relational calculus. Relational completeness has become an important basis for comparing the expressive power of high-level query languages. However, as we saw in Section 6.4, certain frequently required queries in database applications cannot be expressed in basic relational algebra or calculus. Most relational query languages are relationally complete but have *more expressive power* than relational algebra or relational calculus because of additional operations such as aggregate functions, grouping, and ordering. As we mentioned in the introduction to this chapter, the relational calculus is important for two reasons. First, it has a firm basis in mathematical logic. Second, the standard query language (SQL) for RDBMSs has some of its foundations in the tuple relational calculus.

¹²When queries are optimized (see Chapter 19), the system will choose a particular sequence of operations that corresponds to an execution strategy that can be executed efficiently.

Our examples refer to the database shown in Figures 3.6 and 3.7. We will use the same queries that were used in Section 6.5. Sections 6.6.6, 6.6.7, and 6.6.8 discuss dealing with universal quantifiers and safety of expression issues. (Students interested in a basic introduction to tuple relational calculus may skip these sections.)

6.6.1 Tuple Variables and Range Relations

The tuple relational calculus is based on specifying a number of **tuple variables**. Each tuple variable usually *ranges over* a particular database relation, meaning that the variable may take as its value any individual tuple from that relation. A simple tuple relational calculus query is of the form:

$$\{t \mid \text{COND}(t)\}$$

where t is a tuple variable and $\text{COND}(t)$ is a conditional (Boolean) expression involving t that evaluates to either TRUE or FALSE for different assignments of tuples to the variable t . The result of such a query is the set of all tuples t that evaluate $\text{COND}(t)$ to TRUE. These tuples are said to **satisfy** $\text{COND}(t)$. For example, to find all employees whose salary is above \$50,000, we can write the following tuple calculus expression:

$$\{t \mid \text{EMPLOYEE}(t) \text{ AND } t.\text{Salary} > 50000\}$$

The condition $\text{EMPLOYEE}(t)$ specifies that the **range relation** of tuple variable t is **EMPLOYEE**. Each **EMPLOYEE** tuple t that satisfies the condition $t.\text{Salary} > 50000$ will be retrieved. Notice that $t.\text{Salary}$ references attribute **Salary** of tuple variable t ; this notation resembles how attribute names are qualified with relation names or aliases in SQL, as we saw in Chapter 4. In the notation of Chapter 3, $t.\text{Salary}$ is the same as writing $t[\text{Salary}]$.

The above query retrieves all attribute values for each selected **EMPLOYEE** tuple t . To retrieve only *some* of the attributes—say, the first and last names—we write

$$\{t.\text{Fname}, t.\text{Lname} \mid \text{EMPLOYEE}(t) \text{ AND } t.\text{Salary} > 50000\}$$

Informally, we need to specify the following information in a tuple relational calculus expression:

- For each tuple variable t , the **range relation** R of t . This value is specified by a condition of the form $R(t)$. If we do not specify a range relation, then the variable t will range over all possible tuples “in the universe” as it is not restricted to any one relation.
- A condition to select particular combinations of tuples. As tuple variables range over their respective range relations, the condition is evaluated for every possible combination of tuples to identify the **selected combinations** for which the condition evaluates to TRUE.
- A set of attributes to be retrieved, the **requested attributes**. The values of these attributes are retrieved for each selected combination of tuples.

Before we discuss the formal syntax of tuple relational calculus, consider another query.

Query 0. Retrieve the birth date and address of the employee (or employees) whose name is John B. Smith.

Q0: $\{t.Bdate, t.Address \mid \text{EMPLOYEE}(t) \text{ AND } t.Fname='John' \text{ AND } t.Minit='B' \text{ AND } t.Lname='Smith'\}$

In tuple relational calculus, we first specify the requested attributes $t.Bdate$ and $t.Address$ for each selected tuple t . Then we specify the condition for selecting a tuple following the bar ($|$)—namely, that t be a tuple of the **EMPLOYEE** relation whose *Fname*, *Minit*, and *Lname* attribute values are 'John', 'B', and 'Smith', respectively.

6.6.2 Expressions and Formulas in Tuple Relational Calculus

A general **expression** of the tuple relational calculus is of the form

$$\{t_1.A_j, t_2.A_k, \dots, t_n.A_m \mid \text{COND}(t_1, t_2, \dots, t_n, t_{n+1}, t_{n+2}, \dots, t_{n+m})\}$$

where $t_1, t_2, \dots, t_n, t_{n+1}, \dots, t_{n+m}$ are tuple variables, each A_i is an attribute of the relation on which t_i ranges, and **COND** is a **condition** or **formula**.¹³ of the tuple relational calculus. A formula is made up of predicate calculus **atoms**, which can be one of the following:

1. An atom of the form $R(t_i)$, where R is a relation name and t_i is a tuple variable. This atom identifies the range of the tuple variable t_i as the relation whose name is R . It evaluates to **TRUE** if t_i is a tuple in the relation R , and evaluates to **FALSE** otherwise.
2. An atom of the form $t_i.A \text{ op } t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, and B is an attribute of the relation on which t_j ranges.
3. An atom of the form $t_i.A \text{ op } c$ or $c \text{ op } t_j.B$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, t_i and t_j are tuple variables, A is an attribute of the relation on which t_i ranges, B is an attribute of the relation on which t_j ranges, and c is a constant value.

Each of the preceding atoms evaluates to either **TRUE** or **FALSE** for a specific combination of tuples; this is called the **truth value** of an atom. In general, a tuple variable t ranges over all possible tuples *in the universe*. For atoms of the form $R(t)$, if t is assigned to a tuple that is a *member of the specified relation* R , the atom is **TRUE**; otherwise, it is **FALSE**. In atoms of types 2 and 3, if the tuple variables are assigned to tuples such that the values of the specified attributes of the tuples satisfy the condition, then the atom is **TRUE**.

A **formula** (Boolean condition) is made up of one or more atoms connected via the logical operators **AND**, **OR**, and **NOT** and is defined recursively by Rules 1 and 2 as follows:

- **Rule 1:** Every atom is a formula.

¹³Also called a **well-formed formula**, or **WFF**, in mathematical logic.

- **Rule 2:** If F_1 and F_2 are formulas, then so are $(F_1 \text{ AND } F_2)$, $(F_1 \text{ OR } F_2)$, $\text{NOT}(F_1)$, and $\text{NOT}(F_2)$. The truth values of these formulas are derived from their component formulas F_1 and F_2 as follows:
 - a. $(F_1 \text{ AND } F_2)$ is TRUE if both F_1 and F_2 are TRUE; otherwise, it is FALSE.
 - b. $(F_1 \text{ OR } F_2)$ is FALSE if both F_1 and F_2 are FALSE; otherwise, it is TRUE.
 - c. $\text{NOT}(F_1)$ is TRUE if F_1 is FALSE; it is FALSE if F_1 is TRUE.
 - d. $\text{NOT}(F_2)$ is TRUE if F_2 is FALSE; it is FALSE if F_2 is TRUE.

6.6.3 The Existential and Universal Quantifiers

In addition, two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** (\forall) and the **existential quantifier** (\exists). Truth values for formulas with quantifiers are described in Rules 3 and 4 below; first, however, we need to define the concepts of free and bound tuple variables in a formula. Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an $(\exists t)$ or $(\forall t)$ clause; otherwise, it is free. Formally, we define a tuple variable in a formula as **free** or **bound** according to the following rules:

- An occurrence of a tuple variable in a formula F that is *an atom* is free in F .
- An occurrence of a tuple variable t is free or bound in a formula made up of logical connectives— $(F_1 \text{ AND } F_2)$, $(F_1 \text{ OR } F_2)$, $\text{NOT}(F_1)$, and $\text{NOT}(F_2)$ —depending on whether it is free or bound in F_1 or F_2 (if it occurs in either). Notice that in a formula of the form $F = (F_1 \text{ AND } F_2)$ or $F = (F_1 \text{ OR } F_2)$, a tuple variable may be free in F_1 and bound in F_2 , or vice versa; in this case, one occurrence of the tuple variable is bound and the other is free in F .
- All *free* occurrences of a tuple variable t in F are **bound** in a formula F' of the form $F' = (\exists t)(F)$ or $F' = (\forall t)(F)$. The tuple variable is bound to the quantifier specified in F' . For example, consider the following formulas:

$F_1 : d.\text{Dname} = \text{'Research'}$

$F_2 : (\exists t)(d.\text{Dnumber} = t.\text{Dno})$

$F_3 : (\forall d)(d.\text{Mgr_ssn} = \text{'333445555'})$

The tuple variable d is free in both F_1 and F_2 , whereas it is bound to the (\forall) quantifier in F_3 . Variable t is bound to the (\exists) quantifier in F_2 .

We can now give Rules 3 and 4 for the definition of a formula we started earlier:

- **Rule 3:** If F is a formula, then so is $(\exists t)(F)$, where t is a tuple variable. The formula $(\exists t)(F)$ is TRUE if the formula F evaluates to TRUE for *some* (at least one) tuple assigned to free occurrences of t in F ; otherwise, $(\exists t)(F)$ is FALSE.
- **Rule 4:** If F is a formula, then so is $(\forall t)(F)$, where t is a tuple variable. The formula $(\forall t)(F)$ is TRUE if the formula F evaluates to TRUE for *every tuple* (in the universe) assigned to free occurrences of t in F ; otherwise, $(\forall t)(F)$ is FALSE.

The (\exists) quantifier is called an existential quantifier because a formula $(\exists t)(F)$ is TRUE if *there exists* some tuple that makes F TRUE. For the universal quantifier,

$(\forall t)(F)$ is TRUE if every possible tuple that can be assigned to free occurrences of t in F is substituted for t , and F is TRUE for *every such substitution*. It is called the universal or *for all* quantifier because every tuple in *the universe of tuples* must make F TRUE to make the quantified formula TRUE.

6.6.4 Sample Queries in Tuple Relational Calculus

We will use some of the same queries from Section 6.5 to give a flavor of how the same queries are specified in relational algebra and in relational calculus. Notice that some queries are easier to specify in the relational algebra than in the relational calculus, and vice versa.

Query 1. List the name and address of all employees who work for the ‘Research’ department.

Q1: $\{t.Fname, t.Lname, t.Address \mid \text{EMPLOYEE}(t) \text{ AND } (\exists d)(\text{DEPARTMENT}(d) \text{ AND } d.Dname = \text{‘Research’ AND } d.Dnumber = t.Dno)\}$

The *only free tuple variables* in a tuple relational calculus expression should be those that appear to the left of the bar (\mid). In Q1, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified after the bar in Q1, the attributes Fname, Lname, and Address are retrieved for each such tuple. The conditions $\text{EMPLOYEE}(t)$ and $\text{DEPARTMENT}(d)$ specify the range relations for t and d . The condition $d.Dname = \text{‘Research’}$ is a **selection condition** and corresponds to a SELECT operation in the relational algebra, whereas the condition $d.Dnumber = t.Dno$ is a **join condition** and is similar in purpose to the (INNER) JOIN operation (see Section 6.3).

Query 2. For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, birth date, and address.

Q2: $\{p.Pnumber, p.Dnum, m.Lname, m.Bdate, m.Address \mid \text{PROJECT}(p) \text{ AND EMPLOYEE}(m) \text{ AND } p.Plocation = \text{‘Stafford’ AND } ((\exists d)(\text{DEPARTMENT}(d) \text{ AND } p.Dnum = d.Dnumber \text{ AND } d.Mgr_ssn = m.Ssn))\}$

In Q2 there are two free tuple variables, p and m . Tuple variable d is bound to the existential quantifier. The query condition is evaluated for every combination of tuples assigned to p and m , and out of all possible combinations of tuples to which p and m are bound, only the combinations that satisfy the condition are selected.

Several tuple variables in a query can range over the same relation. For example, to specify Q8—for each employee, retrieve the employee’s first and last name and the first and last name of his or her immediate supervisor—we specify two tuple variables e and s that both range over the EMPLOYEE relation:

Q8: $\{e.Fname, e.Lname, s.Fname, s.Lname \mid \text{EMPLOYEE}(e) \text{ AND EMPLOYEE}(s) \text{ AND } e.Super_ssn = s.Ssn\}$

Query 3’. List the name of each employee who works on *some* project controlled by department number 5. This is a variation of Q3 in which *all* is

changed to *some*. In this case we need two join conditions and two existential quantifiers.

Q0': $\{e.Lname, e.Fname \mid \text{EMPLOYEE}(e) \text{ AND } ((\exists x)(\exists w)(\text{PROJECT}(x) \text{ AND } \text{WORKS_ON}(w) \text{ AND } x.Dnum=5 \text{ AND } w.Essn=e.Ssn \text{ AND } x.Pnumber=w.Pno)))\}$

Query 4. Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as manager of the controlling department for the project.

Q4: $\{ p.Pnumber \mid \text{PROJECT}(p) \text{ AND } (((\exists e)(\exists w)(\text{EMPLOYEE}(e) \text{ AND } \text{WORKS_ON}(w) \text{ AND } w.Pno=p.Pnumber \text{ AND } e.Lname='Smith' \text{ AND } e.Ssn=w.Essn)) \text{ OR } ((\exists m)(\exists d)(\text{EMPLOYEE}(m) \text{ AND } \text{DEPARTMENT}(d) \text{ AND } p.Dnum=d.Dnumber \text{ AND } d.Mgr_ssn=m.Ssn \text{ AND } m.Lname='Smith'))))\}$

Compare this with the relational algebra version of this query in Section 6.5. The UNION operation in relational algebra can usually be substituted with an OR connective in relational calculus.

6.6.5 Notation for Query Graphs

In this section we describe a notation that has been proposed to represent relational calculus queries that do not involve complex quantification in a graphical form. These types of queries are known as **select-project-join queries**, because they only involve these three relational algebra operations. The notation may be expanded to more general queries, but we do not discuss these extensions here. This graphical representation of a query is called a **query graph**. Figure 6.13 shows the query graph for Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges** (the lines that connect the nodes), as shown in Figure 6.13. Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

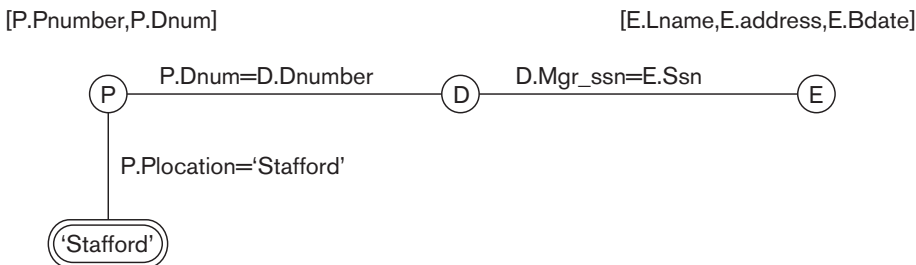


Figure 6.13
Query graph for Q2.

The query graph representation does not indicate a particular order to specify which operations to perform first, and is hence a more neutral representation of a select-project-join query than the query tree representation (see Section 6.3.5), where the order of execution is implicitly specified. There is only a single query graph corresponding to each query. Although some query optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

In the next section we discuss the relationship between the universal and existential quantifiers and show how one can be transformed into the other.

6.6.6 Transforming the Universal and Existential Quantifiers

We now introduce some well-known transformations from mathematical logic that relate the universal and existential quantifiers. It is possible to transform a universal quantifier into an existential quantifier, and vice versa, to get an equivalent expression. One general transformation can be described informally as follows: Transform one type of quantifier into the other with negation (preceded by **NOT**); **AND** and **OR** replace one another; a negated formula becomes unnegated; and an unnegated formula becomes negated. Some special cases of this transformation can be stated as follows, where the \equiv symbol stands for **equivalent to**:

$$\begin{aligned}
 (\forall x) (P(x)) &\equiv \text{NOT } (\exists x) (\text{NOT } (P(x))) \\
 (\exists x) (P(x)) &\equiv \text{NOT } (\forall x) (\text{NOT } (P(x))) \\
 (\forall x) (P(x) \text{ AND } Q(x)) &\equiv \text{NOT } (\exists x) (\text{NOT } (P(x)) \text{ OR } \text{NOT } (Q(x))) \\
 (\forall x) (P(x) \text{ OR } Q(x)) &\equiv \text{NOT } (\exists x) (\text{NOT } (P(x)) \text{ AND } \text{NOT } (Q(x))) \\
 (\exists x) (P(x) \text{ OR } Q(x)) &\equiv \text{NOT } (\forall x) (\text{NOT } (P(x)) \text{ AND } \text{NOT } (Q(x))) \\
 (\exists x) (P(x) \text{ AND } Q(x)) &\equiv \text{NOT } (\forall x) (\text{NOT } (P(x)) \text{ OR } \text{NOT } (Q(x)))
 \end{aligned}$$

Notice also that the following is TRUE, where the \Rightarrow symbol stands for **implies**:

$$\begin{aligned}
 (\forall x)(P(x)) &\Rightarrow (\exists x)(P(x)) \\
 \text{NOT } (\exists x)(P(x)) &\Rightarrow \text{NOT } (\forall x)(P(x))
 \end{aligned}$$

6.6.7 Using the Universal Quantifier in Queries

Whenever we use a universal quantifier, it is quite judicious to follow a few rules to ensure that our expression makes sense. We discuss these rules with respect to the query Q3.

Query 3. List the names of employees who work on *all* the projects controlled by department number 5. One way to specify this query is to use the universal quantifier as shown:

Q3: $\{e.\text{Lname}, e.\text{Fname} \mid \text{EMPLOYEE}(e) \text{ AND } ((\forall x)(\text{NOT}(\text{PROJECT}(x)) \text{ OR } \text{NOT } (x.\text{Dnum}=5) \text{ OR } ((\exists w)(\text{WORKS_ON}(w) \text{ AND } w.\text{Essn}=e.\text{Ssn} \text{ AND } x.\text{Pnumber}=w.\text{Pno}))))))\}$

We can break up Q3 into its basic components as follows:

Q3: $\{e.\text{Lname}, e.\text{Fname} \mid \text{EMPLOYEE}(e) \text{ AND } F'\}$
 $F' = ((\forall x)(\text{NOT}(\text{PROJECT}(x)) \text{ OR } F_1))$
 $F_1 = \text{NOT}(x.\text{Dnum}=5) \text{ OR } F_2$
 $F_2 = ((\exists w)(\text{WORKS_ON}(w) \text{ AND } w.\text{Essn}=e.\text{Ssn}$
 $\text{AND } x.\text{Pnumber}=w.\text{Pno}))$

We want to make sure that a selected employee e works on *all the projects* controlled by department 5, but the *definition of universal quantifier* says that to make the quantified formula TRUE, *the inner formula* must be TRUE *for all tuples in the universe*. The trick is to exclude from the universal quantification all tuples that we are not interested in by making the condition TRUE *for all such tuples*. This is necessary because a universally quantified tuple variable, such as x in Q3, must evaluate to TRUE *for every possible tuple* assigned to it to make the quantified formula TRUE.

The first tuples to exclude (by making them evaluate automatically to TRUE) are those that are not in the relation R of interest. In Q3, using the expression $\text{NOT}(\text{PROJECT}(x))$ inside the universally quantified formula evaluates to TRUE all tuples x that are not in the PROJECT relation. Then we exclude the tuples we are not interested in from R itself. In Q3, using the expression $\text{NOT}(x.\text{Dnum}=5)$ evaluates to TRUE all tuples x that are in the PROJECT relation but are not controlled by department 5. Finally, we specify a condition F_2 that must hold on all the remaining tuples in R . Hence, we can explain Q3 as follows:

1. For the formula $F' = (\forall x)(F)$ to be TRUE, we must have the formula F be TRUE *for all tuples in the universe that can be assigned to x* . However, in Q3 we are only interested in F being TRUE for all tuples of the PROJECT relation that are controlled by department 5. Hence, the formula F is of the form $(\text{NOT}(\text{PROJECT}(x)) \text{ OR } F_1)$. The ' $\text{NOT}(\text{PROJECT}(x)) \text{ OR } \dots$ ' condition is TRUE for all tuples *not in the PROJECT relation* and has the effect of eliminating these tuples from consideration in the truth value of F_1 . For every tuple in the PROJECT relation, F_1 must be TRUE if F' is to be TRUE.
2. Using the same line of reasoning, we do not want to consider tuples in the PROJECT relation that are not controlled by department number 5, since we are only interested in PROJECT tuples whose $\text{Dnum}=5$. Therefore, we can write:

IF $(x.\text{Dnum}=5)$ **THEN** F_2

which is equivalent to

(NOT $(x.\text{Dnum}=5)$ **OR** $F_2)$

3. Formula F_1 , hence, is of the form $\text{NOT}(x.\text{Dnum}=5) \text{ OR } F_2$. In the context of Q3, this means that, for a tuple x in the PROJECT relation, either its $\text{Dnum} \neq 5$ or it must satisfy F_2 .
4. Finally, F_2 gives the condition that we want to hold for a selected EMPLOYEE tuple: that the employee works on *every PROJECT tuple that has not been excluded yet*. Such employee tuples are selected by the query.

In English, Q3 gives the following condition for selecting an EMPLOYEE tuple e : For every tuple x in the PROJECT relation with $x.Dnum=5$, there must exist a tuple w in WORKS_ON such that $w.Essn=e.Ssn$ and $w.Pno=x.Pnumber$. This is equivalent to saying that EMPLOYEE e works on every PROJECT x in DEPARTMENT number 5. (Whew!)

Using the general transformation from universal to existential quantifiers given in Section 6.6.6, we can rephrase the query in Q3 as shown in Q3A, which uses a negated existential quantifier instead of the universal quantifier:

Q3A: $\{e.Lname, e.Fname \mid \text{EMPLOYEE}(e) \text{ AND } (\text{NOT } (\exists x) (\text{PROJECT}(x) \text{ AND } (x.Dnum=5) \text{ AND } (\text{NOT } (\exists w)(\text{WORKS_ON}(w) \text{ AND } w.Essn=e.Ssn \text{ AND } x.Pnumber=w.Pno)))))\}$

We now give some additional examples of queries that use quantifiers.

Query 6. List the names of employees who have no dependents.

Q6: $\{e.Fname, e.Lname \mid \text{EMPLOYEE}(e) \text{ AND } (\text{NOT } (\exists d)(\text{DEPENDENT}(d) \text{ AND } e.Ssn=d.Essn))\}$

Using the general transformation rule, we can rephrase Q6 as follows:

Q6A: $\{e.Fname, e.Lname \mid \text{EMPLOYEE}(e) \text{ AND } ((\forall d)(\text{NOT}(\text{DEPENDENT}(d) \text{ OR NOT}(e.Ssn=d.Essn))))\}$

Query 7. List the names of managers who have at least one dependent.

Q7: $\{e.Fname, e.Lname \mid \text{EMPLOYEE}(e) \text{ AND } ((\exists d)(\exists p)(\text{DEPARTMENT}(d) \text{ AND } \text{DEPENDENT}(p) \text{ AND } e.Ssn=d.Mgr_ssn \text{ AND } p.Essn=e.Ssn))\}$

This query is handled by interpreting *managers who have at least one dependent* as *managers for whom there exists some dependent*.

6.6.8 Safe Expressions

Whenever we use universal quantifiers, existential quantifiers, or negation of predicates in a calculus expression, we must make sure that the resulting expression makes sense. A **safe expression** in relational calculus is one that is guaranteed to yield a *finite number of tuples* as its result; otherwise, the expression is called **unsafe**. For example, the expression

$\{t \mid \text{NOT } (\text{EMPLOYEE}(t))\}$

is *unsafe* because it yields all tuples in the universe that are *not* EMPLOYEE tuples, which are infinitely numerous. If we follow the rules for Q3 discussed earlier, we will get a safe expression when using universal quantifiers. We can define safe expressions more precisely by introducing the concept of the *domain of a tuple relational calculus expression*: This is the set of all values that either appear as constant values in the expression or exist in any tuple in the relations referenced in the expression. For example, the domain of $\{t \mid \text{NOT}(\text{EMPLOYEE}(t))\}$ is the set of all attribute values appearing in some tuple of the EMPLOYEE relation (for any attribute). The domain

of the expression Q3A would include all values appearing in EMPLOYEE, PROJECT, and WORKS_ON (unioned with the value 5 appearing in the query itself).

An expression is said to be **safe** if all values in its result are from the domain of the expression. Notice that the result of $\{t \mid \text{NOT}(\text{EMPLOYEE}(t))\}$ is unsafe, since it will, in general, include tuples (and hence values) from outside the EMPLOYEE relation; such values are not in the domain of the expression. All of our other examples are safe expressions.

6.7 The Domain Relational Calculus

There is another type of relational calculus called the domain relational calculus, or simply, **domain calculus**. Historically, while SQL (see Chapters 4 and 5), which was based on tuple relational calculus, was being developed by IBM Research at San Jose, California, another language called QBE (Query-By-Example), which is related to domain calculus, was being developed almost concurrently at the IBM T.J. Watson Research Center in Yorktown Heights, New York. The formal specification of the domain calculus was proposed after the development of the QBE language and system.

Domain calculus differs from tuple calculus in the *type of variables* used in formulas: Rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree n for a query result, we must have n of these **domain variables**—one for each attribute. An expression of the domain calculus is of the form

$$\{x_1, x_2, \dots, x_n \mid \text{COND}(x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m})\}$$

where $x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m}$ are domain variables that range over domains (of attributes), and COND is a **condition** or **formula** of the domain relational calculus.

A formula is made up of **atoms**. The atoms of a formula are slightly different from those for the tuple calculus and can be one of the following:

1. An atom of the form $R(x_1, x_2, \dots, x_j)$, where R is the name of a relation of degree j and each x_i , $1 \leq i \leq j$, is a domain variable. This atom states that a list of values of $\langle x_1, x_2, \dots, x_j \rangle$ must be a tuple in the relation whose name is R , where x_i is the value of the i th attribute value of the tuple. To make a domain calculus expression more concise, we can *drop the commas* in a list of variables; thus, we can write:

$$\{x_1, x_2, \dots, x_n \mid R(x_1 x_2 x_3) \text{ AND } \dots\}$$

instead of:

$$\{x_1, x_2, \dots, x_n \mid R(x_1, x_2, x_3) \text{ AND } \dots\}$$

2. An atom of the form $x_i \text{ op } x_j$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, and x_i and x_j are domain variables.
3. An atom of the form $x_i \text{ op } c$ or $c \text{ op } x_j$, where **op** is one of the comparison operators in the set $\{=, <, \leq, >, \geq, \neq\}$, x_i and x_j are domain variables, and c is a constant value.

As in tuple calculus, atoms evaluate to either TRUE or FALSE for a specific set of values, called the **truth values** of the atoms. In case 1, if the domain variables are assigned values corresponding to a tuple of the specified relation R , then the atom is TRUE. In cases 2 and 3, if the domain variables are assigned values that satisfy the condition, then the atom is TRUE.

In a similar way to the tuple relational calculus, formulas are made up of atoms, variables, and quantifiers, so we will not repeat the specifications for formulas here. Some examples of queries specified in the domain calculus follow. We will use lowercase letters l, m, n, \dots, x, y, z for domain variables.

Query 0. List the birth date and address of the employee whose name is ‘John B. Smith’.

Q0: $\{u, v \mid (\exists q) (\exists r) (\exists s) (\exists t) (\exists w) (\exists x) (\exists y) (\exists z)$
 $(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } q=\text{'John'} \text{ AND } r=\text{'B'} \text{ AND } s=\text{'Smith'})\}$

We need ten variables for the EMPLOYEE relation, one to range over each of the domains of attributes of EMPLOYEE in order. Of the ten variables q, r, s, \dots, z , only u and v are free, because they appear to the left of the bar and hence should not be bound to a quantifier. We first specify the *requested attributes*, Bdate and Address, by the free domain variables u for BDATE and v for ADDRESS. Then we specify the condition for selecting a tuple following the bar (\mid)—namely, that the sequence of values assigned to the variables $qrstuvwxyz$ be a tuple of the EMPLOYEE relation and that the values for q (Fname), r (Minit), and s (Lname) be equal to ‘John’, ‘B’, and ‘Smith’, respectively. For convenience, we will quantify only those variables *actually appearing in a condition* (these would be q, r , and s in Q0) in the rest of our examples.¹⁴

An alternative shorthand notation, used in QBE, for writing this query is to assign the constants ‘John’, ‘B’, and ‘Smith’ directly as shown in Q0A. Here, all variables not appearing to the left of the bar are implicitly existentially quantified:¹⁵

Q0A: $\{u, v \mid \text{EMPLOYEE}(\text{'John'}, \text{'B'}, \text{'Smith'}, t, u, v, w, x, y, z) \}$

Query 1. Retrieve the name and address of all employees who work for the ‘Research’ department.

Q1: $\{q, s, v \mid (\exists z) (\exists l) (\exists m) (\text{EMPLOYEE}(qrstuvwxyz) \text{ AND}$
 $\text{DEPARTMENT}(lmno) \text{ AND } l=\text{'Research'} \text{ AND } m=z)\}$

A condition relating two domain variables that range over attributes from two relations, such as $m = z$ in Q1, is a **join condition**, whereas a condition that relates a domain variable to a constant, such as $l = \text{'Research'}$, is a **selection condition**.

¹⁴Note that the notation of quantifying only the domain variables actually used in conditions and of showing a predicate such as $\text{EMPLOYEE}(qrstuvwxyz)$ without separating domain variables with commas is an abbreviated notation used for convenience; it is not the correct formal notation.

¹⁵Again, this is not a formally accurate notation.

Query 2. For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, birth date, and address.

Q2: $\{i, k, s, u, v \mid (\exists j)(\exists m)(\exists n)(\exists t)(\text{PROJECT}(hijk) \text{ AND } \text{EMPLOYEE}(qrstuvwxyz) \text{ AND } \text{DEPARTMENT}(lmno) \text{ AND } k=m \text{ AND } n=t \text{ AND } j=\text{'Stafford'})\}$

Query 6. List the names of employees who have no dependents.

Q6: $\{q, s \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } (\text{NOT}(\exists l)(\text{DEPENDENT}(lmnop) \text{ AND } t=l))))\}$

Q6 can be restated using universal quantifiers instead of the existential quantifiers, as shown in Q6A:

Q6A: $\{q, s \mid (\exists t)(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } ((\forall l)(\text{NOT}(\text{DEPENDENT}(lmnop)) \text{ OR } \text{NOT}(t=l))))\}$

Query 7. List the names of managers who have at least one dependent.

Q7: $\{s, q \mid (\exists t)(\exists j)(\exists l)(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } \text{DEPARTMENT}(hijk) \text{ AND } \text{DEPENDENT}(lmnop) \text{ AND } t=j \text{ AND } l=t)\}$

As we mentioned earlier, it can be shown that any query that can be expressed in the basic relational algebra can also be expressed in the domain or tuple relational calculus. Also, any *safe expression* in the domain or tuple relational calculus can be expressed in the basic relational algebra.

The QBE language was based on the domain relational calculus, although this was realized later, after the domain calculus was formalized. QBE was one of the first graphical query languages with minimum syntax developed for database systems. It was developed at IBM Research and is available as an IBM commercial product as part of the Query Management Facility (QMF) interface option to DB2. The basic ideas used in QBE have been applied in several other commercial products. Because of its important place in the history of relational languages, we have included an overview of QBE in Appendix C.

6.8 Summary

In this chapter we presented two formal languages for the relational model of data. They are used to manipulate relations and produce new relations as answers to queries. We discussed the relational algebra and its operations, which are used to specify a sequence of operations to specify a query. Then we introduced two types of relational calculi called tuple calculus and domain calculus.

In Sections 6.1 through 6.3, we introduced the basic relational algebra operations and illustrated the types of queries for which each is used. First, we discussed the unary relational operators SELECT and PROJECT, as well as the RENAME operation. Then, we discussed binary set theoretic operations requiring that relations on which they

are applied be union (or type) compatible; these include UNION, INTERSECTION, and SET DIFFERENCE. The CARTESIAN PRODUCT operation is a set operation that can be used to combine tuples from two relations, producing all possible combinations. It is rarely used in practice; however, we showed how CARTESIAN PRODUCT followed by SELECT can be used to define matching tuples from two relations and leads to the JOIN operation. Different JOIN operations called THETA JOIN, EQUIJOIN, and NATURAL JOIN were introduced. Query trees were introduced as a graphical representation of relational algebra queries, which can also be used as the basis for internal data structures that the DBMS can use to represent a query.

We discussed some important types of queries that *cannot* be stated with the basic relational algebra operations but are important for practical situations. We introduced GENERALIZED PROJECTION to use functions of attributes in the projection list and the AGGREGATE FUNCTION operation to deal with aggregate types of statistical requests that summarize the information in the tables. We discussed recursive queries, for which there is no direct support in the algebra but which can be handled in a step-by-step approach, as we demonstrated. Then we presented the OUTER JOIN and OUTER UNION operations, which extend JOIN and UNION and allow all information in source relations to be preserved in the result.

The last two sections described the basic concepts behind relational calculus, which is based on the branch of mathematical logic called predicate calculus. There are two types of relational calculi: (1) the tuple relational calculus, which uses tuple variables that range over tuples (rows) of relations, and (2) the domain relational calculus, which uses domain variables that range over domains (columns of relations). In relational calculus, a query is specified in a single declarative statement, without specifying any order or method for retrieving the query result. Hence, relational calculus is often considered to be a higher-level *declarative* language than the relational algebra, because a relational calculus expression states *what* we want to retrieve regardless of *how* the query may be executed.

We discussed the syntax of relational calculus queries using both tuple and domain variables. We introduced query graphs as an internal representation for queries in relational calculus. We also discussed the existential quantifier (\exists) and the universal quantifier (\forall). We saw that relational calculus variables are bound by these quantifiers. We described in detail how queries with universal quantification are written, and we discussed the problem of specifying safe queries whose results are finite. We also discussed rules for transforming universal into existential quantifiers, and vice versa. It is the quantifiers that give expressive power to the relational calculus, making it equivalent to the basic relational algebra. There is no analog to grouping and aggregation functions in basic relational calculus, although some extensions have been suggested.

Review Questions

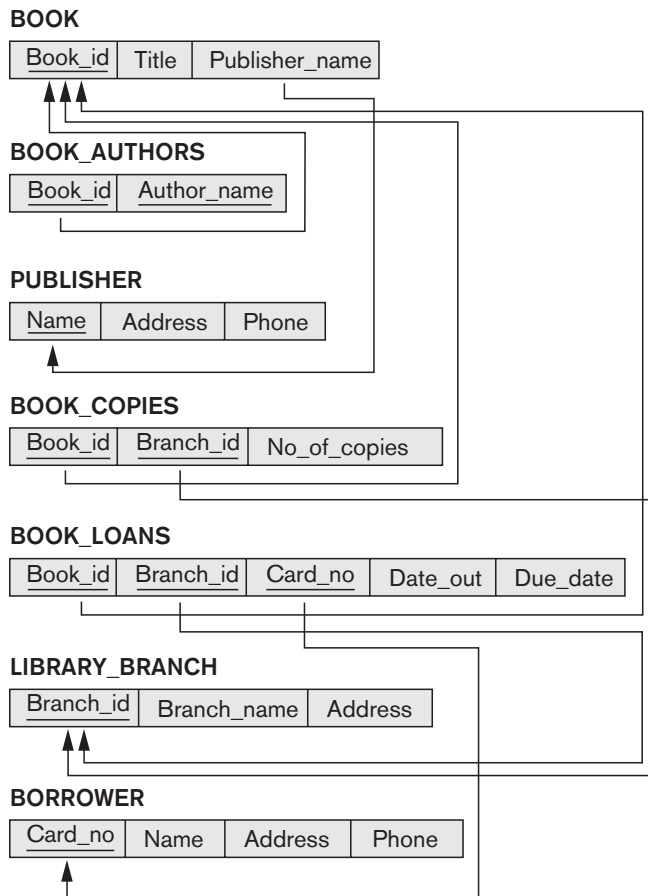
- 6.1. List the operations of relational algebra and the purpose of each.

- 6.2. What is union compatibility? Why do the UNION, INTERSECTION, and DIFFERENCE operations require that the relations on which they are applied be union compatible?
- 6.3. Discuss some types of queries for which renaming of attributes is necessary in order to specify the query unambiguously.
- 6.4. Discuss the various types of *inner join* operations. Why is theta join required?
- 6.5. What role does the concept of *foreign key* play when specifying the most common types of meaningful join operations?
- 6.6. What is the FUNCTION operation? What is it used for?
- 6.7. How are the OUTER JOIN operations different from the INNER JOIN operations? How is the OUTER UNION operation different from UNION?
- 6.8. In what sense does relational calculus differ from relational algebra, and in what sense are they similar?
- 6.9. How does tuple relational calculus differ from domain relational calculus?
- 6.10. Discuss the meanings of the existential quantifier (\exists) and the universal quantifier (\forall).
- 6.11. Define the following terms with respect to the tuple calculus: *tuple variable*, *range relation*, *atom*, *formula*, and *expression*.
- 6.12. Define the following terms with respect to the domain calculus: *domain variable*, *range relation*, *atom*, *formula*, and *expression*.
- 6.13. What is meant by a *safe expression* in relational calculus?
- 6.14. When is a query language called relationally complete?

Exercises

- 6.15. Show the result of each of the sample queries in Section 6.5 as it would apply to the database state in Figure 3.6.
- 6.16. Specify the following queries on the COMPANY relational database schema shown in Figure 5.5, using the relational operators discussed in this chapter. Also show the result of each query as it would apply to the database state in Figure 3.6.
 - a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the ProductX project.
 - b. List the names of all employees who have a dependent with the same first name as themselves.
 - c. Find the names of all employees who are directly supervised by 'Franklin Wong'.
 - d. For each project, list the project name and the total hours per week (by all employees) spent on that project.

- e. Retrieve the names of all employees who work on every project.
 - f. Retrieve the names of all employees who do not work on any project.
 - g. For each department, retrieve the department name and the average salary of all employees working in that department.
 - h. Retrieve the average salary of all female employees.
 - i. Find the names and addresses of all employees who work on at least one project located in Houston but whose department has no location in Houston.
 - j. List the last names of all department managers who have no dependents.
- 6.17.** Consider the AIRLINE relational database schema shown in Figure 3.8, which was described in Exercise 3.12. Specify the following queries in relational algebra:
- a. For each flight, list the flight number, the departure airport for the first leg of the flight, and the arrival airport for the last leg of the flight.
 - b. List the flight numbers and weekdays of all flights or flight legs that depart from Houston Intercontinental Airport (airport code 'IAH') and arrive in Los Angeles International Airport (airport code 'LAX').
 - c. List the flight number, departure airport code, scheduled departure time, arrival airport code, scheduled arrival time, and weekdays of all flights or flight legs that depart from some airport in the city of Houston and arrive at some airport in the city of Los Angeles.
 - d. List all fare information for flight number 'CO197'.
 - e. Retrieve the number of available seats for flight number 'CO197' on '2009-10-09'.
- 6.18.** Consider the LIBRARY relational database schema shown in Figure 6.14, which is used to keep track of books, borrowers, and book loans. Referential integrity constraints are shown as directed arcs in Figure 6.14, as in the notation of Figure 3.7. Write down relational expressions for the following queries:
- a. How many copies of the book titled *The Lost Tribe* are owned by the library branch whose name is 'Sharpstown'?
 - b. How many copies of the book titled *The Lost Tribe* are owned by each library branch?
 - c. Retrieve the names of all borrowers who do not have any books checked out.
 - d. For each book that is loaned out from the Sharpstown branch and whose Due_date is today, retrieve the book title, the borrower's name, and the borrower's address.
 - e. For each library branch, retrieve the branch name and the total number of books loaned out from that branch.

**Figure 6.14**

A relational database schema for a LIBRARY database.

- f. Retrieve the names, addresses, and number of books checked out for all borrowers who have more than five books checked out.
 - g. For each book authored (or coauthored) by Stephen King, retrieve the title and the number of copies owned by the library branch whose name is Central.
- 6.19.** Specify the following queries in relational algebra on the database schema given in Exercise 3.14:
- a. List the Order# and Ship_date for all orders shipped from Warehouse# W2.
 - b. List the WAREHOUSE information from which the CUSTOMER named Jose Lopez was supplied his orders. Produce a listing: Order#, Warehouse#.

- c. Produce a listing Cname, No_of_orders, Avg_order_amt, where the middle column is the total number of orders by the customer and the last column is the average order amount for that customer.
 - d. List the orders that were not shipped within 30 days of ordering.
 - e. List the Order# for orders that were shipped from *all* warehouses that the company has in New York.
- 6.20.** Specify the following queries in relational algebra on the database schema given in Exercise 3.15:
- a. Give the details (all attributes of trip relation) for trips that exceeded \$2,000 in expenses.
 - b. Print the Ssns of salespeople who took trips to Honolulu.
 - c. Print the total trip expenses incurred by the salesperson with SSN = '234-56-7890'.
- 6.21.** Specify the following queries in relational algebra on the database schema given in Exercise 3.16:
- a. List the number of courses taken by all students named John Smith in Winter 2009 (i.e., Quarter=W09).
 - b. Produce a list of textbooks (include Course#, Book_isbn, Book_title) for courses offered by the 'CS' department that have used more than two books.
 - c. List any department that has all its adopted books published by 'Pearson Publishing'.
- 6.22.** Consider the two tables *T1* and *T2* shown in Figure 6.15. Show the results of the following operations:
- a. $T1 \bowtie_{T1.P = T2.A} T2$
 - b. $T1 \bowtie_{T1.Q = T2.B} T2$
 - c. $T1 \bowtie_{T1.P = T2.A} T2$
 - d. $T1 \bowtie_{T1.Q = T2.B} T2$
 - e. $T1 \cup T2$
 - f. $T1 \bowtie_{(T1.P = T2.A \text{ AND } T1.R = T2.C)} T2$

Figure 6.15
A database state for the relations *T1* and *T2*.

TABLE T1			TABLE T2		
P	Q	R	A	B	C
10	a	5	10	b	6
15	b	8	25	c	3
25	a	6	10	b	5

- 6.23.** Specify the following queries in relational algebra on the database schema in Exercise 3.17:
- For the salesperson named 'Jane Doe', list the following information for all the cars she sold: Serial#, Manufacturer, Sale_price.
 - List the Serial# and Model of cars that have no options.
 - Consider the NATURAL JOIN operation between SALESPERSON and SALE. What is the meaning of a left outer join for these tables (do not change the order of relations)? Explain with an example.
 - Write a query in relational algebra involving selection and one set operation and say in words what the query does.
- 6.24.** Specify queries a, b, c, e, f, i, and j of Exercise 6.16 in both tuple and domain relational calculus.
- 6.25.** Specify queries a, b, c, and d of Exercise 6.17 in both tuple and domain relational calculus.
- 6.26.** Specify queries c, d, and f of Exercise 6.18 in both tuple and domain relational calculus.
- 6.27.** In a tuple relational calculus query with n tuple variables, what would be the typical minimum number of join conditions? Why? What is the effect of having a smaller number of join conditions?
- 6.28.** Rewrite the domain relational calculus queries that followed Q0 in Section 6.7 in the style of the abbreviated notation of Q0A, where the objective is to minimize the number of domain variables by writing constants in place of variables wherever possible.
- 6.29.** Consider this query: Retrieve the Ssns of employees who work on at least those projects on which the employee with Ssn=123456789 works. This may be stated as (**FORALL** x) (**IF** P **THEN** Q), where
- x is a tuple variable that ranges over the PROJECT relation.
 - $P \equiv$ EMPLOYEE with Ssn=123456789 works on PROJECT x .
 - $Q \equiv$ EMPLOYEE e works on PROJECT x .
- Express the query in tuple relational calculus, using the rules
- $(\forall x)(P(x)) \equiv \text{NOT}(\exists x)(\text{NOT}(P(x)))$.
 - $(\text{IF } P \text{ THEN } Q) \equiv (\text{NOT}(P) \text{ OR } Q)$.
- 6.30.** Show how you can specify the following relational algebra operations in both tuple and domain relational calculus.
- $\sigma_{A=C}(R(A, B, C))$
 - $\pi_{\langle A, B \rangle}(R(A, B, C))$
 - $R(A, B, C) * S(C, D, E)$
 - $R(A, B, C) \cup S(A, B, C)$
 - $R(A, B, C) \cap S(A, B, C)$

- f. $R(A, B, C) = S(A, B, C)$
 - g. $R(A, B, C) \times S(D, E, F)$
 - h. $R(A, B) \div S(A)$
- 6.31.** Suggest extensions to the relational calculus so that it may express the following types of operations that were discussed in Section 6.4: (a) aggregate functions and grouping; (b) OUTER JOIN operations; (c) recursive closure queries.
- 6.32.** A nested query is a query within a query. More specifically, a nested query is a parenthesized query whose result can be used as a value in a number of places, such as instead of a relation. Specify the following queries on the database specified in Figure 3.5 using the concept of nested queries and the relational operators discussed in this chapter. Also show the result of each query as it would apply to the database state in Figure 3.6.
- a. List the names of all employees who work in the department that has the employee with the highest salary among all employees.
 - b. List the names of all employees whose supervisor's supervisor has '888665555' for Ssn.
 - c. List the names of employees who make at least \$10,000 more than the employee who is paid the least in the company.
- 6.33.** State whether the following conclusions are true or false:
- a. **NOT** ($P(x)$ **OR** $Q(x)$) \rightarrow (**NOT** ($P(x)$) **AND** (**NOT** ($Q(x)$)))
 - b. **NOT** ($\exists x$) ($P(x)$) $\rightarrow \forall x$ (**NOT** ($P(x)$))
 - c. ($\exists x$) ($P(x)$) $\rightarrow \forall x$ (($P(x)$))

Laboratory Exercises

- 6.34.** Specify and execute the following queries in relational algebra (RA) using the RA interpreter on the COMPANY database schema in Figure 3.5.
- a. List the names of all employees in department 5 who work more than 10 hours per week on the ProductX project.
 - b. List the names of all employees who have a dependent with the same first name as themselves.
 - c. List the names of employees who are directly supervised by Franklin Wong.
 - d. List the names of employees who work on every project.
 - e. List the names of employees who do not work on any project.
 - f. List the names and addresses of employees who work on at least one project located in Houston but whose department has no location in Houston.
 - g. List the names of department managers who have no dependents.

- 6.35.** Consider the following MAILORDER relational schema describing the data for a mail order company.

PARTS(Pno, Pname, Qoh, Price, Olevel)
 CUSTOMERS(Cno, Cname, Street, Zip, Phone)
 EMPLOYEES(Eno, Ename, Zip, Hdate)
 ZIP_CODES(Zip, City)
 ORDERS(Ono, Cno, Eno, Received, Shipped)
 ODETAILS(Ono, Pno, Qty)

Qoh stands for *quantity on hand*: the other attribute names are self-explanatory. Specify and execute the following queries using the RA interpreter on the MAILORDER database schema.

- a. Retrieve the names of parts that cost less than \$20.00.
 - b. Retrieve the names and cities of employees who have taken orders for parts costing more than \$50.00.
 - c. Retrieve the pairs of customer number values of customers who live in the same ZIP Code.
 - d. Retrieve the names of customers who have ordered parts from employees living in Wichita.
 - e. Retrieve the names of customers who have ordered parts costing less than \$20.00.
 - f. Retrieve the names of customers who have not placed an order.
 - g. Retrieve the names of customers who have placed exactly two orders.
- 6.36.** Consider the following GRADEBOOK relational schema describing the data for a grade book of a particular instructor. (*Note*: The attributes A, B, C, and D of COURSES store grade cutoffs.)

CATALOG(Cno, Ctitle)
 STUDENTS(Sid, Fname, Lname, Minit)
 COURSES(Term, Sec_no, Cno, A, B, C, D)
 ENROLLS(Sid, Term, Sec_no)

Specify and execute the following queries using the RA interpreter on the GRADEBOOK database schema.

- a. Retrieve the names of students enrolled in the Automata class during the fall 2009 term.
- b. Retrieve the Sid values of students who have enrolled in CSc226 and CSc227.
- c. Retrieve the Sid values of students who have enrolled in CSc226 or CSc227.
- d. Retrieve the names of students who have not enrolled in any class.
- e. Retrieve the names of students who have enrolled in all courses in the CATALOG table.

6.37. Consider a database that consists of the following relations.

SUPPLIER(Sno, Sname)
 PART(Pno, Pname)
 PROJECT(Jno, Jname)
 SUPPLY(Sno, Pno, Jno)

The database records information about suppliers, parts, and projects and includes a ternary relationship between suppliers, parts, and projects. This relationship is a many-many-many relationship. Specify and execute the following queries using the RA interpreter.

- a. Retrieve the part numbers that are supplied to exactly two projects.
 - b. Retrieve the names of suppliers who supply more than two parts to project 'J1'.
 - c. Retrieve the part numbers that are supplied by every supplier.
 - d. Retrieve the project names that are supplied by supplier 'S1' only.
 - e. Retrieve the names of suppliers who supply at least two different parts each to at least two different projects.
- 6.38.** Specify and execute the following queries for the database in Exercise 3.16 using the RA interpreter.
- a. Retrieve the names of students who have enrolled in a course that uses a textbook published by Addison-Wesley.
 - b. Retrieve the names of courses in which the textbook has been changed at least once.
 - c. Retrieve the names of departments that adopt textbooks published by Addison-Wesley only.
 - d. Retrieve the names of departments that adopt textbooks written by Navathe and published by Addison-Wesley.
 - e. Retrieve the names of students who have never used a book (in a course) written by Navathe and published by Addison-Wesley.
- 6.39.** Repeat Laboratory Exercises 6.34 through 6.38 in domain relational calculus (DRC) by using the DRC interpreter.

Selected Bibliography

Codd (1970) defined the basic relational algebra. Date (1983a) discusses outer joins. Work on extending relational operations is discussed by Carlis (1986) and Ozsoyoglu et al. (1985). Cammarata et al. (1989) extends the relational model integrity constraints and joins.

Codd (1971) introduced the language Alpha, which is based on concepts of tuple relational calculus. Alpha also includes the notion of aggregate functions, which goes beyond relational calculus. The original formal definition of relational calculus

was given by Codd (1972), which also provided an algorithm that transforms any tuple relational calculus expression to relational algebra. The QUEL (Stonebraker et al. 1976) is based on tuple relational calculus, with implicit existential quantifiers, but no universal quantifiers, and was implemented in the INGRES system as a commercially available language. Codd defined relational completeness of a query language to mean at least as powerful as relational calculus. Ullman (1988) describes a formal proof of the equivalence of relational algebra with the safe expressions of tuple and domain relational calculus. Abiteboul et al. (1995) and Atzeni and deAntonellis (1993) give a detailed treatment of formal relational languages.

Although ideas of domain relational calculus were initially proposed in the QBE language (Zloof 1975), the concept was formally defined by Lacroix and Pirotte (1977a). The experimental version of the Query-By-Example system is described in Zloof (1975). The ILL (Lacroix and Pirotte 1977b) is based on domain relational calculus. Whang et al. (1990) extends QBE with universal quantifiers. Visual query languages, of which QBE is an example, are being proposed as a means of querying databases; conferences such as the Visual Database Systems Working Conference (e.g., Arisawa and Catarci (2000) or Zhou and Pu (2002)) have a number of proposals for such languages.

This page intentionally left blank

part 3

Conceptual Modeling and Database Design

This page intentionally left blank

Data Modeling Using the Entity-Relationship (ER) Model

Conceptual modeling is a very important phase in designing a successful database application. Generally, the term **database application** refers to a particular database and the associated programs that implement the database queries and updates. For example, a **BANK** database application that keeps track of customer accounts would include programs that implement database updates corresponding to customer deposits and withdrawals. These programs provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus for the end users of the application—the bank tellers, in this example. Hence, a major part of the database application will require the design, implementation, and testing of these application programs. Traditionally, the design and testing of **application programs** has been considered to be part of *software engineering* rather than *database design*. In many software design tools, the database design methodologies and software engineering methodologies are intertwined since these activities are strongly related.

In this chapter, we follow the traditional approach of concentrating on the database structures and constraints during conceptual database design. The design of application programs is typically covered in software engineering courses. We present the modeling concepts of the **Entity-Relationship (ER) model**, which is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. We describe the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications. We also present the diagrammatic notation associated with the ER model, known as **ER diagrams**.

Object modeling methodologies such as the **Unified Modeling Language (UML)** are becoming increasingly popular in both database and software design. These methodologies go beyond database design to specify detailed design of software modules and their interactions using various types of diagrams. An important part of these methodologies—namely, *class diagrams*¹—are similar in many ways to the ER diagrams. In class diagrams, *operations* on objects are specified, in addition to specifying the database schema structure. Operations can be used to specify the *functional requirements* during database design, as we will discuss in Section 7.1. We present some of the UML notation and concepts for class diagrams that are particularly relevant to database design in Section 7.8, and briefly compare these to ER notation and concepts. Additional UML notation and concepts are presented in Section 8.6 and in Chapter 10.

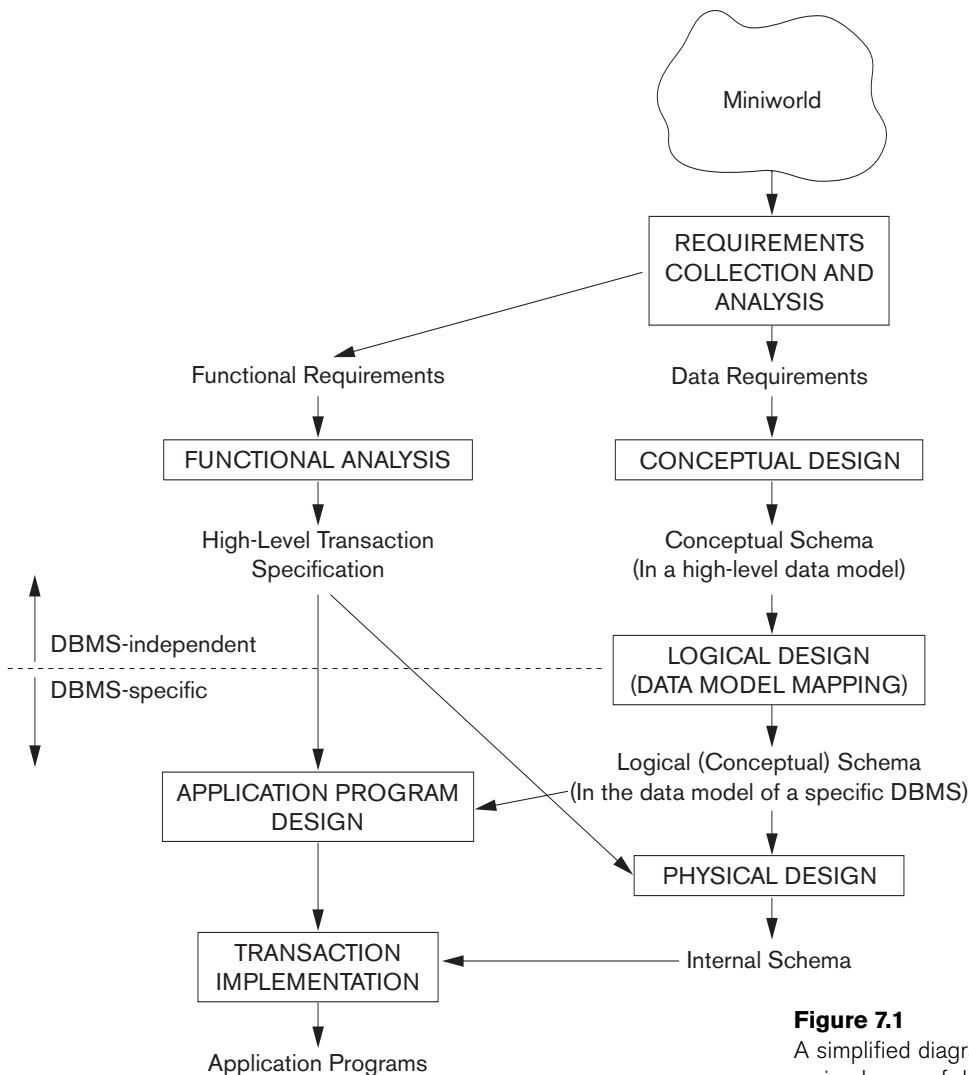
This chapter is organized as follows: Section 7.1 discusses the role of high-level conceptual data models in database design. We introduce the requirements for a sample database application in Section 7.2 to illustrate the use of concepts from the ER model. This sample database is also used throughout the book. In Section 7.3 we present the concepts of entities and attributes, and we gradually introduce the diagrammatic technique for displaying an ER schema. In Section 7.4 we introduce the concepts of binary relationships and their roles and structural constraints. Section 7.5 introduces weak entity types. Section 7.6 shows how a schema design is refined to include relationships. Section 7.7 reviews the notation for ER diagrams, summarizes the issues and common pitfalls that occur in schema design, and discusses how to choose the names for database schema constructs. Section 7.8 introduces some UML class diagram concepts, compares them to ER model concepts, and applies them to the same database example. Section 7.9 discusses more complex types of relationships. Section 7.10 summarizes the chapter.

The material in Sections 7.8 and 7.9 may be excluded from an introductory course. If a more thorough coverage of data modeling concepts and conceptual database design is desired, the reader should continue to Chapter 8, where we describe extensions to the ER model that lead to the Enhanced-ER (EER) model, which includes concepts such as specialization, generalization, inheritance, and union types (categories). We also introduce some additional UML concepts and notation in Chapter 8.

7.1 Using High-Level Conceptual Data Models for Database Design

Figure 7.1 shows a simplified overview of the database design process. The first step shown is **requirements collection and analysis**. During this step, the database designers interview prospective database users to understand and document their **data requirements**. The result of this step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify

¹A **class** is similar to an *entity type* in many ways.

**Figure 7.1**

A simplified diagram to illustrate the main phases of database design.

the known **functional requirements** of the application. These consist of the user-defined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates. In software design, it is common to use *data flow diagrams*, *sequence diagrams*, *scenarios*, and other techniques to specify functional requirements. We will not discuss any of these techniques here; they are usually described in detail in software engineering texts. We give an overview of some of these techniques in Chapter 10.

Once the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design**. The conceptual schema is a concise description of

the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with nontechnical users. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not conflict. This approach enables database designers to concentrate on specifying the properties of the data, without being concerned with storage and implementation details. This makes it easier to create a good conceptual database design.

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified using the initial schema.

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object-relational database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping**; its result is a database schema in the implementation data model of the DBMS. Data model mapping is often automated or semiautomated within the database design tools.

The last step is the **physical design** phase, during which the internal storage structures, file organizations, indexes, access paths, and physical design parameters for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications. We discuss the database design process in more detail in Chapter 10.

We present only the basic ER model concepts for conceptual schema design in this chapter. Additional modeling concepts are discussed in Chapter 8, when we introduce the EER model.

7.2 A Sample Database Application

In this section we describe a sample database application, called COMPANY, which serves to illustrate the basic ER model concepts and their use in schema design. We list the data requirements for the database here, and then create its conceptual schema step-by-step as we introduce the modeling concepts of the ER model. The COMPANY database keeps track of a company's employees, departments, and projects. Suppose that after the requirements collection and analysis phase, the database designers provide the following description of the *miniworld*—the part of the company that will be represented in the database.

- The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
- A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
- We store each employee's name, Social Security number,² address, salary, sex (gender), and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. We keep track of the current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).
- We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relationship to the employee.

Figure 7.2 shows how the schema for this database application can be displayed by means of the graphical notation known as **ER diagrams**. This figure will be explained gradually as the ER model concepts are presented. We describe the step-by-step process of deriving this schema from the stated requirements—and explain the ER diagrammatic notation—as we introduce the ER model concepts.

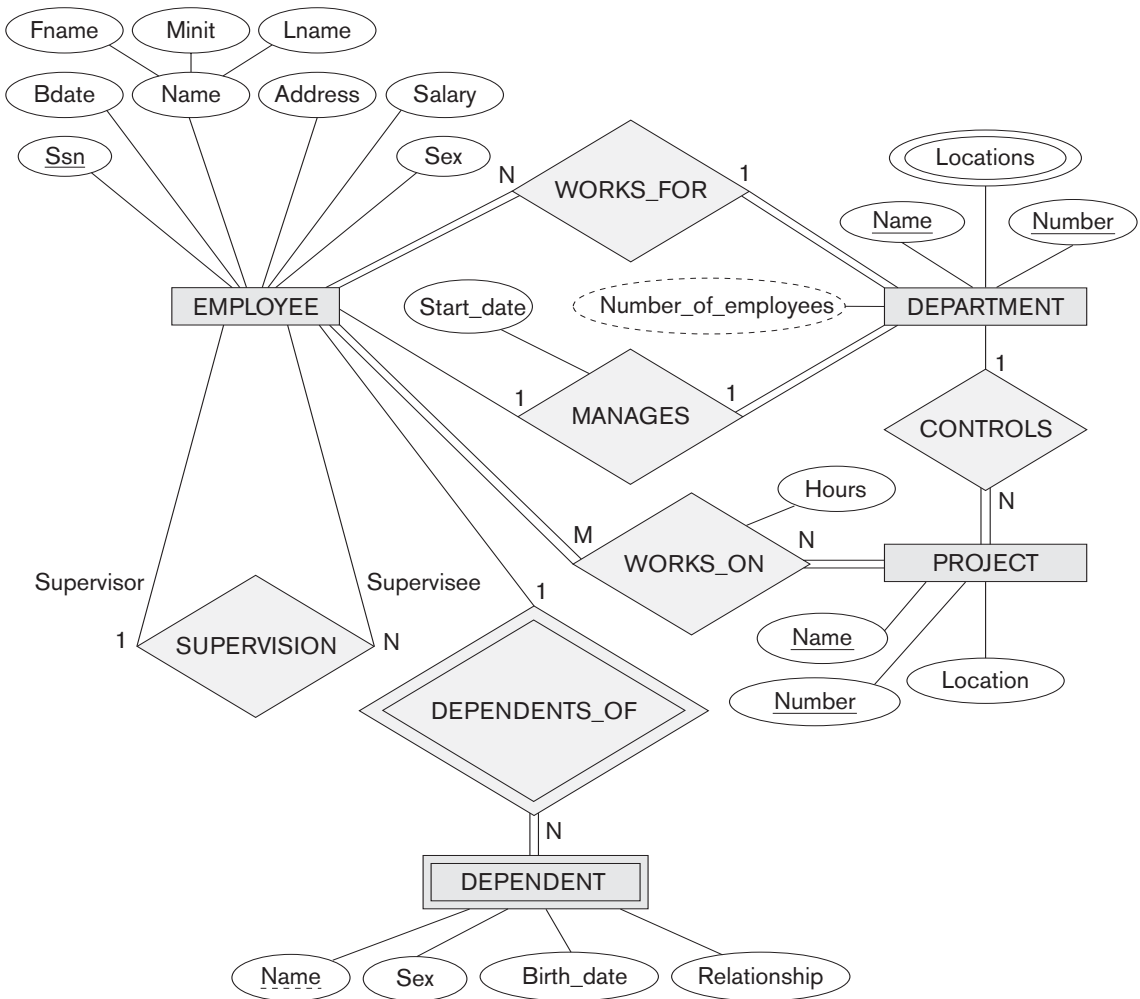
7.3 Entity Types, Entity Sets, Attributes, and Keys

The ER model describes data as *entities*, *relationships*, and *attributes*. In Section 7.3.1 we introduce the concepts of entities and their attributes. We discuss entity types and key attributes in Section 7.3.2. Then, in Section 7.3.3, we specify the initial conceptual design of the entity types for the COMPANY database. Relationships are described in Section 7.4.

7.3.1 Entities and Attributes

Entities and Their Attributes. The basic object that the ER model represents is an **entity**, which is a *thing* in the real world with an independent existence. An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course). Each entity has **attributes**—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job. A particular entity will have a

²The Social Security number, or SSN, is a unique nine-digit identifier assigned to each individual in the United States to keep track of his or her employment, benefits, and taxes. Other countries may have similar identification schemes, such as personal identification card numbers.

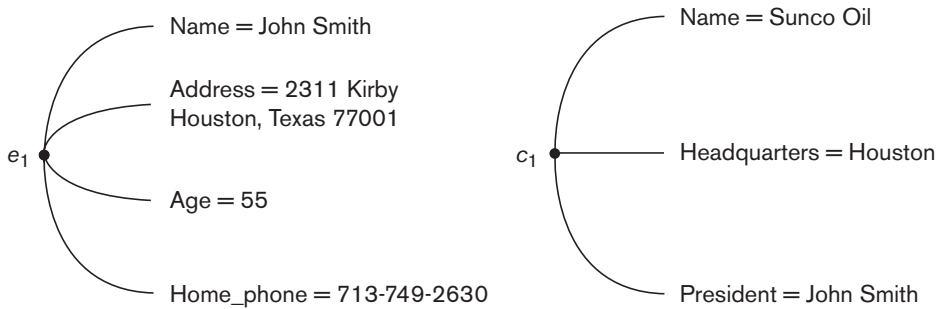
**Figure 7.2**

An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter and is summarized in Figure 7.14.

value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

Figure 7.3 shows two entities and the values of their attributes. The EMPLOYEE entity e_1 has four attributes: Name, Address, Age, and Home_phone; their values are 'John Smith,' '2311 Kirby, Houston, Texas 77001,' '55', and '713-749-2630', respectively. The COMPANY entity c_1 has three attributes: Name, Headquarters, and President; their values are 'Sunco Oil,' 'Houston', and 'John Smith', respectively.

Several types of attributes occur in the ER model: *simple* versus *composite*, *single-valued* versus *multivalued*, and *stored* versus *derived*. First we define these attribute

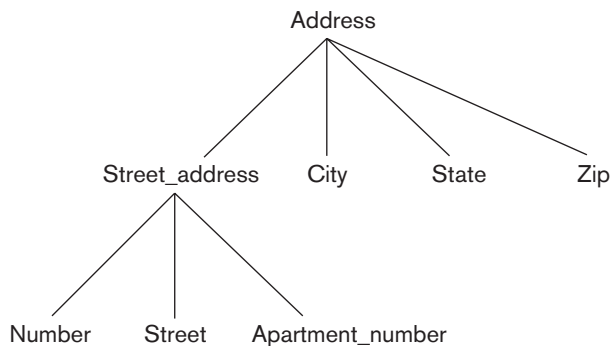
**Figure 7.3**

Two entities, EMPLOYEE e_1 , and COMPANY c_1 , and their attributes.

types and illustrate their use via examples. Then we discuss the concept of a *NULL value* for an attribute.

Composite versus Simple (Atomic) Attributes. **Composite attributes** can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity shown in Figure 7.3 can be subdivided into Street_address, City, State, and Zip,³ with the values ‘2311 Kirby’, ‘Houston’, ‘Texas’, and ‘77001.’ Attributes that are not divisible are called **simple** or **atomic attributes**. Composite attributes can form a hierarchy; for example, Street_address can be further subdivided into three simple component attributes: Number, Street, and Apartment_number, as shown in Figure 7.4. The value of a composite attribute is the concatenation of the values of its component simple attributes.

Composite attributes are useful to model situations in which a user sometimes refers to the composite attribute as a unit but at other times refers specifically to its components. If the composite attribute is referenced only as a whole, there is no

**Figure 7.4**

A hierarchy of composite attributes.

³Zip Code is the name used in the United States for a five-digit postal code, such as 76019, which can be extended to nine digits, such as 76019-0015. We use the five-digit Zip in our examples.

need to subdivide it into component attributes. For example, if there is no need to refer to the individual components of an address (Zip Code, street, and so on), then the whole address can be designated as a simple attribute.

Single-Valued versus Multivalued Attributes. Most attributes have a single value for a particular entity; such attributes are called **single-valued**. For example, Age is a single-valued attribute of a person. In some cases an attribute can have a set of values for the same entity—for instance, a Colors attribute for a car, or a College_degrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two color values. Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees; therefore, different people can have different *numbers of values* for the College_degrees attribute. Such attributes are called **multivalued**. A multivalued attribute may have lower and upper bounds to constrain the *number of values* allowed for each individual entity. For example, the Colors attribute of a car may be restricted to have between one and three values, if we assume that a car can have three colors at most.

Stored versus Derived Attributes. In some cases, two (or more) attribute values are related—for example, the Age and Birth_date attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth_date. The Age attribute is hence called a **derived attribute** and is said to be **derivable from** the Birth_date attribute, which is called a **stored attribute**. Some attribute values can be derived from *related entities*; for example, an attribute Number_of_employees of a DEPARTMENT entity can be derived by counting the number of employees related to (working for) that department.

NULL Values. In some cases, a particular entity may not have an applicable value for an attribute. For example, the Apartment_number attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. Similarly, a College_degrees attribute applies only to people with college degrees. For such situations, a special value called NULL is created. An address of a single-family home would have NULL for its Apartment_number attribute, and a person with no college degree would have NULL for College_degrees. NULL can also be used if we do not know the value of an attribute for a particular entity—for example, if we do not know the home phone number of 'John Smith' in Figure 7.3. The meaning of the former type of NULL is *not applicable*, whereas the meaning of the latter is *unknown*. The *unknown* category of NULL can be further classified into two cases. The first case arises when it is known that the attribute value exists but is *missing*—for instance, if the Height attribute of a person is listed as NULL. The second case arises when it is *not known* whether the attribute value exists—for example, if the Home_phone attribute of a person is NULL.

Complex Attributes. Notice that, in general, composite and multivalued attributes can be nested arbitrarily. We can represent arbitrary nesting by grouping com-

ponents of a composite attribute between parentheses () and separating the components with commas, and by displaying multivalued attributes between braces { }. Such attributes are called **complex attributes**. For example, if a person can have more than one residence and each residence can have a single address and multiple phones, an attribute Address_phone for a person can be specified as shown in Figure 7.5.⁴ Both Phone and Address are themselves composite attributes.

7.3.2 Entity Types, Entity Sets, Keys, and Value Sets

Entity Types and Entity Sets. A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has its *own value(s)* for each attribute. An **entity type** defines a *collection* (or *set*) of entities that have the same attributes. Each entity type in the database is described by its name and attributes. Figure 7.6 shows two entity types: EMPLOYEE and COMPANY, and a list of some of the attributes for

```
{Address_phone( {Phone(Area_code,Phone_number)},Address(Street_address
(Number,Street,Apartment_number),City,State,Zip) )}
```

Figure 7.5
A complex attribute:
Address_phone.

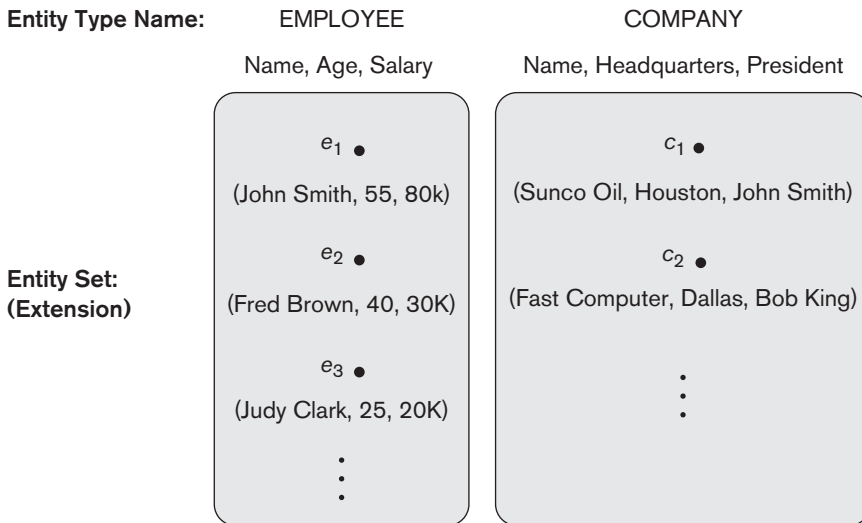


Figure 7.6
Two entity types,
EMPLOYEE and
COMPANY, and some
member entities of
each.

⁴For those familiar with XML, we should note that complex attributes are similar to complex elements in XML (see Chapter 12).

each. A few individual entities of each type are also illustrated, along with the values of their attributes. The collection of all entities of a particular entity type in the database at any point in time is called an **entity set**; the entity set is usually referred to using the same name as the entity type. For example, **EMPLOYEE** refers to both a *type of entity* as well as the current set of *all employee entities* in the database.

An entity type is represented in ER diagrams⁵ (see Figure 7.2) as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals. Figure 7.7(a) shows a **CAR** entity type in this notation.

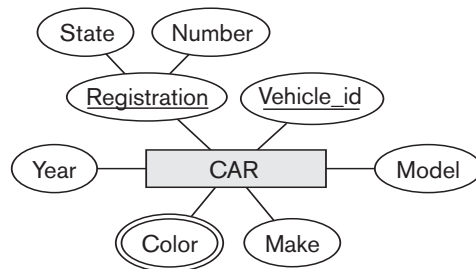
An entity type describes the **schema** or **intension** for a *set of entities* that share the same structure. The collection of entities of a particular entity type is grouped into an entity set, which is also called the **extension** of the entity type.

Key Attributes of an Entity Type. An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually

Figure 7.7

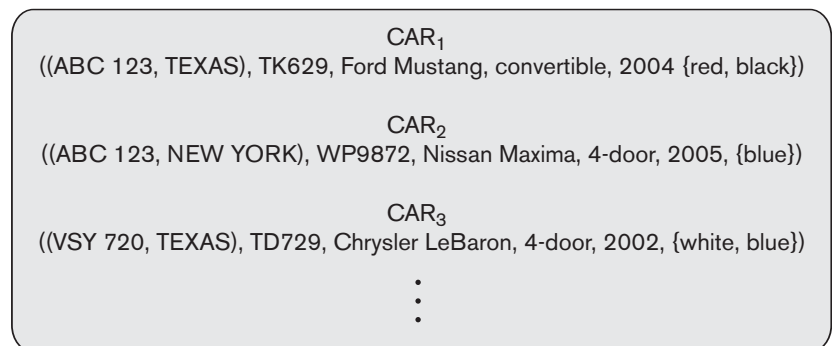
The **CAR** entity type with two key attributes, **Registration** and **Vehicle_id**. (a) ER diagram notation. (b) Entity set with three entities.

(a)



(b)

CAR
Registration (Number, State), Vehicle_id, Make, Model, Year, {Color}



⁵We use a notation for ER diagrams that is close to the original proposed notation (Chen 1976). Many other notations are in use; we illustrate some of them later in this chapter when we present UML class diagrams and in Appendix A.

has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely. For example, the Name attribute is a key of the COMPANY entity type in Figure 7.6 because no two companies are allowed to have the same name. For the PERSON entity type, a typical key attribute is Ssn (Social Security number). Sometimes several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, the proper way to represent this in the ER model that we describe here is to define a *composite attribute* and designate it as a key attribute of the entity type. Notice that such a composite key must be *minimal*; that is, all component attributes must be included in the composite attribute to have the uniqueness property. Superfluous attributes must not be included in a key. In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval, as illustrated in Figure 7.7(a).

Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for *every entity set* of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time. It is not the property of a particular entity set; rather, it is a constraint on *any entity set* of the entity type at any point in time. This key constraint (and other constraints we discuss later) is derived from the constraints of the miniworld that the database represents.

Some entity types have *more than one* key attribute. For example, each of the Vehicle_id and Registration attributes of the entity type CAR (Figure 7.7) is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, State and Number, neither of which is a key on its own. An entity type may also have *no key*, in which case it is called a *weak entity type* (see Section 7.5).

In our diagrammatic notation, if two attributes are underlined separately, then *each is a key on its own*. Unlike the relational model (see Section 3.2.2), there is no concept of primary key in the ER model that we present here; the primary key will be chosen during mapping to a relational schema (see Chapter 9).

Value Sets (Domains) of Attributes. Each simple attribute of an entity type is associated with a **value set** (or **domain** of values), which specifies the set of values that may be assigned to that attribute for each individual entity. In Figure 7.6, if the range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70. Similarly, we can specify the value set for the Name attribute to be the set of strings of alphabetic characters separated by blank characters, and so on. Value sets are not displayed in ER diagrams, and are typically specified using the basic **data types** available in most programming languages, such as integer, string, Boolean, float, enumerated type, subrange, and so on. Additional data types to represent common database types such as date, time, and other concepts are also employed.

Mathematically, an attribute A of entity set E whose value set is V can be defined as a **function** from E to the power set⁶ $P(V)$ of V :

$$A : E \rightarrow P(V)$$

We refer to the value of attribute A for entity e as $A(e)$. The previous definition covers both single-valued and multivalued attributes, as well as NULLs. A NULL value is represented by the *empty set*. For single-valued attributes, $A(e)$ is restricted to being a *singleton set* for each entity e in E , whereas there is no restriction on multivalued attributes.⁷ For a composite attribute A , the value set V is the power set of the Cartesian product of $P(V_1), P(V_2), \dots, P(V_n)$, where V_1, V_2, \dots, V_n are the value sets of the simple component attributes that form A :

$$V = P(P(V_1) \times P(V_2) \times \dots \times P(V_n))$$

The value set provides all possible values. Usually only a small number of these values exist in the database at a particular time. Those values represent the data from the current state of the miniworld. They correspond to the data as it actually exists in the miniworld.

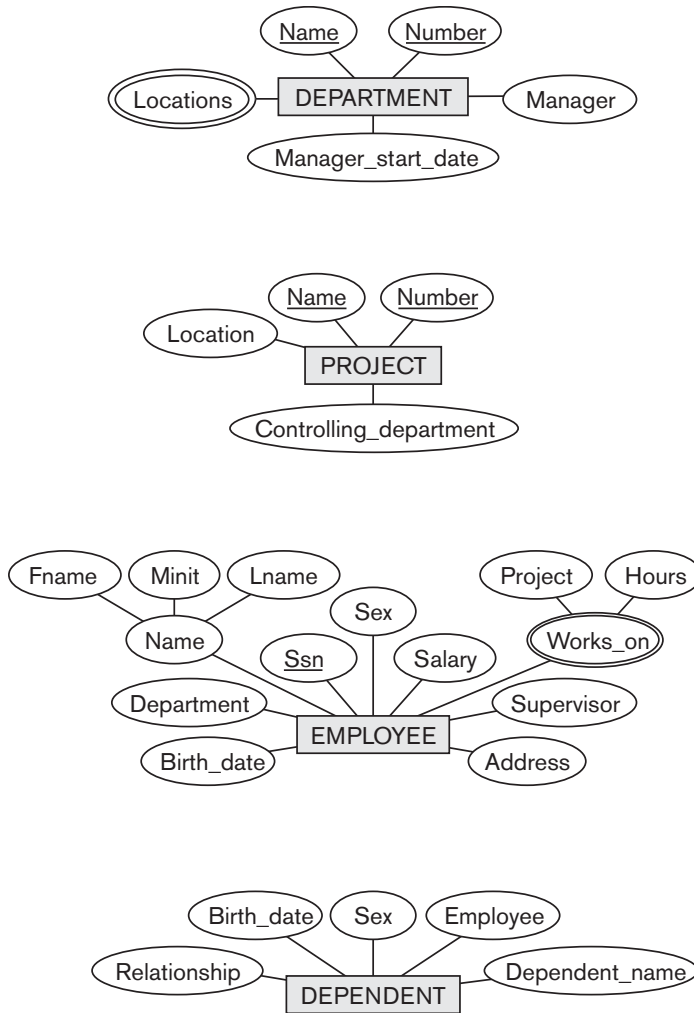
7.3.3 Initial Conceptual Design of the COMPANY Database

We can now define the entity types for the COMPANY database, based on the requirements described in Section 7.2. After defining several entity types and their attributes here, we refine our design in Section 7.4 after we introduce the concept of a relationship. According to the requirements listed in Section 7.2, we can identify four entity types—one corresponding to each of the four items in the specification (see Figure 7.8):

1. An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and Manager_start_date. Locations is the only multivalued attribute. We can specify that both Name and Number are (separate) key attributes because each was specified to be unique.
2. An entity type PROJECT with attributes Name, Number, Location, and Controlling_department. Both Name and Number are (separate) key attributes.
3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth_date, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name—First_name, Middle_initial, Last_name—or of Address.
4. An entity type DEPENDENT with attributes Employee, Dependent_name, Sex, Birth_date, and Relationship (to the employee).

⁶The **power set** $P(V)$ of a set V is the set of all subsets of V .

⁷A **singleton** set is a set with only one element (value).

**Figure 7.8**

Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

So far, we have not represented the fact that an employee can work on several projects, nor have we represented the number of hours per week an employee works on each project. This characteristic is listed as part of the third requirement in Section 7.2, and it can be represented by a multivalued composite attribute of **EMPLOYEE** called Works_on with the simple components (Project, Hours). Alternatively, it can be represented as a multivalued composite attribute of **PROJECT** called Workers with the simple components (Employee, Hours). We choose the first alternative in Figure 7.8, which shows each of the entity types just described. The Name attribute of **EMPLOYEE** is shown as a composite attribute, presumably after consultation with the users.

7.4 Relationship Types, Relationship Sets, Roles, and Structural Constraints

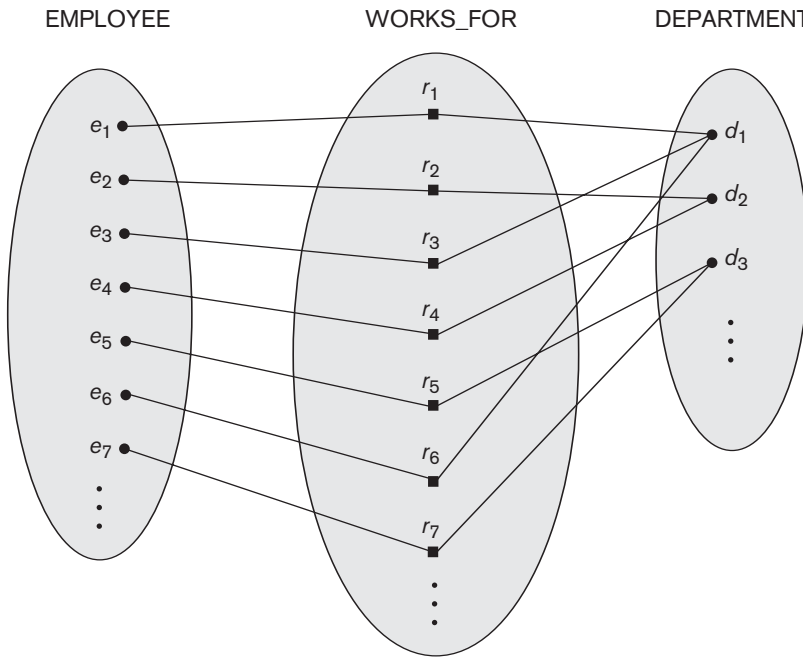
In Figure 7.8 there are several *implicit relationships* among the various entity types. In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists. For example, the attribute `Manager` of `DEPARTMENT` refers to an employee who manages the department; the attribute `Controlling_department` of `PROJECT` refers to the department that controls the project; the attribute `Supervisor` of `EMPLOYEE` refers to another employee (the one who supervises this employee); the attribute `Department` of `EMPLOYEE` refers to the department for which the employee works; and so on. In the ER model, these references should not be represented as attributes but as **relationships**, which are discussed in this section. The `COMPANY` database schema will be refined in Section 7.6 to represent relationships explicitly. In the initial design of entity types, relationships are typically captured in the form of attributes. As the design is refined, these attributes get converted into relationships between entity types.

This section is organized as follows: Section 7.4.1 introduces the concepts of relationship types, relationship sets, and relationship instances. We define the concepts of relationship degree, role names, and recursive relationships in Section 7.4.2, and then we discuss structural constraints on relationships—such as cardinality ratios and existence dependencies—in Section 7.4.3. Section 7.4.4 shows how relationship types can also have attributes.

7.4.1 Relationship Types, Sets, and Instances

A **relationship type** R among n entity types E_1, E_2, \dots, E_n defines a set of associations—or a **relationship set**—among entities from these entity types. As for the case of entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the *same name*, R . Mathematically, the relationship set R is a set of **relationship instances** r_i , where each r_i associates n individual entities (e_1, e_2, \dots, e_n) , and each entity e_j in r_i is a member of entity set E_j , $1 \leq j \leq n$. Hence, a relationship set is a mathematical relation on E_1, E_2, \dots, E_n ; alternatively, it can be defined as a subset of the Cartesian product of the entity sets $E_1 \times E_2 \times \dots \times E_n$. Each of the entity types E_1, E_2, \dots, E_n is said to participate in the relationship type R ; similarly, each of the individual entities e_1, e_2, \dots, e_n is said to **participate** in the relationship instance $r_i = (e_1, e_2, \dots, e_n)$.

Informally, each relationship instance r_i in R is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance r_i represents the fact that the entities participating in r_i are related in some way in the corresponding miniworld situation. For example, consider a relationship type `WORKS_FOR` between the two entity types `EMPLOYEE` and `DEPARTMENT`, which associates each employee with the department for which the employee works in the corresponding entity set. Each relationship instance in the relationship set `WORKS_FOR` associates one `EMPLOYEE` entity and one `DEPARTMENT` entity. Figure 7.9 illustrates this example, where each relationship

**Figure 7.9**

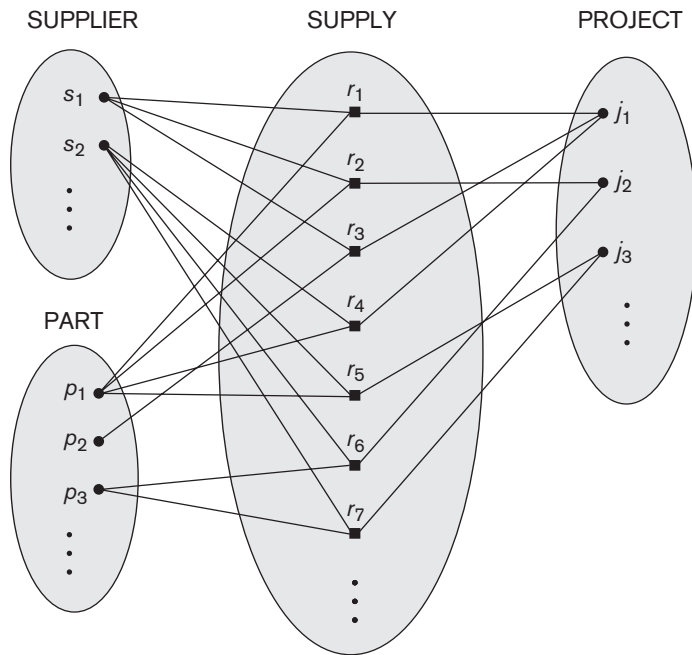
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

instance r_i is shown connected to the EMPLOYEE and DEPARTMENT entities that participate in r_i . In the miniworld represented by Figure 7.9, employees e_1 , e_3 , and e_6 work for department d_1 ; employees e_2 and e_4 work for department d_2 ; and employees e_5 and e_7 work for department d_3 .

In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box (see Figure 7.2).

7.4.2 Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type. The **degree** of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called **binary**, and one of degree three is called **ternary**. An example of a ternary relationship is SUPPLY, shown in Figure 7.10, where each relationship instance r_i associates three entities—a supplier s , a part p , and a project j —whenever s supplies part p to project j . Relationships can generally be of any degree, but the ones most common are binary relationships. Higher-degree relationships are generally more complex than binary relationships; we characterize them further in Section 7.9.

**Figure 7.10**

Some relationship instances in the **SUPPLY** ternary relationship set.

Relationships as Attributes. It is sometimes convenient to think of a binary relationship type in terms of attributes, as we discussed in Section 7.3.3. Consider the **WORKS_FOR** relationship type in Figure 7.9. One can think of an attribute called **Department** of the **EMPLOYEE** entity type, where the value of **Department** for each **EMPLOYEE** entity is (a reference to) the **DEPARTMENT** entity for which that employee works. Hence, the value set for this **Department** attribute is the set of *all* **DEPARTMENT** entities, which is the **DEPARTMENT** entity set. This is what we did in Figure 7.8 when we specified the initial design of the entity type **EMPLOYEE** for the **COMPANY** database. However, when we think of a binary relationship as an attribute, we always have two options. In this example, the alternative is to think of a multivalued attribute **Employee** of the entity type **DEPARTMENT** whose values for each **DEPARTMENT** entity is the set of **EMPLOYEE** entities who work for that department. The value set of this **Employee** attribute is the power set of the **EMPLOYEE** entity set. Either of these two attributes—**Department** of **EMPLOYEE** or **Employee** of **DEPARTMENT**—can represent the **WORKS_FOR** relationship type. If both are represented, they are constrained to be inverses of each other.⁸

⁸This concept of representing relationship types as attributes is used in a class of data models called **functional data models**. In object databases (see Chapter 11), relationships can be represented by reference attributes, either in one direction or in both directions as inverses. In relational databases (see Chapter 3), foreign keys are a type of reference attribute used to represent relationships.

Role Names and Recursive Relationships. Each entity type that participates in a relationship type plays a particular role in the relationship. The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance, and helps to explain what the relationship means. For example, in the WORKS_FOR relationship type, EMPLOYEE plays the role of *employee* or *worker* and DEPARTMENT plays the role of *department* or *employer*.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each participating entity type name can be used as the role name. However, in some cases the *same* entity type participates more than once in a relationship type in *different* roles. In such cases the role name becomes essential for distinguishing the meaning of the role that each participating entity plays. Such relationship types are called **recursive relationships**. Figure 7.11 shows an example. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. Hence, the EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or *boss*), and once in the role of *supervisee* (or *subordinate*). Each relationship instance r_i in SUPERVISION associates two employee entities e_j and e_k , one of which plays the role of supervisor and the other the role of supervisee. In Figure 7.11, the lines marked '1' represent the supervisor role, and those marked '2' represent the supervisee role; hence, e_1 supervises e_2 and e_3 , e_4 supervises e_6 and e_7 , and e_5 supervises e_1 and e_4 . In this example, each relationship instance must be connected with two lines, one marked with '1' (supervisor) and the other with '2' (supervisee).

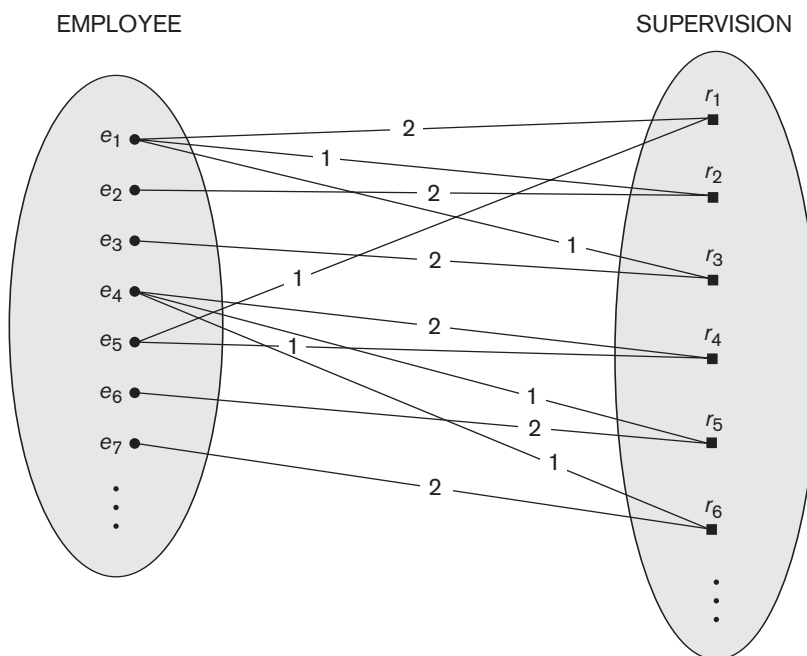


Figure 7.11

A recursive relationship SUPERVISION between EMPLOYEE in the *supervisor* role (1) and EMPLOYEE in the *subordinate* role (2).

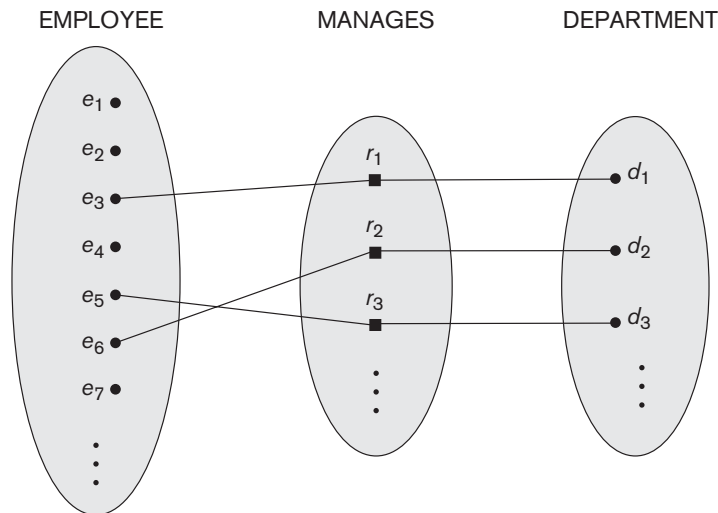
7.4.3 Constraints on Binary Relationship Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the miniworld situation that the relationships represent. For example, in Figure 7.9, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of binary relationship constraints: *cardinality ratio* and *participation*.

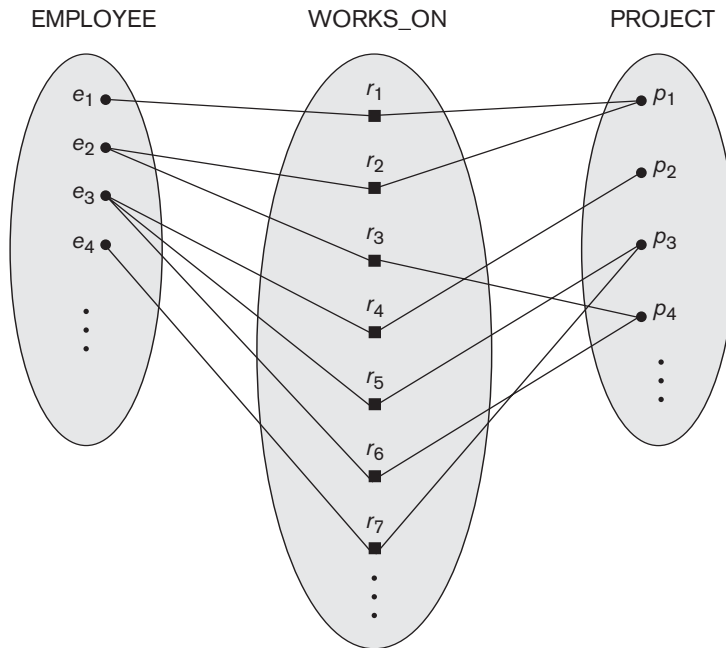
Cardinality Ratios for Binary Relationships. The **cardinality ratio** for a binary relationship specifies the *maximum* number of relationship instances that an entity can participate in. For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to (that is, employs) any number of employees,⁹ but an employee can be related to (work for) only one department. This means that for this particular relationship WORKS_FOR, a particular department entity can be related to any number of employees (N indicates there is no maximum number). On the other hand, an employee can be related to a maximum of one department. The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N.

An example of a 1:1 binary relationship is MANAGES (Figure 7.12), which relates a department entity to the employee who manages that department. This represents the miniworld constraints that—at any point in time—an employee can manage one department only and a department can have one manager only. The relationship type WORKS_ON (Figure 7.13) is of cardinality ratio M:N, because the mini-

Figure 7.12
A 1:1 relationship,
MANAGES.



⁹N stands for *any number* of related entities (zero or more).

**Figure 7.13**An M:N relationship,
WORKS_ON.

world rule is that an employee can work on several projects and a project can have several employees.

Cardinality ratios for binary relationships are represented on ER diagrams by displaying 1, M, and N on the diamonds as shown in Figure 7.2. Notice that in this notation, we can either specify no maximum (N) or a maximum of one (1) on participation. An alternative notation (see Section 7.7.4) allows the designer to specify a specific *maximum number* on participation, such as 4 or 5.

Participation Constraints and Existence Dependencies. The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. This constraint specifies the *minimum* number of relationship instances that each entity can participate in, and is sometimes called the **minimum cardinality constraint**. There are two types of participation constraints—total and partial—that we illustrate by example. If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in at least one WORKS_FOR relationship instance (Figure 7.9). Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation**, meaning that every entity in the *total set* of employee entities must be related to a department entity via WORKS_FOR. Total participation is also called **existence dependency**. In Figure 7.12 we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial**, meaning that *some* or *part of the set* of employee entities are related to some department entity via MANAGES, but not necessarily all. We will

refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

In ER diagrams, total participation (or existence dependency) is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* (see Figure 7.2). Notice that in this notation, we can either specify no minimum (partial participation) or a minimum of one (total participation). The alternative notation (see Section 7.7.4) allows the designer to specify a specific *minimum number* on participation in the relationship, such as 4 or 5.

We will discuss constraints on higher-degree relationships in Section 7.9.

7.4.4 Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that an employee works on a particular project, we can include an attribute *Hours* for the *WORKS_ON* relationship type in Figure 7.13. Another example is to include the date on which a manager started managing a department via an attribute *Start_date* for the *MANAGES* relationship type in Figure 7.12.

Notice that attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types. For example, the *Start_date* attribute for the *MANAGES* relationship can be an attribute of either *EMPLOYEE* or *DEPARTMENT*, although conceptually it belongs to *MANAGES*. This is because *MANAGES* is a 1:1 relationship, so every department or employee entity participates in *at most one* relationship instance. Hence, the value of the *Start_date* attribute can be determined separately, either by the participating department entity or by the participating employee (manager) entity.

For a 1:N relationship type, a relationship attribute can be migrated *only* to the entity type on the N-side of the relationship. For example, in Figure 7.9, if the *WORKS_FOR* relationship also has an attribute *Start_date* that indicates when an employee started working for a department, this attribute can be included as an attribute of *EMPLOYEE*. This is because each employee works for only one department, and hence participates in at most one relationship instance in *WORKS_FOR*. In both 1:1 and 1:N relationship types, the decision where to place a relationship attribute—as a relationship type attribute or as an attribute of a participating entity type—is determined subjectively by the schema designer.

For M:N relationship types, some attributes may be determined by the *combination of participating entities* in a relationship instance, not by any single entity. Such attributes *must be specified as relationship attributes*. An example is the *Hours* attribute of the M:N relationship *WORKS_ON* (Figure 7.13); the number of hours per week an employee currently works on a project is determined by an employee-project combination and not separately by either entity.

7.5 Weak Entity Types

Entity types that do not have key attributes of their own are called **weak entity types**. In contrast, **regular entity types** that do have a key attribute—which include all the examples discussed so far—are called **strong entity types**. Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. We call this other entity type the **identifying** or **owner entity type**,¹⁰ and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type.¹¹ A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship because a weak entity cannot be identified without an owner entity. However, not every existence dependency results in a weak entity type. For example, a DRIVER_LICENSE entity cannot exist unless it is related to a PERSON entity, even though it has its own key (License_number) and hence is not a weak entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship (Figure 7.2). In our example, the attributes of DEPENDENT are Name (the first name of the dependent), Birth_date, Sex, and Relationship (to the employee). Two dependents of *two distinct employees* may, by chance, have the same values for Name, Birth_date, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the *particular employee entity* to which each dependent is related. Each employee entity is said to *own* the dependent entities that are related to it.

A weak entity type normally has a **partial key**, which is the attribute that can uniquely identify weak entities that are *related to the same owner entity*.¹² In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key.

In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with double lines (see Figure 7.2). The partial key attribute is underlined with a dashed or dotted line.

Weak entity types can sometimes be represented as complex (composite, multivalued) attributes. In the preceding example, we could specify a multivalued attribute Dependents for EMPLOYEE, which is a composite attribute with component attributes Name, Birth_date, Sex, and Relationship. The choice of which representation to use is made by the database designer. One criterion that may be used is to choose the

¹⁰The identifying entity type is also sometimes called the **parent entity type** or the **dominant entity type**.

¹¹The weak entity type is also sometimes called the **child entity type** or the **subordinate entity type**.

¹²The partial key is sometimes called the **discriminator**.

weak entity type representation if there are many attributes. If the weak entity participates independently in relationship types other than its identifying relationship type, then it should not be modeled as a complex attribute.

In general, any number of levels of weak entity types can be defined; an owner entity type may itself be a weak entity type. In addition, a weak entity type may have more than one identifying entity type and an identifying relationship type of degree higher than two, as we illustrate in Section 7.9.

7.6 Refining the ER Design for the COMPANY Database

We can now refine the database design in Figure 7.8 by changing the attributes that represent relationships into relationship types. The cardinality ratio and participation constraint of each relationship type are determined from the requirements listed in Section 7.2. If some cardinality ratio or dependency cannot be determined from the requirements, the users must be questioned further to determine these structural constraints.

In our example, we specify the following relationship types:

- MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at all times, which implies total participation.¹³ The attribute Start_date is assigned to this relationship type.
- WORKS_FOR, a 1:N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.
- CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users indicates that some departments may control no projects.
- SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.
- WORKS_ON, determined to be an M:N relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.
- DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity

¹³The rules in the miniworld that determine the constraints are sometimes called the *business rules*, since they are determined by the *business* or organization that will utilize the database.

type **DEPENDENT**. The participation of **EMPLOYEE** is partial, whereas that of **DEPENDENT** is total.

After specifying the above six relationship types, we remove from the entity types in Figure 7.8 all attributes that have been refined into relationships. These include **Manager** and **Manager_start_date** from **DEPARTMENT**; **Controlling_department** from **PROJECT**; **Department**, **Supervisor**, and **Works_on** from **EMPLOYEE**; and **Employee** from **DEPENDENT**. It is important to have the least possible redundancy when we design the conceptual schema of a database. If some redundancy is desired at the storage level or at the user view level, it can be introduced later, as discussed in Section 1.6.1.

7.7 ER Diagrams, Naming Conventions, and Design Issues

7.7.1 Summary of Notation for ER Diagrams

Figures 7.9 through 7.13 illustrate examples of the participation of entity types in relationship types by displaying their sets or extensions—the individual entity instances in an entity set and the individual relationship instances in a relationship set. In ER diagrams the emphasis is on representing the schemas rather than the instances. This is more useful in database design because a database schema changes rarely, whereas the contents of the entity sets change frequently. In addition, the schema is obviously easier to display, because it is much smaller.

Figure 7.2 displays the **COMPANY ER database schema** as an **ER diagram**. We now review the full ER diagram notation. Entity types such as **EMPLOYEE**, **DEPARTMENT**, and **PROJECT** are shown in rectangular boxes. Relationship types such as **WORKS_FOR**, **MANAGES**, **CONTROLS**, and **WORKS_ON** are shown in diamond-shaped boxes attached to the participating entity types with straight lines. Attributes are shown in ovals, and each attribute is attached by a straight line to its entity type or relationship type. Component attributes of a composite attribute are attached to the oval representing the composite attribute, as illustrated by the **Name** attribute of **EMPLOYEE**. Multivalued attributes are shown in double ovals, as illustrated by the **Locations** attribute of **DEPARTMENT**. Key attributes have their names underlined. Derived attributes are shown in dotted ovals, as illustrated by the **Number_of_employees** attribute of **DEPARTMENT**.

Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds, as illustrated by the **DEPENDENT** entity type and the **DEPENDENTS_OF** identifying relationship type. The partial key of the weak entity type is underlined with a dotted line.

In Figure 7.2 the cardinality ratio of each *binary* relationship type is specified by attaching a 1, M, or N on each participating edge. The cardinality ratio of **DEPARTMENT:EMPLOYEE** in **MANAGES** is 1:1, whereas it is 1:N for **DEPARTMENT:EMPLOYEE** in **WORKS_FOR**, and M:N for **WORKS_ON**. The participation

constraint is specified by a single line for partial participation and by double lines for total participation (existence dependency).

In Figure 7.2 we show the role names for the SUPERVISION relationship type because the same EMPLOYEE entity type plays two distinct roles in that relationship. Notice that the cardinality ratio is 1:N from supervisor to supervisee because each employee in the role of supervisee has at most one direct supervisor, whereas an employee in the role of supervisor can supervise zero or more employees.

Figure 7.14 summarizes the conventions for ER diagrams. It is important to note that there are many other alternative diagrammatic notations (see Section 7.7.4 and Appendix A).

7.7.2 Proper Naming of Schema Constructs

When designing a database schema, the choice of names for entity types, attributes, relationship types, and (particularly) roles is not always straightforward. One should choose names that convey, as much as possible, the meanings attached to the different constructs in the schema. We choose to use *singular names* for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type. In our ER diagrams, we will use the convention that entity type and relationship type names are uppercase letters, attribute names have their initial letter capitalized, and role names are lowercase letters. We have used this convention in Figure 7.2.

As a general practice, given a narrative description of the database requirements, the *nouns* appearing in the narrative tend to give rise to entity type names, and the *verbs* tend to indicate names of relationship types. Attribute names generally arise from additional nouns that describe the nouns corresponding to entity types.

Another naming consideration involves choosing binary relationship names to make the ER diagram of the schema readable from left to right and from top to bottom. We have generally followed this guideline in Figure 7.2. To explain this naming convention further, we have one exception to the convention in Figure 7.2—the DEPENDENTS_OF relationship type, which reads from bottom to top. When we describe this relationship, we can say that the DEPENDENT entities (bottom entity type) are DEPENDENTS_OF (relationship name) an EMPLOYEE (top entity type). To change this to read from top to bottom, we could rename the relationship type to HAS_DEPENDENTS, which would then read as follows: An EMPLOYEE entity (top entity type) HAS_DEPENDENTS (relationship name) of type DEPENDENT (bottom entity type). Notice that this issue arises because each binary relationship can be described starting from either of the two participating entity types, as discussed in the beginning of Section 7.4.

7.7.3 Design Choices for ER Conceptual Design

It is occasionally difficult to decide whether a particular concept in the miniworld should be modeled as an entity type, an attribute, or a relationship type. In this


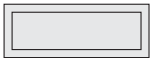

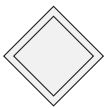

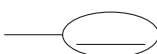

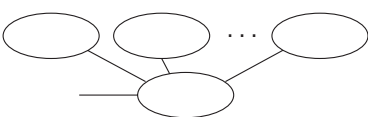
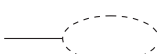
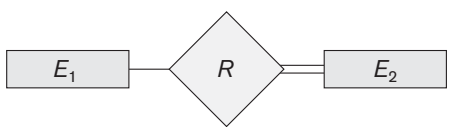
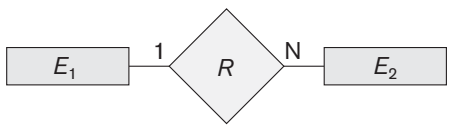
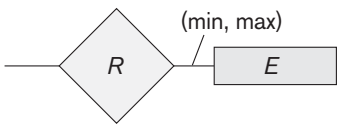
Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Identifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
	Total Participation of E_2 in R
	Cardinality Ratio 1 : N for $E_1:E_2$ in R
	Structural Constraint (min, max) on Participation of E in R

Figure 7.14

Summary of the notation for ER diagrams.

section, we give some brief guidelines as to which construct should be chosen in particular situations.

In general, the schema design process should be considered an iterative refinement process, where an initial design is created and then iteratively refined until the most suitable design is reached. Some of the refinements that are often used include the following:

- A concept may be first modeled as an attribute and then refined into a relationship because it is determined that the attribute is a reference to another entity type. It is often the case that a pair of such attributes that are inverses of one another are refined into a binary relationship. We discussed this type of refinement in detail in Section 7.6. It is important to note that in our notation, once an attribute is replaced by a relationship, the attribute itself should be removed from the entity type to avoid duplication and redundancy.
- Similarly, an attribute that exists in several entity types may be elevated or promoted to an independent entity type. For example, suppose that several entity types in a UNIVERSITY database, such as STUDENT, INSTRUCTOR, and COURSE, each has an attribute Department in the initial design; the designer may then choose to create an entity type DEPARTMENT with a single attribute Dept_name and relate it to the three entity types (STUDENT, INSTRUCTOR, and COURSE) via appropriate relationships. Other attributes/relationships of DEPARTMENT may be discovered later.
- An inverse refinement to the previous case may be applied—for example, if an entity type DEPARTMENT exists in the initial design with a single attribute Dept_name and is related to only one other entity type, STUDENT. In this case, DEPARTMENT may be reduced or demoted to an attribute of STUDENT.
- Section 7.9 discusses choices concerning the degree of a relationship. In Chapter 8, we discuss other refinements concerning specialization/generalization. Chapter 10 discusses additional top-down and bottom-up refinements that are common in large-scale conceptual schema design.

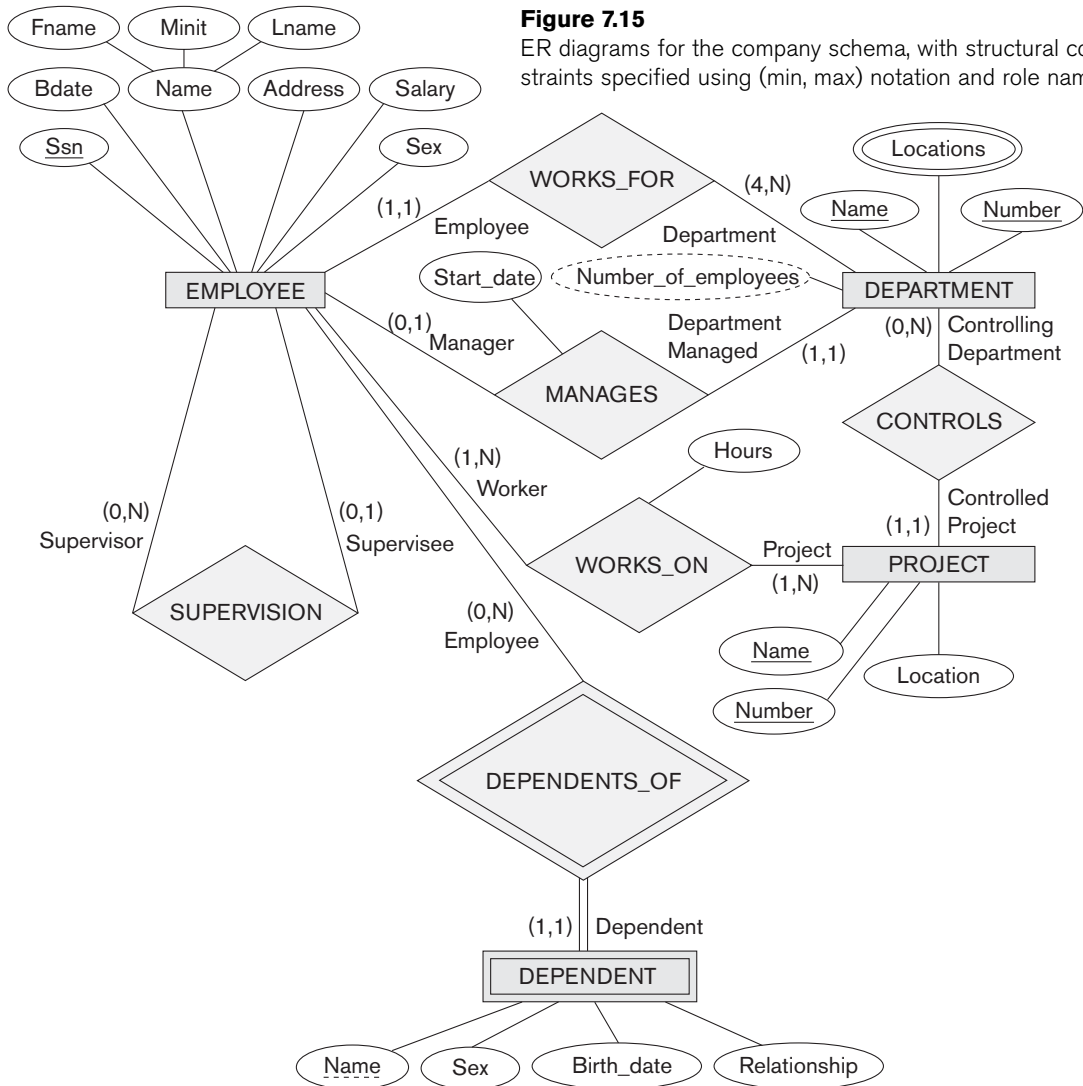
7.7.4 Alternative Notations for ER Diagrams

There are many alternative diagrammatic notations for displaying ER diagrams. Appendix A gives some of the more popular notations. In Section 7.8, we introduce the Unified Modeling Language (UML) notation for class diagrams, which has been proposed as a standard for conceptual object modeling.

In this section, we describe one alternative ER notation for specifying structural constraints on relationships, which replaces the cardinality ratio (1:1, 1:N, M:N) and single/double line notation for participation constraints. This notation involves associating a pair of integer numbers (min, max) with each *participation* of an entity type E in a relationship type R , where $0 \leq \min \leq \max$ and $\max \geq 1$. The numbers mean that for each entity e in E , e must participate in at least min and at most

max relationship instances in R at any point in time. In this method, $\min = 0$ implies partial participation, whereas $\min > 0$ implies total participation.

Figure 7.15 displays the COMPANY database schema using the (min, max) notation.¹⁴ Usually, one uses either the cardinality ratio/single-line/double-line notation or the (min, max) notation. The (min, max)



¹⁴In some notations, particularly those used in object modeling methodologies such as UML, the (min, max) is placed on the *opposite sides* to the ones we have shown. For example, for the WORKS_FOR relationship in Figure 7.15, the (1,1) would be on the DEPARTMENT side, and the (4,N) would be on the EMPLOYEE side. Here we used the original notation from Abrial (1974).

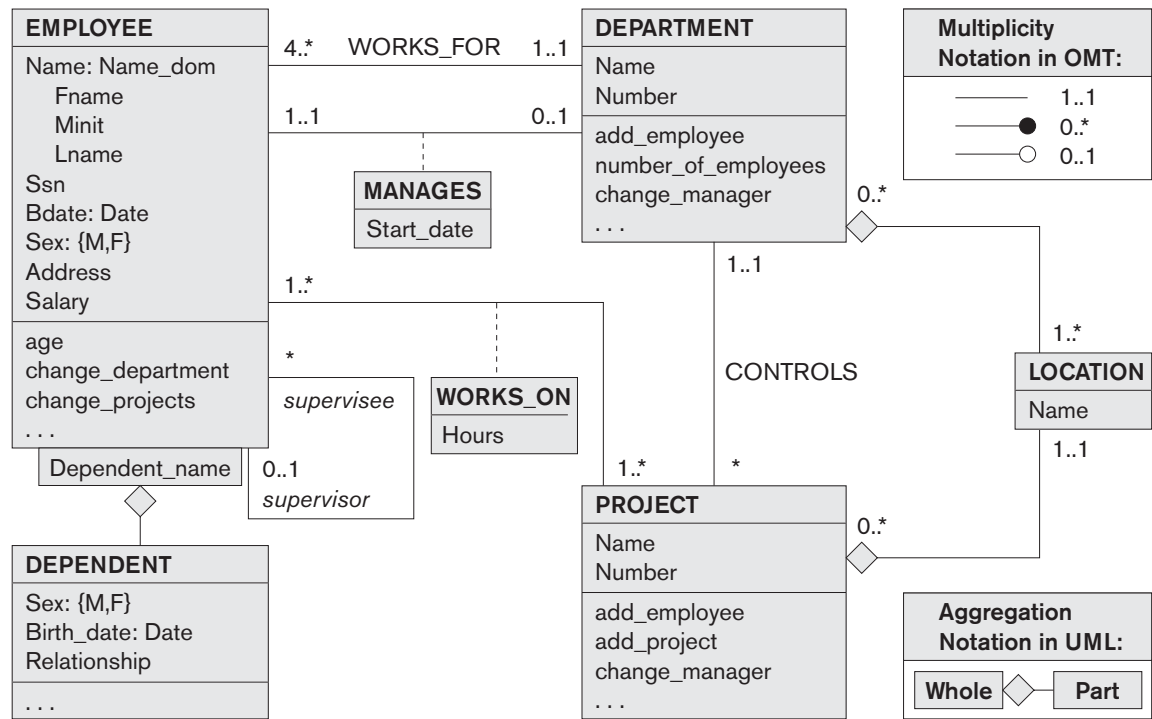
notation is more precise, and we can use it to specify some structural constraints for relationship types of *higher degree*. However, it is not sufficient for specifying some key constraints on higher-degree relationships, as discussed in Section 7.9.

Figure 7.15 also displays all the role names for the COMPANY database schema.

7.8 Example of Other Notation: UML Class Diagrams

The UML methodology is being used extensively in software design and has many types of diagrams for various software design purposes. We only briefly present the basics of **UML class diagrams** here, and compare them with ER diagrams. In some ways, class diagrams can be considered as an alternative notation to ER diagrams. Additional UML notation and concepts are presented in Section 8.6, and in Chapter 10. Figure 7.16 shows how the COMPANY ER database schema in Figure 7.15 can be displayed using UML class diagram notation. The *entity types* in Figure 7.15 are modeled as *classes* in Figure 7.16. An *entity* in ER corresponds to an *object* in UML.

Figure 7.16
The COMPANY conceptual schema
in UML class diagram notation.



In UML class diagrams, a **class** (similar to an entity type in ER) is displayed as a box (see Figure 7.16) that includes three sections: The top section gives the **class name** (similar to entity type name); the middle section includes the **attributes**; and the last section includes **operations** that can be applied to individual objects (similar to individual entities in an entity set) of the class. Operations are *not* specified in ER diagrams. Consider the EMPLOYEE class in Figure 7.16. Its attributes are Name, Ssn, Bdate, Sex, Address, and Salary. The designer can optionally specify the **domain** of an attribute if desired, by placing a colon (:) followed by the domain name or description, as illustrated by the Name, Sex, and Bdate attributes of EMPLOYEE in Figure 7.16. A composite attribute is modeled as a **structured domain**, as illustrated by the Name attribute of EMPLOYEE. A multivalued attribute will generally be modeled as a separate class, as illustrated by the LOCATION class in Figure 7.16.

Relationship types are called **associations** in UML terminology, and relationship instances are called **links**. A **binary association** (binary relationship type) is represented as a line connecting the participating classes (entity types), and may optionally have a name. A relationship attribute, called a **link attribute**, is placed in a box that is connected to the association's line by a dashed line. The (min, max) notation described in Section 7.7.4 is used to specify relationship constraints, which are called **multiplicities** in UML terminology. Multiplicities are specified in the form *min..max*, and an asterisk (*) indicates no maximum limit on participation. However, the multiplicities are placed *on the opposite ends of the relationship* when compared with the notation discussed in Section 7.7.4 (compare Figures 7.15 and 7.16). In UML, a single asterisk indicates a multiplicity of 0..*, and a single 1 indicates a multiplicity of 1..1. A recursive relationship (see Section 7.4.2) is called a **reflexive association** in UML, and the role names—like the multiplicities—are placed at the opposite ends of an association when compared with the placing of role names in Figure 7.15.

In UML, there are two types of relationships: association and aggregation. **Aggregation** is meant to represent a relationship between a whole object and its component parts, and it has a distinct diagrammatic notation. In Figure 7.16, we modeled the locations of a department and the single location of a project as aggregations. However, aggregation and association do not have different structural properties, and the choice as to which type of relationship to use is somewhat subjective. In the ER model, both are represented as relationships.

UML also distinguishes between **unidirectional** and **bidirectional** associations (or aggregations). In the unidirectional case, the line connecting the classes is displayed with an arrow to indicate that only one direction for accessing related objects is needed. If no arrow is displayed, the bidirectional case is assumed, which is the default. For example, if we always expect to access the manager of a department starting from a DEPARTMENT object, we would draw the association line representing the MANAGES association with an arrow from DEPARTMENT to EMPLOYEE. In addition, relationship instances may be specified to be **ordered**. For example, we could specify that the employee objects related to each department through the WORKS_FOR association (relationship) should be ordered by their Salary attribute

value. Association (relationship) names are *optional* in UML, and relationship attributes are displayed in a box attached with a dashed line to the line representing the association/aggregation (see *Start_date* and *Hours* in Figure 7.16).

The operations given in each class are derived from the functional requirements of the application, as we discussed in Section 7.1. It is generally sufficient to specify the operation names initially for the logical operations that are expected to be applied to individual objects of a class, as shown in Figure 7.16. As the design is refined, more details are added, such as the exact argument types (parameters) for each operation, plus a functional description of each operation. UML has *function descriptions* and *sequence diagrams* to specify some of the operation details, but these are beyond the scope of our discussion. Chapter 10 will introduce some of these diagrams.

Weak entities can be modeled using the construct called **qualified association** (or **qualified aggregation**) in UML; this can represent both the identifying relationship and the partial key, which is placed in a box attached to the owner class. This is illustrated by the *DEPENDENT* class and its qualified aggregation to *EMPLOYEE* in Figure 7.16. The partial key *Dependent_name* is called the **discriminator** in UML terminology, since its value distinguishes the objects associated with (related to) the same *EMPLOYEE*. Qualified associations are not restricted to modeling weak entities, and they can be used to model other situations in UML.

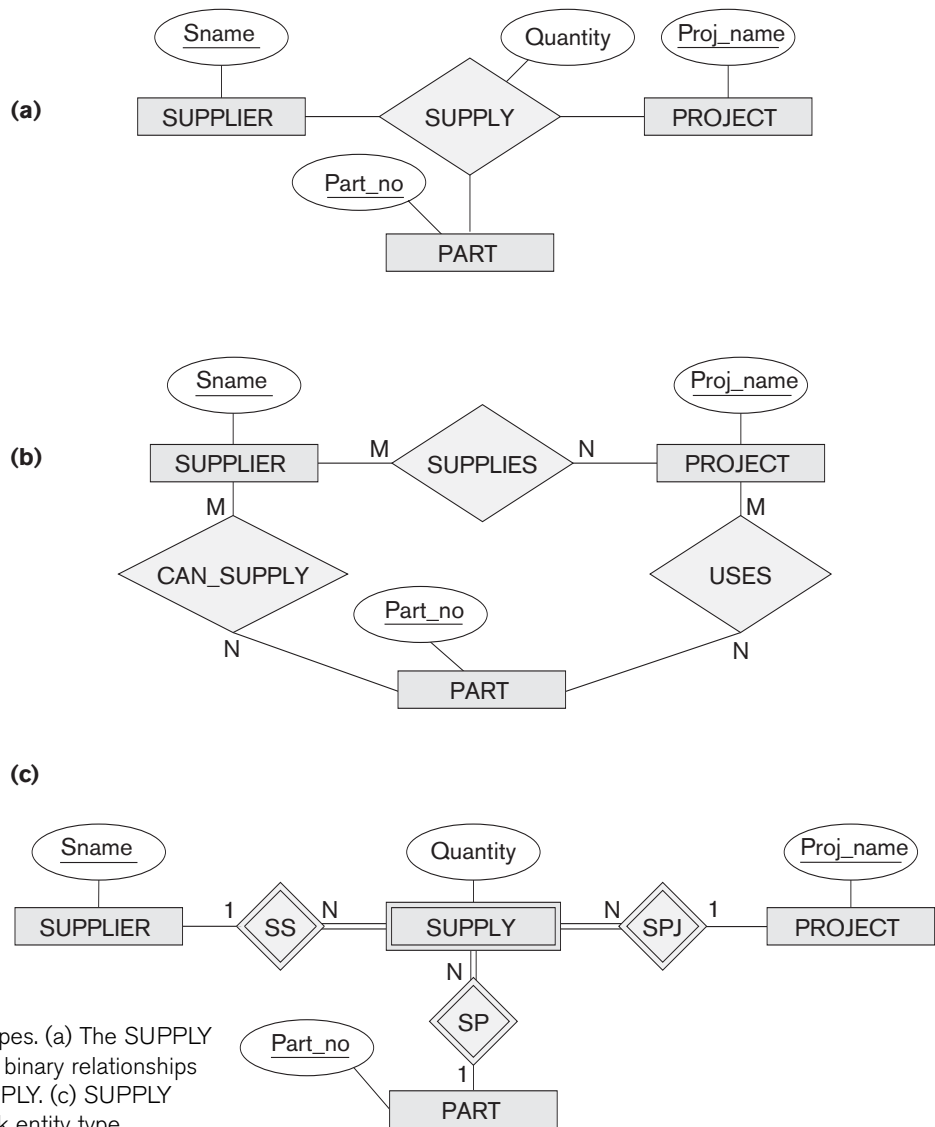
This section is not meant to be a complete description of UML class diagrams, but rather to illustrate one popular type of alternative diagrammatic notation that can be used for representing ER modeling concepts.

7.9 Relationship Types of Degree Higher than Two

In Section 7.4.2 we defined the **degree** of a relationship type as the number of participating entity types and called a relationship type of degree two *binary* and a relationship type of degree three *ternary*. In this section, we elaborate on the differences between binary and higher-degree relationships, when to choose higher-degree versus binary relationships, and how to specify constraints on higher-degree relationships.

7.9.1 Choosing between Binary and Ternary (or Higher-Degree) Relationships

The ER diagram notation for a ternary relationship type is shown in Figure 7.17(a), which displays the schema for the *SUPPLY* relationship type that was displayed at the entity set/relationship set or instance level in Figure 7.10. Recall that the relationship set of *SUPPLY* is a set of relationship instances (s, j, p) , where s is a *SUPPLIER* who is currently supplying a *PART* p to a *PROJECT* j . In general, a relationship type R of degree n will have n edges in an ER diagram, one connecting R to each participating entity type.

**Figure 7.17**

Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

Figure 7.17(b) shows an ER diagram for three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. In general, a ternary relationship type represents different information than do three binary relationship types. Consider the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. Suppose that CAN_SUPPLY, between SUPPLIER and PART, includes an instance (s, p) whenever supplier s can supply part p (to any project); USES, between PROJECT and PART, includes an instance (j, p) whenever project j uses part p ; and SUPPLIES, between SUPPLIER and PROJECT, includes an instance (s, j) whenever supplier s supplies

some part to project j . The existence of three relationship instances (s, p) , (j, p) , and (s, j) in CAN_SUPPLY, USES, and SUPPLIES, respectively, does not necessarily imply that an instance (s, j, p) exists in the ternary relationship SUPPLY, because the *meaning is different*. It is often tricky to decide whether a particular relationship should be represented as a relationship type of degree n or should be broken down into several relationship types of smaller degrees. The designer must base this decision on the semantics or meaning of the particular situation being represented. The typical solution is to include the ternary relationship *plus* one or more of the binary relationships, if they represent different meanings and if all are needed by the application.

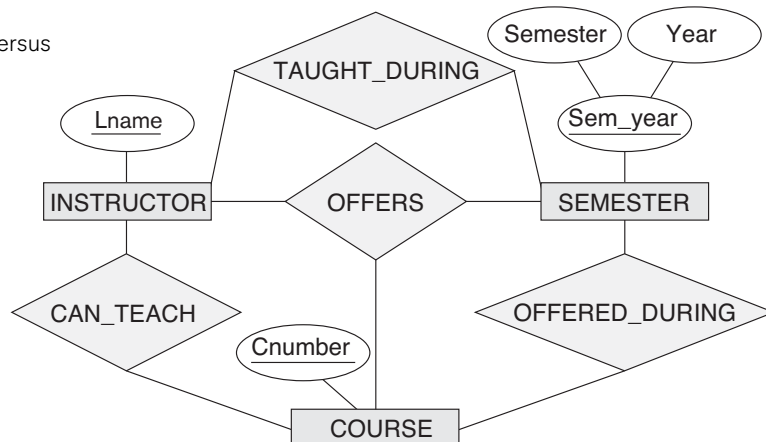
Some database design tools are based on variations of the ER model that permit only binary relationships. In this case, a ternary relationship such as SUPPLY must be represented as a weak entity type, with no partial key and with three identifying relationships. The three participating entity types SUPPLIER, PART, and PROJECT are together the owner entity types (see Figure 7.17(c)). Hence, an entity in the weak entity type SUPPLY in Figure 7.17(c) is identified by the combination of its three owner entities from SUPPLIER, PART, and PROJECT.

It is also possible to represent the ternary relationship as a regular entity type by introducing an artificial or surrogate key. In this example, a key attribute Supply_id could be used for the supply entity type, converting it into a regular entity type. Three binary N:1 relationships relate SUPPLY to the three participating entity types.

Another example is shown in Figure 7.18. The ternary relationship type OFFERS represents information on instructors offering courses during particular semesters; hence it includes a relationship instance (i, s, c) whenever INSTRUCTOR i offers COURSE c during SEMESTER s . The three binary relationship types shown in Figure 7.18 have the following meanings: CAN_TEACH relates a course to the instructors who *can teach* that course, TAUGHT_DURING relates a semester to the instructors who *taught some course* during that semester, and OFFERED_DURING

Figure 7.18

Another example of ternary versus binary relationship types.



relates a semester to the courses offered during that semester *by any instructor*. These ternary and binary relationships represent different information, but certain constraints should hold among the relationships. For example, a relationship instance (i, s, c) should not exist in OFFERS *unless* an instance (i, s) exists in TAUGHT_DURING, an instance (s, c) exists in OFFERED_DURING, and an instance (i, c) exists in CAN_TEACH. However, the reverse is not always true; we may have instances (i, s) , (s, c) , and (i, c) in the three binary relationship types with no corresponding instance (i, s, c) in OFFERS. Note that in this example, based on the meanings of the relationships, we can infer the instances of TAUGHT_DURING and OFFERED_DURING from the instances in OFFERS, but we cannot infer the instances of CAN_TEACH; therefore, TAUGHT_DURING and OFFERED_DURING are redundant and can be left out.

Although in general three binary relationships *cannot* replace a ternary relationship, they may do so under certain *additional constraints*. In our example, if the CAN_TEACH relationship is 1:1 (an instructor can teach one course, and a course can be taught by only one instructor), then the ternary relationship OFFERS can be left out because it can be inferred from the three binary relationships CAN_TEACH, TAUGHT_DURING, and OFFERED_DURING. The schema designer must analyze the meaning of each specific situation to decide which of the binary and ternary relationship types are needed.

Notice that it is possible to have a weak entity type with a ternary (or n -ary) identifying relationship type. In this case, the weak entity type can have *several* owner entity types. An example is shown in Figure 7.19. This example shows part of a database that keeps track of candidates interviewing for jobs at various companies, and may be part of an employment agency database, for example. In the requirements, a candidate can have multiple interviews with the same company (for example, with different company departments or on separate dates), but a job offer is made based on one of the interviews. Here, INTERVIEW is represented as a weak entity type with two owners CANDIDATE and COMPANY, and with the partial key Dept_date. An INTERVIEW entity is uniquely identified by a candidate, a company, and the combination of the date and department of the interview.

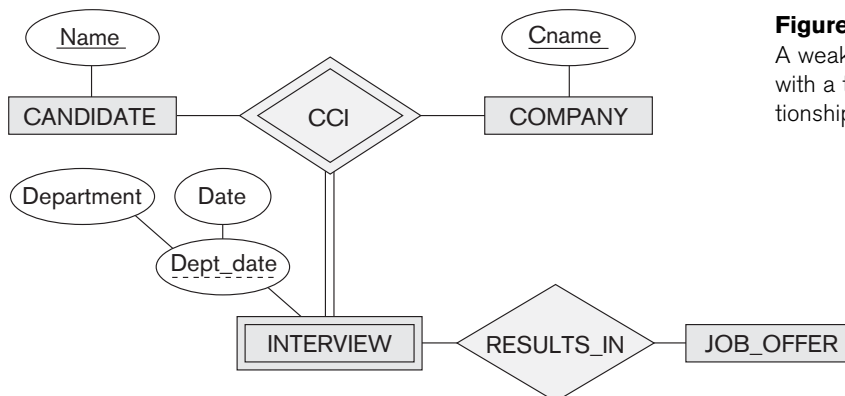


Figure 7.19
A weak entity type INTERVIEW
with a ternary identifying rela-
tionship type.

7.9.2 Constraints on Ternary (or Higher-Degree) Relationships

There are two notations for specifying structural constraints on n -ary relationships, and they specify different constraints. They should thus *both be used* if it is important to fully specify the structural constraints on a ternary or higher-degree relationship. The first notation is based on the cardinality ratio notation of binary relationships displayed in Figure 7.2. Here, a 1, M, or N is specified on each participation arc (both M and N symbols stand for *many* or *any number*).¹⁵ Let us illustrate this constraint using the SUPPLY relationship in Figure 7.17.

Recall that the relationship set of SUPPLY is a set of relationship instances (s, j, p) , where s is a SUPPLIER, j is a PROJECT, and p is a PART. Suppose that the constraint exists that for a particular project-part combination, only one supplier will be used (only one supplier supplies a particular part to a particular project). In this case, we place 1 on the SUPPLIER participation, and M, N on the PROJECT, PART participations in Figure 7.17. This specifies the constraint that a particular (j, p) combination can appear at most once in the relationship set because each such (PROJECT, PART) combination uniquely determines a single supplier. Hence, any relationship instance (s, j, p) is uniquely identified in the relationship set by its (j, p) combination, which makes (j, p) a key for the relationship set. In this notation, the participations that have a 1 specified on them are not required to be part of the identifying key for the relationship set.¹⁶ If all three cardinalities are M or N, then the key will be the combination of all three participants.

The second notation is based on the (min, max) notation displayed in Figure 7.15 for binary relationships. A (min, max) on a participation here specifies that each entity is related to at least *min* and at most *max relationship instances* in the relationship set. These constraints have no bearing on determining the key of an n -ary relationship, where $n > 2$,¹⁷ but specify a different type of constraint that places restrictions on how many relationship instances each entity can participate in.

7.10 Summary

In this chapter we presented the modeling concepts of a high-level conceptual data model, the Entity-Relationship (ER) model. We started by discussing the role that a high-level data model plays in the database design process, and then we presented a sample set of database requirements for the COMPANY database, which is one of the examples that is used throughout this book. We defined the basic ER model concepts of entities and their attributes. Then we discussed NULL values and presented

¹⁵This notation allows us to determine the key of the *relationship relation*, as we discuss in Chapter 9.

¹⁶This is also true for cardinality ratios of binary relationships.

¹⁷The (min, max) constraints can determine the keys for binary relationships, though.

the various types of attributes, which can be nested arbitrarily to produce complex attributes:

- Simple or atomic
- Composite
- Multivalued

We also briefly discussed stored versus derived attributes. Then we discussed the ER model concepts at the schema or “intension” level:

- Entity types and their corresponding entity sets
- Key attributes of entity types
- Value sets (domains) of attributes
- Relationship types and their corresponding relationship sets
- Participation roles of entity types in relationship types

We presented two methods for specifying the structural constraints on relationship types. The first method distinguished two types of structural constraints:

- Cardinality ratios (1:1, 1:N, M:N for binary relationships)
- Participation constraints (total, partial)

We noted that, alternatively, another method of specifying structural constraints is to specify minimum and maximum numbers (min, max) on the participation of each entity type in a relationship type. We discussed weak entity types and the related concepts of owner entity types, identifying relationship types, and partial key attributes.

Entity-Relationship schemas can be represented diagrammatically as ER diagrams. We showed how to design an ER schema for the COMPANY database by first defining the entity types and their attributes and then refining the design to include relationship types. We displayed the ER diagram for the COMPANY database schema. We discussed some of the basic concepts of UML class diagrams and how they relate to ER modeling concepts. We also described ternary and higher-degree relationship types in more detail, and discussed the circumstances under which they are distinguished from binary relationships.

The ER modeling concepts we have presented thus far—entity types, relationship types, attributes, keys, and structural constraints—can model many database applications. However, more complex applications—such as engineering design, medical information systems, and telecommunications—require additional concepts if we want to model them with greater accuracy. We discuss some advanced modeling concepts in Chapter 8 and revisit further advanced data modeling techniques in Chapter 26.

Review Questions

- 7.1. Discuss the role of a high-level data model in the database design process.
- 7.2. List the various cases where use of a NULL value would be appropriate.
- 7.3. Define the following terms: *entity*, *attribute*, *attribute value*, *relationship instance*, *composite attribute*, *multivalued attribute*, *derived attribute*, *complex attribute*, *key attribute*, and *value set (domain)*.
- 7.4. What is an entity type? What is an entity set? Explain the differences among an entity, an entity type, and an entity set.
- 7.5. Explain the difference between an attribute and a value set.
- 7.6. What is a relationship type? Explain the differences among a relationship instance, a relationship type, and a relationship set.
- 7.7. What is a participation role? When is it necessary to use role names in the description of relationship types?
- 7.8. Describe the two alternatives for specifying structural constraints on relationship types. What are the advantages and disadvantages of each?
- 7.9. Under what conditions can an attribute of a binary relationship type be migrated to become an attribute of one of the participating entity types?
- 7.10. When we think of relationships as attributes, what are the value sets of these attributes? What class of data models is based on this concept?
- 7.11. What is meant by a recursive relationship type? Give some examples of recursive relationship types.
- 7.12. When is the concept of a weak entity used in data modeling? Define the terms *owner entity type*, *weak entity type*, *identifying relationship type*, and *partial key*.
- 7.13. Can an identifying relationship of a weak entity type be of a degree greater than two? Give examples to illustrate your answer.
- 7.14. Discuss the conventions for displaying an ER schema as an ER diagram.
- 7.15. Discuss the naming conventions used for ER schema diagrams.

Exercises

- 7.16. Consider the following set of requirements for a UNIVERSITY database that is used to keep track of students' transcripts. This is similar but not identical to the database shown in Figure 1.2:
 - a. The university keeps track of each student's name, student number, Social Security number, current address and phone number, permanent address and phone number, birth date, sex, class (freshman, sophomore, ..., graduate), major department, minor department (if any), and degree program

(B.A., B.S., ..., Ph.D.). Some user applications need to refer to the city, state, and ZIP Code of the student's permanent address and to the student's last name. Both Social Security number and student number have unique values for each student.

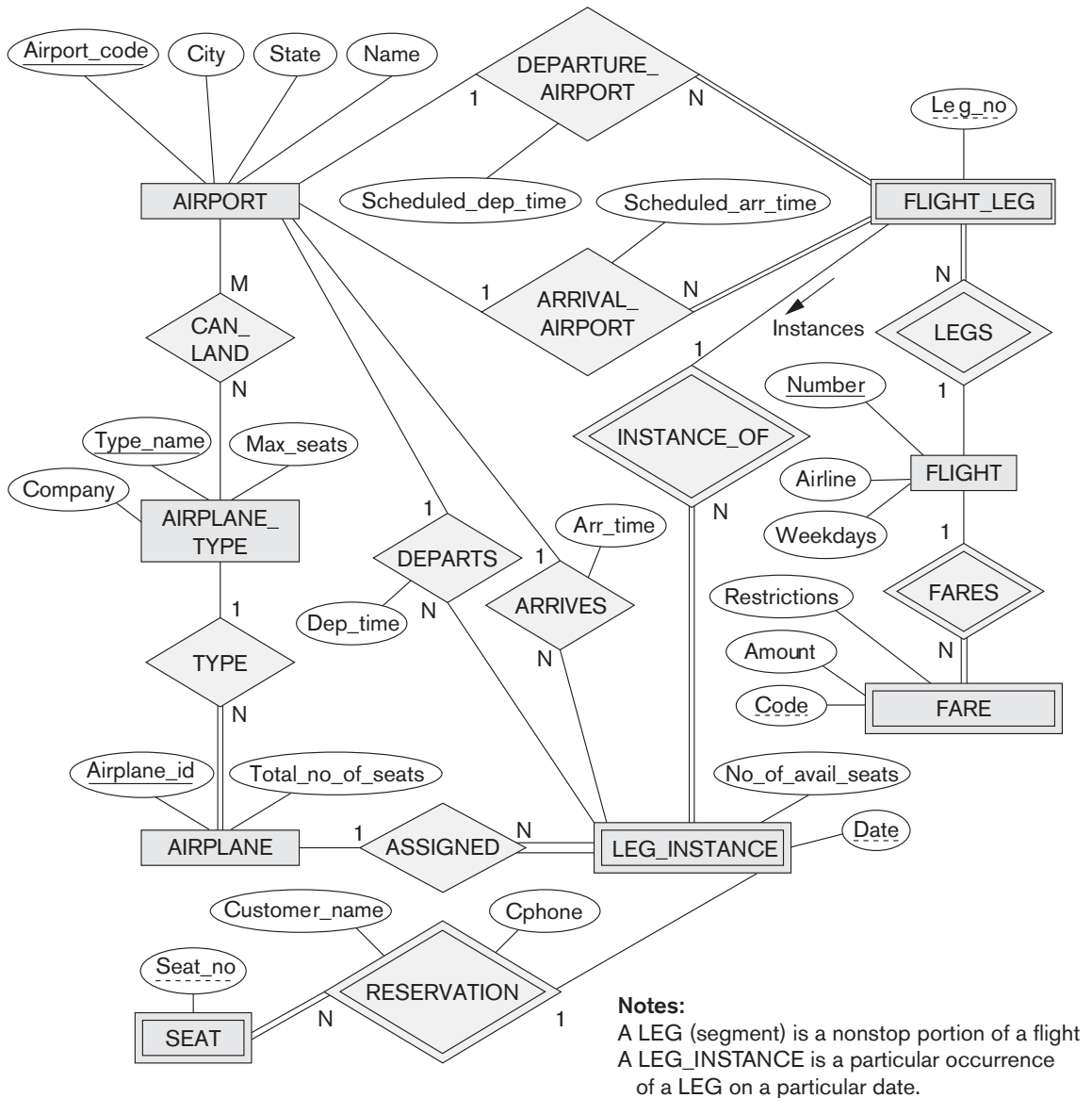
- b. Each department is described by a name, department code, office number, office phone number, and college. Both name and code have unique values for each department.
- c. Each course has a course name, description, course number, number of semester hours, level, and offering department. The value of the course number is unique for each course.
- d. Each section has an instructor, semester, year, course, and section number. The section number distinguishes sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ..., up to the number of sections taught during each semester.
- e. A grade report has a student, section, letter grade, and numeric grade (0, 1, 2, 3, or 4).

Design an ER schema for this application, and draw an ER diagram for the schema. Specify key attributes of each entity type, and structural constraints on each relationship type. Note any unspecified requirements, and make appropriate assumptions to make the specification complete.

- 7.17.** Composite and multivalued attributes can be nested to any number of levels. Suppose we want to design an attribute for a STUDENT entity type to keep track of previous college education. Such an attribute will have one entry for each college previously attended, and each such entry will be composed of college name, start and end dates, degree entries (degrees awarded at that college, if any), and transcript entries (courses completed at that college, if any). Each degree entry contains the degree name and the month and year the degree was awarded, and each transcript entry contains a course name, semester, year, and grade. Design an attribute to hold this information. Use the conventions in Figure 7.5.
- 7.18.** Show an alternative design for the attribute described in Exercise 7.17 that uses only entity types (including weak entity types, if needed) and relationship types.
- 7.19.** Consider the ER diagram in Figure 7.20, which shows a simplified schema for an airline reservations system. Extract from the ER diagram the requirements and constraints that produced this schema. Try to be as precise as possible in your requirements and constraints specification.
- 7.20.** In Chapters 1 and 2, we discussed the database environment and database users. We can consider many entity types to describe such an environment, such as DBMS, stored database, DBA, and catalog/data dictionary. Try to specify all the entity types that can fully describe a database system and its environment; then specify the relationship types among them, and draw an ER diagram to describe such a general database environment.

Figure 7.20

An ER diagram for an AIRLINE database schema.



7.21. Design an ER schema for keeping track of information about votes taken in the U.S. House of Representatives during the current two-year congressional session. The database needs to keep track of each U.S. STATE's Name (e.g., 'Texas', 'New York', 'California') and include the Region of the state (whose domain is {'Northeast', 'Midwest', 'Southeast', 'Southwest', 'West'}). Each

CONGRESS_PERSON in the House of Representatives is described by his or her Name, plus the District represented, the Start_date when the congressperson was first elected, and the political Party to which he or she belongs (whose domain is {'Republican', 'Democrat', 'Independent', 'Other'}). The database keeps track of each BILL (i.e., proposed law), including the Bill_name, the Date_of_vote on the bill, whether the bill Passed_or_failed (whose domain is {'Yes', 'No'}), and the Sponsor (the congressperson(s) who sponsored—that is, proposed—the bill). The database also keeps track of how each congressperson voted on each bill (domain of Vote attribute is {'Yes', 'No', 'Abstain', 'Absent'}). Draw an ER schema diagram for this application. State clearly any assumptions you make.

- 7.22.** A database is being constructed to keep track of the teams and games of a sports league. A team has a number of players, not all of whom participate in each game. It is desired to keep track of the players participating in each game for each team, the positions they played in that game, and the result of the game. Design an ER schema diagram for this application, stating any assumptions you make. Choose your favorite sport (e.g., soccer, baseball, football).
- 7.23.** Consider the ER diagram shown in Figure 7.21 for part of a BANK database. Each bank can have multiple branches, and each branch can have multiple accounts and loans.
- List the strong (nonweak) entity types in the ER diagram.
 - Is there a weak entity type? If so, give its name, partial key, and identifying relationship.
 - What constraints do the partial key and the identifying relationship of the weak entity type specify in this diagram?
 - List the names of all relationship types, and specify the (min, max) constraint on each participation of an entity type in a relationship type. Justify your choices.
 - List concisely the user requirements that led to this ER schema design.
 - Suppose that every customer must have at least one account but is restricted to at most two loans at a time, and that a bank branch cannot have more than 1,000 loans. How does this show up on the (min, max) constraints?
- 7.24.** Consider the ER diagram in Figure 7.22. Assume that an employee may work in up to two departments or may not be assigned to any department. Assume that each department must have one and may have up to three phone numbers. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions you make.* Under what conditions would the relationship HAS_PHONE be redundant in this example?
- 7.25.** Consider the ER diagram in Figure 7.23. Assume that a course may or may not use a textbook, but that a text by definition is a book that is used in some course. A course may not use more than five books. Instructors teach from

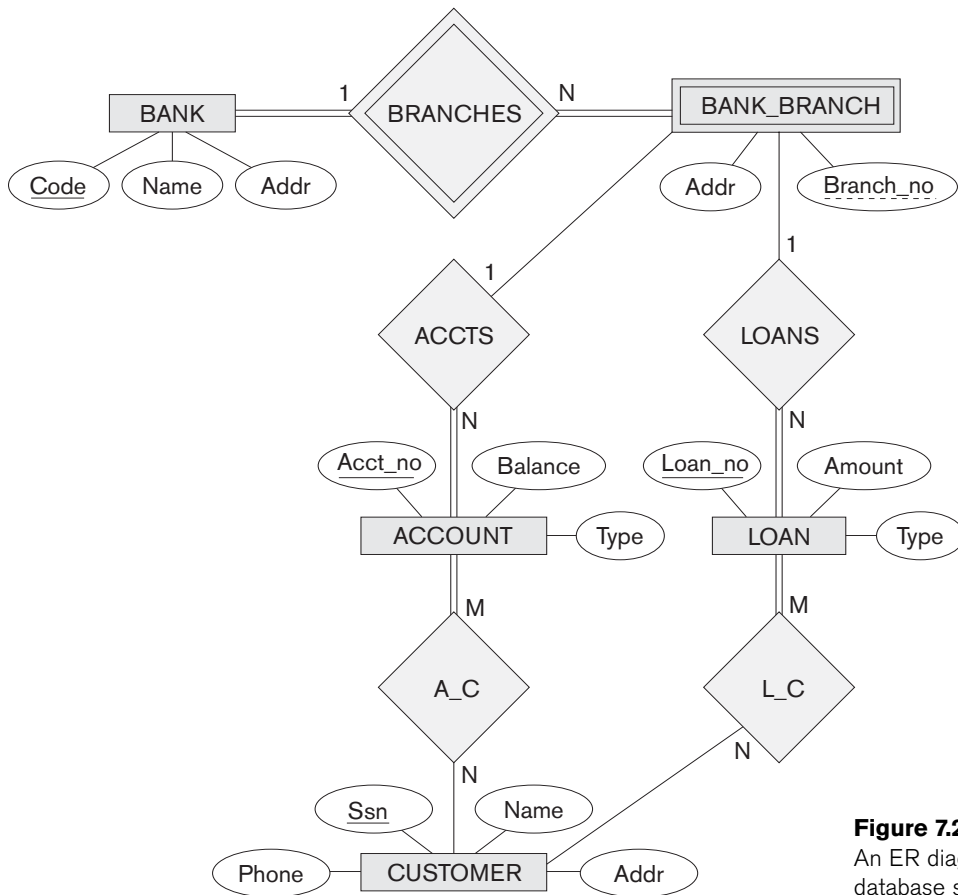


Figure 7.21
An ER diagram for a BANK database schema.

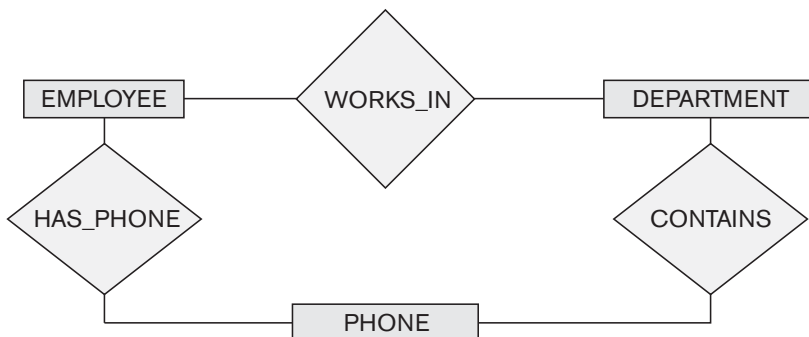


Figure 7.22
Part of an ER diagram for a COMPANY database.

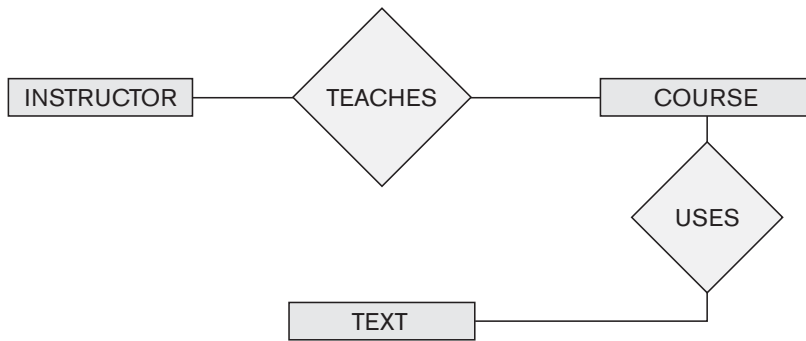


Figure 7.23
Part of an ER diagram
for a COURSES data-
base.

two to four courses. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions you make.* If we add the relationship ADOPTS, to indicate the textbook(s) that an instructor uses for a course, should it be a binary relationship between INSTRUCTOR and TEXT, or a ternary relationship between all three entity types? What (min, max) constraints would you put on it? Why?

- 7.26.** Consider an entity type SECTION in a UNIVERSITY database, which describes the section offerings of courses. The attributes of SECTION are Section_number, Semester, Year, Course_number, Instructor, Room_no (where section is taught), Building (where section is taught), Weekdays (domain is the possible combinations of weekdays in which a section can be offered {'MWF', 'MW', 'TT', and so on}), and Hours (domain is all possible time periods during which sections are offered {'9–9:50 A.M.', '10–10:50 A.M.', ..., '3:30–4:50 P.M.', '5:30–6:20 P.M.', and so on}). Assume that Section_number is unique for each course within a particular semester/year combination (that is, if a course is offered multiple times during a particular semester, its section offerings are numbered 1, 2, 3, and so on). There are several composite keys for section, and some attributes are components of more than one key. Identify three composite keys, and show how they can be represented in an ER schema diagram.
- 7.27.** Cardinality ratios often dictate the detailed design of a database. The cardinality ratio depends on the real-world meaning of the entity types involved and is defined by the specific application. For the following binary relationships, suggest cardinality ratios based on the common-sense meaning of the entity types. Clearly state any assumptions you make.

Entity 1	Cardinality Ratio	Entity 2
1. STUDENT	_____	SOCIAL_SECURITY_CARD
2. STUDENT	_____	TEACHER
3. CLASSROOM	_____	WALL

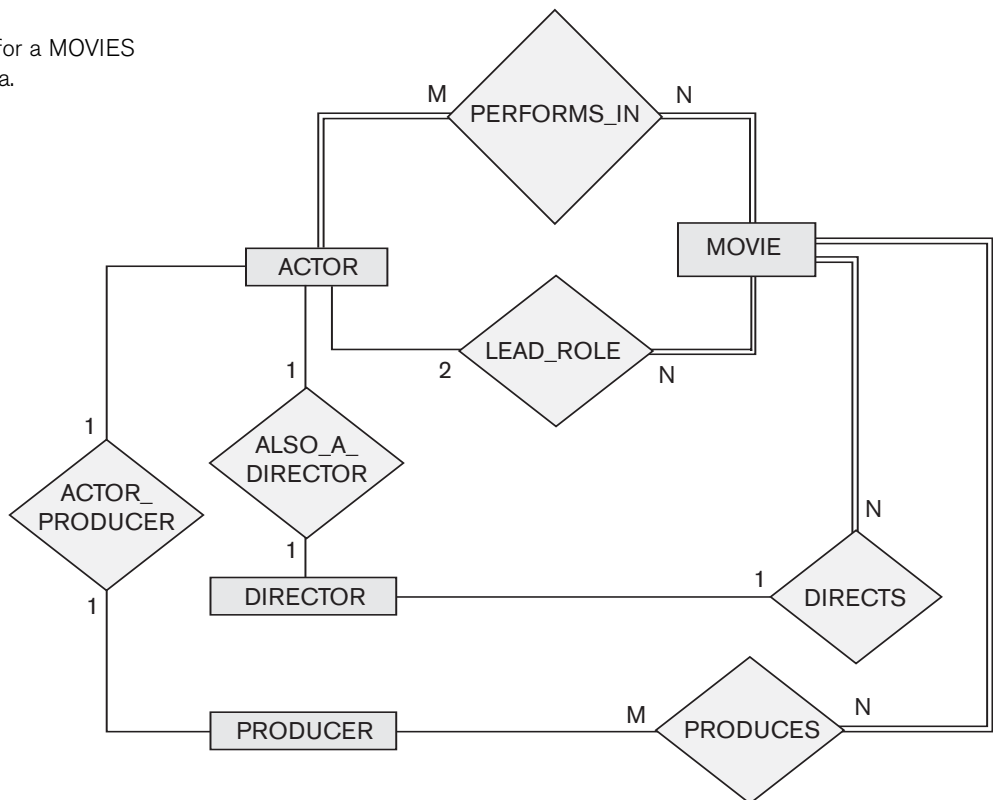
4. COUNTRY	_____	CURRENT_PRESIDENT
5. COURSE	_____	TEXTBOOK
6. ITEM (that can be found in an order)	_____	ORDER
7. STUDENT	_____	CLASS
8. CLASS	_____	INSTRUCTOR
9. INSTRUCTOR	_____	OFFICE
10. EBAY_AUCTION _ITEM	_____	EBAY_BID

7.28. Consider the ER schema for the MOVIES database in Figure 7.24.

Assume that MOVIES is a populated database. ACTOR is used as a generic term and includes actresses. Given the constraints shown in the ER schema, respond to the following statements with *True*, *False*, or *Maybe*. Assign a response of *Maybe* to statements that, while not explicitly shown to be *True*, cannot be proven *False* based on the schema as shown. Justify each answer.

Figure 7.24

An ER diagram for a MOVIES database schema.



- a. There are no actors in this database that have been in no movies.
- b. There are some actors who have acted in more than ten movies.
- c. Some actors have done a lead role in multiple movies.
- d. A movie can have only a maximum of two lead actors.
- e. Every director has been an actor in some movie.
- f. No producer has ever been an actor.
- g. A producer cannot be an actor in some other movie.
- h. There are movies with more than a dozen actors.
- i. Some producers have been a director as well.
- j. Most movies have one director and one producer.
- k. Some movies have one director but several producers.
- l. There are some actors who have done a lead role, directed a movie, and produced some movie.
- m. No movie has a director who also acted in that movie.

7.29. Given the ER schema for the MOVIES database in Figure 7.24, draw an instance diagram using three movies that have been released recently. Draw instances of each entity type: MOVIES, ACTORS, PRODUCERS, DIRECTORS involved; make up instances of the relationships as they exist in reality for those movies.

7.30. Illustrate the UML Diagram for Exercise 7.16. Your UML design should observe the following requirements:

- a. A student should have the ability to compute his/her GPA and add or drop majors and minors.
- b. Each department should be able to add or delete courses and hire or terminate faculty.
- c. Each instructor should be able to assign or change a student's grade for a course.

Note: Some of these functions may be spread over multiple classes.

Laboratory Exercises

7.31. Consider the UNIVERSITY database described in Exercise 7.16. Build the ER schema for this database using a data modeling tool such as ERwin or Rational Rose.

7.32. Consider a MAIL_ORDER database in which employees take orders for parts from customers. The data requirements are summarized as follows:

- The mail order company has employees, each identified by a unique employee number, first and last name, and Zip Code.
- Each customer of the company is identified by a unique customer number, first and last name, and Zip Code.

- Each part sold by the company is identified by a unique part number, a part name, price, and quantity in stock.
- Each order placed by a customer is taken by an employee and is given a unique order number. Each order contains specified quantities of one or more parts. Each order has a date of receipt as well as an expected ship date. The actual ship date is also recorded.

Design an Entity-Relationship diagram for the mail order database and build the design using a data modeling tool such as ERwin or Rational Rose.

7.33. Consider a MOVIE database in which data is recorded about the movie industry. The data requirements are summarized as follows:

- Each movie is identified by title and year of release. Each movie has a length in minutes. Each has a production company, and each is classified under one or more genres (such as horror, action, drama, and so forth). Each movie has one or more directors and one or more actors appear in it. Each movie also has a plot outline. Finally, each movie has zero or more quotable quotes, each of which is spoken by a particular actor appearing in the movie.
- Actors are identified by name and date of birth and appear in one or more movies. Each actor has a role in the movie.
- Directors are also identified by name and date of birth and direct one or more movies. It is possible for a director to act in a movie (including one that he or she may also direct).
- Production companies are identified by name and each has an address. A production company produces one or more movies.

Design an Entity-Relationship diagram for the movie database and enter the design using a data modeling tool such as ERwin or Rational Rose.

7.34. Consider a CONFERENCE_REVIEW database in which researchers submit their research papers for consideration. Reviews by reviewers are recorded for use in the paper selection process. The database system caters primarily to reviewers who record answers to evaluation questions for each paper they review and make recommendations regarding whether to accept or reject the paper. The data requirements are summarized as follows:

- Authors of papers are uniquely identified by e-mail id. First and last names are also recorded.
- Each paper is assigned a unique identifier by the system and is described by a title, abstract, and the name of the electronic file containing the paper.
- A paper may have multiple authors, but one of the authors is designated as the contact author.
- Reviewers of papers are uniquely identified by e-mail address. Each reviewer's first name, last name, phone number, affiliation, and topics of interest are also recorded.

- Each paper is assigned between two and four reviewers. A reviewer rates each paper assigned to him or her on a scale of 1 to 10 in four categories: technical merit, readability, originality, and relevance to the conference. Finally, each reviewer provides an overall recommendation regarding each paper.
- Each review contains two types of written comments: one to be seen by the review committee only and the other as feedback to the author(s).

Design an Entity-Relationship diagram for the `CONFERENCE_REVIEW` database and build the design using a data modeling tool such as ERwin or Rational Rose.

- 7.35.** Consider the ER diagram for the `AIRLINE` database shown in Figure 7.20. Build this design using a data modeling tool such as ERwin or Rational Rose.

Selected Bibliography

The Entity-Relationship model was introduced by Chen (1976), and related work appears in Schmidt and Swenson (1975), Wiederhold and Elmasri (1979), and Senko (1975). Since then, numerous modifications to the ER model have been suggested. We have incorporated some of these in our presentation. Structural constraints on relationships are discussed in Abrial (1974), Elmasri and Wiederhold (1980), and Lenzerini and Santucci (1983). Multivalued and composite attributes are incorporated in the ER model in Elmasri et al. (1985). Although we did not discuss languages for the ER model and its extensions, there have been several proposals for such languages. Elmasri and Wiederhold (1981) proposed the GORDAS query language for the ER model. Another ER query language was proposed by Markowitz and Raz (1983). Senko (1980) presented a query language for Senko's DIAM model. A formal set of operations called the ER algebra was presented by Parent and Spaccapietra (1985). Gogolla and Hohenstein (1991) presented another formal language for the ER model. Campbell et al. (1985) presented a set of ER operations and showed that they are relationally complete. A conference for the dissemination of research results related to the ER model has been held regularly since 1979. The conference, now known as the International Conference on Conceptual Modeling, has been held in Los Angeles (ER 1979, ER 1983, ER 1997), Washington, D.C. (ER 1981), Chicago (ER 1985), Dijon, France (ER 1986), New York City (ER 1987), Rome (ER 1988), Toronto (ER 1989), Lausanne, Switzerland (ER 1990), San Mateo, California (ER 1991), Karlsruhe, Germany (ER 1992), Arlington, Texas (ER 1993), Manchester, England (ER 1994), Brisbane, Australia (ER 1995), Cottbus, Germany (ER 1996), Singapore (ER 1998), Paris, France (ER 1999), Salt Lake City, Utah (ER 2000), Yokohama, Japan (ER 2001), Tampere, Finland (ER 2002), Chicago, Illinois (ER 2003), Shanghai, China (ER 2004), Klagenfurt, Austria (ER 2005), Tucson, Arizona (ER 2006), Auckland, New Zealand (ER 2007), Barcelona, Catalonia, Spain (ER 2008), and Gramado, RS, Brazil (ER 2009). The 2010 conference is to be held in Vancouver, BC, Canada.

This page intentionally left blank

The Enhanced Entity-Relationship (EER) Model

The ER modeling concepts discussed in Chapter 7 are sufficient for representing many database schemas for *traditional* database applications, which include many data-processing applications in business and industry. Since the late 1970s, however, designers of database applications have tried to design more accurate database schemas that reflect the data properties and constraints more precisely. This was particularly important for newer applications of database technology, such as databases for engineering design and manufacturing (CAD/CAM),¹ telecommunications, complex software systems, and Geographic Information Systems (GIS), among many other applications. These types of databases have more complex requirements than do the more traditional applications. This led to the development of additional *semantic data modeling* concepts that were incorporated into conceptual data models such as the ER model. Various semantic data models have been proposed in the literature. Many of these concepts were also developed independently in related areas of computer science, such as the **knowledge representation** area of artificial intelligence and the **object modeling** area in software engineering.

In this chapter, we describe features that have been proposed for semantic data models, and show how the ER model can be enhanced to include these concepts, leading to the **Enhanced ER (EER)** model.² We start in Section 8.1 by incorporating the concepts of *class/subclass relationships* and *type inheritance* into the ER model. Then, in Section 8.2, we add the concepts of *specialization* and *generalization*. Section 8.3

¹CAD/CAM stands for computer-aided design/computer-aided manufacturing.

²EER has also been used to stand for *Extended* ER model.

discusses the various types of *constraints* on specialization/generalization, and Section 8.4 shows how the UNION construct can be modeled by including the concept of *category* in the EER model. Section 8.5 gives a sample UNIVERSITY database schema in the EER model and summarizes the EER model concepts by giving formal definitions. We will use the terms *object* and *entity* interchangeably in this chapter, because many of these concepts are commonly used in object-oriented models.

We present the UML class diagram notation for representing specialization and generalization in Section 8.6, and briefly compare these with EER notation and concepts. This serves as an example of alternative notation, and is a continuation of Section 7.8, which presented basic UML class diagram notation that corresponds to the basic ER model. In Section 8.7, we discuss the fundamental abstractions that are used as the basis of many semantic data models. Section 8.8 summarizes the chapter.

For a detailed introduction to conceptual modeling, Chapter 8 should be considered a continuation of Chapter 7. However, if only a basic introduction to ER modeling is desired, this chapter may be omitted. Alternatively, the reader may choose to skip some or all of the later sections of this chapter (Sections 8.4 through 8.8).

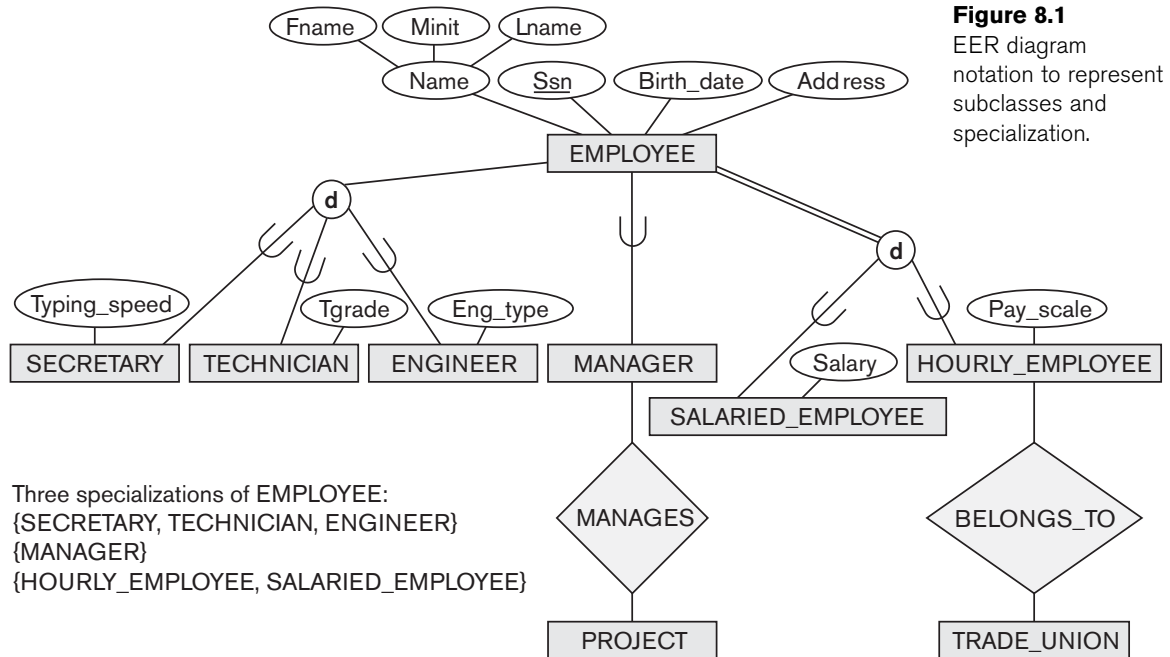
8.1 Subclasses, Superclasses, and Inheritance

The EER model includes *all the modeling concepts of the ER model* that were presented in Chapter 7. In addition, it includes the concepts of **subclass** and **superclass** and the related concepts of **specialization** and **generalization** (see Sections 8.2 and 8.3). Another concept included in the EER model is that of a **category** or **union type** (see Section 8.4), which is used to represent a collection of objects (entities) that is the *union* of objects of different entity types. Associated with these concepts is the important mechanism of **attribute and relationship inheritance**. Unfortunately, no standard terminology exists for these concepts, so we use the most common terminology. Alternative terminology is given in footnotes. We also describe a diagrammatic technique for displaying these concepts when they arise in an EER schema. We call the resulting schema diagrams **enhanced ER** or **EER diagrams**.

The first Enhanced ER (EER) model concept we take up is that of a **subtype** or **subclass** of an entity type. As we discussed in Chapter 7, an entity type is used to represent both a *type of entity* and the *entity set* or *collection of entities of that type* that exist in the database. For example, the entity type EMPLOYEE describes the type (that is, the attributes and relationships) of each employee entity, and also refers to the current set of EMPLOYEE entities in the COMPANY database. In many cases an entity type has numerous subgroupings or subtypes of its entities that are meaningful and need to be represented explicitly because of their significance to the database application. For example, the entities that are members of the EMPLOYEE entity type may be distinguished further into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, and so on. The set of entities in each of the latter groupings is a subset of the entities that belong to the EMPLOYEE entity set, meaning that every entity that is a member of one of these

subgroupings is also an employee. We call each of these subgroupings a **subclass** or **subtype** of the EMPLOYEE entity type, and the EMPLOYEE entity type is called the **superclass** or **supertype** for each of these subclasses. Figure 8.1 shows how to represent these concepts diagrammatically in EER diagrams. (The circle notation in Figure 8.1 will be explained in Section 8.2.)

We call the relationship between a superclass and any one of its subclasses a **superclass/subclass** or **supertype/subtype** or simply **class/subclass relationship**.³ In our previous example, EMPLOYEE/SECRETARY and EMPLOYEE/TECHNICIAN are two class/subclass relationships. Notice that a member entity of the subclass represents the *same real-world entity* as some member of the superclass; for example, a SECRETARY entity 'Joan Logano' is also the EMPLOYEE 'Joan Logano.' Hence, the subclass member is the same as the entity in the superclass, but in a distinct *specific role*. When we implement a superclass/subclass relationship in the database system, however, we may represent a member of the subclass as a distinct database object—say, a distinct record that is related via the key attribute to its superclass entity. In Section 9.2, we discuss various options for representing superclass/subclass relationships in relational databases.



³A class/subclass relationship is often called an **IS-A** (or **IS-AN**) **relationship** because of the way we refer to the concept. We say a SECRETARY *is an* EMPLOYEE, a TECHNICIAN *is an* EMPLOYEE, and so on.

An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass. Such an entity can be included optionally as a member of any number of subclasses. For example, a salaried employee who is also an engineer belongs to the two subclasses `ENGINEER` and `SALARIED_EMPLOYEE` of the `EMPLOYEE` entity type. However, it is not necessary that every entity in a superclass is a member of some subclass.

An important concept associated with subclasses (subtypes) is that of **type inheritance**. Recall that the *type* of an entity is defined by the attributes it possesses and the relationship types in which it participates. Because an entity in the subclass represents the same real-world entity from the superclass, it should possess values for its specific attributes *as well as* values of its attributes as a member of the superclass. We say that an entity that is a member of a subclass **inherits** all the attributes of the entity as a member of the superclass. The entity also inherits all the relationships in which the superclass participates. Notice that a subclass, with its own specific (or local) attributes and relationships together with all the attributes and relationships it inherits from the superclass, can be considered an *entity type* in its own right.⁴

8.2 Specialization and Generalization

8.2.1 Specialization

Specialization is the process of defining a *set of subclasses* of an entity type; this entity type is called the **superclass** of the specialization. The set of subclasses that forms a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass. For example, the set of subclasses {`SECRETARY`, `ENGINEER`, `TECHNICIAN`} is a specialization of the superclass `EMPLOYEE` that distinguishes among employee entities based on the *job type* of each employee entity. We may have several specializations of the same entity type based on different distinguishing characteristics. For example, another specialization of the `EMPLOYEE` entity type may yield the set of subclasses {`SALARIED_EMPLOYEE`, `HOURLY_EMPLOYEE`}; this specialization distinguishes among employees based on the *method of pay*.

Figure 8.1 shows how we represent a specialization diagrammatically in an EER diagram. The subclasses that define a specialization are attached by lines to a circle that represents the specialization, which is connected in turn to the superclass. The *subset symbol* on each line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship.⁵ Attributes that apply only to entities of a particular subclass—such as `TypingSpeed` of `SECRETARY`—are attached to the rectangle representing that subclass. These are called **specific attributes** (or **local**

⁴In some object-oriented programming languages, a common restriction is that an entity (or object) has *only one type*. This is generally too restrictive for conceptual database modeling.

⁵There are many alternative notations for specialization; we present the UML notation in Section 8.6 and other proposed notations in Appendix A.

attributes) of the subclass. Similarly, a subclass can participate in **specific relationship types**, such as the `HOURLY_EMPLOYEE` subclass participating in the `BELONGS_TO` relationship in Figure 8.1. We will explain the **d** symbol in the circles in Figure 8.1 and additional EER diagram notation shortly.

Figure 8.2 shows a few entity instances that belong to subclasses of the `{SECRETARY, ENGINEER, TECHNICIAN}` specialization. Again, notice that an entity that belongs to a subclass represents *the same real-world entity* as the entity connected to it in the `EMPLOYEE` superclass, even though the same entity is shown twice; for example, e_1 is shown in both `EMPLOYEE` and `SECRETARY` in Figure 8.2. As the figure suggests, a superclass/subclass relationship such as `EMPLOYEE/SECRETARY` somewhat resembles a 1:1 relationship *at the instance level* (see Figure 7.12). The main difference is that in a 1:1 relationship two *distinct entities* are related, whereas in a superclass/subclass relationship the entity in the subclass is the same real-world entity as the entity in the superclass but is playing a *specialized role*—for example, an `EMPLOYEE` specialized in the role of `SECRETARY`, or an `EMPLOYEE` specialized in the role of `TECHNICIAN`.

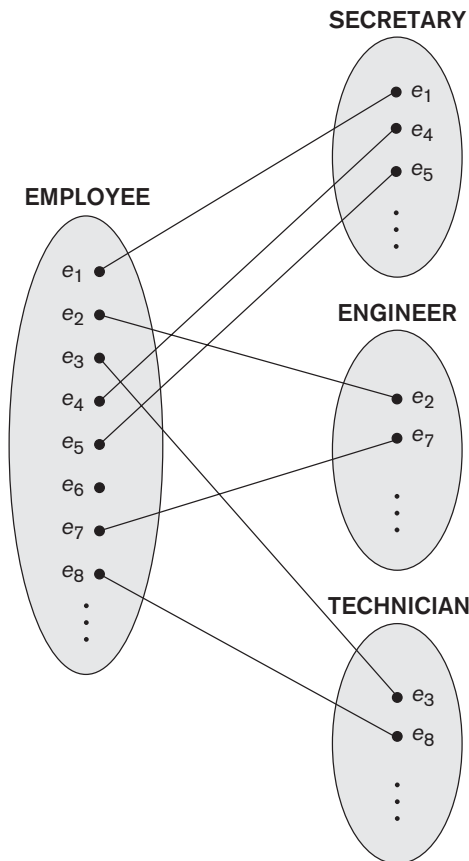


Figure 8.2
Instances of a specialization.

There are two main reasons for including class/subclass relationships and specializations in a data model. The first is that certain attributes may apply to some but not all entities of the superclass. A subclass is defined in order to group the entities to which these attributes apply. The members of the subclass may still share the majority of their attributes with the other members of the superclass. For example, in Figure 8.1 the SECRETARY subclass has the specific attribute *Typing_speed*, whereas the ENGINEER subclass has the specific attribute *Eng_type*, but SECRETARY and ENGINEER share their other inherited attributes from the EMPLOYEE entity type.

The second reason for using subclasses is that some relationship types may be participated in only by entities that are members of the subclass. For example, if only HOURLY_EMPLOYEES can belong to a trade union, we can represent that fact by creating the subclass HOURLY_EMPLOYEE of EMPLOYEE and relating the subclass to an entity type TRADE_UNION via the BELONGS_TO relationship type, as illustrated in Figure 8.1.

In summary, the specialization process allows us to do the following:

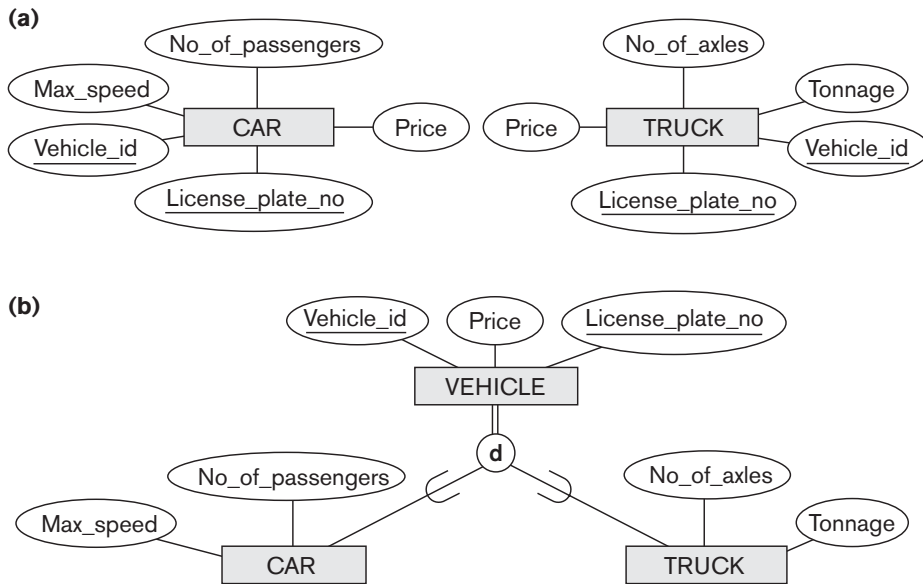
- Define a set of subclasses of an entity type
- Establish additional specific attributes with each subclass
- Establish additional specific relationship types between each subclass and other entity types or other subclasses

8.2.2 Generalization

We can think of a *reverse process* of abstraction in which we suppress the differences among several entity types, identify their common features, and **generalize** them into a single **superclass** of which the original entity types are special **subclasses**. For example, consider the entity types CAR and TRUCK shown in Figure 8.3(a). Because they have several common attributes, they can be generalized into the entity type VEHICLE, as shown in Figure 8.3(b). Both CAR and TRUCK are now subclasses of the **generalized superclass** VEHICLE. We use the term **generalization** to refer to the process of defining a generalized entity type from the given entity types.

Notice that the generalization process can be viewed as being functionally the inverse of the specialization process. Hence, in Figure 8.3 we can view {CAR, TRUCK} as a specialization of VEHICLE, rather than viewing VEHICLE as a generalization of CAR and TRUCK. Similarly, in Figure 8.1 we can view EMPLOYEE as a generalization of SECRETARY, TECHNICIAN, and ENGINEER. A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclasses represent a specialization. We will *not* use this notation because the decision as to which process is followed in a particular situation is often subjective. Appendix A gives some of the suggested alternative diagrammatic notations for schema diagrams and class diagrams.

So far we have introduced the concepts of subclasses and superclass/subclass relationships, as well as the specialization and generalization processes. In general, a

**Figure 8.3**

Generalization. (a) Two entity types, CAR and TRUCK. (b) Generalizing CAR and TRUCK into the superclass VEHICLE.

superclass or subclass represents a collection of entities of the same type and hence also describes an *entity type*; that is why superclasses and subclasses are all shown in rectangles in EER diagrams, like entity types. Next, we discuss the properties of specializations and generalizations in more detail.

8.3 Constraints and Characteristics of Specialization and Generalization Hierarchies

First, we discuss constraints that apply to a single specialization or a single generalization. For brevity, our discussion refers only to *specialization* even though it applies to *both* specialization and generalization. Then, we discuss differences between specialization/generalization *lattices* (*multiple inheritance*) and *hierarchies* (*single inheritance*), and elaborate on the differences between the specialization and generalization processes during conceptual database schema design.

8.3.1 Constraints on Specialization and Generalization

In general, we may have several specializations defined on the same entity type (or superclass), as shown in Figure 8.1. In such a case, entities may belong to subclasses

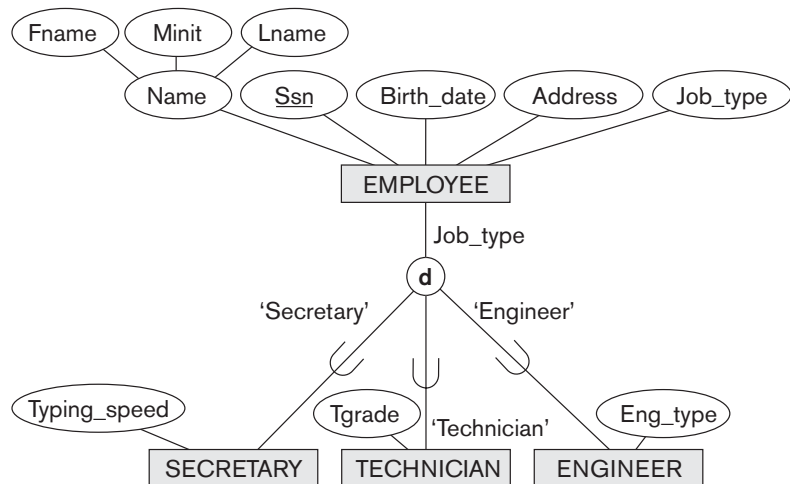
in each of the specializations. However, a specialization may also consist of a *single* subclass only, such as the {MANAGER} specialization in Figure 8.1; in such a case, we do not use the circle notation.

In some specializations we can determine exactly the entities that will become members of each subclass by placing a condition on the value of some attribute of the superclass. Such subclasses are called **predicate-defined** (or **condition-defined**) **subclasses**. For example, if the EMPLOYEE entity type has an attribute Job_type, as shown in Figure 8.4, we can specify the condition of membership in the SECRETARY subclass by the condition (Job_type = 'Secretary'), which we call the **defining predicate** of the subclass. This condition is a *constraint* specifying that exactly those entities of the EMPLOYEE entity type whose attribute value for Job_type is 'Secretary' belong to the subclass. We display a predicate-defined subclass by writing the predicate condition next to the line that connects the subclass to the specialization circle.

If *all* subclasses in a specialization have their membership condition on the *same* attribute of the superclass, the specialization itself is called an **attribute-defined specialization**, and the attribute is called the **defining attribute** of the specialization.⁶ In this case, all the entities with the same value for the attribute belong to the same subclass. We display an attribute-defined specialization by placing the defining attribute name next to the arc from the circle to the superclass, as shown in Figure 8.4.

When we do not have a condition for determining membership in a subclass, the subclass is called **user-defined**. Membership in such a subclass is determined by the database users when they apply the operation to add an entity to the subclass; hence, membership is *specified individually for each entity by the user*, not by any condition that may be evaluated automatically.

Figure 8.4
EER diagram notation
for an attribute-defined
specialization on
Job_type.



⁶Such an attribute is called a *discriminator* in UML terminology.

Two other constraints may apply to a specialization. The first is the **disjointness (or disjointedness) constraint**, which specifies that the subclasses of the specialization must be disjoint. This means that an entity can be a member of *at most* one of the subclasses of the specialization. A specialization that is attribute-defined implies the disjointness constraint (if the attribute used to define the membership predicate is single-valued). Figure 8.4 illustrates this case, where the **d** in the circle stands for *disjoint*. The **d** notation also applies to user-defined subclasses of a specialization that must be disjoint, as illustrated by the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Figure 8.1. If the subclasses are not constrained to be disjoint, their sets of entities may be **overlapping**; that is, the same (real-world) entity may be a member of more than one subclass of the specialization. This case, which is the default, is displayed by placing an **o** in the circle, as shown in Figure 8.5.

The second constraint on specialization is called the **completeness (or totalness) constraint**, which may be total or partial. A **total specialization** constraint specifies that *every* entity in the superclass must be a member of at least one subclass in the specialization. For example, if every EMPLOYEE must be either an HOURLY_EMPLOYEE or a SALARIED_EMPLOYEE, then the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Figure 8.1 is a total specialization of EMPLOYEE. This is shown in EER diagrams by using a double line to connect the superclass to the circle. A single line is used to display a **partial specialization**, which allows an entity not to belong to any of the subclasses. For example, if some EMPLOYEE entities do not belong to any of the subclasses {SECRETARY, ENGINEER, TECHNICIAN} in Figures 8.1 and 8.4, then that specialization is partial.⁷

Notice that the disjointness and completeness constraints are *independent*. Hence, we have the following four possible constraints on specialization:

- Disjoint, total
- Disjoint, partial
- Overlapping, total
- Overlapping, partial

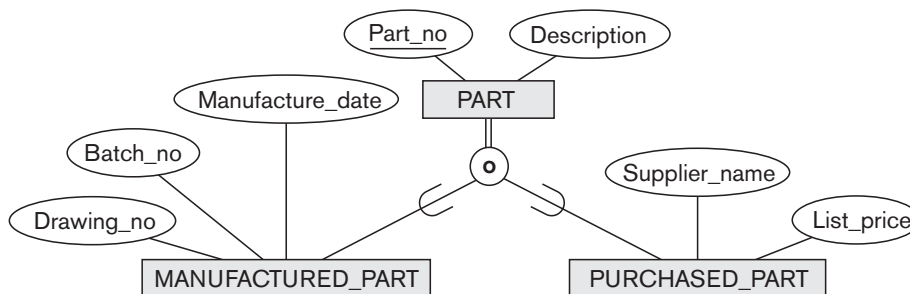


Figure 8.5
EER diagram notation
for an overlapping
(nondisjoint)
specialization.

⁷The notation of using single or double lines is similar to that for partial or total participation of an entity type in a relationship type, as described in Chapter 7.

Of course, the correct constraint is determined from the real-world meaning that applies to each specialization. In general, a superclass that was identified through the *generalization* process usually is **total**, because the superclass is *derived from* the subclasses and hence contains only the entities that are in the subclasses.

Certain insertion and deletion rules apply to specialization (and generalization) as a consequence of the constraints specified earlier. Some of these rules are as follows:

- Deleting an entity from a superclass implies that it is automatically deleted from all the subclasses to which it belongs.
- Inserting an entity in a superclass implies that the entity is mandatorily inserted in all *predicate-defined* (or *attribute-defined*) subclasses for which the entity satisfies the defining predicate.
- Inserting an entity in a superclass of a *total specialization* implies that the entity is mandatorily inserted in at least one of the subclasses of the specialization.

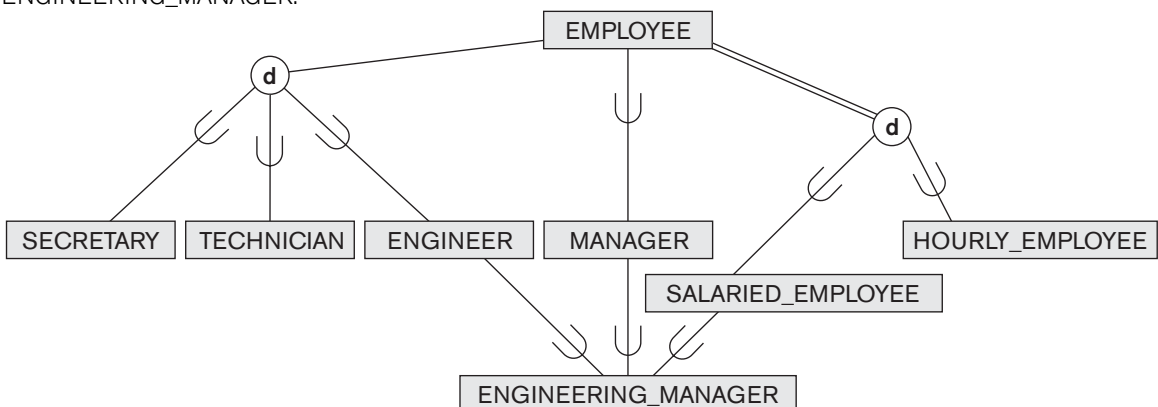
The reader is encouraged to make a complete list of rules for insertions and deletions for the various types of specializations.

8.3.2 Specialization and Generalization Hierarchies and Lattices

A subclass itself may have further subclasses specified on it, forming a hierarchy or a lattice of specializations. For example, in Figure 8.6 ENGINEER is a subclass of EMPLOYEE and is also a superclass of ENGINEERING_MANAGER; this represents the real-world constraint that every engineering manager is required to be an engineer. A **specialization hierarchy** has the constraint that every subclass participates *as a subclass* in *only one* class/subclass relationship; that is, each subclass has only

Figure 8.6

A specialization lattice with shared subclass ENGINEERING_MANAGER.



one parent, which results in a **tree structure** or **strict hierarchy**. In contrast, for a **specialization lattice**, a subclass can be a subclass in *more than one* class/subclass relationship. Hence, Figure 8.6 is a lattice.

Figure 8.7 shows another specialization lattice of more than one level. This may be part of a conceptual schema for a UNIVERSITY database. Notice that this arrangement would have been a hierarchy except for the STUDENT_ASSISTANT subclass, which is a subclass in two distinct class/subclass relationships.

The requirements for the part of the UNIVERSITY database shown in Figure 8.7 are the following:

1. The database keeps track of three types of persons: employees, alumni, and students. A person can belong to one, two, or all three of these types. Each person has a name, SSN, sex, address, and birth date.
2. Every employee has a salary, and there are three types of employees: faculty, staff, and student assistants. Each employee belongs to exactly one of these types. For each alumnus, a record of the degree or degrees that he or she

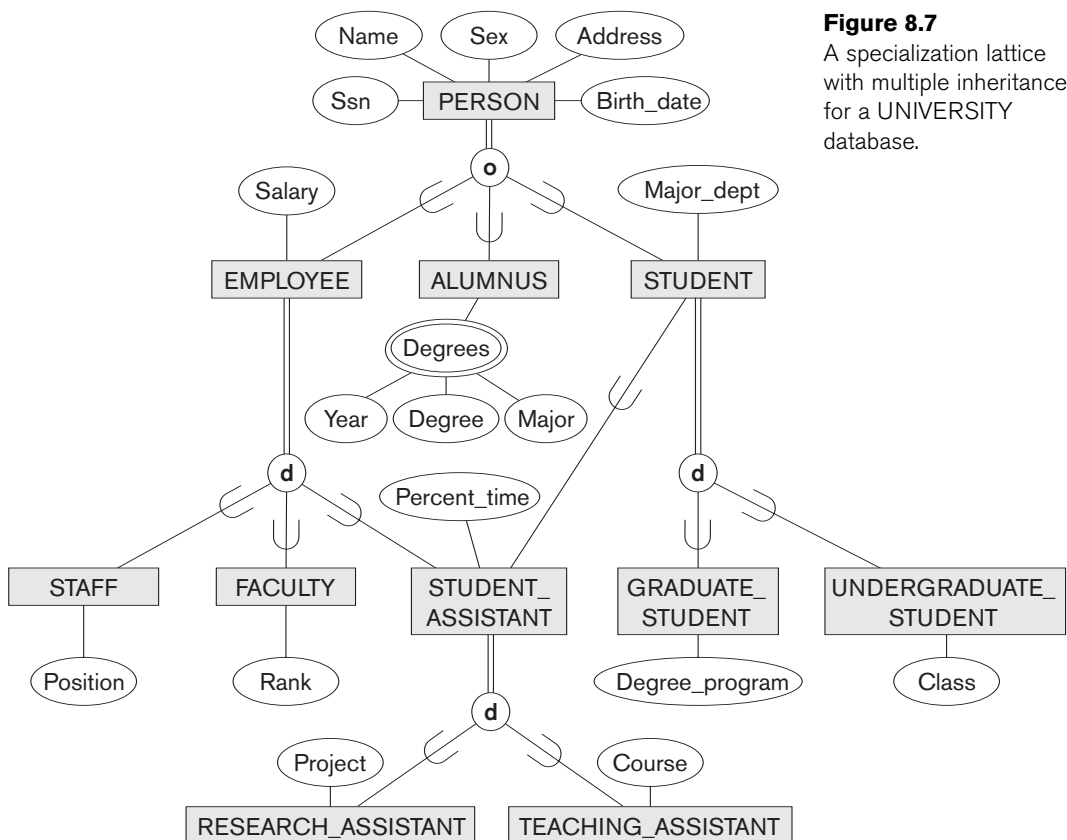


Figure 8.7

A specialization lattice with multiple inheritance for a UNIVERSITY database.

- earned at the university is kept, including the name of the degree, the year granted, and the major department. Each student has a major department.
3. Each faculty has a rank, whereas each staff member has a staff position. Student assistants are classified further as either research assistants or teaching assistants, and the percent of time that they work is recorded in the database. Research assistants have their research project stored, whereas teaching assistants have the current course they work on.
 4. Students are further classified as either graduate or undergraduate, with the specific attributes degree program (M.S., Ph.D., M.B.A., and so on) for graduate students and class (freshman, sophomore, and so on) for undergraduates.

In Figure 8.7, all person entities represented in the database are members of the PERSON entity type, which is specialized into the subclasses {EMPLOYEE, ALUMNUS, STUDENT}. This specialization is overlapping; for example, an alumnus may also be an employee and may also be a student pursuing an advanced degree. The subclass STUDENT is the superclass for the specialization {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}, while EMPLOYEE is the superclass for the specialization {STUDENT_ASSISTANT, FACULTY, STAFF}. Notice that STUDENT_ASSISTANT is also a subclass of STUDENT. Finally, STUDENT_ASSISTANT is the superclass for the specialization into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}.

In such a specialization lattice or hierarchy, a subclass inherits the attributes not only of its direct superclass, but also of all its predecessor superclasses *all the way to the root* of the hierarchy or lattice if necessary. For example, an entity in GRADUATE_STUDENT inherits all the attributes of that entity as a STUDENT *and* as a PERSON. Notice that an entity may exist in several *leaf nodes* of the hierarchy, where a **leaf node** is a class that has *no subclasses of its own*. For example, a member of GRADUATE_STUDENT may also be a member of RESEARCH_ASSISTANT.

A subclass with *more than one* superclass is called a **shared subclass**, such as ENGINEERING_MANAGER in Figure 8.6. This leads to the concept known as **multiple inheritance**, where the shared subclass ENGINEERING_MANAGER directly inherits attributes and relationships from multiple classes. Notice that the existence of at least one shared subclass leads to a lattice (and hence to *multiple inheritance*); if no shared subclasses existed, we would have a hierarchy rather than a lattice and only **single inheritance** would exist. An important rule related to multiple inheritance can be illustrated by the example of the shared subclass STUDENT_ASSISTANT in Figure 8.7, which inherits attributes from both EMPLOYEE and STUDENT. Here, both EMPLOYEE and STUDENT inherit *the same attributes* from PERSON. The rule states that if an attribute (or relationship) originating in the *same superclass* (PERSON) is inherited more than once via different paths (EMPLOYEE and STUDENT) in the lattice, then it should be included only once in the shared subclass (STUDENT_ASSISTANT). Hence, the attributes of PERSON are inherited *only once* in the STUDENT_ASSISTANT subclass in Figure 8.7.

It is important to note here that some models and languages are limited to **single inheritance** and *do not allow* multiple inheritance (shared subclasses). It is also important to note that some models do not allow an entity to have multiple types, and hence an entity can be a member of *only one leaf class*.⁸ In such a model, it is necessary to create additional subclasses as leaf nodes to cover all possible combinations of classes that may have some entity that belongs to all these classes simultaneously. For example, in the overlapping specialization of PERSON into {EMPLOYEE, ALUMNUS, STUDENT} (or {E, A, S} for short), it would be necessary to create seven subclasses of PERSON in order to cover all possible types of entities: E, A, S, E_A, E_S, A_S, and E_A_S. Obviously, this can lead to extra complexity.

Although we have used specialization to illustrate our discussion, similar concepts *apply equally* to generalization, as we mentioned at the beginning of this section. Hence, we can also speak of **generalization hierarchies** and **generalization lattices**.

8.3.3 Utilizing Specialization and Generalization in Refining Conceptual Schemas

Now we elaborate on the differences between the specialization and generalization processes, and how they are used to refine conceptual schemas during conceptual database design. In the specialization process, we typically start with an entity type and then define subclasses of the entity type by successive specialization; that is, we repeatedly define more specific groupings of the entity type. For example, when designing the specialization lattice in Figure 8.7, we may first specify an entity type PERSON for a university database. Then we discover that three types of persons will be represented in the database: university employees, alumni, and students. We create the specialization {EMPLOYEE, ALUMNUS, STUDENT} for this purpose and choose the overlapping constraint, because a person may belong to more than one of the subclasses. We specialize EMPLOYEE further into {STAFF, FACULTY, STUDENT_ASSISTANT}, and specialize STUDENT into {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}. Finally, we specialize STUDENT_ASSISTANT into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}. This successive specialization corresponds to a **top-down conceptual refinement process** during conceptual schema design. So far, we have a hierarchy; then we realize that STUDENT_ASSISTANT is a shared subclass, since it is also a subclass of STUDENT, leading to the lattice.

It is possible to arrive at the same hierarchy or lattice from the other direction. In such a case, the process involves generalization rather than specialization and corresponds to a **bottom-up conceptual synthesis**. For example, the database designers may first discover entity types such as STAFF, FACULTY, ALUMNUS, GRADUATE_STUDENT, UNDERGRADUATE_STUDENT, RESEARCH_ASSISTANT, TEACHING_ASSISTANT, and so on; then they generalize {GRADUATE_STUDENT,

⁸In some models, the class is further restricted to be a *leaf node* in the hierarchy or lattice.

UNDERGRADUATE_STUDENT} into STUDENT; then they generalize {RESEARCH_ASSISTANT, TEACHING_ASSISTANT} into STUDENT_ASSISTANT; then they generalize {STAFF, FACULTY, STUDENT_ASSISTANT} into EMPLOYEE; and finally they generalize {EMPLOYEE, ALUMNUS, STUDENT} into PERSON.

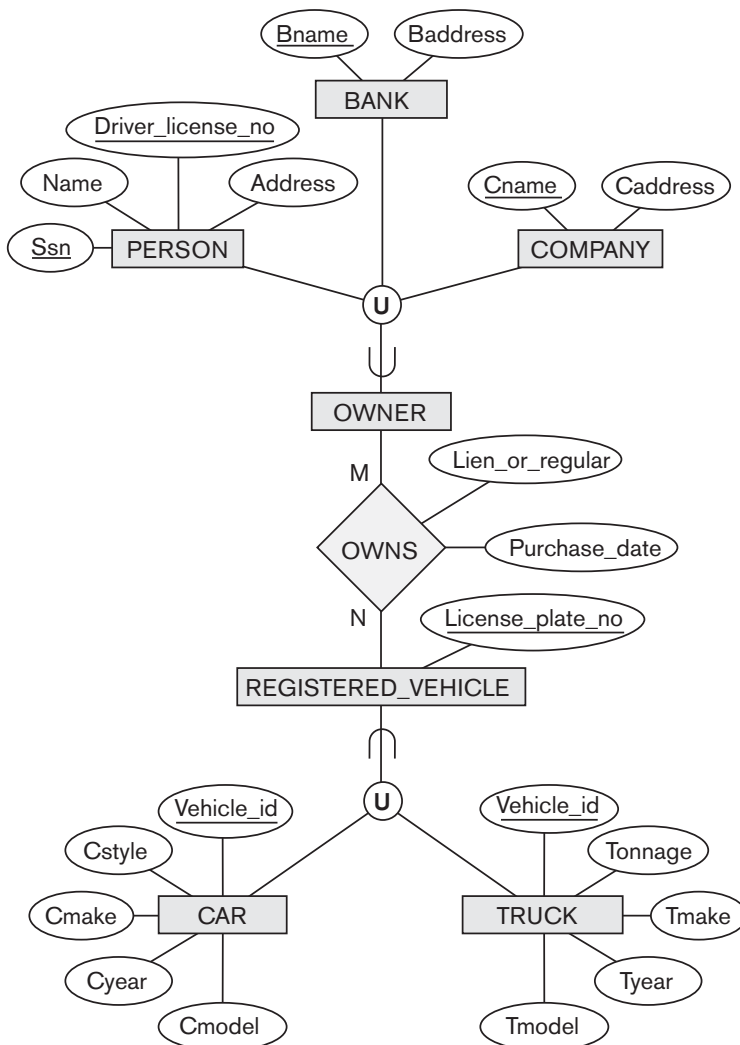
In structural terms, hierarchies or lattices resulting from either process may be identical; the only difference relates to the manner or order in which the schema super-classes and subclasses were created during the design process. In practice, it is likely that neither the generalization process nor the specialization process is followed strictly, but that a combination of the two processes is employed. New classes are continually incorporated into a hierarchy or lattice as they become apparent to users and designers. Notice that the notion of representing data and knowledge by using superclass/subclass hierarchies and lattices is quite common in knowledge-based systems and expert systems, which combine database technology with artificial intelligence techniques. For example, frame-based knowledge representation schemes closely resemble class hierarchies. Specialization is also common in software engineering design methodologies that are based on the object-oriented paradigm.

8.4 Modeling of UNION Types Using Categories

All of the superclass/subclass relationships we have seen thus far have a *single superclass*. A shared subclass such as ENGINEERING_MANAGER in the lattice in Figure 8.6 is the subclass in three *distinct* superclass/subclass relationships, where each of the three relationships has a *single* superclass. However, it is sometimes necessary to represent a single superclass/subclass relationship with *more than one* superclass, where the superclasses represent different entity types. In this case, the subclass will represent a collection of objects that is a subset of the UNION of distinct entity types; we call such a *subclass* a **union type** or a **category**.⁹

For example, suppose that we have three entity types: PERSON, BANK, and COMPANY. In a database for motor vehicle registration, an owner of a vehicle can be a person, a bank (holding a lien on a vehicle), or a company. We need to create a class (collection of entities) that includes entities of all three types to play the role of *vehicle owner*. A category (union type) OWNER that is a *subclass of the UNION* of the three entity sets of COMPANY, BANK, and PERSON can be created for this purpose. We display categories in an EER diagram as shown in Figure 8.8. The superclasses COMPANY, BANK, and PERSON are connected to the circle with the \cup symbol, which stands for the *set union operation*. An arc with the subset symbol connects the circle to the (subclass) OWNER category. If a defining predicate is needed, it is displayed next to the line from the superclass to which the predicate applies. In Figure 8.8 we have two categories: OWNER, which is a subclass of the union of PERSON, BANK, and COMPANY; and REGISTERED_VEHICLE, which is a subclass of the union of CAR and TRUCK.

⁹Our use of the term *category* is based on the ECR (Entity-Category-Relationship) model (Elmasri et al. 1985).

**Figure 8.8**

Two categories (union types): OWNER and REGISTERED_VEHICLE.

A category has two or more superclasses that may represent *distinct entity types*, whereas other superclass/subclass relationships always have a single superclass. To better understand the difference, we can compare a category, such as OWNER in Figure 8.8, with the ENGINEERING_MANAGER shared subclass in Figure 8.6. The latter is a subclass of *each of* the three superclasses ENGINEER, MANAGER, and SALARIED_EMPLOYEE, so an entity that is a member of ENGINEERING_MANAGER must exist in *all three*. This represents the constraint that an engineering manager must be an ENGINEER, a MANAGER, *and* a SALARIED_EMPLOYEE; that is, ENGINEERING_MANAGER is a subset of the *intersection* of the three classes (sets of entities). On the other hand, a category is a subset of the *union* of its superclasses. Hence, an entity that is a member of OWNER must exist in *only one* of the super-

classes. This represents the constraint that an OWNER may be a COMPANY, a BANK, or a PERSON in Figure 8.8.

Attribute inheritance works more selectively in the case of categories. For example, in Figure 8.8 each OWNER entity inherits the attributes of a COMPANY, a PERSON, or a BANK, depending on the superclass to which the entity belongs. On the other hand, a shared subclass such as ENGINEERING_MANAGER (Figure 8.6) inherits *all* the attributes of its superclasses SALARIED_EMPLOYEE, ENGINEER, and MANAGER.

It is interesting to note the difference between the category REGISTERED_VEHICLE (Figure 8.8) and the generalized superclass VEHICLE (Figure 8.3(b)). In Figure 8.3(b), every car and every truck is a VEHICLE; but in Figure 8.8, the REGISTERED_VEHICLE category includes some cars and some trucks but not necessarily all of them (for example, some cars or trucks may not be registered). In general, a specialization or generalization such as that in Figure 8.3(b), if it were *partial*, would not preclude VEHICLE from containing other types of entities, such as motorcycles. However, a category such as REGISTERED_VEHICLE in Figure 8.8 implies that only cars and trucks, but not other types of entities, can be members of REGISTERED_VEHICLE.

A category can be **total** or **partial**. A total category holds the *union* of all entities in its superclasses, whereas a partial category can hold a *subset of the union*. A total category is represented diagrammatically by a double line connecting the category and the circle, whereas a partial category is indicated by a single line.

The superclasses of a category may have different key attributes, as demonstrated by the OWNER category in Figure 8.8, or they may have the same key attribute, as demonstrated by the REGISTERED_VEHICLE category. Notice that if a category is total (not partial), it may be represented alternatively as a total specialization (or a total generalization). In this case, the choice of which representation to use is subjective. If the two classes represent the same type of entities and share numerous attributes, including the same key attributes, specialization/generalization is preferred; otherwise, categorization (union type) is more appropriate.

It is important to note that some modeling methodologies do not have union types. In these models, a union type must be represented in a roundabout way (see Section 9.2).

8.5 A Sample UNIVERSITY EER Schema, Design Choices, and Formal Definitions

In this section, we first give an example of a database schema in the EER model to illustrate the use of the various concepts discussed here and in Chapter 7. Then, we discuss design choices for conceptual schemas, and finally we summarize the EER model concepts and define them formally in the same manner in which we formally defined the concepts of the basic ER model in Chapter 7.

8.5.1 The UNIVERSITY Database Example

For our sample database application, consider a UNIVERSITY database that keeps track of students and their majors, transcripts, and registration as well as of the university's course offerings. The database also keeps track of the sponsored research projects of faculty and graduate students. This schema is shown in Figure 8.9. A discussion of the requirements that led to this schema follows.

For each person, the database maintains information on the person's Name [Name], Social Security number [Ssn], address [Address], sex [Sex], and birth date [Bdate]. Two subclasses of the PERSON entity type are identified: FACULTY and STUDENT. Specific attributes of FACULTY are rank [Rank] (assistant, associate, adjunct, research, visiting, and so on), office [Foffice], office phone [Fphone], and salary [Salary]. All faculty members are related to the academic department(s) with which they are affiliated [BELONGS] (a faculty member can be associated with several departments, so the relationship is M:N). A specific attribute of STUDENT is [Class] (freshman=1, sophomore=2, ..., graduate student=5). Each STUDENT is also related to his or her major and minor departments (if known) [MAJOR] and [MINOR], to the course sections he or she is currently attending [REGISTERED], and to the courses completed [TRANSCRIPT]. Each TRANSCRIPT instance includes the grade the student received [Grade] in a section of a course.

GRAD_STUDENT is a subclass of STUDENT, with the defining predicate Class = 5. For each graduate student, we keep a list of previous degrees in a composite, multi-valued attribute [Degrees]. We also relate the graduate student to a faculty advisor [ADVISOR] and to a thesis committee [COMMITTEE], if one exists.

An academic department has the attributes name [Dname], telephone [Dphone], and office number [Office] and is related to the faculty member who is its chairperson [CHAIRS] and to the college to which it belongs [CD]. Each college has attributes college name [Cname], office number [Coffice], and the name of its dean [Dean].

A course has attributes course number [C#], course name [Cname], and course description [Cdesc]. Several sections of each course are offered, with each section having the attributes section number [Sec#] and the year and quarter in which the section was offered ([Year] and [Qtr]).¹⁰ Section numbers uniquely identify each section. The sections being offered during the current quarter are in a subclass CURRENT_SECTION of SECTION, with the defining predicate Qtr = Current_qtr and Year = Current_year. Each section is related to the instructor who taught or is teaching it ([TEACH]), if that instructor is in the database.

The category INSTRUCTOR_RESEARCHER is a subset of the union of FACULTY and GRAD_STUDENT and includes all faculty, as well as graduate students who are supported by teaching or research. Finally, the entity type GRANT keeps track of research grants and contracts awarded to the university. Each grant has attributes grant title [Title], grant number [No], the awarding agency [Agency], and the starting

¹⁰We assume that the *quarter* system rather than the *semester* system is used in this university.

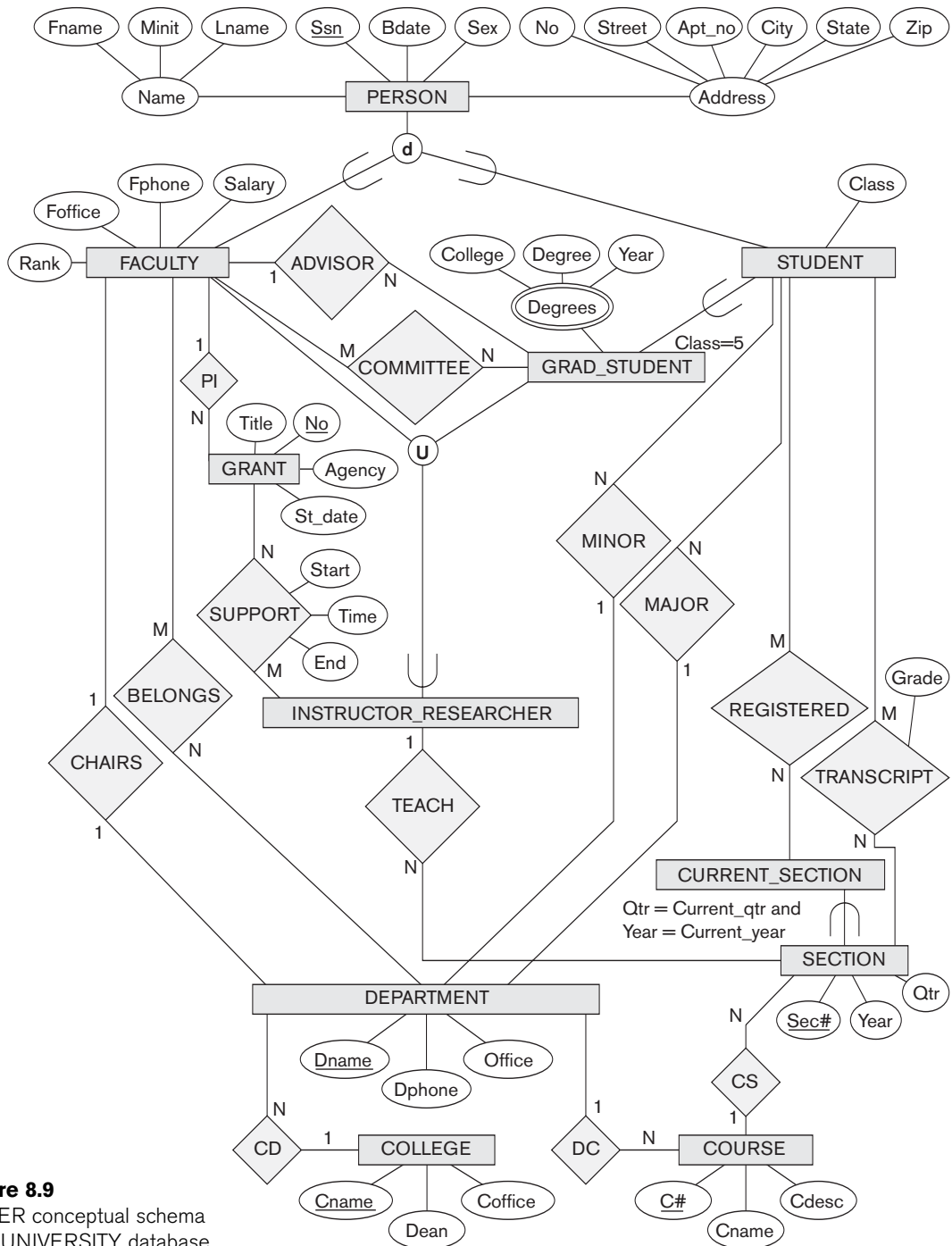


Figure 8.9
An EER conceptual schema
for a UNIVERSITY database.

date [St_date]. A grant is related to one principal investigator [PI] and to all researchers it supports [SUPPORT]. Each instance of support has as attributes the starting date of support [Start], the ending date of the support (if known) [End], and the percentage of time being spent on the project [Time] by the researcher being supported.

8.5.2 Design Choices for Specialization/Generalization

It is not always easy to choose the most appropriate conceptual design for a database application. In Section 7.7.3, we presented some of the typical issues that confront a database designer when choosing among the concepts of entity types, relationship types, and attributes to represent a particular miniworld situation as an ER schema. In this section, we discuss design guidelines and choices for the EER concepts of specialization/generalization and categories (union types).

As we mentioned in Section 7.7.3, conceptual database design should be considered as an iterative refinement process until the most suitable design is reached. The following guidelines can help to guide the design process for EER concepts:

- In general, many specializations and subclasses can be defined to make the conceptual model accurate. However, the drawback is that the design becomes quite cluttered. It is important to represent only those subclasses that are deemed necessary to avoid extreme cluttering of the conceptual schema.
- If a subclass has few specific (local) attributes and no specific relationships, it can be merged into the superclass. The specific attributes would hold NULL values for entities that are not members of the subclass. A *type* attribute could specify whether an entity is a member of the subclass.
- Similarly, if all the subclasses of a specialization/generalization have few specific attributes and no specific relationships, they can be merged into the superclass and replaced with one or more *type* attributes that specify the subclass or subclasses that each entity belongs to (see Section 9.2 for how this criterion applies to relational databases).
- Union types and categories should generally be avoided unless the situation definitely warrants this type of construct, which does occur in some practical situations. If possible, we try to model using specialization/generalization as discussed at the end of Section 8.4.
- The choice of disjoint/overlapping and total/partial constraints on specialization/generalization is driven by the rules in the miniworld being modeled. If the requirements do not indicate any particular constraints, the default would generally be overlapping and partial, since this does not specify any restrictions on subclass membership.

As an example of applying these guidelines, consider Figure 8.6, where no specific (local) attributes are shown. We could merge all the subclasses into the **EMPLOYEE** entity type, and add the following attributes to **EMPLOYEE**:

- An attribute **Job_type** whose value set {'Secretary', 'Engineer', 'Technician'} would indicate which subclass in the first specialization each employee belongs to.
- An attribute **Pay_method** whose value set {'Salaried', 'Hourly'} would indicate which subclass in the second specialization each employee belongs to.
- An attribute **Is_a_manager** whose value set {'Yes', 'No'} would indicate whether an individual employee entity is a manager or not.

8.5.3 Formal Definitions for the EER Model Concepts

We now summarize the EER model concepts and give formal definitions. A **class**¹¹ is a set or collection of entities; this includes any of the EER schema constructs of group entities, such as entity types, subclasses, superclasses, and categories. A **subclass** S is a class whose entities must always be a subset of the entities in another class, called the **superclass** C of the **superclass/subclass** (or **IS-A**) **relationship**. We denote such a relationship by C/S . For such a superclass/subclass relationship, we must always have

$$S \subseteq C$$

A **specialization** $Z = \{S_1, S_2, \dots, S_n\}$ is a set of subclasses that have the same superclass G ; that is, G/S_i is a superclass/subclass relationship for $i = 1, 2, \dots, n$. G is called a **generalized entity type** (or the **superclass** of the specialization, or a **generalization** of the subclasses $\{S_1, S_2, \dots, S_n\}$). Z is said to be **total** if we always (at any point in time) have

$$\bigcup_{i=1}^n S_i = G$$

Otherwise, Z is said to be **partial**. Z is said to be **disjoint** if we always have

$$S_i \cap S_j = \emptyset \text{ (empty set) for } i \neq j$$

Otherwise, Z is said to be **overlapping**.

A subclass S of C is said to be **predicate-defined** if a predicate p on the attributes of C is used to specify which entities in C are members of S ; that is, $S = C[p]$, where $C[p]$ is the set of entities in C that satisfy p . A subclass that is not defined by a predicate is called **user-defined**.

A specialization Z (or generalization G) is said to be **attribute-defined** if a predicate ($A = c_i$), where A is an attribute of G and c_i is a constant value from the domain of A ,

¹¹The use of the word *class* here differs from its more common use in object-oriented programming languages such as C++. In C++, a class is a structured type definition along with its applicable functions (operations).

is used to specify membership in each subclass S_i in Z . Notice that if $c_i \neq c_j$ for $i \neq j$, and A is a single-valued attribute, then the specialization will be disjoint.

A **category** T is a class that is a subset of the union of n defining superclasses D_1, D_2, \dots, D_n , $n > 1$, and is formally specified as follows:

$$T \subseteq (D_1 \cup D_2 \dots \cup D_n)$$

A predicate p_i on the attributes of D_i can be used to specify the members of each D_i that are members of T . If a predicate is specified on every D_i , we get

$$T = (D_1[p_1] \cup D_2[p_2] \dots \cup D_n[p_n])$$

We should now extend the definition of **relationship type** given in Chapter 7 by allowing any class—not only any entity type—to participate in a relationship. Hence, we should replace the words *entity type* with *class* in that definition. The graphical notation of EER is consistent with ER because all classes are represented by rectangles.

8.6 Example of Other Notation: Representing Specialization and Generalization in UML Class Diagrams

We now discuss the UML notation for generalization/specialization and inheritance. We already presented basic UML class diagram notation and terminology in Section 7.8. Figure 8.10 illustrates a possible UML class diagram corresponding to the EER diagram in Figure 8.7. The basic notation for specialization/generalization (see Figure 8.10) is to connect the subclasses by vertical lines to a horizontal line, which has a triangle connecting the horizontal line through another vertical line to the superclass. A blank triangle indicates a specialization/generalization with the *disjoint* constraint, and a filled triangle indicates an *overlapping* constraint. The root superclass is called the **base class**, and the subclasses (leaf nodes) are called **leaf classes**.

The above discussion and example in Figure 8.10, and the presentation in Section 7.8 gave a brief overview of UML class diagrams and terminology. We focused on the concepts that are relevant to ER and EER database modeling, rather than those concepts that are more relevant to software engineering. In UML, there are many details that we have not discussed because they are outside the scope of this book and are mainly relevant to software engineering. For example, classes can be of various types:

- Abstract classes define attributes and operations but do not have objects corresponding to those classes. These are mainly used to specify a set of attributes and operations that can be inherited.
- Concrete classes can have objects (entities) instantiated to belong to the class.
- Template classes specify a template that can be further used to define other classes.

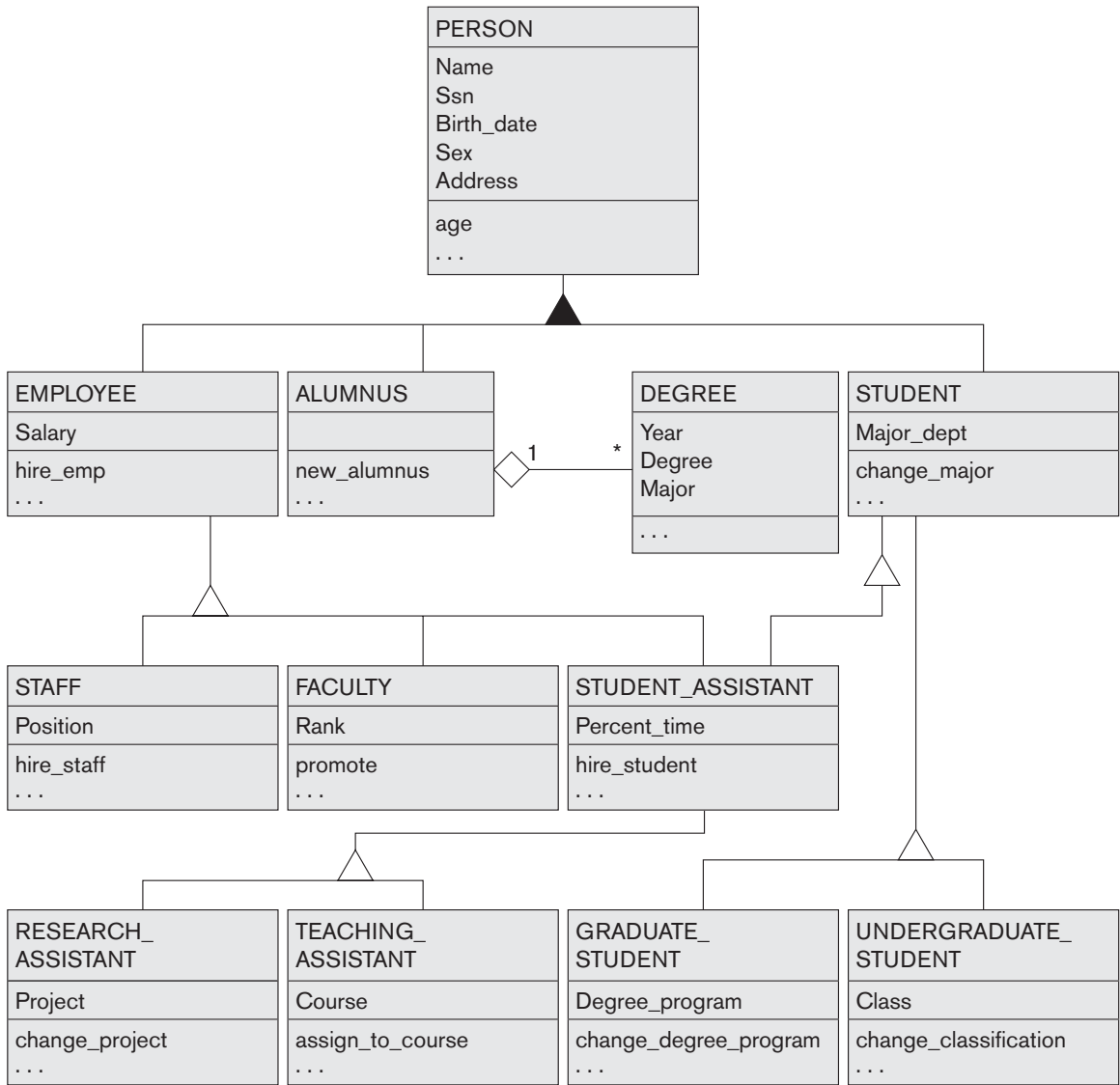


Figure 8.10
A UML class diagram corresponding to the EER diagram in Figure 8.7, illustrating UML notation for specialization/generalization.

In database design, we are mainly concerned with specifying concrete classes whose collections of objects are permanently (or persistently) stored in the database. The bibliographic notes at the end of this chapter give some references to books that describe complete details of UML. Additional material related to UML is covered in Chapter 10.

8.7 Data Abstraction, Knowledge Representation, and Ontology Concepts

In this section we discuss in general terms some of the modeling concepts that we described quite specifically in our presentation of the ER and EER models in Chapter 7 and earlier in this chapter. This terminology is not only used in conceptual data modeling but also in artificial intelligence literature when discussing **knowledge representation (KR)**. This section discusses the similarities and differences between conceptual modeling and knowledge representation, and introduces some of the alternative terminology and a few additional concepts.

The goal of KR techniques is to develop concepts for accurately modeling some **domain of knowledge** by creating an **ontology**¹² that describes the concepts of the domain and how these concepts are interrelated. Such an ontology is used to store and manipulate knowledge for drawing inferences, making decisions, or answering questions. The goals of KR are similar to those of semantic data models, but there are some important similarities and differences between the two disciplines:

- Both disciplines use an abstraction process to identify common properties and important aspects of objects in the miniworld (also known as *domain of discourse* in KR) while suppressing insignificant differences and unimportant details.
- Both disciplines provide concepts, relationships, constraints, operations, and languages for defining data and representing knowledge.
- KR is generally broader in scope than semantic data models. Different forms of knowledge, such as rules (used in inference, deduction, and search), incomplete and default knowledge, and temporal and spatial knowledge, are represented in KR schemes. Database models are being expanded to include some of these concepts (see Chapter 26).
- KR schemes include **reasoning mechanisms** that deduce additional facts from the facts stored in a database. Hence, whereas most current database systems are limited to answering direct queries, knowledge-based systems using KR schemes can answer queries that involve **inferences** over the stored data. Database technology is being extended with inference mechanisms (see Section 26.5).
- Whereas most data models concentrate on the representation of database schemas, or meta-knowledge, KR schemes often mix up the schemas with the instances themselves in order to provide flexibility in representing exceptions. This often results in inefficiencies when these KR schemes are implemented, especially when compared with databases and when a large amount of data (facts) needs to be stored.

¹²An *ontology* is somewhat similar to a conceptual schema, but with more knowledge, rules, and exceptions.

We now discuss four **abstraction concepts** that are used in semantic data models, such as the EER model as well as in KR schemes: (1) classification and instantiation, (2) identification, (3) specialization and generalization, and (4) aggregation and association. The paired concepts of classification and instantiation are inverses of one another, as are generalization and specialization. The concepts of aggregation and association are also related. We discuss these abstract concepts and their relation to the concrete representations used in the EER model to clarify the data abstraction process and to improve our understanding of the related process of conceptual schema design. We close the section with a brief discussion of *ontology*, which is being used widely in recent knowledge representation research.

8.7.1 Classification and Instantiation

The process of **classification** involves systematically assigning similar objects/entities to object classes/entity types. We can now describe (in DB) or reason about (in KR) the classes rather than the individual objects. Collections of objects that share the same types of attributes, relationships, and constraints are classified into classes in order to simplify the process of discovering their properties. **Instantiation** is the inverse of classification and refers to the generation and specific examination of distinct objects of a class. An object instance is related to its object class by the **IS-AN-INSTANCE-OF** or **IS-A-MEMBER-OF** relationship. Although EER diagrams do not display instances, the UML diagrams allow a form of instantiation by permitting the display of individual objects. We *did not* describe this feature in our introduction to UML class diagrams.

In general, the objects of a class should have a similar type structure. However, some objects may display properties that differ in some respects from the other objects of the class; these **exception objects** also need to be modeled, and KR schemes allow more varied exceptions than do database models. In addition, certain properties apply to the class as a whole and not to the individual objects; KR schemes allow such **class properties**. UML diagrams also allow specification of class properties.

In the EER model, entities are classified into entity types according to their basic attributes and relationships. Entities are further classified into subclasses and categories based on additional similarities and differences (exceptions) among them. Relationship instances are classified into relationship types. Hence, entity types, subclasses, categories, and relationship types are the different concepts that are used for classification in the EER model. The EER model does not provide explicitly for class properties, but it may be extended to do so. In UML, objects are classified into classes, and it is possible to display both class properties and individual objects.

Knowledge representation models allow multiple classification schemes in which one class is an *instance* of another class (called a **meta-class**). Notice that this *cannot* be represented directly in the EER model, because we have only two levels—classes and instances. The only relationship among classes in the EER model is a super-class/subclass relationship, whereas in some KR schemes an additional class/instance relationship can be represented directly in a class hierarchy. An instance may itself be another class, allowing multiple-level classification schemes.

8.7.2 Identification

Identification is the abstraction process whereby classes and objects are made uniquely identifiable by means of some **identifier**. For example, a class name uniquely identifies a whole class within a schema. An additional mechanism is necessary for telling distinct object instances apart by means of object identifiers. Moreover, it is necessary to identify multiple manifestations in the database of the same real-world object. For example, we may have a tuple <‘Matthew Clarke’, ‘610618’, ‘376-9821’> in a PERSON relation and another tuple <‘301-54-0836’, ‘CS’, 3.8> in a STUDENT relation that happen to represent the same real-world entity. There is no way to identify the fact that these two database objects (tuples) represent the same real-world entity unless we make a provision *at design time* for appropriate cross-referencing to supply this identification. Hence, identification is needed at two levels:

- To distinguish among database objects and classes
- To identify database objects and to relate them to their real-world counterparts

In the EER model, identification of schema constructs is based on a system of unique names for the constructs in a schema. For example, every class in an EER schema—whether it is an entity type, a subclass, a category, or a relationship type—must have a distinct name. The names of attributes of a particular class must also be distinct. Rules for unambiguously identifying attribute name references in a specialization or generalization lattice or hierarchy are needed as well.

At the object level, the values of key attributes are used to distinguish among entities of a particular entity type. For weak entity types, entities are identified by a combination of their own partial key values and the entities they are related to in the owner entity type(s). Relationship instances are identified by some combination of the entities that they relate to, depending on the cardinality ratio specified.

8.7.3 Specialization and Generalization

Specialization is the process of classifying a class of objects into more specialized subclasses. **Generalization** is the inverse process of generalizing several classes into a higher-level abstract class that includes the objects in all these classes. Specialization is conceptual refinement, whereas generalization is conceptual synthesis. Subclasses are used in the EER model to represent specialization and generalization. We call the relationship between a subclass and its superclass an **IS-A-SUBCLASS-OF** relationship, or simply an **IS-A** relationship. This is the same as the IS-A relationship discussed earlier in Section 8.5.3.

8.7.4 Aggregation and Association

Aggregation is an abstraction concept for building composite objects from their component objects. There are three cases where this concept can be related to the EER model. The first case is the situation in which we aggregate attribute values of

an object to form the whole object. The second case is when we represent an aggregation relationship as an ordinary relationship. The third case, which the EER model does not provide for explicitly, involves the possibility of combining objects that are related by a particular relationship instance into a *higher-level aggregate object*. This is sometimes useful when the higher-level aggregate object is itself to be related to another object. We call the relationship between the primitive objects and their aggregate object **IS-A-PART-OF**; the inverse is called **IS-A-COMPONENT-OF**. UML provides for all three types of aggregation.

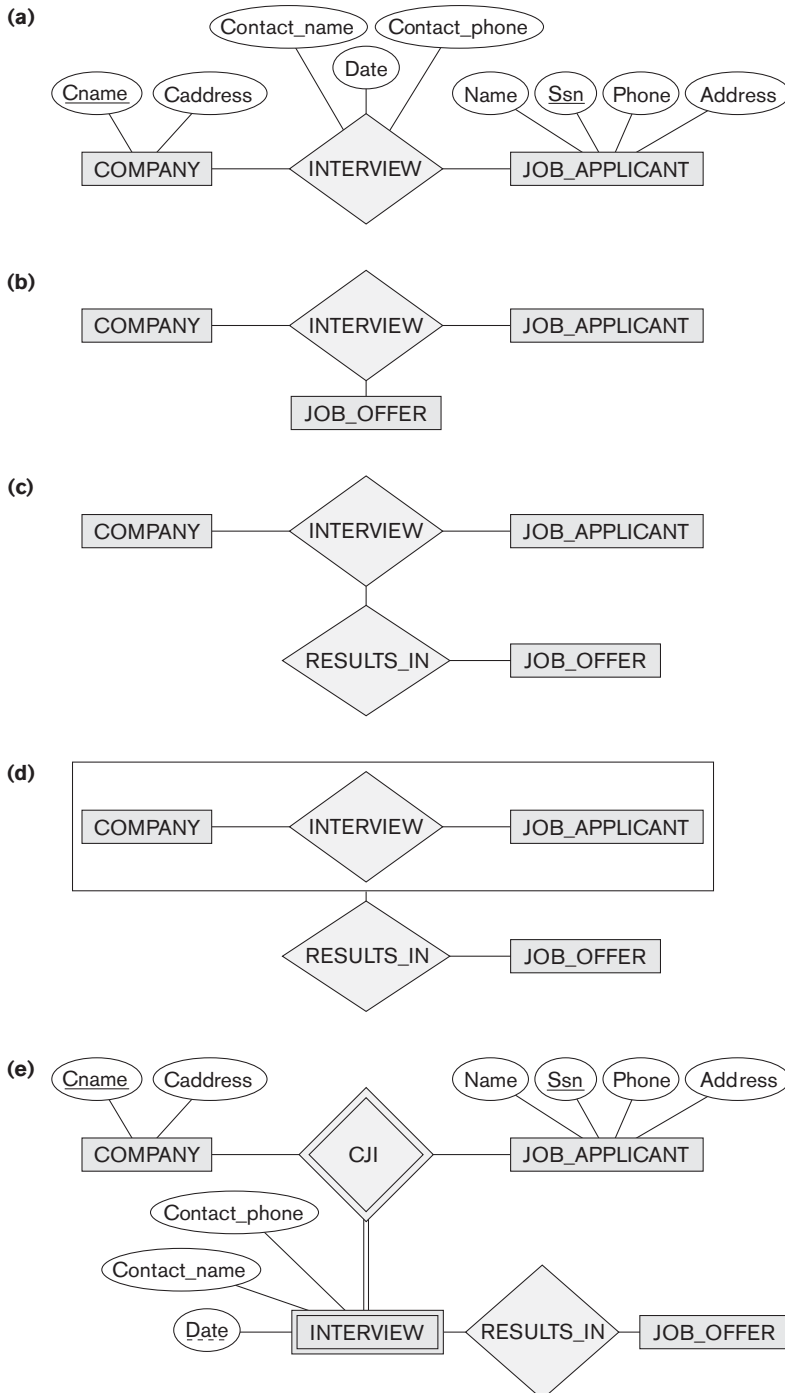
The abstraction of **association** is used to associate objects from several *independent classes*. Hence, it is somewhat similar to the second use of aggregation. It is represented in the EER model by relationship types, and in UML by associations. This abstract relationship is called **IS-ASSOCIATED-WITH**.

In order to understand the different uses of aggregation better, consider the ER schema shown in Figure 8.11(a), which stores information about interviews by job applicants to various companies. The class COMPANY is an aggregation of the attributes (or component objects) Cname (company name) and Caddress (company address), whereas JOB_APPLICANT is an aggregate of Ssn, Name, Address, and Phone. The relationship attributes Contact_name and Contact_phone represent the name and phone number of the person in the company who is responsible for the interview. Suppose that some interviews result in job offers, whereas others do not. We would like to treat INTERVIEW as a class to associate it with JOB_OFFER. The schema shown in Figure 8.11(b) is *incorrect* because it requires each interview relationship instance to have a job offer. The schema shown in Figure 8.11(c) is *not allowed* because the ER model does not allow relationships among relationships.

One way to represent this situation is to create a higher-level aggregate class composed of COMPANY, JOB_APPLICANT, and INTERVIEW and to relate this class to JOB_OFFER, as shown in Figure 8.11(d). Although the EER model as described in this book does not have this facility, some semantic data models do allow it and call the resulting object a **composite** or **molecular object**. Other models treat entity types and relationship types uniformly and hence permit relationships among relationships, as illustrated in Figure 8.11(c).

To represent this situation correctly in the ER model as described here, we need to create a new weak entity type INTERVIEW, as shown in Figure 8.11(e), and relate it to JOB_OFFER. Hence, we can always represent these situations correctly in the ER model by creating additional entity types, although it may be conceptually more desirable to allow direct representation of aggregation, as in Figure 8.11(d), or to allow relationships among relationships, as in Figure 8.11(c).

The main structural distinction between aggregation and association is that when an association instance is deleted, the participating objects may continue to exist. However, if we support the notion of an aggregate object—for example, a CAR that is made up of objects ENGINE, CHASSIS, and TIRES—then deleting the aggregate CAR object amounts to deleting all its component objects.

**Figure 8.11**

Aggregation. (a) The relationship type INTERVIEW. (b) Including JOB_OFFER in a ternary relationship type (incorrect). (c) Having the RESULTS_IN relationship participate in other relationships (not allowed in ER). (d) Using aggregation and a composite (molecular) object (generally not allowed in ER but allowed by some modeling tools). (e) Correct representation in ER.

8.7.5 Ontologies and the Semantic Web

In recent years, the amount of computerized data and information available on the Web has spiraled out of control. Many different models and formats are used. In addition to the database models that we present in this book, much information is stored in the form of **documents**, which have considerably less structure than database information does. One ongoing project that is attempting to allow information exchange among computers on the Web is called the **Semantic Web**, which attempts to create knowledge representation models that are quite general in order to allow meaningful information exchange and search among machines. The concept of *ontology* is considered to be the most promising basis for achieving the goals of the Semantic Web and is closely related to knowledge representation. In this section, we give a brief introduction to what ontology is and how it can be used as a basis to automate information understanding, search, and exchange.

The study of ontologies attempts to describe the structures and relationships that are possible in reality through some common vocabulary; therefore, it can be considered as a way to describe the knowledge of a certain community about reality. Ontology originated in the fields of philosophy and metaphysics. One commonly used definition of **ontology** is *a specification of a conceptualization*.¹³

In this definition, a **conceptualization** is the set of concepts that are used to represent the part of reality or knowledge that is of interest to a community of users. **Specification** refers to the language and vocabulary terms that are used to specify the conceptualization. The ontology includes both *specification* and *conceptualization*. For example, the same conceptualization may be specified in two different languages, giving two separate ontologies. Based on this quite general definition, there is no consensus on what an ontology is exactly. Some possible ways to describe ontologies are as follows:

- A **thesaurus** (or even a **dictionary** or a **glossary** of terms) describes the relationships between words (vocabulary) that represent various concepts.
- A **taxonomy** describes how concepts of a particular area of knowledge are related using structures similar to those used in a specialization or generalization.
- A detailed **database schema** is considered by some to be an ontology that describes the concepts (entities and attributes) and relationships of a mini-world from reality.
- A **logical theory** uses concepts from mathematical logic to try to define concepts and their interrelationships.

Usually the concepts used to describe ontologies are quite similar to the concepts we discussed in conceptual modeling, such as entities, attributes, relationships, specializations, and so on. The main difference between an ontology and, say, a database schema, is that the schema is usually limited to describing a small subset of a mini-

¹³This definition is given in Gruber (1995).

world from reality in order to store and manage data. An ontology is usually considered to be more general in that it attempts to describe a part of reality or a domain of interest (for example, medical terms, electronic-commerce applications, sports, and so on) as completely as possible.

8.8 Summary

In this chapter we discussed extensions to the ER model that improve its representational capabilities. We called the resulting model the enhanced ER or EER model. We presented the concept of a subclass and its superclass and the related mechanism of attribute/relationship inheritance. We saw how it is sometimes necessary to create additional classes of entities, either because of additional specific attributes or because of specific relationship types. We discussed two main processes for defining superclass/subclass hierarchies and lattices: specialization and generalization.

Next, we showed how to display these new constructs in an EER diagram. We also discussed the various types of constraints that may apply to specialization or generalization. The two main constraints are total/partial and disjoint/overlapping. In addition, a defining predicate for a subclass or a defining attribute for a specialization may be specified. We discussed the differences between user-defined and predicate-defined subclasses and between user-defined and attribute-defined specializations. Finally, we discussed the concept of a category or union type, which is a subset of the union of two or more classes, and we gave formal definitions of all the concepts presented.

We introduced some of the notation and terminology of UML for representing specialization and generalization. In Section 8.7 we briefly discussed the discipline of knowledge representation and how it is related to semantic data modeling. We also gave an overview and summary of the types of abstract data representation concepts: classification and instantiation, identification, specialization and generalization, and aggregation and association. We saw how EER and UML concepts are related to each of these.

Review Questions

- 8.1. What is a subclass? When is a subclass needed in data modeling?
- 8.2. Define the following terms: *superclass of a subclass*, *superclass/subclass relationship*, *IS-A relationship*, *specialization*, *generalization*, *category*, *specific (local) attributes*, and *specific relationships*.
- 8.3. Discuss the mechanism of attribute/relationship inheritance. Why is it useful?
- 8.4. Discuss user-defined and predicate-defined subclasses, and identify the differences between the two.
- 8.5. Discuss user-defined and attribute-defined specializations, and identify the differences between the two.

- 8.6. Discuss the two main types of constraints on specializations and generalizations.
- 8.7. What is the difference between a specialization hierarchy and a specialization lattice?
- 8.8. What is the difference between specialization and generalization? Why do we not display this difference in schema diagrams?
- 8.9. How does a category differ from a regular shared subclass? What is a category used for? Illustrate your answer with examples.
- 8.10. For each of the following UML terms (see Sections 7.8 and 8.6), discuss the corresponding term in the EER model, if any: *object*, *class*, *association*, *aggregation*, *generalization*, *multiplicity*, *attributes*, *discriminator*, *link*, *link attribute*, *reflexive association*, and *qualified association*.
- 8.11. Discuss the main differences between the notation for EER schema diagrams and UML class diagrams by comparing how common concepts are represented in each.
- 8.12. List the various data abstraction concepts and the corresponding modeling concepts in the EER model.
- 8.13. What aggregation feature is missing from the EER model? How can the EER model be further enhanced to support it?
- 8.14. What are the main similarities and differences between conceptual database modeling techniques and knowledge representation techniques?
- 8.15. Discuss the similarities and differences between an ontology and a database schema.

Exercises

- 8.16. Design an EER schema for a database application that you are interested in. Specify all constraints that should hold on the database. Make sure that the schema has at least five entity types, four relationship types, a weak entity type, a superclass/subclass relationship, a category, and an n -ary ($n > 2$) relationship type.
- 8.17. Consider the BANK ER schema in Figure 7.21, and suppose that it is necessary to keep track of different types of ACCOUNTS (SAVINGS_ACCTS, CHECKING_ACCTS, ...) and LOANS (CAR_LOANS, HOME_LOANS, ...). Suppose that it is also desirable to keep track of each ACCOUNT's TRANSACTIONS (deposits, withdrawals, checks, ...) and each LOAN's PAYMENTS; both of these include the amount, date, and time. Modify the BANK schema, using ER and EER concepts of specialization and generalization. State any assumptions you make about the additional requirements.

- 8.18.** The following narrative describes a simplified version of the organization of Olympic facilities planned for the summer Olympics. Draw an EER diagram that shows the entity types, attributes, relationships, and specializations for this application. State any assumptions you make. The Olympic facilities are divided into sports complexes. Sports complexes are divided into *one-sport* and *multisport* types. Multisport complexes have areas of the complex designated for each sport with a location indicator (e.g., center, NE corner, and so on). A complex has a location, chief organizing individual, total occupied area, and so on. Each complex holds a series of events (e.g., the track stadium may hold many different races). For each event there is a planned date, duration, number of participants, number of officials, and so on. A roster of all officials will be maintained together with the list of events each official will be involved in. Different equipment is needed for the events (e.g., goal posts, poles, parallel bars) as well as for maintenance. The two types of facilities (one-sport and multisport) will have different types of information. For each type, the number of facilities needed is kept, together with an approximate budget.
- 8.19.** Identify all the important concepts represented in the library database case study described below. In particular, identify the abstractions of classification (entity types and relationship types), aggregation, identification, and specialization/generalization. Specify (min, max) cardinality constraints whenever possible. List details that will affect the eventual design but that have no bearing on the conceptual design. List the semantic constraints separately. Draw an EER diagram of the library database.

Case Study: The Georgia Tech Library (GTL) has approximately 16,000 members, 100,000 titles, and 250,000 volumes (an average of 2.5 copies per book). About 10 percent of the volumes are out on loan at any one time. The librarians ensure that the books that members want to borrow are available when the members want to borrow them. Also, the librarians must know how many copies of each book are in the library or out on loan at any given time. A catalog of books is available online that lists books by author, title, and subject area. For each title in the library, a book description is kept in the catalog that ranges from one sentence to several pages. The reference librarians want to be able to access this description when members request information about a book. Library staff includes chief librarian, departmental associate librarians, reference librarians, check-out staff, and library assistants.

Books can be checked out for 21 days. Members are allowed to have only five books out at a time. Members usually return books within three to four weeks. Most members know that they have one week of grace before a notice is sent to them, so they try to return books before the grace period ends. About 5 percent of the members have to be sent reminders to return books. Most overdue books are returned within a month of the due date. Approximately 5 percent of the overdue books are either kept or never returned. The most active members of the library are defined as those who

borrow books at least ten times during the year. The top 1 percent of membership does 15 percent of the borrowing, and the top 10 percent of the membership does 40 percent of the borrowing. About 20 percent of the members are totally inactive in that they are members who never borrow.

To become a member of the library, applicants fill out a form including their SSN, campus and home mailing addresses, and phone numbers. The librarians issue a numbered, machine-readable card with the member's photo on it. This card is good for four years. A month before a card expires, a notice is sent to a member for renewal. Professors at the institute are considered automatic members. When a new faculty member joins the institute, his or her information is pulled from the employee records and a library card is mailed to his or her campus address. Professors are allowed to check out books for three-month intervals and have a two-week grace period. Renewal notices to professors are sent to their campus address.

The library does not lend some books, such as reference books, rare books, and maps. The librarians must differentiate between books that can be lent and those that cannot be lent. In addition, the librarians have a list of some books they are interested in acquiring but cannot obtain, such as rare or out-of-print books and books that were lost or destroyed but have not been replaced. The librarians must have a system that keeps track of books that cannot be lent as well as books that they are interested in acquiring. Some books may have the same title; therefore, the title cannot be used as a means of identification. Every book is identified by its International Standard Book Number (ISBN), a unique international code assigned to all books. Two books with the same title can have different ISBNs if they are in different languages or have different bindings (hardcover or softcover). Editions of the same book have different ISBNs.

The proposed database system must be designed to keep track of the members, the books, the catalog, and the borrowing activity.

8.20. Design a database to keep track of information for an art museum. Assume that the following requirements were collected:

- The museum has a collection of **ART_OBJECTS**. Each **ART_OBJECT** has a unique **Id_no**, an **Artist** (if known), a **Year** (when it was created, if known), a **Title**, and a **Description**. The art objects are categorized in several ways, as discussed below.
- **ART_OBJECTS** are categorized based on their type. There are three main types: **PAINTING**, **SCULPTURE**, and **STATUE**, plus another type called **OTHER** to accommodate objects that do not fall into one of the three main types.
- A **PAINTING** has a **Paint_type** (oil, watercolor, etc.), material on which it is **Drawn_on** (paper, canvas, wood, etc.), and **Style** (modern, abstract, etc.).
- A **SCULPTURE** or a statue has a **Material** from which it was created (wood, stone, etc.), **Height**, **Weight**, and **Style**.

- An art object in the OTHER category has a Type (print, photo, etc.) and Style.
- ART_OBJECTs are categorized as either PERMANENT_COLLECTION (objects that are owned by the museum) and BORROWED. Information captured about objects in the PERMANENT_COLLECTION includes Date_acquired, Status (on display, on loan, or stored), and Cost. Information captured about BORROWED objects includes the Collection from which it was borrowed, Date_borrowed, and Date_returned.
- Information describing the country or culture of Origin (Italian, Egyptian, American, Indian, and so forth) and Epoch (Renaissance, Modern, Ancient, and so forth) is captured for each ART_OBJECT.
- The museum keeps track of ARTIST information, if known: Name, DateBorn (if known), Date_died (if not living), Country_of_origin, Epoch, Main_style, and Description. The Name is assumed to be unique.
- Different EXHIBITIONS occur, each having a Name, Start_date, and End_date. EXHIBITIONS are related to all the art objects that were on display during the exhibition.
- Information is kept on other COLLECTIONS with which the museum interacts, including Name (unique), Type (museum, personal, etc.), Description, Address, Phone, and current Contact_person.

Draw an EER schema diagram for this application. Discuss any assumptions you make, and that justify your EER design choices.

- 8.21.** Figure 8.12 shows an example of an EER diagram for a small private airport database that is used to keep track of airplanes, their owners, airport employees, and pilots. From the requirements for this database, the following information was collected: Each AIRPLANE has a registration number [Reg#], is of a particular plane type [OF_TYPE], and is stored in a particular hangar [STORED_IN]. Each PLANE_TYPE has a model number [Model], a capacity [Capacity], and a weight [Weight]. Each HANGAR has a number [Number], a capacity [Capacity], and a location [Location]. The database also keeps track of the OWNERS of each plane [OWNS] and the EMPLOYEES who have maintained the plane [MAINTAIN]. Each relationship instance in OWNS relates an AIRPLANE to an OWNER and includes the purchase date [Pdate]. Each relationship instance in MAINTAIN relates an EMPLOYEE to a service record [SERVICE]. Each plane undergoes service many times; hence, it is related by [PLANE_SERVICE] to a number of SERVICE records. A SERVICE record includes as attributes the date of maintenance [Date], the number of hours spent on the work [Hours], and the type of work done [Work_code]. We use a weak entity type [SERVICE] to represent airplane service, because the airplane registration number is used to identify a service record. An OWNER is either a person or a corporation. Hence, we use a union type (category) [OWNER] that is a subset of the union of corporation [CORPORATION] and person [PERSON] entity types. Both pilots [PILOT] and employees [EMPLOYEE] are subclasses of PERSON. Each PILOT has

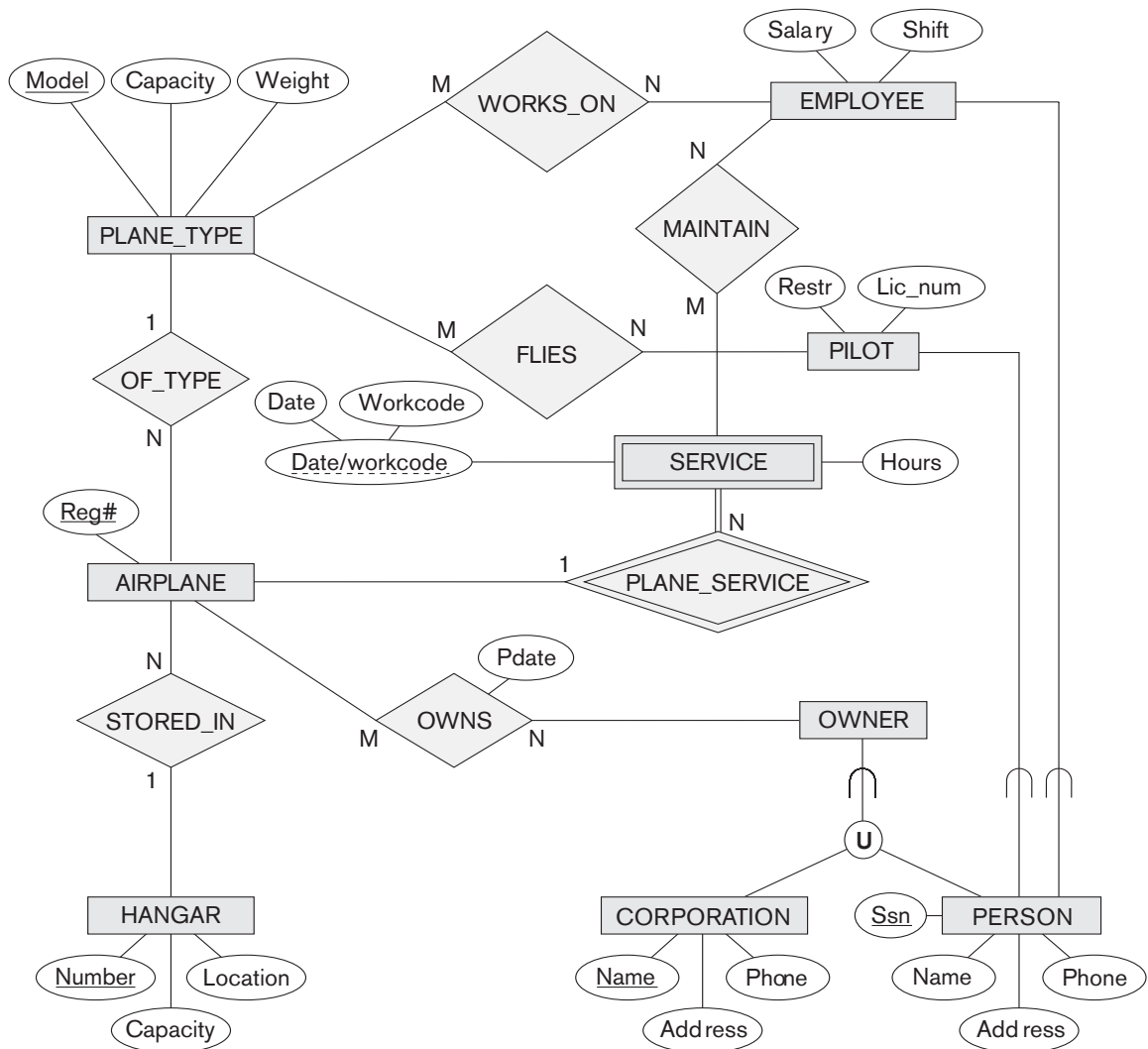


Figure 8.12
EER schema for a SMALL_AIRPORT database.

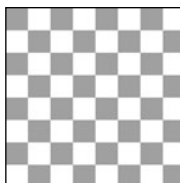
specific attributes license number [Lic_num] and restrictions [Restr]; each EMPLOYEE has specific attributes salary [Salary] and shift worked [Shift]. All PERSON entities in the database have data kept on their Social Security number [Ssn], name [Name], address [Address], and telephone number [Phone]. For CORPORATION entities, the data kept includes name [Name], address [Address], and telephone number [Phone]. The database also keeps track of the types of planes each pilot is authorized to fly [FLIES] and the types of planes each employee can do maintenance work on [WORKS_ON].

Show how the SMALL_AIRPORT EER schema in Figure 8.12 may be represented in UML notation. (*Note:* We have not discussed how to represent categories (union types) in UML, so you do not have to map the categories in this and the following question.)

- 8.22.** Show how the UNIVERSITY EER schema in Figure 8.9 may be represented in UML notation.
- 8.23.** Consider the entity sets and attributes shown in the table below. Place a checkmark in one column in each row to indicate the relationship between the far left and right columns.
- The left side has a relationship with the right side.
 - The right side is an attribute of the left side.
 - The left side is a specialization of the right side.
 - The left side is a generalization of the right side.

Entity Set	(a) Has a Relationship with	(b) Has an Attribute that is	(c) Is a Specialization of	(d) Is a Generalization of	Entity Set or Attribute
1. MOTHER					PERSON
2. DAUGHTER					MOTHER
3. STUDENT					PERSON
4. STUDENT					Student_id
5. SCHOOL					STUDENT
6. SCHOOL					CLASS_ROOM
7. ANIMAL					HORSE
8. HORSE					Breed
9. HORSE					Age
10. EMPLOYEE					SSN
11. FURNITURE					CHAIR
12. CHAIR					Weight
13. HUMAN					WOMAN
14. SOLDIER					PERSON
15. ENEMY_COMBATANT					PERSON

- 8.24.** Draw a UML diagram for storing a played game of chess in a database. You may look at <http://www.chessgames.com> for an application similar to what you are designing. State clearly any assumptions you make in your UML diagram. A sample of assumptions you can make about the scope is as follows:
- The game of chess is played between two players.
 - The game is played on an 8×8 board like the one shown below:



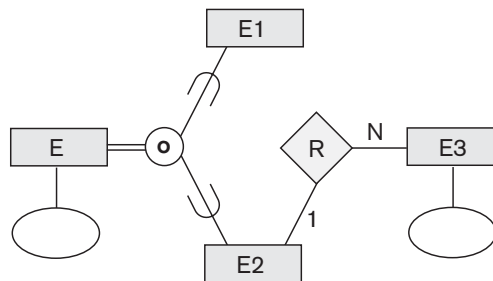
3. The players are assigned a color of black or white at the start of the game.
4. Each player starts with the following pieces (traditionally called chess-men):

a. king	d. 2 bishops
b. queen	e. 2 knights
c. 2 rooks	f. 8 pawns
5. Every piece has its own initial position.
6. Every piece has its own set of legal moves based on the state of the game. You do not need to worry about which moves are or are not legal except for the following issues:
 - a. A piece may move to an empty square or capture an opposing piece.
 - b. If a piece is captured, it is removed from the board.
 - c. If a pawn moves to the last row, it is “promoted” by converting it to another piece (queen, rook, bishop, or knight).

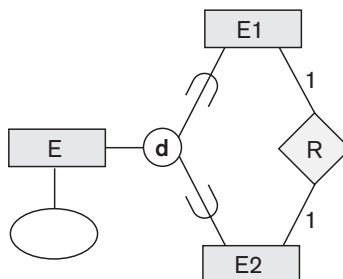
Note: Some of these functions may be spread over multiple classes.

- 8.25. Draw an EER diagram for a game of chess as described in Exercise 8.24. Focus on persistent storage aspects of the system. For example, the system would need to retrieve all the moves of every game played in sequential order.
- 8.26. Which of the following EER diagrams is/are incorrect and why? State clearly any assumptions you make.

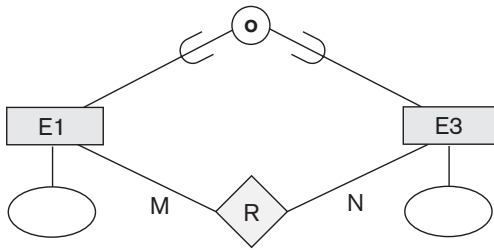
a.



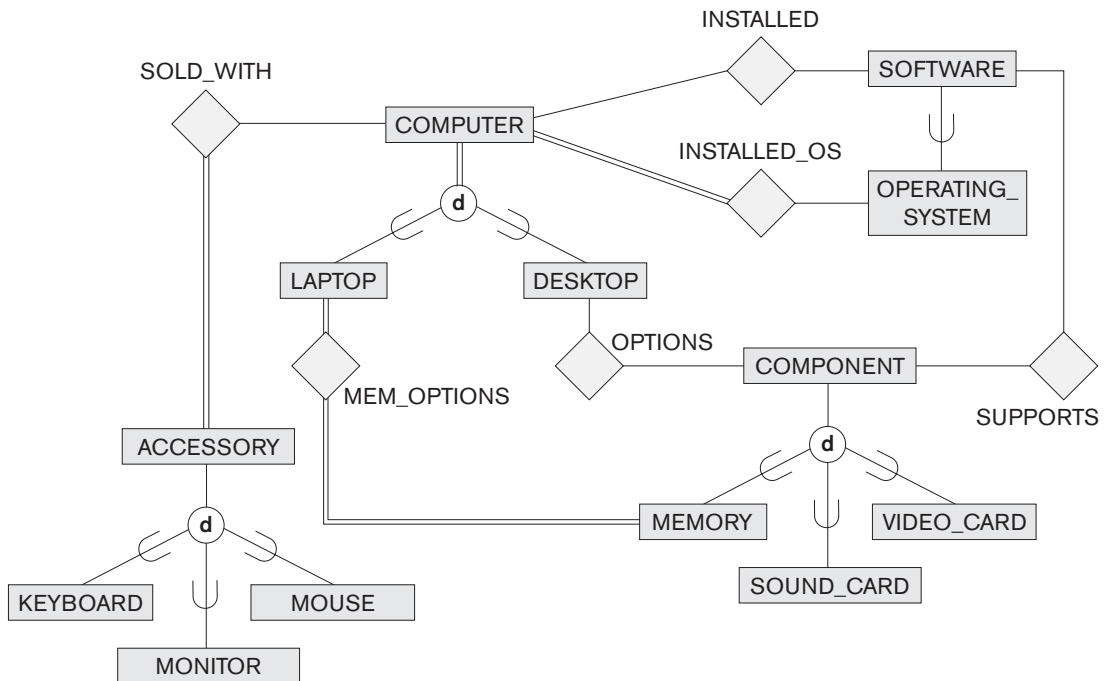
b.



c.



8.27. Consider the following EER diagram that describes the computer systems at a company. Provide your own attributes and key for each entity type. Supply max cardinality constraints justifying your choice. Write a complete narrative description of what this EER diagram represents.



Laboratory Exercises

8.28. Consider a `GRADE_BOOK` database in which instructors within an academic department record points earned by individual students in their classes. The data requirements are summarized as follows:

- Each student is identified by a unique identifier, first and last name, and an e-mail address.
- Each instructor teaches certain courses each term. Each course is identified by a course number, a section number, and the term in which it is taught. For each course he or she teaches, the

instructor specifies the minimum number of points required in order to earn letter grades A, B, C, D, and F. For example, 90 points for an A, 80 points for a B, 70 points for a C, and so forth.

- Students are enrolled in each course taught by the instructor.
- Each course has a number of grading components (such as midterm exam, final exam, project, and so forth). Each grading component has a maximum number of points (such as 100 or 50) and a weight (such as 20% or 10%). The weights of all the grading components of a course usually total 100.
- Finally, the instructor records the points earned by each student in each of the grading components in each of the courses. For example, student 1234 earns 84 points for the midterm exam grading component of the section 2 course CSc2310 in the fall term of 2009. The midterm exam grading component may have been defined to have a maximum of 100 points and a weight of 20% of the course grade.

Design an Enhanced Entity-Relationship diagram for the grade book database and build the design using a data modeling tool such as ERwin or Rational Rose.

- 8.29.** Consider an ONLINE_AUCTION database system in which members (buyers and sellers) participate in the sale of items. The data requirements for this system are summarized as follows:
- The online site has members, each of whom is identified by a unique member number and is described by an e-mail address, name, password, home address, and phone number.
 - A member may be a buyer or a seller. A buyer has a shipping address recorded in the database. A seller has a bank account number and routing number recorded in the database.
 - Items are placed by a seller for sale and are identified by a unique item number assigned by the system. Items are also described by an item title, a description, starting bid price, bidding increment, the start date of the auction, and the end date of the auction.
 - Items are also categorized based on a fixed classification hierarchy (for example, a modem may be classified as COMPUTER→HARDWARE→MODEM).
 - Buyers make bids for items they are interested in. Bid price and time of bid is recorded. The bidder at the end of the auction with the highest bid price is declared the winner and a transaction between buyer and seller may then proceed.
 - The buyer and seller may record feedback regarding their completed transactions. Feedback contains a rating of the other party participating in the transaction (1–10) and a comment.

Design an Enhanced Entity-Relationship diagram for the ONLINE_AUCTION database and build the design using a data modeling tool such as ERwin or Rational Rose.

8.30. Consider a database system for a baseball organization such as the major leagues. The data requirements are summarized as follows:

- The personnel involved in the league include players, coaches, managers, and umpires. Each is identified by a unique personnel id. They are also described by their first and last names along with the date and place of birth.
- Players are further described by other attributes such as their batting orientation (left, right, or switch) and have a lifetime batting average (BA).
- Within the players group is a subset of players called pitchers. Pitchers have a lifetime ERA (earned run average) associated with them.
- Teams are uniquely identified by their names. Teams are also described by the city in which they are located and the division and league in which they play (such as Central division of the American League).
- Teams have one manager, a number of coaches, and a number of players.
- Games are played between two teams with one designated as the home team and the other the visiting team on a particular date. The score (runs, hits, and errors) are recorded for each team. The team with the most runs is declared the winner of the game.
- With each finished game, a winning pitcher and a losing pitcher are recorded. In case there is a save awarded, the save pitcher is also recorded.
- With each finished game, the number of hits (singles, doubles, triples, and home runs) obtained by each player is also recorded.

Design an Enhanced Entity-Relationship diagram for the BASEBALL database and enter the design using a data modeling tool such as ERwin or Rational Rose.

8.31. Consider the EER diagram for the UNIVERSITY database shown in Figure 8.9. Enter this design using a data modeling tool such as ERwin or Rational Rose. Make a list of the differences in notation between the diagram in the text and the corresponding equivalent diagrammatic notation you end up using with the tool.

8.32. Consider the EER diagram for the small AIRPORT database shown in Figure 8.12. Build this design using a data modeling tool such as ERwin or Rational Rose. Be careful as to how you model the category OWNER in this diagram. (*Hint:* Consider using CORPORATION_IS_OWNER and PERSON_IS_OWNER as two distinct relationship types.)

8.33. Consider the UNIVERSITY database described in Exercise 7.16. You already developed an ER schema for this database using a data modeling tool such as

ERwin or Rational Rose in Lab Exercise 7.31. Modify this diagram by classifying COURSES as either UNDERGRAD_COURSES or GRAD_COURSES and INSTRUCTORS as either JUNIOR_PROFESSORS or SENIOR_PROFESSORS. Include appropriate attributes for these new entity types. Then establish relationships indicating that junior instructors teach undergraduate courses while senior instructors teach graduate courses.

Selected Bibliography

Many papers have proposed conceptual or semantic data models. We give a representative list here. One group of papers, including Abrial (1974), Senko's DIAM model (1975), the NIAM method (Verheijen and VanBekkum 1982), and Bracchi et al. (1976), presents semantic models that are based on the concept of binary relationships. Another group of early papers discusses methods for extending the relational model to enhance its modeling capabilities. This includes the papers by Schmid and Swenson (1975), Navathe and Schkolnick (1978), Codd's RM/T model (1979), Furtado (1978), and the structural model of Wiederhold and Elmasri (1979).

The ER model was proposed originally by Chen (1976) and is formalized in Ng (1981). Since then, numerous extensions of its modeling capabilities have been proposed, as in Scheuermann et al. (1979), Dos Santos et al. (1979), Teorey et al. (1986), Gogolla and Hohenstein (1991), and the Entity-Category-Relationship (ECR) model of Elmasri et al. (1985). Smith and Smith (1977) present the concepts of generalization and aggregation. The semantic data model of Hammer and McLeod (1981) introduced the concepts of class/subclass lattices, as well as other advanced modeling concepts.

A survey of semantic data modeling appears in Hull and King (1987). Eick (1991) discusses design and transformations of conceptual schemas. Analysis of constraints for n -ary relationships is given in Soutou (1998). UML is described in detail in Booch, Rumbaugh, and Jacobson (1999). Fowler and Scott (2000) and Stevens and Pooley (2000) give concise introductions to UML concepts.

Fensel (2000, 2003) discuss the Semantic Web and application of ontologies. Uschold and Gruninger (1996) and Gruber (1995) discuss ontologies. The June 2002 issue of *Communications of the ACM* is devoted to ontology concepts and applications. Fensel (2003) is a book that discusses ontologies and e-commerce.

Relational Database Design by ER- and EER-to-Relational Mapping

This chapter discusses how to **design a relational database schema** based on a conceptual schema design. Figure 7.1 presented a high-level view of the database design process, and in this chapter we focus on the logical database design or data model mapping step of database design. We present the procedures to create a relational schema from an Entity-Relationship (ER) or an Enhanced ER (EER) schema. Our discussion relates the constructs of the ER and EER models, presented in Chapters 7 and 8, to the constructs of the relational model, presented in Chapters 3 through 6. Many computer-aided software engineering (CASE) tools are based on the ER or EER models, or other similar models, as we have discussed in Chapters 7 and 8. Many tools use ER or EER diagrams or variations to develop the schema graphically, and then convert it automatically into a relational database schema in the DDL of a specific relational DBMS by employing algorithms similar to the ones presented in this chapter.

We outline a seven-step algorithm in Section 9.1 to convert the basic ER model constructs—entity types (strong and weak), binary relationships (with various structural constraints), n -ary relationships, and attributes (simple, composite, and multivalued)—into relations. Then, in Section 9.2, we continue the mapping algorithm by describing how to map EER model constructs—specialization/generalization and union types (categories)—into relations. Section 9.3 summarizes the chapter.

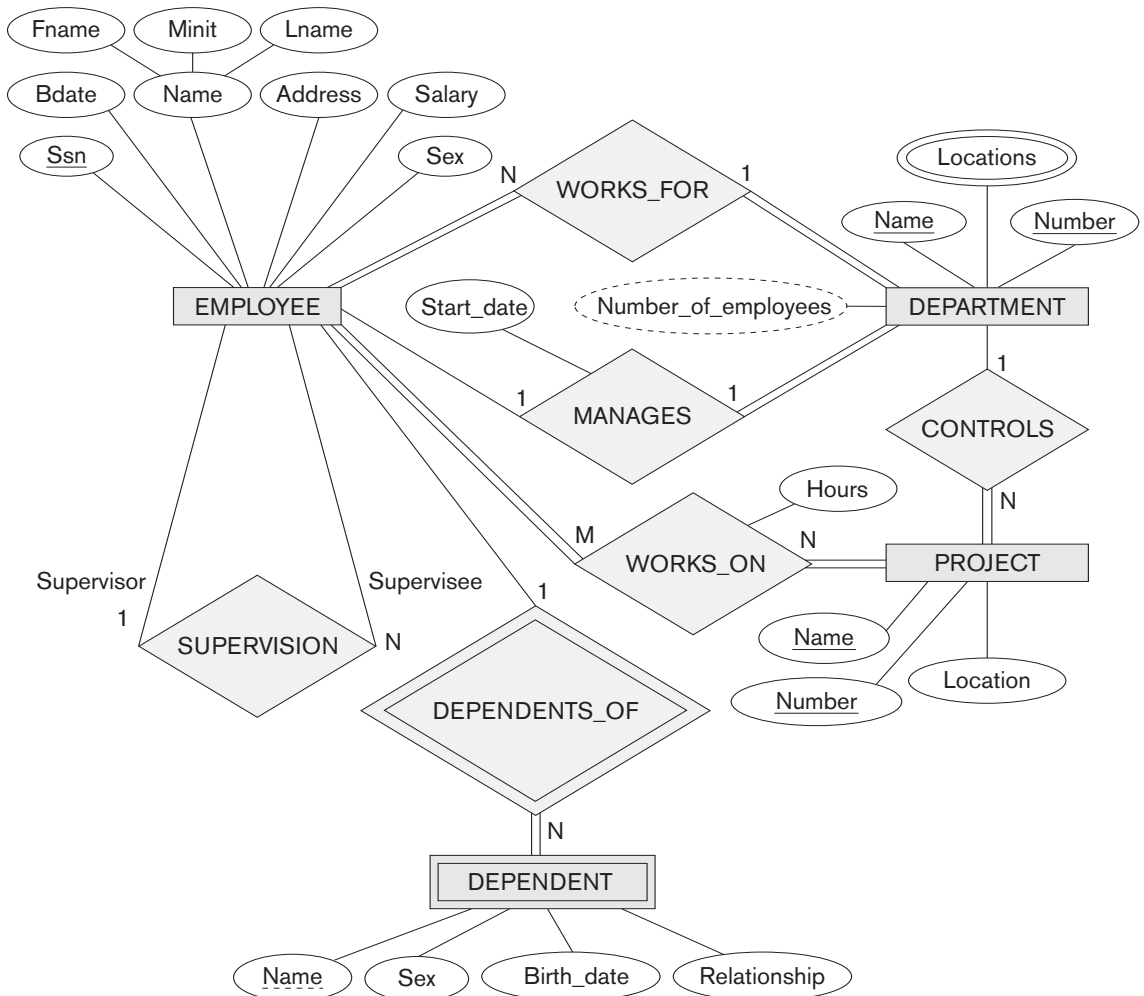
9.1 Relational Database Design Using ER-to-Relational Mapping

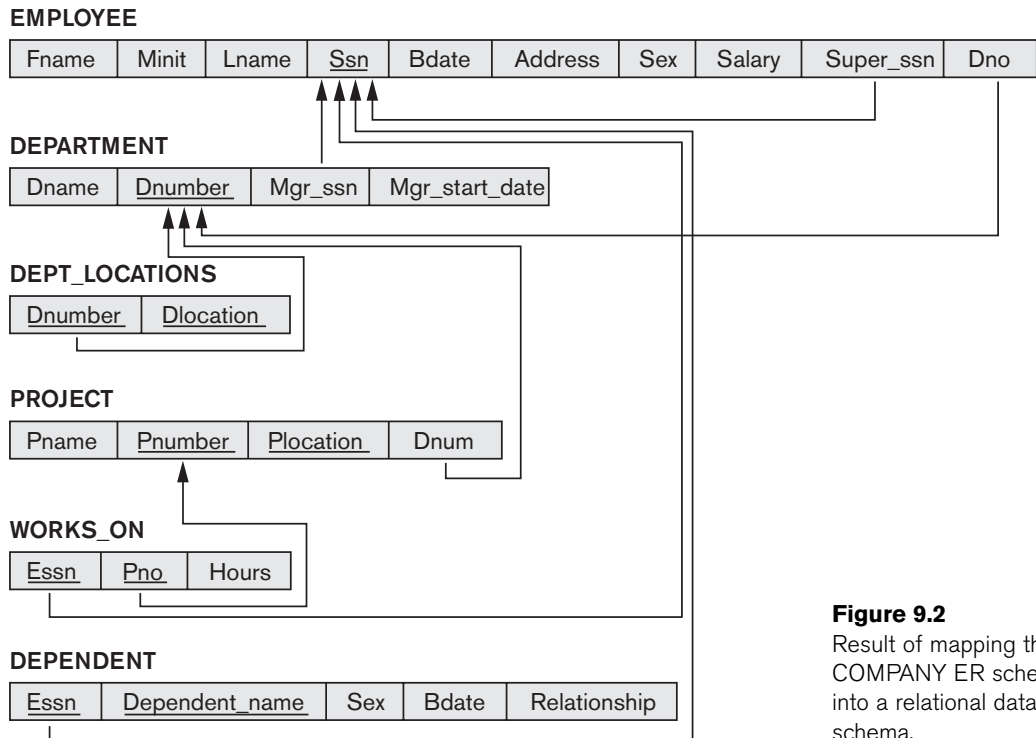
9.1.1 ER-to-Relational Mapping Algorithm

In this section we describe the steps of an algorithm for ER-to-relational mapping. We use the COMPANY database example to illustrate the mapping procedure. The COMPANY ER schema is shown again in Figure 9.1, and the corresponding COMPANY relational database schema is shown in Figure 9.2 to illustrate the map-

Figure 9.1

The ER conceptual schema diagram for the COMPANY database.



**Figure 9.2**

Result of mapping the COMPANY ER schema into a relational database schema.

ping steps. We assume that the mapping will create tables with simple single-valued attributes. The relational model constraints defined in Chapter 3, which include primary keys, unique keys (if any), and referential integrity constraints on the relations, will also be specified in the mapping results.

Step 1: Mapping of Regular Entity Types. For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E . Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as the primary key for R . If the chosen key of E is a composite, then the set of simple attributes that form it will together form the primary key of R .

If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation R . Knowledge about keys is also kept for indexing purposes and other types of analyses.

In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT in Figure 9.2 to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT in Figure 9.1. The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps. These include the

attributes Super_ssn and Dno of EMPLOYEE, Mgr_ssn and Mgr_start_date of DEPARTMENT, and Dnum of PROJECT. In our example, we choose Ssn, Dnumber, and Pnumber as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT, respectively. Knowledge that Dname of DEPARTMENT and Pname of PROJECT are secondary keys is kept for possible use later in the design.

The relations that are created from the mapping of entity types are sometimes called **entity relations** because each tuple represents an entity instance. The result after this mapping step is shown in Figure 9.3(a).

Step 2: Mapping of Weak Entity Types. For each weak entity type W in the ER schema with owner entity type E , create a relation R and include all simple attributes (or simple components of composite attributes) of W as attributes of R . In addition, include as foreign key attributes of R , the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping the identifying relationship type of W . The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W , if any.

If there is a weak entity type E_2 whose owner is also a weak entity type E_1 , then E_1 should be mapped before E_2 to determine its primary key first.

In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT (see Figure 9.3(b)). We include the primary key Ssn of the EMPLOYEE relation—which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we rename it Essn, although this is not necessary.

Figure 9.3
Illustration of some mapping steps.
(a) *Entity relations* after step 1.
(b) Additional *weak entity* relation after step 2.
(c) *Relationship* relation after step 5.
(d) Relation representing multivalued attribute after step 6.

(a)

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary
-------	-------	-------	------------	-------	---------	-----	--------

DEPARTMENT

Dname	<u>Dnumber</u>
-------	----------------

PROJECT

Pname	<u>Pnumber</u>	Plocation
-------	----------------	-----------

(b)

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

(c)

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

(d)

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

The primary key of the `DEPENDENT` relation is the combination {`Essn`, `Dependent_name`}, because `Dependent_name` (also renamed from `Name` in Figure 9.1) is the partial key of `DEPENDENT`.

It is common to choose the propagate (`CASCADE`) option for the referential triggered action (see Section 4.2) on the foreign key in the relation corresponding to the weak entity type, since a weak entity has an existence dependency on its owner entity. This can be used for both `ON UPDATE` and `ON DELETE`.

Step 3: Mapping of Binary 1:1 Relationship Types. For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R . There are three possible approaches: (1) the foreign key approach, (2) the merged relationship approach, and (3) the cross-reference or relationship relation approach. The first approach is the most useful and should be followed unless special conditions exist, as we discuss below.

1. **Foreign key approach:** Choose one of the relations— S , say—and include as a foreign key in S the primary key of T . It is better to choose an entity type with *total participation* in R in the role of S . Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S .

In our example, we map the 1:1 relationship type `MANAGES` from Figure 9.1 by choosing the participating entity type `DEPARTMENT` to serve in the role of S because its participation in the `MANAGES` relationship type is total (every department has a manager). We include the primary key of the `EMPLOYEE` relation as foreign key in the `DEPARTMENT` relation and rename it `Mgr_ssn`. We also include the simple attribute `Start_date` of the `MANAGES` relationship type in the `DEPARTMENT` relation and rename it `Mgr_start_date` (see Figure 9.2).

Note that it is possible to include the primary key of S as a foreign key in T instead. In our example, this amounts to having a foreign key attribute, say `Department_managed` in the `EMPLOYEE` relation, but it will have a `NULL` value for employee tuples who do not manage a department. If only 2 percent of employees manage a department, then 98 percent of the foreign keys would be `NULL` in this case. Another possibility is to have foreign keys in both relations S and T redundantly, but this creates redundancy and incurs a penalty for consistency maintenance.

2. **Merged relation approach:** An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation. This is possible when *both participations are total*, as this would indicate that the two tables will have the exact same number of tuples at all times.
3. **Cross-reference or relationship relation approach:** The third option is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types. As we will see, this approach is required for binary $M:N$ relationships. The relation R is called a **relationship relation** (or sometimes a **lookup table**), because each

tuple in R represents a relationship instance that relates one tuple from S with one tuple from T . The relation R will include the primary key attributes of S and T as foreign keys to S and T . The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R . The drawback is having an extra relation, and requiring an extra join operation when combining related tuples from the tables.

Step 4: Mapping of Binary 1:N Relationship Types. For each regular binary 1:N relationship type R , identify the relation S that represents the participating entity type at the N -side of the relationship type. Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R ; we do this because each entity instance on the N -side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of S .

In our example, we now map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION from Figure 9.1. For WORKS_FOR we include the primary key Dnumber of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it Dno. For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it Super_ssn. The CONTROLS relationship is mapped to the foreign key attribute Dnum of PROJECT, which references the primary key Dnumber of the DEPARTMENT relation. These foreign keys are shown in Figure 9.2.

An alternative approach is to use the **relationship relation** (cross-reference) option as in the third option for binary 1:1 relationships. We create a separate relation R whose attributes are the primary keys of S and T , which will also be foreign keys to S and T . The primary key of R is the same as the primary key of S . This option can be used if few tuples in S participate in the relationship to avoid excessive NULL values in the foreign key.

Step 5: Mapping of Binary M:N Relationship Types. For each binary M:N relationship type R , create a new relation S to represent R . Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their *combination* will form the primary key of S . Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S . Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for 1:1 or 1:N relationship types) because of the M:N cardinality ratio; we must create a separate *relationship relation* S .

In our example, we map the M:N relationship type WORKS_ON from Figure 9.1 by creating the relation WORKS_ON in Figure 9.2. We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them Pno and Essn, respectively. We also include an attribute Hours in WORKS_ON to represent the Hours attribute of the relationship type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {Essn, Pno}. This **relationship relation** is shown in Figure 9.3(c).

The propagate (CASCADE) option for the referential triggered action (see Section 4.2) should be specified on the foreign keys in the relation corresponding to the relationship R , since each relationship instance has an existence dependency on each of the entities it relates. This can be used for both ON UPDATE and ON DELETE.

Notice that we can always map 1:1 or 1:N relationships in a manner similar to M:N relationships by using the cross-reference (relationship relation) approach, as we discussed earlier. This alternative is particularly useful when few relationship instances exist, in order to avoid NULL values in foreign keys. In this case, the primary key of the relationship relation will be *only one* of the foreign keys that reference the participating entity relations. For a 1:N relationship, the primary key of the relationship relation will be the foreign key that references the entity relation on the N-side. For a 1:1 relationship, either foreign key can be used as the primary key of the relationship relation.

Step 6: Mapping of Multivalued Attributes. For each multivalued attribute A , create a new relation R . This relation R will include an attribute corresponding to A , plus the primary key attribute K —as a foreign key in R —of the relation that represents the entity type or relationship type that has A as a multivalued attribute. The primary key of R is the combination of A and K . If the multivalued attribute is composite, we include its simple components.

In our example, we create a relation DEPT_LOCATIONS (see Figure 9.3(d)). The attribute Dlocation represents the multivalued attribute LOCATIONS of DEPARTMENT, while Dnumber—as foreign key—represents the primary key of the DEPARTMENT relation. The primary key of DEPT_LOCATIONS is the combination of {Dnumber, Dlocation}. A separate tuple will exist in DEPT_LOCATIONS for each location that a department has.

The propagate (CASCADE) option for the referential triggered action (see Section 4.2) should be specified on the foreign key in the relation R corresponding to the multivalued attribute for both ON UPDATE and ON DELETE. We should also note that the key of R when mapping a composite, multivalued attribute requires some analysis of the meaning of the component attributes. In some cases, when a multivalued attribute is composite, only some of the component attributes are required to be part of the key of R ; these attributes are similar to a partial key of a weak entity type that corresponds to the multivalued attribute (see Section 7.5).

Figure 9.2 shows the COMPANY relational database schema obtained with steps 1 through 6, and Figure 3.6 shows a sample database state. Notice that we did not yet discuss the mapping of n -ary relationship types ($n > 2$) because none exist in Figure 9.1; these are mapped in a similar way to M:N relationship types by including the following additional step in the mapping algorithm.

Step 7: Mapping of N -ary Relationship Types. For each n -ary relationship type R , where $n > 2$, create a new relation S to represent R . Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the n -ary relationship type (or

simple components of composite attributes) as attributes of S . The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types. However, if the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E' corresponding to E (see the discussion in Section 7.9.2 concerning constraints on n -ary relationships).

For example, consider the relationship type SUPPLY in Figure 7.17. This can be mapped to the relation SUPPLY shown in Figure 9.4, whose primary key is the combination of the three foreign keys {Sname, Part_no, Proj_name}.

9.1.2 Discussion and Summary of Mapping for ER Model Constructs

Table 9.1 summarizes the correspondences between ER and relational model constructs and constraints.

One of the main points to note in a relational schema, in contrast to an ER schema, is that relationship types are not represented explicitly; instead, they are represented by having two attributes A and B , one a primary key and the other a foreign key (over the same domain) included in two relations S and T . Two tuples in S and T are related when they have the same value for A and B . By using the EQUIJOIN operation (or NATURAL JOIN if the two join attributes have the same name) over $S.A$ and $T.B$, we can combine all pairs of related tuples from S and T and materialize the relationship. When a binary 1:1 or 1:N relationship type is involved, a single join operation is usually needed. For a binary M:N relationship type, two join operations are needed, whereas for n -ary relationship types, n joins are needed to fully materialize the relationship instances.

Figure 9.4
Mapping the n -ary
relationship type
SUPPLY from Figure
7.17(a).

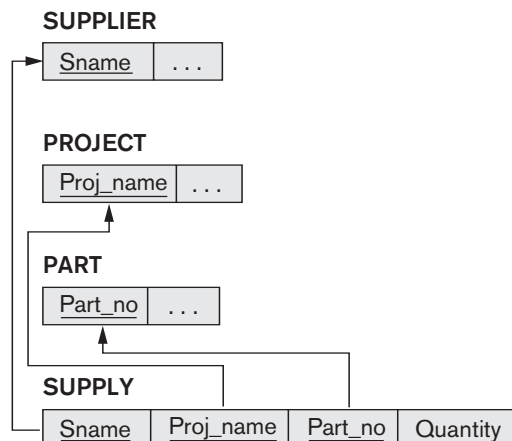


Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	<i>Entity</i> relation
1:1 or 1:N relationship type	Foreign key (or <i>relationship</i> relation)
M:N relationship type	<i>Relationship</i> relation and <i>two</i> foreign keys
<i>n</i> -ary relationship type	<i>Relationship</i> relation and <i>n</i> foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

For example, to form a relation that includes the employee name, project name, and hours that the employee works on each project, we need to connect each EMPLOYEE tuple to the related PROJECT tuples via the WORKS_ON relation in Figure 9.2. Hence, we must apply the EQUIJOIN operation to the EMPLOYEE and WORKS_ON relations with the join condition $Ssn = Essn$, and then apply another EQUIJOIN operation to the resulting relation and the PROJECT relation with join condition $Pno = Pnumber$. In general, when multiple relationships need to be traversed, numerous join operations must be specified. A relational database user must always be aware of the foreign key attributes in order to use them correctly in combining related tuples from two or more relations. This is sometimes considered to be a drawback of the relational data model, because the foreign key/primary key correspondences are not always obvious upon inspection of relational schemas. If an EQUIJOIN is performed among attributes of two relations that do not represent a foreign key/primary key relationship, the result can often be meaningless and may lead to spurious data. For example, the reader can try joining the PROJECT and DEPT_LOCATIONS relations on the condition $Dlocation = Plocation$ and examine the result (see the discussion of spurious tuples in Section 15.1.4).

In the relational schema we create a separate relation for *each* multivalued attribute. For a particular entity with a set of values for the multivalued attribute, the key attribute value of the entity is repeated once for each value of the multivalued attribute in a separate tuple because the basic relational model does *not* allow multiple values (a list, or a set of values) for an attribute in a single tuple. For example, because department 5 has three locations, three tuples exist in the DEPT_LOCATIONS relation in Figure 3.6; each tuple specifies one of the locations. In our example, we apply EQUIJOIN to DEPT_LOCATIONS and DEPARTMENT on the Dnumber attribute to get the values of all locations along with other DEPARTMENT attributes. In the resulting relation, the values of the other DEPARTMENT attributes are repeated in separate tuples for every location that a department has.

The basic relational algebra does not have a NEST or COMPRESS operation that would produce a set of tuples of the form $\{<1, \text{'Houston'}>, <4, \text{'Stafford'}>, <5, \text{'Bellaire'}>, \text{'Sugarland'}>, \text{'Houston'}>\}$ from the DEPT_LOCATIONS relation in Figure 3.6. This is a serious drawback of the basic normalized or *flat* version of the relational model. The object data model and object-relational systems (see Chapter 11) do allow multivalued attributes.

9.2 Mapping EER Model Constructs to Relations

Next, we discuss the mapping of EER model constructs to relations by extending the ER-to-relational mapping algorithm that was presented in Section 9.1.1.

9.2.1 Mapping of Specialization or Generalization

There are several options for mapping a number of subclasses that together form a specialization (or alternatively, that are generalized into a superclass), such as the {SECRETARY, TECHNICIAN, ENGINEER} subclasses of EMPLOYEE in Figure 8.4. We can add a further step to our ER-to-relational mapping algorithm from Section 9.1.1, which has seven steps, to handle the mapping of specialization. Step 8, which follows, gives the most common options; other mappings are also possible. We discuss the conditions under which each option should be used. We use $\text{Attrs}(R)$ to denote *the attributes of relation R*, and $\text{PK}(R)$ to denote *the primary key of R*. First we describe the mapping formally, then we illustrate it with examples.

Step 8: Options for Mapping Specialization or Generalization. Convert each specialization with m subclasses $\{S_1, S_2, \dots, S_m\}$ and (generalized) superclass C , where the attributes of C are $\{k, a_1, \dots, a_n\}$ and k is the (primary) key, into relation schemas using one of the following options:

- **Option 8A: Multiple relations—superclass and subclasses.** Create a relation L for C with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\}$ and $\text{PK}(L) = k$. Create a relation L_i for each subclass S_i , $1 \leq i \leq m$, with the attributes $\text{Attrs}(L_i) = \{k\} \cup \{\text{attributes of } S_i\}$ and $\text{PK}(L_i) = k$. This option works for any specialization (total or partial, disjoint or overlapping).
- **Option 8B: Multiple relations—subclass relations only.** Create a relation L_i for each subclass S_i , $1 \leq i \leq m$, with the attributes $\text{Attrs}(L_i) = \{\text{attributes of } S_i\} \cup \{k, a_1, \dots, a_n\}$ and $\text{PK}(L_i) = k$. This option only works for a specialization whose subclasses are *total* (every entity in the superclass must belong to (at least) one of the subclasses). Additionally, it is only recommended if the specialization has the *disjointness constraint* (see Section 8.3.1). If the specialization is *overlapping*, the same entity may be duplicated in several relations.
- **Option 8C: Single relation with one type attribute.** Create a single relation L with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t\}$ and $\text{PK}(L) = k$. The attribute t is called a **type** (or

discriminating) attribute whose value indicates the subclass to which each tuple belongs, if any. This option works only for a specialization whose subclasses are *disjoint*, and has the potential for generating many NULL values if many specific attributes exist in the subclasses.

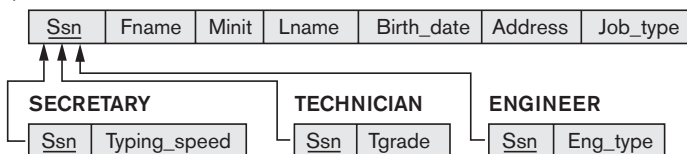
- **Option 8D: Single relation with multiple type attributes.** Create a single relation schema L with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t_1, t_2, \dots, t_m\}$ and $\text{PK}(L) = k$. Each t_i , $1 \leq i \leq m$, is a **Boolean type attribute** indicating whether a tuple belongs to subclass S_i . This option is used for a specialization whose subclasses are *overlapping* (but will also work for a disjoint specialization).

Options 8A and 8B can be called the **multiple-relation options**, whereas options 8C and 8D can be called the **single-relation options**. Option 8A creates a relation L for the superclass C and its attributes, plus a relation L_i for each subclass S_i ; each L_i includes the specific (or local) attributes of S_i , plus the primary key of the superclass C , which is propagated to L_i and becomes its primary key. It also becomes a foreign key to the superclass relation. An EQUIJOIN operation on the primary key between any L_i and L produces all the specific and inherited attributes of the entities in S_i . This option is illustrated in Figure 9.5(a) for the EER schema in Figure 8.4. Option 8A works for any constraints on the specialization: disjoint or overlapping, total or partial. Notice that the constraint

$$\pi_{\langle k \rangle}(L_i) \subseteq \pi_{\langle k \rangle}(L)$$

must hold for each L_i . This specifies a foreign key from each L_i to L , as well as an *inclusion dependency* $L_i.k < L.k$ (see Section 16.5).

(a) EMPLOYEE



(b) CAR

<u>Vehicle_id</u>	License_plate_no	Price	Max_speed	No_of_passengers
-------------------	------------------	-------	-----------	------------------

TRUCK

<u>Vehicle_id</u>	License_plate_no	Price	No_of_axles	Tonnage
-------------------	------------------	-------	-------------	---------

(c) EMPLOYEE

<u>Ssn</u>	Fname	Minit	Lname	Birth_date	Address	Job_type	Typing_speed	Tgrade	Eng_type
------------	-------	-------	-------	------------	---------	----------	--------------	--------	----------

(d) PART

<u>Part_no</u>	Description	Mflag	Drawing_no	Manufacture_date	Batch_no	Pflag	Supplier_name	List_price
----------------	-------------	-------	------------	------------------	----------	-------	---------------	------------

Figure 9.5

Options for mapping specialization or generalization. (a) Mapping the EER schema in Figure 8.4 using option 8A. (b) Mapping the EER schema in Figure 8.3(b) using option 8B. (c) Mapping the EER schema in Figure 8.4 using option 8C. (d) Mapping Figure 8.5 using option 8D with Boolean type fields Mflag and Pflag.

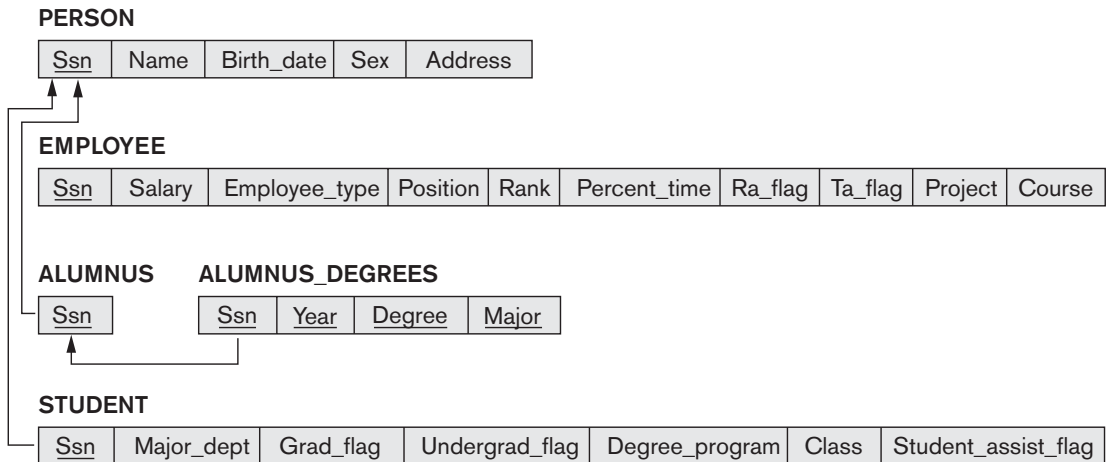
In option 8B, the EQUIJOIN operation between each subclass and the superclass is *built into* the schema and the relation L is done away with, as illustrated in Figure 9.5(b) for the EER specialization in Figure 8.3(b). This option works well only when *both* the disjoint and total constraints hold. If the specialization is not total, an entity that does not belong to any of the subclasses S_i is lost. If the specialization is not disjoint, an entity belonging to more than one subclass will have its inherited attributes from the superclass C stored redundantly in more than one L_i . With option 8B, no relation holds all the entities in the superclass C ; consequently, we must apply an OUTER UNION (or FULL OUTER JOIN) operation (see Section 6.4) to the L_i relations to retrieve all the entities in C . The result of the outer union will be similar to the relations under options 8C and 8D except that the type fields will be missing. Whenever we search for an arbitrary entity in C , we must search all the m relations L_i .

Options 8C and 8D create a single relation to represent the superclass C and all its subclasses. An entity that does not belong to some of the subclasses will have NULL values for the specific attributes of these subclasses. These options are not recommended if many specific attributes are defined for the subclasses. If few specific subclass attributes exist, however, these mappings are preferable to options 8A and 8B because they do away with the need to specify EQUIJOIN and OUTER UNION operations; therefore, they can yield a more efficient implementation.

Option 8C is used to handle disjoint subclasses by including a single **type** (or **image** or **discriminating**) **attribute** t to indicate to which of the m subclasses each tuple belongs; hence, the domain of t could be $\{1, 2, \dots, m\}$. If the specialization is partial, t can have NULL values in tuples that do not belong to any subclass. If the specialization is attribute-defined, that attribute serves the purpose of t and t is not needed; this option is illustrated in Figure 9.5(c) for the EER specialization in Figure 8.4.

Option 8D is designed to handle overlapping subclasses by including m **Boolean type** (or **flag**) fields, one for *each* subclass. It can also be used for disjoint subclasses. Each type field t_i can have a domain $\{\text{yes}, \text{no}\}$, where a value of yes indicates that the tuple is a member of subclass S_i . If we use this option for the EER specialization in Figure 8.4, we would include three type attributes—`Is_a_secretary`, `Is_a_engineer`, and `Is_a_technician`—instead of the `Job_type` attribute in Figure 9.5(c). Notice that it is also possible to create a single type attribute of m *bits* instead of the m type fields. Figure 9.5(d) shows the mapping of the specialization from Figure 8.5 using option 8D.

When we have a multilevel specialization (or generalization) hierarchy or lattice, we do not have to follow the same mapping option for all the specializations. Instead, we can use one mapping option for part of the hierarchy or lattice and other options for other parts. Figure 9.6 shows one possible mapping into relations for the EER lattice in Figure 8.6. Here we used option 8A for PERSON/{EMPLOYEE, ALUMNUS, STUDENT}, option 8C for EMPLOYEE/{STAFF, FACULTY, STUDENT_ASSISTANT} by including the type attribute `Employee_type`, and option 8D for STUDENT_ASSISTANT/{RESEARCH_ASSISTANT, TEACHING_ASSISTANT} by including the type attributes `Ta_flag` and `Ra_flag` in EMPLOYEE, STUDENT/

**Figure 9.6**

Mapping the EER specialization lattice in Figure 8.8 using multiple options.

STUDENT_ASSISTANT by including the type attributes *Student_assist_flag* in STUDENT, and STUDENT/{GRADUATE_STUDENT, UNDERGRADUATE_STUDENT} by including the type attributes *Grad_flag* and *Undergrad_flag* in STUDENT. In Figure 9.6, all attributes whose names end with *type* or *flag* are type fields.

9.2.2 Mapping of Shared Subclasses (Multiple Inheritance)

A shared subclass, such as ENGINEERING_MANAGER in Figure 8.6, is a subclass of several superclasses, indicating multiple inheritance. These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category (union type) as we discussed in Section 8.4. We can apply any of the options discussed in step 8 to a shared subclass, subject to the restrictions discussed in step 8 of the mapping algorithm. In Figure 9.6, options 8C and 8D are used for the shared subclass STUDENT_ASSISTANT. Option 8C is used in the EMPLOYEE relation (*Employee_type* attribute) and option 8D is used in the STUDENT relation (*Student_assist_flag* attribute).

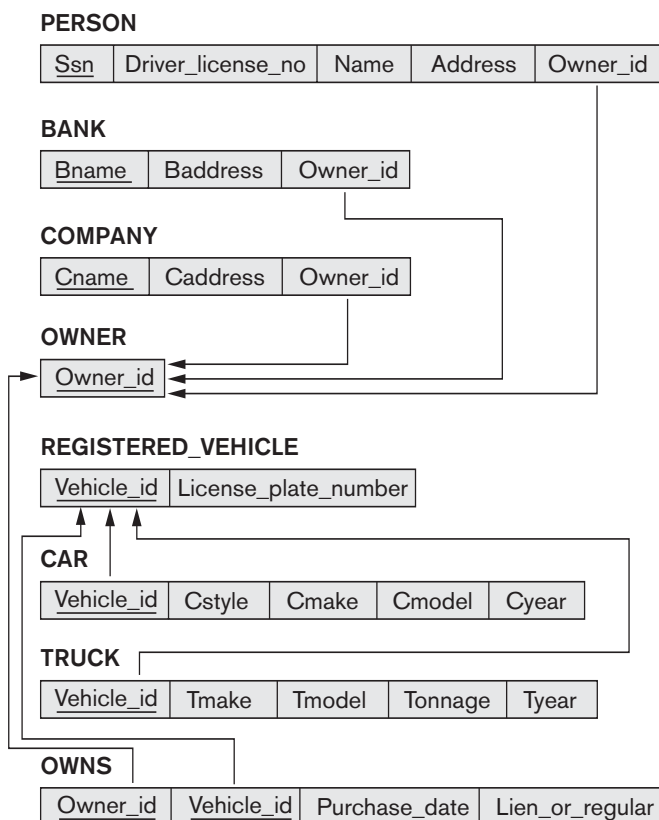
9.2.3 Mapping of Categories (Union Types)

We add another step to the mapping procedure—step 9—to handle categories. A category (or union type) is a subclass of the *union* of two or more superclasses that can have different keys because they can be of different entity types (see Section 8.4). An example is the OWNER category shown in Figure 8.8, which is a subset of the union of three entity types PERSON, BANK, and COMPANY. The other category in that figure, REGISTERED_VEHICLE, has two superclasses that have the same key attribute.

Step 9: Mapping of Union Types (Categories). For mapping a category whose defining superclasses have different keys, it is customary to specify a new key attribute, called a **surrogate key**, when creating a relation to correspond to the category. The keys of the defining classes are different, so we cannot use any one of them exclusively to identify all entities in the category. In our example in Figure 8.8, we create a relation OWNER to correspond to the OWNER category, as illustrated in Figure 9.7, and include any attributes of the category in this relation. The primary key of the OWNER relation is the surrogate key, which we called Owner_id. We also include the surrogate key attribute Owner_id as foreign key in each relation corresponding to a superclass of the category, to specify the correspondence in values between the surrogate key and the key of each superclass. Notice that if a particular PERSON (or BANK or COMPANY) entity is not a member of OWNER, it would have a NULL value for its Owner_id attribute in its corresponding tuple in the PERSON (or BANK or COMPANY) relation, and it would not have a tuple in the OWNER relation. It is also recommended to add a type attribute (not shown in Figure 9.7) to the OWNER relation to indicate the particular entity type to which each tuple belongs (PERSON or BANK or COMPANY).

Figure 9.7

Mapping the EER categories (union types) in Figure 8.8 to relations.



For a category whose superclasses have the same key, such as **VEHICLE** in Figure 8.8, there is no need for a surrogate key. The mapping of the **REGISTERED_VEHICLE** category, which illustrates this case, is also shown in Figure 9.7.

9.3 Summary

In Section 9.1, we showed how a conceptual schema design in the ER model can be mapped to a relational database schema. An algorithm for ER-to-relational mapping was given and illustrated by examples from the **COMPANY** database. Table 9.1 summarized the correspondences between the ER and relational model constructs and constraints. Next, we added additional steps to the algorithm in Section 9.2 for mapping the constructs from the EER model into the relational model. Similar algorithms are incorporated into graphical database design tools to create a relational schema from a conceptual schema design automatically.

Review Questions

- 9.1. Discuss the correspondences between the ER model constructs and the relational model constructs. Show how each ER model construct can be mapped to the relational model and discuss any alternative mappings.
- 9.2. Discuss the options for mapping EER model constructs to relations.

Exercises

- 9.3. Try to map the relational schema in Figure 6.14 into an ER schema. This is part of a process known as *reverse engineering*, where a conceptual schema is created for an existing implemented database. State any assumptions you make.
- 9.4. Figure 9.8 shows an ER schema for a database that can be used to keep track of transport ships and their locations for maritime authorities. Map this schema into a relational schema and specify all primary keys and foreign keys.
- 9.5. Map the **BANK** ER schema of Exercise 7.23 (shown in Figure 7.21) into a relational schema. Specify all primary keys and foreign keys. Repeat for the **AIRLINE** schema (Figure 7.20) of Exercise 7.19 and for the other schemas for Exercises 7.16 through 7.24.
- 9.6. Map the EER diagrams in Figures 8.9 and 8.12 into relational schemas. Justify your choice of mapping options.
- 9.7. Is it possible to successfully map a binary M:N relationship type without requiring a new relation? Why or why not?

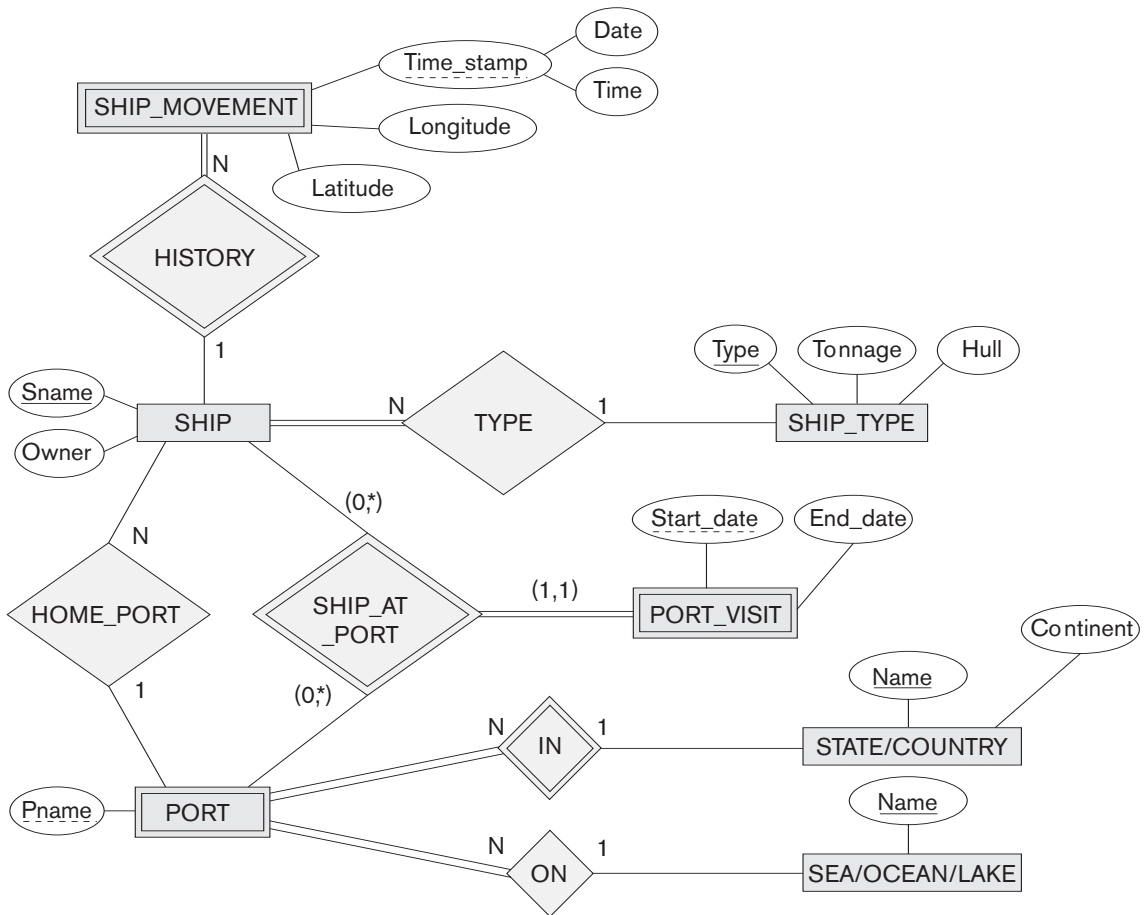


Figure 9.8
An ER schema for a SHIP_TRACKING database.

9.8. Consider the EER diagram in Figure 9.9 for a car dealer.

Map the EER schema into a set of relations. For the VEHICLE to CAR/TRUCK/SUV generalization, consider the four options presented in Section 9.2.1 and show the relational schema design under each of those options.

9.9. Using the attributes you provided for the EER diagram in Exercise 8.27, map the complete schema into a set of relations. Choose an appropriate option out of 8A thru 8D from Section 9.2.1 in doing the mapping of generalizations and defend your choice.

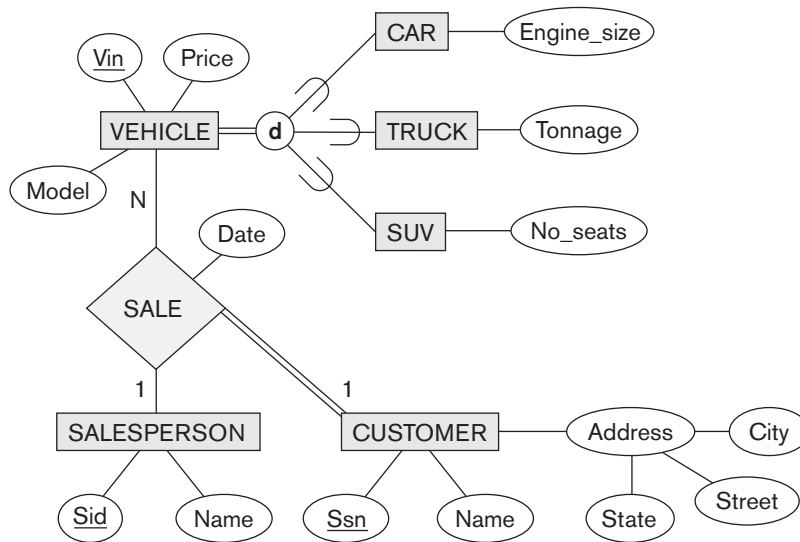


Figure 9.9
EER diagram for
a car dealer

Laboratory Exercises

- 9.10. Consider the ER design for the UNIVERSITY database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 7.31. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.
- 9.11. Consider the ER design for the MAIL_ORDER database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 7.32. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.
- 9.12. Consider the ER design for the CONFERENCE_REVIEW database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 7.34. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.
- 9.13. Consider the EER design for the GRADE_BOOK database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 8.28. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.
- 9.14. Consider the EER design for the ONLINE_AUCTION database that was modeled using a tool like ERwin or Rational Rose in Laboratory Exercise 8.29. Using the SQL schema generation feature of the modeling tool, generate the SQL schema for an Oracle database.

Selected Bibliography

The original ER-to-relational mapping algorithm was described in Chen's classic paper (Chen 1976) that presented the original ER model. Batini et al. (1992) discuss a variety of mapping algorithms from ER and EER models to legacy models and vice versa.

Basics of Functional Dependencies and Normalization for Relational Databases

In Chapters 3 through 6, we presented various aspects of the relational model and the languages associated with it. Each *relation schema* consists of a number of attributes, and the *relational database schema* consists of a number of relation schemas. So far, we have assumed that attributes are grouped to form a relation schema by using the common sense of the database designer or by mapping a database schema design from a conceptual data model such as the ER or Enhanced-ER (EER) data model. These models make the designer identify entity types and relationship types and their respective attributes, which leads to a natural and logical grouping of the attributes into relations when the mapping procedures discussed in Chapter 9 are followed. However, we still need some formal way of analyzing why one grouping of attributes into a relation schema may be better than another. While discussing database design in Chapters 7 through 10, we did not develop any measure of appropriateness or *goodness* to measure the quality of the design, other than the intuition of the designer. In this chapter we discuss some of the theory that has been developed with the goal of evaluating relational schemas for design quality—that is, to measure formally why one set of groupings of attributes into relation schemas is better than another.

There are two levels at which we can discuss the *goodness* of relation schemas. The first is the **logical** (or **conceptual**) **level**—how users interpret the relation schemas and the meaning of their attributes. Having good relation schemas at this level enables users to understand clearly the meaning of the data in the relations, and hence to formulate their queries correctly. The second is the **implementation** (or

physical storage) level—how the tuples in a base relation are stored and updated. This level applies only to schemas of base relations—which will be physically stored as files—whereas at the logical level we are interested in schemas of both base relations and views (virtual relations). The relational database design theory developed in this chapter applies mainly to *base relations*, although some criteria of appropriateness also apply to views, as shown in Section 15.1.

As with many design problems, database design may be performed using two approaches: bottom-up or top-down. A **bottom-up design methodology** (also called *design by synthesis*) considers the basic relationships *among individual attributes* as the starting point and uses those to construct relation schemas. This approach is not very popular in practice¹ because it suffers from the problem of having to collect a large number of binary relationships among attributes as the starting point. For practical situations, it is next to impossible to capture binary relationships among all such pairs of attributes. In contrast, a **top-down design methodology** (also called *design by analysis*) starts with a number of groupings of attributes into relations that exist together naturally, for example, on an invoice, a form, or a report. The relations are then analyzed individually and collectively, leading to further decomposition until all desirable properties are met. The theory described in this chapter is applicable to both the top-down and bottom-up design approaches, but is more appropriate when used with the top-down approach.

Relational database design ultimately produces a set of relations. The implicit goals of the design activity are *information preservation* and *minimum redundancy*. Information is very hard to quantify—hence we consider information preservation in terms of maintaining all concepts, including attribute types, entity types, and relationship types as well as generalization/specialization relationships, which are described using a model such as the EER model. Thus, the relational design must preserve all of these concepts, which are originally captured in the conceptual design after the conceptual to logical design mapping. Minimizing redundancy implies minimizing redundant storage of the same information and reducing the need for multiple updates to maintain consistency across multiple copies of the same information in response to real-world events that require making an update.

We start this chapter by informally discussing some criteria for good and bad relation schemas in Section 15.1. In Section 15.2, we define the concept of *functional dependency*, a formal constraint among attributes that is the main tool for formally measuring the appropriateness of attribute groupings into relation schemas. In Section 15.3, we discuss normal forms and the process of normalization using functional dependencies. Successive normal forms are defined to meet a set of desirable constraints expressed using functional dependencies. The normalization procedure consists of applying a series of tests to relations to meet these increasingly stringent requirements and decompose the relations when necessary. In Section 15.4, we dis-

¹An exception in which this approach is used in practice is based on a model called the *binary relational model*. An example is the NIAM methodology (Verheijen and VanBekkum, 1982).

cuss more general definitions of normal forms that can be directly applied to any given design and do not require step-by-step analysis and normalization. Sections 15.5 to 15.7 discuss further normal forms up to the fifth normal form. In Section 15.6 we introduce the multivalued dependency (MVD), followed by the join dependency (JD) in Section 15.7. Section 15.8 summarizes the chapter.

Chapter 16 continues the development of the theory related to the design of good relational schemas. We discuss desirable properties of relational decomposition—nonadditive join property and functional dependency preservation property. A general algorithm that tests whether or not a decomposition has the nonadditive (or *lossless*) join property (Algorithm 16.3 is also presented). We then discuss properties of functional dependencies and the concept of a minimal cover of dependencies. We consider the bottom-up approach to database design consisting of a set of algorithms to design relations in a desired normal form. These algorithms assume as input a given set of functional dependencies and achieve a relational design in a target normal form while adhering to the above desirable properties. In Chapter 16 we also define additional types of dependencies that further enhance the evaluation of the *goodness* of relation schemas.

If Chapter 16 is not covered in a course, we recommend a quick introduction to the desirable properties of decomposition and the discussion of Property NJB in Section 16.2.

15.1 Informal Design Guidelines for Relation Schemas

Before discussing the formal theory of relational database design, we discuss four *informal guidelines* that may be used as *measures to determine the quality* of relation schema design:

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples

These measures are not always independent of one another, as we will see.

15.1.1 Imparting Clear Semantics to Attributes in Relations

Whenever we group attributes to form a relation schema, we assume that attributes belonging to one relation have certain real-world meaning and a proper interpretation associated with them. The **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple. In Chapter 3 we discussed how a relation can be interpreted as a set of facts. If the conceptual design described in Chapters 7 and 8 is done carefully and the mapping procedure in Chapter 9 is followed systematically, the relational schema design should have a clear meaning.

In general, the easier it is to explain the semantics of the relation, the better the relation schema design will be. To illustrate this, consider Figure 15.1, a simplified version of the COMPANY relational database schema in Figure 3.5, and Figure 15.2, which presents an example of populated relation states of this schema. The meaning of the EMPLOYEE relation schema is quite simple: Each tuple represents an employee, with values for the employee's name (Ename), Social Security number (Ssn), birth date (Bdate), and address (Address), and the number of the department that the employee works for (Dnumber). The Dnumber attribute is a foreign key that represents an *implicit relationship* between EMPLOYEE and DEPARTMENT. The semantics of the DEPARTMENT and PROJECT schemas are also straightforward: Each DEPARTMENT tuple represents a department entity, and each PROJECT tuple represents a project entity. The attribute Dmgr_ssn of DEPARTMENT relates a department to the employee who is its manager, while Dnum of PROJECT relates a project to its controlling department; both are foreign key attributes. The ease with which the meaning of a relation's attributes can be explained is an *informal measure* of how well the relation is designed.

Figure 15.1

A simplified COMPANY relational database schema.

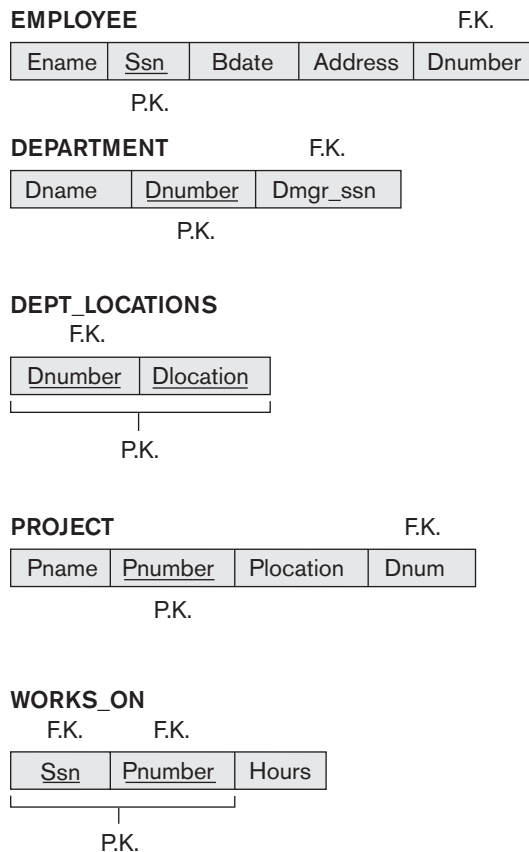


Figure 15.2

Sample database state for the relational database schema in Figure 15.1.

EMPLOYEE

Ename	<u>Ssn</u>	Bdate	Address	Dnumber
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

<u>Ssn</u>	<u>Pnumber</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	Null

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

The semantics of the other two relation schemas in Figure 15.1 are slightly more complex. Each tuple in `DEPT_LOCATIONS` gives a department number (`Dnumber`) and *one of* the locations of the department (`Dlocation`). Each tuple in `WORKS_ON` gives an employee Social Security number (`Ssn`), the project number of *one of* the projects that the employee works on (`Pnumber`), and the number of hours per week that the employee works on that project (`Hours`). However, both schemas have a well-defined and unambiguous interpretation. The schema `DEPT_LOCATIONS` represents a multivalued attribute of `DEPARTMENT`, whereas `WORKS_ON` represents an M:N relationship between `EMPLOYEE` and `PROJECT`. Hence, all the relation schemas in Figure 15.1 may be considered as easy to explain and therefore good from the standpoint of having clear semantics. We can thus formulate the following informal design guideline.

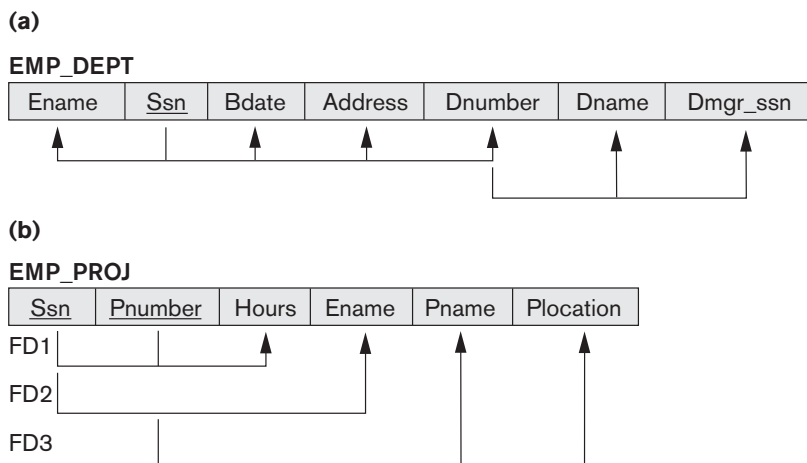
Guideline 1

Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Intuitively, if a relation schema corresponds to one entity type or one relationship type, it is straightforward to interpret and to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Examples of Violating Guideline 1. The relation schemas in Figures 15.3(a) and 15.3(b) also have clear semantics. (The reader should ignore the lines under the relations for now; they are used to illustrate functional dependency notation, discussed in Section 15.2.) A tuple in the `EMP_DEPT` relation schema in Figure 15.3(a) represents a single employee but includes additional information—namely, the name (`Dname`) of the department for which the employee works and the Social Security number (`Dmgr_ssn`) of the department manager. For the `EMP_PROJ` relation in Figure 15.3(b), each tuple relates an employee to a project but also includes

Figure 15.3

Two relation schemas suffering from update anomalies. (a) `EMP_DEPT` and (b) `EMP_PROJ`.



the employee name (Ename), project name (Pname), and project location (Plocation). Although there is nothing wrong logically with these two relations, they violate Guideline 1 by mixing attributes from distinct real-world entities: EMP_DEPT mixes attributes of employees and departments, and EMP_PROJ mixes attributes of employees and projects and the WORKS_ON relationship. Hence, they fare poorly against the above measure of design quality. They may be used as views, but they cause problems when used as base relations, as we discuss in the following section.

15.1.2 Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space. For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT in Figure 15.2 with that for an EMP_DEPT base relation in Figure 15.4, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT. In EMP_DEPT, the attribute values pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for *every employee who works for that department*. In contrast, each department's information appears only once in the DEPARTMENT relation in Figure 15.2. Only the department number (Dnumber) is repeated in the EMPLOYEE relation for each employee who works in that department as a foreign key. Similar comments apply to the EMP_PROJ relation (see Figure 15.4), which augments the WORKS_ON relation with additional attributes from EMPLOYEE and PROJECT.

Storing natural joins of base relations leads to an additional problem referred to as **update anomalies**. These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.²

Insertion Anomalies. Insertion anomalies can be differentiated into two types, illustrated by the following examples based on the EMP_DEPT relation:

- To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs (if the employee does not work for a department as yet). For example, to insert a new tuple for an employee who works in department number 5, we must enter all the attribute values of department 5 correctly so that they are *consistent* with the corresponding values for department 5 in other tuples in EMP_DEPT. In the design of Figure 15.2, we do not have to worry about this consistency problem because we enter only the department number in the employee tuple; all other attribute values of department 5 are recorded only once in the database, as a single tuple in the DEPARTMENT relation.
- It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place NULL values in the

²These anomalies were identified by Codd (1972a) to justify the need for normalization of relations, as we shall discuss in Section 15.3.

					Redundancy	
EMP_DEPT						
Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

			Redundancy		Redundancy	
EMP_PROJ						
Ssn	Pnumber	Hours	Ename	Pname	Plocation	
123456789	1	32.5	Smith, John B.	ProductX	Bellaire	
123456789	2	7.5	Smith, John B.	ProductY	Sugarland	
666884444	3	40.0	Narayan, Ramesh K.	ProductZ	Houston	
453453453	1	20.0	English, Joyce A.	ProductX	Bellaire	
453453453	2	20.0	English, Joyce A.	ProductY	Sugarland	
333445555	2	10.0	Wong, Franklin T.	ProductY	Sugarland	
333445555	3	10.0	Wong, Franklin T.	ProductZ	Houston	
333445555	10	10.0	Wong, Franklin T.	Computerization	Stafford	
333445555	20	10.0	Wong, Franklin T.	Reorganization	Houston	
999887777	30	30.0	Zelaya, Alicia J.	Newbenefits	Stafford	
999887777	10	10.0	Zelaya, Alicia J.	Computerization	Stafford	
987987987	10	35.0	Jabbar, Ahmad V.	Computerization	Stafford	
987987987	30	5.0	Jabbar, Ahmad V.	Newbenefits	Stafford	
987654321	30	20.0	Wallace, Jennifer S.	Newbenefits	Stafford	
987654321	20	15.0	Wallace, Jennifer S.	Reorganization	Houston	
888665555	20	Null	Borg, James E.	Reorganization	Houston	

Figure 15.4

Sample states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations in Figure 15.2. These may be stored as base relations for performance reasons.

attributes for employee. This violates the entity integrity for EMP_DEPT because Ssn is its primary key. Moreover, when the first employee is assigned to that department, we do not need this tuple with NULL values any more. This problem does not occur in the design of Figure 15.2 because a department is entered in the DEPARTMENT relation whether or not any employees work for it, and whenever an employee is assigned to that department, a corresponding tuple is inserted in EMPLOYEE.

Deletion Anomalies. The problem of deletion anomalies is related to the second insertion anomaly situation just discussed. If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database. This problem does not occur in the database of Figure 15.2 because DEPARTMENT tuples are stored separately.

Modification Anomalies. In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of *all* employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.³

It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data as well as require unnecessary updates that can be avoided; hence, we can state the next guideline as follows.

Guideline 2

Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present,⁴ note them clearly and make sure that the programs that update the database will operate correctly.

The second guideline is consistent with and, in a way, a restatement of the first guideline. We can also see the need for a more formal approach to evaluating whether a design meets these guidelines. Sections 15.2 through 15.4 provide these needed formal concepts. It is important to note that these guidelines may sometimes *have to be violated* in order to *improve the performance* of certain queries. If EMP_DEPT is used as a stored relation (known otherwise as a *materialized view*) in addition to the base relations of EMPLOYEE and DEPARTMENT, the anomalies in EMP_DEPT must be noted and accounted for (for example, by using triggers or stored procedures that would make automatic updates). This way, whenever the base relation is updated, we do not end up with inconsistencies. In general, it is advisable to use anomaly-free base relations and to specify views that include the joins for placing together the attributes frequently referenced in important queries.

15.1.3 NULL Values in Tuples

In some schema designs we may group many attributes together into a “fat” relation. If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level and may

³This is not as serious as the other problems, because all tuples can be updated by a single SQL query.

⁴Other application considerations may dictate and make certain anomalies unavoidable. For example, the EMP_DEPT relation may correspond to a query or a report that is frequently required.

also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level.⁵ Another problem with NULLs is how to account for them when aggregate operations such as COUNT or SUM are applied. SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable.⁶ Moreover, NULLs can have multiple interpretations, such as the following:

- The attribute *does not apply* to this tuple. For example, Visa_status may not apply to U.S. students.
- The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.
- The value is *known but absent*; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we may state another guideline.

Guideline 3

As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL. If NULLs are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.

Using space efficiently and avoiding joins with NULL values are the two overriding criteria that determine whether to include the columns that may have NULLs in a relation or to have a separate relation for those columns (with the appropriate key columns). For example, if only 15 percent of employees have individual offices, there is little justification for including an attribute Office_number in the EMPLOYEE relation; rather, a relation EMP_OFFICES(Essn, Office_number) can be created to include tuples for only the employees with individual offices.

15.1.4 Generation of Spurious Tuples

Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure 15.5(a), which can be used instead of the single EMP_PROJ relation in Figure 15.3(b). A tuple in EMP_LOCS means that the employee whose name is Ename works on *some project* whose location is Plocation. A tuple in EMP_PROJ1 refers to the fact that the employee whose Social Security number is Ssn works Hours per week on the project whose name, number, and location are Pname, Pnumber, and Plocation. Figure 15.5(b) shows relation states of EMP_LOCS and EMP_PROJ1 corresponding to the

⁵This is because inner and outer joins produce different results when NULLs are involved in joins. The users must thus be aware of the different meanings of the various types of joins. Although this is reasonable for sophisticated users, it may be difficult for others.

⁶In Section 5.5.1 we presented comparisons involving NULL values where the outcome (in three-valued logic) are TRUE, FALSE, and UNKNOWN.

(a)

EMP_LOCS

Ename	Plocation
-------	-----------

P.K.

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
-----	---------	-------	-------	-----------

P.K.

Figure 15.5

Particularly poor design for the EMP_PROJ relation in Figure 15.3(b). (a) The two relation schemas EMP_LOCS and EMP_PROJ1. (b) The result of projecting the extension of EMP_PROJ from Figure 15.4 onto the relations EMP_LOCS and EMP_PROJ1.

(b)

EMP_LOCS

Ename	Plocation
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelaya, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
999887777	30	30.0	Newbenefits	Stafford
999887777	10	10.0	Computerization	Stafford
987987987	10	35.0	Computerization	Stafford
987987987	30	5.0	Newbenefits	Stafford
987654321	30	20.0	Newbenefits	Stafford
987654321	20	15.0	Reorganization	Houston
888665555	20	NULL	Reorganization	Houston

EMP_PROJ relation in Figure 15.4, which are obtained by applying the appropriate PROJECT (π) operations to EMP_PROJ (ignore the dashed lines in Figure 15.5(b) for now).

Suppose that we used EMP_PROJ1 and EMP_LOCS as the base relations instead of EMP_PROJ. This produces a particularly bad schema design because we cannot recover the information that was originally in EMP_PROJ from EMP_PROJ1 and EMP_LOCS. If we attempt a NATURAL JOIN operation on EMP_PROJ1 and EMP_LOCS, the result produces many more tuples than the original set of tuples in EMP_PROJ. In Figure 15.6, the result of applying the join to only the tuples *above* the dashed lines in Figure 15.5(b) is shown (to reduce the size of the resulting relation). Additional tuples that were not in EMP_PROJ are called **spurious tuples**

	Ssn	Pnumber	Hours	Pname	Plocation	Ename
	123456789	1	32.5	ProductX	Bellaire	Smith, John B.
*	123456789	1	32.5	ProductX	Bellaire	English, Joyce A.
	123456789	2	7.5	ProductY	Sugarland	Smith, John B.
*	123456789	2	7.5	ProductY	Sugarland	English, Joyce A.
*	123456789	2	7.5	ProductY	Sugarland	Wong, Franklin T.
	666884444	3	40.0	ProductZ	Houston	Narayan, Ramesh K.
*	666884444	3	40.0	ProductZ	Houston	Wong, Franklin T.
*	453453453	1	20.0	ProductX	Bellaire	Smith, John B.
	453453453	1	20.0	ProductX	Bellaire	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Smith, John B.
	453453453	2	20.0	ProductY	Sugarland	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	2	10.0	ProductY	Sugarland	Smith, John B.
*	333445555	2	10.0	ProductY	Sugarland	English, Joyce A.
	333445555	2	10.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	3	10.0	ProductZ	Houston	Narayan, Ramesh K.
	333445555	3	10.0	ProductZ	Houston	Wong, Franklin T.
	333445555	10	10.0	Computerization	Stafford	Wong, Franklin T.
*	333445555	20	10.0	Reorganization	Houston	Narayan, Ramesh K.
	333445555	20	10.0	Reorganization	Houston	Wong, Franklin T.

*
*
*

Figure 15.6

Result of applying NATURAL JOIN to the tuples above the dashed lines in EMP_PROJ1 and EMP_LOCS of Figure 15.5. Generated spurious tuples are marked by asterisks.

because they represent spurious information that is not valid. The spurious tuples are marked by asterisks (*) in Figure 15.6.

Decomposing EMP_PROJ into EMP_LOCS and EMP_PROJ1 is undesirable because when we JOIN them back using NATURAL JOIN, we do not get the correct original information. This is because in this case Plocation is the attribute that relates EMP_LOCS and EMP_PROJ1, and Plocation is neither a primary key nor a foreign key in either EMP_LOCS or EMP_PROJ1. We can now informally state another design guideline.

Guideline 4

Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated. Avoid relations that contain

matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

This informal guideline obviously needs to be stated more formally. In Section 16.2 we discuss a formal condition called the nonadditive (or lossless) join property that guarantees that certain joins do not produce spurious tuples.

15.1.5 Summary and Discussion of Design Guidelines

In Sections 15.1.1 through 15.1.4, we informally discussed situations that lead to problematic relation schemas and we proposed informal guidelines for a good relational design. The problems we pointed out, which can be detected without additional tools of analysis, are as follows:

- Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values
- Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

In the rest of this chapter we present formal concepts and theory that may be used to define the *goodness* and *badness* of *individual* relation schemas more precisely. First we discuss functional dependency as a tool for analysis. Then we specify the three normal forms and Boyce-Codd normal form (BCNF) for relation schemas. The strategy for achieving a good design is to decompose a badly designed relation appropriately. We also briefly introduce additional normal forms that deal with additional dependencies. In Chapter 16, we discuss the properties of decomposition in detail, and provide algorithms that design relations bottom-up by using the functional dependencies as a starting point.

15.2 Functional Dependencies

So far we have dealt with the informal measures of database design. We now introduce a formal tool for analysis of relational schemas that enables us to detect and describe some of the above-mentioned problems in precise terms. The single most important concept in relational schema design theory is that of a functional dependency. In this section we formally define the concept, and in Section 15.3 we see how it can be used to define normal forms for relation schemas.

15.2.1 Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has n attributes A_1, A_2, \dots, A_n ; let us think of the whole database as being described by a single **universal**

relation schema $R = \{A_1, A_2, \dots, A_n\}$.⁷ We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.⁸

Definition. A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a *constraint* on the possible tuples that can form a relation state r of R . The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are *determined by*, the values of the X component; alternatively, the values of the X component of a tuple uniquely (or **functionally**) *determine* the values of the Y component. We also say that there is a functional dependency from X to Y , or that Y is **functionally dependent** on X . The abbreviation for functional dependency is **FD** or **f.d.** The set of attributes X is called the **left-hand side** of the FD, and Y is called the **right-hand side**.

Thus, X functionally determines Y in a relation schema R if, and only if, whenever two tuples of $r(R)$ agree on their X -value, they must necessarily agree on their Y -value. Note the following:

- If a constraint on R states that there cannot be more than one tuple with a given X -value in any relation instance $r(R)$ —that is, X is a **candidate key** of R —this implies that $X \rightarrow Y$ for any subset of attributes Y of R (because the key constraint implies that no two tuples in any legal state $r(R)$ will have the same value of X). If X is a candidate key of R , then $X \rightarrow R$.
- If $X \rightarrow Y$ in R , this does not say whether or not $Y \rightarrow X$ in R .

A functional dependency is a property of the **semantics** or **meaning of the attributes**. The database designers will use their understanding of the semantics of the attributes of R —that is, how they relate to one another—to specify the functional dependencies that should hold on *all* relation states (extensions) r of R . Whenever the semantics of two sets of attributes in R indicate that a functional dependency should hold, we specify the dependency as a constraint. Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of R . Hence, the main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold *at all times*. Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning. For example, $\{\text{State, Driver_license_number}\} \rightarrow \text{Ssn}$ should hold for any adult in the United States and hence should hold whenever these attributes appear in a relation. It is also possible that certain functional dependencies may cease to

⁷This concept of a universal relation is important when we discuss the algorithms for relational database design in Chapter 16.

⁸This assumption implies that every attribute in the database should have a distinct name. In Chapter 3 we prefixed attribute names by relation names to achieve uniqueness whenever attributes in distinct relations had the same name.

exist in the real world if the relationship changes. For example, the FD $\text{Zip_code} \rightarrow \text{Area_code}$ used to exist as a relationship between postal codes and telephone number codes in the United States, but with the proliferation of telephone area codes it is no longer true.

Consider the relation schema EMP_PROJ in Figure 15.3(b); from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

- a. $\text{Ssn} \rightarrow \text{Ename}$
- b. $\text{Pnumber} \rightarrow \{\text{Pname}, \text{Plocation}\}$
- c. $\{\text{Ssn}, \text{Pnumber}\} \rightarrow \text{Hours}$

These functional dependencies specify that (a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename), (b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation), and (c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours). Alternatively, we say that Ename is functionally determined by (or functionally dependent on) Ssn, or *given a value of Ssn, we know the value of Ename*, and so on.

A functional dependency is a *property of the relation schema R*, not of a particular legal relation state r of R . Therefore, an FD *cannot* be inferred automatically from a given relation extension r but must be defined explicitly by someone who knows the semantics of the attributes of R . For example, Figure 15.7 shows a particular state of the TEACH relation schema. Although at first glance we may think that $\text{Text} \rightarrow \text{Course}$, we cannot confirm this unless we know that it is true *for all possible legal states* of TEACH. It is, however, sufficient to demonstrate *a single counterexample* to disprove a functional dependency. For example, because 'Smith' teaches both 'Data Structures' and 'Data Management,' we can conclude that Teacher *does not* functionally determine Course.

Given a populated relation, one cannot determine which FDs hold and which do not unless the meaning of and the relationships among the attributes are known. All one can say is that a certain FD *may* exist if it holds in that particular extension. One cannot guarantee its existence until the meaning of the corresponding attributes is clearly understood. One can, however, emphatically state that a certain FD *does not*

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Figure 15.7

A relation state of TEACH with a *possible* functional dependency $\text{TEXT} \rightarrow \text{COURSE}$. However, $\text{TEACHER} \rightarrow \text{COURSE}$ is ruled out.

hold if there are tuples that show the violation of such an FD. See the illustrative example relation in Figure 15.8. Here, the following FDs *may hold* because the four tuples in the current extension have no violation of these constraints: $B \rightarrow C$; $C \rightarrow B$; $\{A, B\} \rightarrow C$; $\{A, B\} \rightarrow D$; and $\{C, D\} \rightarrow B$. However, the following *do not hold* because we already have violations of them in the given extension: $A \rightarrow B$ (tuples 1 and 2 violate this constraint); $B \rightarrow A$ (tuples 2 and 3 violate this constraint); $D \rightarrow C$ (tuples 3 and 4 violate it).

Figure 15.3 introduces a **diagrammatic notation** for displaying FDs: Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, while the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.

We denote by F the set of functional dependencies that are specified on relation schema R . Typically, the schema designer specifies the functional dependencies that are *semantically obvious*; usually, however, numerous other functional dependencies hold in *all* legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in F . Those other dependencies can be *inferred* or *deduced* from the FDs in F . We defer the details of inference rules and properties of functional dependencies to Chapter 16.

15.3 Normal Forms Based on Primary Keys

Having introduced functional dependencies, we are now ready to use them to specify some aspects of the semantics of relation schemas. We assume that a set of functional dependencies is given for each relation, and that each relation has a designated primary key; this information combined with the tests (conditions) for normal forms drives the *normalization process* for relational schema design. Most practical relational design projects take one of the following two approaches:

- Perform a conceptual schema design using a conceptual model such as ER or EER and map the conceptual design into a set of relations
- Design the relations based on external knowledge derived from an existing implementation of files or forms or reports

Following either of these approaches, it is then useful to evaluate the relations for goodness and decompose them further as needed to achieve higher normal forms, using the normalization theory presented in this chapter and the next. We focus in

Figure 15.8

A relation $R(A, B, C, D)$ with its extension.

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

this section on the first three normal forms for relation schemas and the intuition behind them, and discuss how they were developed historically. More general definitions of these normal forms, which take into account all candidate keys of a relation rather than just the primary key, are deferred to Section 15.4.

We start by informally discussing normal forms and the motivation behind their development, as well as reviewing some definitions from Chapter 3 that are needed here. Then we discuss the first normal form (1NF) in Section 15.3.4, and present the definitions of second normal form (2NF) and third normal form (3NF), which are based on primary keys, in Sections 15.3.5 and 15.3.6, respectively.

15.3.1 Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**. The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as *relational design by analysis*. Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively; these are briefly discussed in Sections 15.6 and 15.7.

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies discussed in Section 15.1.2. It can be considered as a “filtering” or “purification” process to make the design have successively better quality. Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database designers with the following:

- A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes
- A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree

Definition. The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

Normal forms, when considered *in isolation* from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each

relation schema in the database is, say, in BCNF or 3NF. Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:

- The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem discussed in Section 15.1.4 does not occur with respect to the relation schemas created after decomposition.
- The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

The nonadditive join property is extremely critical and **must be achieved at any cost**, whereas the dependency preservation property, although desirable, is sometimes sacrificed, as we discuss in Section 16.1.2. We defer the presentation of the formal concepts and techniques that guarantee the above two properties to Chapter 16.

15.3.2 Practical Use of Normal Forms

Most practical design projects acquire existing designs of databases from previous designs, designs in legacy models, or from existing files. Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously. Although several higher normal forms have been defined, such as the 4NF and 5NF that we discuss in Sections 15.6 and 15.7, the practical utility of these normal forms becomes questionable when the constraints on which they are based are rare, and hard to understand or to detect by the database designers and users who must discover these constraints. Thus, database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.

Another point worth noting is that the database designers *need not* normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF, for performance reasons, such as those discussed at the end of Section 15.1.2. Doing so incurs the corresponding penalties of dealing with the anomalies.

Definition. Denormalization is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

15.3.3 Definitions of Keys and Attributes Participating in Keys

Before proceeding further, let's look again at the definitions of keys of a relation schema from Chapter 3.

Definition. A **superkey** of a relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more.

The difference between a key and a superkey is that a key has to be *minimal*; that is, if we have a key $K = \{A_1, A_2, \dots, A_k\}$ of R , then $K - \{A_i\}$ is not a key of R for any A_i , $1 \leq i \leq k$. In Figure 15.1, $\{\text{Ssn}\}$ is a key for EMPLOYEE, whereas $\{\text{Ssn}\}$, $\{\text{Ssn}, \text{Ename}\}$, $\{\text{Ssn}, \text{Ename}, \text{Bdate}\}$, and any set of attributes that includes Ssn are all superkeys.

If a relation schema has more than one key, each is called a **candidate key**. One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called secondary keys. In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a default superkey. In Figure 15.1, $\{\text{Ssn}\}$ is the only candidate key for EMPLOYEE, so it is also the primary key.

Definition. An attribute of relation schema R is called a **prime attribute** of R if it is a member of *some candidate key* of R . An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

In Figure 15.1, both Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

We now present the first three normal forms: 1NF, 2NF, and 3NF. These were proposed by Codd (1972a) as a sequence to achieve the desirable state of 3NF relations by progressing through the intermediate states of 1NF and 2NF if needed. As we shall see, 2NF and 3NF attack different problems. However, for historical reasons, it is customary to follow them in that sequence; hence, by definition a 3NF relation *already satisfies* 2NF.

15.3.4 First Normal Form

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple*. In other words, 1NF disallows *relations within relations* or *relations as attribute values within tuples*. The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.

Consider the DEPARTMENT relation schema shown in Figure 15.1, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in Figure 15.9(a). We assume that each department can have a *number of* locations. The DEPARTMENT schema and a sample relation state are shown in Figure 15.9. As we can see, this is not in 1NF because Dlocations is not an atomic attribute, as illustrated by the first tuple in Figure 15.9(b). There are two ways we can look at the Dlocations attribute:

- The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnumber.

(a)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations

(b)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	<u>Dlocation</u>
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 15.9

Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

- The domain of Dlocations contains sets of values and hence is nonatomic. In this case, $Dnumber \rightarrow Dlocations$ because each set is considered a single member of the attribute domain.⁹

In either case, the DEPARTMENT relation in Figure 15.9 is not in 1NF; in fact, it does not even qualify as a relation according to our definition of relation in Section 3.1. There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute Dlocations that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key Dnumber of DEPARTMENT. The primary key of this relation is the combination {Dnumber, Dlocation}, as shown in Figure 15.2. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.

⁹In this case we can consider the domain of Dlocations to be the **power set** of the set of single locations; that is, the domain is made up of all possible subsets of the set of single locations.

2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 15.9(c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing *redundancy* in the relation.
3. If a *maximum number of values* is known for the attribute—for example, if it is known that *at most three locations* can exist for a department—replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3. This solution has the disadvantage of introducing *NULL values* if most departments have fewer than three locations. It further introduces spurious semantics about the ordering among the location values that is not originally intended. Querying on this attribute becomes more difficult; for example, consider how you would write the query: *List the departments that have ‘Bellaire’ as one of their locations* in this design.

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.

First normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it*. Figure 15.10 shows how the EMP_PROJ relation could appear if nesting is allowed. Each tuple represents an employee entity, and a relation PROJS(Pnumber, Hours) *within each tuple* represents the employee’s projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

EMP_PROJ(Ssn, Ename, {PROJS(Pnumber, Hours)})

The set braces { } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses (). Interestingly, recent trends for supporting complex objects (see Chapter 11) and XML data (see Chapter 12) attempt to allow and formalize nested relations within relational database systems, which were disallowed early on by 1NF.

Notice that Ssn is the primary key of the EMP_PROJ relation in Figures 15.10(a) and (b), while Pnumber is the **partial** key of the nested relation; that is, within each tuple, the nested relation must have unique values of Pnumber. To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2, as shown in Figure 15.10(c).

This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations. This is useful in converting an unnormalized relation schema with many levels of nesting into 1NF relations. The

(a)

EMP_PROJ		Projs	
Ssn	Ename	Pnumber	Hours

(b)

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

Figure 15.10

Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a *nested relation* attribute PROJS. (b) Sample extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

(c)

EMP_PROJ1	
<u>Ssn</u>	Ename

EMP_PROJ2

<u>Ssn</u>	<u>Pnumber</u>	Hours
------------	----------------	-------

existence of more than one multivalued attribute in one relation must be handled carefully. As an example, consider the following non-1NF relation:

PERSON (Ss#, {Car_lic#}, {Phone#})

This relation represents the fact that a person has multiple cars and multiple phones. If strategy 2 above is followed, it results in an all-key relation:

PERSON_IN_1NF (Ss#, Car_lic#, Phone#)

To avoid introducing any extraneous relationship between Car_lic# and Phone#, all possible combinations of values are represented for every Ss#, giving rise to redundancy. This leads to the problems handled by multivalued dependencies and 4NF, which we will discuss in Section 15.6. The right way to deal with the two multivalued attributes in PERSON shown previously is to decompose it into two separate relations, using strategy 1 discussed above: P1(Ss#, Car_lic#) and P2(Ss#, Phone#).

15.3.5 Second Normal Form

Second normal form (2NF) is based on the concept of *full functional dependency*. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does *not* functionally determine Y . A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figure 15.3(b), $\{Ssn, Pnumber\} \rightarrow Hours$ is a full dependency (neither $Ssn \rightarrow Hours$ nor $Pnumber \rightarrow Hours$ holds). However, the dependency $\{Ssn, Pnumber\} \rightarrow Ename$ is partial because $Ssn \rightarrow Ename$ holds.

Definition. A relation schema R is in 2NF if every nonprime attribute A in R is *fully functionally dependent* on the primary key of R .

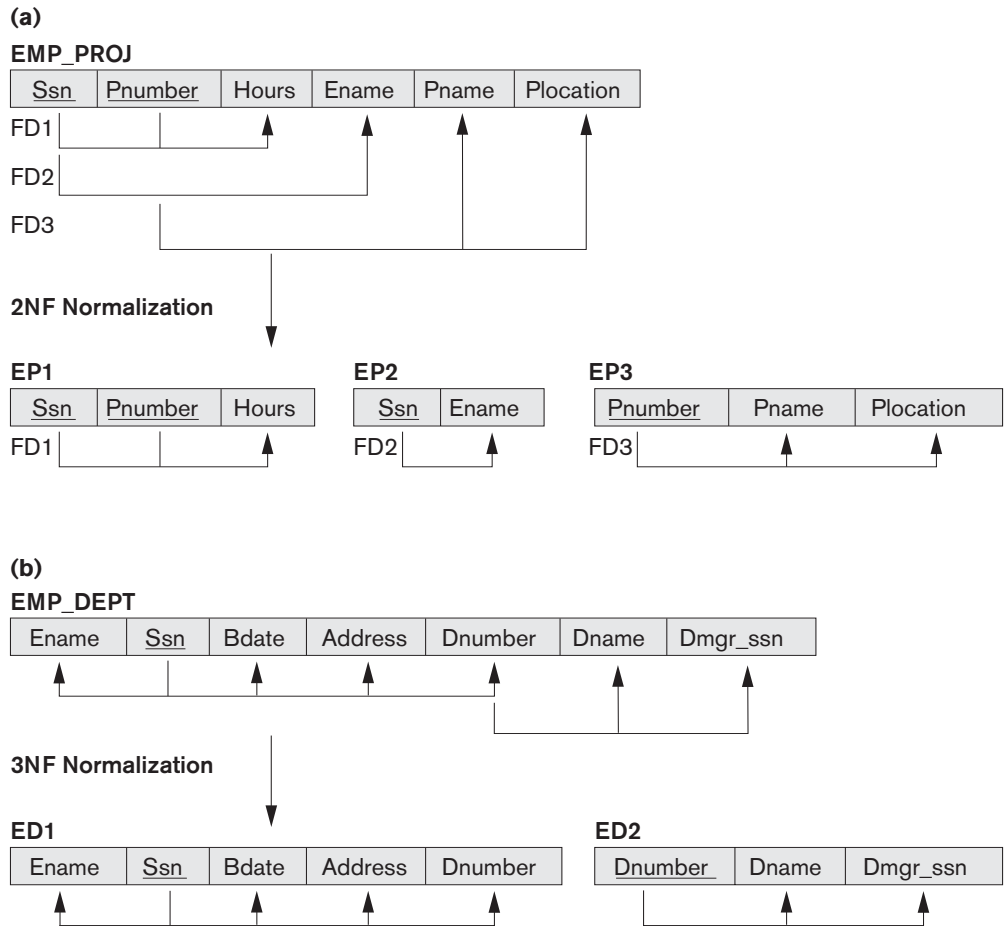
The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure 15.3(b) is in 1NF but is not in 2NF. The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3. The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key $\{Ssn, Pnumber\}$ of EMP_PROJ, thus violating the 2NF test.

If a relation schema is not in 2NF, it can be *second normalized* or *2NF normalized* into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure 15.3(b) lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 15.11(a), each of which is in 2NF.

15.3.6 Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R ,¹⁰ and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency $Ssn \rightarrow Dmgr_ssn$ is transitive through Dnumber in EMP_DEPT in Figure 15.3(a), because both the

¹⁰This is the general definition of transitive dependency. Because we are concerned only with primary keys in this section, we allow transitive dependencies where X is the primary key but Z may be (a subset of) a candidate key.

**Figure 15.11**

Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

dependencies $Ssn \rightarrow Dnumber$ and $Dnumber \rightarrow Dmgr_ssn$ hold *and* $Dnumber$ is neither a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of $Dmgr_ssn$ on $Dnumber$ is undesirable in EMP_DEPT since $Dnumber$ is not a key of EMP_DEPT.

Definition. According to Codd's original definition, a relation schema R is in **3NF** if it satisfies 2NF *and* no nonprime attribute of R is transitively dependent on the primary key.

The relation schema EMP_DEPT in Figure 15.3(a) is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of $Dmgr_ssn$ (and also $Dname$) on Ssn via $Dnumber$. We can normalize

EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure 15.11(b). Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

Intuitively, we can see that any functional dependency in which the left-hand side is part (a proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute, is a *problematic* FD. 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF. Table 15.1 informally summarizes the three normal forms based on primary keys, the tests used in each case, and the corresponding *remedy* or normalization performed to achieve the normal form.

15.4 General Definitions of Second and Third Normal Forms

In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies discussed in Section 15.1.2. The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on the *primary key*. The normalization procedure described so far is useful for analysis in practical situations for a given database where primary keys have already been defined. These definitions, however, do not take other candidate keys of a relation, if

Table 15.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

any, into account. In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account. Notice that this does not affect the definition of 1NF since it is independent of keys and functional dependencies. As a general definition of **prime attribute**, an attribute that is part of *any candidate key* will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation.

15.4.1 General Definition of Second Normal Form

Definition. A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is not partially dependent on *any* key of R .¹¹

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure 15.12(a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id\# and $\{\text{County_name}, \text{Lot\#}\}$; that is, lot numbers are unique only within each county, but Property_id\# numbers are unique across counties for the entire state.

Based on the two candidate keys Property_id\# and $\{\text{County_name}, \text{Lot\#}\}$, the functional dependencies FD1 and FD2 in Figure 15.12(a) hold. We choose Property_id\# as the primary key, so it is underlined in Figure 15.12(a), but no special consideration will be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: $\text{County_name} \rightarrow \text{Tax_rate}$

FD4: $\text{Area} \rightarrow \text{Price}$

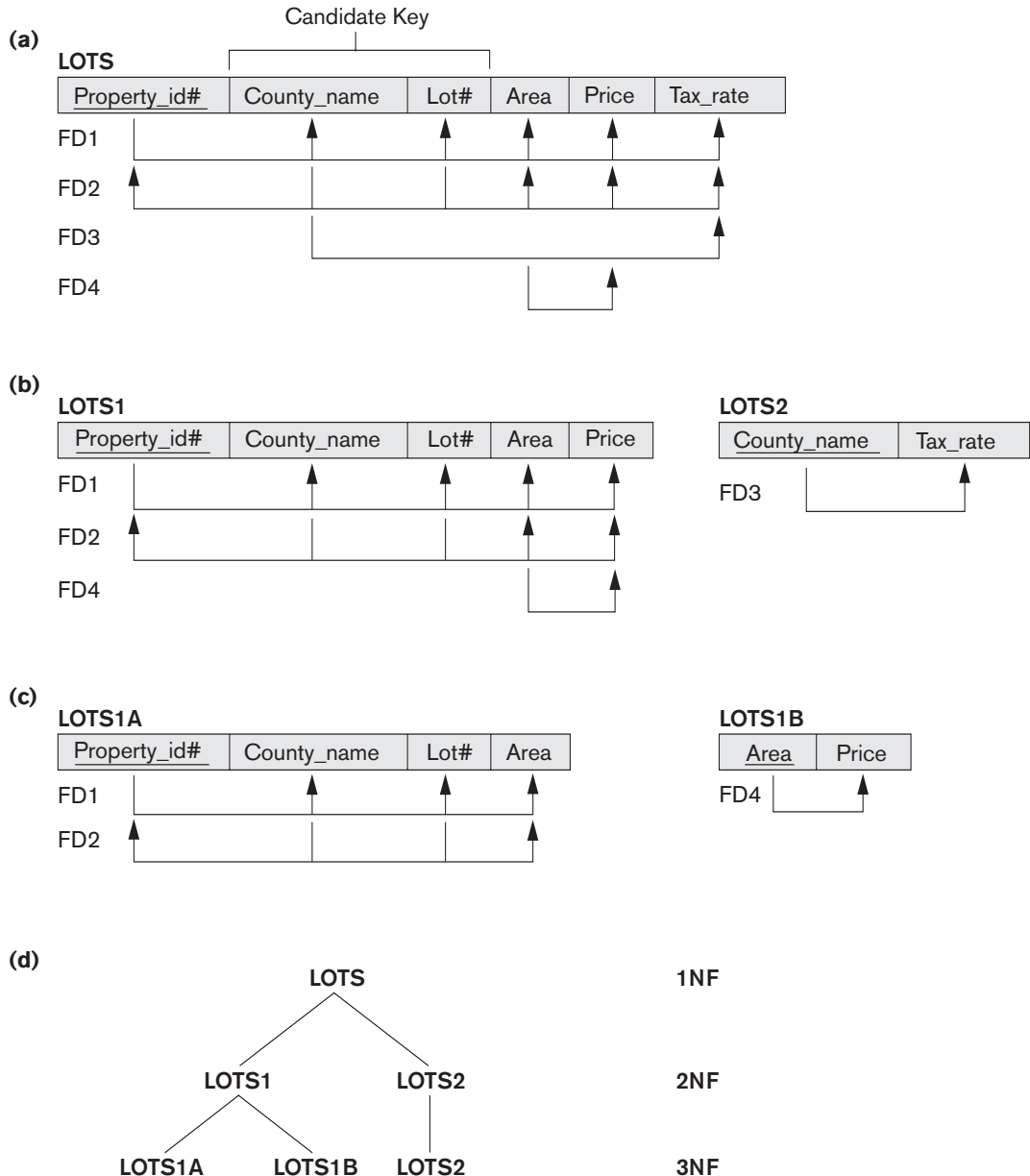
In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), while FD4 says that the price of a lot is determined by its area regardless of which county it is in. (Assume that this is the price of the lot for tax purposes.)

The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key $\{\text{County_name}, \text{Lot\#}\}$, due to FD3. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 15.12(b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

¹¹This definition can be restated as follows: A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on *every* key of R .

Figure 15.12

Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.



15.4.2 General Definition of Third Normal Form

Definition. A relation schema R is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R , either (a) X is a superkey of R , or (b) A is a prime attribute of R .

According to this definition, LOTS2 (Figure 15.12(b)) is in 3NF. However, FD4 in LOTS1 violates 3NF because *Area* is not a superkey and *Price* is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 15.12(c). We construct LOTS1A by removing the attribute *Price* that violates 3NF from LOTS1 and placing it with *Area* (the left-hand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

Two points are worth noting about this example and the general definition of 3NF:

- LOTS1 violates 3NF because *Price* is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute *Area*.
- This general definition can be applied *directly* to test whether a relation schema is in 3NF; it does *not* have to go through 2NF first. If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that *both* FD3 and FD4 violate 3NF. Therefore, we could decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence, the transitive and partial dependencies that violate 3NF can be removed *in any order*.

15.4.3 Interpreting the General Definition of Third Normal Form

A relation schema R violates the general definition of 3NF if a functional dependency $X \rightarrow A$ holds in R that does not meet either condition—meaning that it violates *both* conditions (a) and (b) of 3NF. This can occur due to two types of problematic functional dependencies:

- A nonprime attribute determines another nonprime attribute. Here we typically have a transitive dependency that violates 3NF.
- A proper subset of a key of R functionally determines a nonprime attribute. Here we have a partial dependency that violates 3NF (and also 2NF).

Therefore, we can state a **general alternative definition of 3NF** as follows:

Alternative Definition. A relation schema R is in 3NF if every nonprime attribute of R meets both of the following conditions:

- It is fully functionally dependent on every key of R .
- It is nontransitively dependent on every key of R .

15.5 Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema in Figure 15.12(a) with its four functional dependencies FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: DeKalb and Fulton. Suppose also that lot sizes in DeKalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, ..., 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: $\text{Area} \rightarrow \text{County_name}$. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because *County_name* is a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation $R(\text{Area}, \text{County_name})$, since there are only 16 possible Area values (see Figure 15.13). This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

Definition. A relation schema R is in **BCNF** if whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in R , then X is a superkey of R .

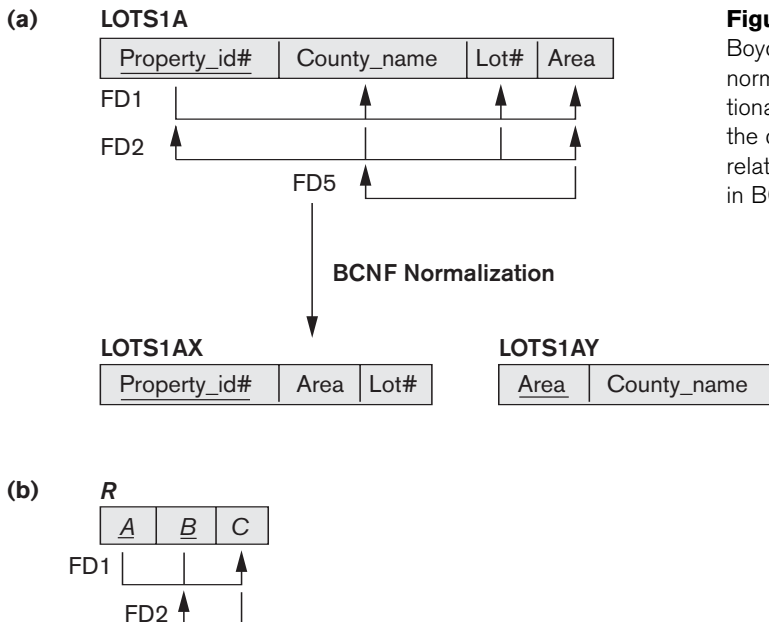


Figure 15.13

Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

The formal definition of BCNF differs from the definition of 3NF in that condition (b) of 3NF, which allows A to be prime, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF. In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. Note that FD5 satisfies 3NF in LOTS1A because County_name is a prime attribute (condition b), but this condition does not exist in the definition of BCNF. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 15.13(a). This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if $X \rightarrow A$ holds in a relation schema R with X not being a superkey *and* A being a prime attribute will R be in 3NF but not in BCNF. The relation schema R shown in Figure 15.13(b) illustrates the general case of such a relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema. Achieving the normalization status of just 1NF or 2NF is not considered adequate, since they were developed historically as stepping stones to 3NF and BCNF.

As another example, consider Figure 15.14, which shows a relation TEACH with the following dependencies:

FD1: {Student, Course} \rightarrow Instructor
 FD2:¹² Instructor \rightarrow Course

Note that {Student, Course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 15.13(b), with Student as A , Course as B , and Instructor as C . Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be

Figure 15.14

A relation TEACH that is in 3NF but not BCNF.

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

¹²This dependency means that *each instructor teaches one course* is a constraint for this application.

decomposed into one of the three following possible pairs:

1. {Student, Instructor} and {Student, Course}.
2. {Course, Instructor} and {Course, Student}.
3. {Instructor, Course} and {Instructor, Student}.

All three decompositions *lose* the functional dependency FD1. The *desirable decomposition* of those just shown is 3 because it will not generate spurious tuples after a join.

A test to determine whether a decomposition is nonadditive (or lossless) is discussed in Section 16.2.4 under Property NJB. In general, a relation not in BCNF should be decomposed so as to meet this property.

We make sure that we meet this property, because nonadditive decomposition is a must during normalization. We may have to possibly forgo the preservation of all functional dependencies in the decomposed relations, as is the case in this example. Algorithm 16.5 does that and could be used above to give decomposition 3 for TEACH, which yields two relations in BCNF as:

(Instructor, Course) and (Instructor, Student)

Note that if we designate (Student, Instructor) as a primary key of the relation TEACH, the FD $\text{instructor} \rightarrow \text{Course}$ causes a partial (non-full-functional) dependency of Course on a part of this key. This FD may be removed as a part of second normalization yielding exactly the same two relations in the result. This is an example of a case where we may reach the same ultimate BCNF design via alternate paths of normalization.

15.6 Multivalued Dependency and Fourth Normal Form

So far we have discussed the concept of functional dependency, which is by far the most important type of dependency in relational database design theory, and normal forms based on functional dependencies. However, in many cases relations have constraints that cannot be specified as functional dependencies. In this section, we discuss the concept of *multivalued dependency* (MVD) and define *fourth normal form*, which is based on this dependency. A more formal discussion of MVDs and their properties is deferred to Chapter 16. Multivalued dependencies are a consequence of first normal form (1NF) (see Section 15.3.4), which disallows an attribute in a tuple to have a *set of values*, and the accompanying process of converting an unnormalized relation into 1NF. If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

For example, consider the relation EMP shown in Figure 15.15(a). A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname. An employee may work on several projects and may have several dependents, and the employee's projects and dependents are independent of one another.¹³ To keep the relation state consistent, and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. This constraint is spec-

Figure 15.15

Fourth and fifth normal forms.

(a) The EMP relation with two MVDs: $Ename \twoheadrightarrow Pname$ and $Ename \twoheadrightarrow Dname$.

(b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.

(c) The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(R_1, R_2, R_3).

(d) Decomposing the relation SUPPLY into the 5NF relations R_1, R_2, R_3 .

(a) EMP

<u>Ename</u>	<u>Pname</u>	<u>Dname</u>
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(c) SUPPLY

<u>Sname</u>	<u>Part_name</u>	<u>Proj_name</u>
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(b) EMP_PROJECTS

<u>Ename</u>	<u>Pname</u>
Smith	X
Smith	Y

EMP_DEPENDENTS

<u>Ename</u>	<u>Dname</u>
Smith	John
Smith	Anna

(d) R_1

<u>Sname</u>	<u>Part_name</u>
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

 R_2

<u>Sname</u>	<u>Proj_name</u>
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

 R_3

<u>Part_name</u>	<u>Proj_name</u>
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

¹³In an ER diagram, each would be represented as a multivalued attribute or as a weak entity type (see Chapter 7).

ified as a multivalued dependency on the EMP relation, which we define in this section. Informally, whenever two *independent* 1:N relationships $A:B$ and $A:C$ are mixed in the same relation, $R(A, B, C)$, an MVD may arise.¹⁴

15.6.1 Formal Definition of Multivalued Dependency

Definition. A multivalued dependency $X \twoheadrightarrow Y$ specified on relation schema R , where X and Y are both subsets of R , specifies the following constraint on any relation state r of R : If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties,¹⁵ where we use Z to denote $(R - (X \cup Y))$:¹⁶

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

Whenever $X \twoheadrightarrow Y$ holds, we say that X **multidetermines** Y . Because of the symmetry in the definition, whenever $X \twoheadrightarrow Y$ holds in R , so does $X \twoheadrightarrow Z$. Hence, $X \twoheadrightarrow Y$ implies $X \twoheadrightarrow Z$, and therefore it is sometimes written as $X \twoheadrightarrow Y|Z$.

An MVD $X \twoheadrightarrow Y$ in R is called a **trivial MVD** if (a) Y is a subset of X , or (b) $X \cup Y = R$. For example, the relation EMP_PROJECTS in Figure 15.15(b) has the trivial MVD $\text{Ename} \twoheadrightarrow \text{Pname}$. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**. A trivial MVD will hold in *any* relation state r of R ; it is called trivial because it does not specify any significant or meaningful constraint on R .

If we have a *nontrivial MVD* in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure 15.15(a), the values ‘X’ and ‘Y’ of Pname are repeated with each value of Dname (or, by symmetry, the values ‘John’ and ‘Anna’ of Dname are repeated with each value of Pname). This redundancy is clearly undesirable. However, the EMP schema is in BCNF because *no* functional dependencies hold in EMP. Therefore, we need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP. Notice that relations containing nontrivial MVDs tend to be **all-key relations**—that is, their key is all their attributes taken together. Furthermore, it is rare that such all-key relations with a combinatorial occurrence of repeated values would be designed in practice. However, recognition of MVDs as a potential problematic dependency is essential in relational design.

We now present the definition of **fourth normal form (4NF)**, which is violated when a relation has undesirable multivalued dependencies, and hence can be used to identify and decompose such relations.

¹⁴This MVD is denoted as $A \twoheadrightarrow B|C$.

¹⁵The tuples t_1 , t_2 , t_3 , and t_4 are not necessarily distinct.

¹⁶ Z is shorthand for the attributes in R after the attributes in $(X \cup Y)$ are removed from R .

Definition. A relation schema R is in 4NF with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency $X \twoheadrightarrow Y$ in F^{+17} X is a superkey for R .

We can state the following points:

- An all-key relation is always in BCNF since it has no FDs.
- An all-key relation such as the EMP relation in Figure 15.15(a), which has no FDs but has the MVD $\text{Ename} \twoheadrightarrow \text{Pname} \mid \text{Dname}$, is not in 4NF.
- A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF.
- The decomposition removes the redundancy caused by the MVD.

The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD. Consider the EMP relation in Figure 15.15(a). EMP is not in 4NF because in the nontrivial MVDs $\text{Ename} \twoheadrightarrow \text{Pname}$ and $\text{Ename} \twoheadrightarrow \text{Dname}$, and Ename is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure 15.15(b). Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs $\text{Ename} \twoheadrightarrow \text{Pname}$ in EMP_PROJECTS and $\text{Ename} \twoheadrightarrow \text{Dname}$ in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

15.7 Join Dependencies and Fifth Normal Form

In our discussion so far, we have pointed out the problematic functional dependencies and showed how they were eliminated by a process of repeated binary decomposition to remove them during the process of normalization to achieve 1NF, 2NF, 3NF and BCNF. These binary decompositions must obey the NJB property from Section 16.2.4 that we referenced while discussing the decomposition to achieve BCNF. Achieving 4NF typically involves eliminating MVDs by repeated binary decompositions as well. However, in some cases there may be no nonadditive join decomposition of R into *two* relation schemas, but there may be a nonadditive join decomposition into *more than two* relation schemas. Moreover, there may be no functional dependency in R that violates any normal form up to BCNF, and there may be no nontrivial MVD present in R either that violates 4NF. We then resort to another dependency called the *join dependency* and, if it is present, carry out a *multiway decomposition* into fifth normal form (5NF). It is important to note that such a dependency is a very peculiar semantic constraint that is very difficult to detect in practice; therefore, normalization into 5NF is very rarely done in practice.

¹⁷ F^+ refers to the cover of functional dependencies F , or all dependencies that are implied by F . This is defined in Section 16.1.

Definition. A **join dependency (JD)**, denoted by $JD(R_1, R_2, \dots, R_n)$, specified on relation schema R , specifies a constraint on the states r of R . The constraint states that every legal state r of R should have a nonadditive join decomposition into R_1, R_2, \dots, R_n . Hence, for every such r we have

$$* (\pi_{R_1}(r), \pi_{R_2}(r), \dots, \pi_{R_n}(r)) = r$$

Notice that an MVD is a special case of a JD where $n = 2$. That is, a JD denoted as $JD(R_1, R_2)$ implies an MVD $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$ (or, by symmetry, $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$). A join dependency $JD(R_1, R_2, \dots, R_n)$, specified on relation schema R , is a **trivial** JD if one of the relation schemas R_i in $JD(R_1, R_2, \dots, R_n)$ is equal to R . Such a dependency is called trivial because it has the nonadditive join property for any relation state r of R and thus does not specify any constraint on R . We can now define fifth normal form, which is also called *project-join normal form*.

Definition. A relation schema R is in **fifth normal form (5NF)** (or **project-join normal form (PJNF)**) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency $JD(R_1, R_2, \dots, R_n)$ in F^+ (that is, implied by F),¹⁸ every R_i is a superkey of R .

For an example of a JD, consider once again the SUPPLY all-key relation in Figure 15.15(c). Suppose that the following additional constraint always holds: Whenever a supplier s supplies part p , and a project j uses part p , and the supplier s supplies at least one part to project j , then supplier s will also be supplying part p to project j . This constraint can be restated in other ways and specifies a join dependency $JD(R_1, R_2, R_3)$ among the three projections $R_1(\text{Sname}, \text{Part_name})$, $R_2(\text{Sname}, \text{Proj_name})$, and $R_3(\text{Part_name}, \text{Proj_name})$ of SUPPLY. If this constraint holds, the tuples below the dashed line in Figure 15.15(c) must exist in any legal state of the SUPPLY relation that also contains the tuples above the dashed line. Figure 15.15(d) shows how the SUPPLY relation *with the join dependency* is decomposed into three relations R_1 , R_2 , and R_3 that are each in 5NF. Notice that applying a natural join to *any two* of these relations *produces spurious tuples*, but applying a natural join to *all three together* does not. The reader should verify this on the sample relation in Figure 15.15(c) and its projections in Figure 15.15(d). This is because only the JD exists, but no MVDs are specified. Notice, too, that the $JD(R_1, R_2, R_3)$ is specified on *all* legal relation states, not just on the one shown in Figure 15.15(c).

Discovering JDs in practical databases with hundreds of attributes is next to impossible. It can be done only with a great degree of intuition about the data on the part of the designer. Therefore, the current practice of database design pays scant attention to them.

15.8 Summary

In this chapter we discussed several pitfalls in relational database design using intuitive arguments. We identified informally some of the measures for indicating

¹⁸Again, F^+ refers to the cover of functional dependencies F , or all dependencies that are implied by F . This is defined in Section 16.1.

whether a relation schema is *good* or *bad*, and provided informal guidelines for a good design. These guidelines are based on doing a careful conceptual design in the ER and EER model, following the mapping procedure in Chapter 9 correctly to map entities and relationships into relations. Proper enforcement of these guidelines and lack of redundancy will avoid the insertion/deletion/update anomalies, and generation of spurious data. We recommended limiting NULL values, which cause problems during SELECT, JOIN, and aggregation operations. Then we presented some formal concepts that allow us to do relational design in a top-down fashion by analyzing relations individually. We defined this process of design by analysis and decomposition by introducing the process of normalization.

We defined the concept of functional dependency, which is the basic tool for analyzing relational schemas, and discussed some of its properties. Functional dependencies specify semantic constraints among the attributes of a relation schema. Next we described the normalization process for achieving good designs by testing relations for undesirable types of *problematic* functional dependencies. We provided a treatment of successive normalization based on a predefined primary key in each relation, and then relaxed this requirement and provided more general definitions of second normal form (2NF) and third normal form (3NF) that take all candidate keys of a relation into account. We presented examples to illustrate how by using the general definition of 3NF a given relation may be analyzed and decomposed to eventually yield a set of relations in 3NF.

We presented Boyce-Codd normal form (BCNF) and discussed how it is a stronger form of 3NF. We also illustrated how the decomposition of a non-BCNF relation must be done by considering the nonadditive decomposition requirement. Then we introduced the fourth normal form based on multivalued dependencies that typically arise due to mixing independent multivalued attributes into a single relation. Finally, we introduced the fifth normal form, which is based on join dependency, and which identifies a peculiar constraint that causes a relation to be decomposed into several components so that they always yield the original relation back after a join. In practice, most commercial designs have followed the normal forms up to BCNF. Need for decomposing into 5NF rarely arises in practice, and join dependencies are difficult to detect for most practical situations, making 5NF more of theoretical value.

Chapter 16 presents synthesis as well as decomposition algorithms for relational database design based on functional dependencies. Related to decomposition, we discuss the concepts of *nonadditive* (or *lossless*) *join* and *dependency preservation*, which are enforced by some of these algorithms. Other topics in Chapter 16 include a more detailed treatment of functional and multivalued dependencies, and other types of dependencies.

Review Questions

- 15.1. Discuss attribute semantics as an informal measure of goodness for a relation schema.

- 15.2. Discuss insertion, deletion, and modification anomalies. Why are they considered bad? Illustrate with examples.
- 15.3. Why should NULLs in a relation be avoided as much as possible? Discuss the problem of spurious tuples and how we may prevent it.
- 15.4. State the informal guidelines for relation schema design that we discussed. Illustrate how violation of these guidelines may be harmful.
- 15.5. What is a functional dependency? What are the possible sources of the information that defines the functional dependencies that hold among the attributes of a relation schema?
- 15.6. Why can we not infer a functional dependency automatically from a particular relation state?
- 15.7. What does the term *unnormalized relation* refer to? How did the normal forms develop historically from first normal form up to Boyce-Codd normal form?
- 15.8. Define first, second, and third normal forms when only primary keys are considered. How do the general definitions of 2NF and 3NF, which consider all keys of a relation, differ from those that consider only primary keys?
- 15.9. What undesirable dependencies are avoided when a relation is in 2NF?
- 15.10. What undesirable dependencies are avoided when a relation is in 3NF?
- 15.11. In what way do the generalized definitions of 2NF and 3NF extend the definitions beyond primary keys?
- 15.12. Define Boyce-Codd normal form. How does it differ from 3NF? Why is it considered a stronger form of 3NF?
- 15.13. What is multivalued dependency? When does it arise?
- 15.14. Does a relation with two or more columns always have an MVD? Show with an example.
- 15.15. Define fourth normal form. When is it violated? When is it typically applicable?
- 15.16. Define join dependency and fifth normal form.
- 15.17. Why is 5NF also called project-join normal form (PJNF)?
- 15.18. Why do practical database designs typically aim for BCNF and not aim for higher normal forms?

Exercises

- 15.19. Suppose that we have the following requirements for a university database that is used to keep track of students' transcripts:
 - a. The university keeps track of each student's name (Sname), student number (Snum), Social Security number (Ssn), current address (Sc_addr) and

phone (Sc_phone), permanent address (Sp_addr) and phone (Sp_phone), birth date (Bdate), sex (Sex), class (Class) ('freshman', 'sophomore', ... , 'graduate'), major department (Major_code), minor department (Minor_code) (if any), and degree program (Prog) ('b.a.', 'b.s.', ... , 'ph.d.'). Both Ssn and student number have unique values for each student.

- b. Each department is described by a name (Dname), department code (Dcode), office number (Doffice), office phone (Dphone), and college (Dcollege). Both name and code have unique values for each department.
- c. Each course has a course name (Cname), description (Cdesc), course number (Cnum), number of semester hours (Credit), level (Level), and offering department (Cdept). The course number is unique for each course.
- d. Each section has an instructor (Iname), semester (Semester), year (Year), course (Sec_course), and section number (Sec_num). The section number distinguishes different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ..., up to the total number of sections taught during each semester.
- e. A grade record refers to a student (Ssn), a particular section, and a grade (Grade).

Design a relational database schema for this database application. First show all the functional dependencies that should hold among the attributes. Then design relation schemas for the database that are each in 3NF or BCNF. Specify the key attributes of each relation. Note any unspecified requirements, and make appropriate assumptions to render the specification complete.

- 15.20. What update anomalies occur in the EMP_PROJ and EMP_DEPT relations of Figures 15.3 and 15.4?
- 15.21. In what normal form is the LOTS relation schema in Figure 15.12(a) with respect to the restrictive interpretations of normal form that take *only the primary key* into account? Would it be in the same normal form if the general definitions of normal form were used?
- 15.22. Prove that any relation schema with two attributes is in BCNF.
- 15.23. Why do spurious tuples occur in the result of joining the EMP_PROJ1 and EMP_LOCS relations in Figure 15.5 (result shown in Figure 15.6)?
- 15.24. Consider the universal relation $R = \{A, B, C, D, E, F, G, H, I, J\}$ and the set of functional dependencies $F = \{ \{A, B\} \rightarrow \{C\}, \{A\} \rightarrow \{D, E\}, \{B\} \rightarrow \{F\}, \{F\} \rightarrow \{G, H\}, \{D\} \rightarrow \{I, J\} \}$. What is the key for R ? Decompose R into 2NF and then 3NF relations.
- 15.25. Repeat Exercise 15.24 for the following different set of functional dependencies $G = \{ \{A, B\} \rightarrow \{C\}, \{B, D\} \rightarrow \{E, F\}, \{A, D\} \rightarrow \{G, H\}, \{A\} \rightarrow \{I\}, \{H\} \rightarrow \{J\} \}$.

15.26. Consider the following relation:

A	B	C	TUPLE#
10	b1	c1	1
10	b2	c2	2
11	b4	c1	3
12	b3	c4	4
13	b1	c1	5
14	b3	c4	6

- Given the previous extension (state), which of the following dependencies *may hold* in the above relation? If the dependency cannot hold, explain why *by specifying the tuples that cause the violation*.
 i. $A \rightarrow B$, ii. $B \rightarrow C$, iii. $C \rightarrow B$, iv. $B \rightarrow A$, v. $C \rightarrow A$
- Does the above relation have a potential candidate key? If it does, what is it? If it does not, why not?

15.27. Consider a relation $R(A, B, C, D, E)$ with the following dependencies:

$$AB \rightarrow C, CD \rightarrow E, DE \rightarrow B$$

Is AB a candidate key of this relation? If not, is ABD ? Explain your answer.

15.28. Consider the relation R , which has attributes that hold schedules of courses and sections at a university; $R = \{\text{Course_no}, \text{Sec_no}, \text{Offering_dept}, \text{Credit_hours}, \text{Course_level}, \text{Instructor_ssn}, \text{Semester}, \text{Year}, \text{Days_hours}, \text{Room_no}, \text{No_of_students}\}$. Suppose that the following functional dependencies hold on R :

$$\begin{aligned} \{\text{Course_no}\} &\rightarrow \{\text{Offering_dept}, \text{Credit_hours}, \text{Course_level}\} \\ \{\text{Course_no}, \text{Sec_no}, \text{Semester}, \text{Year}\} &\rightarrow \{\text{Days_hours}, \text{Room_no}, \\ &\quad \text{No_of_students}, \text{Instructor_ssn}\} \\ \{\text{Room_no}, \text{Days_hours}, \text{Semester}, \text{Year}\} &\rightarrow \{\text{Instructor_ssn}, \text{Course_no}, \\ &\quad \text{Sec_no}\} \end{aligned}$$

Try to determine which sets of attributes form keys of R . How would you normalize this relation?

15.29. Consider the following relations for an order-processing application database at ABC, Inc.

$$\begin{aligned} \text{ORDER}(\text{O\#}, \text{Odate}, \text{Cust\#}, \text{Total_amount}) \\ \text{ORDER_ITEM}(\text{O\#}, \text{I\#}, \text{Qty_ordered}, \text{Total_price}, \text{Discount\%}) \end{aligned}$$

Assume that each item has a different discount. The *Total_price* refers to one item, *Odate* is the date on which the order was placed, and the *Total_amount* is the amount of the order. If we apply a natural join on the relations *ORDER_ITEM* and *ORDER* in this database, what does the resulting relation schema look like? What will be its key? Show the FDs in this resulting relation. Is it in 2NF? Is it in 3NF? Why or why not? (State assumptions, if you make any.)

15.30. Consider the following relation:

CAR_SALE(Car#, Date_sold, Salesperson#, Commission%, Discount_amt)

Assume that a car may be sold by multiple salespeople, and hence {Car#, Salesperson#} is the primary key. Additional dependencies are

Date_sold \rightarrow Discount_amt and

Salesperson# \rightarrow Commission%

Based on the given primary key, is this relation in 1NF, 2NF, or 3NF? Why or why not? How would you successively normalize it completely?

15.31. Consider the following relation for published books:

BOOK (Book_title, Author_name, Book_type, List_price, Author_affil, Publisher)

Author_affil refers to the affiliation of author. Suppose the following dependencies exist:

Book_title \rightarrow Publisher, Book_type

Book_type \rightarrow List_price

Author_name \rightarrow Author_affil

- a. What normal form is the relation in? Explain your answer.
- b. Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

15.32. This exercise asks you to convert business statements into dependencies. Consider the relation DISK_DRIVE (Serial_number, Manufacturer, Model, Batch, Capacity, Retailer). Each tuple in the relation DISK_DRIVE contains information about a disk drive with a unique Serial_number, made by a manufacturer, with a particular model number, released in a certain batch, which has a certain storage capacity and is sold by a certain retailer. For example, the tuple Disk_drive ('1978619', 'WesternDigital', 'A2235X', '765234', 500, 'CompUSA') specifies that WesternDigital made a disk drive with serial number 1978619 and model number A2235X, released in batch 765234; it is 500GB and sold by CompUSA.

Write each of the following dependencies as an FD:

- a. The manufacturer and serial number uniquely identifies the drive.
- b. A model number is registered by a manufacturer and therefore can't be used by another manufacturer.
- c. All disk drives in a particular batch are the same model.
- d. All disk drives of a certain model of a particular manufacturer have exactly the same capacity.

15.33. Consider the following relation:

R (Doctor#, Patient#, Date, Diagnosis, Treat_code, Charge)

In the above relation, a tuple describes a visit of a patient to a doctor along with a treatment code and daily charge. Assume that diagnosis is determined (uniquely) for each patient by a doctor. Assume that each treatment code has a fixed charge (regardless of patient). Is this relation in 2NF? Justify your answer and decompose if necessary. Then argue whether further normalization to 3NF is necessary, and if so, perform it.

15.34. Consider the following relation:

CAR_SALE (Car_id, Option_type, Option_listprice, Sale_date,
Option_discountedprice)

This relation refers to options installed in cars (e.g., cruise control) that were sold at a dealership, and the list and discounted prices of the options.

If $\text{CarID} \rightarrow \text{Sale_date}$ and $\text{Option_type} \rightarrow \text{Option_listprice}$ and $\text{CarID}, \text{Option_type} \rightarrow \text{Option_discountedprice}$, argue using the generalized definition of the 3NF that this relation is not in 3NF. Then argue from your knowledge of 2NF, why it is not even in 2NF.

15.35. Consider the relation:

BOOK (Book_Name, Author, Edition, Year)

with the data:

Book_Name	Author	Edition	Copyright_Year
DB_fundamentals	Navathe	4	2004
DB_fundamentals	Elmasri	4	2004
DB_fundamentals	Elmasri	5	2007
DB_fundamentals	Navathe	5	2007

- Based on a common-sense understanding of the above data, what are the possible candidate keys of this relation?
- Justify that this relation has the MVD $\{ \text{Book} \} \twoheadrightarrow \{ \text{Author} \} \mid \{ \text{Edition}, \text{Year} \}$.
- What would be the decomposition of this relation based on the above MVD? Evaluate each resulting relation for the highest normal form it possesses.

15.36. Consider the following relation:

TRIP (Trip_id, Start_date, Cities_visited, Cards_used)

This relation refers to business trips made by company salespeople. Suppose the TRIP has a single Start_date, but involves many Cities and salespeople may use multiple credit cards on the trip. Make up a mock-up population of the table.

- Discuss what FDs and/or MVDs exist in this relation.
- Show how you will go about normalizing it.

Laboratory Exercise

Note: The following exercise use the DBD (Data Base Designer) system that is described in the laboratory manual. The relational schema R and set of functional dependencies F need to be coded as lists. As an example, R and F for this problem is coded as:

$$R = [a, b, c, d, e, f, g, h, i, j]$$

$$F = [[a, b], [c]],$$

$$[[a], [d, e]],$$

$$[[b], [f]],$$

$$[[f], [g, h]],$$

$$[[d], [i, j]]$$

Since DBD is implemented in Prolog, use of uppercase terms is reserved for variables in the language and therefore lowercase constants are used to code the attributes. For further details on using the DBD system, please refer to the laboratory manual.

15.37. Using the DBD system, verify your answers to the following exercises:

- a. 15.24 (3NF only)
- b. 15.25
- c. 15.27
- d. 15.28

Selected Bibliography

Functional dependencies were originally introduced by Codd (1970). The original definitions of first, second, and third normal form were also defined in Codd (1972a), where a discussion on update anomalies can be found. Boyce-Codd normal form was defined in Codd (1974). The alternative definition of third normal form is given in Ullman (1988), as is the definition of BCNF that we give here. Ullman (1988), Maier (1983), and Atzeni and De Antonellis (1993) contain many of the theorems and proofs concerning functional dependencies.

Additional references to relational design theory are given in Chapter 16.